CS211 Assignment 1: Concordance

Owain Jones <odj@aber.ac.uk>

1. Data Structure

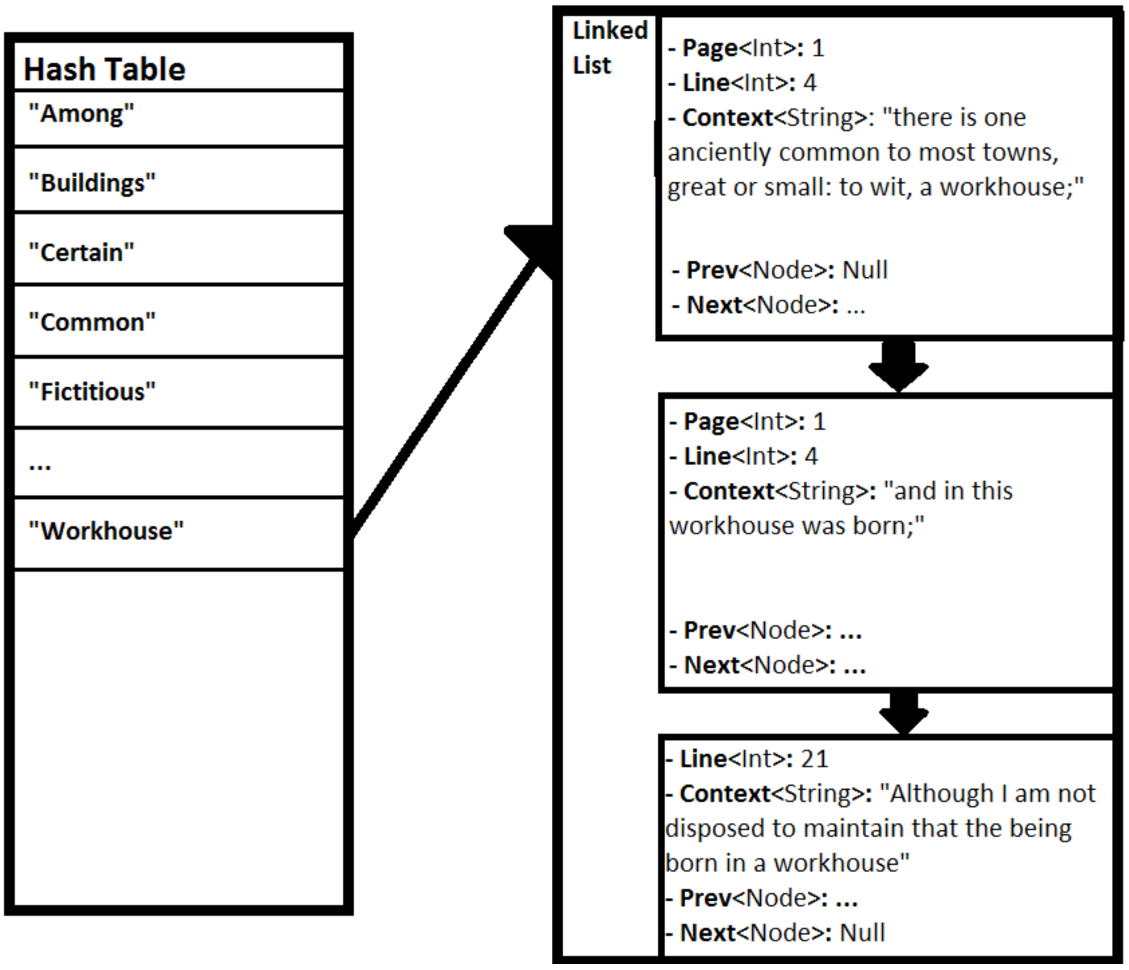
I decided to make use of Java's Generics and create a class called *Concordance* which would extend on Hashtable and only take Strings for keys and Linked Lists as values. I could then add my own methods to it (like *scan()* to populate the concordance) and not have to worry about creating get/set methods for accessing the underlying Hashtable.

I then decided that the data structure for storing references (page numbers, line numbers and contexts) for words will be a Linked List implementation, but one of my own design.

The Node objects will contain, instead of pointers to objects, several parameters: the page number, line number and a string containing the sentence in which the specific word has been found.

Since the text file will be read sequentially in a known order (from front to back), I feel confident that a Doubly-Linked List will be an efficient way of holding the indexes for every word.

A more visual representation of how my data structure may look like is as follows:



If I were to use a conventional Linked List, using conventional Nodes which simply point to other objects, the number of objects/pointers used per entry in the concordance would increase.

Considering that in a conventional Linked List, the encapsulating object counts as one pointer, and holds *head, tail* and *length* properties – that's 4 pointers per Linked List, minimum (Java has other properties in the LinkedList class). Then every conventional Node has minimum of 3 properties: *previous node, next node* and *data*. That *data* property is a pointer to an object which holds the actual data for each index for that given word, which holds 3 more properties; *page, line* and *context*. So the total number of references per word is at least:

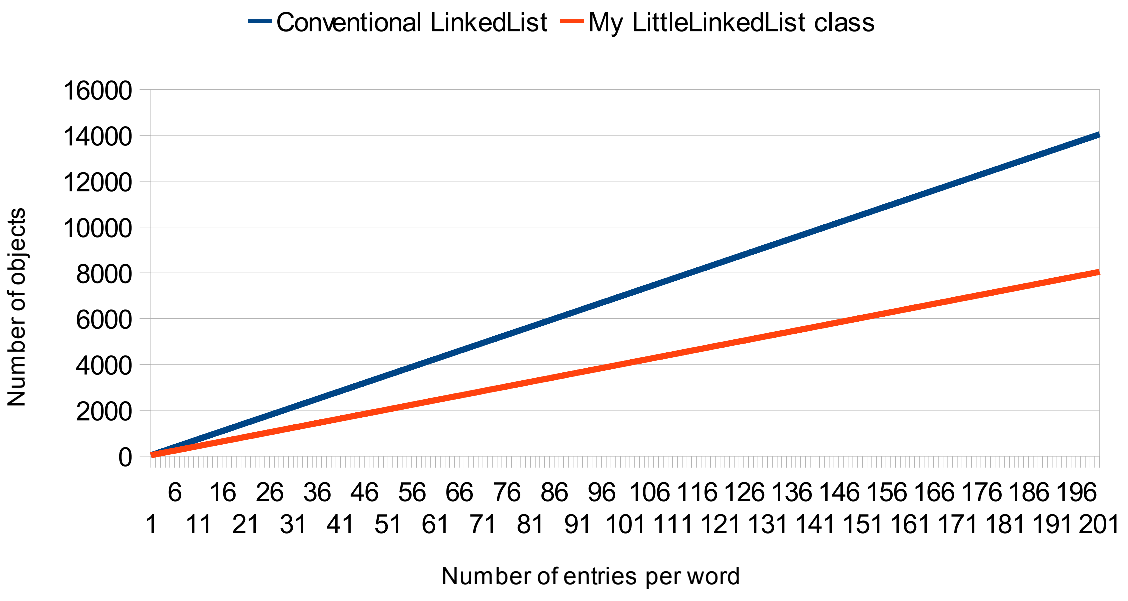
$$4+(n*7)$$

where *n* is the number of entries for any given word, and *7* is the sum of the objects for the Node, its properties and the object it points to.

In my approach the nodes don't encapsulate an object, they contain the properties for that entry, and we don't necessarily need to keep reference of the previous sibling of each Node, only the next sibling, so the total number of references per word is potentially down to:

$$4+(n*4)$$

Whilst the number of references used in the concordance's underlying data structure isn't important when you're only dealing with a small number of words, or a small piece of text, the amount of storage needed for the concordance in a large text (The word 'Jesus' in the *King James Bible* appears over 900 times, for example) can increase exponentially, as seen in this graph:

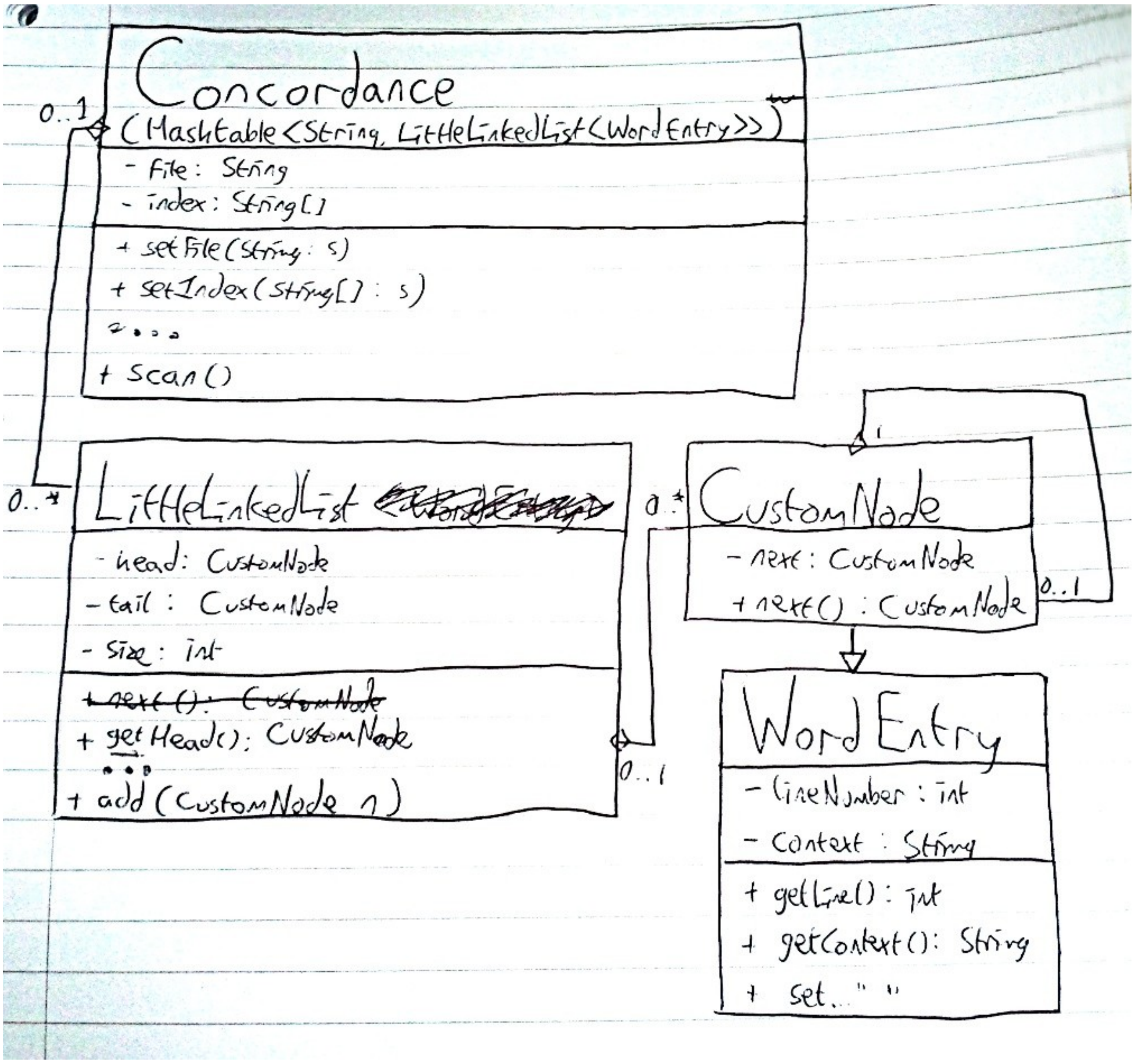


(Number of index words in concordance: 10)

When first thinking up this design, I wasn't not sure how much of an impact on the overall CPU/memory usage the slightly altered data structure would have – it may be minimal, but when dealing with large texts I think the almost 50% reduction in created objects definitely helps, in terms of performance.

I also wanted to be able to re-use some of the classes I wrote for this assignment in other things, so I've decided to make LittleLinkedList as generic as possible – for its nodes, it can use any class which is a child of the abstract CustomNode class I created. LittleLinkedList uses Generics, to make it harder to accidentally mix different CustomNode-extending classes.

A UML sketch of how my classes link together to create my data structure is on the following page.



2. Algorithm

When writing the algorithm for finding occurrences of words in a text, I focused on reducing the time complexity as much as possible. I also know that loading an entire text file into memory would be a bad idea (as you can't predict how big a file a user could give it). Even reading it line-by-line could be intensive if the file has no line-breaks. Taking these things into consideration, I decided my algorithm should scan the file character-by-character. This means that all the scanning and context finding would happen in a single loop – making its time complexity O(n).

When the end of a word or sentence is reached, I go into one of two inner loops checking if part of that word is in the concordance, or whether any index words were found before the end of the sentence – though I don't think this makes the time complexity of the algorithm O(n^2), because these loops only occur at the ends of words and sentences. So the time complexity after considering the inner loops, is approximately anywhere between *O(N)* and *O(N^2)*, I believe. I find it difficult to even approximate the time complexity as it depends so heavily on its input.

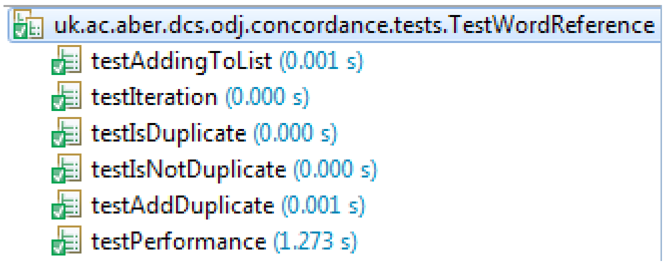
Before I implemented it in Java, I thought about it quite a bit and wrote it in pseudo-code, which is shown on the following page.

```
line number = 0
while (there are characters left in file) {
        if(character is alpha-numeric)
        {
                append to 'current word' object
        }
        else if(character is apostrophe (') or hyphen (-)
        {
                if('current word' is not empty)
                {
                        append to 'current word' object
                }
        }
        else if(character is punctuation (commas, full stops,
                [semi]colons, brackets, ...)
        {
                for each(character in 'current word')
                {
                        s = character in lower-case
                        if(concordance contains key s)
                        {
                                create new reference to word, w
                                add w to 'indexed words' list
                                also add w to hashtable, as key
                        }
                }
                clear/empty 'current word' object
        }
        else if(character is line-terminating character (\n))
        {
                increment line number
        }
        if(c is punctuation)
        {
                for each(each word in 'indexed words' list)
                {
                        set word context to 'current sentence'
                }
                clear/empty 'current sentence' object
                clear/empty 'indexed words' list
        }
        append character to 'current sentence' list
}
```

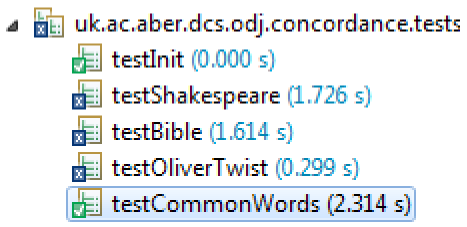
3 Testing

I did a lot of extensive testing – though a lot of it wasn't formalized inside JUnit tests. That said, I have two sets of JUnit tests (in the package `uk.ac.aber.dcs.odj.concordance.tests`) that test the integrity & performance of my data structure, and also that my Concordance class is operating correctly.

The results of these tests, after my final code revision, have the following results:



```
uk.ac.aber.dcs.odj.concordance.tests.TestWordReference
    testAddingToList (0.001 s)
    testIteration (0.000 s)
    testIsDuplicate (0.000 s)
    testIsNotDuplicate (0.000 s)
    testAddDuplicate (0.001 s)
    testPerformance (1.273 s)
```



```
uk.ac.aber.dcs.odj.concordance.tests
    testInit (0.000 s)
    testShakespeare (1.726 s)
    testBible (1.614 s)
    testOliverTwist (0.299 s)
    testCommonWords (2.314 s)
```

The one on the right is the results of the Concordance test – where tests have failed, it's because the concordance has under-reported the word count for an index word, in that text. I used the works of Shakespeare, the King James Bible and the Oliver Twist novel, all from *Project Gutenberg*, when testing my code. I used the the JUnit tests on the right mostly to look at timing; how long it took my algorithm to scan through an entire text for a single index word. `testCommonWords` looks for the most common words (and,if,of,but,the,it,a,...) in the English language, as an edge-case test for the algorithm.

I think JUnit is over-reporting the time taken however, as I also had my own test class outside of JUnit I used for performance testing – `uk.ac.aber.dcs.odj.concordance.Run`. That class times how long it takes the concordance to scan for 'Jesus' in the King James Bible; printing the result and the timings for initialization, scanning and iterating through results – that uses `System.nanoTime()` to get the times for each. On average, it takes around 600ms for `Concordance.scan()` to complete – which is about 2.5 times faster than what JUnit reports. However, JUnit was useful in checking edge cases; large indexes/very common words for example.

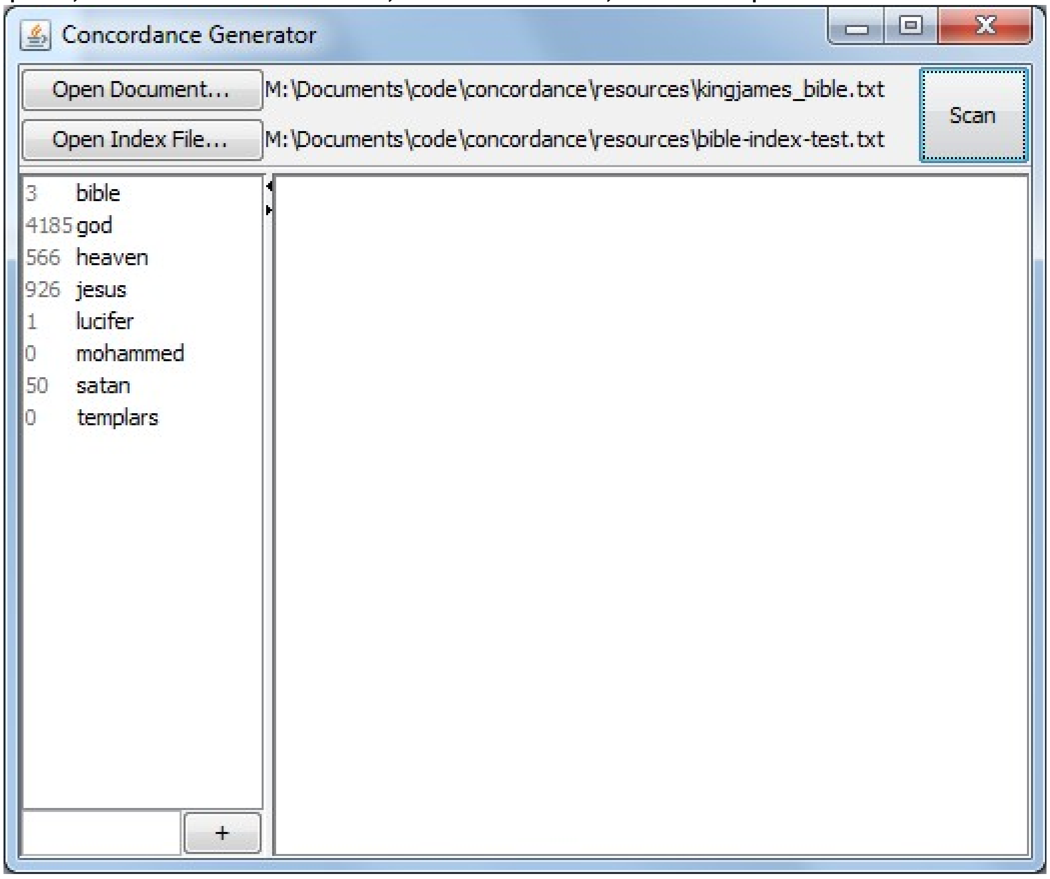
I started testing how much memory my approach was taking after I'd finished writing the GUI for my code. Without any good profilers available for my version of Eclipse, I had to check memory usage for every state my program was in by using Task Manager:

Memory Usage	Test 1 - King James Bible	Test 2 - Oliver Twist
Program Started	23,548K	21,144K
After scanning text	25,540K	37,248K
Largest index printed	38,956K	67,788K
Words Used (and count)	Bible (3), God (4,185), heaven (566), Jesus (926), Lucifer (1), Mohammed (0), Satan (50), Templars (0)	A (3277), and (4744), but (695), how (189), not (784), of (3426), or (2141), that (1663), the (7310), this (797), what (480), when (471), which (824), who (475)
Estimated Concordance memory usage	1,992K (~2MB)	16,104K (~16MB)

This isn't the most reliable way of testing; I'm ignoring Java's GC when testing memory usage – but I couldn't find a better way without

Evidence of me testing my program and it functioning correctly are on the next pages.

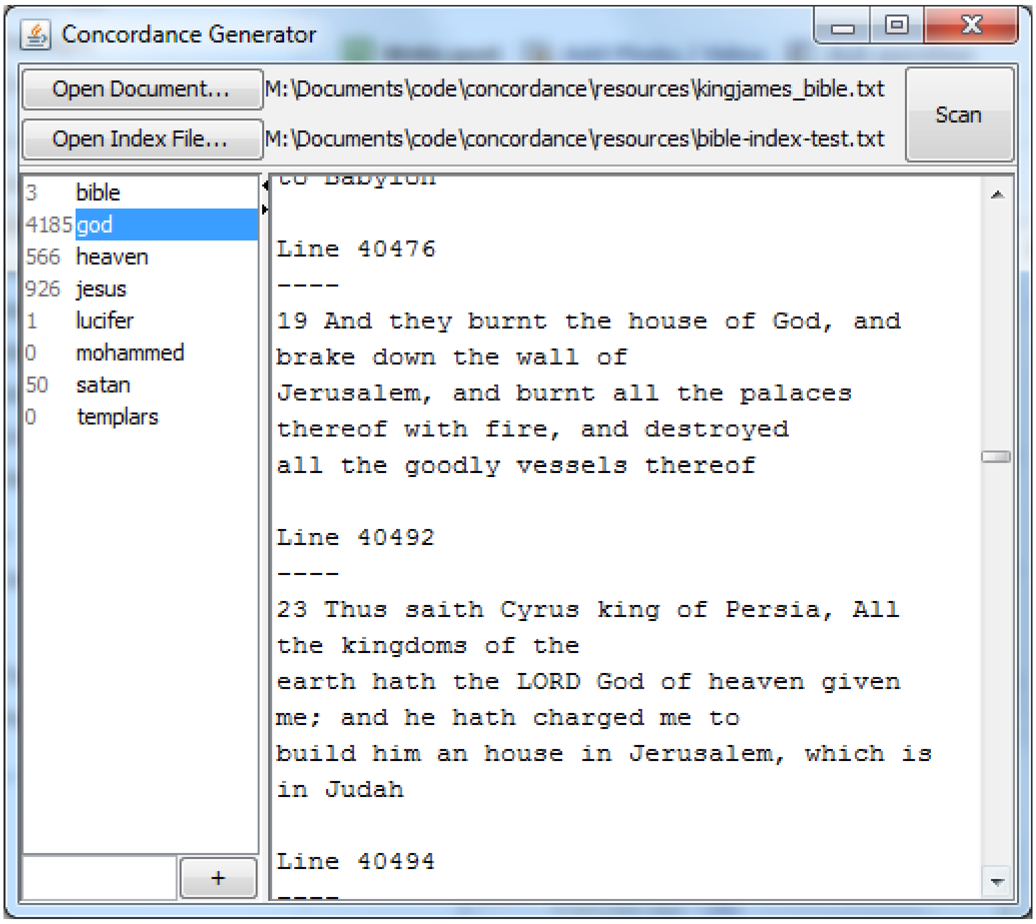
1. **Program opened, file selected and index loaded, 'scan' button clicked, word count updated**



M:\Documents\code\concordance\resources\kingjames_bible.txt
M:\Documents\code\concordance\resources\bible-index-test.txt

```
3     bible
4185  god
566   heaven
926   jesus
1     lucifer
0     mohammed
50    satan
0     templars
```

2. **Most commonly seen word clicked, checking that contexts are correct etc.**



M:\Documents\code\concordance\resources\kingjames_bible.txt
M:\Documents\code\concordance\resources\bible-index-test.txt

```
3     bible
4185  god
566   heaven
926   jesus
1     lucifer
0     mohammed
50    satan
0     templars
```

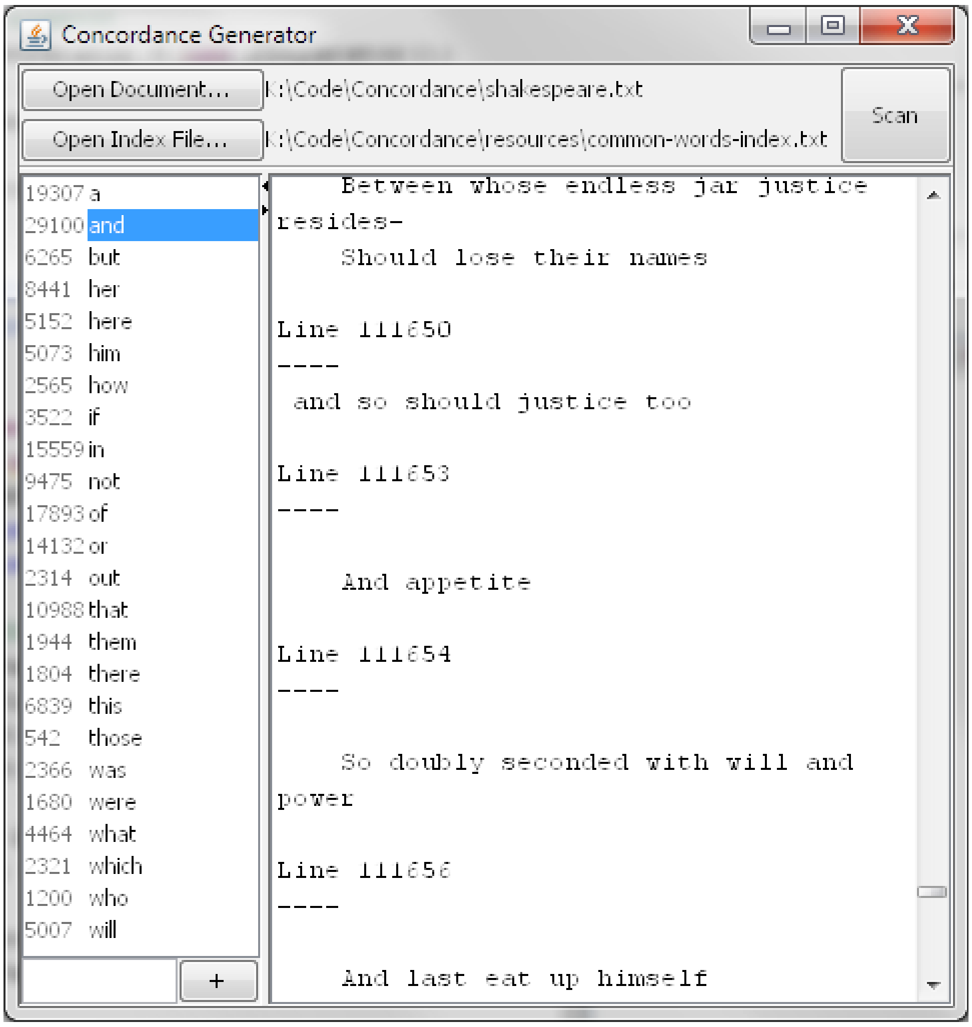
```
to Babylon

Line 40476
----
19 And they burnt the house of God, and
brake down the wall of
Jerusalem, and burnt all the palaces
thereof with fire, and destroyed
all the goodly vessels thereof

Line 40492
----
23 Thus saith Cyrus king of Persia, All
the kingdoms of the
earth hath the LORD God of heaven given
me; and he hath charged me to
build him an house in Jerusalem, which is
in Judah

Line 40494
----
```

3. **Testing concordance using extremely common words; stress-testing implementation to see if it'll run out of memory** (it didn't)



K:\Code\Concordance\shakespeare.txt
K:\Code\Concordance\resources\common-words-index.txt

19307	a
29100	and
6265	but
8441	her
5152	here
5073	him
2565	how
3522	if
15559	in
9475	not
17893	of
14132	or
2314	out
10988	that
1944	them
1804	there
6839	this
542	those
2366	was
1680	were
4464	what
2321	which
1200	who
5007	will

```
    Between whose endless jar justice
resides-
    Should lose their names

Line 111650
----
 and so should justice too

Line 111653
----


    And appetite

Line 111654
----


    So doubly seconded with will and
power

Line 111656
----


    And last eat up himself
```

4 Evaluation

I'm pretty pleased with the performance of my algorithm, given that, under normal usage, it takes just under a second to scan a relatively large text (lots of my tests resulted in 500-600ms scan times), with a reasonable index size (of ~20 words). It's quick enough in normal usage for me to not have worried about doing the scanning in a `BackgroundWorker` when running it with Swing.

The hardest part was finding a good context for every word – my algorithm does this well when a piece of text has perfect grammar, but most texts, especially older ones such as the works of Shakespeare and the Bible, don't have perfect grammar. Consequently some contexts can span multiple lines, whilst others only a few words. If I had more spare time, I'd write a more intelligent parser that uses stacks to keep track of punctuation elements – like the syntax checking algorithms used in compilers.

My scanning algorithm also appears to under-count the number of times a word is found in a file when compared to `` `grep -i <word> -c <file>` `` – around <10% less than `grep`. I think this is attributed to the fact that `grep` scans files per-line, whilst mine does it per context (which can be less than or greater than a single line of text), and only counts a word as being found once for every context, even if there are multiple occurrences (e.g. it would only count 'and' in *"and him and him and him... and him!"* twice, because the first 3 instances are in the same sentence fragment). This is intentional, and because my `LittleLinkedList` implementation has an `identical()` method which

Is used in `add()` to stop duplicate entries being added.

In terms of outputting the data from a concordance, I decided that that should be implementation-dependent. The Concordance itself is a Hashtable, so its keys won't be traversed in alphabetical order. I took this into consideration when writing the Swing front-end; the list of index words, on the left of the interface, is kept as a `ListModel` and sorted in the interface class, and then when one of those words are clicked on, `Concordance.get(word)` is called. The Concordance class itself doesn't make output to be ordered alphabetically.

There are a couple of limitations to my algorithm, which, given more time, I'd like to solve:

- No support for internationalization (e.g. no accented characters like *ü*) – due to the way I check for characters being alphanumeric (in the ranges a-z, A-Z, 0-9), accented characters return false when supplied to the `isAlphaNumeric` method. I am also aware that some Unicode characters are more than one character long when stored, even if they appear like they're a single character on-screen – the algorithm wouldn't currently be able to check these because it only checks a character at a time. I could solve this by implementing a Unicode Character class to replace my use of `char` within the algorithm, which can do equality tests etc. on multi-byte characters.
- Potential to run out of memory when scanning extremely long strings with no spaces – this is another edge-case, and could easily be solved by adding hard limits to the maximum length a word could be. But I didn't want to add hard limits, aka 'magic numbers', to my code at this point. But if the string we wanted to scan had no whitespaces or punctuation, my algorithm would continue adding characters to the 'current word'/'current sentence' buffers, potentially until the VM runs out of memory.

I'm aware of these limitations, but they're implementation-dependent, rather than limitations with my theoretical data structure & algorithm – so in the name of time management I've left these problems unsolved.