Mini Search Engine

## Xingzhi Zhang Buyi Lv Wei Zhou

## 3220104936 322010xxxx 3210103790

## Group 4

# Introduction

In this project, we have developed our own miniature search engine tailored to handle inquiries related to "[The Complete Works of William Shakespeare](http://shakespeare.mit.edu/)". The tasks undertaken include:

(1) Conducting a word count analysis on the Shakespearean corpus to discern stop words, also known as noisy words. This entails delineating between what constitutes "interesting" versus "noisy" terms.

(2) Constructing an inverted index over the Shakespearean corpus with word stemming. Notably, the stop words identified in the previous task are excluded from this index.

(3) Designing a query program that operates atop the inverted file index. This program is designed to accept user-specified words or phrases and return the corresponding document IDs.

(4) Conducting tests to illustrate the impact of query thresholds on the resultant search outcomes.

Throughout this endeavor, we have developed three executable files aimed at building the dictionary from the provided documents, identifying stop words within the constructed dictionary, and facilitating user word searches. Furthermore, we have meticulously compared the search efficiency under various search strategies and dictionary construction methodologies. Notably, we have leveraged a stop words table to enhance search efficiency and ensure more precise results. Additionally, we have evaluated different stop words thresholds to optimize performance.

# Data Structure/Algorithms

## Build Dictionary

This module read the string from the file and store it in the map structure, "<keyword, vector<>>", where the vector stores, part of the information from the book. In the process of getting the word, first the input of the file stream, to the variable of type string, and then, in reading the characters in the variable one by one, to form the word. For the structure of map, it can build the one-to-one correspondence between the keywords and the articles containing the keywords, which is convenient for the users to check, and so on.

**Algorithm1**: Input: Files including the articles

**1 Input:** Files including the articles

**2 Define** buf

**3 While** reading the file stream

**4 for** deal character in buf **do**

**5** **if**(character is an alpha)**then**

**6**  *start to build word*;

**7 end**

**8 else then**

**9**  finish build word, and update the index

**10 end**

**11 end**

## Stop Words

This module will read the keyword frequency file processed in the previous module, in this module, we chose to select the first 100 occurrences of high-frequency keywords as a stop word, because we have been tested and found that around 100, the frequency of keyword occurrences began to decline significantly

**Algorithm 2:** rank the most frequent words

**1** Input index files

**1** Define map<int,string> rank

**2 while** iterator the index **do**

**3** add cnt;

**4** rank[cnt]=key word

**5 end**

## Search

**Algorithm 3:** Search words in existing dictionary

**1 Input:** Serveral words or a sentence

**2 Define** *Singleword = empty string*;

**3 while** extract *Singleword* from input **do**

**4 if** *Singleword* is a stop word **then**

**5** continue;

**6 end**

**7** stem(*Singleword*);

**8** tolower(*Singleword*);

**9 if** the *Singleword’s* first letter havn’t been in map **then**

**10** InsertToMap(*SearchMap*, *Singleword*)

**11** Set\_intersection(*SearchMap* that have been found);

**12 end**

**13 end**

**14 print**(*intersection*)

This module is for users to search in the documents from the dictionary. We adopt kinds of methods to improve the efficiency. Firstly, we use stop words to exclude the meaningless words such as “the” “a” and so on. In that case, we avoid outputting many uncorrelated results and improving precision. Secondly, we make use a thought of hash. We classification the dictionary according to its first letter into different .txt files. Such as “apple” store in “a.txt”. Thus, we don’t need to load all dictionary into the memory and decrease the searching quantity then speedup the searching. Lastly, if we input more then one words, we find its intersection to find the documents that include all the keywords we input.

# Testing Results

## BuildDictionary

In the section on building the inverted index, we provide two ways to build it. One is to store them all in an "index.txt" file. Here we store the number of times the word appears, the name of the text it appears, and the number of times it appears in the text in a txt file after going through stem.

The format after creation is:

|  |
| --- |
| Word |
| Number of the article which has the Word |
| Article (which has the word) |
| Number of the appearance of the word in Article |
| Article (which has the word) |
| Number of the appearance of the word in Article |

**BuildDictionary\_single:** output a single “index.txt” file

**BuildDictionary\_multi:** output a series of “\*.txt” file (such as ,’ a.txt’,’ b.txt’, etc)

## Stop Words

We conducted a word count of the Shakespearean corpus and found a total of **812,622** words. We conducted a statistical analysis of the total occurrences of each word (referred to as '**Count**' below) and the frequency of their appearance in the texts (referred to as '**Article**' below).

We respectively generated word cloud diagrams based on Article (left) and Count (right).

|  |  |
| --- | --- |
|  |  |

The distribution and relationship between the two can be roughly summarized as follows:

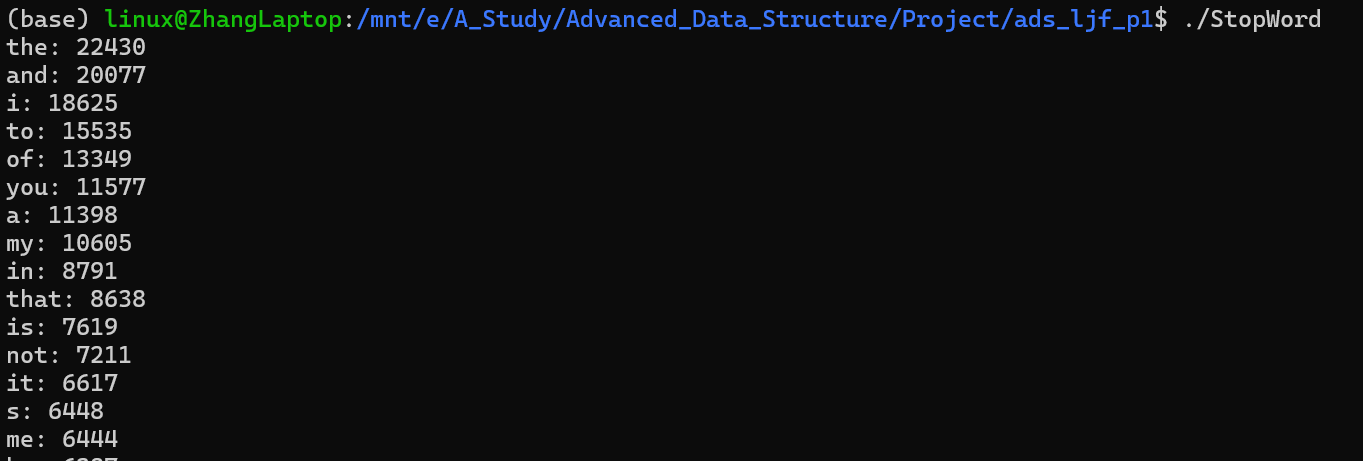
|  |  |
| --- | --- |
|  |  |

We selected the top 200 words based on their Count values and plotted their distribution (left). To analyze our truncated observations, we computed the growth rate using and as follows:

|  |  |
| --- | --- |
|  |  |

Based on our observation of the distribution, we found that both the absolute and relative increases in quantity decrease significantly after 100. Therefore, we have chosen the first 100 counts as our stopwords.

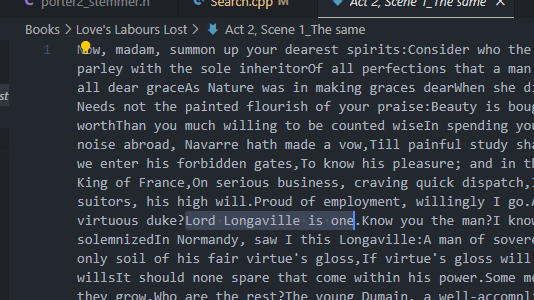
In the stop-word part, we set a threshold value of 100. The program picks the 100 most frequent words and stores them in a text file. In order to facilitate the subsequent search when excluded from the search results. You can change the threshold and output noisy words

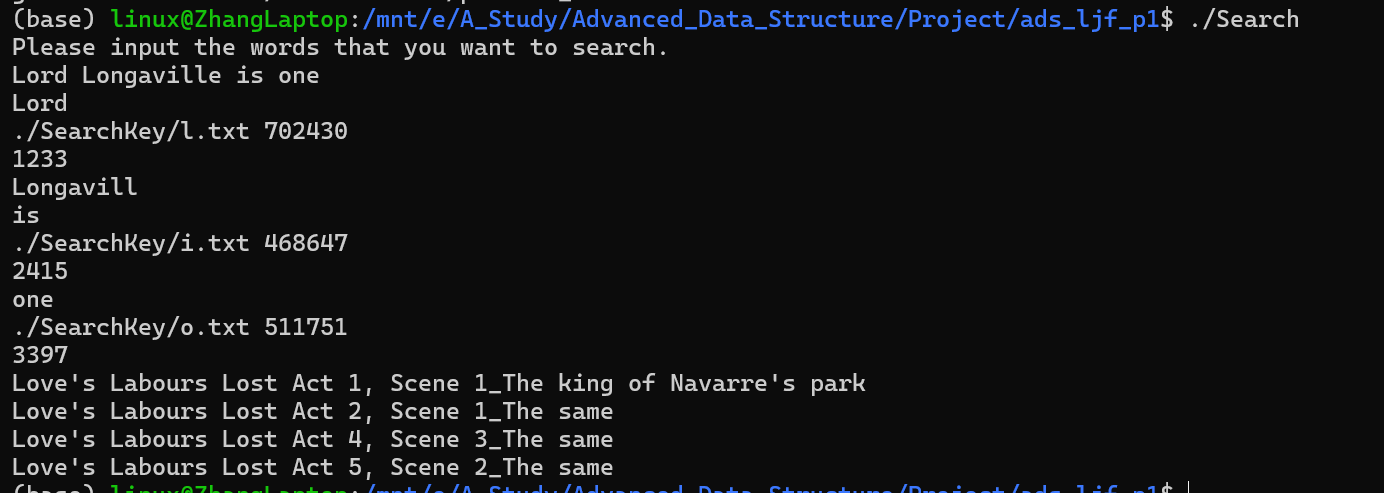


## Search

### Test search sentence 测试按照ppt再完善一下

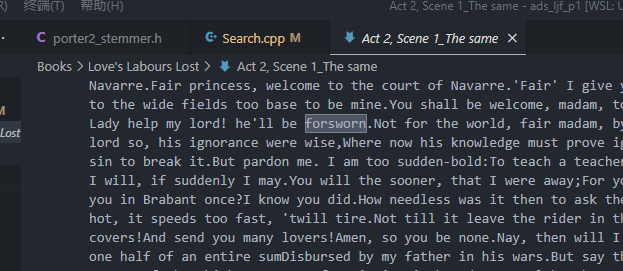
### We randomly find a more likely unique sentence from the document *Love’s Labours Lost Act 2, Scene 1\_The Same* as *“Lord Longaville is one”*. Then we use Seaech to search the sentence. We can find its output correspond the result. After verification, the results are accurate.

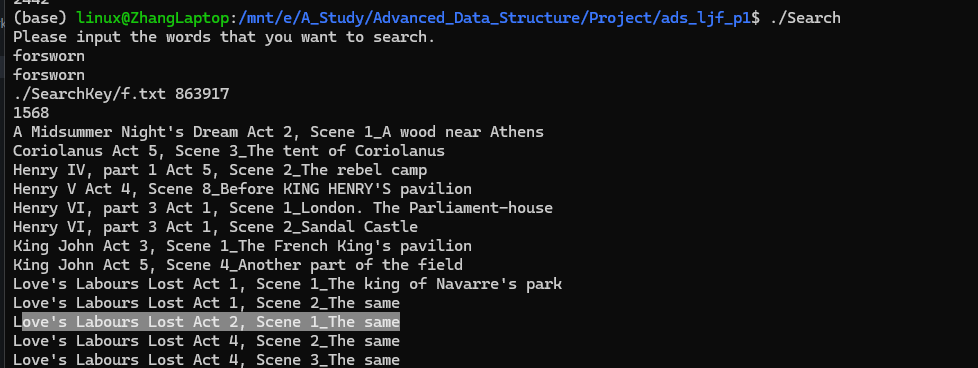




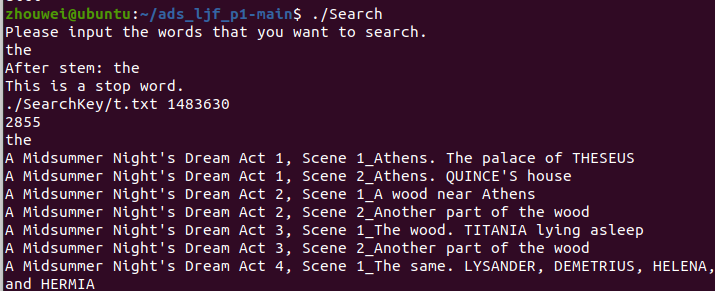
### Test search single word

We randomly chose some so what unique words to test the search of a single word. Here is a example. We choose “forsworn” from “Love’s Labours Lost Act 2, Scene 1\_The same”. And search it using our program. Easily we can find the result is satisfying.



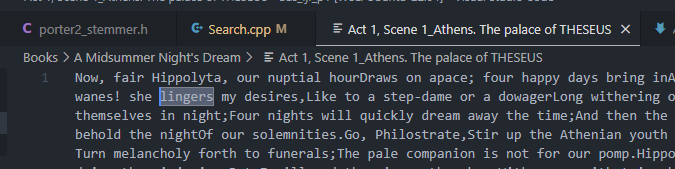


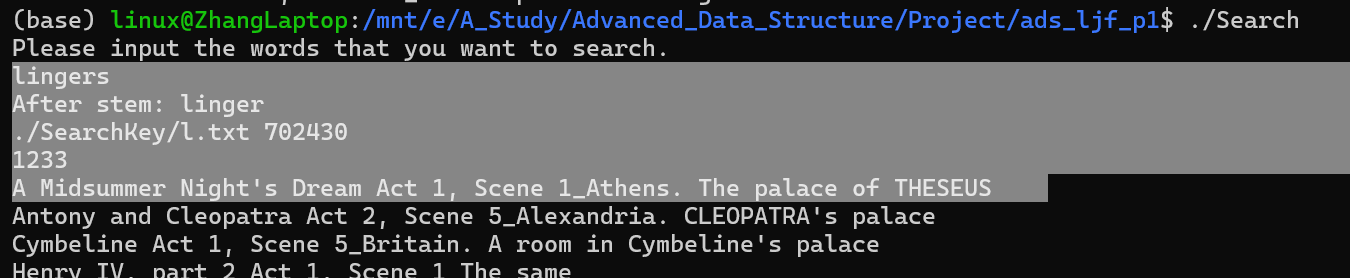
If it is a stop word:



### Test search single stem word

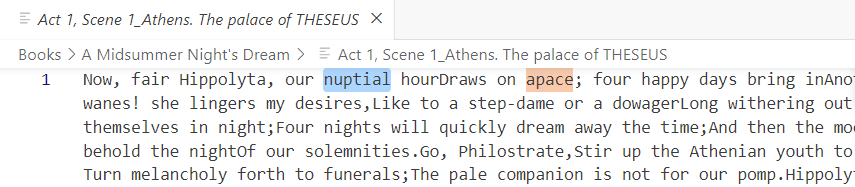
We randomly chose some so what unique words to test the search of a single word that needs to stem. Here is a example. We choose “*lingers*” from “*A Midsummer Night's Dream Act 1, Scene 1\_Athens. The palace of THESEUS”*, whilch is a plural word. And search it using our program. Easily we can find the result is properly stemmed and searched.

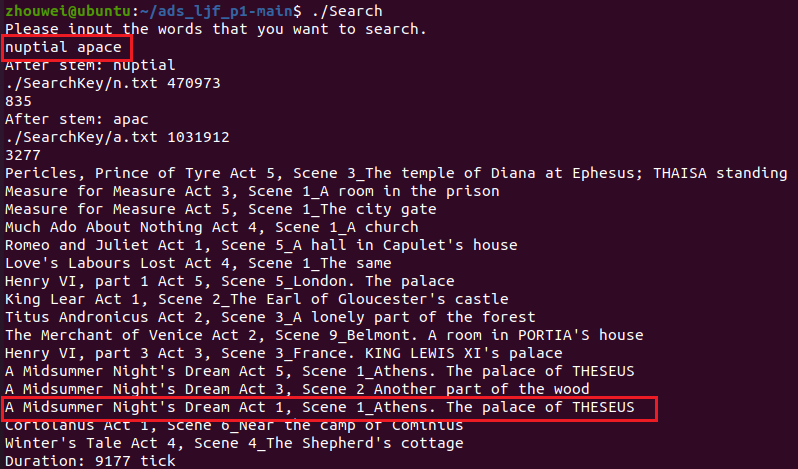




### Test multi words

We also tested finding the location of content where different words intersect





# Analysis and Comments

In the following test cases for runtime evaluation, we utilized a sample consisting of 500 words and 100 sentences, Threshold = 0.5 as default. The final results represent the average retrieval time, conducted in an Ubuntu 20 environment.

## Complexity Analysis

分析算法的时间和空间复杂度，分析你们的测试结果（如果是时间复杂度类型的请以图表形式展示），给出与现有最优结果的比较或者可能的改进方案（可选项）等。

## Search Strategy

In the implementation of the search function, if we do not split the index dictionary by initial letter, we will read a single file containing all the indexes. With the current amount of data, good results can be achieved. But there is a problem, if the dictionary grows large enough, it will take up too much space in memory. At the same time, due to the height of the search tree, the search time will increase. I think it should be improved.

The first method cost **186228.0333 ticks** for a random search.

This is where we get the idea if we are going to make a hash of the letter we are going to search for. The index terms with the same initial letter are stored in the same file, so that if we want to find the corresponding index, we can open the corresponding file. In this way, when a certain number of searches can reduce the memory space read, and the height of the search tree will be reduced.

The first method cost **8147.233333 ticks** for a random search. With the enhance of **22.858** times.

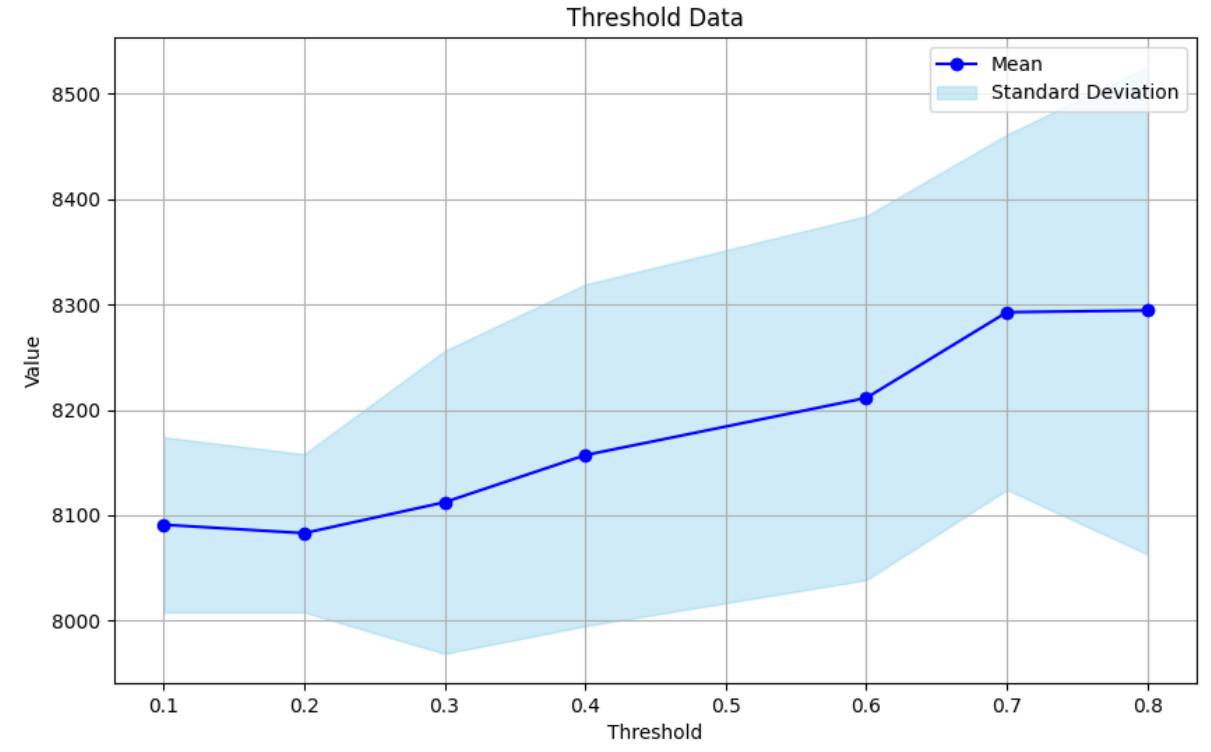


When the two methods reach a certain amount of search after continuous search, for example, when all 26 beginning letters have been searched, they will reach the same space complexity. However, if we follow the second approach, hashing by index, we can reduce the height of the search tree and achieve faster search speed.

When the size of the data increases further, we can further split the index. For example, split words into "aa.txt" and "ab.txt" and store them separately. When the memory capacity is not enough, some strategy should be adopted to expel the search tree with low usage from memory in time.

## Thresholds Rate

We changed the value of the threshold, conducted ten measurements to obtain the mean and variance, and plotted the results. We observed that as the threshold increased, the time gradually increased, but the magnitude of increase was small. We speculate that this is because high-frequency words themselves are scarce (and some are even filtered out as stop words), so increasing the threshold does not significantly increase the retrieval quantity.



# References

[1] *C++ STL Map*

[2] [*port2\_stemmer Library*](https://github.com/smassung/porter2_stemmer)