```
173
174         const_iterator end()const {
175               const_iterator i;
176               i.it=datos.end();
177               return i;
178         }
179 };
```

□

5.2.3 Contendores asociativos no ordenados

Este tipo de contenedores sirven para representar las tablas hash. La clasificación de estos contenedores se hará: 1) dependiendo si admiten valores repetidos o no; 2) y si las claves tienen valores asociados

La elementos se encuentran en un orden particular, y la recuperación de los mismo se hacen por su valor de una forma muy rápida. En un conjunto no ordenado el valor de un elemento es a su vez su llave. Este valor una vez insertado en el conjunto no ordenado no puede modificarse. Solamente podemos insertar, consultar y eliminar. Internamente los elementos no esta ordenados, pero se organizan en cubetas dependiendo del valor hash asociado. Por lo tanto este tipo de contenedor es el mas eficiente para acceder a elementos individuales pero no es eficiente cuando se quiere consultar un rango de valores. Los datos se almacenan en cubetas. Todos aquellos datos que tengan la misma función hash se almacenan en la misma cubeta. Por lo tanto ocurre colisión cuando la función hash para dos claves diferentes devuelve la misma cubeta.

Las funciones mas relevantes de estos contendores son:

- Funciones de capacidad:
  - *empty*: comprueba si el contenedor esta vacío
  - *size*: devuelve el tamaño del contenedor
  - *max_size*: devuelve el máximo tamaño del contendor
- Iteradores
  - *begin*: devuelve un iterador al principio del contenedor
  - *end*: devuelve un iterador al final
  - *cbegin*: devuelve un iterador constante al principio del contenedor
  - *cend*: devuelve un iterador constante al final
- Consulta
  - *find*: obtiene un iterador el al elemento
  - *count*: nos da el numero de elementos con un valor determinado (0 o 1).
  - *equal_range*: consigue un rango de elemento con una llave especifica.
- Modificadores
  - *insert*:inserta elementos
  - *erase*: elimina elementos
  - *clear*: limpia el contenido
  - *swap*: intercambia el contenido
- 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

- Aspectos de la función hash
  - *load_factor*: devuelve el factor de carga.
  - *max_load_factor*: maximo factor de carga.
  - *rehash*: modifica el numero de cubetas
  - *reserve*: solicita un cambio de capacidad
- Observadores
  - *hash_function*: obtiene la funcion hash
  - *key_eq*: toma dos elementos y devuelve un booleano indicadno si lo elementos son equivalentes porque tienen la misma funcion hash

Factor de Carga:*Razón entre el numero de elementos del contendor y el número de cubetas (valor obtenido con la función bucket_count).*
Hay que tener en cuenta que el factor de carga afecta a la probabilidad de colisión en la tabla hash (probabilidad de que dos elementos estén localizados en la misma cubeta). Así el contenedor usa el valor *max_load_factor* como el umbral para forzar un incremento en el número de cubetas y de esta forma procediendo a aplicar un *rehashing*.

Unordered_set/Unordered_multiset

Son contenedores que almacenan claves no repetidas (unordered_set) o si permiten claves repetidas tenemos el contenedor unordered_multiset. Para poder usar estos contenedores debemos hacer el include de la biblioteca *unordered_set.*

Ejemplo 5.2.3

En este ejemplo se muestra como inicializar un unordered_set y especialmente veremos como localizar un elemento usando la función *find.*

```
1  #include <iostream>
2  #include <string>
3  #include <unordered_set>
4   int main ()
5   {
6   std::unordered_set<std::string> myset = { "red","green","blue" };
7   std::string input;
8   std::cout << "color: ";
9   getline (std::cin,input);
10 //find devuelve un const_iterator ya
11 // que no podemos modificar el elemento
12  std::unordered_set<std::string>::const_iterator got = myset.find (input);
13
14  if ( got == myset.end() )//no lo hemos encontrador
```

```
15  std::cout << "no esta el elemento "<<input;
16  else
17  std::cout << *got << "si esta el elemento"<<input;
18  std::cout << std::endl;
19  return 0;
20  }
```

□

Ejemplo 5.2.4
En este otro ejemplo se hace uso de la función *count*. También hacer especial interes en la variable *auto* en el for para recorrrer los elementos de un conjunto. Para poder compilar este código deberemos hacerlo usando *-std=c++11*.

```
1  #include <iostream>
2  #include <string>
3  #include <unordered_set>
4   int main ()
5   {
6    std::unordered_set<std::string> myset = { "hat", "umbrella", "suit" };
7
8    //for con una variable auto
9    for (auto& x: {"hat","sunglasses","suit","t-shirt"}) {
10         if (myset.count(x)>0)
11         std::cout << "myset tiene " << x << std::endl;
12         else
13         std::cout << "myset no tiene " << x << std::endl;
14
15   }
16   return 0;
17
18 }
```

□

Ejemplo 5.2.5
En este ejemplo se revisan las funciones: *insert, erase, clear, size, bucket_count, load_factor, rehash, hash_function.*

```
1  #include <iostream>
2  #include <string>
3  #include <array>
4  #include <unordered_set>
5   int main ()
```

```
6  {
7  std::unordered_set<std::string> myset = {"yellow","green","blue"};
8  std::array<std::string,2> myarray = {"black","white"};
9  std::string mystring = "red";
10
11 myset.insert (mystring);
12 //insertando reddish
13 myset.insert(mystring+"dish");
14 //insertando elementos del array
15 myset.insert(myarray.begin(), myarray.end());
16 //insertando desde un conjunto constante
17 myset.insert( {"purple","orange"} );
18
19 std::cout << "myset contiene:";
20 for (const std::string& x: myset) std::cout << " " << x;
21 std::cout << std::endl;
22
23 myset.clear();//vaciamos el conjunto
24 myset.rehash(12); //reserva un numero como minimo de 12 cubetas aunque
25                   //realmente reserva el siguiente primo 13.
26
27 myset.insert({"USA","Canada","France","UK","Japan","Germany","Italy"});
28 myset.erase ( myset.begin() );// eliminacion por iterador
29
30 myset.erase ( "France" );// eliminacion por valor
31
32 myset.erase ( myset.find("Japan"), myset.end() ); //eliminacion por rango
33
34 std::cout << "myset contiene:";
35 for ( const std::string& x: myset ) std::cout << " " << x;
36 std::cout << std::endl;
37
38
39 std::cout<<"size = " << myset.size() << std::endl;//no. elementos
40 std::cout<<"bucket_count = " << myset.bucket_count() << std::endl;//no. cubetas
41 std::cout<<"load_factor = " << myset.load_factor() << std::endl;//factor de carga
42 //maximo factor de carga
43 std::cout<< "max_load_factor = " << myset.max_load_factor() << std::endl;
44
45 //como se usa la hash_function
46 stringset::hasher fn = myset.hash_function();
47 std::cout << "purple tiene funcion hash: " << fn ("purple") << std::endl;
48 std::cout << "orange tiene funcion hash: " << fn ("orange") << std::endl;
```

```
49 return 0;
50
51  return 0;
52 }
```

□

Unordered_map/Unordered_multimap

Almacenan elementos formados por la combinacion valor clave y valor asociado. Estos contenedores son aconsejables cuando se necesita búsquedas rápidas por clave. Al igual que en los unordered_set los pares clave valor asociado, se almacenan internamente en la cubeta determinada por el valor hash asociado a la clave. Las funciones que hemos visto anteriormente para Unordered_set/Unordered_multiset son aplicables tambien para Unordered_map/Unordered_multimap.

Ejemplo 5.2.6

Un pequeño ejemplo para definir un unordered_map y ver su contenido y borrar elementos.

```
1  // unordered_map::erase
2  #include <iostream>
3  #include <string>
4  #include <unordered_map>
5  int main ()
6  {
7    std::unordered_map<std::string,std::string> mymap;
8
9   // Iniciando los valores
10   mymap["U.S."] = "Washington";
11   mymap["U.K."] = "London";
12   mymap["France"] = "Paris";
13   mymap["Russia"] = "Moscow";
14   mymap["China"] = "Beijing";
15   mymap["Germany"] = "Berlin";
16   mymap["Japan"] = "Tokyo";
17
18   mymap.erase ( mymap.begin() );// erasing by iterator
19
20   mymap.erase ("France");// erasing by key
21
22   mymap.erase ( mymap.find("China"), mymap.end() ); //borrando por rango
23
24   // contenido
25   std::unordered_map<std::string,std::string>::iterator it;
26   for (it= mymap.begin(); it!=mymap.end();++it) )
27     std::cout << it->first << ": " << it->second << std::endl;
```

```
28
29
30   // mostrar el contenido por cubetas
31    std::cout << "mymap lista por cubetas:\n";
32    for ( unsigned i = 0; i < mymap.bucket_count(); ++i) {
33
34     std::cout << "bucket #" << i << " contiene:";
35     for ( auto local_it = mymap.begin(i); local_it!= mymap.end(i); ++local_it )
36      std::cout << " " << local_it->first << ":" << local_it->second;
37     std::cout << std::endl;
38    }
39
40  return 0;
41
42  }
```

□

Ejemplo 5.2.7

El siguiente ejemplo muestra un diccionario usando para su representación un *unordered_map* de pares string,string. El primer string es para la palabra y el segundo para la definición. Las operaciones que se han definido son:

- *operador []*: para consultar una palabra
- *find*: para buscar una palabra.
- *insert*: para insertar una nueva palabra junto con su definición
- *erase*: para borrar un palabra.

```
1  #include <unordered_map>
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  class Diccionario{
6
7
8   private:
9   //string primero palabra, string segundo definicion
10    unordered_map<string,string> datos;
11
12  public:
13    /**
14    * @brief obtiene la definicion de una palabra
15    * @param pal: palabra a buscar
16    */
```

```
17     string & operator[]( const string &pal){
18        return datos[pal];
19      }
20
21    //iterator
22    class const_iterator ;//declaracion adelatanda
23    class iterator {
24     private:
25          unordered_map<string,string>::iterator it;
26    public:
27       iterator operator ++(){
28         ++it;
29         return *this;
30       }
31       pair<const string,string> & operator *(){
32           return *it;
33       }
34       bool operator==(const iterator &i){
35          return i.it==it;
36       }
37       bool operator!=(const iterator &i){
38          return i.it!=it;
39       }
40       friend class Diccionario;
41       friend class const_iterator;
42    };
43
44    class const_iterator {
45     private:
46          unordered_map<string,string>::const_iterator it;
47    public:
48       const_iterator (){}
49
50       const_iterator (const iterator &i):it(i.it){}
51
52       const_iterator operator ++(){
53         ++it;
54         return *this;
55       }
56       const pair<const string,string> & operator *(){
57           return *it;
58       }
59       bool operator==(const const_iterator &i){
```

```
60          return i.it==it;
61       }
62       bool operator!=(const const_iterator &i){
63          return i.it!=it;
64       }
65       friend class Diccionario;
66
67    };
68
69
70    /**
71     * @brief devuelve un iterador a la palabra y su definicion
72     * @param pal: palabra a buscar
73     * @note si no esta devuelve end
74     */
75
76
77    const_iterator find(const string &pal)const{
78       //usamos find del unordered_map
79       unordered_map<string,string>::const_iterator got = datos.find (pal);
80       //ahora definimos un iterador de Diccionario
81       const_iterator i;
82       i.it=got;
83       return i;
84    }
85
86    /**
87     * @brief devuelve el numero de elementos del diccionario
88     */
89    int size()const{ return datos.size();}
90
91    /**
92     @brief inserta una palabra con su definicion
93     @param pal: palabra
94     @param def: definicion de la palabra
95     */
96    pair<iterator,bool> insert( string pal, string  def){
97         pair<string,string> a(pal,def);
98         pair<unordered_map<string,string>::iterator,bool> it= datos.insert(a);
99         iterator i;
100        i.it=it.first;
101        return pair<iterator,bool>(i,it.second);
102   }
```

```
103
104     /**
105      @brief borra una palabra
106      @param pal: palabra a borrar
107     */
108     void erase(const string &pal){
109        datos.erase(pal);
110     }
111
112
113     /**
114      * @brief inicializa los iteradores al principio
115      */
116     iterator begin(){
117       iterator i;
118       i.it=datos.begin();
119       return i;
120     }
121     const_iterator begin()const{
122        const_iterator i;
123        i.it =datos.begin();
124        return i;
125     }
126
127     /**
128      * @brief inicializa los iteradores al final
129      */
130
131     iterator end(){
132       iterator i;
133       i.it=datos.end();
134       return i;
135     }
136     const_iterator end()const{
137        const_iterator i;
138        i.it =datos.end();
139        return i;
140     }
141 };
```

Un ejemplo de uso de Diccionario sería el siguiente

```
1  int main(){
2     Diccionario d;
```

```
3     //insertamos elementos
4     d.insert("gato","mamifero felino");
5     d.insert("perro","mamifero canino");
6     d.insert("gorrion","ave del mediterraneo");
7     d.insert("pinguino","ave polar");
8
9     //No sale ordenado por clave
10    //ya que el orden es propio
11    Diccionario::iterator i;
12    for (i=d.begin(); i!=d.end();++i)
13       cout<<(*i).first<<" "<<(*i).second<<endl;
14
15    //ejemplo de find
16    Diccionario::const_iterator i2;
17    i2= d.find("pinguino");
18
19    cout<<"Definicion de pinguino "<<(*i2).second<<endl;
20
21    cout<<"Definicio de perro: "<<d["perro"]<<endl;
22
23    if (d.find("perro")!=d.end()){
24       cout<<"La entrada perro esta"<<endl;;
25    }
26    else cout<<"La entrada perro no esta"<<endl;
27
28    //ejemplo de borrado
29    d.erase("perro");
30    if (d.find("perro")!=d.end()){
31       cout<<"La entrada perro esta"<<endl;;
32    }
33    else cout<<"La entrada perro no esta"<<endl;
34 }
```

□

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(); //datos.clear();
10   }
11   void clear(){
12     Borrar(); //datos.clear();
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<clave.size();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:

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

□