# News Aggregator

**Topics:** Indexing, information retrieval, ethical reasoning, algorithm analysis, comparing records, sorted and ordered structures, map structures.

**Due Date:** April 16th @ 11:59pm

*Starter Code*

## Background

Many people get their news from online news aggregators and search engines. Those sites can return very different results, as shown in the above pictures. Compare the homepages of Google News and Yahoo News on 2022 March 31 at between midnight and 1am. The discrepancies in the results stem from the following:

1. **News outlets:** News aggregator sites retrieve their content from news outlet websites, which often use different content sources.
2. **Algorithms:** News aggregators can use different algorithms to rank news articles for display. News aggregators can also automatically collect user data (such as location) to customize what content is presented to users.
3. **Personalization:** User personalization allows users to modify the content displayed on their personal homepages.

In this assignment, you will write a program that will reproduce the functionality of a news aggregator. A news aggregator is a type of search engine, and its implementation follows the same steps.

1. The first component of a web search engine is the ***crawler***. This is a program that downloads web page content that we wish to search for.
2. The second component is the ***indexer***, which will take these downloaded pages and create an inverted index.
3. The third component is ***retrieval***, which answers a user's query by talking to the user's browser. The browser will show the search results and allow the user to interact with the web.

Our news aggregator will also have a default page and will provide autocomplete capabilities (from HW5).

### Step 1: Parsing RSS and HTML Documents (the Crawler)

In this step, you will implement the method `Map<String, List<String>> parseFeed(List<String> feeds)`.

*Intro to RSS*

RSS:

- RSS (**R**eally **S**imple **S**yndication) is a Web content syndication format that can be used to provide a summary of a website
- News organizations use RSS to provide information about the content of their website
- RSS is a subdialect of XML and is a markup language
- **RSS elements** are the building blocks of RSS feeds (files) and are represented by tags
- **RSS tags** annotate/label pieces of content such as “title”, “link”, “description”, and so on

REQUIRED ELEMENTS:

| Element | Description | Example |
|---|---|---|
| title | The name of the channel. It’s how people refer to your service. If you have an HTML website that contains the same information as your RSS file, the title of your channel should be the same as the title of your website | GoUpstate.com News Headlines |
| link | The URL to the HTML website corresponding to the channel | http://www.goupstate.com/ |
| description | Phrase or sentence describing the channel. | The latest news from GoUpstate.com, a Spartanburg Hearld-Journal Web site |

SAMPLE RSS FILE:

```
<rss version="2.0">
  <title>Hw6 Sample RSS Feed</title>
  <description>Sample RSS feed for CIT5940 news aggregator</description>
  <link>http://localhost:8090/page1.html</link>
  <link>http://localhost:8090/page2.html</link>
  <link>http://localhost:8090/page3.html</link>
  <link>http://localhost:8090/page4.html</link>
  <link>http://localhost:8090/page5.html</link>
</rss>
```

PARSING RSS FEEDS:

We will use `jsoup` to scrape and parse documents (RSS and HTML).

Download the `jsoup` jar and add it to your project classpath.

**NOTE**
To add the project to Eclipse, right-click on the project in Eclipse -> Properties -> Java build path -> Libraries -> Add External Jars.

To add the project to IntelliJ, drag the .jar file into your `/lib` directory, then right-click on it and select “Add as Library…” and follow the prompts.)

Parsing an RSS feed means retrieving the URLs stored in (all) the link tags.

To open a document (RSS or HTML) using JSoup do:

```
Document doc = Jsoup.connect(url_of_the_document).get();
```

**NOTE**

Learn more about the Jsoup Document here!

Use the Document variable to get all the elements in the documents representd by a specific tag do:

```
Elements elements = doc.getElementsByTag(tag_name);
```

For example, to get all the `link` elements in an RSS feed, you will have:

```
Elements links = doc.getElementsByTag("link");
```

`Elements` is actually an ArrayList of `Element` objects. You can use a `for-each` loop to iterate through all the links in the collection.

```
for (Element link : links){
  //do something with link
}
```

To retrieve the text of a specific `link` `Element`, do

```
String linkText = link.text();
```

`linkText` will contain the url inside the Link tag. You can retrieve the content of the HTML document located at the URL you just retrieved.

### *Parsing HTML documents:*

The procedure to parse an HTML document is similar to the process for RSS documents. The content of a website is surrounded by `body` tags. You can retrieve the text contained inside the `body` tag using the steps described above.

Remove all punctuation and set everything to lowercase in the words before storing the content of a webpage. For example, modify the word `“Let’s”` to `“lets”` and modify the word `“CIT5940”` to `“cit5940”`.

Create a class named `IndexBuilder` (and `IndexBuilderTest`) that implements `IIndexBuilder`. You will implement `parseFeed()`, which extracts HTML documents for each RSS feed and maps a document’s URL (the key) to all the words in that document (the value) for all documents from all the RSS feeds. Your map key should be in the format of `“http://localhost:8090/page1.html”` and not `“page1.html”` so that the original document can be retrieved. Note that in the `<Map<String, List<String>>` returned by `parseFeed()`, the `List<String>` will have duplicate strings for words that get repeated. This is fine.

### *Testing:*

The RSS feed file is available here

Before submitting your code make sure that you comment out any tests not using the feed file above—you can't run your local tests on Gradescope since there won't be a server to host those files! Watch the lecture recording to see how to use the test files and the GUI. If you come up with an interesting test case that you'd like to have hosted on the internet for you, let Harry know and he can help you set that up.

Test that:

- Your map has the correct number of files
- Your map contains the names of the documents (URLs/keys)
- Your map contains the correct number of terms in the lists (values)

## Step 2: Create the forward index (indexer)

A *forward index* is used for applications such as topic modeling and most classification tasks. In this program, it will be used for classification (tagging) of news articles.

A forward index maps documents to a list of terms that occur in them. To identify the "relevant terms" a document, we will use an algorithm called TF-IDF which stands for Term Frequency-Inverse Document Frequency. The term frequency represents the count of a term in a document and the inverse document frequency penalizes words that appear in many documents (like articles, pronouns, etc.). You can read more about TF-IDF here.

TF-IDF is computed for each term in each document (for all the documents) using the following formulas:

- $$TF(t,d) = \frac{\text{Number of times term t appears in a document d}}{\text{Total number of terms in the document d}}$$
- $$IDF(t) = \log_e\left(\frac{\text{Total number of documents}}{\text{Number of documents with term t in it}}\right)$$
- $$TF\backslash\text{-}IDF(t,d) = TF(t,d) * IDF(t)$$

In your program, the forward index will be represented as a map of a document (key) and a map (value) of the tag terms and their TF-IDF values `(Map<String, Double>)`. The value maps are sorted by lexicographic order on the key (tag term).

Step 3 is performed by the method `Map<String, Map<String, Double>> buildIndex(Map<String, List<String>> docs);`

**Testing:** Below is a list of some TF-IDF values from the test files that you can use for testing.

| Term | Document | TF-IDF |
|---|---|---|
| data | page1.html | 0.1021 |
| structures | page1.html | 0.183 |
| linkedlist | page1.html | 0.091 |
| stacks | page1.html | 0.091 |
| lists | page1.html | 0.0916 |
| data | page2.html | 0.046 |

| Term | Document | TF-IDF |
|---|---|---|
| structures | page2.html | 0.0666 |
| search | page2.html | 0.0585 |
| binary | page2.html | 0.0499 |
| queues | page2.html | 0.0166 |
| implement | page3.html | 0.04877 |
| java | page3.html | 0.0487 |
| trees | page3.html | 0.0832 |
| treeset | page3.html | 0.0487 |
| binary | page3.html | 0.0277 |
| mallarme | page4.html | 0.0731 |
| poem | page4.html | 0.0731 |
| do | page4.html | 0.1463 |
| think | page4.html | 0.0731 |
| others | page4.html | 0.0731 |
| categorization | page5.html | 0.0894 |
| random | page5.html | 0.0894 |
| topics | page5.html | 0.0894 |
| files | page5.html | 0.0509 |
| completely | page5.html | 0.0894 |

## Step 3: Create the inverted index (indexer)

An inverted index is the main data structure used in a search engine. It allows for a quick lookup of documents that contain any given term. An inverted index is a map of a tag term (the key) and a Collection (value) of entries consisting of a document and the TF-IDF value of the term (in that document). The collection is sorted by reverse tag term TF-IDF value (the document in which a term has the highest TF-IDF should be visited first).

Step 4 is performed by the method `Map<?,?> buildInvertedIndex(Map<String, Map<String, Double>> index);`

**Wildcard generics (<?>)** are used here to allow you to practice data structure selection. Think of wildcard as "unknown types". The general rules of subtyping and Java generics also apply to wildcards. In this case, your data structure must be a subtype of Map and contains unknown types (at compile time) for the key and the value. Your goal here is to decide what implementation of Map is appropriate and what are the appropriate typesfor the Keys and the Values.

The Java Entry interface allows you to manipulate key-value pairs (similar to a tuple). You can use the AbstractMap.SimpleEntry<K,V> class that implements Entry everywhere you need to store key-value pairs in a data structure that is not a subtype of Map.

Choosing a collection: some implementations are sorted, and some are not. Some collections allow duplicates and others do not. If your collection is not sorted, you can create a Comparator to sort it (you can also use it with sorted collections) using the collections.sort() static method.

> **NOTE**
> For Step 4 and thereafter, you shouldn't modify the wildcard generics in the signatures. In the instances when you can't use wildcard generics and need to specify types, you should just cast.

*Testing:*

Test that:

- Your map is of the correct type
- Your map associates the correct files to a term
- Your map stores the documents in the correct order

## Step 4: Generate the home page

The homepage displays the tag terms and their associated articles. Tag terms are sorted by the number of articles. If two terms have the same number of articles, then they should be sorted by reverse lexicographic order.

Before displaying the homepage, we must remove *stop words*. Stop words are "words which are filtered out before or after processing of natural language text." Removing stop words gets rid of the noisy high-frequency words that don't give any information about the content of the document. You will remove all stop words when creating the collection of entries (Entry objects) that will be displayed on the home page. A `HashSet` of stop words is provided in `IIndexBuilder` (STOPWORDS)

Step 5 is performed by the method `public Collection<Entry<String, List<String>>> buildHomePage(Map<?, ?> invertedIndex)`

*Testing:*

Test that:

- Your collection is of the correct type
- Your collection stores the entries are in the correct order

## Step 5: Searching (retrieval)

The users should be able to enter a query term and our news aggregator will return all the articles related (tagged) to that term. The relevant articles are retrieved from the inverted index.

Step 6 is performed by the method `public List<String> searchArticles(String queryTerm, Map<?, ?> invertedIndex);`

*Testing:*

Test that:

- Your list contains the correct number of articles
- Your list contains the correct of articles

## Step 6: Autocomplete integration

**NOTE**
This step relies on the previous homework. If you did not complete HW5 and intend to resubmit it during finals week, **you will not be penalized by the autograder**. As long as your autocomplete file `autocomplete.txt` is generated correctly by your HW5 code, you will not be penalized.

All good search engines include autocomplete capabilities. We will use the autocomplete system built in the previous homework.

Copy your hw4 files (you do not need to include test files) and paste them into the source directory of your project. Our news aggregator needs to generate the file of tag terms that will be used to initialize our autocomplete module. By default, the autocomplete file is named `autocomplete.txt` and is located at the root of the assignment folder. `autocomplete.txt` format:

- The first line contains the number of words in the file
- All the other lines are formatted as follows (without the single quotes): ‘spaces 0 word’

**NOTE**
We are ignoring the weight (of a term) in this application so it is perfectly fine to set it to 0 (zero) for all the terms

**POTENTIAL ERROR:**
If after generating the autocomplete file, the new list (of the suggested terms) is not updated, just restart the application, select the same RSS feed but do not rebuild the autocomplete file. The suggestions should then be accurate.

**POTENTIAL ERROR:**
If your NewsAggregator GUI has issues in successfully opening, make sure that the URLs contain “http://” and not “https://”. **The autograder does not test the GUI.**

Step 6 is performed by the method `public Collection<?> createAutocompleteFile(Collection<Entry<String, List<String>>> homepage);`

*Testing:*

Test that:

- Your collection is of the correct type

- Your collection contains the correct number of words

## Step 7: Big-O Estimate

Provide a Big-O estimate of all the methods in `IIndexbuilder`. Put your answers with a short justification inside an (ASCII) file named Algorithm_analysis.txt file.

Polish your code and be proud of your work. You have just built a search engine from scratch!!!

## Step 8: Submit to Gradescope

Submit `IndexBuilder.java`, `IndexBuilderTest.java`, `Algorithm_Analysis.txt`, and your `README` to Gradescope. Make sure that you follow all of the CIT 5940 testing guidelines so that the autograder runs properly on your submission. Submissions without a test class will result in a grade of zero on the assignment.

# Grading

Your grade will consist of points you earn for submitting the homework on time and for the final product you submit.

The assignment is worth 300 points.

| Description | Points |
|---|---|
| Autograder | 155 |
| Testing (correctness & code coverage) | 50 |
| Algorithm Analysis | 35 |
| Readme | 5 |

This assignment was created by Eric Fouh, based on Jerry Cain's assignment. It has been updated in 2023 by Harry Smith & Daniel Paik.

References: ChengXiang Zhai and Sean Massung. 2016. Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA.