TABLAS HASH

TABLA HASH.- Dados un conjunto de datos identificados por una clave **(k)** se quiere obtener una función **h(k)** (función hash) que nos de la posición dentro de una tabla (tablas hash) en la que almacenamos un par formado por clave (k) y la dirección en un fichero donde se encuentra la información asociada a la clave (k).

TABLA HASH

	Clave (k)	dirección
0	239	1
1	500	n
.		
.		
.		
n	733	t

FICHERO

	Clave (k)	Contenido
0		
1	239	
.		
.		
.		
t	733	blablabla
.		
.		
.		
n	500	t

OBJETIVO: Realizar las búsquedas en O(1)!!

PROBLEMA de la Colisión : Dadas dos claves k_1 y k_2 la función hash aplicada a estas claves obtiene la misma posición en la Tabla hash, es decir , $h(k_1) = h(k_2)$. Objetivo elegir una función hash que produzca el menor número de colisiones.

	Clave (k)	dirección
0	239	1
1	500	n
.		
.		
.		
n	733	t

FICHERO

	Clave (k)	Contenido
0		
1	239	
.		
.		
.		
t	733	blablabla
.		
.		
.		
n	500	t

OBJETIVO: Realizar las búsquedas en O(1)!!

Ejemplos de funciones hash.

- **Simples**
  - Truncamiento : consiste en quedarse con unos dígitos determinados de la clave.
    Por ejemplo: h(123456789) = 123
  - Plegado: consiste en dividir la clave en, al menos, dos partes y sumar dichas partes.
    Por ejemplo: h(123456) = 123+456 = 579

- **Multiplicativo**: Trabaja multiplicando la clave *k* por sí misma o por una constante, usando después alguna porción de los bits del producto como una localización de la tabla hash.
  - Cuadrado Medio.- multiplicar k por sí misma y quedarse con algunos de los bits centrales Por ejemplo h(123456789) =456^2=207936
  - $h(k) = Int\big(M\ frac(C * k)\big)$ siendo M el tamaño de la tabla y C un valor entre 0 y 1.

- **Por división.-** $h(k) = k\ mod\ M$

- **Cadenas de Caracteres**.-$s = (s_0, s_1, \dots, s_n)$

$$h(s) = s_0 a^0 + s_1 a^1 + \cdots s_n a^n$$

Ejemplo: a=24

$$h(\text{Hola}) = 72 * 24^0 + 111 * 24^1 + 108 * 24^2 + 97 * 24^3 = 1.405.872$$

TABLAS HASH

Carácterísticas de las funciones hash.-

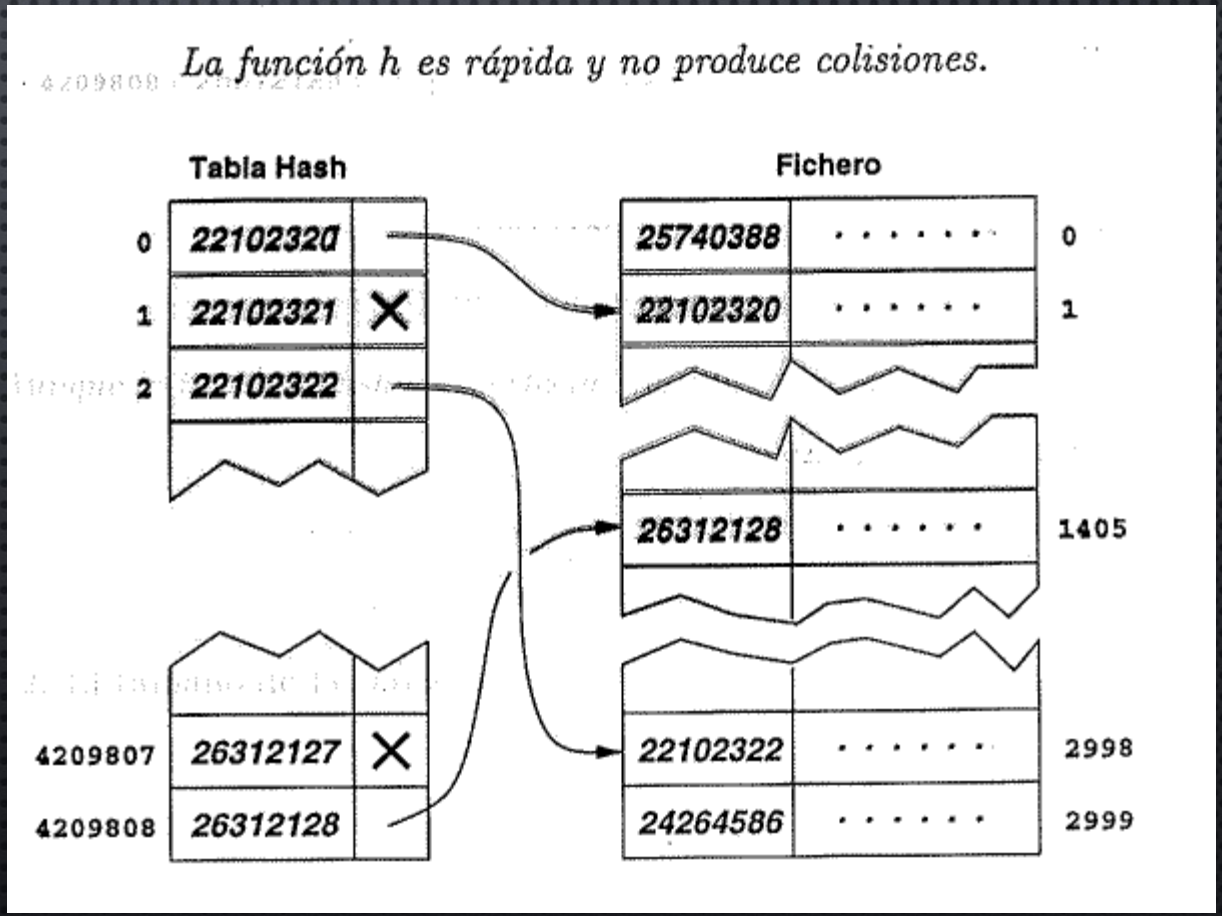
1. Deben ser fáciles de calcular (no consuman mucho tiempo de ejecución)
2. Que no produzcan colisiones

Ejemplo:

- Fichero de alumnos con 3000 alumnos
- Clave es el DNI
- Función hash

$$h(k) = k - 22102320$$

- Dimensión de la tabla hash: 4209808



Ejemplo: M=11 h(k)=k%M

h(k)	k	Dirección
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

Fichero de Dato

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

Ejemplo: M=11 h(k)=k%M

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

Fichero de Dato

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

Buscar la información de la clave 52

Ejemplo: M=11 h(k)=k%M

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

Fichero de Dato

	k	Nombre
0	12	Abad Ruiz
1	21	Bernabe Perez
2	68	Carrasco Ruiz
3	38	Domingo Coca
4	52	Fdez Sánchez
5	70	Juan Ruiz
6	44	Martín Pérez
7	18	Perez Galiano
8	**33**	**Lopez Frias**

Añadir una nueva entrada a la tabla

Como resolver las colisiones:
1. Hashing abierto
2. Hashing cerrado

Hashing abierto. Las colisiones se resuelven asignando a cada entrada de la tabla una lista donde se almacenan las claves con la misma función hash

M=5 Función hash h(k)=k mod 5

Fichero de Dato

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

tabla hash

h(k)	k
0	
1	
2	
3	
4	

Hashing abierto.

Se debe procurar que las listas no sean muy grandes, cuando lo sean debemos redimensionar la tabla.
Y esto nos llevará a colocar de nuevo todas la claves ya almacenadas en la nueva tabla.

Para establecer cuando redimensionar, se debe atender al **Factor de Carga**.

Factor de Carga: Razón entre el número de elementos total y el número de listas. En general a las listas se las denomina cubetas y dichas cubetas pueden ser cualquier otra estructura contenedora, p.ej. un AVL.

Hashing abierto. Implementación

```
#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 {
private:
 vector<list<Celda> > tabla;

  int fhash (int clave) {
    return clave % tabla.size();
  }

pair<bool,list<Celda>::iterator> Esta (int clave) {
    list<Celda>::iterator it;
    int pos = fhash (clave);
   for (it = tabla[pos].begin(); it != tabla[pos].end()
         && (*it).clave() != clave; ++it);

    bool find = false;
    if (it != tabla[pos].end()) find = true;
   return pair<bool, list<Celda>::iterator> (find, it);
}
public:
   ....
```

Hashing abierto. Implementación

```
#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 {
 private:
    vector<list<Celda> > tabla;
...
```

```
public:
 TH (int tam) {
              assert (tam > 0);
              tabla.resize (tam);
 }
bool Existe (int clave) {
   pair<bool,list<Celda>::iterator> a;
   a = Esta (clave);
   return a.first;
}
bool Insertar (int clave, int d) {
  pair<bool,list<Celda>::iterator> a;
  a = Esta (clave);
  if (!a.first) {
   int pos = fhash (clave);
   tabla[pos].push_back (Celda(clave,d));
   return true;
}
else   return false;}
```

Hashing abierto. Implementación

```
#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 {
 private:
   vector<list<Celda> > tabla;
```

```
public:
bool CambiarDir (int clave, int nuevadir) {
 pair<bool,list<Celda>::iterator> a;
  a = Esta(clave);
  if (a.first) {
   (*(a.second)).posicion() = nuevadir;
   return true;
 }
  else
    return false;
}
int ObtenDir (int clave) {
  pair<bool,list<Celda>::iterator> a = Esta (clave);
  if (a.first)
    return (*(a.second)).posicion ();
 else
   return -1;
 }
```

Hashing abierto. Implementación

```
#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 {
 private:
   vector<list<Celda> > tabla;
public:
```

```
bool Borrar (int clave) {
  pair<bool,list<Celda>::iterator> a = Esta (clave);
  if (a.first) {
    int pos = fhash (clave);
    tabla[pos].erase (a.second);
    return true;
  }
  else return false; }
friend ostream & operator<< (ostream & os, const TH & T) {
   vector<list<Celda> >::iterator it1; int pos = 0;
   for (it1 = T.tabla.begin(); it1 != T.tabla.end(); ++it1; ++pos) {
         os << "Datos en posicion " << pos << ':';
         list<Celda>::iterator it2;
         for (it2 = (*it1).begin(); it2 != (*it1).end(); ++it2)
            os << (*it2).clave() << ' ' << (*it2).posicion()
                                        << ' ';os << endl;
          }
        return os;
   }
};
```

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

- La clave
- La posición de dicha información dentro del fichero
- 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:

- Lineal
- Doble

Hashing cerrado. Rehashing Lineal

- M es el número de filas de la tabla hash. Debe ser mayor o igual que el número de registros y debe ser un numero primo.
- Función Hash: h(k)=k mod M.
- Funcion reshashing (la que se aplica cuando existe colisión):

$$h_i(k) = (h(k) + i - 1) mod\ M \ \ con\ i = 2,3, \ldots$$

Siendo $h_1(k) = h(k)$

Forma de Proceder para insertar una clave k:

Se calcula h(k)=pos. Si la tabla hash en la posición pos ya existe una clave insertada entonces se obtiene $h_2(k)$ dando lugar a otra posición en la tabla hash. Si en esa posición no existe ninguna clave ahí insertamos k. En el caso de que haya una clave insertada entonces obtenemos $\mathrm{h}_3(k)$. Este proceso se repite hasta encontrar una casilla libre.

Hashing cerrado. Rehashing Lineal

$M=11 \quad h(k) = k \, mod \, 11$

$h_i(k) = (h[redacted](k) + i - 1) mod \, 11 \quad con \, i = 2,3, \ldots$

Ejemplo

h(k)	k	Dirección	Estado
0	32	3	X
1	12	0	X
2	68	2	X
3	56	4	X
4	77	5	X
5	91	6	X
6			
7	18	7	X
8			
9			
10	21	1	X

Fichero de Dato

	k	Nombre
0	12	Abad Ruiz
1	21	Bernabe Perez
2	68	Carrasco Ruiz
3	32	Domingo Coca
4	56	Fdez Sánchez
5	77	Juan Ruiz
6	91	Martín Pérez
7	18	Perez Galiano

- h(12)=1
- h(21)=10
- h(68)=2
- h(32)=10→Colision
  - $h_2(32) = (10 + 1)\%11 = 0$
- h(56)=1→Colision
  - $h_2(56) = (1 + 1)\%11 = 2$
  - $h_3(56) = (1 + 2)\%11 = 3$
- h(77)=0→Colision
  - $h_2(77) = (0 + 1)\%11 = 1$
  - $h_3(77) = (0 + 2)\%11 = 2$
  - $h_4(77) = (0 + 3)\%11 = 3$
  - $h_5(77) = (0 + 4)\%11 = 4$
- h(91)=3→Colision
  - $h_2(91) = (3 + 1)\%11 = 4$
  - $h_3(91) = (3 + 2)\%11 = 5$
- h(18)=7

Rendimiento=17

Hashing cerrado. Rehashing Lineal

Problema

Produce agrupamientos primarios (sucesión de casillas ocupadas a distancia 1). Esta característica hace más lento los procesos de búsqueda e inserción.

TABLAS HASH Hashing cerrado. Rehashing Lineal

Ejemplo

$M=11 \quad h(k) = k \bmod 11$

$h_i(k) = (h(k) + i - 1) \bmod 11 \ \ con\ i = 2,3, \ldots$

h(k)	k	Dirección	Estado
0	32	3	X
1	12	0	X
2	68	2	X
3	56	4	X
4	77	5	X
5	91	6	X
6			
7	18	7	X
8			
9			
10	21	1	X

Fichero de Dato

	k	Nombre
0	12	Abad Ruiz
1	21	Bernabe Perez
2	68	Carrasco Ruiz
3	32	Domingo Coca
4	56	Fdez Sánchez
5	77	Juan Ruiz
6	91	Martín Pérez
7	18	Perez Galiano

- h(12)=1
- h(21)=10
- h(68)=2
- h(32)=10→Colision
  - $h_2(32) = (10 + 1)\%11 = 0$
- h(56)=1→Colision
  - $h_2(56) = (1 + 1)\%11 = 2$
  - $h_3(56) = (1 + 2)\%11 = 3$
- h(77)=0→Colision
  - $h_2(77) = (0 + 1)\%11 = 1$
  - $h_3(77) = (0 + 2)\%11 = 2$
  - $h_4(77) = (0 + 3)\%11 = 3$
  - $h_5(77) = (0 + 4)\%11 = 4$
- h(91)=3→Colision
  - $h_2(91) = (3 + 1)\%11 = 4$
  - $h_3(91) = (3 + 2)\%11 = 5$
- h(18)=7

Rendimiento=17

Rosa Mª Rodríguez Sánchez 1

Hashing cerrado. Rehashing Doble

- M es el número de filas de la tabla hash. Debe ser mayor o igual que el número de registros y debe ser un numero primo.
- Función Hash: h(k)=k mod M.
- Funcion reshashing (la que se aplica cuando existe colisión):

$$h_i(k) = (h_{i-1}(k) + h_0(k)) mod\ M \ \ con\ i = 2,3, \ldots$$

Siendo $h_1(k) = h(k)$ y $h_0(k) = 1 + k\ mod\ (M-2)$ con M y M-2 siendo primos relativos.

Forma de Proceder para insertar una clave k:

Se calcula h(k)=pos. Si la tabla hash en la posición pos ya existe una clave insertada entonces se obtiene $h_2(k)$ dando lugar a otra posición en la tabla hash. Si en esa posición no existe ninguna clave ahí insertamos k. En el caso de que haya una clave insertada entonces obtenemos $\mathrm{h}_3(k)$. Este proceso se repite hasta encontrar una casilla libre.

Hashing cerrado. Rehashing Doble. Ejemplo

- M =13
- Función Hash: h(k)=k mod 13 Función Rehashing $h_i(k) = (h_{i-1}(k) + h_0(k)) mod\ M\ \ con\ i = 2,3, \ldots$
  Siendo $h_1(k) = h(k)$ y $h_0(k) = 1 + (k\ \ mod\ 11))$ con M y M-2 siendo primos relativos.

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

$h(k)$	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

Hashing cerrado. Rehashing Doble. Ejemplo

TABLAS HASH Hashing cerrado. Rehashing Doble. Ejemplo

- M =13
- Función Hash: h(k)=k mod 13 Función Rehashing $h_i(k) = (h_{i-1}(k) + h_0(k)) mod\ M \ \ con\ i = 2,3,\ldots$
  Siendo $h_1(k) = h(k)$ y $h_0(k) = 1 + (k \ \ mod\ 11))$ con M y M-2 siendo primos relativos.

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

$h(k)$	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) $h_3(101) = (3+0)\%13 = 3$

Hashing cerrado.

Estados de las casillas de la Tabla Hash

- Ocupada
- Vacía
- Borrada

- **Operación de Inserción**: Vacía y Borrada tiene el mismo significado, es decir se puede usar para insertar una nueva clave.
- **Operación de Búsqueda**: Vacía y Borrada no significa lo mismo. Si al buscar una clave accedemos a una casilla con estado vacía entonces podemos parar la búsqueda y decir que la clave no está. Pero si el estado de la casilla es Borrada debemos proseguir la búsqueda.

h(k)	k	Dirección	Estado
0	32	3	X
1	12	0	X
2	68	2	X
3	56	4	X
4	77	5	X
5	91	6	X
6			
7	18	7	X
8			
9			
10	21	1	X

Hashing cerrado.

Algoritmo de búsqueda de clave en el hashing cerrado

```
int busquedadeclave(T,k){
1.   //Buscar la clave $k$ en la tabla $T$
2.   Calcular(h(c)); int posición;
3.   if (Estado(h(c)) != borrada && clave(h(c)) == c ){
4.      posicion =registro(h(c))
5.   }
6.   else
7.    while (!AnalizadaTodsaLatabla  && !salir){
8.      n←h_i(c ); i=i+1;}
9.      if ( Estado(n(c)!=borrado && (n(c))==c || Estado(n(c))==vacia ){
10.       if (Estado(n(c))!=vacia) {
11.              if clave (n(c))==c){
12.                posicion = registro(n(c));
13.                salir=true;
14.             }
15.      }
16.       else{
17.         posicion = -1; salir=true;
18.       }
19.    }
20.   }
21.   return posicion;
22. }
```

Hashing abierto. STL

Contendores Asociativos no Ordenados.- Este tipo de contenedores sirven para representar las tablas hash. Se clasifican:

1. Dependiendo si admiten valores repetidos o no
2. Y si las claves tiene valores asociados.

Funciones más relevantes

- De capacidad: empty, size, max_size
- Iteradores: begin,end,cbegin, cend
- Consulta: find, count, equal_range
- Modificadores: insert, erase,clear, swap
- Cubetas:
  - bucket_count: devuelve el número de cubetas
  - max_bucket_count: devuelve el número máximo de cubetas
  - bucket_size: devuelve el tamaño de la cubeta
  - bucket: localiza la cubeta de un elemento

Hashing abierto. STL

Contendores Asociativos no Ordenados.-

Funciones más relevantes

- Aspectos de la función hash :
  - load_factor: devuelve el factor de carga
  - max_load_factor: máximo factor de carga
  - rehash: modifica el numero de cubetas
  - reserve: solicita un cambio de capacidad
- Observadores
  - hash_function: obtiene la función hash
  - key_eq: recibe dos elementos y devuelve true indicando que tiene la misma función hash

Hashing abierto. STL unordered_set, unordered_multiset. Biblioteca: unordered_set

Se usan cuando se quiere almacenar 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

```
//USO DE FIND
#include <iostream>
#include <string>
#include <unordered_set>
using namespace std;
int main () {
  unordered_set<string> mycon = { "red","green","blue" };
  string input;
  cout << "color? ";
  getline (cin,input);
  unordered_set<string>::const_iterator got = mycon.find (input);
  if ( got == myset.end() )
     cout << "not found in myset";
  else
     cout << *got << " is in myset"<<endl;
  return 0;
}
```

Hashing abierto. STL unordered_set, unordered_multiset. Biblioteca: unordered_set

```
#include <iostream>
#include <string>
#include <unordered_set>
using namespace std;
typedef std::unordered_set<std::string> stringset;
int main () {
  unordered_set<string> mycon = { "red","green","blue" };
  string input;
  cout << "color? ";
  getline (cin,input);
  unordered_set<string>::const_iterator got = mycon.find (input);
  if ( got == myset.end() )
     cout << "not found in myset";
  else
     cout << *got << " is in myset"<<endl;
  stringset::hasher fn = mycon.hash_function();
  cout<<"that: "<<fn("that");
  cout<<"than: "<<fn("than");
  return 0; }
```

Hashing abierto. STL unordered_map, unordered_multimap. Biblioteca: unordered_map

Se usan cuando se quiere almacenar pares formadas por clave e información asociado. Los accesos por clave se hacen muy rápidos. Si se admiten claves repetidas se usará un unordered_multimap, en caso de que no se admita claves repetidas se usará un unordered_map

Ejercicio. Implementar el TDA Diccionario usando la clase unordered_map

```
#include <unorderd_map>
#include <iostream>
using namespace std;

template <class T,class U>
class Diccionario{
 private:
   unordered_map<T,U> datos;
 public:
//Implementación de la clase iterator
class iterator{
  private:
    typename unordered_map<T,U>::iterator it;
  public:
    iterator(){}
    iterator & operator ++(){
       ++it;
       return *this;
     }
     pair<constT,U>& operator*()
          return *it;
     }
     bool operator==(const iterator &i)const{
           return i.it==it;
     }
     bool operator!=(const iterator &i)const{
           return i.it!=it;
     }
     friend class Diccionario;
};//end iterator
```

Hashing abierto. STL unordered_map, unordered_multimap. Biblioteca: unordered_map

Se usan cuando se quiere almacenar pares formadas por clave e información asociado. Los accesos por clave se hacen muy rápidos. Si se admiten claves repetidas se usará un unordered_multimap, en caso de que no se admita claves repetidas se usará un unordered_map

Ejercicio. Implementar el TDA Diccionario usando la clase unordered_map

```
#include <unorderd_map>
#include <iostream>
using namespace std;

template <class T,class U>
class Diccionario{
 private:
   unordered_map<T,U> datos;
 public:
   iterator begin(){
     iterator i;
     i.it  =datos.begin();
     return i;
   }
```

```
iterator end(){
  iterator i;
  i.it=datos.end();
  return i;
 }
 int size()const{
    return datos.size()
 }
 typedef Diccionario<T,U>::iterator iterdiccio;
```

Hashing abierto. STL unordered_map, unordered_multimap. Biblioteca: unordered_map

Se usan cuando se quiere almacenar pares formadas por clave e información asociado. Los accesos por clave se hacen muy rápidos. Si se admiten claves repetidas se usará un unordered_multimap, en caso de que no se admita claves repetidas se usará un unordered_map

Ejercicio. Implementar el TDA Diccionario usando la clase unordered_map

```
#include <unorderd_map>
#include <iostream>
using namespace std;

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

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

pair<bool,iterdiccio> Esta_Clave(const T &p){
  pair<bool,iterdiccio> res;
  if (datos.size()>0){
   typename unordered_map<T,U> ::iterator it;
   it = datos.find(p);
   if (it==datos.end())
    res = {false,end()};
   else {
    iterdiccio i;
    i.it =it;
    res = {true,i};
   }
   return res;
  }
  res ={false,end()}; return res;
}
```

Hashing abierto. STL unordered_map, unordered_multimap. Biblioteca: unordered_map

Se usan cuando se quiere almacenar pares formadas por clave e información asociado. Los accesos por clave se hacen muy rápidos. Si se admiten claves repetidas se usará un unordered_multimap, en caso de que no se admita claves repetidas se usará un unordered_map

Ejercicio. Implementar el TDA Diccionario usando la clase unordered_map

```
#include <unorderd_map>
#include <iostream>
using namespace std;

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

```
typedef Diccionario<T,U>::iterator iterdiccio;
void insert(const T& clave,const U &info){
  pair<bool,iterdiccio> res =Esta_Clave(clave);
  if (!res.first){
   pair<T,U> p(clave,info);
   datos.insert(p);
  }
}

void erase(const T& clave){
  pair<bool,iterdiccio> res =Esta_Clave(clave);
  if (res.first){
     datos.erase (res.second.it);
  }
}

}
```

Hashing abierto. STL unordered_map, unordered_multimap. Biblioteca: unordered_map

Ejercicio. Implementar una función que obtenga si una cadena contiene a otra.

Ejercicio 2 . Es correcto en un esquema de hashing cerrado el uso como función hash de la función h(k)= [k + random(M)] % M, M primo y con random(M) una función que devuelve un número entero aleatorio entre 0 y M-1

Ejercicio 3 . Es correcto en un esquema de hashing doble el uso como función hash secundaria de la función h_0 (x)=[(B-1)-(x % B)] % B con B primo

Ejercicio 4 Si A es una **tabla hash cerrada** con un 50% de elementos vacíos y un 40% de elementos borrados y B una tabla hash cerrada con un 50% de elementos vacíos y sin elementos borrados, A y B son igual de eficientes cara a la búsqueda de un elemento.

Ejercicio 5.-Tenemos un contenedor de pares de elementos, {clave, bintree<int>} definida como:

```
template <typename T>
class contenedor {
private:
  unordered_map<T, bintree<int> > datos;
……..
……..
}
```

Implementa un **iterador** que itere sobre los elementos que cumplan la propiedad de que la suma de los elementos del bintree<int> sea un número par. Debes implementar (aparte de las de la clase iterator) las funciones begin() y end().

Ejercicio 5.-Solucion

```
template <typename T>
class contenedor {
private:
  unordered_map<T, bintree<int> > datos;
```