**Advanced Data Structures and**

**Algorithm Analysis Project-2**

**Roll Your Own Mini Search Engine**

**Author Names**

**Date: 2019-03-16**

### Chapter 1: Introduction

We have learnt the inverted file index last class, so in this project, we should create our own mini search engine that can handle queries from the "William Shakespeare Collection" (http://shakespeare.mit.edu/).

(1) Run the word count on the Shakespeare Set and try to identify stop words.

(2) Use the stem to create an inverted index on the Shakespeare set. The stop words identified in section (1) must not be included.

(3) Write a query program on top of your reverse file index that will accept the user-specified word (or phrase) and return the ID of the document containing the word.

(4) Run a test to show how the threshold of the query affects the outcome.

### Chapter 2: Data Structure / Algorithm Specification

#### Part One : Data Structure

1. **struct** HashNode {
2. string page;
3. string formerword;
4. **int** line;
5. HashNode\* Next;
6. };
7. **class** Document {
8. **public**:
9. **void** Files\_getOrigin(**void**);
10. //Files save the original data while database dir save the data manipulated
11. **int** stop\_words = 0;
12. //stop\_words refer to those without actual meaning in the text
13. **int** number\_words = 0;
14. //Returns with the number of different word
15. **int** words\_button = 0;
16. //Control the words\_count button and you can close it to speed up
17. **int** text\_button = 0;
18. //Control the text button and you can
19. **int** file\_button = 0;
20. //whether to save it in files
21. **void** SearchWord(string name);
22. //Search the word in the document
23. string Query(**void**);
24. //search the elements in ducument
25. **int** blank\_number = 0;
26. //record the blank number of every word
27. **void** Searchphrase(string name);
28. //Search the word in the document
29. **void** CalculateStop();
30. //Calculate the stop\_words
31. **void** SearchKey(string name);
32. //
33. **private**:
34. **int** word\_Camp(**const** string word);
35. //whether the word is a stopword or not
36. **void** Replace\_annotation(string\* content);
37. //Replace anotation character with blank space
38. **void** Document\_LoadOrigin(**const** string path, **const** string name);
39. //Load in the FILE CONTENT
40. vector<string>number\_wordscamp;
41. //Save and find string in camp
42. unordered\_map<string, HashNode>HashTable;
43. //Use unordered\_map to store
44. unordered\_map<string, HashNode>::iterator Poslocator;
45. //iterator to locate the content
46. **void** HashInsert(string page, **int** line, **struct** HashNode\* Header, string formerword);
47. //Insert the elements into the hashtable and save its position
48. map<string, **int**>Number\_wordAppearence;
49. //Sava each time every character appears
50. };

In this project, we use the hash-table to reflect on every word. And our hash-table node recorded the page, last word, line and the next node.

And we construct a class document, and it has lots of public and private variable, function and map, in order to solve the problem. And each intention of the variable and the function we have commented under them. And it is our fundamental data structure.

#### Part Two : Algorithm Specification

In this part we will introduce our main algorithm and how we deal with it. Step as follows:

1. We have already download all of the poems and articles of William Shakespeare, so we should read all the things by using the document.files\_getorigin to get names and id of all the files.
2. **void** Document::Files\_getOrigin(**void**)
3. //Get all the file in the directory
4. {
5. **struct** dirent \*dirp;
6. string name;
7. progress\_display pd(195);
8. DIR\* dir = opendir("./FILE");//use dir pointer
9. **while** ((dirp = readdir(dir)) != nullptr)
10. {
11. **if** (dirp->d\_type == DT\_REG)//case 1:it is a file
12. {
13. name = dirp->d\_name;
14. Document\_LoadOrigin("FILE", name);
15. **if** (text\_button == 0)
16. ++pd;
17. }
18. }
19. closedir(dir);
20. }
21. In this function, which contains another important function—“Document::Document\_LoadOrigin (const string path, const string name)“, have already help us count the number of the stop word and the all other words.
22. In the loadorigin function:
23. **void** Document::Document\_LoadOrigin(**const** string path, **const** string name)
24. {
25. **int** word = 0;
26. //Show The Document Function
27. ifstream Origin\_File("./" + path + "/" + name);
28. //Load in the Origin FIle
29. ofstream Rectify\_File("./DataBase/" + name);
30. //OUTPUT FILE
31. string file\_content;
32. //GET FILE CONTENT
33. vector<string> split\_string;
34. //SPLIT THE WHOLE STRING
35. string formerword;
36. //Save the former word in the page of the specific word
37. **while** (getline(Origin\_File, file\_content))
38. {
39. transform(file\_content.begin(), file\_content.end(), file\_content.begin(), ::tolower);
40. Replace\_annotation(&file\_content);
41. split(split\_string, file\_content, is\_any\_of(" "));
43. **for** (**int** i = 0; i < split\_string.size(); i++)
44. {
45. **if** (!word\_Camp(split\_string[i]) && split\_string[i] != " ")
46. {
47. word++;
48. **if** (file\_button)
49. Rectify\_File << split\_string[i] << " ";
50. //Number\_wordAppearence[split\_string[i]]++;
51. Poslocator = HashTable.find(split\_string[i]);
52. **if** (Poslocator != HashTable.end())//case 1:the key already exists
53. HashInsert(name, word, &Poslocator->second, formerword);
54. **else**//case 2:the key doesn't exist we need to create
55. {
56. HashNode \*NewNode = **new** HashNode;
57. NewNode->page = name;
58. NewNode->line = word;
59. NewNode->Next = NULL;
60. NewNode->formerword = formerword;//Save its former word
61. HashTable.insert(
62. pair<string, HashNode>(split\_string[i], \*NewNode));
63. //use pair to insert the element
64. }
65. formerword = split\_string[i];
66. }
67. **else**//case 2: the word are stop\_words
68. stop\_words++;
70. }
71. }
73. Origin\_File.close();
74. Rectify\_File.close();
75. }

And we perform word segmentation, first pre-processing, that is, turn all punctuation into spaces, and change all uppercase and lowercase to lowercase. In the file and word creation process, we use the hash table insert to project the same words (except stop words) into the hash table, and then each word forms a linked list in the slot.

Finally, we used unordered\_map (the underlying implementation is a hash table) for searching and other operations. Due to the characteristics of the hash table, we can see that we have a long time in the setup, but the search operation is fast, almost O(K), where K Is the number of words to be found.

### Chapter 3: Testing Results

Table of test cases. Each test case usually consists of a brief description of the purpose of this case, the expected result, the actual behavior of your program, the possible cause of a bug if your program does not function as expected, and the current status (“*pass*”, or “*corrected*”, or “*pending*”).

### Chapter 4: Analysis and Comments

Analysis of the time and space complexities of the algorithms. Comments on comparing with other known data structures and algorithms. Further possible improvements.

### References

[1] Jeremy Hylton, “The Complete Works of William Shakespeare”, <http://shakespeare.mit.edu/>

### Author List

Programmer: XXX

Tester: XXX

Reporter: XXX

### Declaration

***We hereby declare that all the work done in this project titled "*** ***Roll Your Own Mini Search Engine" is of our independent effort as a group.***

### Signatures