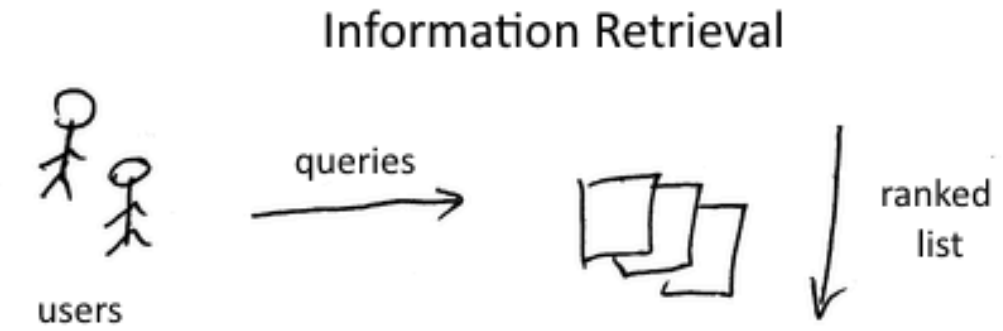
# Práctica 2: Indexador




Autor: Antonio Martínez Santa
Universidad de Alicante

## Contexto, problema a solucionar e hipótesis de partida

Una vez que ya hicimos el proyecto del tokenizador ahora nos metemos en un paso más en el campo de la recuperación de información, en este proyecto se pide la implementación de un indexador. ¿Qué es un indexador? Básicamente un indexador es un sistema que almacena las palabras en un índice que se han obtenido de un documento.

En este proyecto ya que se hará uso del anterior tokenizador que desarrollé en la anterior práctica voy a hacer un pequeño resumen para saber en qué consistirá este proyecto. A grosso modo el tokenizador dividirá las palabras de un fichero que queramos indexar y el indexador se encargará de almacenar esos términos en caso de no ser palabras de parada, esto es una cosa que explicaré más tarde.

La implementación de un indexador es realmente relevante a la hora de realizar un buscador debido a que necesitamos tener constancia de qué documentos proviene cada término para así calcular la relevancia de un documento frente a la query que requiere el usuario a contestar.

Como en la anterior práctica se parte de un código de cabecera que nos proporciona el profesor para implementar con cosas básicas de las clases y un diagrama sobre cuál será el funcionamiento del indexador que posteriormente veremos como implementar. El problema que debemos solucionar es que nos enfrentamos a una cantidad ingente de términos a indexar ya que como podemos imaginar un indexador tendrá miles de términos en su índice ya que los documentos suelen tener demasiados de ellos al igual que puede ser este

documento de la memoria un claro ejemplo, esto como lo pensamos en insostenible pero tenemos un comodín que son las palabras de parada, éstas nos serán muy útiles ya que son palabras que no aportan nada al texto que se está leyendo que suelen ser preposiciones, determinantes, etc…

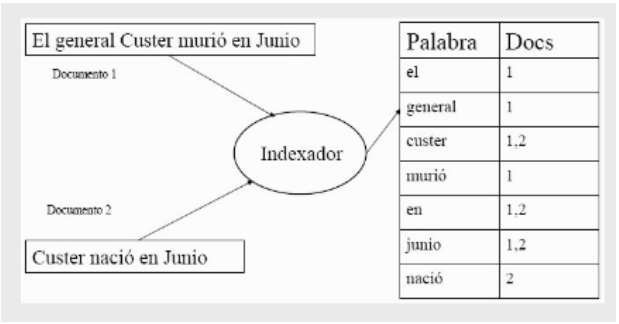
Una vez revisé el documento de ayuda para comprender el funcionamiento del indexador propuse una hipótesis de partida que fue:

1. Crear un fichero que contenga los nombres de los ficheros que queremos indexar
2. Coger cada uno de los ficheros y tokenizarlo
3. Una vez obtenidos estos tokens eliminamos palabras de parada y aplicamos stem
4. Indexamos los términos que han quedado después de aplicar stemming y stopwords
5. Indexamos el documento
    a. Si el documento ya está indexado comprobar su fecha de modificación para saber si hay que volver a indexarlo o tenemos la última versión

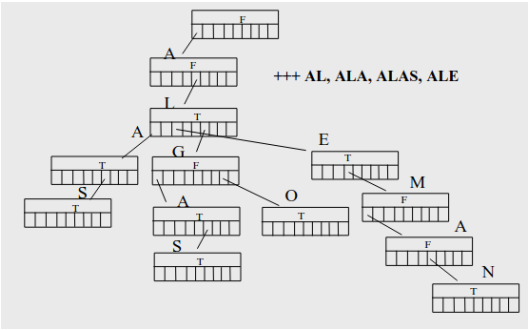
Con este prototipo inicial podemos ir viendo cómo podemos indexar cada uno de los distintos documentos que queramos indexar en nuestro índice que como ya se prevé tendremos un índice de documentos y un índice de términos.

## Análisis de la solución implementada

La solución que he propuesto ha sido el manejo de las distintas tablas Hash que están descritas en la cabecera del fichero *indexadorHash.h*. Hay otras alternativas para implementar como lo son el de los índices invertidos que vimos en clase de teoría o el método de indexación por trie.




Método de índices invertidos

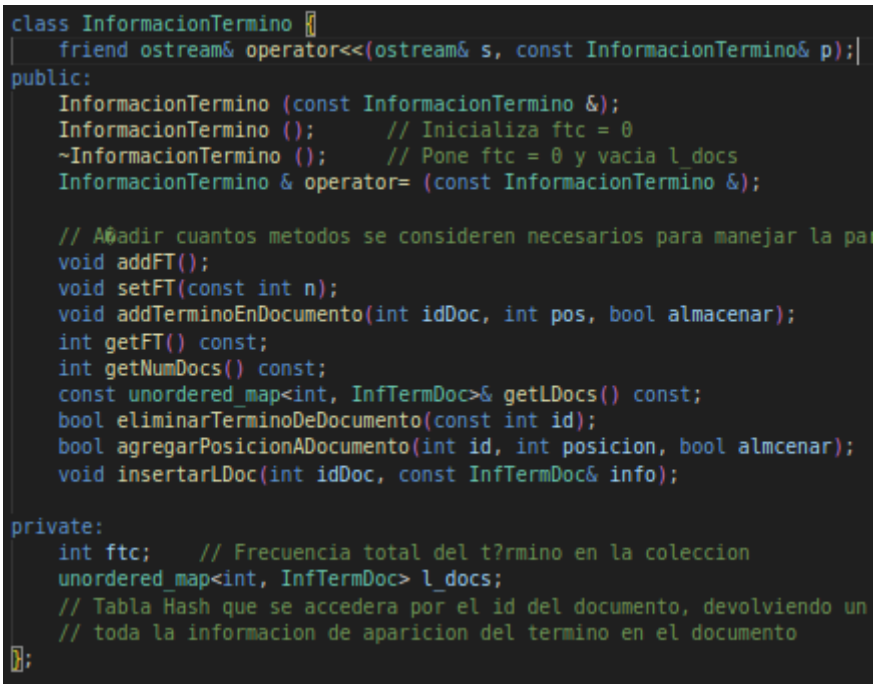



Método del trie

Yo consideré que estos métodos eran más complicados de implementar ya que como tal no estaba así explicado en el pdf donde se veía la estructura que tenía el indexador con la cabecera que se nos proporcionó. En resumen, lo he resuelto por tablas hash. La solución implementada voy a hacer hincapié en las partes más importantes. En la cabecera del fichero *indexadorInformacion.h* están todos los métodos y clases que tenemos que implementar que son constructores y destructores básicamente, aparte de esto yo he implementado métodos que me han sido convenientes para implementar el resto de funcionalidades, estos métodos han variado con respecto a la clase que se trataba, voy a ir haciendo un resumen de cada una y vemos cómo se ha implementado.

## Clase InformacionTermino

Esta clase representa el almacenamiento de los datos asociados a un término específico indexado. Con respecto a sus atributos privados esta clase guarda la frecuencia total de un término en el corpus de documentos y un hash map de posiciones por cada documento que indica en qué documentos aparece el término y en qué posiciones específicas.



```
class InformacionTermino {
    friend ostream& operator<<(ostream& s, const InformacionTermino& p);
public:
    InformacionTermino (const InformacionTermino &);
    InformacionTermino ();      // Inicializa ftc = 0
    ~InformacionTermino ();     // Pone ftc = 0 y vacia l_docs
    InformacionTermino & operator= (const InformacionTermino &);

    // A@adir cuantos metodos se consideren necesarios para manejar la pa
    void addFT();
    void setFT(const int n);
    void addTerminoEnDocumento(int idDoc, int pos, bool almacenar);
    int getFT() const;
    int getNumDocs() const;
    const unordered_map<int, InfTermDoc>& getLDocs() const;
    bool eliminarTerminoDeDocumento(const int id);
    bool agregarPosicionADocumento(int id, int posicion, bool almcenar);
    void insertarLDoc(int idDoc, const InfTermDoc& info);

private:
    int ftc;    // Frecuencia total del t?rmino en la coleccion
    unordered_map<int, InfTermDoc> l_docs;
    // Tabla Hash que se accedera por el id del documento, devolviendo un
    // toda la informacion de aparicion del termino en el documento
};
```

Aquí se ha hecho uso de métodos auxiliares que han sido útiles para controlar la parte privada como los que podemos observar debajo de los getters. Estos métodos posteriormente serán explicados en el análisis del indexador.

## Clase InfTermDoc

Esta clase representa la información de un término dentro de un solo documento. Su función principal es almacenar la frecuencia de un término en concreto dentro de ese documento y las posiciones en las que aparece. Esto permite que posteriormente se pueda calcular la relevancia en el buscador. Esta clase complementa la clase anterior ya que es el *valor* del hash map al que se accede por idDocumento.

```
class InfTermDoc {
    friend ostream& operator<<(ostream& s, const InfTermDoc& p);
public:
    InfTermDoc (const InfTermDoc &);
    InfTermDoc ();      // Inicializa ft = 0
    ~InfTermDoc ();     // Pone ft = 0
    InfTermDoc & operator= (const InfTermDoc &);

    // A?adir cuantos m?todos se consideren necesarios para manej
    void addFT();
    void setFT(int n);
    bool addPositionTerm(int pos, bool almacenar);
    bool addPositionDocument(int id, int pos, bool almacenar);
    int getFT() const;
    const list<int>& getPositionTerm() const;
private:
    int ft; // Frecuencia del t?rmino en el documento
    list<int> posTerm;
    // Solo se almacenar? esta informaci?n si el campo privado de
    // Lista de n?meros de palabra en los que aparece el t?rmino
    //(la primera palabra del documento). Se numerar?n las palabr
};
```

Como podemos observar no han hecho falta más que setters y getters como en el resto para manejar la parte privada de la clase.

## Clase InfDoc

Esta clase representa la información de un documento, sus atributos privados son id, totalPalabras, totalPalabrasSinParada, totalPalabrasDiferentes, tamañoBytes y un atributo que nos dejan a libre elección sobre la fecha, en mi caso por sencillez fue un *time_t*. La cabecera final de esta clase es la siguiente:

```
class InfDoc {
    friend ostream& operator<<(ostream& s, const InfDoc& p);
public:
    InfDoc (const InfDoc &);
    InfDoc ();
    ~InfDoc ();
    InfDoc & operator= (const InfDoc &);

    // A?adir cuantos metodos se consideren necesarios para manejar la parte privada de la clase
    int getIdDoc() const;
    int getNumPal() const;
    int getNumPalSinParada() const;
    int getNumPalDiferentes() const;
    int getTamBytes() const;
    time_t getFechaModificacion() const;

    void setIdDoc(int id);
    void setNumPal(int pal);
    void setNumPalSinParada(int pal);
    void setNumPalDifernetes(int pal);
    void setTamBytes(int bytes);
    void setFechaModificacion(const time_t& fecha);
    void addNumPal();
    void addNumPalSinParada();
    void addNumPalDiferentes();

private:
    int idDoc;
    // Identificador del documento. El primer documento indexado en la colecci?n ser? el identificador 1
    int numPal; // N? total de palabras del documento
    int numPalSinParada;    // N? total de palabras sin stop-words del documento
    int numPalDiferentes;
    // N? total de palabras diferentes que no sean stop-words (sin acumular la frecuencia de cada una de ellas)
    int tamBytes;   // Tama?o en bytes del documento
    // Atributo correspondiente a la fecha y hora (completa) de modificaci?n del documento. El tipo "Fecha/hora
    time_t fechaModificacion;
};
```

En esta clase únicamente se han implementado getters y setters, no me ha hecho falta nada más allá de este tipo de métodos, por comodidad ha habido “duplicidad de métodos” setters ya que era mucho más cómodo un método que sumara +1 a cada campo por

separado y otros como setter tal cual se sustituir completamente el valor por el que se le indique en el parámetro, éste último ha sido especialmente útil a la hora de leer un fichero de indexación.

## Clase InfColeccionDocs

Esta clase se encarga de almacenar el conjunto completo de documentos indexados, no se encarga de almacenar los documentos como tal si no de almacenar toda la información estadística que hay sobre ellos como por ejemplo la cantidad de documentos o la cantidad de palabras totales que hay entre todos.

```
class InfColeccionDocs {
    friend ostream& operator<<(ostream& s, const InfColeccionDocs& p);
public:
    InfColeccionDocs (const InfColeccionDocs &);
    InfColeccionDocs ();
    ~InfColeccionDocs ();
    InfColeccionDocs & operator= (const InfColeccionDocs &);

    // A@adir cuantos metodos se consideren necesarios para manejar la
    void setNumDocs(int docs);
    void setTotalPalabras(int pal);
    void setPalabrasSinParada(int pal);
    void setPalabrasDiferentes(int pal);
    void setBytes(int bytes);

    int getNumDocs() const;
    int getNumTotalPal() const;
    int getNumTotalPalSinParada() const;
    int getNumTotalPalDiferentes() const;
    int getTamBytes() const;

    void agregarDocumentoAColeccion(const InfDoc& documento);
    void borrarDocumentoDeColeccion(const InfDoc& documento);
private:
    int numDocs;    // N@ total de documentos en la colecci?n
    int numTotalPal;
    // N@ total de palabras en la colecci?n
    int numTotalPalSinParada;
    // N@ total de palabras sin stop-words en la colecci?n
    int numTotalPalDiferentes;
    // N@ total de palabras diferentes en la colecci?n que no sean stop
    int tamBytes;   // Tama@o total en bytes de la coleccion
};
```

Aquí se implementaron dos funciones extremadamente importantes que fueron agregarDocumentoAColeccion y borrarDocumentoDeColeccion. El funcionamiento se puede intuir qué es lo que hace y esto fue muy útil aplicarlo en el método BorraDoc de la clase IndexadorHash. ¿Por qué? Como he dicho antes, esta clase se encarga de almacenar la información de todos los ficheros que hay indexador como el número, la cantidad de palabras sin parada, etc… Por lo que tener un método auxiliar que reste estos campos del documento que le pasemos por parámetro es muy cómodo para no estar repitiendo getters y setters. Repito: **ESTE MÉTODO NO LOS BORRA DEL ÍNDICE DE DOCUMENTOS, SOLO LOS DECREMENTA O INCREMENTA SI SE BORRAN O SE LEEN DE FICHERO INDEXADO.**

Hasta aquí hemos implementado las clases que tienen que ver con los términos y los documentos. A la práctica también se le ha añadido una parte de implementación sobre las preguntas y los términos de las preguntas. Las clases son las siguientes:

## Clase InformacionPregunta

Esta clase representa una query realizada por el usuario al indexador. Almacena la cantidad de palabras de la query, palabras de parada, palabras diferentes y un hash map almacenando el término de la query como clave y un InformacionTerminoPregunta (vamos a hablar de él ahora mismo).

```
class InformacionPregunta {
    friend ostream& operator<<(ostream& s, const InformacionPregunta& p);
public:
    InformacionPregunta (const InformacionPregunta &);
    InformacionPregunta ();
    ~InformacionPregunta ();
    InformacionPregunta & operator= (const InformacionPregunta &);

    // A@adir cuantos metodos se consideren necesarios para manejar la parte priva
    void addTermino(const string& termino, int pos, bool esStopword);
    int getNumTotalPal() const;
    int getNumTotalPalSinParada() const;
    int getNumTotalPalDiferentes() const;
    const unordered_map<string, InformacionTerminoPregunta>& getTerminos() const;
    void incrementarNPalabras(const int n);
    void incrementarNPalabrasSinParada(const int n);
    void incrementarNPalabrasDiferentes(const int n);
private:
    int numTotalPal;
    // N@ total de palabras en la pregunta
    int numTotalPalSinParada;
    // N@ total de palabras sin stop-words en la pregunta
    int numTotalPalDiferentes;
    // N@ total de palabras diferentes en la pregunta que no sean stop-words
    // (sin acumular la frecuencia de cada una de ellas)
    unordered_map<string, InformacionTerminoPregunta> terminos; // hash map string
};
```

De nuevo solo hacen falta los métodos getters y setters para controlar la parte privada. Nada relevante.

## Clase InformacionTerminoPregunta

Esta clase representa la información asociada a un término específico de la query de la pregunta del usuario. Como podemos observar en los atributos privados esta clase guarda la frecuencia del término en la pregunta del usuario y la posición en la que aparece en la pregunta.

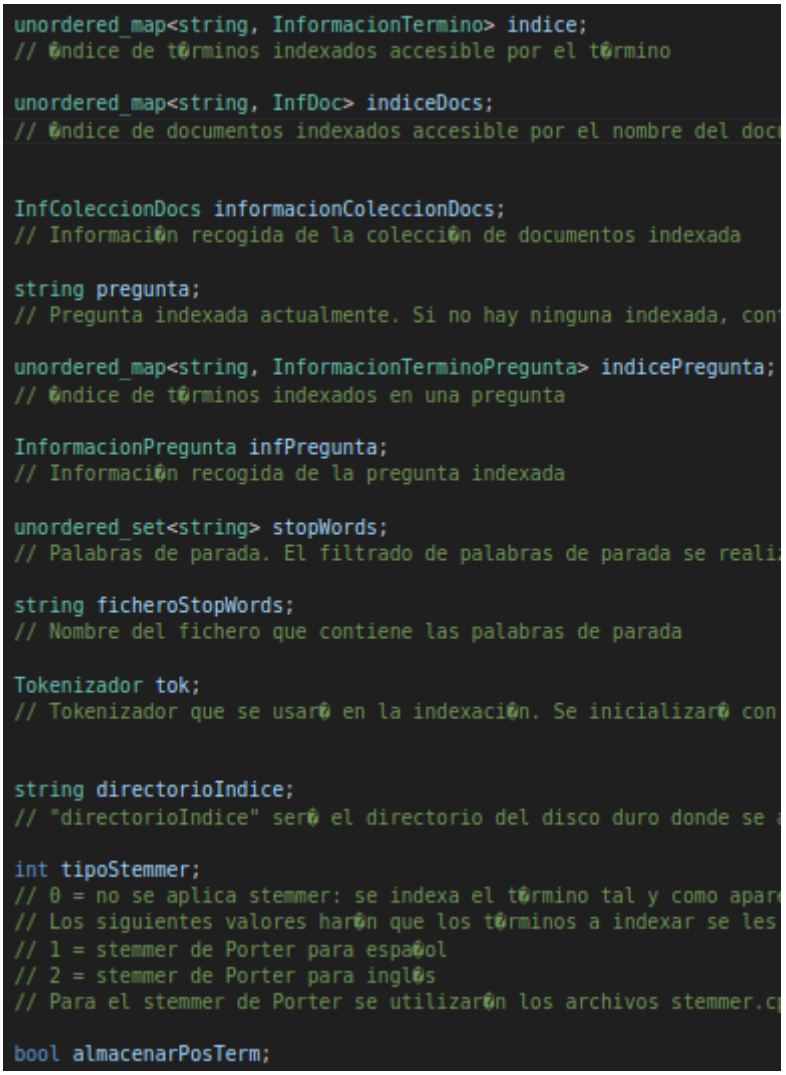
```
class InformacionTerminoPregunta {
    friend ostream& operator<<(ostream& s, const InformacionTerminoPregunta& p);
public:
    InformacionTerminoPregunta (const InformacionTerminoPregunta &);
    InformacionTerminoPregunta ();
    ~InformacionTerminoPregunta ();
    InformacionTerminoPregunta & operator= (const InformacionTerminoPregunta &);

    // A?adir cuantos m?todos se consideren necesarios para manejar la parte priv
    void addFT();
    void setFT(int n);
    void addItemToPos(int pos);
    int getFT() const;
    list<int> getSortedPositions() const;
private:
    int ft; // Frecuencia total del termino en la pregunta
    list<int> posTerm;
    // Solo se almacenara esta informaci?on si el campo privado del indexador alm
    // Lista de numeros de palabra en los que aparece el termino en la pregunta.
    //Los n?meros de palabra comenzar?n desde cero (la primera palabra de la preg
    //Se numerar?n las palabras de parada. Estar? ordenada de menor a mayor posic
};
```

Solo hay un método sobre el que destacar algo de importancia es getSortedPositions() ya que se pedía como pone en el comentario de abajo estará ordenado de menor a mayor por cada término.

Todas las clases anteriores son la base del indexador ya que vamos a hacer uso de la inmensa mayoría de ellas para implementar el indexador. La nueva cabecera que se nos proporcionó para implementar el indexador fue *indexadorHash.h*, este es el cerebro del indexador, no pongo imagen de la cabecera de esta clase porque es inmensa, tiene muchos métodos y atributos, solo voy a explicar los atributos y qué hace cada uno, los métodos posteriormente se irán viendo.

## Clase IndexadorHash

Esta clase tiene como función principal gestionar todo el proceso completo del análisis e incorporación de los documentos al índice mediante distintas tablas hash y el tokenizador previamente implementado. Esta clase será la encargada de coordinar la tokenización con la normalización + stemming de los términos incluidos en cada uno de los documentos indexados. Implementa métodos para indexar, borrar, guardar y recuperar indexaciones y documentos. Se puede considerar el núcleo central del indexador.



```
unordered_map<string, InformacionTermino> indice;
// Índice de términos indexados accesible por el término

unordered_map<string, InfDoc> indiceDocs;
// Índice de documentos indexados accesible por el nombre del doc


InfColeccionDocs informacionColeccionDocs;
// Información recogida de la colección de documentos indexada

string pregunta;
// Pregunta indexada actualmente. Si no hay ninguna indexada, con

unordered_map<string, InformacionTerminoPregunta> indicePregunta;
// Índice de términos indexados en una pregunta

InformacionPregunta infPregunta;
// Información recogida de la pregunta indexada

unordered_set<string> stopWords;
// Palabras de parada. El filtrado de palabras de parada se reali

string ficheroStopWords;
// Nombre del fichero que contiene las palabras de parada

Tokenizador tok;
// Tokenizador que se usará en la indexación. Se inicializará con


string directorioIndice;
// "directorioIndice" será el directorio del disco duro donde se

int tipoStemmer;
// 0 = no se aplica stemmer: se indexa el término tal y como apar
// Los siguientes valores harán que los términos a indexar se les
// 1 = stemmer de Porter para español
// 2 = stemmer de Porter para inglés
// Para el stemmer de Porter se utilizarán los archivos stemmer.c

bool almacenarPosTerm;
```

Estos son todos los campos privados que vamos a ir analizando en el constructor para ver qué hace cada uno de ellos. Este es el análisis de cada uno de ellos:

## Método IndexadorHash (constructor parametrizado):

Este constructor es el encargado de inicializar cada uno de los valores con los pasados por parámetro, muchos de ellos son directos: el nombre de fichero de las stopwords, los delimitadores, los casos especiales (no se usan en esta práctica), pasar a minúsculas sin acentos… Estos se pueden asignar con los métodos del tokenizador por lo que ahí ya tenemos la inicialización del tokenizador interno, el tipo de stemmer utilizado y el fichero además de almacenar las posiciones de los términos. Con el directorio donde almacenaremos la indexación como pone en los requerimientos pondremos si está a vacío el directorio actual y el vector de palabras de paradas tenemos que abrir el fichero para leer una a una las palabras de parada y añadirlas.

```
IndexadorHash::IndexadorHash(const string& fichStopWords, const string& delimitadores, const bool& detectComp,
    ficheroStopWords = fichStopWords;        // fichero que contiene las stopwords
    tok.DelimitadoresPalabra(delimitadores);    // campos del tokenizador
    tok.CasosEspeciales(detectComp);
    tok.PasarAminuscSinAcentos(minuscSinAcentos);
    tipoStemmer = tStemmer;             // tipo de stemmer (0, 1, 2)
    almacenarPosTerm = almPosTerm;
    if(dirIndice == ""){        // obtener el directorio actual si dirIndice == ""
        char ruta[PATH_MAX];
        if(getcwd(ruta, sizeof(ruta)) != nullptr){      // si puede obtener el directorio actual
            directorioIndice = string(ruta);
        }else{
            directorioIndice = ".";     // si no lo pondrá a "."
        }
    }else{
        directorioIndice = dirIndice;
    }
    // meter las stopswords en la variable stopWords list<string>
    ifstream ficherito(fichStopWords);
    if(ficherito.is_open()){
        string stopword;
        while(getline(ficherito, stopword)){
            stopWords.insert(stopword);     // metemos la stopword
        }
        ficherito.close();
    }else{
        cerr << "No se ha podido abrir el fichero de las stopWords\n";
    }
    // el resto de atributos se inicializan por defecto al ser tipos de datos no primitivos
}
```

## Método IndexadorHash (Directorio y Copia)

Hay otros dos métodos constructores, uno a partir de un directorio de indexación para recuperar esa indexación que hay en ese directorio, en este método directamente se llama al método *IndexadorHash::RecuperarIndexacion* y se procedería a leer toda la información indexada. El otro método constructor es el de copia, es sencillamente la copia 1 a 1 de los campos privados del indexador.

## Método Indexar(const string& ficheroDocumentos)

Este método se encarga de añadir documentos al índice. Básicamente se encarga de ir leyendo los documentos y así ir procesándolos. He dividido en funciones el código porque se me quedaba enorme. Finalmente después de todos los cambios actualizamos la cantidad de palabras diferentes que hay en el índice, es decir, términos indexados en el índice.

```
bool IndexadorHash::Indexar(const string& ficheroDocumentos){
    if (!ifstream(ficheroDocumentos)) {
        cerr << "No se puede abrir el fichero de documentos: " << ficheroDocumentos << '\n';
        return false;
    }

    ifstream listado(ficheroDocumentos);
    int nextID = 1;
    string rutaDocumento;
    stemmerPorter stem;

    while(getline(listado, rutaDocumento)){
        procesarDocumento(rutaDocumento, nextID, stem);
        ++nextID;
    }
    informacionColeccionDocs.setPalabrasDiferentes(indice.size());      // actualizamos el
    return true;
}
```

Esta función se encarga de indexar un documento en el índice de documentos. Aquí se realiza la lectura del documento y su posterior indexación además de la extracción de los tokens contenidos en el documento además del uso del stemmer en caso de que esté activo. Este proceso se complementa con la función procesarTokensDeDocumento.

```
void IndexadorHash::procesarDocumento(const string& rutaDocumento, int& idActual, stemmerPorter& stem) {
    struct stat infoArchivo{};
    if(stat(rutaDocumento.c_str(), &infoArchivo) != 0){     // comprobamos que el fichero existe
        cerr << "ERROR al acceder a " << rutaDocumento << ": " << strerror(errno) << '\n';
        return;
    }

    auto itIndexado = indiceDocs.find(rutaDocumento);   // si el fichero existe comprobamos
    bool yaIndexado = itIndexado != indiceDocs.end();
    if(yaIndexado && infoArchivo.st_mtim.tv_sec <= itIndexado->second.getFechaModificacion()){   // si el
        idActual++;     // saltamos el documento (indexacion innecesaria)
        return;
    }

    tok.Tokenizar(rutaDocumento);   // tokenizamos el fichero si ha sido modificado para indexarlo
    ifstream archivoTokens(rutaDocumento + ".tk");
    if(!archivoTokens){     // si no se ha podido abrir el archivo de tokens lo saltamos
        idActual++;
        return;
    }

    if(yaIndexado){             // si ya estaba indexado borramos el documento del indice
        idActual = itIndexado->second.getIdDoc();  // reutilizamos el ID
        BorraDoc(rutaDocumento);                   // lo eliminamos antes de indexar
    }

    InfDoc infoDoc{};                   // creamos el documento a indexar
    infoDoc.setIdDoc(idActual);         // con idActual y el tamaño en bytes
    infoDoc.setTamBytes(infoArchivo.st_size);
    procesarTokensDeDocumento(archivoTokens, idActual, infoDoc, stem);  // leemos los tokens, aplicamos s

    indiceDocs[rutaDocumento] = infoDoc;                            // agregamos el documento a la colecc
    informacionColeccionDocs.agregarDocumentoAColeccion(infoDoc);
}
```

Esta función se encarga de la indexación de cada uno de los términos que hemos obtenido. Recorre cada uno de los tokens y actualiza el índice. Aplica stemming si así se indica y se aplica un pequeño filtrado para no meter palabras que sean stopwords ni palabras de 1 sola letra. En caso de que ese token ya estuviera en el índice se incrementan sus valores estadísticos y si no lo indexa, para ello se hace uso de un indice temporal para hacer posteriormente la copia.

```
void IndexadorHash::procesarTokensDeDocumento(istream& archivoTokens, int idDoc, InfDoc& infoDoc, stemmerPorter& stem) {
    unordered_map<string, InformacionTermino> indice_sub;
    istream_iterator<string> it(archivoTokens);
    istream_iterator<string> end;

    int pos = 0;
    for(; it != end; ++it, ++pos){
        string termino = *it;
        infoDoc.addNumPal();    // metemos 1 palabra

        if(tipoStemmer != 0){ // aplicamos stemming si está activado
            stem.stemmer(termino, tipoStemmer);
        }

        if(EsStopWord(stopWords, termino) || termino.length() <= 1) continue;     // filtrar las palabras no validas par

        // añadir al índice principal si ya existía
        if(indice[termino].getFT() > 0){
            infoDoc.addNumPalSinParada();
            if (indice[termino].agregarPosicionADocumento(idDoc, pos, almacenarPosTerm)) {
                infoDoc.addNumPalDiferentes();
            }
        }
        else {
            if (indice_sub[termino].getFT() == 0) {
                indice_sub[termino].agregarPosicionADocumento(idDoc, pos, almacenarPosTerm);
            }
            else {
                indice_sub.erase(termino);  // moverlo al índice principal
                infoDoc.addNumPalSinParada();
                if (indice[termino].agregarPosicionADocumento(idDoc, pos, almacenarPosTerm)) {
                    infoDoc.addNumPalDiferentes();
                }
            }
        }
    }

    indice_sub.clear();
}
```

## Método IndexarDirectorio

Este método se encarga de indexar todos los ficheros de un directorio en concreto pero hace uso del método anterior de Indexar, obtiene todos los ficheros de forma recursiva del directorio y posteriormente los indexa.

```
bool IndexadorHash::IndexarDirectorio(const string &dirAIndexar){
    struct stat dir;
    int err = stat(dirAIndexar.c_str(), &dir);

    if (err == -1 || !S_ISDIR(dir.st_mode)){
        return false;
    }else{
        string cmd = "find " + dirAIndexar + " -follow -type f | sort > .lista_ficheros_indexador";
        system(cmd.c_str());

        return Indexar(".lista_ficheros_indexador");
    }
}
```

## Método GuardarIndexacion() y RecuperarIndexacion()

Este método es el encargado de guardar el contenido del indexador en un fichero binario en la ruta que se le especificó en el constructor. He separado en funciones todo el tema de Guardar la indexación y Recuperar, adjunto las imágenes de cómo lo he guardado porque no hay mucho más que comentar de la forma en la que he almacenado toda esta información.

```
bool IndexadorHash::GuardarIndexacion() const{
    bool correcto = true;
    string fileIndexacion = directorioIndice + "/indexacion-amsm30.ua";
    struct stat info{};
    bool directorioExiste = (stat(directorioIndice.c_str(), &info) == 0) && S_ISDIR(info.st_mode);

    if(directorioExiste == false){
        int resultado = mkdir(directorioIndice.c_str(), 0777);  // resultado != 0 no se ha creado el directorio
        if(resultado != 0){
            std::cerr << "No se pudo crear el directorio \"" << directorioIndice << "\": " << strerror(errno) << '\n';  // error de creacion
            correcto = false;
        }
    }
    // si todo ha ido bien y se ha creado el directorio de indexacion
    ofstream archivo(fileIndexacion, ios::binary);  // binario para guardar la indexacion
    if(archivo.is_open() == false){
        correcto = false;
        cerr << "No se pudo abrir el archivo" << '\n';
    }
    // guardamos 1 a 1 los campos privados
    archivo << ficheroStopWords << '\n';
    archivo << tok.DelimitadoresPalabra() << '\n';
    archivo << tok.CasosEspeciales() << '\n';
    archivo << tok.PasarAminuscSinAcentos() << '\n';
    archivo << directorioIndice << '\n';
    archivo << tipoStemmer << '\n';
    archivo << almacenarPosTerm << '\n';

    guardarTerminosIndexadosIndice(archivo, indice);
    for(const auto& doc : indiceDocs)
        guardarInformacionDocumento(archivo, doc);
    guardarInformacionColeccionDocs(archivo, informacionColeccionDocs);
    guardarInformacionPregunta(archivo, infPregunta);
    for(const auto& terminoPregunta : indicePregunta)
        guardarTerminoPregunta(archivo, terminoPregunta);

    archivo.close();
    return correcto;
}
```

Y este método es el encargado de hacer precisamente lo contrario, leer de ese fichero binario cada uno de los campos y asignarlo a cada una de los atributos privados del indexador.

```
bool IndexadorHash::RecuperarIndexacion(const string &directorioIndexacion){
    bool correcto = true;
    ifstream archivo(directorioIndexacion + "/indexacion-amsm30.ua", ios::binary);
    string delimitadores;
    bool casosEspeciales, pasarMinusculas;

    indice.clear();     // borramos los posibles datos en estos campos
    indiceDocs.clear();
    indicePregunta.clear();

    if(archivo.is_open() == false){
        cerr << "No se pudo abrir el archivo de indexación." << '\n';   // no se abre el fichero que contiene la indexacion
        correcto = false;
    }else{
        getline(archivo, ficheroStopWords); // leemos todos los campos del fichero de indexacion
        getline(archivo, delimitadores);    // está literalmente arriba como leerlo
        archivo >> casosEspeciales;
        archivo >> pasarMinusculas;
        archivo.ignore();                   // salto de linea
        getline(archivo, directorioIndice);
        archivo >> tipoStemmer;
        archivo >> almacenarPosTerm;

        ifstream stopwordsFile(ficheroStopWords);   // abrimos el fichero de stopwords
        if(stopwordsFile.is_open() == true){
            string palabra;
            while(stopwordsFile >> palabra){
                stopWords.insert(palabra);          // metemos palabra a palabra en stopWords
            }
        }else{
            cerr << "No se pudo abrir el archivo de stopwords." << '\n';
            correcto = false;
        }
        tok = Tokenizador(delimitadores, casosEspeciales, pasarMinusculas);     // casi se me olvida xd
        // recuperar informacion de los documentos y terminos
        leerTerminosIndexadosIndice(archivo, indice);   // leemos cada uno de los campos privados (clases externas)
        int cantidadDocumentos = informacionColeccionDocs.getNumDocs();
        leerInformacionDocumentos(archivo, indiceDocs, cantidadDocumentos);     // NO SE PUEDE SABER DESDE INDICEDOCS LA CANTIDAD DE DOCUM
        leerInformacionColeccionDocs(archivo, informacionColeccionDocs);
        // recuperar informacion de la pregunta
        leerInformacionPregunta(archivo, infPregunta);
        leerTerminoPregunta(archivo, indicePregunta);
        archivo.close();
    }

    return correcto;
}
```

Aquí como observamos se encuentra la creación del directorio especificado y el fichero de indexación llamado *directorioIndice + /indexacion-amsm30.ua*. La forma de escribir y leer este fichero es la siguiente:

```cpp
void guardarTerminosIndexadosIndice(ofstream& salida, const unordered_map<string, InformacionTermino>& indice){
    salida << indice.size() << ' ';
    for(const auto& [termino, infoTermino] : indice){
        salida << termino << ' ' << infoTermino.getFT() << ' ';     // frecuencia del termino en TODA LA COLECCI

        const auto& apariciones = infoTermino.getLDocs();           // documentos en los que aparece el termino
        salida << apariciones.size() << ' ';

        for(const auto& [idDoc, datosDoc] : apariciones){
            salida << idDoc << ' ' << datosDoc.getFT() << ' ';  // frecuencia del termino en el documento idDoc

            const auto& posiciones = datosDoc.getPositionTerm();    // donde aparece el termino en el documento
            salida << posiciones.size() << ' ';
            for(int pos : posiciones){
                salida << pos << ' ';
            }
        }
    }
    salida << '\n';
}

void leerTerminosIndexadosIndice(ifstream& archivo, unordered_map<string, InformacionTermino>& indice){
    int numTerminos;
    archivo >> numTerminos;

    for(int i = 0; i < numTerminos; ++i){
        string termino;
        int ftTotal, numDocs;
        archivo >> termino >> ftTotal >> numDocs;

        for(int j = 0; j < numDocs; ++j){
            int idDoc, ftDoc, numPos;
            archivo >> idDoc >> ftDoc >> numPos;

            for(int k = 0; k < numPos; ++k){
                int pos;
                archivo >> pos;
                indice[termino].agregarPosicionADocumento(idDoc, pos, true);
            }
        }
    }
}
```

```cpp
void guardarInformacionDocumento(ofstream& salida, const pair<const string, InfDoc>& entradaDoc){
    const string& nombreDocumento = entradaDoc.first;
    const InfDoc& info = entradaDoc.second;

    salida << nombreDocumento << ' '
           << info.getIdDoc() << ' '
           << info.getNumPal() << ' '
           << info.getNumPalSinParada() << ' '
           << info.getNumPalDiferentes() << ' '
           << info.getTamBytes() << ' '
           << info.getFechaModificacion() << '\n';
}

void leerInformacionDocumentos(ifstream& entrada, unordered_map<string, InfDoc>& indiceDocs, int cantidadDocs){
    string nombreDocumento;
    int idDocumento, totalPalabras, palabrasSinParada, palabrasDiferentes, bytes = 0;
    time_t fecha = 0;
    for(int i = 0; i < cantidadDocs; i++){
        entrada >> nombreDocumento >> idDocumento >> totalPalabras >> palabrasSinParada >> palabrasDiferentes >> bytes >> fecha;
        InfDoc informacionDocumento{};
        informacionDocumento.setIdDoc(idDocumento);
        informacionDocumento.setNumPal(totalPalabras);
        informacionDocumento.setNumPalSinParada(palabrasSinParada);
        informacionDocumento.setNumPalDifernetes(palabrasDiferentes);
        informacionDocumento.setTamBytes(bytes);
        informacionDocumento.setFechaModificacion(fecha);
        indiceDocs.emplace(nombreDocumento, informacionDocumento);  // insertamos directamente en el hashmap <nombreDoc, infoDocu
    }

}
```

```cpp
void guardarInformacionColeccionDocs(ofstream& salida, const InfColeccionDocs& coleccion){
    int nDocs = coleccion.getNumDocs();
    int totalPal = coleccion.getNumTotalPal();
    int sinStop = coleccion.getNumTotalPalSinParada();
    int diferentes = coleccion.getNumTotalPalDiferentes();
    int tam = coleccion.getTamBytes();

    salida << nDocs << ' ' << totalPal << ' ' << sinStop << ' '<< diferentes << ' ' << tam << '\n';
}

void leerInformacionColeccionDocs(ifstream& entrada, InfColeccionDocs& coleccionDocumentos){
    int cantidadDocumentos, totalPal, palabrasSinStop, palabrasDiferentes, tamBytes = 0;
    entrada >> cantidadDocumentos >> totalPal >> palabrasSinStop >> palabrasDiferentes >> tamBytes;     // leemos los campos de la coleccion
    coleccionDocumentos.setNumDocs(coleccionDocumentos.getNumDocs() + cantidadDocumentos);              // añadimos los campos leidos a cada uno de
    coleccionDocumentos.setTotalPalabras(coleccionDocumentos.getNumTotalPal() + totalPal);              // existente + leido
    coleccionDocumentos.setPalabrasSinParada(coleccionDocumentos.getNumTotalPalSinParada() + palabrasSinStop);
    coleccionDocumentos.setPalabrasDiferentes(coleccionDocumentos.getNumTotalPalDiferentes() + palabrasDiferentes);
    coleccionDocumentos.setBytes(coleccionDocumentos.getTamBytes() + tamBytes);
}
```

```cpp
void guardarInformacionPregunta(ofstream& salida, const InformacionPregunta& pregunta){
    salida << pregunta.getNumTotalPal() << ' ' << pregunta.getNumTotalPalSinParada() << ' ' << pregunta.getNumTotalPalDiferentes() << '\n';
}

void leerInformacionPregunta(ifstream& entrada, InformacionPregunta& pregunta){
    int totalPalabras, palabrasSinParada, palabrasDiferentes = 0;

    entrada >> totalPalabras >> palabrasSinParada >> palabrasDiferentes;
    pregunta.incrementarNPalabras(totalPalabras);
    pregunta.incrementarNPalabrasSinParada(palabrasSinParada);
    pregunta.incrementarNPalabrasDiferentes(palabrasDiferentes);

}
```

```cpp
void guardarTerminoPregunta(ofstream& salida, const pair<const string, InformacionTerminoPregunta>& entrada){
    const string& termino = entrada.first;
    const InformacionTerminoPregunta& info = entrada.second;

    salida << termino << ' ' << info.getFT() << ' ';

    const auto& posiciones = info.getSortedPositions();
    for(int pos : posiciones){
        salida << pos << ' ';
    }

    salida << '\n';
}

void leerTerminoPregunta(ifstream& entrada, unordered_map<string, InformacionTerminoPregunta>& indicePregunta) {
    string termino;
    int ft;
    while(entrada >> termino >> ft){
        InformacionTerminoPregunta info;
        for (int i = 0; i < ft; ++i) {
            int pos;
            if(!(entrada >> pos)) break;  // por si el archivo termina mal
            info.addItemToPos(pos);
            info.addFT();
        }
        indicePregunta.emplace(termino, info);
    }
}
```

Aquí se muestra cómo se almacena y lee en este fichero de indexación todos y cada uno de los campos almacenados en el índice como los términos, los documentos, la colección, los términos de la pregunta y los términos de la pregunta. Esta forma ha sido la más sencilla a mi modo de ver y para luego leer también la más sencilla, he hecho una prueba con uno de los tests que se nos ha proporcionado y se genera este fichero de indexación.

indicePrueba > indexacion-amsm30.ua

```
./StopWordsEspanyol.txt
. ,:
0
0
./indicePrueba
0
1
4 pal3 1 1 1 1 1 4 pal2 3 2 2 2 2 0 2 1 1 1 2 pal4 1 1 2 1 1 3 pal1 2 1 1 2 2 0 3
corpus_corto/fichero2.txt 2 5 3 2 23 1744548792
corpus_corto/fichero1.txt 1 6 4 3 30 1744548792
2 11 7 4 53
0 0 0
```

En este se puede ver cómo están distribuidos cada uno de los atributos del indexador.

## Método IndexarPregunta()

Este método es el encargado de procesar la query introducida por el usuario, éste se encarga de tokenizarla y aplicarle el mismo proceso que se le aplicó a los términos en los documentos, es decir, stemming en caso de estar indicado y filtrado de stopwords. Los términos de la pregunta se almacenan en un hash map con clave el término a indexar y como valor una estructura de InformacionPregunta que vimos anteriormente que incluye la frecuencia del término en la pregunta y las posiciones donde aparece.

```
bool IndexadorHash::IndexarPregunta(const string &preg){
    indicePregunta.clear();
    infPregunta = InformacionPregunta();
    stemmerPorter stem;

    list<string> lista_tokens;
    tok.Tokenizar(preg, lista_tokens);    // tokenizamos la pregunta
    infPregunta.incrementarNPalabras(lista_tokens.size());    // actualizamos el total

    if(lista_tokens.empty()){
        return false;
    }

    // para cada token obtenido le aplicamos stem y lo indexamos si no es stopword
    list<string>::iterator it = lista_tokens.begin();
    int pos = 0;

    while(it != lista_tokens.end()){
        string token = *it;
        if(tipoStemmer != 0){
            stem.stemmer(token, tipoStemmer);
        }
        if(EsStopWord(stopWords, token) == false && token.length() > 1){
            indicePregunta[token].addFT();
            infPregunta.incrementarNPalabrasSinParada(1);
            if(almacenarPosTerm){
                indicePregunta[token].addItemToPos(pos);
            }
        }
        ++it;
        ++pos;
    }

    infPregunta.incrementarNPalabrasDiferentes(indicePregunta.size());  // incrementam
    pregunta = preg;

    return true;
}
```

El resto de métodos implementados no eran de gran complejidad como los DevuelvePregunta, Devuelve, Existe y getters. Eso sí encontré un problema con el método Devuelve(const string& word, const string& nomDoc, InfTermDoc& i) ya que no conseguí que me devolviese en el InfTermDoc asignado a un término de un documento, probé a hacer varias pruebas pero no logré conseguirlo y ya iba demasiado justo de tiempo. Métodos de ListarTerminos, ListarDocs, etc no fueron demasiado complicados de implementar y cuyo nombre indica solo acceden a los atributos privados de la clase.

## Justificación de la solución elegida de la indexación

La justificación de por qué he elegido implementar la indexación con tablas hash ha sido simplemente por comodidad y sencillez (aunque luego implementarlo tiene sus cositas también) básicamente porque era el único que tenía diagrama de cómo funcionaba visualmente de manera explicativa, pensé para poder añadirlo en la memoria implementar trie o al menos intentarlo pero es demasiado tarde ya para ello y requeriría un coste adicional de tiempo y conocimientos que no tengo aún. El indexador me ha dado bastantes problemas sobre todo a la hora de implementar el método Indexar ya que toca todas las partes de la parte privada y era bastante lioso de tenerlo claro qué parte hace cada cosa.

## Análisis de las mejoras realizadas en la práctica

Una de las mejoras que he realizado en la práctica es la indexación de términos EXCLUSIVAMENTE relevantes, es decir, he asumido que por defecto vamos a tener un corpus grande de documentos por lo que para ahorrar espacio en memoria solamente se indexarán los términos que tengan en su frecuencia de término un valor superior a 1, esto se comentó en clase de teoría ya que no es necesario indexar un término que sabemos que si aparece una sola vez en la colección. Esto nos proporcionará un ligero decremento en el espacio consumido ya que se indexarán menos términos.

## Análisis de eficiencia computacional

Para ello vamos a calcular el análisis del coste computacional teórico del método Indexar ya que es el cerebro del indexador. Para ello vamos a dividirlo en partes:

- D: números de documentos a indexar
- T: promedio de tokens por documento
- V: tamaño del índice
- P: promedio de posiciones por término en cada documento

Esto se divide en varias fases:

- Fase de lectura de documentos: Cada documento se lee una vez por lo que $O(D \cdot T)$
- Tokenización de documentos: Tokenizar carácter a carácter de cada documento $O(T)$
- Filtrado de las stopwords: Se verifica si está en el conjunto de palabras de parada que está implementado como unordered_set por lo que $O(\log S)$
- Stemming: Casi prácticamente constante ya que las palabras son pequeñas a normalizar $O(1)$
- Actualizar el índice: Introducir en los índices de términos y documentos los términos y documentos indexados, si ya existe $O(1)$, si no existe $O(P)$

Por lo que en total obtenemos una complejidad **$O(D \cdot T \cdot P)$**.