Explotación de la Información.
Curso 2017-2018

Grado en Ingeniería Informática - Universidad de Alicante

Práctica 2: Indexador

Contenido

Fecha entrega 3

Qué se pide 3

Información a obtener en la indexación 3

- Clase “InformacionTermino” 3
- Clase “InfTermDoc” 4
- Clase “InfDoc” 4
- Clase “InfColeccionDocs” 5
- Clase “InformacionTerminoPregunta” 5
- Clase “InformacionPregunta” 6

Prototipo de la clase “IndexadorHash” 6

- public: 7
- private: 10

Aclaraciones 12

- Evaluación de la práctica 12
- Uso de los mapas 12
- Ejemplo de fichero main.cpp 13
- Control de tiempos de ejecución 14
- Cálculo de eficiencia espacial 14
- Fichero makefile 15

Aclaraciones comunes a todas las prácticas 15

- STL 15
- VALGRIND 15
- Operaciones especificadas 15
- Tratamiento de excepciones 16
- Forma de entrega 16
- Ficheros a entregar y formato de entrega 16
- Evaluación 17

Fecha entrega

Del 23 al 27 de abril de 2018

Qué se pide

Se pide construir la clase *IndexadorHash* que a partir de una colección de documentos genera las estructuras que se utilizarán posteriormente en la fase de búsqueda. Para ello se utilizarán tablas Hash (*unordered_map* y *unordered_set*).

Para ello, se utilizarán las librerías STL y la clase *Tokenizador* desarrollada por el alumno en la práctica 1.

Se valorará la eficiencia de la implementación, especialmente cualquier modificación en la representación interna (parte privada de las clases) o el algoritmo utilizado.

Información a obtener en la indexación

Se deberá generar toda la información requerida en la indexación tal y como se ha explicado en clases de teoría mediante la clase *InformacionTermino* y las que se muestran a continuación. El alumno podrá modificar las estructuras de datos con el objetivo de optimizar su eficiencia, pudiendo cambiar los tipos de los datos y añadiendo cuanta información considere oportuna (NO se podrá cambiar el formato de salida que se describe en cada *operator<<*). Asimismo, también se podrá modificar el carácter const de los métodos/argumentos aquí descritos siempre que se mantengan las características de funcionamiento previstas. Todas estas clases aparecerán en los ficheros *indexadorInformacion.h* e *indexadorInformacion.cpp*.

Clase "InformacionTermino"

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

   // Añadir cuantos métodos se consideren necesarios para manejar la parte
   privada de la clase
private:
   int ftc;  // Frecuencia total del término en la colección
   unordered_map<long int, InfTermDoc> l_docs;
       // Tabla Hash que se accederá por el id del documento, devolviendo un
       objeto de la clase InfTermDoc que contiene toda la información de
       aparición del término en el documento
};
ostream& operator<<(ostream& s, const InformacionTermino& p) {
   s << "Frecuencia total: " << p.ftc << "\tfd: " << p.l_docs.size();
   // A continuación se mostrarían todos los elementos de p.l_docs: s <<
   "\tId.Doc: " << idDoc << "\t" << InfTermDoc;

   return s;
}
```

Clase "InfTermDoc"

```
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 &);

   // Añadir cuantos métodos se consideren necesarios para manejar la parte
   privada de la clase
private:
   int ft;   // Frecuencia del término en el documento
   list<int> posTerm;
       // Solo se almacenará esta información si el campo privado del indexador
       almacenarPosTerm == true
       // Lista de números de palabra en los que aparece el término en el
       documento. Los números de palabra comenzarán desde cero (la primera
       palabra del documento). Se numerarán las palabras de parada. Estará
       ordenada de menor a mayor posición.
};
ostream& operator<<(ostream& s, const InfTermDoc& p) {
   s << "ft: " << p.ft;
   // A continuación se mostrarían todos los elementos de p.posTerm ("posicion
   TAB posicion TAB ... posicion, es decir nunca finalizará en un TAB"): s <<
   "\t" << posicion;

   return s;
}
```

Clase "InfDoc"

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

   // Añadir cuantos métodos se consideren necesarios para manejar la parte
   privada de la clase
private:
   long int idDoc;
       // Identificador del documento. El primer documento indexado en la
       colección será el identificador 1
   int numPal;      // Nº total de palabras del documento
   int numPalSinParada;   // Nº total de palabras sin stop-words del documento
   int numPalDiferentes;
       // Nº total de palabras diferentes que no sean stop-words (sin acumular
       la frecuencia de cada una de ellas)
   int tamBytes;    // Tamaño en bytes del documento
   Fecha fechaModificacion;
       // Atributo correspondiente a la fecha y hora de modificación del
       documento. El tipo "Fecha/hora" lo elegirá/implementará el alumno
};
ostream& operator<<(ostream& s, const InfDoc& p) {
   s   <<   "idDoc:   "   <<   p.idDoc   <<   "\tnumPal:   "   <<   p.numPal   <<
   "\tnumPalSinParada: " << p.numPalSinParada << "\tnumPalDiferentes: " <<
   p.numPalDiferentes << "\ttamBytes: " << p.tamBytes;

   return s;
}
```

Clase "InfColeccionDocs"

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

   // Añadir cuantos métodos se consideren necesarios para manejar la parte
   privada de la clase
private:
   long int numDocs;       // Nº total de documentos en la colección
   long int numTotalPal;
       // Nº total de palabras en la colección
   long int numTotalPalSinParada;
       // Nº total de palabras sin stop-words en la colección
   long int numTotalPalDiferentes;
       // Nº total de palabras diferentes en la colección que no sean stop-
       words (sin acumular la frecuencia de cada una de ellas)
   long int tamBytes;      // Tamaño total en bytes de la colección
};
ostream& operator<<(ostream&, const InfColeccionDocs&);
   s << "numDocs: " << p.numDocs << "\tnumTotalPal: " << p.numTotalPal <<
   "\tnumTotalPalSinParada:      "      <<      p.numTotalPalSinParada      <<
   "\tnumTotalPalDiferentes: " << numTotalPalDiferentes << "\ttamBytes: " <<
   p.tamBytes;

   return s;
}
```

Clase "InformacionTerminoPregunta"

A continuación se muestra la información requerida para la indexación de preguntas realizadas por un usuario, es decir, diferente de la indexación de un documento de una colección que requerirá más información. Toda esta información se almacenará en memoria principal.

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

   // Añadir cuantos métodos se consideren necesarios para manejar la parte
   privada de la clase
private:
   int ft;   // Frecuencia total del término en la pregunta
   list<int> posTerm;
       // Solo se almacenará esta información si el campo privado del indexador
       almacenarPosTerm == true
       // Lista de números de palabra en los que aparece el término en la
       pregunta. Los números de palabra comenzarán desde cero (la primera
       palabra de la pregunta). Se numerarán las palabras de parada. Estará
       ordenada de menor a mayor posición.
};
ostream& operator<<(ostream& s, const InformacionTerminoPregunta& p) {
   s << "ft: " << p.ft;
   // A continuación se mostrarían todos los elementos de p.posTerm ("posicion
   TAB posicion TAB ... posicion, es decir nunca finalizará en un TAB"): s <<
   "\t" << posicion;
```

```
   return s;
}
```

Clase "InformacionPregunta"

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

   // Añadir cuantos métodos se consideren necesarios para manejar la parte
   privada de la clase
private:
   long int numTotalPal;
       // Nº total de palabras en la pregunta
   long int numTotalPalSinParada;
       // Nº total de palabras sin stop-words en la pregunta
   long int numTotalPalDiferentes;
       // Nº total de palabras diferentes en la pregunta que no sean stop-words
       (sin acumular la frecuencia de cada una de ellas)
};
ostream& operator<<(ostream&, const InformacionPregunta&);
   s << "numTotalPal: " << p.numTotalPal << "\tnumTotalPalSinParada: "<<
   "\tnumTotalPalDiferentes: " << numTotalPalDiferentes;

   return s;
}
```

Prototipo de la clase "IndexadorHash"

A continuación se muestra el prototipo de la clase *IndexadorHash*. La parte privada de la clase se podrá modificar a decisión del alumno para mejorar al máximo su eficiencia. La parte pública se mantendrá en sus prototipos, también permitiéndose añadir nuevos enriquecimientos siempre que no hagan referencia a la parte privada de la clase. Esta clase aparecerá en los ficheros *indexadorHash.h* e *indexadorHash.cpp*. Destacar que para la evaluación de la práctica, el tokenizador se usará sin la opción de quitar mayúsculas ni tratar con casos especiales.

```
class IndexadorHash {

friend ostream& operator<<(ostream& s, const IndexadorHash& p) {
       s << "Fichero con el listado de palabras de parada: " << p.
       ficheroStopWords << endl;
       s << "Tokenizador: " << p.tok << endl;
       s << "Directorio donde se almacenara el indice generado: " <<
       p.directorioIndice << endl;
       s << "Stemmer utilizado: " << p.tipoStemmer << endl;
       s << "Informacion de la coleccion indexada: " <<
       p.informacionColeccionDocs << endl;
       s << "Se almacenara parte del indice en disco duro: " <<
       p.almacenarEnDisco << endl;
       s << "Se almacenaran las posiciones de los terminos: " <<
       p.almacenarPosTerm;

       return s;
   }
```

public:

```
IndexadorHash(const string& fichStopWords, const string& delimitadores,
const bool& detectComp, const bool& minuscSinAcentos, const string&
dirIndice, const int& tStemmer, const bool& almEnDisco, const bool&
almPosTerm);
    // "fichStopWords" será el nombre del archivo que contendrá todas las
    palabras de parada (una palabra por cada línea del fichero) y se
    almacenará en el campo privado "ficheroStopWords". Asimismo, almacenará
    todas las palabras de parada que contenga el archivo en el campo privado
    "stopWords", el índice de palabras de parada.
    // "delimitadores" será el string que contiene todos los delimitadores
    utilizados por el tokenizador (campo privado "tok")
    // detectComp y minuscSinAcentos serán los parámetros que se pasarán al
    tokenizador
    // "dirIndice" será el directorio del disco duro donde se almacenará el
    índice (campo privado "directorioIndice"). Si dirIndice="" entonces se
    almacenará en el directorio donde se ejecute el programa
    // "tStemmer" inicializará la variable privada "tipoStemmer":
          // 0 = no se aplica stemmer: se indexa el término tal y como
          aparece tokenizado
          // 1 = stemmer de Porter para español
          // 2 = stemmer de Porter para inglés
    // "almEnDisco" inicializará la variable privada "almacenarEnDisco"
    // "almPosTerm" inicializará la variable privada "almacenarPosTerm"
    // Los índices (p.ej. índice, indiceDocs e informacionColeccionDocs)
    quedarán vacíos

IndexadorHash(const string& directorioIndexacion);
    // Constructor para inicializar IndexadorHash a partir de una indexación
    previamente realizada que habrá sido almacenada en
    "directorioIndexacion" mediante el método "bool GuardarIndexacion()".
    Con ello toda la parte privada se inicializará convenientemente, igual
    que si se acabase de indexar la colección de documentos. En caso que no
    exista el directorio o que no contenga los datos de la indexación se
    tratará la excepción correspondiente

IndexadorHash(const IndexadorHash&);

~IndexadorHash();

IndexadorHash& operator= (const IndexadorHash&);

bool Indexar(const string& ficheroDocumentos);
    // Devuelve true si consigue crear el índice para la colección de
    documentos detallada en ficheroDocumentos, el cual contendrá un nombre
    de documento por línea. Los añadirá a los ya existentes anteriormente en
    el índice.
    // Devuelve falso si no finaliza la indexación (p.ej. por falta de
    memoria), mostrando el mensaje de error correspondiente, indicando el
    documento y término en el que se ha quedado.
    // En el caso que aparezcan documentos repetidos o que ya estuviesen
    previamente indexados (ha de coincidir el nombre del documento y el
    directorio en que se encuentre), se mostrará el mensaje de excepción
    correspondiente, y se re-indexarán (borrar el documento previamente
    indexado e indexar el nuevo) en caso que la fecha de modificación del
    documento sea más reciente que la almacenada previamente (class "InfDoc"
    campo "fechaModificacion"). Los casos de reindexación mantendrán el
    mismo idDoc.


bool IndexarDirectorio(const string& dirAIndexar);
    // Devuelve true si consigue crear el índice para la colección de
    documentos que se encuentra en el directorio (y subdirectorios que
    contenga) dirAIndexar (independientemente de la extensión de los
    mismos). Se considerará que todos los documentos del directorio serán
```

ficheros de texto. Los añadirá a los ya existentes anteriormente en el índice.
// Devuelve falso si no finaliza la indexación (p.ej. por falta de memoria o porque no exista "dirAIndexar"), mostrando el mensaje de error correspondiente, indicando el documento y término en el que se ha quedado.
// En el caso que aparezcan documentos repetidos o que ya estuviesen previamente indexados (ha de coincidir el nombre del documento y el directorio en que se encuentre), se mostrará el mensaje de excepción correspondiente, y se re-indexarán (borrar el documento previamente indexado e indexar el nuevo) en caso que la fecha de modificación del documento sea más reciente que la almacenada previamente (class "InfDoc" campo "fechaModificacion"). Los casos de reindexación mantendrán el mismo idDoc.

```
bool GuardarIndexacion() const;
```
// Se guardará en disco duro (directorio contenido en la variable privada "directorioIndice") la indexación actualmente en memoria (incluidos **todos los parámetros de la parte privada**). La forma de almacenamiento la determinará el alumno. El objetivo es que esta indexación se pueda recuperar posteriormente mediante el constructor "IndexadorHash(const string& directorioIndexacion)". Por ejemplo, supongamos que se ejecuta esta secuencia de comandos: "IndexadorHash a("./fichStopWords.txt", "[,.", "./dirIndexPrueba", 0, false); a.Indexar("./fichConDocsAIndexar.txt"); a.GuardarIndexacion();", entonces mediante el comando: "IndexadorHash b("./dirIndexPrueba");" se recuperará la indexación realizada en la secuencia anterior, cargándola en "b"
// Devuelve falso si no finaliza la operación (p.ej. por falta de memoria, o el nombre del directorio contenido en "directorioIndice" no es correcto), mostrando el mensaje de error correspondiente
// En caso que no existiese el directorio directorioIndice, habría que crearlo previamente

```
bool RecuperarIndexacion (const string& directorioIndexacion);
```
// Vacía la indexación que tuviese en ese momento e inicializa IndexadorHash a partir de una indexación previamente realizada que habrá sido almacenada en "directorioIndexacion" mediante el método "bool GuardarIndexacion()". Con ello **toda la parte privada** se inicializará convenientemente, igual que si se acabase de indexar la colección de documentos. En caso que no exista el directorio o que no contenga los datos de la indexación se tratará la excepción correspondiente, y se devolverá *false*

```
void ImprimirIndexacion() const {
    cout << "Terminos indexados: " << endl;
          // A continuación aparecerá un listado del contenido del campo privado "índice" donde para cada término indexado se imprimirá: cout << termino << '\t' << InformacionTermino << endl;
    cout << "Documentos indexados: " << endl;
          // A continuación aparecerá un listado del contenido del campo privado "indiceDocs" donde para cada documento indexado se imprimirá: cout << nomDoc << '\t' << InfDoc << endl;
}

bool IndexarPregunta(const string& preg);
```
// Devuelve true si consigue crear el índice para la pregunta "preg". Antes de realizar la indexación vaciará los campos privados indicePregunta e infPregunta
// Generará la misma información que en la indexación de documentos, pero dejándola toda accesible en memoria principal (mediante las variables privadas "pregunta, indicePregunta, infPregunta")
// Devuelve falso si no finaliza la operación (p.ej. por falta de memoria o bien si la pregunta no contiene ningún término con contenido), mostrando el mensaje de error correspondiente

```
bool DevuelvePregunta(string& preg) const;
```

```
    // Devuelve true si hay una pregunta indexada (con al menos un término
    que no sea palabra de parada, o sea, que haya algún término indexado en
    indicePregunta), devolviéndola en "preg"

bool DevuelvePregunta(const string& word, InformacionTerminoPregunta& inf)
const;
    // Devuelve true si word está indexado en la pregunta, devolviendo su
    información almacenada "inf". En caso que no esté, devolvería "inf"
    vacío

bool DevuelvePregunta(InformacionPregunta& inf) const;
    // Devuelve true si hay una pregunta indexada, devolviendo su
    información almacenada (campo privado "infPregunta") en "inf". En caso
    que no esté, devolvería "inf" vacío

void ImprimirIndexacionPregunta() {
    cout << "Pregunta indexada: " << pregunta << endl;
    cout << "Terminos indexados en la pregunta: " << endl;
          // A continuación aparecerá un listado del contenido de
          "indicePregunta" donde para cada término indexado se imprimirá:
          cout << termino << '\t' << InformacionTerminoPregunta << endl;
    cout << "Informacion de la pregunta: " << infPregunta << endl;
}

void ImprimirPregunta() {
    cout << "Pregunta indexada: " << pregunta << endl;
    cout << "Informacion de la pregunta: " << infPregunta << endl;
}

bool Devuelve(const string& word, InformacionTermino& inf) const;
    // Devuelve true si word está indexado, devolviendo su información
    almacenada "inf". En caso que no esté, devolvería "inf" vacío

bool Devuelve(const string& word, const string& nomDoc, InfTermDoc& InfDoc)
const;
    // Devuelve true si word está indexado y aparece en el documento de
    nombre nomDoc, en cuyo caso devuelve la información almacenada para word
    en el documento. En caso que no esté, devolvería "InfDoc" vacío

bool Existe(const string& word) const;
    // Devuelve true si word aparece como término indexado

bool Borra(const string& word);
    // Devuelve true si se realiza el borrado (p.ej. si word aparece como
    término indexado)

bool BorraDoc(const string& nomDoc);
    // Devuelve true si nomDoc está indexado y se realiza el borrado de
    todos los términos del documento y del documento en los campos privados
    "indiceDocs" e "informacionColeccionDocs"

void VaciarIndiceDocs();
    // Borra todos los términos del índice de documentos

void VaciarIndicePreg();
    // Borra todos los términos del índice de la pregunta

bool Actualiza(const string& word, const InformacionTermino& inf);
    // Será true si word está indexado, sustituyendo la información
    almacenada por "inf"

bool Inserta(const string& word, const InformacionTermino& inf);
    // Será true si se realiza la inserción (p.ej. si word no estaba
    previamente indexado)

int NumPalIndexadas() const;
    // Devolverá el número de términos diferentes indexados (cardinalidad de
    campo privado "índice")
```

```
string DevolverFichPalParada () const;
    // Devuelve el contenido del campo privado "ficheroStopWords"

void ListarPalParada() const;
    // Mostrará por pantalla las palabras de parada almacenadas (originales,
    sin aplicar stemming): una palabra por línea (salto de línea al final de
    cada palabra)

int NumPalParada() const;
    // Devolverá el número de palabras de parada almacenadas

string DevolverDelimitadores () const;
    // Devuelve los delimitadores utilizados por el tokenizador

bool DevolverCasosEspeciales () const;
    // Devuelve si el tokenizador analiza los casos especiales

bool DevolverPasarAminuscSinAcentos () const;
    // Devuelve si el tokenizador pasa a minúsculas y sin acentos

bool DevolverAlmacenarPosTerm () const;
    // Devuelve el valor de almacenarPosTerm

string DevolverDirIndice () const;
    // Devuelve "directorioIndice" (el directorio del disco duro donde se
    almacenará el índice)

int DevolverTipoStemming () const;
    // Devolverá el tipo de stemming realizado en la indexación de acuerdo
    con el valor indicado en la variable privada "tipoStemmer"

bool DevolverAlmEnDisco () const;
    // Devolverá el valor indicado en la variable privada "almEnDisco"

void ListarInfColeccDocs() const;
    // Mostrar por pantalla: cout << informacionColeccionDocs << endl;

void ListarTerminos() const;
    // Mostrar por pantalla el contenido el contenido del campo privado
    "índice": cout << termino << '\t' << InformacionTermino << endl;

bool ListarTerminos(const string& nomDoc) const;
    // Devuelve true si nomDoc existe en la colección y muestra por pantalla
    todos los términos indexados del documento con nombre "nomDoc": cout <<
    termino << '\t' << InformacionTermino << endl; . Si no existe no se
    muestra nada

void ListarDocs() const;
    // Mostrar por pantalla el contenido el contenido del campo privado
    "indiceDocs": cout << nomDoc << '\t' << InfDoc << endl;

bool ListarDocs(const string& nomDoc) const;
    // Devuelve true si nomDoc existe en la colección y muestra por pantalla
    el contenido del campo privado "indiceDocs" para el documento con nombre
    "nomDoc": cout << nomDoc << '\t' << InfDoc << endl; . Si no existe no se
    muestra nada
```

private:

```
IndexadorHash();
    // Este constructor se pone en la parte privada porque no se permitirá
    crear un indexador sin inicializarlo convenientemente. La inicialización
    la decidirá el alumno

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 documento


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

    string pregunta;
        // Pregunta indexada actualmente. Si no hay ninguna indexada, contendría
        el valor "”

    unordered_map<string, InformacionTerminoPregunta> indicePregunta;
        // Índice de términos indexados en una pregunta. Se almacenará en
        memoria principal

    InformacionPregunta infPregunta;
        // Información recogida de la pregunta indexada. Se almacenará en
        memoria principal

    unordered_set<string> stopWords;
        // Palabras de parada. Se almacenará en memoria principal

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

    Tokenizador tok;
        // Tokenizador que se usará en la indexación. Se inicializará con los
        parámetros del constructor: detectComp y minuscSinAcentos


    string directorioIndice;
        // "directorioIndice" será el directorio del disco duro donde se
        almacenará el índice. En caso que contenga la cadena vacía se creará en
        el directorio donde se ejecute el indexador

    int tipoStemmer;
        // 0 = no se aplica stemmer: se indexa el término tal y como aparece
        tokenizado
        // 1 = stemmer de Porter para español
        // 2 = stemmer de Porter para inglés
              // Para el stemmer de Porter se utilizarán los archivos
              stemmer.cpp y stemmer.h, concretamente las funciones de nombre
              "stemmer”

    bool almacenarEnDisco;
        // Esta opción está ideada para poder indexar colecciones de documentos
        lo suficientemente grandes para que su indexación no quepa en memoria,
        por lo que si es true se almacenará la mínima parte de los índices de
        los documentos en memoria principal, p.ej. solo la estructura "indice"
        para saber las palabras indexadas, guardando únicamente punteros a las
        posiciones de los archivos almacenados en disco que contendrán el resto
        de información asociada a cada término (lo mismo para indiceDocs). Para
        los datos de la indexación de la pregunta no habría que hacer nada. En
        caso de que esta variable tenga valor false, se almacenará todo el
        índice en memoria principal (tal y como se ha descrito anteriormente).

    bool almacenarPosTerm;
        // Si es true se almacenará la posición en la que aparecen los términos
        dentro del documento en la clase InfTermDoc
};
```

Aclaraciones

Evaluación de la práctica

La nota se calculará del siguiente modo:

- 40% por la corrección automática con los ficheros de prueba.
- 15% por el cálculo de la eficiencia temporal “en disco” (almacenarEnDisco == true).
- 12,5% por el cálculo de la eficiencia temporal “en memoria” (almacenarEnDisco == false).
- 12,5% por el cálculo de la eficiencia espacial ”en memoria” (almacenarEnDisco == false).
- 20% por la revisión presencial de la práctica.

Uso de los mapas

A continuación se muestra una posible versión de algunos métodos y sobre una estructura *InformacionTermino* simplificada (habría que implementar la descrita anteriormente). Esta versión se sugiere como ejemplo de uso de ficheros y STL, pero se aconseja mejorar su eficiencia:

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <unordered_set>

using namespace std;

class InformacionTermino {
	friend ostream& operator<<(ostream&, const InformacionTermino&);
public:
	InformacionTermino(const int& fft, const int& ffd);
	InformacionTermino();
private:
	int ft;
	int fd;
};

InformacionTermino::InformacionTermino(const int& fft, const int& ffd)
{
    ft = fft;
    fd = ffd;
}

InformacionTermino::InformacionTermino()
{
    ft = fd = 0;
}

ostream&
operator<<(ostream& salida, const InformacionTermino& d)
{
	salida << d.ft << "\t" << d.fd << endl;
}

IndexadorHash::~IndexadorHash()
{
	indice.clear();
}

InformacionTermino
```

```
IndexadorHash::Devuelve(const string& word )
{
       return (indice[word]);
}

bool
IndexadorHash::Borra(const string& word )
{
       return(indice.erase(word) != 0);
}

bool
IndexadorHash::Inserta(const string& word, const InformacionTermino& inf)
{
    if(existe(word))
        return false;
    else
    {
        indice[word] = inf;
        numeroTerminos++;
        return true;
    }
}

bool
IndexadorHash::Existe(const string& word)
{
   return (indice.find(word) != indice.end());
}

bool
IndexadorHash::Actualiza(const string& word, const InformacionTermino& inf)
{
       InformacionTermino temp;
       if(Existe(word))
       {
              indice[word] = inf;
              return true;
       }
       else
       {
              cerr << "ERROR: En actualiza no existe la palabra "<<word<<endl;
              return false;
       }
}

ostream&
operator<<(ostream& salida, IndexadorHash& in)
{
       unordered_map<string, InformacionTermino>::iterator it;
       for ( it=in.indice.begin(); it!=in.indice.end(); it++ ) {
              salida <<  it-> first << '\t' << it->second;
       }
}
```

Ejemplo de fichero main.cpp

Seguidamente se muestra un posible ejemplo de fichero main.cpp:

```
#include <iostream>
#include <string>
#include "indexadorHash.h"

using namespace std;

main() {
    InformacionTermino inf1(1, 2), inf2(3, 4), inf3(5, 6);
```

```
    IndexadorHash a;

    a.Inserta("term1", inf1);
    a.Inserta("term2", inf2);
    a.Inserta("term3", inf3);
    cout << "Tras insertar: \n" << a;

    a.Actualiza("term1", inf3);
    a.Actualiza("term2", inf2);
    a.Actualiza("term3", inf1);
    cout << "Tras actualizar: \n" << a;

    cout << a.Borra("term2") << endl;
    cout << a.Borra("term2") << endl;
    cout << a.Borra("term5") << endl;
    cout << "Tras borrar: \n" << a;
}
```

Control de tiempos de ejecución

Seguidamente se muestra el fichero *main.cpp*, con el que se realizará el estudio de **tiempos de ejecución**, el cual se utilizará para evaluar las prácticas:

```
#include <iostream>
#include <string>
#include <list>
#include <sys/resource.h>
#include "tokenizadorClase.h"

using namespace std;

double getcputime(void) {
       struct timeval tim;
       struct rusage ru;
       getrusage(RUSAGE_SELF, &ru);
       tim=ru.ru_utime;
       double t=(double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;
       tim=ru.ru_stime;
       t+=(double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;
       return t;
}

main() {
       long double aa=getcputime();

       IndexadorHash   b("./StopWordsEspanyol.txt",   ".   ,:",   false,   false,
       "./indicePruebaEspanyol", 0, false, true);
       b.Indexar("listaFicheros.txt");
       cout << "Ha tardado " << getcputime() - aa << " segundos" << endl;
}
```

Cálculo de eficiencia espacial

Igualmente, para la evaluación de la práctica se analizará la **eficiencia espacial**, para lo que se utilizará: **`memory ejecutable`**. Este programa hace una estimación (aproximada) de la máxima cantidad de memoria usada por otro programa cualquiera. Esto se consigue haciendo un muestreo del uso de memoria unas 16 veces por segundo, quedándose con el máximo. Dicho programa se creará mediante la siguiente compilación:

```
g++ memory.cpp -o memory
```

Se mostrará la memoria utilizada por el programa en la línea en negrita que se muestra a continuación:

```
Memoria total usada: 10968 Kbytes
  "   de datos: 10832 Kbytes
  "    de pila: 136 Kbytes
```

Fichero makefile

La práctica se corregirá utilizando el siguiente fichero makefile:

```
.PHONY= clean

CC=g++
OPTIONS= -g -std= gnu++0x
DEBUG= #-D DEBUG
LIBDIR=lib
INCLUDEDIR=include
_OBJ= indexadorHash.o tokenizador.o stemmer.o indexadorInformacion.o
OBJ = $(patsubst %,$(LIBDIR)/%,$(_OBJ))

all: indexador

indexador:    src/main.cpp $(OBJ)
	$(CC)  $(OPTIONS)  $(DEBUG)  -I$(INCLUDEDIR)  src/main.cpp  $(OBJ)  -o
indexador

$(LIBDIR)/%.o : $(LIBDIR)/%.cpp $(INCLUDEDIR)/%.h
	$(CC) $(OPTIONS) $(DEBUG) -c -I$(INCLUDEDIR) -o $@ $<

clean:
	rm -f $(OBJ)
```

Aclaraciones comunes a todas las prácticas

STL

Se puede utilizar la librería STL, para lo que se puede consultar en http://en.cppreference.com/w/ o en http://www.cplusplus.com/ .

VALGRIND

Se aconseja el uso de la herramienta VALGRIND para comprobar el manejo correcto de la memoria dinámica. Modo de uso:

valgrind - -tool=memcheck - -leak-check=full nombre_del_ejecutable

Operaciones especificadas

Todas las operaciones especificadas son obligatorias.

Si una clase hace uso de otra clase, en el código nunca se debe incluir el fichero .cpp, sólo el .h.

En la parte PUBLIC no debe aparecer ninguna operación que haga referencia a la representación del tipo, sólo se pueden añadir operaciones de enriquecimiento de la clase.

En la parte PRIVATE de las clases se pueden añadir todos los atributos y métodos que sean necesarios para la implementación de los tipos.

Tratamiento de excepciones

Todos los métodos darán un mensaje de error (en cerr) cuando el alumno determine que se produzcan excepciones; para ello, se pueden añadir en la parte privada de la clase aquellas operaciones y variables auxiliares que se necesiten para controlar las excepciones.

Se considera excepción aquello que no permite la normal ejecución de un programa (por ejemplo, problemas al reservar memoria, problemas al abrir un fichero, etc.). NO se considera excepción aquellos errores de tipo lógico debidos a las especificidades de cada clase.

De cualquier modo, todos los métodos deben devolver siempre una variable del tipo que se espera.

Los mensajes de error se mostrarán siempre por la salida de error estándar (cerr). El formato será:

ERROR: mensaje_de_error (al final un salto de línea).

Forma de entrega

La práctica se programará en el Sistema Operativo Linux, y en el lenguaje C++. Deberá compilar con la versión instalada en los laboratorios de la Escuela Politécnica Superior.

La entrega de la práctica se realizará:

- En el SERVIDOR DE PRÁCTICAS en http://pracdlsi.dlsi.ua.es/
- A título INDIVIDUAL; por tanto requerirá del alumno que conozca su USUARIO y CONTRASEÑA en el Servidor de Prácticas.

Ficheros a entregar y formato de entrega

La práctica debe ir organizada en 3 subdirectorios:

DIRECTORIO 'include': contiene los ficheros (en MINÚSCULAS):

- "indexadorHash.h"
- "tokenizador.h"
- "stemmer.h"
- "indexadorInformacion.h"

DIRECTORIO 'lib': contiene los ficheros (NO deben entregarse los ficheros objeto ".o"):

- "indexadorHash.cpp"
- "tokenizador.cpp"
- "stemmer.cpp"
- "indexadorInformacion.cpp"

DIRECTORIO 'src': contiene TODOS los ficheros aportados por el alumno para comprobación de la práctica:

- Por ejemplo: "main.cpp" y "main.cpp.sal"

Además, en el directorio raíz, deberá aparecer el fichero “nombres.txt”: fichero de texto con los datos del autor. El formato de este fichero es:

1_DNI: DNI1

1_NOMBRE: APELLIDO1.1 APELLIDO1.2, NOMBRE1

Cuando llegue el momento de la entrega, toda la estructura de directorios ya explicada (¡ATENCIÓN! excepto el MAKEFILE), debe estar comprimida en un fichero de forma que éste NO supere los 300 K. Ejemplo:

```
user@srv4:~$ ls

        include
        lib
        nombres.txt
        src

user@srv4:~$ tar cvzf PRACTICA.tgz *
```

Evaluación

La práctica se corregirá casi en su totalidad de un modo automático, por lo que los nombres de las clases, métodos, ficheros a entregar, ejecutables y formatos de salida descritos en el enunciado de la práctica SE HAN DE RESPETAR EN SU TOTALIDAD.

Uno de los objetivos de la práctica es que el alumno sea capaz de comprender un conjunto de instrucciones y sea capaz de llevarlas a cabo. Por tanto, es esencial ajustarse completamente a las especificaciones de la práctica.

Cuando se corrige la práctica , el corrector automático proporcionará ficheros de corrección llamados “main.cpp”. Este fichero utilizará la sintaxis definida para cada clase y los nombres de los ficheros asignados a cada una de ellas: únicamente contendrá una serie de instrucciones #include con los nombres de los ficheros “.h”.

Las prácticas no se pueden modificar una vez corregidas y evaluadas (no hay revisión del código). Por lo tanto, es esencial ajustarse a las condiciones de entrega establecidas en este enunciado.

En especial, se debe llevar cuidado con los nombres de los ficheros y el formato especificado para la salida.