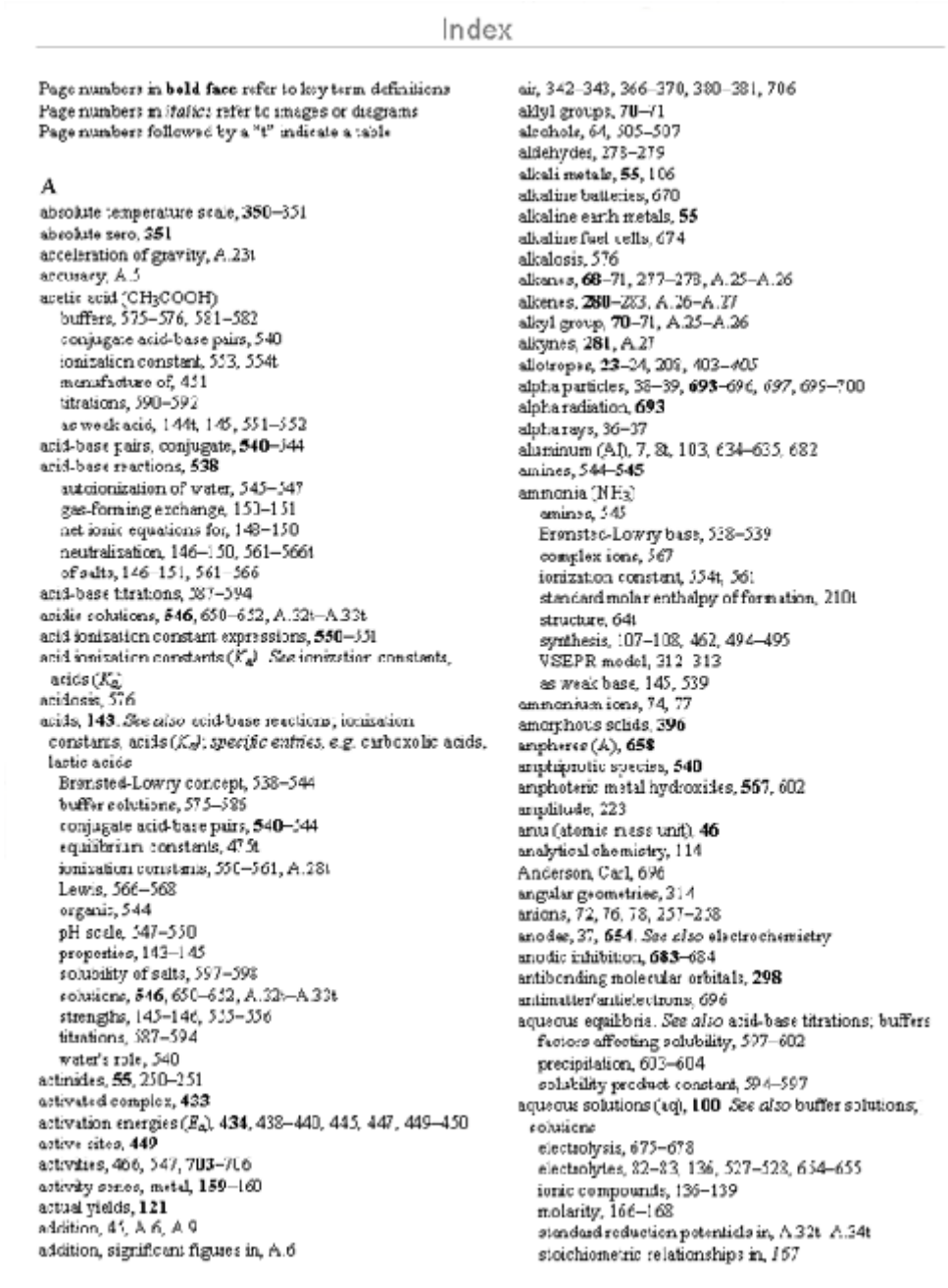
Implement a Java application that builds an Inverted Index

What's an inverted index

An inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a document or a set of documents. The purpose of an inverted index is to allow fast full text searches.

You have millions of documents or webpages or images anything that we may need to retrieve later. Here's an example of an inverted index, matching a word with pages in a book that contain that particular word.



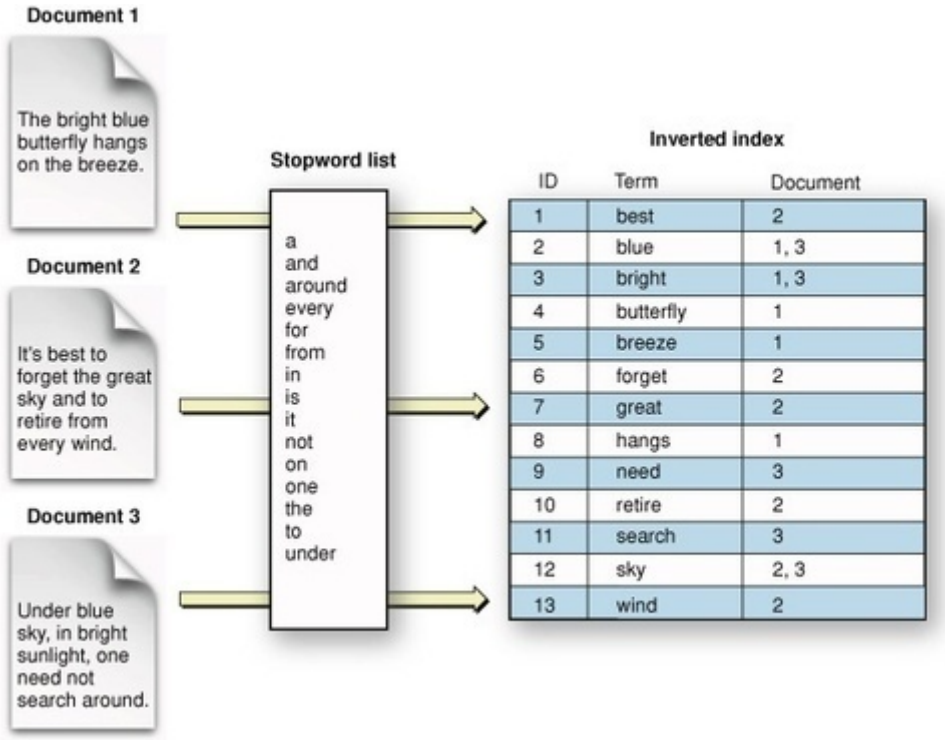
Index

Page numbers in **bold face** refer to key term definitions
Page numbers in *italics* refer to images or diagrams
Page numbers followed by a "t" indicate a table

This is an example from some random textbook. If you want to find something specific, say **activation energies,** you will open up this index and the inverted index will tell you the page numbers where that word is explained in a big bulk of a thousand pages.
As a human, if you were to perform a regular linear search, you'll take hours to reach that page. But now it was hardly a matter of seconds.

How to build an inverted index?

Building an inverted index for maintaining any kind of searching system requires you to perform a series of steps while parsing the pages or documents. Let's have a walkthrough while constructing our own search engine.



Resources: wikipedia.org, quora.com

Homework requirements

I want to create a search engine Java application for the documents on my computer.

1. Fetching the documents
Write a Java application which takes a file path as input. This file path is a directory containing all the documents that need to be indexed.
The application must go through all the documents, parse them and build the inverted index.

```
Enter file path where documents are located:
[user input]: /home/user/MyDocuments

> Indexing process started...
> Indexing Algebra.txt
> Indexing Algorithms.txt
> Indexing Biomechanics.pdf
```

1. Fetching the Document
Focus on reading text file (.txt) documents.
Bonus: Parse doc and pdf documents. You'll need to use some libraries to retrieve their text.

2. Removing the Stop Words
Consider the above paragraph. What were the important words we may be looking for? "text", "libraries", "doc", "pdf", "retrieve", "parse". But most of the other words are just a waste. We denote the most occurring words as "**stop words**". This includes "I", "the", "we", "is", "an", "to".
See the attached **stopwords.txt** files which contain the stop words.
The Java application needs remove the stop words so that they are not to be included in the inverted index. This is to reduce the index size.
It should support English words.
Bonus: Support Romanian words

3. Stem to the Root Word
Then comes Stemming. Now whenever I want to search for "retrieval", I want to see a document that has information about it. But the word present in the document is called "retrieve" instead of "retrieval". To relate the both words, I'll chop some part of each and every word I read so that I could get the "root word". Retrieve may become "retriev". So will "retrieval". We have to be sure about the rules we use to chop the words. There are standard tools for performing this like "Porter's Stemmer". You can play around with a porter stemmer here : Porter Stemmer Online
Bonus: Support Romanian words

4. Record Document IDs
Now get ready for the main task - Indexing.
Every document I have has got an unique document name. We'll take each non-stop word, we will stem and we will save it in the inverted index.

Example:
retriev ==> document1

If the same word is in some other document, we'll store:
retriev ==> document1
retriev ==> document2

But very soon I've to combine them in a single list
retriev ==> document1 AND document2

This needs some specific data structures that simplify your job.

We need to further improve our algorithm by storing how many times did the word occur in each document.
retriev ==> [document1|2 times] AND [document2|5 times]

Using the number of occurrences, we can rank the search results, displaying document2 before document1.

5. Search capability

At this point, the indexing process has finished. We should have an in-memory data structure which holds the inverted index for our documents.
The inverted index will contain mappings between a single word to the documents where they appear.
We want to be able to search **multiple** words. The results should contain the document names that contains **all** the desired words (intersection).

See example below:

```
Enter file path where documents are located:
[user input]: /home/user/MyDocuments

> Indexing...
> Indexing Algebra.txt
> Indexing Algorithms.txt
> Indexing Biomechanics.pdf
> Indexing Precalculus.pdf
> Indexing Statistics.txt
> Indexing Mathematical Methods.doc
> Indexing process finished.

Enter search words:
[user input]: abstract algebra

> Your query matches the following documents, presented in a ranked order:
> /home/user/MyDocuments/Precalculus.pdf
> /home/user/MyDocuments/Statistics.txt
> /home/user/MyDocuments/Mathematical Methods.doc
```