Overview

In this part of the Course Project you will implement a simplified search engine that will be able to answer search queries over a subset of Wikipedia pages. Given a query you will output the set of pages that are relevant for this query.

Please read the whole document carefully before beginning to code. If you are in doubt, ask the TAs.

1 Specification

You will code two programs: one for building the inverted index and the second one for answering queries using this index.

1.1 Building the Inverted Index

Your first program createIndex will build an inverted index from a collection of documents. You will build a positional index, that is, your postings must contain documents’ id’s and positions of each term occurrence within the document.

1.1.1 Parsing the collection

To build the index, we first need to clean up the given collection of documents (web pages). A collection is a plain text file, with the same structure as shown in Figure 1 in the Appendix. Since we downloaded the collection from Wikipedia, some pages might be ill-formed. In particular, the provided collection might not be a parsable XML file, as shown in the example: there might be dangling tags or characters like & (instead of the XML escape &amp;) inside the text of the page. Therefore, you must write your own parsing routine instead of using an XML parsing external library.

To simplify your work, we guarantee that the <page>, <id>, <title>, <text> tags are matched, and that the last three appear in this order inside a <page> element. Furthermore, none of these four tags appears inside the text of a page. The document id’s are unique, and ranges from 0 to N – 1, where N is the number of pages in the collection.

When parsing the collection, you should perform the following operations in this order:

- Obtain the page id, title and text by suitably parsing the <page> element;
- Store the page id and title as they appear in the collection (i.e., with the right capitalization, etc.);
• For the title and text of the page:
  - Concatenate the title and the text (with a newline separator between them), obtaining a stream of words;
  - Lower case all the words in the stream;
  - Obtain all the tokens in the stream: a token is a string of alphanumeric characters ([a-z0-9]), terminated by a non alphanumeric character. For example, `apple2orange, banana++, lemon 2012 and pears!` contains six tokens: `apple2orange`, `banana`, `lemon`, `2012`, `and`, `pears`;
  - Filter out all the tokens matching an element of the provided stop words list (see Section 2);
  - Stem each remaining token using the Porter stemmer (see Section 2), obtaining a stream of terms.

In this way, a page can be represented as a triple: (id, title, stream of terms).

You can find a complete example of the process in Figure 2 in the Appendix.

1.1.2 Inverted index format

You can decide the format on disk of your inverted index. However, after reading only the inverted index, you must be able to obtain the posting list associated with a term, with its position(s) within each document. Your inverted index must be contained in a single file.

1.1.3 Title index format

You should produce a title index too. You can decide its format on disk, provided that you store the id and the original title of each page (e.g., you can simply write the id, followed by a space and the title, one page per line). Your title index must be contained in a single file.

1.1.4 Input/Output

You will submit a bash script `createIndex.sh`, suitably invoking your `createIndex` program. Both your program and the bash script should accept four parameters, representing names of files:

• the list of stop words;
• the collection of pages;
• the inverted index to be built;
• the title index to be built.

To test your program, we will invoke something like this:

```
$ ./createIndex.sh myStopWords.dat myCollection.dat myIndex.dat myTitles.dat
```
1.2 Producing the Search Results

Your program queryIndex will accept queries and, for each query, it will output an ordered list of matching documents, according to the procedure described below.

1.2.1 Query types and query parsing

Your program will handle the following types of queries:

- One word query (OWQ): a single word (e.g., **Spartacus**);
- Free text query (FTQ): a sequence of at least two words, separated by a space (e.g., **Orange Clockwork** or **Eyes Wide Shut**);
- Phrase query (PQ): a sequence of at least two words, separated by spaces and in double quotes (e.g., **"Paths of Glory"** or **"Barry Lindon"**);
- Boolean query (BQ): any expression generated by the following grammar:

  \[
  \text{boolquery} \rightarrow \text{word} | (\text{boolquery}) | \text{boolquery AND boolquery} | \text{boolquery OR boolquery}
  \]

  (e.g., **Full OR Metal**, **2001 AND Odyssey AND Space**, **(Fear AND Desire) OR Shining**, **(Lolita OR Strangelove) AND (Start AND Stop)**)

Please note that "word" here means any string of alphanumeric or punctuation characters (excluding double quotes, reserved for PQs, and parentheses, reserved for BQs) terminated by a blank space: this definition allows the user to input queries like "**2001: A Space Odyssey**" (producing four words: **2001**, **A**, **Space**, **Odyssey**) or **Killer’s Kiss** (producing two words: **Killer’s** and **Kiss**).

Only capitalized AND and OR act as operands for BQs, that is, you can assume that if a query contains AND or OR, they represent the logic operators. The precedence order is the standard one, from highest to lowest: first ( and ), then AND, then OR. (Note that \((x \text{ OR } y \text{ AND } z)\) must be parsed as \(x \text{ OR } (y \text{ AND } z)\)).

You can assume that the PQs and BQs are well-formed. In case a query is composed by a single word, you should consider it as a OWQ (or FTQ, as they are equivalent in this case) rather than a BQ.

PQs differ from FTQs because they require each matching document to contain the terms corresponding to the words specified by the query, one next to the other and in the same order. BQs require the returned documents to contain terms corresponding to the boolean condition. If you code in Python, you can parse the boolean expression using our parser routine (see Section 2). If you code in another language, we suggest you to port our parser routine to your language of choice.

You must parse the queries performing the following operations in this order:

- Determine the type of the query (one word, free text, phrase, boolean), and perform the parsing accordingly;
- Remove all the operators (i.e., AND, OR, (, ), "), if any;
- Lower case all the words in the query;
• Obtain all the tokens in the stream;
• Filter out all the tokens matching an element of the provided stop words list;
• Stem each remaining token using the Porter stemmer, obtaining a stream of terms for the query.

You can find some examples in Figure 3 in the Appendix.

1.2.2 Matching documents

After parsing a query \( Q \), you will have its type \( T \) and a stream of \( k \) terms \( (t_1, t_2, \ldots, t_k) \), which are the stemmed version of the words contained in the query. You should determine the set of documents \( D_Q \) matching the given query \( Q \). This should be done according to the type of the query:

• if \( T \) is a OWQ, then \( D_Q \) is the set of documents containing at least one word whose stemmed version is \( t_1 \);
• if \( T \) is a FTQ, then \( D_Q \) is the set of documents containing at least one word whose stemmed version is one of the \( t_i \)'s;
• if \( T \) is a PQ, then \( D_Q \) is the set of documents containing words which, after the removal of stop words and after being stemmed, produce the subsequence \( (t_1, t_2, \ldots, t_k) \) in this order and with adjacent terms;
• if \( T \) is a BQ, then \( D_Q \) is the set of documents containing words which, after being stemmed, satisfy the boolean condition.

You can find some examples in Figure 3 in the Appendix.

1.2.3 Input/Output

You will submit a bash script \texttt{queryIndex.sh}, suitably invoking your \texttt{queryIndex} program. Both your program and the bash script should accept three parameters, representing names of files:

• the list of stop words;
• the inverted index;
• the title index.

Your program will read queries from the standard input, one at a time, and output to the standard output the list of the id's of the matching documents, sorted by increasing document id, separated by a blank space. If there is no matching document, output a blank line. After retrieving and outputting the documents matching a query, your program will wait for another query, until the user closes the standard input with a \texttt{CTRL+D}.

For example, the following might be a typical execution:
$ ./queryIndex.sh myStopWords.dat myIndex.dat myTitles.dat
Space Odyssey
  0 3 7 10 12 14 20 21 29 90
Space AND Odyssey
  10 12 14 20 29
Odyssey AND Space
  10 12 14 20 29
"2001: A Space Odyssey"
  10 12 20
Titanic
  0 3 7 78 129 1501

Note that the I/O interface specified above allows us to use redirection, as in the following example:

$ ./queryIndex.sh myStopWords.dat myIndex.dat myTitles.dat < myQueries.dat > myDocs.dat

The number of lines of myQueries.dat and myDocs.dat should be the same.

2 Data

For this assignment you will need the following files located in /course/cs158/data/part1/:

- testCollection.dat: the test collection, containing roughly 12% of the full collection (36 MB);
- testQueries.dat: some sample queries on the test collection;
- fullCollection.dat: the full collection (warning: 288 MB!);
- fullQueries.dat: some sample queries on the full collection;
- stopWords.dat: the list of stop words;
- createIndex.sh: sample Bash script to invoke your createIndex program;
- queryIndex.sh: sample Bash script to invoke your queryIndex program;
- readme.txt: a plain text file you should fill in with your information.

You will also need an implementation of the Porter stemming algorithm. You can find the appropriate module for your programming language at http://tartarus.org/~martin/PorterStemmer/. Moreover, if you are coding in Python, you can use our parser of boolean expression which you can find in /course/cs158/src/lib/.

To assist you in checking that your index is built correctly, we will provide you with a few queries and the corresponding expected results. We will release this expected output one week before the due date, announcing the details on the Course Web Page and through email.
3 Report

Please include a report in PDF format answering the questions below. When analyzing your data structures and algorithms, please adopt the big-O notation instead of generic “very long” or “very big”, parameterizing your analysis according to the relevant quantities (number of documents/terms/etc.).

Please make sure that your answer is within the sentence limit, we do not expect more than that, less is fine as long as it is complete. Be concrete, precise and concise. Please do not answer questions that we do not ask! If you are not sure, ask the TAs.

1. What data structures do you use for storing the inverted index? Do you use skip pointers? Explain your choice. In which format do you store the index on disk? (5-7 sentences)

2. How would your data structures for postings change if new documents were added frequently to the collection? (3-5 sentences)

3. Examine the lengths of the postings lists for the terms in the full collection and comment on their distribution. (3 sentences)

4. If you were not provided with the stop words list, how would you have created one? (3 sentences)

5. Please describe how you process phrase and boolean queries and any optimizations you added for faster processing. (3-5 sentences)

4 Collaboration Policy

You can work on the Course Project in pairs. You should work on all the Parts of the Course Project with the same team-mate, and only one submission per team is required. We strongly suggest you to consider working with another student.

Please email cs158tas at cs dot brown dot edu if you have any questions. If you are having trouble finding a partner,

5 Submission

Please copy all the files mentioned below into a separate directory, cd there, and run the handin script from that directory. Be aware that since the handin script copies recursively all the files from the directory it is executed from, if you run the script from, say, your home directory, it will handin all your files and directories! We will not grade handins from directories other than your project directory. The directory from where you run the handin script should contain only:

- Any source code you wrote.
- The two Bash scripts createIndex.sh and queryIndex.sh, suitably modified to call your programs.
- The plain text file readme.txt, filled in with the required information. (Please do not edit the structure of this file (i.e. FULL_NAME, etc.), just substitute our data with yours!)
• Your report, in PDF format, named report.pdf.

Note, as we are limited by the department as to how much space we can use, the TAs reserve the right to simply delete and not grade handins that are too large due to your including unnecessary files or directories. Please submit only the above files (do not submit your index, the collection, or your output!) using the following command:

```
$ /course/cs158/bin/cs158-handin part1
```

6 Evaluation

We will run your programs on a CS machine, using the full collection and the sample queries. We will verify that your programs produce results similar (if not the same) to those of our reference implementation. If you thoroughly follow this specification, you should be able to exactly reproduce our expected results.

It is mandatory that your programs adhere to the I/O interface described in this specification. In particular, check for unwanted prompts, debugging messages, newlines that you might have forgotten in your code. It is your responsibility to make sure that your code runs on the department machines and satisfies naming and usage criteria described in this document.

We will only evaluate submissions that respect the I/O specification, requiring you to fix your code, until it complies with the interface described in this document.

Your grade will depend on:

• The number of queries on the full collection that your program answers correctly.

• The algorithms/data structure you adopted.

• The quality of both your code and report.

We will not grade the speed of your code for this assignment, however we expect that (a) it will run (no syntax or runtime errors are allowed); and (b) it will produce its output in a reasonable time: less than 45 minutes to create/load the index on the full collection and less than 10 seconds to answer each query.

Have fun!

Appendix

Please see the next three pages for some examples of parsing the collection and the queries.
Figure 1: An example of collection file.
Page content, in Wikipedia format.
It might span multiple lines,
with blank lines
and [[internal links]],
punctuation, strange
characters !#$%^&*,
numbers like 2011, etc.

(a) Original page.

(b) After concatenating title and text, and lower casing. (In this and the following figures, strings are on different lines just for reader’s convenience.)

description of a page
description page

(c) Retaining the tokens only.

(d) After removing the stop words of, a, in, it, with.

described page page content wikipedia format
described page page content wikipedia format

(e) The final stream of terms, after stemming.

Figure 2: Parsing a page.
Figure 3: Processing queries to obtain the set of matching documents.