UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica Final: Cifras y Letras

Práctica 4: Contenedores no lineales

(Práctica puntuable)

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Introducción

1.1 Cifras y letras

En esta práctica, y las prácticas sucesivas, nos centraremos en el juego conocido como cifras y letras. Este juego, que se ha popularizado a través de concursos de televisión en distintos países. Nos centraremos de momento en la prueba de las letras. Exploraremos el juego de las cifras en las siguientes prácticas, debido a su mayor complejidad

Prueba de las letras

Esta parte del juego consiste en formar la mejor palabra posible (dependiendo de uno de los dos criterios que explicamos a continuación) a partir de un conjunto de letras extraídas al azar de una bolsa. Por ejemplo, dadas las siguientes letras:

O D Y R M E T

una buena solución posible sería METRO. El número de letras que se juegan en cada partida se decide de antemano, y las letras disponibles pueden repetirse. Existen dos modalidades de juego:

- Juego a longitud: En este modo de juego, se tiene en cuenta sólo la longitud de las palabras, y gana la palabra más larga encontrada
- Juego a puntos: En este modo de juego, a cada letra se le asigna una puntuación, y la puntuación de la palabra será igual a la suma de las puntuaciones de las letras que la componen

En esta práctica y las prácticas siguientes construiremos las estructuras de datos adecuadas para resolver el problema de las letras, así como programas que nos permitan obtener la solución a una partida.

Estaremos especialmente interesados en dar una buena solución al problema de las letras. En esta primera práctica, estudiaremos qué información necesitamos almacenar, y cuál es la forma más adecuada de almacenarla, para poder jugar una partida al juego de las letras. En las prácticas siguientes, estudiaremos dos formas distintas de resolver el juego, que dependerá en parte de la estructura de datos subyacente.

1.2 Información necesaria para una partida de las letras: Archivos de entrada

Teniendo en cuenta la descripción del juego de las letras que hemos hecho en el apartado anterior, parece claro que vamos a necesitar tres almacenes principales de información para poder jugar una partida de las letras

Información sobre las letras

Como hemos dicho anteriormente, uno de los modos de juego a los que podemos jugar asigna una puntuación a cada letra, y la puntuación de la palabra será la suma de las puntuaciones de cada una de sus letras. Por tanto, necesitamos recoger la información de la puntuación para cada letra de algún sitio.

Además, hemos dicho que podríamos tener cada letra repetida un número de veces. No obstante, esto puede llevar a problemas en algunas partidas. Si todas las letras pudieran repetirse un número indeterminado de veces, podría ocurrir que en alguna partida sólo tuviésemos la letra Z muchas veces, lo que dificultaría en gran medida formar una palabra. Además, parece lógico pensar que si

las letras que más se repiten son letras que aparecen mucho en el diccionario, las palabras que se podrán formar serán más largas, haciendo el juego más interesante para los participantes. Por esto, la forma que tendremos de seleccionar las letras de cada partida será considerando un número de repeticiones de cada letra, formando una "bolsa" con todas ellas, y sacando al azar con probabilidad uniforme elementos de esa bolsa. Así, las letras con más repeticiones tendrán mayor probabilidad de salir, mientras que las que tengan pocas repeticiones saldrán menos a menudo (y no se repetirán en la misma partida si no hay más de una copia, ya que haremos extracciones sin reemplazamiento).

Por estos dos motivos, tendremos que guardar la siguiente información para cada letra del abecedario:

- Su puntuación
- El número de repeticiones disponible

Esta información la leeremos de un fichero como el que se muestra a continuación:

```
Letra  Cantidad Puntos
A      12       1
B      2        3
C      5        3
D      5        2
E      12       1
F      1        4
G      2        2
H      2        4
I      6        1
J      1        8
L      1        1
M      2        3
N      5        1
O      9        1
P      2        3
Q      1        5
R      6        1
S      6        1
T      4        1
U      5        1
V      1        4
X      1        8
Y      1        4
Z      1        10
```

Fichero 1: letras.txt

Crearemos una estructura de datos adecuada para almacenar esta información durante una partida, y poder aprovecharla para calcular las puntuaciones de nuestras palabras. Esta estructura de datos se denominará Conjunto de Letras (TDA LettersSet)

Además, como hemos dicho que necesitaremos extraer letras aleatorias para una partida, utilizando para ello las repeticiones del archivo anterior, construiremos también un contenedor específico para ello. Dicho contenedor será capaz de leer un Conjunto de Letras con las repeticiones de cada una

de las letras del juego, y crear la bolsa de la que haremos las extracciones aleatorias. Este TDA se denominará Bolsa de Letras (TDA LettersBag)

Información sobre las palabras

Finalmente, dado que vamos a jugar a un juego que trabaja sobre palabras, es importante establecer qué palabras estarán permitidas en una partida. Una “palabra” formada con las letras del ejemplo anterior puede ser METROYD, pero claramente no podríamos aceptar esta palabra en una partida, ya que no pertenece a nuestro idioma.

Por tanto, necesitaremos trabajar con un listado de palabras permitidas en nuestro juego, que funcionará a modo de diccionario. Sólamente aceptaremos como válidas aquellas palabras que pertenezcan a nuestro diccionario. La información de las palabras la recopilaremos de un archivo como el siguiente:

```
a
aaronita
aarónico
aba
ababa
ababillarse
ababol
abacal
abacalero
...
```

Fichero 2: diccionario.txt

Y necesitaremos una estructura de datos que nos permita consultar este listado de palabras desde nuestro programa. Este contenedor será nuestro TDA Dictionary.

Debido al tipo de información con la que tenemos que trabajar, los contenedores lineales que hemos utilizado hasta ahora no parecen la mejor solución. Ahora, tendremos que trabajar con información que tiene una estructura muy específica, y el orden de inserción de los elementos no parece la forma más cómoda de trabajar. Por este motivo, vamos a utilizar lo que se conoce como contenedores no lineales, o contenedores asociativos.

1.3 Tipos de datos abstractos asociativos

Los TDA no lineales, de forma intuitiva, son aquellos donde no tendría sentido recorrerlo de forma indexada. Entre ellos, los contenedores asociativos son TDAs que permiten almacenar una colección de pares, en la que cada par se conforma de una clave y un valor. No nos importa cómo almacenan los datos sino las capacidades que tienen. Las operaciones más frecuentes de estos contenedores son:

- Añadir un par a la colección
- Eliminar un par de la colección
- Modificar un par existente
- Buscar un valor asociado con una determinada clave.

Las estructuras de datos asociativas que vamos a utilizar y forman parte de la STL son:

- Set
- Map

1.4 Set

La clase set representa un conjunto de elementos que se disponen de manera ordenada y en el que no se repiten elementos. En este caso y a diferencia del TDA map, en la pareja (clave, valor), el valor es siempre nulo. Los datos que insertamos en el set se llaman **claves**.

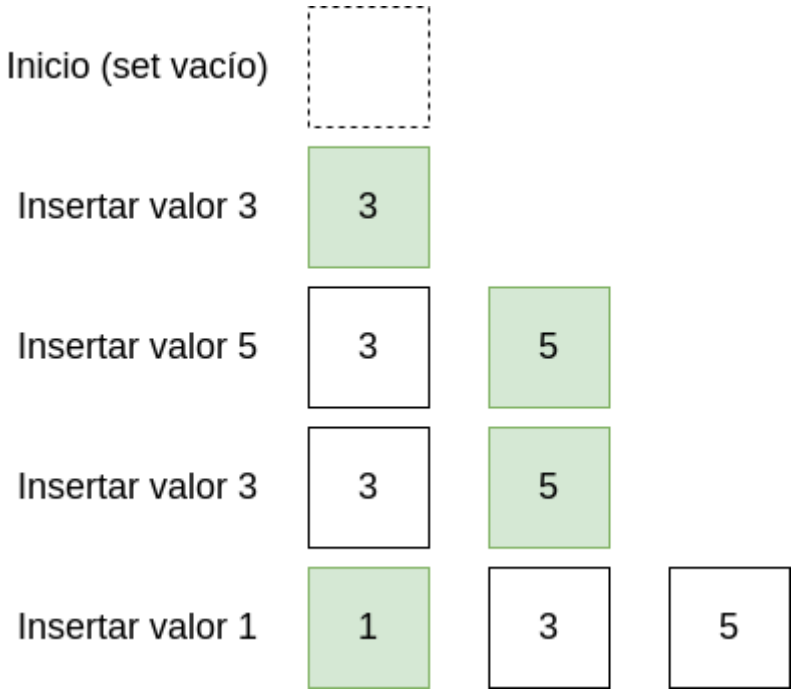


Figura 1: Procedimiento de inserción de valores en un set

Una estructura de tipo set deberá contar, como mínimo, con los siguientes métodos para su correcto funcionamiento:

- Tamaño: size()
- Vacío: empty()
- Borrar: erase()
- Vaciar: clear()
- Insertar: insert()
- Buscar: find()
- Iterador inicio: begin()
- Iterador fin: end()

Se puede consultar la documentación del TDA Set, disponible en la librería standard de C++ (https://www.cplusplus.com/reference/set/set/), para ver la funcionalidad básica de un set (aunque dicho TDA implementa adicionalmente algunas funciones que, a pesar de no ser imprescindibles, facilitan el uso de la estructura en muchos contextos).

1.5 Map

Un map está formado por parejas de valores: al primero se lo conoce como **clave**, y al segundo como el **valor** asociado a dicha clave. No permite valores de clave repetidos y se ordena según su clave. Podemos acceder, de forma directa, al valor asociado a la clave a través de la clave, pero no al revés.

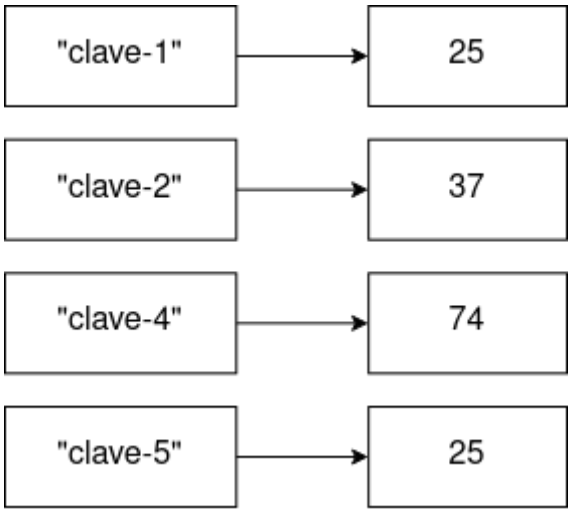


Figura 2: Funcionamiento del TDA Map

Una estructura de tipo map deberá contar, como mínimo, con los siguientes métodos para su correcto funcionamiento:

- Tamaño: size()
- Vacío: empty()
- Borrar: erase()
- Vaciar: clear()
- Insertar: insert()
- Buscar: find()
- Iterador inicio: begin()
- Iterador fin: end()
- Obtención de valor dada una clave: operator[]

Se puede consultar la documentación del TDA Map, disponible en la librería standard de C++ (https://www.cplusplus.com/reference/map/map/), para ver la funcionalidad básica de un map (aunque dicho TDA implementa adicionalmente algunas funciones que, a pesar de no ser imprescindibles, facilitan el uso de la estructura en muchos contextos).

1.6 Iteradores

Los iteradores son un T.D.A. que nos permite acceder a los elementos de distintos contenedores de forma secuencial, abstrayéndonos de la representación interna o **estructura de datos** subyacente. Los pasos a seguir para trabajar con iteradores son:

1. Iniciar el iterador a la primera posición del contenedor (función begin()).
2. Acceder al elemento que apunta (*it, donde it es de tipo iterador)
3. Avanzar el iterador al siguiente elemento del contenedor (++it)
4. Saber cuando hemos recorrido todos los elementos del contenedor (función end()).

2.- T.D.As LettersSet, Bag, LettersBag y Dictionary

En esta práctica trabajaremos con tres TDA abstractos y una template class. Dichos TDAs nos permitirán organizar la información necesaria para jugar una partida del juego de las letras.

2.1 TDA LettersSet

Esta estructura de datos nos va a permitir almacenar la información de un conjunto de letras, del modo en el que las necesitaremos en una partida de las letras, Esta información es similar a la que tendríamos en una partida de *Scrabble*. En concreto, en una partida de las letras, para cada letra disponible, necesitaremos saber:

- El número total de repeticiones de la letra
- La puntuación que nos aporta cada letra al utilizarla en una palabra

La información que tenemos que almacenar apunta directamente al tipo de estructura map. Podemos utilizar como claves cada una de las letras (que sabemos que no se repiten), y como valor un struct que contenga la información para cada una de ellas:

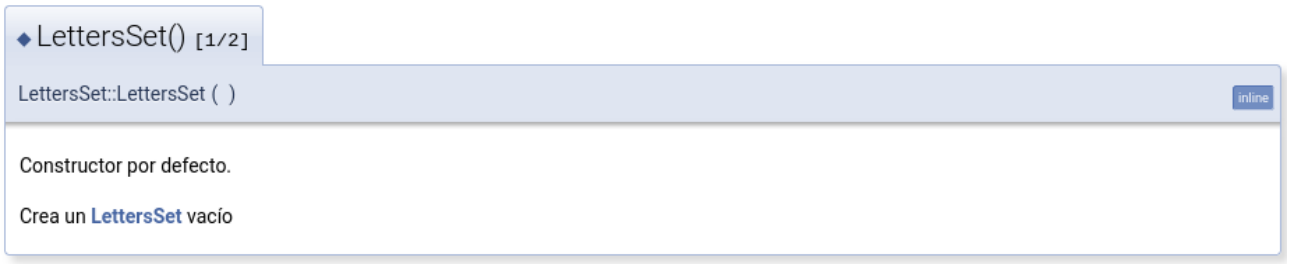
```
struct LetterInfo {
      int repetitions;
      int score;
}
```

Entonces, nuestro TDA tendrá el siguiente dato miembro:

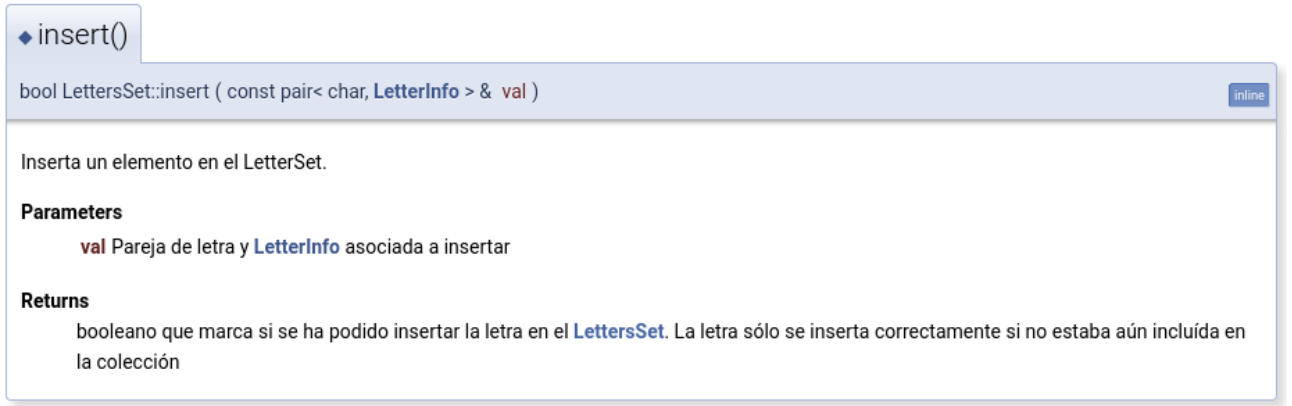
```
map <char, LetterInfo> letters;
```

Que nos permitirá consultar para cada letra su información asociada. Dentro de este TDA, queremos implementar la siguiente funcionalidad:

- Constructores por defecto y de copia



- Funciones para insertar y eliminar un elemento



◆ erase()

bool LettersSet::erase (const char & key) inline

Elimina un carácter del LetterSet.

Parameters

key Carácter a eliminar

Returns

Booleano que indica si se ha podido eliminar correctamente la letra del **LettersSet**

- Funciones para consultar el número de elementos en el conjunto, si el conjunto está vacío, y vaciar el conjunto

◆ clear()

void LettersSet::clear ( ) inline

Limpia el contenido del **LettersSet**.

Elimina el contenido del **LettersSet**

◆ empty()

bool LettersSet::empty ( ) const inline

Consulta si el **LettersSet** es vacío.

Returns

true si el **LettersSet** está vacío, falso en caso contrario

◆ size()

unsigned int LettersSet::size ( ) const inline

Tamaño del LetterSet.

Returns

Número de elementos en el LetterSet

- Función para calcular la puntuación de una palabra a partir de las puntuaciones de sus letras

◆ getScore()

int LettersSet::getScore (string word)

Calcula la puntuación dada una palabra.

Parameters

word String con la palabra cuya puntuación queremos calcular

Returns

Puntuación de la palabra, calculada como la suma de las puntuaciones de cada una de sus letras

- Sobrecarga de los operadores de consulta (es importante observar que no recibe un entero, si no un carácter, y nos devuelve la LetterInfo asociada a ese carácter), asignación, entrada y salida

◆ operator=()

LettersSet & LettersSet::operator= (const LettersSet & cl)

Sobrecarga del operador de asignación.

Parameters

cl LetterSet a copiar

Returns

Referencia al objeto this para poder encadenar el operador

◆ operator[]()

LetterInfo & LettersSet::operator[] (const char & val)

Sobrecarga del operador de consulta.

Permite acceder a los elementos del map que hay en nuestra clase

Parameters

val Carácter a consultar

Returns

Estructura de tipo **LetterInfo** con la información del carácter consultado: Número de repeticiones y puntuación

◆ operator<<

```
ostream& operator<< ( ostream &          os,
                      const LettersSet & cl
                    )
```

friend

Sobrecarga del operador de salida.

Parameters

os Flujo de salida, donde escribir el **LettersSet**

-

cl **LettersSet** que se escribe

◆ operator>>

```
istream& operator>> ( istream &     is,
                      LettersSet & cl
                    )
```

friend

Sobrecarga del operador de entrada.

Parameters

is Flujo de entrada, del que leer el **LettersSet**

-

cl **LettersSet** en el que almacenar la información leída

Programa de prueba - conjunto_palabras.cpp

Para probar el funcionamiento de nuestro TDA, vamos a implementar un pequeño programa que recibirá dos argumentos:

1. Ruta al archivo con información sobre el conjunto de letras
2. Palabra de la que calcular la puntuación

E imprimirá por pantalla la puntuación de dicha palabra.

2.2 Template Bag

Nuestro siguiente TDA deberá funcionar como una bolsa de caracteres. Esto quiere decir que será una colección que contenga caracteres y nos debe permitir extraerlos aleatoriamente y sin reemplazamiento. En la STL no existe ningún contenedor que nos permita extraer cómodamente elementos de forma aleatoria, así que lo que haremos será crear este contenedor por nuestra cuenta.

Ahora, podemos pensar en que este contenedor no tendría por qué tener elementos de tipo char necesariamente. Este contenedor que nos permite introducir elementos y extraerlos aleatoriamente no tiene por qué contener letras. Podría contener números, palabras completas, structs de cualquier tipo, objetos de alguna clase que hubiéramos implementado...

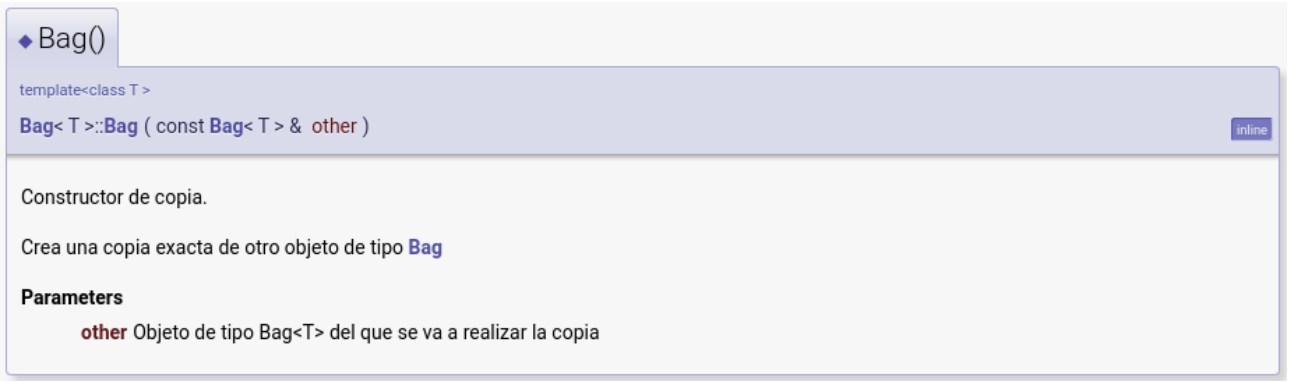
Para dotar a un TDA de esta capacidad de trabajar con cualquier tipo de dato que especifiquemos, C++ nos ofrece una funcionalidad conocida como `templates.` Esta funcionalidad nos permite implementar código genérico, que funcione para cualquier tipo de dato. En particular, nosotros vamos a implementar una clase genérica llamada Bag, que será la que nos permita añadir elementos y extraerlos de forma aleatoria:

```
template <class T>
class Bag {
private:
      vector<T> v;
public:
…
}
```

Como se puede observar en el fragmento anterior, en ningún momento estamos diciendo qué tipo de elementos se almacenan en el vector `elements`. Especificamos que todos los elementos serán de tipo T, porque estamos trabajando con un array, pero no sabemos cuál será ese tipo T.

Una particularidad de las clases template es que su implementación no puede separarse de la declaración del mismo modo en el que lo hacemos con el resto de clases. En este caso, implementaremos la clase completa en el archivo `Bag.h.` Este TDA debe contar, al menos con los siguientes métodos:

- Constructor por defecto y constructor de copia



- Métodos para añadir y extraer elementos de la bolsa, y posibilidad de vaciar la bolsa

◆ add()

```
template<class T >
void Bag< T >::add ( const T &  element )
```

inline

Añade un elemento a la bolsa.

Template Parameters

element elemento del tipo T a añadir a la bolsa

◆ get()

```
template<class T >
T Bag< T >::get ( )
```

inline

Extrae un elemento aleatorio de la bolsa.

Devuelve un elemento aleatorio de la bolsa y lo elimina de la misma

Returns

Elemento de tipo T extraído de la bolsa

Precondition

La bolsa no está vacía

Postcondition

El elemento devuelto se ha eliminado de la bolsa

◆ clear()

```
template<class T >
void Bag< T >::clear ( )
```

inline

Elimina todos los elementos de la bolsa.

Borra todos los elementos almacenados en la bols

- Número de elementos y comprobante de si la bolsa está vacía

◆ size()

```
template<class T >
unsigned int Bag< T >::size ( ) const
```

inline

Tamaño de la bolsa.

Returns

Número de elementos que hay en la bolsa

◆ empty()

```
template<class T >
bool Bag< T >::empty ( )
```

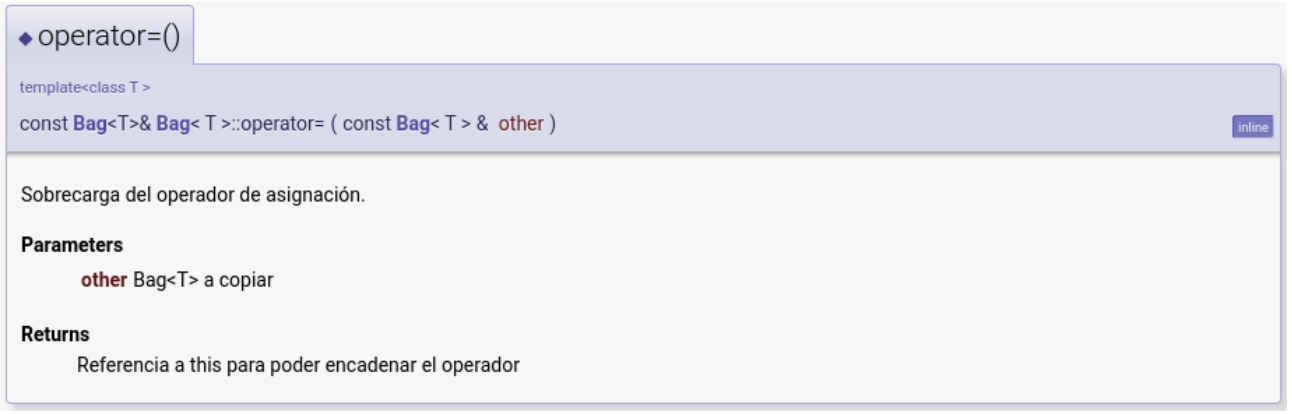
inline

Comprueba si la bolsa está vacía.

Returns

true si la bolsa está vacía, false en caso contrario

- Operador de asignación:



Programa de prueba - bolsa.cpp:

Una vez hemos creado nuestra estructura de datos, ya podemos utilizarla para trabajar con una bolsa de cualquier tipo de datos. Para crear una bolsa de elementos de un tipo concreto, utilizaremos una sintaxis con la que ya estáis familiarizados a trabajar:

```
Bag<int> bolsa_enteros;
```

Para probar el correcto funcionamiento de este TDA, implementaremos un pequeño programa en el archivo `bolsa.cpp`, que se encargará de recibir una serie de argumentos de main:

- El primer argumento podrá ser una letra C o una letra I. Dependiendo de la letra, trabajaremos con una bolsa de caracteres (C) o una bolsa de enteros (I).
- El resto de argumentos serán una lista de enteros o caracteres (según corresponda), que deberán meterse todos en la bolsa, y después ser extraídos aleatoriamente hasta que la bolsa quede vacía.

★ ¡Atención! El único propósito de este ejercicio es evaluar la creación de un TDA usando templates. Los template se resuelven durante la compilación. Esto tiene dos implicaciones de cara a la resolución del ejercicio: primero, nos obliga a crear los dos contenedores en el main y segundo, tenemos que repetir el código para los dos escenarios. En definitiva, NO ES UNA BUENA PRÁCTICA, no podemos definir el tipo de un contenedor basado en templates en tiempo de ejecución y tampoco es buena idea emularlo. El único propósito de este programa es evaluar en el juez vuestra correcta implementación.

2.3 TDA LettersBag

Una vez hemos creado nuestra bolsa, vamos a aprovecharla para crear nuestra bolsa de letras. La bolsa de letras es un TDA que nos va a permitir seleccionar las letras con las que vamos a poder jugar una partida de las letras. Deberá ser capaz de interpretar la información contenida en un LettersSet, y crear una bolsa asociada que tenga tantas letras como hay en el LettersSet, con las repeticiones adecuadas. Además, una vez construida la bolsa de letras, queremos poder extraer letras aleatorias de la misma. La forma de determinar las letras que tendremos disponibles para una partida de las letras será construir una bolsa como esta y extraer aleatoriamente un número de letras determinado de ella.

La template class que hemos implementado antes es la estructura de datos adecuada para trabajar en este caso. Por tanto, nuestra bolsa de palabras tendrá el siguiente atributo privado:

```
class LettersBag {
private:
      Bag <char> letters;
public:
…
}
```

E implementaremos la siguiente funcionalidad en nuestro TDA:

- Constructor por defecto
- Constructor cuyo parámetro es un `LetterSet &.` Es buena práctica reducir el código dentro del constructor AL MÍNIMO, podéis apoyaros en métodos privados para desarrollar la lógica del constructor.

◆ LettersBag()

LettersBag::LettersBag (const LettersSet & letterSet) inline

Constructor dado un LettersSet.

Dado un LettersSet como agumento, este constructor debe rellenar la LettersBag con las letras que contiene el LettersSet, introduciendo cada letra el número de veces indicado por el campo LetterInfo::repetitions. Un ejemplo de ejecución en el que se utilice este constructor es el siguiente:

```
ifstream archivo_letras("letras.txt");

LettersSet conjunto_letras;
archivo_letras >> conjunto_letras;

LettersBag bolsa_letras(conjunto_letras);
```

Parameters

letterSet TDA LettersSet a parsear

- Método para insertar una nueva letra en la bolsa:

◆ insertLetter()

void LettersBag::insertLetter (const char & l) inline

Introduce una letra en la bolsa.

Parameters

l letra a añadir a la LettersBag

- Método que nos permita extraer una sola letra y un conjunto de letras

◆ extractLetter()

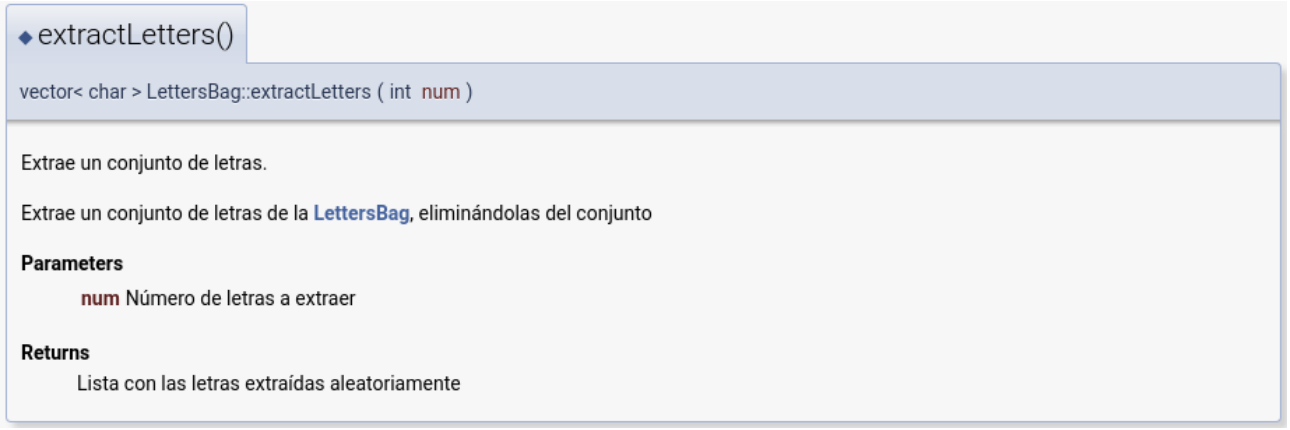
char LettersBag::extractLetter ( )

Extrae una letra aleatoria de la bolsa.

Extrae una letra aleatoria de la bolsa, eliminándola del conjunto

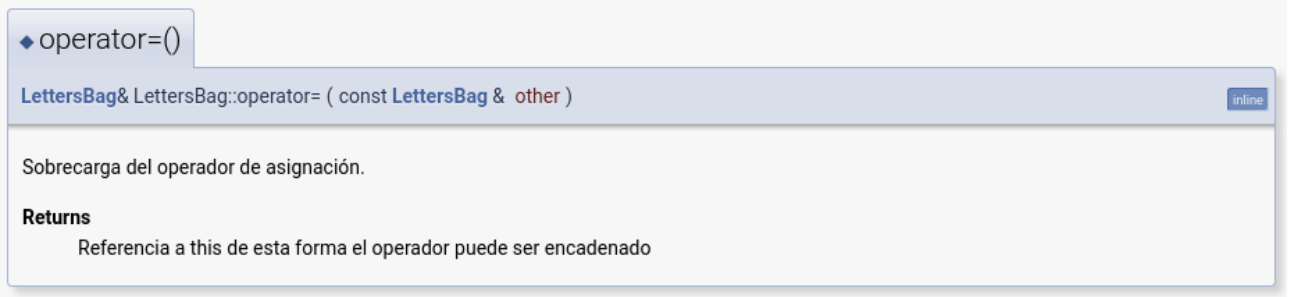
Returns

char representa la letra extraída



- Función para consultar el tamaño de la bolsa, y función para vaciarla

- Sobrecarga de los operadores de asignación y salida



Programa de prueba - bolsa_letras.cpp

Para comprobar el funcionamiento de este TDA, implementaremos un programa de prueba que realice las siguientes operaciones:

- Cree un conjunto de letras (LettersSet), y lo rellene a partir de la información leída de un archivo.
- Cree una bolsa de letras (LettersBag), y la rellene con la información del LettersSet anterior
- Extraiga todas las letras de la LettersBag (aleatoriamente) y las imprima por pantalla.

2.4 TDA Dictionary

El último TDA que implementaremos será el TDA Dictionary. Este TDA nos permitirá mantener en nuestro programa un conjunto de palabras. Como no estaremos interesados en almacenar las definiciones de nuestras palabras, sólo los términos, necesitaremos una estructura que nos permita almacenar strings, sin necesidad de almacenar información más compleja. Además, nos interesará que nuestro conjunto tenga dos propiedades específicas sobre sus elementos:

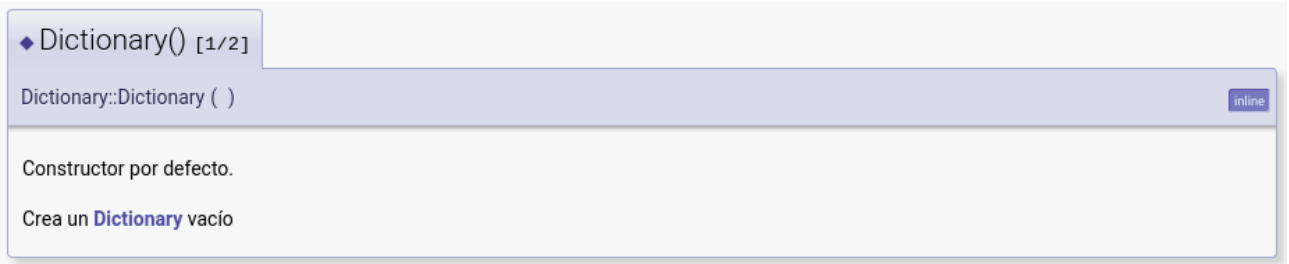
1. Los elementos deben estar ordenados por orden alfabético. Tiene mucho sentido que, en un diccionario, los elementos se ordenen de esta manera, y no en otro orden, independientemente del orden en que se hayan insertado
2. Los elementos no se deben repetir. Estamos manteniendo una lista de palabras de un idioma, no nos interesa tener palabras repetidas, si no saber simplemente qué palabras pertenecen al mismo.

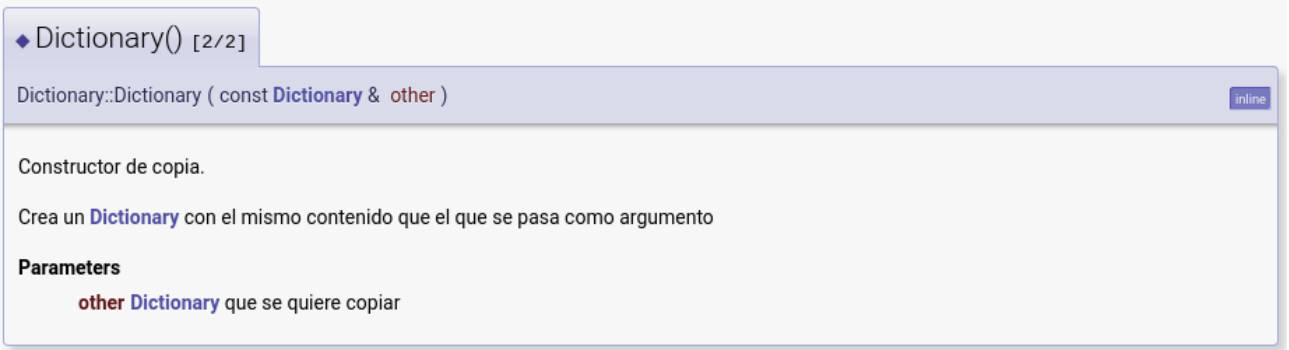
La estructura de datos `set` es muy adecuada para lo que necesitamos, teniendo en cuenta las condiciones que hemos impuesto, así que será el contenedor que utilizaremos para nuestro TDA. Tendremos, por tanto, la siguiente información almacenada:

```
class Dictionary {
private:
      set <string> words;
public:
…
}
```

E implementaremos la siguiente funcionalidad en nuestro TDA:

- Constructores por defecto y de copia





- Métodos para insertar, consultar la presencia, y borrar un elemento

◆ exists()

bool Dictionary::exists (const string & val) const — inline

Indica si una palabra esta en el diccionario o no.

Este método comprueba si una determinada palabra se encuentra o no en el dicccionario

Parameters

palabra la palabra que se quiere buscar.

Returns

Booleano indicando si la palabra existe o no en el diccionario

◆ insert()

bool Dictionary::insert (const string & val) — inline

Inserta una palabra en el diccionario.

Parameters

val palaba a insertar en el diccionario

Returns

Booleano que indica si la inserción ha tenido éxito. Una palabra se inserta con éxito si no existía previamente en el diccionario

◆ erase()

bool Dictionary::erase (const string & val) — inline

Elimina una palabra del diccionario.

Parameters

val Palabra a borrar del diccionario

Returns

Booleano que indica si la palabra se ha borrado del diccionario

- Métodos para consultar el tamaño, consultar si el diccionario está vacío, y limpiar el diccionario

◆ clear()

void Dictionary::clear ( ) — inline

Limpia el **Dictionary**.

Elimina todas las palabras contenidas en el conjunto

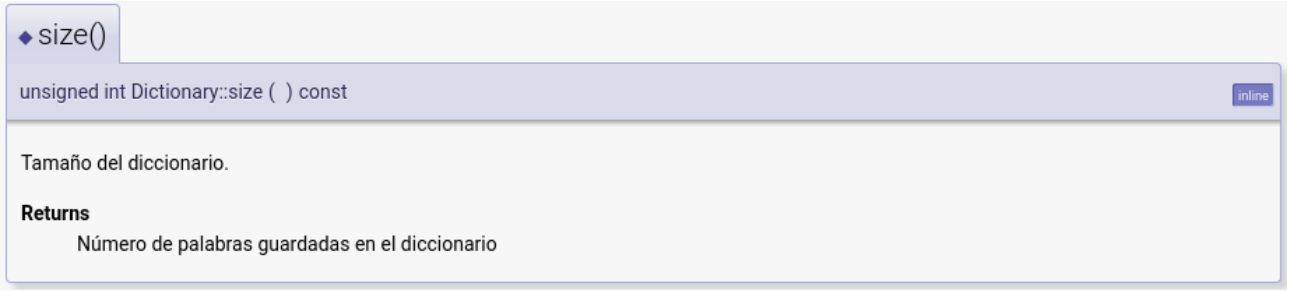
◆ empty()

bool Dictionary::empty ( ) const — inline

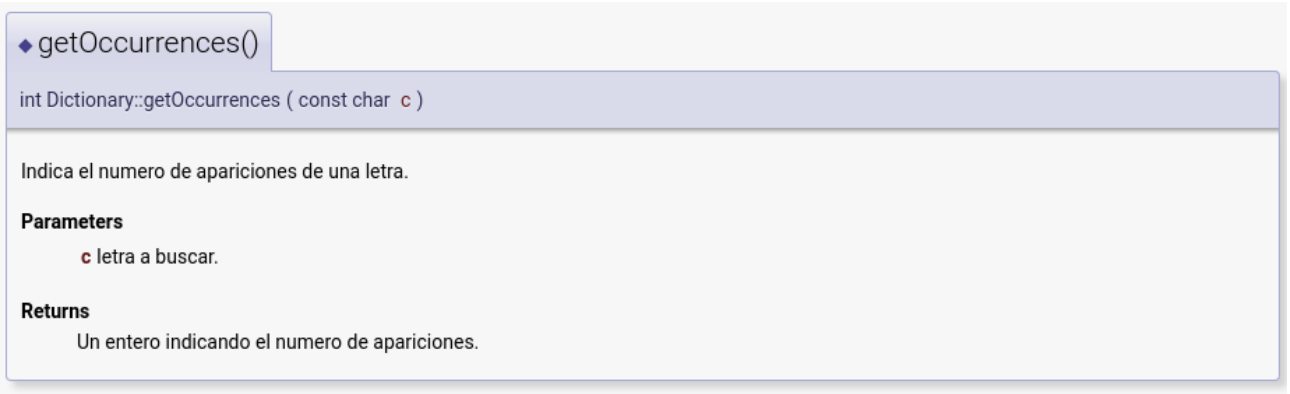
Comprueba si el diccionario está vacio.

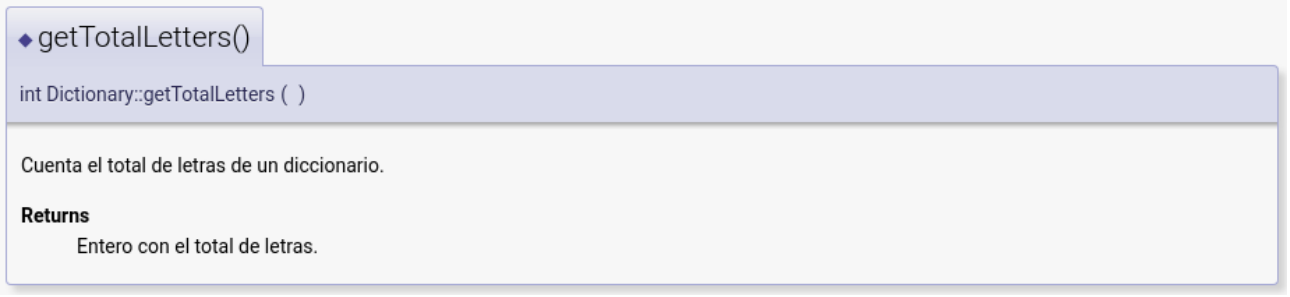
Returns

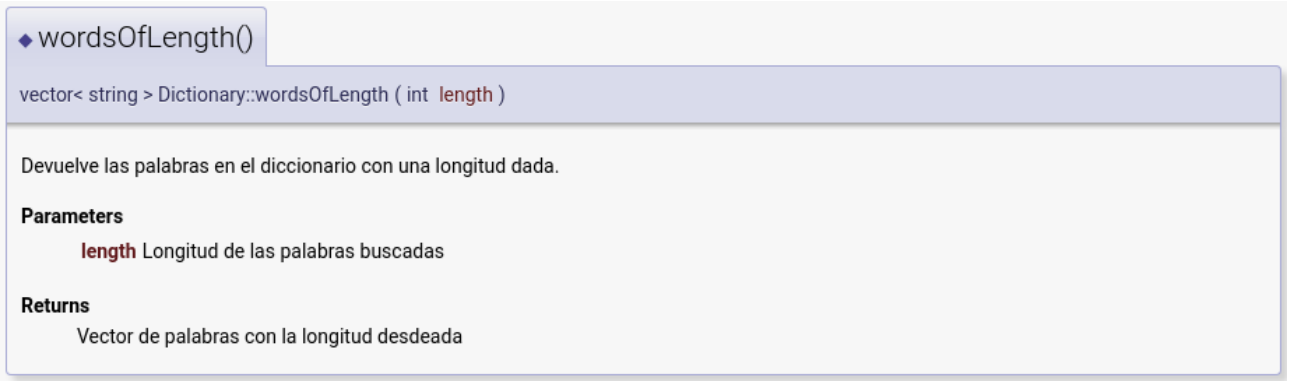
true si el diccionario está vacío, false en caso contrario



- Métodos para obtener el número de ocurrencias de una letra en todo el diccionario, el número total de letras en el diccionario, y las palabras de una determinada longitud







Programas de prueba - palabras_longitud.cpp y cantidad_letras.cpp

Para probar el funcionamiento de este TDA, implementaremos dos programas distintos.

El primero de ellos, palabras_longitud.cpp, recibe dos argumentos

1. Un fichero con las palabras de un diccionario
2. Un entero con la longitud de las palabras que buscamos

Construye un Dictionary con el fichero de las palabras, extrae de dicho diccionario las palabras de la longitud que buscamos y las imprime por pantalla.

El segundo de ellos, cantidad_letras.cpp, recibe dos argumentos

1. Un fichero con las palabras de un diccionario

2. Un fichero de letras

Construye un Dictionary con el fichero de palabras, un LettersSet con el fichero de letras, e imprime por pantalla, para cada letra en el LettersSet, el número de ocurrencias de la letra en el diccionario y su frecuencia relativa (es decir, el número total de ocurrencias de la letra entre el número total de letras en el diccionario).

3.- Práctica a Realizar

1.- Ejercicio obligatorio: Construcción del TDA LettersSet

En el primer ejercicio de la práctica, se propone la implementación del **tipo de dato LettersSet**, que implementa el conjunto de letras que hemos descrito en los apartados anteriores. Para ello, será necesario:

1. Dar una especificación del T.D.A LettersSet.
2. Definir el conjunto de operaciones necesarias para el funcionamiento del conjunto de letras junto con sus especificaciones.
3. Implementar el TDA LettersSet **utilizando como contenedor subyacente un map.** Se podrá utilizar el map que viene implementado en la STL (`map`). Teniendo en cuenta los principios de **ocultamiento de información**, se recomienda separar la declaración y la implementación de dicha clase (se sugieren los nombres de archivo `LettersSet.h` y `LettersSet.cpp`, situados en la carpeta que les corresponda).
4. Se deberán implementar iteradores para poder iterar sobre los diferentes elementos del TDA LettersSet.

2.- Ejercicio obligatorio: Construcción de los TDAs Bag y LettersBag

En el primer ejercicio de la práctica, se propone la implementación de los **tipos de datos Bag y LettersBag**, que implementa la bolsa, y la bolsa de letras que hemos descrito en los apartados anteriores. Para ello, será necesario:

1. Dar una especificación de los T.D.As Bag y LettersBag.
2. Definir el conjunto de operaciones necesarias para el funcionamiento de la bolsa y la bolsa de letras junto con sus especificaciones.
3. Implementar el TDA Bag utilizando `templates` de forma que pueda almacenar cualquier tipo de dato. Los datos deberán almacenarse en un vector estático (opcionalmente se da la opción de implementar utilizando un vector dinámico). Se recomienda mantener la declaración y la implementación de dicha clase en el archivo `Bag.h`.
4. Implementar el TDA LettersBag **utilizando como contenedor subyacente el TDA Bag**. Teniendo en cuenta los principios de **ocultamiento de información**, se recomienda separar la declaración y la implementación de dicha clase (se sugieren los nombres de archivo `LettersBag.h` y `LettersBag.cpp`, situados en la carpeta que les corresponda)

3.- Ejercicio obligatorio: Construcción del TDA Dictionary

En el primer ejercicio de la práctica, se propone la implementación del **tipo de dato Dictionary**, que implementa el diccionario de letras que hemos descrito en los apartados anteriores. Para ello, será necesario:

1. Dar una especificación del T.D.A Dictionary.
2. Definir el conjunto de operaciones necesarias para el funcionamiento del diccionario junto con sus especificaciones.

3. Implementar el TDA Dictionary **utilizando como contenedor subyacente un set.** Se podrá utilizar el set que viene implementado en la STL (`set`). Teniendo en cuenta los principios de **ocultamiento de información**, se recomienda separar la declaración y la implementación de dicha clase (se sugieren los nombres de archivo `dictionary.h y dictionary.cpp`, situados en la carpeta que les corresponda)
4. Se deberán implementar iteradores para poder iterar sobre los diferentes elementos del TDA Dictionary.

4.- Documentación y entrega

Toda la documentación de la práctica se incluirá en el propio Doxygen generado, para ello se utilizarán tanto las directivas Doxygen de los archivos .h y .cpp como los archivos .dox incluídos en la carpeta doc.

1. La documentación debe incluir:
- TODOS los métodos y clases debidamente documentados con la especificación completa. Se valorará positivamente descripciones exhaustivas.
- Las dos tareas de la práctica tienen un ejecutable asociado que debe ser descrito en la página principal o en una página *ad hoc*.
- Todos las páginas de la documentación generada deben ser completas y estar debidamente descritas.

2. A considerar:
- Deben respetarse todos los conocimientos adquiridos en los temas de Abstracción y Eficiencia. En especial el **principio de ocultamiento de información** y las distintas estrategias de **abstracción**.
- Una solución algorítmicamente correcta pero que contradiga el punto anterior será considerada como errónea.
- Una solución algorítmicamente correcta pero que no utilice el contenedor subyacente requerido en cada caso (es decir, debemos implementar LettersSet utilizando un map, Bag utilizando un vector estádido/dinámico, LettersBag utilizando el TDA Bag, y Dictionary utilizando un set), también será considerada como errónea.

3. Código:
- Se dispone de la siguiente estructura de ficheros:
    - /
        - estudiante/ (Aquí irá todo lo que desarrolle el alumno)
            - doc/ (Imágenes y documentos extra para Doxygen)
            - include/ (Archivos cabecera)
            - src/ (Archivos fuente)
        - CMakeList.txt (Instrucciones CMake)
        - Doxyfile.in (Archivo de configuración de Doxygen)
        - juez.sh (Script del juez online **[HAY QUE CONFIGURARLO]**)
- Es importantísimo leer detenidamente y entender qué hace CMakeList.txt.
- El archivo Doxyfile.in no tendríais por qué tocarlo para un uso básico, pero está a vuestra disposición por si queréis añadir alguna variable (no cambiéis las existentes)
- juez.sh crea un archivo submission.zip que incluye ya TODO lo necesario para subir a Prado a la hora de hacer la entrega. No tenéis que añadir nada más, especialmente binarios o documentación ya compilada.

4. Elaboración y puntuación

- La práctica se realizará POR PAREJAS. Los nombres completos se incluirán en la descripción de la entrega en Prado. Cualquiera de los integrantes de la pareja puede subir el archivo a Prado, pero SOLO UNO.
- Desglose:
    - **(0.30)** Ejercicio 1
        - **(0.20)** Implementación
        - **(0.10)** Documentación
    - **(0.30)** Ejercicio 2
        - **(0.20)** Implementación
        - **(0.10)** Documentación
    - **(0.30)** Ejercicio 3
        - **(0.20)** Implementación
        - **(0.10)** Documentación
- La fecha límite de entrega aparecerá en la subida a Prado.