Ejercicio 1: Creación de índices

Paquete es.uam.eps.bmi.search.parsing

- Clase **HTMLSimpleParser** implementa **TextParser**:
  - parse: Dado un texto en formato HTML, se pasa todo a minúsculas, se analiza mediante Jsoup y, por último, limpiamos los caracteres no alfanuméricos.
- Clase **HTMLStopwordsParser** extiende **HTMLSimpleParser**:
  - Constructor: Es necesario suministrar un fichero de *stopwords*, con una línea por cada una de ellas.
  - parse: Además de realizar el parseado de su clase padre, reemplaza todas las apariciones de cada *stopword* de la lista creada en el constructor en el texto a analizar.
- Clase **HTMLStemParser** extiende **HTMLSimpleParser**:
  - Constructor: Es necesario saber que tipo de *stemmizador* se usará durante el parseo. Es posible usar cualquier nombre de clase de SnowballStemmer, sin embargo, se definen dos campos de clase **HTMLStemParser.ENGLISH_STEMMER** para un *stemmizador* genérico en lengua inglesa y **HTMLStemParser.PORTER_STEMMER** para el *stemmizador* Porter.
  - parse: Hace el parseado de su clase padre y pasa cada uno de los *tokens* por el *stemmizador*.
- Clase **HTMLAdvancedParser** extiende **HTMLSimpleParser**:
  - Constructor: Es necesario pasarle tanto la ruta al fichero de *stopwords* como el nombre del *stemmizador* a utilizar.
  - parse: Pasa el texto primero por el método parse del padre, luego por el de **HTMLStopwordParser** y finalmente por **HTMLStemParser**.

Paquete es.uam.eps.bmi.search.indexing

- Clase **BasicIndex**:
  - build: Dada una colección de documentos comprimidos, parsea cada uno de dichos documentos para obtener un array de términos en formato texto. Con ésta información se instancia un objeto de tipo **TextDocument**, al que se le asigna un identificador de documento para añadirlo a una lista de documentos de la propia clase. Por cada uno de estos documentos y su correspondiente contenido, se crean las listas de **Posting** de cada término, añadiendo y actualizando en caso necesario un correspondiente **HashMap** que vincula un **String** término, con una **List<Posting>**.

    Una vez leídos, analizados y construido el índice se procede a escribir a disco tanto el índice como la lista de documentos mediante el método **BinIO.storeObject** de la API **fastutil**.
  - load: Mediante el método **BinIO.loadObject** se recuperan tanto el índice como la lista de documentos en memoria.
  - getDocumentIds: Itera sobre toda la lista de **TextDocument** para construir la lista de identificadores en formato **String**.
  - getTerms: Construye una lista del *keyset* del mapa de término-postings.
  - getTermPostings: Realiza un simple **.get(término)** sobre el mapa.
- Clase **StopwordIndex** extiende **BasicIndex**:
  - No se realiza ninguna sobrecarga de métodos de la clase padre, ya que el cambio es el tipo de parseo que se hace, determinado por el **TextParser** pasado al método **BasicIndex#build**.
- Clase **StemIndex** extiende **BasicIndex**:
  - En este caso si se realiza una sobrecarga del método de **BasicIndex#getTermPostings**, ya que es necesario hacer un *stemmizado* del término del que se quieren obtener la lista de **Posting**.
- Clase **AdvancedIndex** extiende **BasicIndex**:
  - Igual que con **StemIndex**, se sobrecarga **BasicIndex#getTermPostings** para *stemmizar* el término pasado como argumento.
- Clase **IndexBuilder**:
  - Contiene un método *main* que construye los 4 tipos de índices arriba descritos dadas las rutas de dicho índice y de la colección de documentos.

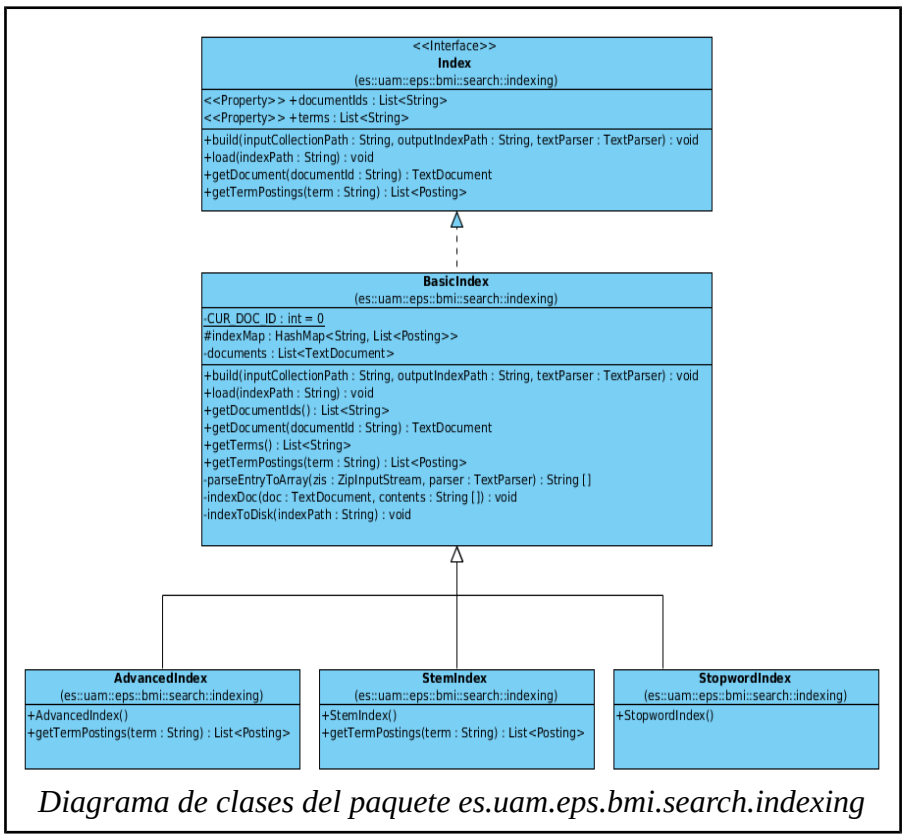

Diagrama de clases del paquete es.uam.eps.bmi.search.indexing

Ejercicio 2: Implementación de modelos de recuperación de información

Paquete es.uam.eps.bmi.search.searching

- Clase **BooleanSearcher** implementa **Searcher**
  - build: Dado un índice en memoria, lo copia a un campo de la clase.
  - setQueryOperator: Define el tipo de operador que se usará en las subsecuentes consultas, ya definidos como campos estáticos de la clase, **OR_OPERATOR** para la disyunción o **AND_OPERATOR.**
  - search: Dada una consulta, la descompone en cláusulas y obtiene sus respectivas listas de **Posting**. Con estas listas, dependiendo de que tipo de operador lógico se haya elegido, llamará a los métodos privados **union** para la disyunción o **intersection** para la conjunción.
- Clase **TFIDFSearcher** implementa **Searcher**
  - build: Dado un índice en memoria, lo copia a un campo de la clase.
  - setTopResults: Define el número máximo de resultados que devolverá las consultas que se realicen contra este buscador.

- search: Dada una consulta, la descompone en cláusulas y obtiene sus respectivas listas de **Posting**. Por cada término de la consulta, se calcula su valor *idf* (que corresponde con el logaritmo binario del tamaño de la lista de Postings). Por cada posting de dicha lista, se calcula el valor *tdf* del término en el documento mediante el atributo **termFrequency** del **Posting**.

  Con estos valores, se construye un objeto **ScoredTextDocument** con el identificador del documento y la puntuación resultante de multiplicar *tf* e *idf*. Cada uno de estos objetos se añaden a un **HashMap**, sabiendo que si ya existe en dicho diccionario se deberá actualizar el *score* de dicho documento.

  Cuando se tienen todos los documentos puntuados, se añaden secuencialmente a una **PriorityQueue** (implementación de un Heap de Java) de tamaño el top de resultados definido anteriormente, teniendo en cuenta que sólo se añadirá un determinado documento si el *head* de dicho Heap tiene una puntuación menor.

- Clase **LiteralMatchingSearcher** implementa **Searcher**
  - build: Dado un índice en memoria, lo copia a un campo de la clase.
  - setTopResults: Define el número máximo de resultados que devolverá las consultas que se realizen contra este buscador.
  - search: Dada una consulta la descompone en cláusulas y obtiene sus respectivas listas de **Posting**. A continuación se hace la intersección simple llevada a cabo por **intersection, devolviendo la lista de Postings en cuyos documentos se encuentran todos los términos de la consulta.** Posteriormente el método **getFullIntersection** realiza filtra aquellos postings en los que los términos de la consulta no estén seguidos así como calcular la frecuencia en la que aparece el término completo en un documento. Finalmente se realiza el mismo procedimiento que en **TFIDFSearcher**, calcula el tf-idf para luego obtener el score e introducirlo en una estructura **PriorityQueue** para presentar el top de resultados.

- Ejemplo del funcionamiento de las intersecciones:

consulta = obama family tree

obama (d1 freq 3 pos <3,7,9>) (d2 freq 1 pos 4) (d3 freq 1 pos 2)
family (d1 freq 2 pos <4,10>) (d2 freq 1 pos 2) (d3 freq 1 pos 10)
tree (d1 freq 2 pos <5,8>) (d2 freq 1 pos 10) (d4 freq 1 pos 11)

primera intersección con **intersection:**

obama (d1 freq 3 pos <3,7,9>) (d2 freq 1 pos 4) ~~(d3 freq 1 pos 2)~~
family (d1 freq 2 pos <4,10>) (d2 freq 1 pos 2) ~~(d3 freq 1 pos 10)~~
tree (d1 freq 2 pos <5,8>) (d2 freq 1 pos 10) ~~(d4 freq 1 pos 11~~)

segunda intersección con **getFullIntersection:**

obama (d1 freq 3 pos <3,7,9>)~~ (d2 freq 1 pos 4)~~
family (d1 freq 2 pos <4,10>) ~~(d2 freq 1 pos 2)~~
tree (d1 freq 2 pos <5,8>)~~ (d2 freq 1 pos 10)~~

obama (d1 freq 3 pos <3,~~7,9~~>)
family (d1 freq 2 pos <4,~~10~~>)
tree (d1 freq 2 pos <5,~~8~~>)

obama family tree en d1 posición 3

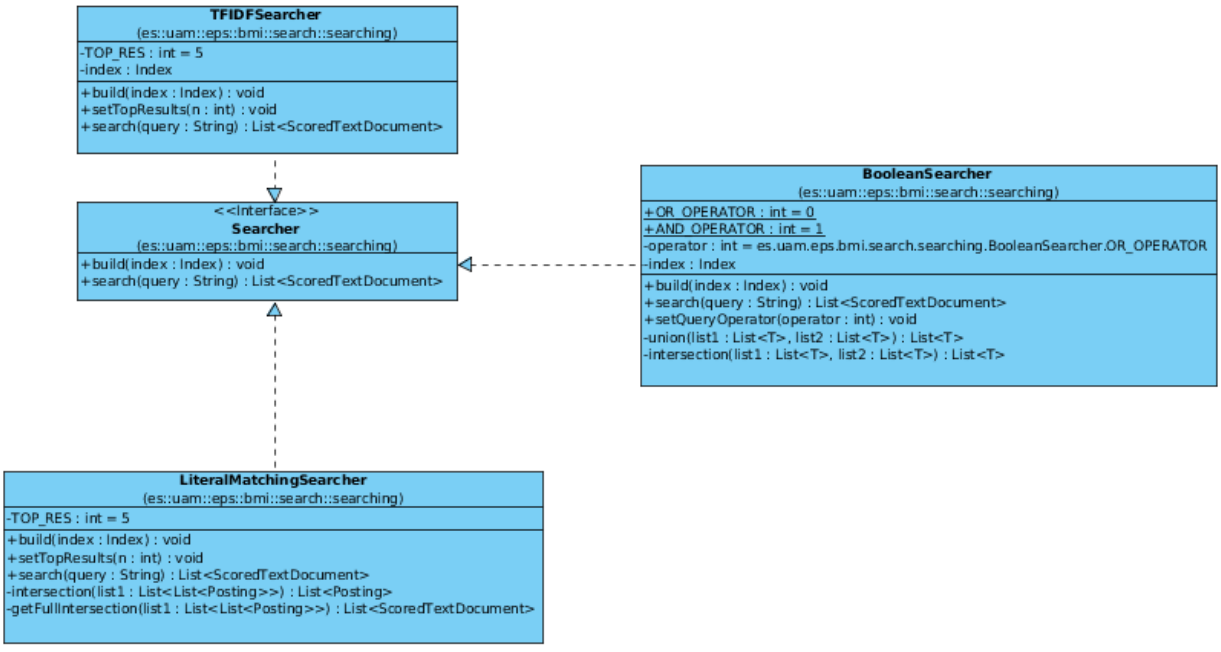

Diagrama de clases del paquete es.uam.eps.bmi.searching.search