

PRACTICA 2: METEORITOS



1. INTRODUCCION



Diseño de dos tipos de datos:

- el primero será un tipo capaz de guardar la **información** asociada a un **meteorito**.
- mientras que el segundo implicará el diseño de un **contenedor básico, capaz de almacenar un gran volumen de meteoritos**

1.2 Conjunto de Datos de Meteoritos

Conjunto de datos contiene información sobre 45716 meteoritos que han caído en la tierra, actualizado a 14 de Mayo de 2013:

- **nombre:** por ejemplo "Andhara" o "Andover", de tipo string.
- **tipo:** secuencia de códigos, separados por comas, que describen el tipo de meteorito, por ejemplo "Iron, IIAB".
- **masa:** masa del meteorito en gramos.
- **caído/encontrado:** valor binario que indica si ha sido encontrado o sólo sabemos que ha caído (fell/found),
- **año:** año en que impactó
- **coordenada de latitud**
- **Coordenada de longitud**

1.2 Conjunto de Datos de Meteoritos

- Cuando alguno de los campos es desconocido, este está en blanco. La base de datos está en formato **csv** donde los distintos campos están separados por punto y coma, por ejemplo, las primeras líneas del fichero son:

name;recclass;mass;fall;year;reclat;reclong

Aachen;L5;21;Fell;1880;50,775;6,08333

Aarhus;H6;720;Fell;1951;56,18333;10,23333

Abee;EH4;107000;Fell;1952;54,21667;-113

Acapulco;Acapulcoite;1914;Fell;1976;16,88333;-99,9

1.2.1 Meteorito

- En nuestro caso, definiremos un **tipo meteorito** como un **par**, donde el campo *first* representa el nombre del meteorito (string) y el campo *second* será del tipo `defMet`, que define al resto de datos asociados que definen el meteorito.

```
typedef pair<nombreM,defM> meteorito;
```


- La especificación del tipo, así como su implementación se realizará en los ficheros `meteorito.h` y `meteorito.hxx`

1.3 Diccionario



Un **diccionario** es un contenedor que permite **almacenar un conjunto de pares de elementos**, el primero será la clave que debe de ser única y el segundo la definición.

- ▣ Clave: nombre del meteorito
- ▣ Definición: contendrá los valores de tipo defM(características del meteorito)

- 
- Como TDA diccionario, lo vamos a dotar de un conjunto **restringido de métodos** (**inserción** de elementos, **consulta** de un elemento por **clave**).
 - Este diccionario "simulará" un diccionario de la stl, no estará dotado de la capacidad de iterar (recorrer) a través de sus elementos, que se hará en las siguientes prácticas

1.4 “Se Entrega/ Se Pide”

1.4.1 Se Entrega

- [Documentacion.pdf](#): Documentación de la práctica
- [dox_diccionario](#): Fichero de configuración de doxygen necesario para generar la documentación del proyecto (html y pdf)

doxygen dox_diccionario

- Html: /html/index.html

- Latex:

cd latex

make

El resultado es el fichero [refman.pdf](#)

1.4 “Se Entrega/ Se Pide”

1.4.1 Se Entrega

- `diccionario.h`: Especificación del TDA diccionario
- `diccionario.hxx` fichero donde debemos implementar la primera versión del diccionario
- `meteorito.h`: Plantilla para la especificación del TDA meteorito
- `Principal.cpp`: fichero que incluye el main del programa. En este caso se toma como entrada el fichero de datos “`meteorites_all.csv`” y se debe cargar en el diccionario

1.4.2 "Se Pide"

- Consideramos dos posibles implementaciones basadas en el tipo de dato vector de la stl, analizando las eficiencia de las mismas:
 - ▣ Primera implementación: `diccionarioV1.hxx`
`g++ -D DICC_V1 -o diccionarioV1 principal.cpp`
 - ▣ Segunda implementación: `diccionarioV2.hxx`
`g++ -D DICC_V2 -o diccionarioV2 principal.cpp`

1.4.2 "Se Pide"

Por tanto se pide:

- ❑ diccionarioV1.hxx
- ❑ diccionarioV2.hxx
- ❑ AnalisisComparativo.pdf Dicho análisis valorará por un lado el tiempo dedicado a la **inserción de toda la colección de meteoritos en un diccionario** y por otro el tiempo necesario para realizar una búsqueda de todos los meteoritos en el mismo

1.6 Representaciones



- **Primera representación**, los elementos se almacenarán **sin** tener en cuenta el valor de la **clave**
- **Segunda representación**: los elementos se encontrarán **ordenados** por el valor de la **clave**

1.7 Primera Representación

$D = (\text{vector} \langle \text{entrada} \rangle \text{dic}) \Rightarrow \text{Diccionario Dic};$

- Un objeto abstracto, `Dic`, representando una colección de pares (`clave`, `def`)
- Dada una entrada, `x`, en `D`, `x.first` representa a una clave válida y `x.second` representa su definición

1.7.2 Invariante de la Representación

Propiedades que debe cumplir cualquier objeto

`Dic.size() == D.dic.size();`

Para todo i , $0 \leq i < D.dic.size()$
se cumple

`D.dic[i].first != "";`

`D.dic[i].first != D.dic[j].first,`
para todo $j \neq i$.

1.8 Segunda Representación

- La representación que se utiliza es un **vector ordenado de entradas**, teniendo en cuenta el valor de la clave
- Un objeto abstracto, Dic, representando una colección ORDENADA de pares (clave,def), se instancia en la clase diccionario como un vector de entradas, definidas como diccionario::entrada. Dada una entrada, x, en D, x.first representa a una clave válida y x.second representa su definición.

1.8.2 Invariante de la Representación

Propiedades que debe cumplir cualquier objeto

`Dic.size() == D.dic.size();`

Para todo i , $0 \leq i < D.dic.size()$ se cumple

`D.dic[i].first != "";`

Para todo i , $0 \leq i < D.dic.size()-1$ se cumple

`D.dic[i].first < D.dic[i+1].first`



TIPO VECTOR DE LA STL

Tipo Vector de la STL

- Permite almacenar cero o más elementos del mismo tipo, pudiendo acceder a ellos individualmente mediante un **índice**
- El **número** de elementos del vector puede variar **dinámicamente** y la gestión de memoria se hace de manera totalmente **transparente** al usuario
- Para usarlo hará que incluir
 - ▣ `#include <vector>`

```
vector<TPO> miVector;
```

Tipo Vector de la STL

```
vector<double> vectorReales;  
vector<string> vectorCadenas;
```

- Crear un vector de 10 enteros:

```
vector<int> vectorEnteros(10);
```

- Para inicializar a un valor concreto los elementos del vector:

```
vector<int> vectorEnteros(10, -1);
```

Algunas Operaciones

```
vector<int> miVector;
```

- Obtener el número de elementos del vector:

```
miVector.size();
```

- Almacenar un elemento al final del vector:

```
miVector.push_back(17);
```

- Acceso a un elemento de un vector:

```
cout << miVector[6];
```

```
miVector[6]=17; //acceso sin comprobación de  
                //rango
```

```
miVector.at(6)=17; //acceso con comprobación de  
                  //rango
```

Algunas Operaciones

- Para cualquier subíndice n , tiene que ser verdadero:

- $0 \leq n \leq \text{size}()$

- Ejemplo:

```
vector<int> miVector;
```

```
miVector[0]=25
```

```
//ERROR!
```

```
miVector.push_back(15);
```

```
miVector[0]=25;
```

```
//OK
```

Algunas Operaciones



- **size_type size():** devuelve el número de elementos almacenados en el vector
- **Bool empty():** true si el numero de elementos es cero y false en caso contrario

Algunas Operaciones

- **void push_back(x):**inserta un elemento x al final del vector. Aumenta el tamaño del vector en 1

- `Vector<int> a`

- `a.push_back(5);`

- `vector<int> vInt;`

- `for(int i=0; i<100;i++)`

- `vInt.push_back(i);`

Algunas Operaciones



- `void pop_back():` elimina el último elemento.Reduce el tamaño en 1.
- `void clear():` Borra todos los elementos de un vector.

Algunas Operaciones

- **Operador de asignación(=)**

```
vector<int> a;
```

```
vector<int> b;
```

```
a.push_back(5);
```

```
a.push_back(10);
```

```
b.push_back(3);
```

```
b=a;    //El vector b contendrá dos elementos 5 y 10 lo mismo  
        //que el vector a
```

- **El operador comparación (==):** comprueba si dos vectores contienen los mismos elementos. Lleva una comparación elemento a elemento.