UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2023/2024

Guion de prácticas

Kmer4

Memoria Dinámica dentro de la clase Profile

Mayo de 2024

Índice

1. Definición del problema

La implementación de la clase `Profile` utilizada hasta ahora, almacena los objetos `KmerFreq` en un vector cuyo tamaño se fija en tiempo de compilación. En la definición de la clase de las prácticas anteriores teníamos lo siguiente:

```
class Profile {
...
private:
    static const int DIM_VECTOR_KMER_FREQ = 2000; ///< The capacity of the array _vectorKmerFreq
    static const std::string MAGIC_STRING_T; ///< A const string with the magic string for text files

    std::string _profileId; ///< profile identifier
    KmerFreq _vectorKmerFreq[DIM_VECTOR_KMER_FREQ]; ///< array of KmerFreq
    int _size; ///< Number of used elements in _vectorKmerFreq
}; // end class Profile
```

Esto hace que cada vez que se instancia un objeto de la clase `Profile`, se crea un vector de un tamaño fijo `DIM_VECTOR_KMER_FREQ = 2000`. Una vez creado un objeto `Profile`, su vector no podrá ni ampliarse ni reducirse. Esto tiene el problema de que estaríamos usando siempre un objeto con una capacidad para 2000 objetos, aunque solo tengamos unos pocos objetos `KmerFreq`; o sea, estaremos despercidiando mucha memoria. También tiene el problema, que si el vector se llena, una vez añadidos 2000 objetos, no será posible ampliar la capacidad del vector. Lo único que podríamos hacer en ese caso sería modificar el programa, cambiando el valor 2000 por uno más alto, pero eso llevaría a mayores desperdicios de memoria cuando tengamos pocos objetos `KmerFreq`.

Una manera de evitar los problemas anteriores es que el vector de objetos `KmerFreq` esté situado en la memoria dinámica. Eso permitirá que el vector tenga reservada justo la memoria que necesite en cada momento de la ejecución del programa. El vector podrá crecer o disminuir de tamaño si es necesario.

Por tanto, se hace necesario un cambio mayor en la clase `Profile` para que el vector de objetos `KmerFreq` esté en memoria dinámica. Se debe redefinir la parte privada de la clase para gestionar memoria dinámica dentro de la clase. Los datos miembro de la clase `Profile` se tienen que modificar. Ahora, se necesita un puntero para apuntar a la memoria dinámica del vector. También necesitamos modificar varios métodos de la clase `Profile` e incluir algunos nuevos. Más detalles en las secciones 4 y 5.1.

```
class Profile {
...
private:
    std::string _profileId; ///< Profile identifier
    KmerFreq* _vectorKmerFreq; ///< Dynamic array of KmerFreq
    int _size; ///< Number of used elements in the dynamic array _vectorKmerFreq
    int _capacity; ///< Number of reserved elements in the dynamic array _vectorKmerFreq

    static const int INITIAL_CAPACITY=10;
        ///< Initial capacity for dynamic array _vectorKmerFreq. Should be a number >= 0
    static const int BLOCK_SIZE=20; ///< Size of new blocks for dynamic array _vectorKmerFreq

    static const std::string MAGIC_STRING_T; ///< A const string with the magic string for text files
}; // end class Profile
```

Por último, al igual que ocurriera en la práctica anterior, la función `main()` almacenará en un vector dinámico los distintos objetos profiles que van a intervenir en el proceso de predicción.

2. Arquitectura de las prácticas

Como ya se indicó en prácticas anteriores, las prácticas `Kmer` se han diseñado por etapas; las primeras contienen estructuras más sencillas, sobre las cuales se asientan otras estructuras más complejas que se van completando con nuevas funcionalidades.

En `Kmer4` se cambia la componente privada de la clase `Profile` para alojar el vector de objetos `KmerFreq` de forma eficiente en memoria dinámica, bloque **C'** de la Figura 1; el resto de clases permanecen iguales.

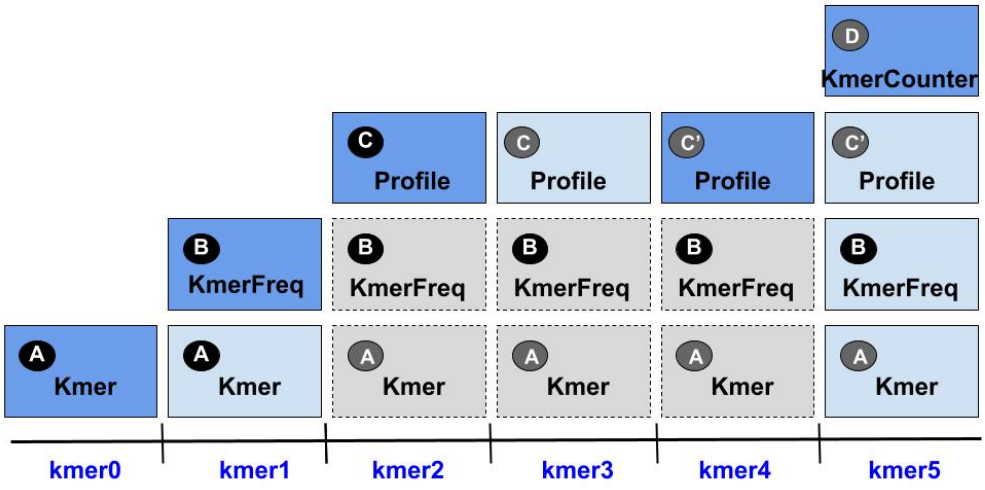

Figura 1: Arquitectura de las prácticas de MP 2024. Los cambios esenciales en las clases (cambio en estructura interna de la clase) se muestran en azul intenso; los que solo incorporan nuevas funcionalidades en azul tenue. En gris se muestran las clases que no sufren cambios en la evolución de las prácticas.

C' Profile.cpp

Manteniendo la interfaz previa, al cambiar la parte privada de la clase con memoria dinámica, se necesita **refactorizar** algunos métodos de la clase y se han de incluir otros nuevos, algunos de ellos privados, para su correcto funcionamiento. Todo esto supone cambios mayores dentro de la clase.

3. Objetivos

El desarrollo de esta práctica `Kmer4` persigue los siguientes objetivos:

- Practicar con punteros y memoria dinámica dentro de una clase.
- Aprender a desarrollar los métodos básicos de una clase que contenga memoria dinámica: constructor de copia, destructor y operador de asignación.

- Practicar con herramientas como el depurador de NetBeans y `valgrind` para rastrear errores en tiempo de ejecución.
- Modificar la clase Profile de prácticas anteriores para incluir gestión de memoria dinámica.
- Refactorizar la clase Profile para reducir al mínimo el número de cambios a realizar en los métodos anteriores de la clase.

4. Vector dinámico con capacidad

Como se ha visto en las clases de teoría, existen varias formas para gestionar vectores dinámicos.

1. Una de ellas sería hacer que en todo momento, la reserva de memoria dinámica del vector se ajuste al número de objetos que estemos usando en ese momento. Esta forma hace que se necesiten dos datos para gestionar un vector dinámico: un puntero (llamado por ejemplo `_array`) al primer elemento del vector y un entero que indique el número de elementos de que disponemos (llamado por ejemplo `_util`). Cada operación de añadir un objeto hará que sea necesario reubicar el vector [1].
2. Una forma alternativa consiste en llevar por un lado, la cuenta del espacio reservado (o sea, la *capacidad*) del vector y por otro, el número de objetos que estemos usando en cada momento. Llamemos por ejemplo, `_capacidad` y `_util` a los datos que guardan estos valores. Dentro de esta alternativa, a su vez, existen varias opciones en la forma en que el vector dinámico se realoja cuando se agota su capacidad:
   a) Una opción sería hacer que el nuevo vector tenga el doble de tamaño al que tenía anteriormente.
   b) Otra opción sería hacer que el nuevo vector tenga como tamaño el que tenía anteriormente más un bloque en el que quepan un número fijo de elementos adicionales.

En esta práctica, usaremos la alternativa 2, ya que permite que el vector dinámico no deba ser continuamente realojado conforme le vamos añadiendo nuevos elementos. Y concretamente nos decantamos por la opción 2*b*.

[1] Como ejercicio didáctico es muy útil pero, está muy lejos de ser realista, dado el esfuerzo que supone la reubicación, que al final se traduce en tiempos.

Ejemplo 1 *Sea un vector reservado en un bloque de capacidad para 5 objetos, con tan solo 3 objetos operativos. Un esquema puede verse en la figura 2.*

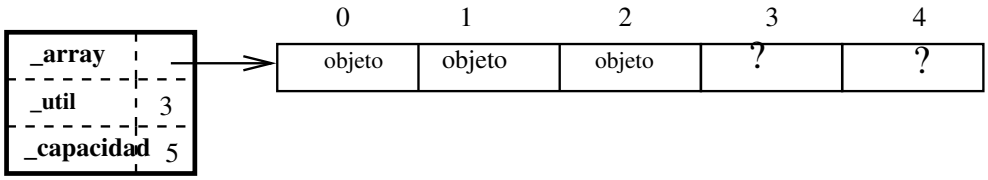

Figura 2: Reserva de espacio para 5 objetos, aunque el vector tan solo disponga de 3 objetos operativos, el resto están a valores por defecto (valor ?).

En el estado del ejemplo 1, a la llegada sucesiva de dos nuevos objetos, estos se podrán alojar en el mismo vector hasta ocupar la capacidad total, sin tener que reubicar el vector. En este caso, basta con incrementar `_util`, como podemos ver en la figura 3.

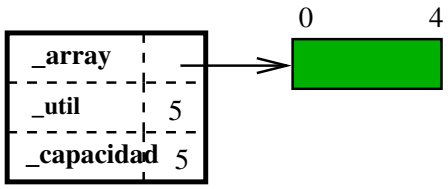

Figura 3: Reserva de espacio para 5 objetos, el vector está completo (`_util` coincide con `_capacidad`).

Siguiendo con nuestro ejemplo, si llegan nuevos objetos, será necesario reubicar el vector en otro espacio redimensionado a una capacidad mayor. Supongamos que se añaden bloques de tamaño 5 cada vez que se agota la capacidad del vector. Al hacerlo, tendríamos un nuevo vector con capacidad para 10 objetos tal como podemos ver en la figura 4 a). Esto es, se ha doblado su capacidad.

Veamos con detalle, cómo se procede cuando se agota la capacidad del vector dinámico (_util coincide con `_capacidad`) y llega un nuevo objeto para ser insertado en el vector. Será necesario realojar el vector anterior en uno más grande, copiar toda la información previa en el nuevo espacio y finalmente liberar el espacio anterior. Gráficamente el proceso puede seguirse en la figura 4.

Los pasos que se han seguido son los siguientes:

1. En (a), el vector está al completo a la llegada del nuevo objeto marcado en rojo.

2. En b) se reserva nueva memoria dinámica, mediante el puntero auxiliar `temp`, con capacidad para $10 + 5$ objetos.

3. En c) se copia el contenido del vector dinámico en el nuevo vector apuntado por `temp`.

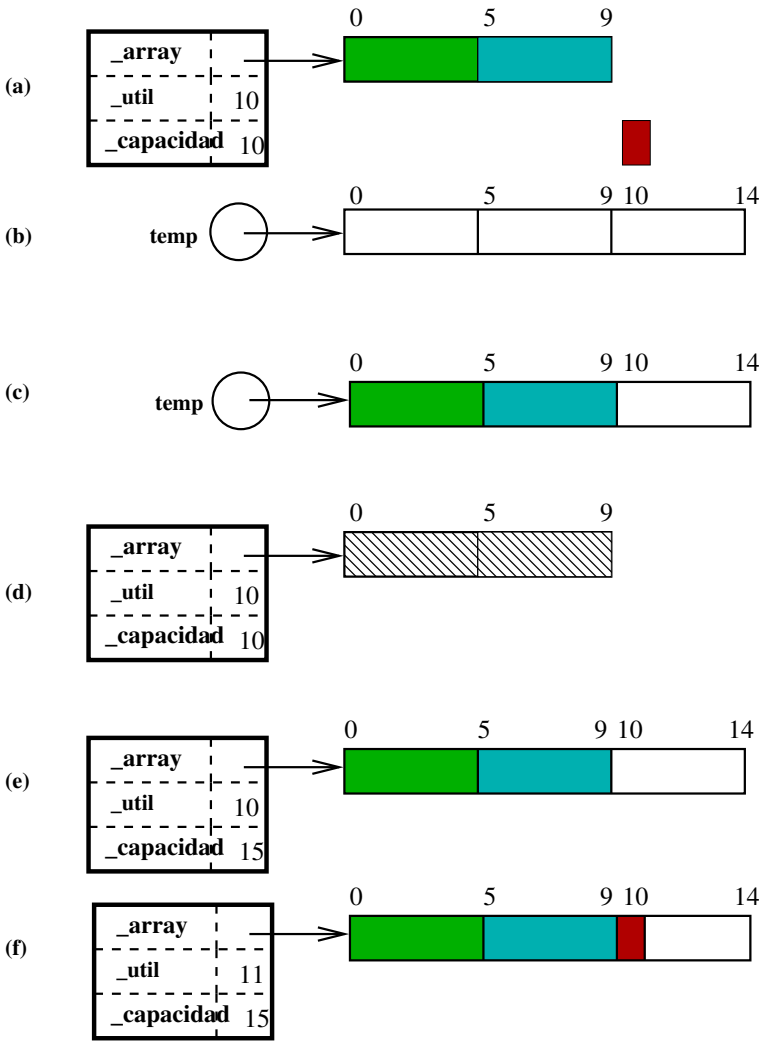

Figura 4: Proceso completo de reubicación y adición de un elemento.

4. En d) se libera la memoria dinámica del vector dinámico original.

5. En e) se corrige la dirección del puntero `_array` para que apunte a la dirección de `temp`, y se ajusta el valor de la capacidad del nuevo vector. Llegado a este punto, ya ha llevado a cabo el realojo del vector en uno de mayor capacidad.

6. Por último, en f) se añade al vector dinámico el nuevo objeto, ya que hay espacio disponible. Se ajusta también el valor del número de elementos usados en el vector.

Si la gestión del vector dinámico se hubiera hecho con la alternativa 1, habría que haber reubicado el vector con cada operación de adición de un objeto. Con la alternativa elegida, se logra reducir el esfuerzo de reubicación.

5. Práctica a entregar

El programa a desarrollar en esta práctica tiene como objetivo exactamente el mismo que el de la práctica anterior. O sea, leer varios ficheros profile que se almacenarán (todos menos el primero) en un array dinámico de objetos `Profile`. Entonces se calculará la distancia del primer profile con todos los profiles del array dinámico. Finalmente, el programa debe mostrar el nombre del fichero que contenga el profile más cercano o lejano al primero.

Para la elaboración de la práctica, puede reutilizar todos los ficheros de código utilizados en la práctica anterior. Tan solo requiere el nuevo fichero `Profile.h`. Configure su proyecto en Netbeans de forma similar a cómo se hizo en prácticas anteriores.

5.1. La clase Profile

A partir de los ficheros de código de la práctica anterior, realizar los siguientes cambios.

- Refactorizar la clase `Profile` utilizando como estructura interna un vector dinámico en la forma que se ha detallado en la sección 4.
  - El constructor sin parámetros de la clase debe reservar memoria. Más concretamente, debe crear un objeto profile vacío (0 componentes `KmerFreq`, pero con una capacidad inicial de `Profile::INITIAL_CAPACITY=10`).
  - El constructor con parámetro (int) debe reservar la memoria para el profile y ajustar el número de componentes, `_size` y `_capacity`, para mantener la consistencia del objeto.
- Implementar constructor de copia, destructor y operador de asignación.
- Revisar el resto de métodos de Profile por si necesitan ser refactorizados, tales como `getCapacity()`, el método `append()`, el de lectura desde un fichero (`load(fileName)`.
- Se recomienda implementar métodos privados como `allocate()`, `deallocate()`, `reallocate()` y `copy()` para evitar repetir código, lo cual sería fuente de futuros errores.

5.2. El módulo main.cpp

El objetivo del programa de esta práctica es exactamente el mismo que el de la práctica Kmer3. O sea, leer diferentes ficheros profile `.prf`, con el fin de calcular la distancia del primero de ellos a cada uno de los demás y hallar el profile con mínima/máxima distancia, según se haya especificado en la línea de comandos el parámetro `min` o bien `max`.

El diseño de la clase `Profile` que se ha hecho en las prácticas anteriores, con una interfaz de la clase (método públicos) y una parte interna (parte privada) permite que los cambios que vamos a hacer en esta clase, no involucren cambios en la función `main()`. Por tanto, no necesita hacer ningún cambio en la función `main()` que implementó para la práctica Kmer3. Sin embargo, puesto que el nombre del programa es ahora `kmer4`, hay un pequeño cambio en la función `showEnglishHelp()` y en los comentarios de la función `main()`, como puede ver en la sección 6.2.

6. Código para la práctica

6.1. Profile.h

```
/*
 * Metodología de la Programación: Kmer4
 * Curso 2023/2024
 */

/**
 * @file Profile.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 17 November 2023, 10:15
 */

#ifndef PROFILE_H
#define PROFILE_H


#include <iostream>
#include "KmerFreq.h"

/**
 * @class Profile
 * @brief It defines a model (profile) for a given biological species. It
 * contains a vector of pairs Kmer-frequency (objects of the class KmerFreq)
 * and an identifier (string) of the profile.
 */
class Profile {
public:

    /**
     * @brief Base constructor. It builds a Profile object with "unknown" as
     * identifier, and an empty vector of pairs Kmer-frequency. The vector will
     * have Kmer::INITIAL_CAPACITY as initial capacity.
     */
    Profile();


    /**
     * @brief It builds a Profile object with "unknown" as
     * identifier, and a vector with a size of @p size pairs
     * Kmer-frequency. The vector will also have @p size as initial capacity.
     * Each pair will be initialized as Kmer::MISSING_NUCLEOTIDE for the Kmer
     * and 0 for the frequency.
     * @throw std::out_of_range Throws a std::out_of_range exception if
     * @p size <0
     * @param size The size for the vector of kmers in this Profile.
     * Input parameter
     */
    Profile(int size);

    /**
     * @brief Copy constructor
     * @param orig the Profile object used as source for the copy. Input
     * parameter
     */
    Profile(Profile orig);

    /**
     * @brief Destructor
     */
    ~Profile();

    /**
     * @brief Overloading of the assignment operator for Profile class.
     * Modifier method
     * @param orig the Profile object used as source for the assignment. Input
     * parameter
     * @return A reference to this object
```

```
 */
Profile operator=(Profile orig);

/**
 * @brief Returns the identifier of this profile object.
 * Query method
 * @return A const reference to the identifier of this profile object
 */
std::string getProfileId();

/**
 * @brief Sets a new identifier for this profile object.
 * Modifier method
 * @param id The new identifier. Input parameter
 */
void setProfileId(std::string id);


/**
 * @brief Gets a const reference to the KmerFreq at the given position
 * of the vector in this object.
 * Query method
 * @param index the position to consider. Input parameter
 * @throw std::out_of_range Throws an std::out_of_range exception if the
 * given index is not valid
 * @return A const reference to the KmerFreq at the given position
 */
KmerFreq at(int index);

/**
 * @brief Gets a reference to the KmerFreq at the given position of the
 * vector in this object
 * Query and modifier method
 * @param index the position to consider. Input parameter
 * @throw std::out_of_range Throws an std::out_of_range exception if the
 * given index is not valid
 * @return A reference to the KmerFreq at the given position
 */
KmerFreq at(int index);

/**
 * @brief Gets the number of KmerFreq objects.
 * Query method
 * @return The number of KmerFreq objects
 */
int getSize();

/**
 * @brief Gets the capacity of the vector of KmerFreq objects.
 * Query method
 * @return The capacity of the vector of KmerFreq objects
 */
int getCapacity();

/**
 * @brief Gets the distance between this Profile object (\f$P_1\f$) and
 * the given argument object @p otherProfile (\f$P_2\f$).
 * The distance between two Profiles \f$P_1\f$ and \f$P_2\f$ is
 * calculated in the following way:
 *
 * \f$d = \frac{ \sum_{kmer_i(P_1)} | rank_{kmer_i(P_1)}^{P_1} -
 * rank_{kmer_i(P_1)}^{P_2} | }{size(P_1) * size(P_2) }\f$,
 *
 * where \f$kmer_i(p_j)\f$ is the kmer \f$i\f$ of the Profile \f$p_j,
 * j \in \{1, 2\}\f$ and \f$rank_{kmer_i(p_j)}^{p_k}\f$ is the ranking
 * of the kmer \f$i\f$ of the Profile \f$p_j, j \in \{1, 2\}\f$ in the
 * Profile \f$p_k\f$.
 *
 * The rank of a kmer is the position in which it
 * appears in the list of KmerFreq. We consider 0 as the
 * first position (rank equals to 0). When calculating
 * \f$rank_{kmer_i(P_1)}^{P_2}\f$, if the kmer \f$kmer_i(P_1)\f$
 * does not appears in the Profile \f$P_2\f$ we consider that the rank
 * is equals to the size of Profile \f$P_2\f$.
 * Query method
 * @param otherProfile A Profile object. Input parameter
 * @pre The list of kmers of this and otherProfile should be ordered in
 * decreasing order of frequency. This is not checked in this method.
 * @throw Throws a std::invalid_argument exception if the implicit object
 * (*this) or the argument Profile object are empty, that is, they do not
 * have any kmer.
 * @return The distance between this Profile object and the given
 * argument @p otherProfile.
 */
double getDistance(Profile otherProfile);

/**
 * @brief Searchs the given kmer in the list of kmers in this
 * Profile. If found, it returns the position where it was found. If not,
 * it returns -1. We consider that position 0 is the first kmer in the
 * list of kmers and this->getSize()-1 the last kmer.
 * Query method
 * @param kmer A kmer. Input parameter
 * @return If found, it returns the position where the kmer
 * was found. If not, it returns -1
 */
int findKmer(Kmer kmer);

/**
 * @brief Obtains a string with the following content:
```

```
     * - In the first line, the profile identifier of this Profile
     * - In the second line, the number of kmers in this Profile
     * - In the following lines, each one of the pairs kmer-frequency
     * (separated by a whitespace).
     * Query method
     * @return A string with the number of kmers and the list of pairs of
     * kmer-frequency in the object
     */
    std::string toString();

    /**
     * @brief Sorts the vector of KmerFreq in decreasing order of frequency.
     * If two KmerFreq objects have the same frequency, then the alphabetical
     * order of the kmers of those objects will be considered (the object
     * with a kmer that comes first alphabetically will appear first).
     * Modifier method
     */
    void sort();

    /**
     * @brief Saves this Profile object in the given file.
     * Query method
     * @param fileName A c-string with the name of the file where this Profile
     * object will be saved. Input parameter
     * @throw std::ios_base::failure Throws a std::ios_base::failure exception
     * if the given file cannot be opened or if an error occurs while writing
     * to the file
     */
    void save(char fileName[]);

    /**
     * @brief Loads into this object the Profile object stored in the given
     * file. Note that this method should remove any Kmer-frequency pairs that
     * this object previously contained.
     * Modifier method
     * @param fileName A c-string with the name of the file where the Profile
     * will be stored. Input parameter
     * @throw std::out_of_range Throws a std::out_of_range exception if the
     * number of kmers in the given file, cannot be allocated in this Profile
     * because it exceeds the maximum capacity or if the number of kmers read
     * from the given file is negative.
     * @throw std::ios_base::failure Throws a std::ios_base::failure exception
     * if the given file cannot be opened or if an error occurs while reading
     * from the file
     * @throw throw std::invalid_argument Throws a std::invalid_argument if
     * an invalid magic string is found in the given file
     */
    void load(char fileName[]);

    /**
     * @brief Appends a copy of the given KmerFreq to this Profile object.
     * If the kmer is found in this object, then its frequency is increased
     * with the one of the given KmerFreq object. If not, a copy of the
     * given KmerFreq object is appended to the end of the list of
     * KmerFreq objects in this Profile.
     * Modifier method
     * @thow std::out_of_range Throws a std::out_of_range exception in case
     * that a new element must be appended to the end of the array and the
     * number of elements in the array of KmerFreq is equals to the capacity
     * of that array. In that case, the array is full, and no more elements
     * can be appended to the array.
     * @param kmerFreq The KmerFreq to append to this object. Input paramether
     */
    void append(KmerFreq kmerFreq);

    /**
     * @brief Normalizes the Kmers of the vector of KmerFreq in this object.
     * That is, for each Kmer in the vector, all its characters are converted
     * to uppercase. Then, invalid characters are replaced by the
     * MISSING_NUCLEOTIDE value.
     * Modifier method
     * @param validNucleotides a string with the list of characters (nucleotides)
     * that should be considered as valid. Input parameter
     */
    void normalize(std::string validNucleotides);

    /**
     * @brief Deletes the KmerFreq object from the vector of KmerFreq in this
     * object at the position @p pos. We consider that the first element has
     * position 0, and the last element position size()-1.
     * Modifier method
     * @param pos The index of the position to be deleted. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if @p pos
     * is not in the range from 0 to size()-1 (both included).
     */
    void deletePos(int pos);

    /**
     * @brief Deletes the KmerFreq objects from the vector of KmerFreq in this
     * object which verifies one the following two criteria:
     *  -# The argument deleteMissing is true and the Kmer contains an unknown
     * nucleotide
     *  -# The frequency is less or equals to @p lowerBound.
     *
     * Note that the number of elements in the argument array could be modified.
     * Modifier method
     * @param deleteMissing A bool value that indicates whether kmers with any
     * unknown nucleotide should be removed. This parameter is false by default.
     * Input parameter
     * @param lowerBound An integer value that defines which KmerFreq objects
```

```
     * should be deleted from the vector of KmerFreq in this object.
     * KmerFreq objects with a frequency less or equals to this value, are
     * deleted. This parameter has zero as default value.
     * Input parameter
     */
    void zip(bool deleteMissing=false, int lowerBound = 0);

    /**
     * @brief Appends to this Profile object, the list of pairs
     * kmer-frequency objects contained in the Profile @p profile. This
     * method uses the method append(const KmerFreq& kmerFreq) to
     * append the pairs kmer-frequency contained in the argument
     * Profile @p profile
     * Modifier method
     * @param profile A Profile object. Input parameter
     */
    void join(Profile profile);

private:
    std::string _profileId; ///< Profile identifier
    KmerFreq* _vectorKmerFreq; ///< Dynamic array of KmerFreq
    int _size; ///< Number of used elements in the dynamic array _vectorKmerFreq
    int _capacity; ///< Number of reserved elements in the dynamic array _vectorKmerFreq

    static const int INITIAL_CAPACITY=10; ///< Default initial capacity for the dynamic array
      _vectorKmerFreq. Should be a number >= 0
    static const int BLOCK_SIZE=20; ///< Size of new blocks in the dynamic array _vectorKmerFreq

    static const std::string MAGIC_STRING_T; ///< A const string with the magic string for text files
};

#endif /* PROFILE_H */
```

6.2. main.cpp

```
/*
 * Metodología de la Programación: Kmer4
 * Curso 2023/2024
 */

/*
 * File:   main.cpp
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 17 November 2023, 12:45
 */

#include <iostream>

#include "Profile.h"

using namespace std;


/**
 * Shows help about the use of this program in the given output stream
 * @param outputStream The output stream where the help will be shown (for example,
 * cout, cerr, etc)
 */
void showEnglishHelp(ostream& outputStream) {
    outputStream << "ERROR in Kmer4 parameters" << endl;
    outputStream << "Run with the following parameters:" << endl;
    outputStream << "kmer4 [-t min|max] <file1.prf><file2.prf> [ ... <filen.prf>]" << endl;
    outputStream << endl;
    outputStream << "Parameters:" << endl;
    outputStream << "-t min | -t max: search for minimun distances or maximum distances (-t min by default)"
      << endl;
    outputStream << "<file1.prf>: source profile file for computing distances" << endl;
    outputStream << "<file2.prf> [ ... <filen.prf>]: target profile files for computing distances" << endl;
    outputStream << endl;
    outputStream << "This program computes the distance from profile <file1.prf> to the rest" << endl;
    outputStream << endl;
}

/**
 * This program reads an undefined number of Profile objects from the set of
 * files passed as parameters to main(). All the Profiles object, except the
 * first one, must be stored in a dynamic array of Profile objects. Then,
 * for each Profile in the dynamic array, this program prints to the
 * standard output the name of the file of that Profile and the distance from
 * the first Profile to the current Profile.
 * Finally, the program should print in the standard output, the name of
 * the file with the Profile with the minimum|maximum distance to the Profile
 * of the first file and its profile identifier.
 *
 * At least, two Profile files are required to run this program.
 *
 * This program assumes that the profile files are already normalized and
 * sorted by frequency. This is not checked in this program. Unexpected results
 * will be obtained if those conditions are not met.
 *
 * Running sintax:
```

```
 * > kmer4 [-t min|max] <file1.prf><file2.prf> [   ... <filen.prf>]
 *
 * Running example:
 * > kmer4 ../Genomes/human1.prf ../Genomes/worm1.prf ../Genomes/mouse1.prf
Distance to ../Genomes/worm1.prf: 0.330618
Distance to ../Genomes/mouse1.prf: 0.224901
Nearest profile file: ../Genomes/mouse1.prf
Identifier of the nearest profile: mus musculus
 *
 * Running example:
 * > kmer4 -t max ../Genomes/human1.prf ../Genomes/worm1.prf ../Genomes/mouse1.prf
Distance to ../Genomes/worm1.prf: 0.330618
Distance to ../Genomes/mouse1.prf: 0.224901
Farthest profile file: ../Genomes/worm1.prf
Identifier of the farthest profile: worm
 */
int main(int argc, char* argv[]) {

    // Process the main() arguments

    // Allocate a dynamic array of Profiles

    // Load the input Profiles

    // Calculate and print the distance from the first Profile to the rest

    // Print name of the file and identifier that takes min|max distance to the first one

    // Deallocate the dynamic array of Profile
    return 0;
}
```