Roll Your Own Mini Search Engine

Changlin Zhang

Zimin Chen

Pu Yang

Date:2016-3-28

Context

Chapter 1: Introduction

1.1 Problem Description

The main purpose of our project is to build our own mini search engine to search through the Shakespeare set.

The Shakespeare set is provided on the website and we must build inverted index and write a query program so that every time we input a word or a phrase and we will receive the document ID that contain the word or phrase.

When building the inverted index, we must deal with the stop word problem, the stemming problem and equally the threshold problem when testing.

1.2 Purpose of this report

The purpose of this report is to introduce our method to solve the problem.

Firstly, we give a brief description of the problem and introduce the background of some algorithms we use in chapter 1.

Secondly, we talk about the data structures and algorithms we used in chapter 2. We will also give the pseudo code in this part.

In chapter 3, we will show our test results of each test case with specific purpose and then comes a detailed analysis.

We then analyze the time and space complexity in chapter 4. And there will be some ideas of improvement as well.

In the appendix, there comes the source code with sufficient comments.

1.3 Background

1.3.1 The Porter Stemming Algorithm

We find the porter stemming algorithm to help with solving the stemming problem.

The porter stemmer was first invented in 1979 and now it's one of the most widely used algorithm to solve the stemming problem.

The Porter stemming algorithm is a process for removing the commoner morphological and inflexional endings from words in English. Its main use is as part of a term normalization process that is usually done when setting up Information Retrieval systems. The purpose of this algorithm is to bring variant forms of a word together, not to map a word onto its "paradigm"form.

Chapter 2: Data Structure and Algorithm Specification

2.1 Data Structure

2.1.1 Open Addressing Hashing with Quadratic Probing

table 1 implementation of hash table

```
enum KindOfEntry { Legitimate, Empty, Deleted };

struct HashEntry
{
   ElementType Element;
   enum KindOfEntry Info;
};

typedef struct HashEntry Cell;
struct HashTbl
{
   int TableSize;
   Cell* TheCells;
};
```

We use an open addressing hash table with quadratic probing to implement the inverted file index.

First, we build a hash table as the dictionary. The size of the hash table will always be a prime number. For each new word to be inserted, we calculate which cell it is going to be placed with the hash function. If there is a collision, we solve the collision with quadratic probing.

table 2 Find(solve collision with quadratic probing)

```
Position Find( ElementType Key, HashTable H){
   Position CurrentPos;
   int CollisionNum;

   CollisionNum = 0;
   CurrentPos = Hash( Key, H->TableSize );
   while(      H->Cells[CurrentPos].Info      !=      Empty      &&
H->Cells[CurrentPos].Element != Key){
       CurrentPos += 2 * ++ColllisinNum -1;
       if( CurrentPos >= H->TableSize)
          CurrentPos -= H->TableSize;
   }
```

```
    return CurrentPos;
}
```

We know that if the size of the hash table is a prime number and the load factor is smaller than 0.5, we can always insert a new word into the hash table. But if there is a failure insertion, we will rehash the hash table by expanding its size to almost two times. We will make sure that the new size of the hash table is a prime number too.

table 3 Rehash

```
HashTable Rehash( HashTable H ){
   int i, OldSize;
   Cell* OldCells;
   OldCells = H->Cells;
   OldSize = H->TableSize;
   /* Get a new, empty table*/
   H = InitilizeTable( 2 * OldSize );

   /* Scan through old table, reinserting into new*/
   for( i=0; i<OldSize; i++ ){
      if( OldCells[i].Info == Legitimate )
         Insert( OldCells[i].Element, H );
   }
   free( OldCells );
}
```

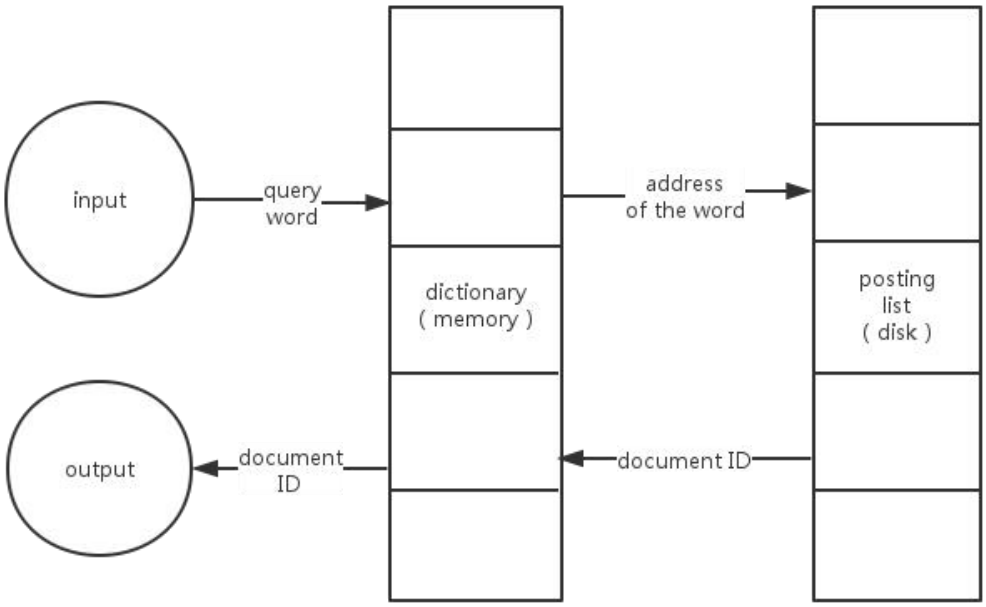
And each cell of the hash table will be stored a word and a pointer pointing to the address where the information of the word is stored in the posting list.

The information of a word in the posting list includes the ID of the documents that contain the word and equally with the frequency and position the word show up in those documents.

2.2 Algorithm Specification

2.2.1 Overall Structure

Roughly speaking, the implementation of our inverted index system can be divided into three parts: dictionary, posting list and posting file. We first search through all the documents and record all the words into the dictionary, which is stored in the memory. The dictionary stores the pointer to the posting list for each word. And then we store the posting lists of words into a posting file in the disk. The posting list for each word stores the ID of the documents containing the word and the frequency and position the word shows up.



2.2.2 Build the inverted file index

Firstly, we build our own inverted index system. We search through all the documents, and each time we read in a char. If it’s a letter, we store

the letter into a string, or we find a word and we insert the word into the dictionary and create or update its posting list.

```
Read In Word(){
    While (find_a_file() is not NULL){
        while it's not the end of a file
            while(read in a char){
                if it's punctuation or space
                    break;
                else if it's a letter
                    store in string[i++];
            }
        mark the end of a word
        insert the word into the dictionary and create or updateits posting list
        }
        close the file;
    }
}
```

We insert the word by checking whether the word has been in the hash table. If it has been in, we test whether it’s the first time for the word to appear in this document. If it is, create a file record for the word at the front of the posting list. If it isn’t, create a new offset record for the word at the front of the offset list of this document. Equally, we update the total frequency and the local frequency. If not the word is not in the hash table, we place it in a specific new cell in the hash table and create its file record in the posting list.

```
Insert X into the dictionary{
    X <- lowercase(X)
    X <- stem(X)
    Find X in the hash table
    if X is already in the hash table
        if it's the first time for X to appear in this document
            create a new file record for X in the file list
        else
            create a new offset record for X in the current file record
```

```
    else
        create a file list for X in the specific position in the hashing table
}
```

Then we are going to deal with the stop word. We count the total frequency of the word. If the total frequency is bigger than the factor we set, we see it as a stop word and delete it from the hashing table and equally insert it into the stop word table.

```
Delete Stop Word(){
    for each word in the hash table
        count the total frequency of the word
        if the frequency > a specific factor
            delete the word from the hashing table and add it into a stop word table
}
```

Then, we write out the inverted index out to the disk as a posting file.

2.2.3 Query

After building the inverted file index, we read in a query word and find it in the hash table with the hash function. We then find the right cell and find the pointer of the word pointing to the right position in the posting list.

```
Query( ){
    Read in a word or a phrase X
    if X is a word{
        X <- lowercase(X);
        if X is a stop word
            Output STOP WORD and return;
        X <- stem(X)
        if X is not in the hash table
            Output NOT FOUND and return;
        else{
            find X in the hash table
```

```
                get the pointer and go to the position of X in the posting list
                sort the document ID according to the local frequency of X in the documents
                return the ID and the local frequency of the documents
            }
        }
        Else if X is a phrase{
            Store the words in an array
            Find the first word X that is not a noisy word
                While( there is still position of X not checked ){
                  while (there is still offset of X not checked in this document){
                    while(there still words in the phrase not checked){
                        if the ith word is in the position i behind the first word
                            check the i+1 word
                        else break;
                    }
                    if all words in the phrase has been checked
                        print out the document ID
                    else
                        check the next offset of the first word
                  }
                check the next document of the first word
                }
            if no document ID has been printed out
                print NOTFOUND
        }
}
```

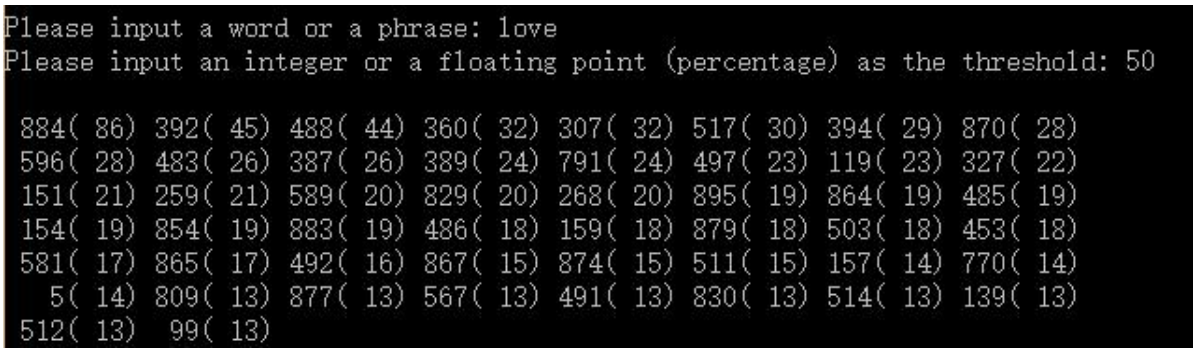
In order to solve the threshold problem, we store the frequency of the word in each document. When querying a word, we rank the document by weight, that is, the frequency and return the top N documents ID and the integer N is decided by thresholds.

Chapter 3 Testing Results

3.1 Test for words and phrase (the same threshold 50)

Before showing the testing result,there are something to know about our program.We write two different programs to deal with the stop word.The first one is to use the stop word table we found in the internet(in the test,we call it stoptable program).The second one is to judge whether a word is noisy or not by its frequency of occurrence(in the test ,we call it frequency program).We believed that if a word occur too often, the word is not so important in the documents.We make 0.0023 as the minimum standard of the stop word, and the standard is made by ourselves after a lot of test. In our test,we will line out the difference from these two ways to cope with the stop word.If we do not mark the difference it means that the result is the same in these two programs.

Case 1:Common word



Graph 1:Test for common word ("love")

PASS.

Case 2:word need to be stemmed

```
Please input a word or a phrase: loved
Please input an integer or a floating point (percentage) as the threshold: 50

 884( 86) 392( 45) 488( 44) 360( 32) 307( 32) 517( 30) 394( 29) 870( 28)
 596( 28) 483( 26) 387( 26) 389( 24) 791( 24) 497( 23) 119( 23) 327( 22)
 151( 21) 259( 21) 589( 20) 829( 20) 268( 20) 895( 19) 864( 19) 485( 19)
 154( 19) 854( 19) 883( 19) 486( 18) 159( 18) 879( 18) 503( 18) 453( 18)
 581( 17) 865( 17) 492( 16) 867( 15) 874( 15) 511( 15) 157( 14) 770( 14)
   5( 14) 809( 13) 877( 13) 567( 13) 491( 13) 830( 13) 514( 13) 139( 13)
 512( 13)  99( 13)
```

Graph 2:Test for word need to be stemmed("loved")

The expected output should be the same as the word love, and the actual output is the same to the expected one.

PASS.

Case 3:Word with capital letter

```
Please input a word or a phrase: Love
Please input an integer or a floating point (percentage) as the threshold: 50

 884( 86) 392( 45) 488( 44) 360( 32) 307( 32) 517( 30) 394( 29) 870( 28)
 596( 28) 483( 26) 387( 26) 389( 24) 791( 24) 497( 23) 119( 23) 327( 22)
 151( 21) 259( 21) 589( 20) 829( 20) 268( 20) 895( 19) 864( 19) 485( 19)
 154( 19) 854( 19) 883( 19) 486( 18) 159( 18) 879( 18) 503( 18) 453( 18)
 581( 17) 865( 17) 492( 16) 867( 15) 874( 15) 511( 15) 157( 14) 770( 14)
   5( 14) 809( 13) 877( 13) 567( 13) 491( 13) 830( 13) 514( 13) 139( 13)
 512( 13)  99( 13)
```

Graph 3:Test for word with capital letter(“Love”)

PASS.

Case 4:Noisy word

```
Please input a word or a phrase: a
Please input an integer or a floating point (percentage) as the threshold: 50
Noisy Word!
```

Graph 4:Test for noisy word("a")

```
Please input a word or a phrase: younger
Please input an integer or a floating point (percentage) as the threshold:

 772(  2) 771(  2) 140(  2) 118(  2)  10(  2) 590(  1) 538(  1) 528(  1)
 499(  1) 388(  1) 371(  1) 360(  1) 326(  1) 265(  1) 259(  1) 258(  1)
 245(  1) 220(  1) 190(  1) 151(  1) 145(  1) 854(  1) 125(  1) 122(  1)
 769(  1) 101(  1)  70(  1)  13(  1) 591(  1)   8(  1)
```

Graph 5:Test for noisy word in the stoptable program("younger")

```
Please input a word or a phrase: younger
Please input an integer or a floating point (percentage) as the threshold: 50
Noisy Word!
```

Graph 6:Test for noisy word in the frequency program("younger")

We can see that,the word “younger” occurs not so frequently in the document, so the frequency program doesn’t treat the word as a noisy word, but the stoptable program does.

PASS.

Case 5:Non-existent word

```
Please input a word or a phrase: Peyton
Please input an integer or a floating point (percentage) as the threshold: 50
Not Found!
```

Graph 7:Test for non-existent word("Peyton")

PASS

Case 6:Common phrase

```
Please input a word or a phrase: prince henry
Please input an integer or a floating point (percentage) as the threshold: 50
   341    340     59     52     50     33     17     16     15     13
    10      8      6      4      1
```

Graph 8:Test for common phrase("prince henry")

PASS.

Case 7:Phrase with punctuation

```
Please input a word or a phrase: prince-henry
Please input an integer or a floating point (percentage) as the threshold: 50
   341    340     59     52     50     33     17     16     15     13
    10      8      6      4      1
```

Graph 9:Test for phrase with punctuation("prince-henry")

In our program, if there is one or more punctuation in the query word or phrase,we will delete the punctuation and treat it as a normal phrase.So result of the test case is the same as the last one.

PASS.

Case 8:Phrase with capital letter

```
Please input a word or a phrase: PRINCE HENRY
Please input an integer or a floating point (percentage) as the threshold: 50
   341    340     59     52     50     33     17     16     15     13
    10      8      6      4      1
```

Graph 10:Test for phrase with capital letter("PRINCE HENRY")

PASS.

Case 9:Phrase with one noisy word

```
Please input a word or a phrase: the rusty curb
Please input an integer or a floating point (percentage) as the threshold: 50
Noisy Word!
      1
```

Graph 11:Test for phrase with one noisy word(the rusty curb)

```
Please input a word or a phrase: a rusty curb
Please input an integer or a floating point (percentage) as the threshold: 50
Noisy Word!
      1
```

Graph 12:Test for phrase with one noisy word(a rusty curb)

```
Please input a word or a phrase: rusty a curb
Please input an integer or a floating point (percentage) as the threshold: 50
Noisy Word!
Not Found!
```

Graph 13:Test for phrase with one noisy word(rusty a curb)

In our program, if there is one or more noisy word in the query phrase, we'll in the first replace the noisy word with a spacing to mark where the word appear in the phrase, and the relative position of the interesting words.Then we go to the index to find where the first interesting word is and whether the next interesting word in the phrase is in the right relative position of the found one.So if we query "the rusty curb" and "a rusty curb", the output should be same but different form the phrase "rusty a curb".

PASS.

Case 10:Phrase with more noisy words

```
Please input a word or a phrase: run out of time
Please input an integer or a floating point (percentage) as the threshold: 50
Noisy Word!
Noisy Word!
     91
```

Graph 14:Test for phrase with more noisy word

The repeat time of “Noisy Word!” means how many time the noisy words appear in your query.

PASS.

Case 11:Non-existent phrase

```
Please input a word or a phrase: DS CLASS
Please input an integer or a floating point (percentage) as the threshold: 50
Not Found!
Not Found!
```

Graph 15:Test for non-existent phrase("DS CLASS")

```
Please input a word or a phrase: ADS class
Please input an integer or a floating point (percentage) as the threshold: 50
Not Found!
    887    884    836    818    814    809    793    778    748    703
    617    598    589    556    547    539    430    414    394    393
    391    389    388    363    355    354    310    304    234    233
     58
```

Graph 16:Test for non-exist phrase("ADS class")

If the query phrase is non-existent, at the first the program will output a ”Not Found” to make people know the phrase isn’t existent.After that, there are two situations.The first one is that the word in the phrase is non-exist in the document, then it output the second “Not Found”.The second situation is that there are some words exists in the document, then the program will regard the non-exist word as a noisy word and execute the query.

PASS.

3.2 Test for threshold

In our program, we make two kinds of threshold.The first one is the number of output record, and the second one is the proportion of the output record to the total record.In this part we take “love”as the word appears frequently ,“boy” as the word appears occasionally and “rusty” as the word rarely appears .

3.2.1.Use the number of output record as the threshold

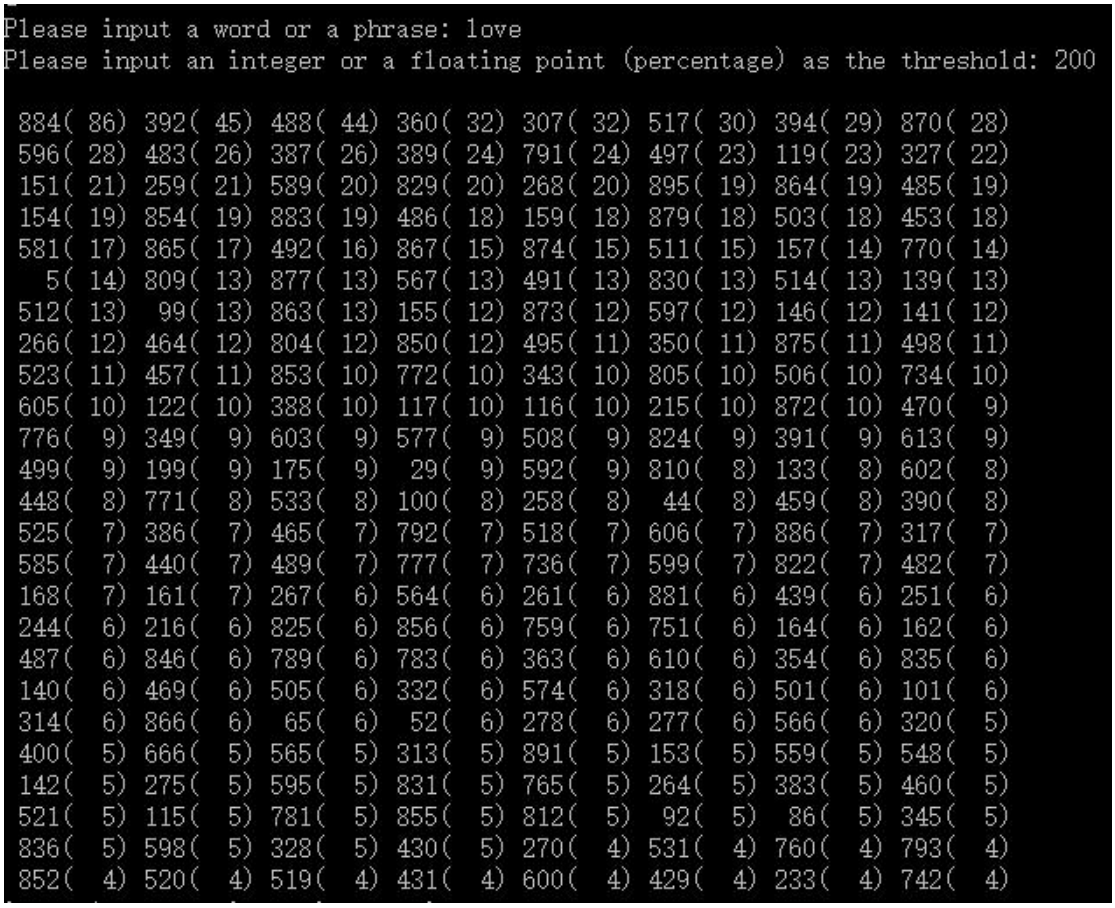
Case 1:Word “love”

```
Please input a word or a phrase: love
Please input an integer or a floating point (percentage) as the threshold: 20

 884( 86) 392( 45) 488( 44) 360( 32) 307( 32) 517( 30) 394( 29) 870( 28)
 596( 28) 483( 26) 387( 26) 389( 24) 791( 24) 497( 23) 119( 23) 327( 22)
 151( 21) 259( 21) 589( 20) 829( 20)
```

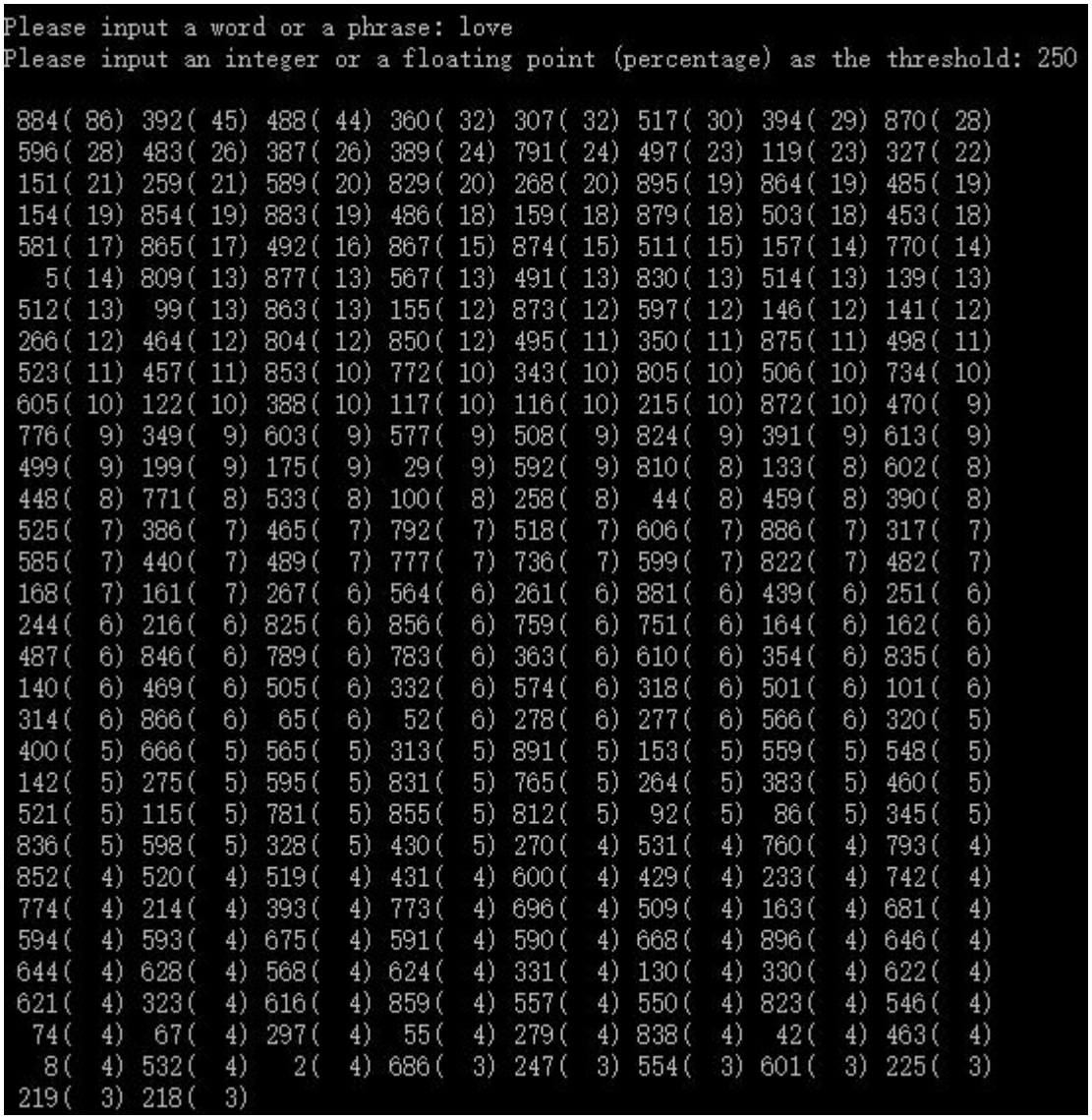
Graph 17:Test for threshold for word "love"(20)

```
Please input a word or a phrase: love
Please input an integer or a floating point (percentage) as the threshold: 100

 884( 86) 392( 45) 488( 44) 360( 32) 307( 32) 517( 30) 394( 29) 870( 28)
 596( 28) 483( 26) 387( 26) 389( 24) 791( 24) 497( 23) 119( 23) 327( 22)
 151( 21) 259( 21) 589( 20) 829( 20) 268( 20) 895( 19) 864( 19) 485( 19)
 154( 19) 854( 19) 883( 19) 486( 18) 159( 18) 879( 18) 503( 18) 453( 18)
 581( 17) 865( 17) 492( 16) 867( 15) 874( 15) 511( 15) 157( 14) 770( 14)
   5( 14) 809( 13) 877( 13) 567( 13) 491( 13) 830( 13) 514( 13) 139( 13)
 512( 13)  99( 13) 863( 13) 155( 12) 873( 12) 597( 12) 146( 12) 141( 12)
 266( 12) 464( 12) 804( 12) 850( 12) 495( 11) 350( 11) 875( 11) 498( 11)
 523( 11) 457( 11) 853( 10) 772( 10) 343( 10) 805( 10) 506( 10) 734( 10)
 605( 10) 122( 10) 388( 10) 117( 10) 116( 10) 215( 10) 872( 10) 470(  9)
 776(  9) 349(  9) 603(  9) 577(  9) 508(  9) 824(  9) 391(  9) 613(  9)
 499(  9) 199(  9) 175(  9)  29(  9) 592(  9) 810(  8) 133(  8) 602(  8)
 448(  8) 771(  8) 533(  8) 100(  8)
```

Graph 18:Test for threshold for word "love"(100)



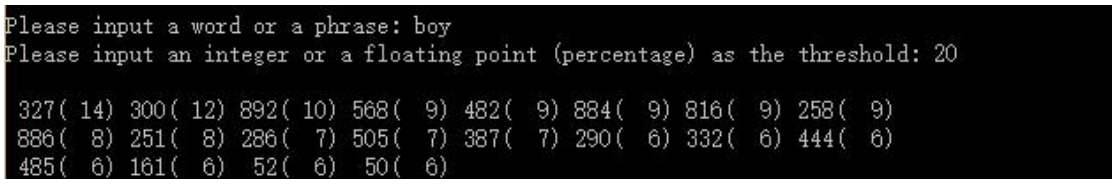
Graph 19:Test for threshold for word "love"(200)



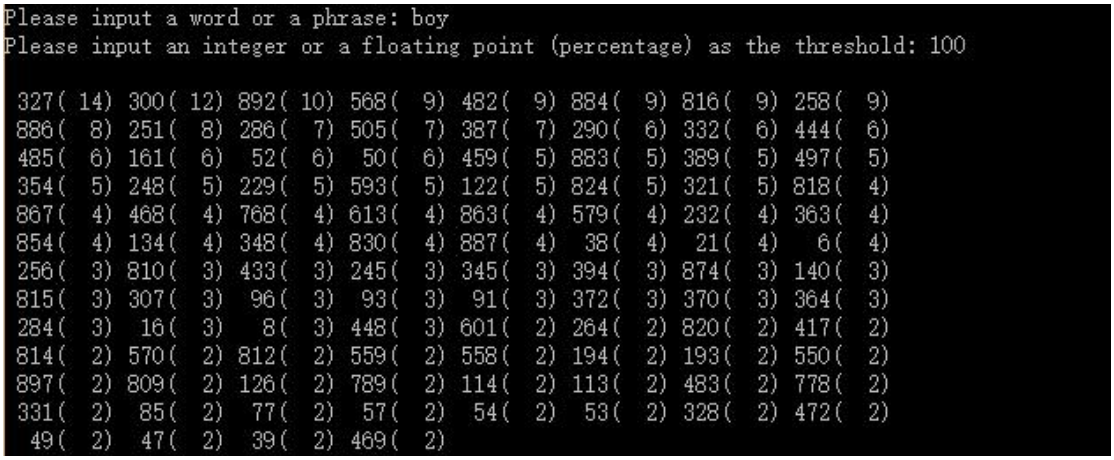
Graph 20:Test for threshold for word "love"(250)

PASS.

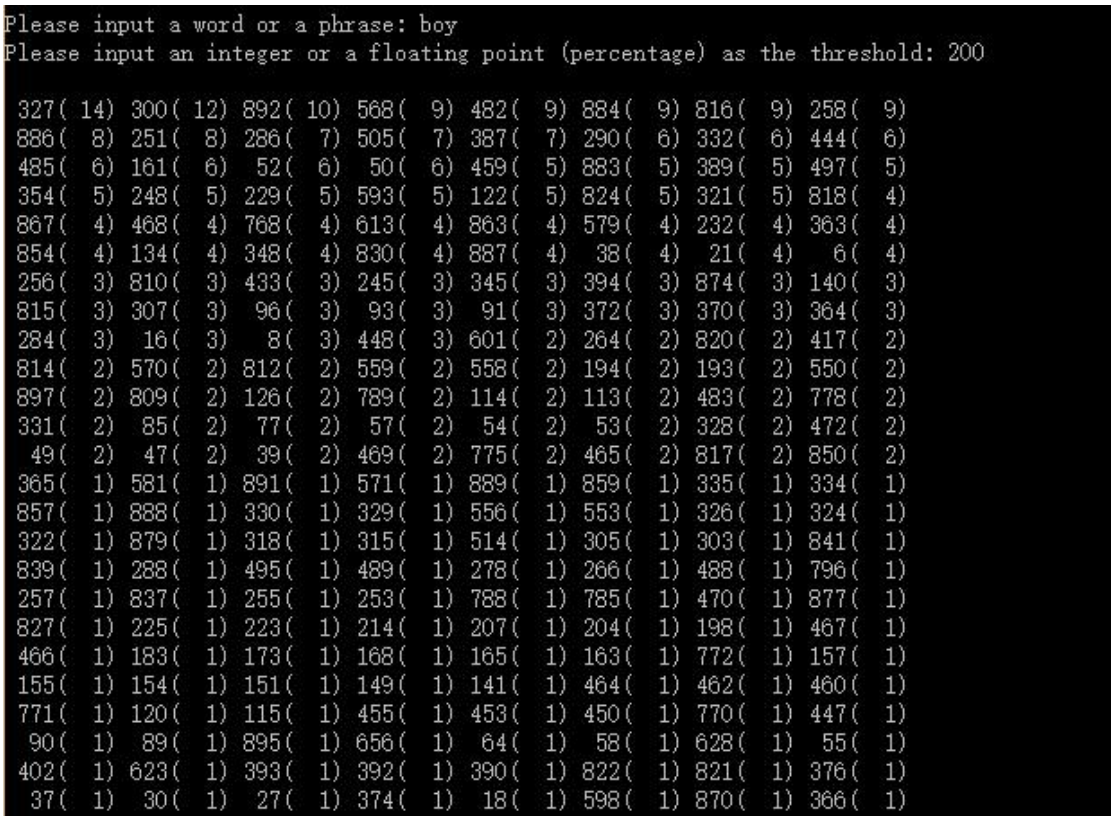
Case 2:Word “boy”



Graph 21:Test for threshold for word "boy"(20)



Graph 22:Test for the threshold for word "boy"(100)



Graph 23:Test for the threshold word "boy" (200)

```
Please input a word or a phrase: boy
Please input an integer or a floating point (percentage) as the threshold: 250

 327( 14) 300( 12) 892( 10) 568(  9) 482(  9) 884(  9) 816(  9) 258(  9)
 886(  8) 251(  8) 286(  7) 505(  7) 387(  7) 290(  6) 332(  6) 444(  6)
 485(  6) 161(  6)  52(  6)  50(  6) 459(  5) 883(  5) 389(  5) 497(  5)
 354(  5) 248(  5) 229(  5) 593(  5) 122(  5) 824(  5) 321(  5) 818(  4)
 867(  4) 468(  4) 768(  4) 613(  4) 863(  4) 579(  4) 232(  4) 363(  4)
 854(  4) 134(  4) 348(  4) 830(  4) 887(  4)  38(  4)  21(  4)   6(  4)
 256(  3) 810(  3) 433(  3) 245(  3) 345(  3) 394(  3) 874(  3) 140(  3)
 815(  3) 307(  3)  96(  3)  93(  3)  91(  3) 372(  3) 370(  3) 364(  3)
 284(  3)  16(  3)   8(  3) 448(  3) 601(  2) 264(  2) 820(  2) 417(  2)
 814(  2) 570(  2) 812(  2) 559(  2) 558(  2) 194(  2) 193(  2) 550(  2)
 897(  2) 809(  2) 126(  2) 789(  2) 114(  2) 113(  2) 483(  2) 778(  2)
 331(  2)  85(  2)  77(  2)  57(  2)  54(  2)  53(  2) 328(  2) 472(  2)
  49(  2)  47(  2)  39(  2) 469(  2) 775(  2) 465(  2) 817(  2) 850(  2)
 365(  1) 581(  1) 891(  1) 571(  1) 889(  1) 859(  1) 335(  1) 334(  1)
 857(  1) 888(  1) 330(  1) 329(  1) 556(  1) 553(  1) 326(  1) 324(  1)
 322(  1) 879(  1) 318(  1) 315(  1) 514(  1) 305(  1) 303(  1) 841(  1)
 839(  1) 288(  1) 495(  1) 489(  1) 278(  1) 266(  1) 488(  1) 796(  1)
 257(  1) 837(  1) 255(  1) 253(  1) 788(  1) 785(  1) 470(  1) 877(  1)
 827(  1) 225(  1) 223(  1) 214(  1) 207(  1) 204(  1) 198(  1) 467(  1)
 466(  1) 183(  1) 173(  1) 168(  1) 165(  1) 163(  1) 772(  1) 157(  1)
 155(  1) 154(  1) 151(  1) 149(  1) 141(  1) 464(  1) 462(  1) 460(  1)
 771(  1) 120(  1) 115(  1) 455(  1) 453(  1) 450(  1) 770(  1) 447(  1)
  90(  1)  89(  1) 895(  1) 656(  1)  64(  1)  58(  1) 628(  1)  55(  1)
 402(  1) 623(  1) 393(  1) 392(  1) 390(  1) 822(  1) 821(  1) 376(  1)
  37(  1)  30(  1)  27(  1) 374(  1)  18(  1) 598(  1) 870(  1) 366(  1)
   4(  1)
```

Graph 24:Test for the threshold for word "boy"(250)

PASS

Case 3:Word “rusty”

```
Please input a word or a phrase: rusty
Please input an integer or a floating point (percentage) as the threshold: 20

 831(  1) 825(  1) 774(  1) 553(  1) 530(  1) 529(  1) 298(  1) 266(  1)
   1(  1)
```

Graph 25:Test for the threshold for word "rusty"(20)

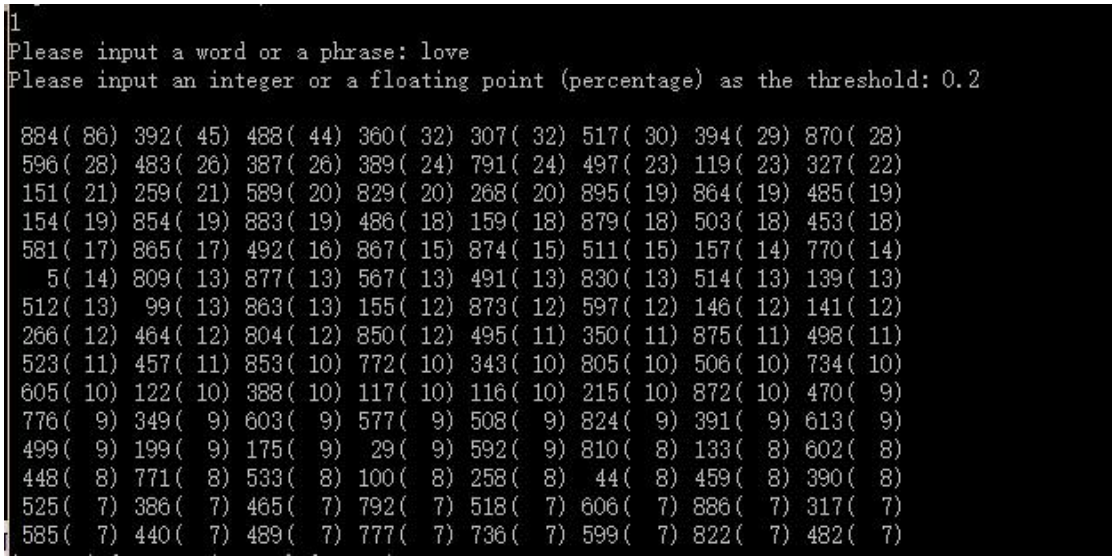
```
Please input a word or a phrase: rusty
Please input an integer or a floating point (percentage) as the threshold: 100

 831(  1) 825(  1) 774(  1) 553(  1) 530(  1) 529(  1) 298(  1) 266(  1)
   1(  1)
```

Graph 26:Test for the threshold for word "rusty"(100)

PASS.

3.2.2the proportion of the output record to the total record as the threshold

Case 1:Word “love”



Graph 27:Test for the threshold for word "love"(0.2)

```
Please input a word or a phrase: love
Please input an integer or a floating point (percentage) as the threshold: 0.8

884( 86) 392( 45) 488( 44) 360( 32) 307( 32) 517( 30) 394( 29) 870( 28)
596( 28) 483( 26) 387( 26) 389( 24) 791( 24) 497( 23) 119( 23) 327( 22)
151( 21) 259( 21) 589( 20) 829( 20) 268( 20) 895( 19) 864( 19) 485( 19)
154( 19) 854( 19) 883( 19) 486( 18) 159( 18) 879( 18) 503( 18) 453( 18)
581( 17) 865( 17) 492( 16) 867( 15) 874( 15) 511( 15) 157( 14) 770( 14)
  5( 14) 809( 13) 877( 13) 567( 13) 491( 13) 830( 13) 514( 13) 139( 13)
512( 13)  99( 13) 863( 13) 155( 12) 873( 12) 597( 12) 146( 12) 141( 12)
266( 12) 464( 12) 804( 12) 850( 12) 495( 11) 350( 11) 875( 11) 498( 11)
523( 11) 457( 11) 853( 10) 772( 10) 343( 10) 805( 10) 506( 10) 734( 10)
605( 10) 122( 10) 388( 10) 117( 10) 116( 10) 215( 10) 872( 10) 470(  9)
776(  9) 349(  9) 603(  9) 577(  9) 508(  9) 824(  9) 391(  9) 613(  9)
499(  9) 199(  9) 175(  9)  29(  9) 592(  9) 810(  8) 133(  8) 602(  8)
448(  8) 771(  8) 533(  8) 100(  8) 258(  8)  44(  8) 459(  8) 390(  8)
525(  7) 386(  7) 465(  7) 792(  7) 518(  7) 606(  7) 886(  7) 317(  7)
585(  7) 440(  7) 489(  7) 777(  7) 736(  7) 599(  7) 822(  7) 482(  7)
168(  7) 161(  7) 267(  6) 564(  6) 261(  6) 881(  6) 439(  6) 251(  6)
244(  6) 216(  6) 825(  6) 856(  6) 759(  6) 751(  6) 164(  6) 162(  6)
487(  6) 846(  6) 789(  6) 783(  6) 363(  6) 610(  6) 354(  6) 835(  6)
140(  6) 469(  6) 505(  6) 332(  6) 574(  6) 318(  6) 501(  6) 101(  6)
314(  6) 866(  6)  65(  6)  52(  6) 278(  6) 277(  6) 566(  6) 320(  5)
400(  5) 666(  5) 565(  5) 313(  5) 891(  5) 153(  5) 559(  5) 548(  5)
142(  5) 275(  5) 595(  5) 831(  5) 765(  5) 264(  5) 383(  5) 460(  5)
521(  5) 115(  5) 781(  5) 855(  5) 812(  5)  92(  5)  86(  5) 345(  5)
836(  5) 598(  5) 328(  5) 430(  5) 270(  4) 531(  4) 760(  4) 793(  4)
852(  4) 520(  4) 519(  4) 431(  4) 600(  4) 429(  4) 233(  4) 742(  4)
774(  4) 214(  4) 393(  4) 773(  4) 696(  4) 509(  4) 163(  4) 681(  4)
594(  4) 593(  4) 675(  4) 591(  4) 590(  4) 668(  4) 896(  4) 646(  4)
644(  4) 628(  4) 568(  4) 624(  4) 331(  4) 130(  4) 330(  4) 622(  4)
621(  4) 323(  4) 616(  4) 859(  4) 557(  4) 550(  4) 823(  4) 546(  4)
 74(  4)  67(  4) 297(  4)  55(  4) 279(  4) 838(  4)  42(  4) 463(  4)
  8(  4) 532(  4)   2(  4) 686(  3) 247(  3) 554(  3) 601(  3) 225(  3)
219(  3) 218(  3) 549(  3) 385(  3) 685(  3) 207(  3) 780(  3) 197(  3)
179(  3) 361(  3) 769(  3) 167(  3) 484(  3) 779(  3) 778(  3) 346(  3)
529(  3) 885(  3) 338(  3) 645(  3) 755(  3) 637(  3) 149(  3) 629(  3)
833(  3) 839(  3) 455(  3) 739(  3) 450(  3) 816(  3) 128(  3) 443(  3)
310(  3) 578(  3) 620(  3) 294(  3) 108(  3) 285(  3) 432(  3) 814(  3)
722(  3)  88(  3) 721(  3)  79(  3) 273(  3) 406(  3) 404(  3)  61(  3)
 58(  3) 718(  3) 701(  3) 265(  3) 604(  3)  33(  3) 262(  3)  13(  3)
563(  3) 561(  3) 795(  3) 640(  2) 749(  2) 462(  2) 636(  2) 562(  2)
635(  2) 633(  2) 748(  2) 451(  2) 847(  2) 552(  2) 627(  2) 442(  2)
441(  2) 625(  2) 858(  2) 623(  2) 246(  2) 544(  2) 241(  2) 238(  2)
537(  2) 229(  2) 228(  2) 227(  2) 857(  2) 224(  2) 223(  2) 428(  2)
425(  2) 217(  2) 424(  2) 423(  2) 414(  2) 413(  2) 205(  2) 204(  2)
201(  2) 808(  2) 733(  2) 194(  2) 193(  2) 186(  2) 401(  2) 177(  2)
731(  2) 172(  2) 528(  2) 614(  2) 165(  2) 726(  2) 611(  2) 724(  2)
878(  2) 160(  2) 837(  2) 798(  2) 515(  2) 713(  2) 709(  2) 380(  2)
379(  2) 147(  2) 372(  2) 370(  2) 368(  2) 851(  2) 699(  2) 136(  2)
697(  2) 868(  2) 353(  2) 127(  2) 126(  2) 695(  2) 687(  2) 118(  2)
819(  2) 817(  2) 768(  2) 114(  2) 109(  2) 849(  2) 334(  2) 496(  2)
671(  2)  97(  2)  96(  2) 763(  2) 786(  2) 661(  2)  85(  2) 326(  2)
325(  2)  69(  2) 659(  2) 321(  2) 579(  2)  59(  2) 653(  2)  56(  2)
647(  2) 316(  2)  51(  2) 575(  2) 785(  2) 312(  2) 571(  2)  27(  2)
 16(  2) 473(  2)  11(  2) 832(  2)   7(  2)   6(  2) 813(  2) 643(  2)
271(  1) 797(  1) 828(  1) 641(  1) 775(  1) 449(  1) 639(  1) 447(  1)
446(  1) 260(  1) 445(  1) 558(  1) 257(  1) 254(  1) 253(  1) 252(  1)
794(  1) 250(  1) 248(  1) 556(  1) 843(  1) 245(  1) 553(  1) 435(  1)
240(  1) 434(  1) 237(  1) 234(  1) 433(  1) 232(  1) 230(  1) 834(  1)
634(  1) 725(  1) 226(  1) 631(  1) 547(  1) 811(  1) 221(  1) 220(  1)
545(  1) 790(  1) 422(  1) 417(  1) 416(  1) 542(  1) 213(  1) 210(  1)
209(  1) 539(  1) 410(  1) 409(  1) 407(  1) 200(  1) 840(  1) 626(  1)
```

Graph 28:Test for threshold for "love"(0.8)

PASS.

Case 2:Word “boy”

```
Please input a word or a phrase: boy
Please input an integer or a floating point (percentage) as the threshold: 0.2

327( 14) 300( 12) 892( 10) 568(  9) 482(  9) 884(  9) 816(  9) 258(  9)
886(  8) 251(  8) 286(  7) 505(  7) 387(  7) 290(  6) 332(  6) 444(  6)
485(  6) 161(  6)  52(  6)  50(  6) 459(  5) 883(  5) 389(  5) 497(  5)
354(  5) 248(  5) 229(  5) 593(  5) 122(  5) 824(  5) 321(  5) 818(  4)
867(  4) 468(  4) 768(  4) 613(  4) 863(  4) 579(  4) 232(  4) 363(  4)
854(  4)
```

Graph 29:Test for threshold "boy"(0.2)

```
Please input a word or a phrase: boy
Please input an integer or a floating point (percentage) as the threshold: 0.8

327( 14) 300( 12) 892( 10) 568(  9) 482(  9) 884(  9) 816(  9) 258(  9)
886(  8) 251(  8) 286(  7) 505(  7) 387(  7) 290(  6) 332(  6) 444(  6)
485(  6) 161(  6)  52(  6)  50(  6) 459(  5) 883(  5) 389(  5) 497(  5)
354(  5) 248(  5) 229(  5) 593(  5) 122(  5) 824(  5) 321(  5) 818(  4)
867(  4) 468(  4) 768(  4) 613(  4) 863(  4) 579(  4) 232(  4) 363(  4)
854(  4) 134(  4) 348(  4) 830(  4) 887(  4)  38(  4)  21(  4)   6(  4)
256(  3) 810(  3) 433(  3) 245(  3) 345(  3) 394(  3) 874(  3) 140(  3)
815(  3) 307(  3)  96(  3)  93(  3)  91(  3) 372(  3) 370(  3) 364(  3)
284(  3)  16(  3)   8(  3) 448(  3) 601(  2) 264(  2) 820(  2) 417(  2)
814(  2) 570(  2) 812(  2) 559(  2) 558(  2) 194(  2) 193(  2) 550(  2)
897(  2) 809(  2) 126(  2) 789(  2) 114(  2) 113(  2) 483(  2) 778(  2)
331(  2)  85(  2)  77(  2)  57(  2)  54(  2)  53(  2) 328(  2) 472(  2)
 49(  2)  47(  2)  39(  2) 469(  2) 775(  2) 465(  2) 817(  2) 850(  2)
365(  1) 581(  1) 891(  1) 571(  1) 889(  1) 859(  1) 335(  1) 334(  1)
857(  1) 888(  1) 330(  1) 329(  1) 556(  1) 553(  1) 326(  1) 324(  1)
322(  1) 879(  1) 318(  1) 315(  1) 514(  1) 305(  1) 303(  1) 841(  1)
839(  1) 288(  1) 495(  1) 489(  1) 278(  1) 266(  1) 488(  1) 796(  1)
257(  1) 837(  1) 255(  1) 253(  1) 788(  1) 785(  1) 470(  1) 877(  1)
827(  1) 225(  1) 223(  1) 214(  1) 207(  1) 204(  1) 198(  1) 467(  1)
466(  1) 183(  1) 173(  1) 168(  1) 165(  1) 163(  1) 772(  1) 157(  1)
155(  1)
```

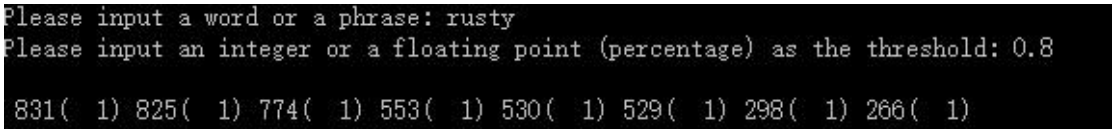
Graph 30:Test for the threshold "boy"(0.8)

PASS.

Case 3:Word “rusty”

```
Please input a word or a phrase: rusty
Please input an integer or a floating point (percentage) as the threshold: 0.2

831(  1) 825(  1)
```

Graph 31:Test for the threshold "rusty"(0.2)



Graph 32:Test for the threshold "rusty"(0.8)

PASS.

3.2.3 Summary

We assume that for the word “love”,if it occurs more than 5 times(include 5) in a file,then the file relative.As for the word ”boy”,when it occurs more than 2 times(include 2),it is a relative file.And for word “rusty” all the word should be relative according to these three words’ occurent frequency.

table 4:conclusion of word "love"

items / threshold	The count of output files	The count of relative files	precision	recall
20	20	188	100%	10.6%
50	50	188	100%	26.6%
100	100	188	100%	53.2%
150	150	188	100%	79.8%
200	200	188	94%	100%
250	250	188	75.2%	100%
20%	120	188	100%	95.7%
40%	240	188	78.3%	100%
60%	360	188	52.2%	100%
80%	480	188	39.2%	100%
100%	599	188	31.4%	100%

Graph 33:Conclusion of word "boy"

items / threshold	The count of output files	The count of relative files	precision	recall
20	20	104	100%	19.2%
50	50	104	100%	48.1%
100	100	104	100%	96.2%
150	150	104	69.3%	100%
200	200	104	52%	100%
250	201	104	51.7%	100%
20%	41	104	100%	35.4%
40%	81	104	100%	77.9%
60%	121	104	86.0%	100%
80%	153	104	68.0%	100%
100%	201	104	51.7%	100%

Graph 34:Conclusion of word “rusty”

items / threshold	The count of output files	The count of relative files	precision	recall
20	9	9	100%	100%
50	9	9	100%	100%
100	9	9	100%	100%
150	9	9	100%	100%
200	9	9	100%	100%
250	9	9	100%	100%
20%	2	9	100%	22.2%
40%	4	9	100%	44.4%
60%	6	9	100%	66.7%
80%	8	9	100%	88.9%
100%	9	9	100%	100%

Chapter 4:Analysis and Comments

4.1 Time and space complicity

The overall analysis of time and space complexity can be divided into three parts: reading in the files, making the index, and making queries, corresponding respectively to the procedures Read,Insert,Query.

4.1.1Read

If we let C denote the total number of characters in all the N files, then the time and space complexity for Read are both $\Theta(C)$.

4.1.2 Insert

The Insert can be divided into two parts: finding the place for the record and updating the record.Since the files are read in sequentially,for each word, the file ID are in non-decreasing order. So the updating of a record can both be done in O(1) time. The space spent on storing records is obviously $\Theta(P)$,with P denoting total number of positions inserted for all the words in the index.The time and space complexity of finding the correct place to insert the record is decided by the structure of hashing .

In our implementation,we let the load factor never exceed 0.5. To reduce the number of rehashing and save space,quadratic probing is

used and the table size is always a prime number. According to the theorem in our textbook[1],if quadratic probing is used,and the table size is prime,then a new element can always be inserted if the table is at least half empty. So the time of finding the correct place for one word without rehashing is O(1).

Every time we rehash,we approximately double the table size,therefore, the space needed for the hash table alone,is Θ (W) (W denote the number of words indexed). And now we will show the prove of the time complicity of the rehashing is O(W).If the hash table has been rehashed for k times. Let T1,...,Tk denote the table size after the ith rehashing, with $1 \leqslant i \leqslant k$, T0 denote the original table size, and S denote the total number of insertions spent on rehashing. According to the above analysis, we have $2Ti-1 \leqslant Ti$ for $1 \leqslant i \leqslant k$, and $S \leqslant \sum_{i=0}^{k-1} T_i \leqslant T_k(\sum_{i=1}^{k} \frac{1}{2^i})$ < T_k, and the time consumed for initialization is also $O(\sum_{i=0}^{k-1} T_i) = O(T_k)$. Since T_k is Θ (W), the time spent on rehashing is $O(T_k)$ = O(W).

So the total time complicity of insert should be O(W+P)=O(W),and the space consuming is Θ (W+P)= Θ (P).

4.1.3 Query

The process of query can be divided to two steps: the first one is to find the word by hashing function,and the second step is to print the file

ID and its frequency in the file in decreasing order.

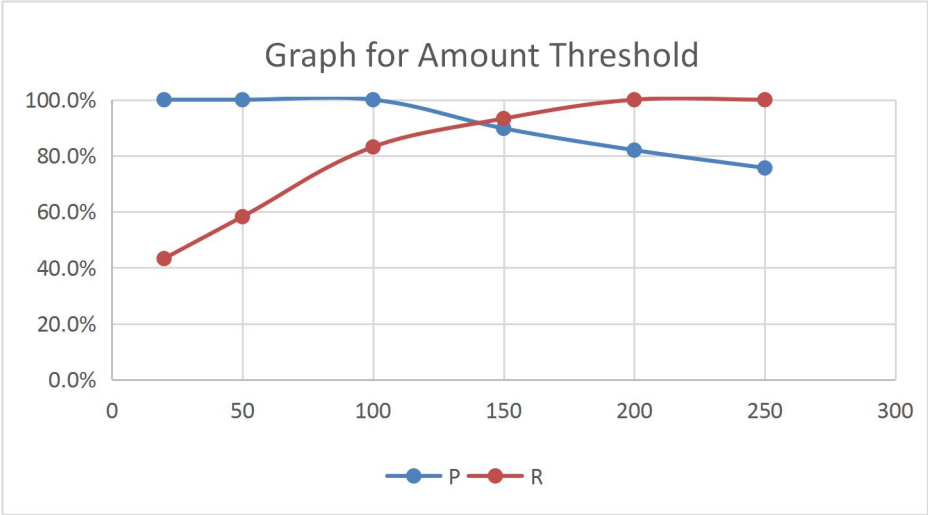
It's obvious that the time complexity of finding a word by hashing function is O(1).Then we should sort the index record of this word in decreasing order. The time spent in this step is determined by the number of the total number of position inserted into this word which is denoted as Pn,and the time complexity should be O(Pn2).

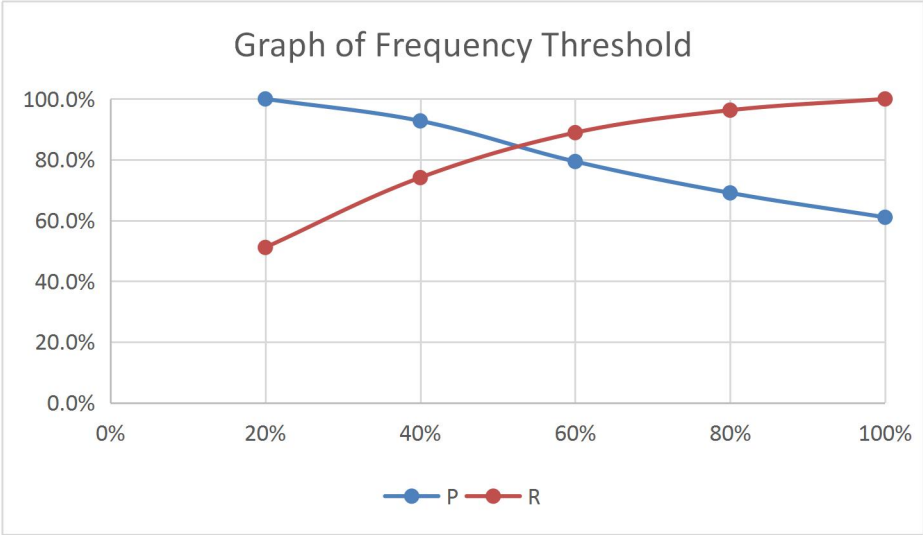
4.2 Performance of threshold and stop word

4.2.1 Threshold

table 5:The Statistics of Threshold

Items / Threshold	Average precision	Average recall
20	100.0%	43.3%
50	100.0%	58.2%
100	100.0%	83.1%
150	89.8%	93.3%
200	82.0%	100.0%
250	75.6%	100.0%
20%	100.0%	51.1%
40%	92.8%	74.1%
60%	79.4%	88.9%
80%	69.1%	96.3%
100%	61.0%	100.0%





We have shown our two kinds of threshold in the above part,and have done some tests(for simplicity, we call the threshold using the number of output record as amount threshold and call the threshold using the proportion of the output record to the total record as proportion threshold). Now we're going to tell some advantages and disadvantages about these two thresholds .

For the proportion threshold,it has characteristic as universality.It means that even we don't know the total words in the documents or how frequency a word occurs ,we can roughly decide the threshold.

Taking word “love” and “boy” as examples, considering the occurence frequency, if we take 0.6 as the threshold, we can roughly elect the relative files to these two words according to the test without lots of junks .(However if the standard of relativity is changed, the threshold should be also changed)But there are also some disadvantages that if a word rarely appear, then all the records are important and relative.But with certain proportion threshold,it’s impossible for the search engine to elect all the relative file,the recall can be really low.So,we think that this kind of threshold is suitable for the documents with a great amount of words.It can improve the precision which seemed more important than recall in the documents with lots of words.

As for the amount threshold,contrary to the proportion threshold,it’s more appropriate to the documents contain less words.Because this kind of documents do not contain a lot of junk, we can reduce the precision to improve the recall.

4.2.2 Stop word

In this part we’d like to discuss the two ways to deal with the stop word.

The way that we find the stop word by ourselves has characteristic as universality.It means that if we have enough data we can get a more accurate stop table for our own documents than the other way.But the

judge of the stop words may be seriously affected by the minimum of stop word frequency.The parameter in our program is decided after a lot test, but it may not be so accurate for we can only take consideration of limited words.

4.3 Improvements

4.3.1 Time and Space

Some improvements on time and space of our solution is possible. If we use hashing, better hash functions can be used to achieve better distribution of words.Also,we had thought to make our engine in other ways using trie. But due to the limit of time and ability, and we also consider that if our data is not big enough, using hashing may be the better way to finish the project, for the trie may use more space than hashing but the improvement of time may not be so evident.And if we consider other data structures such as AVL tree,B-Tree,there may be a better way to deal with the problem of searching engine.

4.3.2 Big Data

Though,in our program we have a function to send the index to the CRT if the film number is bigger than 1000.We haven’t write the code to realize the function to query in the CRT.

4.3.3 Parameter

Our parameter of stop word and threshold may not be so accurate.If more time is given, we can make more elaborate experiment to decide the parameter and make the program not only universal but also accurate.

4.3.4 Threshold for Phrase

In our program,threshold for phrase is not completed yet. It means that the output phrase are not given in the decreasing-order of frequency which can be further improved.

Appendix: Source Code

Version 1:The search engine for large data and deal with the stop word with stop word frequency

```
/*
How to use:
Please change the DATAPATH in define module according to where you put data in.

In general, what this program do are:
    Get words from files.
 -> Filter and sort words into terms.
 -> Record occurences of terms in table.
 -> Search word of query.
*/

/*
Header file declaration.
*/
```

```
#include "stdio.h"
#include "stdlib.h"
#include <string.h>
#include <math.h>
#include <ctype.h>
#include "io.h"
#include "direct.h" //You can use "direct.h" or "dir.h" depengding on environment.
#include "stem.h"
#include <windows.h>

/*
Define constant.
*/
#define DATAPATH ".\\data"
#define STOPWORDFREQUENCY 0.0023//Parameter to find out a stop word.
#define STOPWORDSTART 1000000 //Parameter to find out a stop word.
#define StopWordNumber 50000 //Max number of stop words.
#define MaxFileNumber 900 //Max number of files.
#define FILENAMELENGTH 30 //Max length of file name.
#define BufferSize 500 //Size of buffer.
#define WordLength 30 //Suppose words are not longer than 30.
#define max_length 50 // Suppose input word or phrase are no longer than 50
#define max_num 10    // Suppose the number of words in a phrase are no longer than 10
#define Legitimate 1 //State of TableNode: Legitimate.
#define Empty 0  //State of TableNode: Empty.

/*
Global variable declaration.
*/
int TotalWordCount=0; //Number of total words.
int StopWordNo=0;
int PrimeNumberTable[] = { 10159,20533,41233,82471,165719,348937,700001,1400017 }; //Table
of prime numbers.
char StopWordTable[StopWordNumber][WordLength];

/*
Structure declaration.
*/
typedef struct FileNode* File;
struct FileNode {
    FILE* Location; //Pointer to where the file is.
    int FileNumber; //Position of file in a database.
};
```

```
typedef struct  OffsetNode* Offset;
struct OffsetNode {
    int OffsetInFile; //Location in the file.
    Offset NextOffset; //Pointer to next Location of the term.
};

typedef struct RecordNode* Record;
struct RecordNode {
    int LocalFrequency; //Occurrence number of this term in the file.
    int HostFile; //Numeber of the file it belongs to.
    Offset FileOffset; //Location in the file.
    Record NextRecord; //Pointer to next record of the term.
};

typedef struct TermNode* Term;
struct TermNode {
    char Content[WordLength]; //Term as string.
    int TotalFrequency; //Occurrence number of this term in the database.
    int TotalFiles;           //Occurrence number of files which contain this term
    Record RecordList; //Pointer to a list of its appearance records.
};

typedef struct TableNode* Table;
struct TableNode {
    int State; //Mark whether it is empty or not.
    Term Word; //Term in this node.
};

/*
Functions declaration.
*/
extern int stem(char * p, int i, int j); //Get the stem of a word.
char* WordToTerm(char* word); //Pre-process word into term
int IsStopWord(char* word); //Determine whether it is a stop word
int Hashing(char* word, int* TotalTableSize); //Hash function.
void Insert(Table** Table_1, char* word, int fileNo, int lineNo, int* CurrentTableSize, int*
TotalTableSize, int* WhichPrime);  //Insert an input.
int Find(char* word, Table** Table_1, int* TotalTableSize, int* WhichPrime); //Find the
position of a word.
void Rehashing(Table** Table_1, int* TotalTableSize, int* WhichPrime); //Rehashing when
half of the table is full.
void my_strcpy(char* s1, char* s2, int m, int n);//string copy operation
void sort(Record *List, int num);  //sort the document ID by the frequency of the term
Term Query_word(char *word, Table** Table_1, int* TotalTableSize, int* WhichPrime);
```

```
    //find the proper position of a term in the data base
void Query(char* word, float threshold, Table** Table_1, int* TotalTableSize, int*
WhichPrime); //solve 2 different situations of query for both word and phrases
void WriteOut(Table** Table_1, int* TotalTableSize,char* filename); //Write out current
table into hard disk.
void DeleteTable(Table** Table_1, int* TotalTableSize); //Delete table to free space.
void DeleteStopWord(Table **Table_1,int pos); //Delete stop word from table.

/*
Pre-process word into term.
Functions as follow:
1、If it is upper case, transform it to lower case.
2、Get the stem of a word.
*/
char* WordToTerm(char* word) { //Pre-process word into term
    int i;

    for (i = 0; (word[i] != '\0') && (i < 30); i++) {
        if ((word[i] >= 'A') && (word[i] <= 'Z')) {
            word[i] = word[i] - 'A' + 'a'; //upper case->lower case.
        }
    }

    i--;
    i = stem(word, 0, i);
    word[i + 1] = '\0'; //Get the stem of a word.
    return word;
}

/*
Find out whether it is a stop word.
Way to do that:
Search the word in stopWordTable by binary search since the stopWordTable is sorted.
*/
int IsStopWord(char* word) {
    int i;
    for(i=0;i<StopWordNo;i++){
        if(strcmp(StopWordTable[i], word)==0){
            return 1;
        }
    }
    return 0;
}
```

```
/*
Get the key of hash.
Way to do that:
Using quadratic probing.
*/
int Hashing(char* word, int* TotalTableSize) {
    int result = 0, i;

    //the algorithm to calculate the hash value
    for (i = 0; (word[i] != '\0') && (i < 30); i++) {
        result = 32 * result + word[i];//Using 32 as weight of a char,
                                       //and multipling 32 is indeed shifting bits, which
is fast in hardware.
    }
    result = result % (*TotalTableSize);
    if (result < 0) { //Overflow may make the result negative, so here is a makeup.
        result = result + (*TotalTableSize);
    }
    return result;
}

/*
Record one time ocurrence of word.
Possible situations:
1、It is a new word.
2、It is already in table but first time in this file.
3、It is already in table and also in this file.
*/
void Insert(Table** Table_1, char* word, int fileNo, int WordOffset, int* CurrentTableSize,
int* TotalTableSize, int* WhichPrime) {
    int pos,flag = 1;
    Record CurrentRecord;
    Term NewTerm = NULL;
    Record NewRecord = NULL;
    Offset NewOffset=NULL;
    int i;

    TotalWordCount++; //Increase number of total words.

    word=WordToTerm(word); //Make word standardization.
    pos = Find(word, Table_1, TotalTableSize, WhichPrime);  //Get the position.

    if(IsStopWord(word)){ //It is already stop word which is already sorted out.
        return; //Do nothing.
```

```
    }

    if ((*Table_1)[pos]->State != Legitimate) { //It is a new word.
        (*CurrentTableSize)++;
        (*Table_1)[pos]->State = Legitimate;
        NewTerm = (Term)malloc(sizeof(struct TermNode));
        (*Table_1)[pos]->Word = NewTerm;
        strcpy(NewTerm->Content,word);
        NewTerm->TotalFrequency = 1;
        NewTerm->TotalFiles = 1;
        NewRecord = (Record)malloc(sizeof(struct RecordNode));
        NewTerm->RecordList = NewRecord;
        NewRecord->LocalFrequency = 1;
        NewRecord->NextRecord = NULL;
        NewRecord->HostFile = fileNo;
        NewOffset = (Offset)malloc(sizeof(struct OffsetNode));
        NewRecord->FileOffset = NewOffset;
        NewOffset->OffsetInFile = WordOffset;
        NewOffset->NextOffset = NULL;
    }
    else if ((*Table_1)[pos]->State == Legitimate) { //It is already in hashtable.

    if(TotalWordCount>STOPWORDSTART&&((*Table_1)[pos]->Word->TotalFrequency/(double)Tot
alWordCount)>STOPWORDFREQUENCY) { //It is already stop word which is newly sorted out.
            strcpy(StopWordTable[StopWordNo++],word); //Add it to stop table.
            DeleteStopWord(Table_1,pos); //Delete it from table.
            return;
        }
        ((*Table_1)[pos]->Word->TotalFrequency)++;
        CurrentRecord = (*Table_1)[pos]->Word->RecordList;
        while (CurrentRecord != NULL) {
            if (CurrentRecord->HostFile == fileNo) {
                flag = 0;
                break;
            }
            CurrentRecord = CurrentRecord->NextRecord;
        }
        if (flag) { //It is the first time it appears in this file.
            ((*Table_1)[pos]->Word->TotalFiles)++;
            NewRecord = (Record)malloc(sizeof(struct RecordNode));
            NewRecord->NextRecord = (*Table_1)[pos]->Word->RecordList;
            (*Table_1)[pos]->Word->RecordList = NewRecord;
            NewRecord->LocalFrequency = 1;
            NewRecord->HostFile = fileNo;
```

```
                NewRecord->FileOffset = NULL;
                NewOffset = (Offset)malloc(sizeof(struct OffsetNode));
                NewOffset->NextOffset = NewRecord->FileOffset;
                NewRecord->FileOffset = NewOffset;
                NewOffset->OffsetInFile = WordOffset;
            }
            else {// It has appeared in this file.
                (CurrentRecord->LocalFrequency)++;
                NewOffset = (Offset)malloc(sizeof(struct OffsetNode));
                NewOffset->NextOffset = CurrentRecord->FileOffset;
                CurrentRecord->FileOffset = NewOffset;
                NewOffset->OffsetInFile = WordOffset;
            }
        }
        return;
}

/*
Find whether and where is the word in hash table.
Way to deal with collision:
Open Addressing & Quadratic Probing.
*/
int Find(char* word, Table** Table_1, int* TotalTableSize, int* WhichPrime) {
        int CurrentPos;   //Key,
        int CollisionNum = 0;
        int test=0;

        CurrentPos = Hashing(word, TotalTableSize); //Initial hash key.
        while (((*Table_1)[CurrentPos]->State==Legitimate) &&
((strcmp((*Table_1)[CurrentPos]->Word->Content,word)!=0))) {
            CurrentPos += 2 * ++CollisionNum - 1;  //Equal to square operation.
            CurrentPos = CurrentPos % (*TotalTableSize);
            if (CollisionNum > ((*TotalTableSize) / 2 + 1)) { //Load density > 0.5
                Rehashing(Table_1, TotalTableSize, WhichPrime);
                CollisionNum = 0;  //Reset CollisionNum.
                CurrentPos = Hashing(word, TotalTableSize);  //Reset CurrentPos.
            }
        }
        return CurrentPos;
}

/*
Rehashing.
Trigger: Load density of hash table > 0.5
```

```
*/
void Rehashing(Table** Table_1, int* TotalTableSize, int* WhichPrime) {
     int newpos, i,j, temp = *TotalTableSize; //Temp keeps the last size.
     Table* temptable;

     temptable=*Table_1;
     *WhichPrime = *WhichPrime + 1; //Next prime.
     *TotalTableSize = PrimeNumberTable[*WhichPrime]; //New tablesize.
     (*Table_1) = (Table*)malloc((*TotalTableSize)*sizeof(Table)); //Establish new table.
     for(j=0;j<(*TotalTableSize);j++){
          (*Table_1)[j] = (Table)malloc(sizeof(struct TableNode));
     }
     for(j=0;j<(*TotalTableSize);j++){  //Initialize.
          (*Table_1)[j]->State=Empty;
     }
     for (i = 0; i < temp; i++) {
          if (temptable[i]->State == Legitimate) {  //If it has a term, put it to newtable
               newpos = Find(temptable[i]->Word->Content,
Table_1,TotalTableSize,WhichPrime);
               (*Table_1)[newpos] = temptable[i];
          }
     }
     free(temptable);

     return;
}

/*copy s2[m]~s2[n-1] to s1*/
void my_strcpy(char* s1, char* s2, int m, int n){
     int i;
     for (i = 0; i + m < n; i++)
          s1[i] = s2[m + i];
     s1[i] = '\0';
}

Term Query_word(char *word, Table** Table_1, int* TotalTableSize, int* WhichPrime){
     int pos;
     /*tunrns to lower case, solve stemming and noisy word */
     word = WordToTerm(word);
     if (IsStopWord(word)) { //Filter stop words.
          strcpy(word," ");
     }
     if (strcmp(word," ")==0){ //it's a noisy word
          printf("Noisy Word!\n");
```

```
            return NULL;
        }
            /*find the position in the hash table*/
        else{
            pos = Find(word, Table_1, TotalTableSize, WhichPrime);

            /*if it's not in the hash table*/
            if ((*Table_1)[pos]->State != Legitimate){
                printf("Not Found!\n");
                return NULL;
            }
            else
                return (*Table_1)[pos]->Word;  //return the position in the hash table
        }
}

/*Bubble sort to sort the record list by local frequency decreasingly*/
void sort(Record *List, int num){
    int i, j;
    for (i = 0; i < num; i++){
        for (j = i; j < num; j++){
            if (List[i]->LocalFrequency < List[j]->LocalFrequency){
                Record newRecord = List[j];
                List[j] = List[i];
                List[i] = newRecord;
            }
        }
    }
}

void Query(char* word, float threshold, Table** Table_1, int* TotalTableSize, int*
WhichPrime){
    char word_temp[max_num][max_length];
    int flag = 0;// flag marks whether the input is a word or a phrase
    int i = 0, j = 0, k=0;
    int initial=-1;
    int num = 0; // num represent the number of words in the phrase
    Term Foundterm;
    Term *Foundterms;
    Record PresentRecord;
    Record* List;
    Record* CurrentRecord;// use to find whether the documents contain the phrase
    Offset* CurrentOffset;//// use to find whether the documents contain the phrase
    int count;
```

```
    int NotRightFile=0;
    int NotFound=1;
    int wordnum=0;

    /* if the input is a phrase, copy all the words into word_temp[][] */

    j = -1;  // j represent the last index of space
    while (word[i] != '\0'){
        if (word[i] == ' '||word[i]==' ' ||word[i]=='\'' ||word[i]=='-' ||word[i]=='['
||word[i]== ']' ||word[i]== '|' ||word[i]== '\"'||word[i]== '.' ||word[i]== '\?'
||word[i]== '!' ||word[i]== ':' ||word[i]== ';'||word[i]== ',' ){      // i represent the
present index of space
            if (i - j>1){
                flag = 1;
                /*copy a word into s_temp[num][] */
                my_strcpy(word_temp[num++], word, j + 1, i);
                word[i] = '\0';
            }
            j = i;
        }
        i++;
    }
    if (i - j>1)
        my_strcpy(word_temp[num++], word, j + 1, i);

    /* if the input is a phrase */
    if (flag == 1){
        /*set aside some space for Foundterms[], CurrentRecord[] and CurrentOffset[] */
        /*num represent the number of words in a phrase*/
        Foundterms = (Term *)malloc(sizeof(Term)* num);
        CurrentRecord = (Record *)malloc(sizeof(Record)*num);
        CurrentOffset = (Offset *)malloc(sizeof(Offset)*num);
        for (i = 0; i<num; i++){
            Foundterms[i] = Query_word(word_temp[i], Table_1, TotalTableSize,
WhichPrime);
            /*deal with noisy word or not found word*/
            if (Foundterms[i] == NULL){
                CurrentRecord[i] = NULL;
                CurrentOffset[i] = NULL;
                continue;
            }
            /*CurrentRecord[i] represent the current record of the ith word in the phrase*/
            /*CurrentOffset[i] represent the current offset of the ith word in the phrase*/
            else {
```

```
                /*avoid starting from noisy word or not found word*/
                if(initial == -1)
                    initial =i;
                CurrentRecord[i] = Foundterms[i]->RecordList;
                CurrentOffset[i] = CurrentRecord[i]->FileOffset;
                wordnum++;
            }
        }
        if(wordnum == 0)
            return;
        count = 0;    // count the number of documents
        for (j = 0; j < Foundterms[initial]->TotalFiles; j++){    // go through the
recordlist of the first word in the phrase
            for (k = 0; k < CurrentRecord[initial]->LocalFrequency; k++){ // go through
the position of the first word in a specific file
                for (i = 0; i < num; i++){// go through the rest words in the phrase
                    /*the records are stored by documentID decreasingly*/
                    while (CurrentRecord[i] != NULL && CurrentRecord[i]->HostFile >
CurrentRecord[initial]->HostFile){
                            CurrentRecord[i] = CurrentRecord[i]->NextRecord;
                            if(CurrentRecord[i] !=NULL)
                                CurrentOffset[i] = CurrentRecord[i]->FileOffset;
                    }
                    /*could not find a word in the database anymore, all the documents
contain the phrase has been printed out */
                    if (CurrentRecord[i] == NULL){
                        //if(NotFound ==1)    // no documents ID has been printed out
before
                        //  printf("Not Found!");
                        //  return ;
                        continue;
                    }
                    /*find the file contains both word[0] and word[i]*/
                    else if (CurrentRecord[i]->HostFile ==
CurrentRecord[initial]->HostFile){

                        /*the offset are stored decreasingly*/
                        while (CurrentOffset[i] != NULL &&
CurrentOffset[i]->OffsetInFile > (CurrentOffset[initial]->OffsetInFile + i - initial ))
                            CurrentOffset[i] = CurrentOffset[i]->NextOffset;

                        /*could not find the phrase in this file*/
                        if (CurrentOffset[i] == NULL){
                            NotRightFile = 1;     //mark in order to jump out 2 level
```

```
of loops
                                break;
                            }
                            /*find the offset that offset(word[i]) = offset(word[0])+i*/
                            else if (CurrentOffset[i]->OffsetInFile !=
CurrentOffset[initial]->OffsetInFile + i-initial )
                                break;

                        }
                        else{
                            /*could not find the phrase in this file*/
                            NotRightFile = 1;
                            break;
                        }
                    }
                    /*the file contains the phrase*/
                    if (i == num && count < threshold){
                        NotFound =0;
                        printf("%7d", CurrentRecord[initial]->HostFile);
                        count++;
                        /*output 10 files in a line*/
                        if (count % 10 == 0)
                            printf("\n");
                        NotRightFile = 1;
                    }
                    /*jump out the second level of loops*/
                    if(NotRightFile ==1 ){
                        NotRightFile =0;
                        break;
                    }
                    /*search the next offset of word[0]*/
                    else CurrentOffset[initial] = CurrentOffset[initial]->NextOffset;
                }
                /*search the next file of word[0]*/
                CurrentRecord[initial] = CurrentRecord[initial]->NextRecord;
                if(CurrentRecord[initial] !=NULL)
                    CurrentOffset[initial] = CurrentRecord[initial]->FileOffset;
            }
            if(NotFound ==1 )
                printf("Not Found!\n");
        }

        /* the input is a word */
        else{
```

```
        Foundterm = Query_word(word, Table_1, TotalTableSize, WhichPrime);
        if (Foundterm == NULL)
            return;
        if( threshold < 1 )
            threshold = threshold * Foundterm->TotalFiles;
        /*store the record list into an array List[]*/
        List = (Record*)malloc(sizeof(Record)* Foundterm->TotalFiles);
        PresentRecord = Foundterm->RecordList;
        for (i = 0; i<Foundterm->TotalFiles; i++){
            List[i] = PresentRecord;
            PresentRecord = PresentRecord->NextRecord;
        }
        /*sort the record list by the local frequency decreasingly*/
        sort(List,i);
        /*output 8 files in a line*/
        PresentRecord = Foundterm->RecordList;
        for (count = 0; count <  Foundterm->TotalFiles && count < threshold; count++){
            if (count % 8 == 0)
                printf("\n");
                    //  print document ID and frequency in the document
                printf("%4d(%3d)", List[count]->HostFile,
List[count]->LocalFrequency);
        }
    }
}


/*
Delete stop word.
Trigger: Frequency of a word > 1000/TotalWordCount.
*/
void DeleteStopWord(Table **Table_1,int pos){
    Record currentrecord,freerecord;
    Offset currentoffset,freeoffset;

    (*Table_1)[pos]->State=Empty;

    currentrecord=(*Table_1)[pos]->Word->RecordList;
    while(currentrecord!=NULL){
        freerecord=currentrecord;
        currentrecord=currentrecord->NextRecord;
        currentoffset=freerecord->FileOffset;
        while(currentoffset!=NULL){
            freeoffset=currentoffset;
```

```
            currentoffset=currentoffset->NextOffset;
            free(freeoffset);
        }
        free(freerecord);
    }
    return;
}

/*
Write currrent table into hard disk.
Trigger: Numbers of file > 1000.
*/
void WriteOut(Table** Table_1, int* TotalTableSize, char* filename){
    int i;
    Record currentrecord;
    Offset currentoffset;
    FILE *fp;

    fp=fopen(filename, "w+");
    /* Output current table. */
    for(i=0;i<(*TotalTableSize);i++){
        if((*Table_1)[i]->State==Legitimate){
            fprintf(fp,"Term:%s\n",(*Table_1)[i]->Word->Content);
            fprintf(fp,"TotalFrequency:%d\n",(*Table_1)[i]->Word->TotalFrequency);
            currentrecord=(*Table_1)[i]->Word->RecordList;
            while(currentrecord!=NULL){
                fprintf(fp,"HostFile:%d\n",currentrecord->HostFile);
                fprintf(fp,"LocalFrequency:%d\n",currentrecord->LocalFrequency);
                currentoffset=currentrecord->FileOffset;
                while(currentoffset!=NULL){
                    fprintf(fp,"FileOffset:%d\n",currentoffset->NextOffset);
                    currentoffset=currentoffset->NextOffset;
                }
                currentrecord=currentrecord->NextRecord;
            }
        }
    }
    /* Output ending. */
    fclose(fp);
    return;
}

/*
Delete currrent table.
```

```
Trigger: Numbers of file > 1000.
*/
void DeleteTable(Table** Table_1, int* TotalTableSize){
    int i;
    Record currentrecord,freerecord;
    Offset currentoffset,freeoffset;

    for(i=0;i<(*TotalTableSize);i++){
        if((*Table_1)[i]->State==Legitimate){
            currentrecord=(*Table_1)[i]->Word->RecordList;
            while(currentrecord!=NULL){
                freerecord=currentrecord;
                currentrecord=currentrecord->NextRecord;
                currentoffset=currentrecord->FileOffset;
                while(currentoffset!=NULL){
                    freeoffset=currentoffset;
                    currentoffset=currentoffset->NextOffset;
                    free(freeoffset);
                }
                free(freerecord);
            }
        }
        free((*Table_1)[i]);
    }
    free(*Table_1);

    return;
}

int main() {
    int a=0,b,c=0;
    int *WhichPrime = &a, *TotalTableSize=&b, *CurrentTableSize=&c;
    Table** Table_1;
    Table* TableArray;
    int FileNo = 0, WordOffset = 0; //Postion of file and word.
    char InputWord[WordLength];
    long file;
    struct _finddata_t find;
    struct FileNode data[MaxFileNumber];
    char temp;
    int i,j,k, flag = 1;
    char s[max_length]; // store the input word or phrase.
    float threshold;
    FILE* fp; //Pointer to output file.
```

```
    char filename[FILENAMELENGTH]; //store output filename.
    int index=0,digit=1;

    /*Initialize stopword table.*/
    for(k=0;k<StopWordNumber;k++){
        StopWordTable[k][0]='\n';
    }

    /*Initialize output filename.*/
    strcpy(filename,"table0");
    strcat(filename,".txt");

    /*begin to build the inverted file index*/
    *TotalTableSize = PrimeNumberTable[*WhichPrime];
    TableArray = (Table*)malloc((*TotalTableSize)*sizeof(Table));
    Table_1=&TableArray;
    for(j=0;j<(*TotalTableSize);j++){
        (*Table_1)[j] = (Table)malloc(sizeof(struct TableNode)); //Establish table.
    }
    for(j=0;j<(*TotalTableSize);j++){
        (*Table_1)[j]->State=Empty;
    }

    _chdir(DATAPATH);  //Change it if you put data in other directory.
    if ((file = _findfirst("*.txt", &find)) == -1L)
    {
        printf("Directory is empty!\n");
        exit(0);
    }
    data[FileNo].FileNumber = FileNo; //Number each file.
    data[FileNo].Location = fopen(find.name, "r"); //Open file.
    fseek(data[FileNo].Location, 0L, SEEK_SET); //Make file pointer points to the beginning
of file.
    //Get words from file.
    WordOffset = 0;
    while (flag) {
        i = 0;
        while (temp = fgetc(data[FileNo].Location)) {
            if (temp == EOF) {  //Reach the end of file.
                flag = 0;
                break;
            }
            //Ignore certain symbols.
            if (temp == ' ' ||temp == '\'' ||temp == '-' ||temp == '[' ||temp == ']' ||temp
```

```
== '|' ||temp == '\"'|| temp == '.' || temp == '\?' || temp == '!' || temp == ':' || temp
== ';'|| temp == ',' || temp == '\n') {
                    break;
               }
               else {
                    InputWord[i++] = temp;
               }
          }
          if (flag&&(i!=0)) {
               InputWord[i] = '\0'; //Mark the ending of a word.
               Insert(Table_1, InputWord, FileNo, WordOffset, CurrentTableSize,
TotalTableSize, WhichPrime);
               WordOffset++;
          }
     }
     fclose(data[FileNo].Location);//Close file.
     FileNo++;
     while (_findnext(file, &find) == 0) {
          flag=1;
          data[FileNo].FileNumber = FileNo;
          data[FileNo].Location = fopen(find.name, "r"); //Open file.
          fseek(data[FileNo].Location, 0L, SEEK_SET); //Make file pointer points to the
beginning of file.
          //Get words from file.
          WordOffset = 0;
          while (flag) {
               i = 0;
               while (temp = fgetc(data[FileNo].Location)) {
                    if (temp == EOF) {  //Reach the end of file.
                         flag = 0;
                         break;
                    }
                    //Ignore certain symbols.
                    if (temp == ' ' ||temp == '\'' ||temp == '-' ||temp == '[' ||temp == ']'
||temp == '|' ||temp == '\"'|| temp == '.'|| temp == ',' || temp == '\?' || temp == '!' ||
temp == ':' || temp == ';'|| temp == ',' || temp == '\n') {

                         break;
                    }
                    else {
                         InputWord[i++] = temp;
                    }
               }
               if (flag) {
```

```
                    InputWord[i] = '\0'; //Mark the ending of a word.
                    Insert(Table_1, InputWord, FileNo, WordOffset, CurrentTableSize,
TotalTableSize, WhichPrime);
                    WordOffset++;
                }
            }
            fclose(data[FileNo].Location);//关闭文件
            FileNo++;

            /*In order to handle huge data, write out table into hard disk every 1000 data
files.*/
            if(FileNo%1000==0){
                WriteOut(Table_1,TotalTableSize,filename);
                /*Prepare for the next output filename.*/
                index++;
            if(index%((int)pow(10,digit))==0){
                strncpy(filename,filename,5+digit);
                filename[6+digit]='\0';
                digit++;
                filename[3+digit]='1';
                filename[4+digit]='0'-1;
                strcat(filename,".txt");
                }
                filename[4+digit]++;
                /*Prepare ending.*/

                DeleteTable(Table_1,TotalTableSize);

                /*Initialize new table.*/
                *WhichPrime=0;
                *TotalTableSize = PrimeNumberTable[*WhichPrime];
                TableArray = (Table*)malloc((*TotalTableSize)*sizeof(Table));
                Table_1=&TableArray;
                for(j=0;j<(*TotalTableSize);j++){
                    (*Table_1)[j] = (Table)malloc(sizeof(struct TableNode)); //Establish
table.
                }
                for(j=0;j<(*TotalTableSize);j++){
                    (*Table_1)[j]->State=Empty;
                }
            }

        }
        _findclose(file);
```

```
    /*begin to do the query program*/
    /*input the word and threshold*/
    while(1){
        if(flag!=0){
            printf("\ninput 1 for continue, 0 for exit\n");
            scanf("%d",&j);
            if(!j)
                break;
        }
        printf("Please input a word or a phrase: ");
        for(i=0;i<max_length;i++)
            s[i]=0;
        if(flag!=0)
            getchar();
        gets(s);
        printf("Please input an integer or a floating point (percentage) as the threshold:
");
        scanf("%f", &threshold);
        Query(s,threshold, Table_1, TotalTableSize, WhichPrime);
        flag=1;
    }
    system("PAUSE");
    return 0;

}
```

Version 2: The search engine for large data and deal with the stop word with stop table

```
/*
How to use:
Please change the DATAPATH in define module according to where you put data in.

In general, what this program do are:
    Get words from files.
 -> Filter and sort words into terms.
 -> Record occurences of terms in table.
 -> Search word of query.
*/
```

```
/*
Header file declaration.
*/
#include "stdio.h"
#include "stdlib.h"
#include <string.h>
#include <math.h>
#include <ctype.h>
#include "io.h"
#include "direct.h" //You can use "direct.h" or "dir.h" depengding on environment.
#include "stem.h"

/*
Define constant.
*/
#define DATAPATH ".\\data"
#define MaxFileNumber 900 //Max number of files.
#define FILENAMELENGTH 30 //Max length of file name.
#define BufferSize 500 //Size of buffer.
#define StopWordNumber 428 //Number of stop words.
#define WordLength 30 //Suppose words are not longer than 30.
#define max_length 50 // Suppose input word or phrase are no longer than 50
#define max_num 10    // Suppose the number of words in a phrase are no longer than 10
#define Legitimate 1 //State of TableNode: Legitimate.
#define Empty 0  //State of TableNode: Empty.

/*
Global variable declaration.
*/
int TotalWordCount=0; //Number of total words.
int PrimeNumberTable[] = { 10159,20533,41233,82471,165719,348937,700001,1400017 }; //Table
of prime numbers.
char StopWordTable[][StopWordNumber] = { //Table of stop words.
    "a", "about","above", "across", "after", "again", "against", "all", "almost",
    "alone", "along", "already", "also", "although", "always", "among", "an",
    "and", "another", "any", "anybody", "anyone", "anything", "anywhere", "are"
    , "area", "areas", "around", "as", "ask", "asked", "asking", "asks", "at",
    "away", "b", "back", "backed", "backing", "backs", "be", "became", "because"
    , "become", "becomes", "been", "before", "began", "behind", "being",
    "beings", "best", "better", "between", "big", "both", "but", "by", "c", "came"
    , "can", "cannot", "case", "cases", "certain", "certainly", "clear",
    "clearly", "come", "could",
    "d", "did", "differ", "different", "differently", "do", "does", "done", "down"
```

```
    , "down", "downed", "downing", "downs", "during", "e", "each", "early",
    "either", "end", "ended", "ending", "ends", "enough", "even", "evenly",
    "ever", "every", "everybody", "everyone", "everything", "everywhere", "f",
    "face", "faces", "fact", "facts", "far", "felt", "few", "find", "finds",
    "first", "for", "four", "from", "full", "fully", "further", "furthered",
    "furthering", "furthers", "g", "gave", "general", "generally", "get", "gets"
    , "give", "given", "gives", "go", "going", "good", "goods", "got", "great",
    "greater", "greatest", "group", "grouped", "grouping", "groups",
    "h", "had", "has", "have", "having", "he", "her", "here", "herself", "high",
    "high", "high", "higher", "highest", "him", "himself", "his", "how",
    "however", "i", "if", "important", "in", "interest", "interested",
    "interesting", "interests", "into", "is", "it", "its", "itself", "j", "just"
    , "k", "keep", "keeps", "kind", "knew", "know", "known", "knows", "l", "large"
    , "largely", "last", "later", "latest", "least", "less", "let", "lets",
    "like", "likely", "long", "longer", "longest", "m", "made", "make", "making"
    , "man", "many", "may", "me", "member", "members", "men", "might", "more",
    "most", "mostly", "mr", "mrs", "much", "must", "my", "myself", "n",
    "necessary", "need", "needed", "needing", "needs", "never", "new", "newer",
    "newest", "next", "no", "nobody", "non", "noone", "not", "nothing", "now",
    "nowhere", "number", "numbers",
    "o", "of", "off", "often", "old", "older", "oldest", "on", "once", "one", "only"
    , "open", "opened", "opening", "opens", "or", "order", "ordered", "ordering"
    , "orders", "other", "others", "our", "out", "over", "p", "part", "parted",
    "parting", "parts", "per", "perhaps", "place", "places", "point", "pointed"
    , "pointing", "points", "possible", "present", "presented", "presenting"
    , "presents", "problem", "problems", "put", "puts", "q", "quite", "r",
    "rather", "really", "right", "right", "room", "rooms", "s", "said", "same",
    "saw", "say", "says", "second", "seconds", "see", "seem", "seemed", "seeming"
    , "seems", "sees", "several", "shall", "she", "should", "show", "showed",
    "showing", "shows", "side", "sides", "since", "small", "smaller", "smallest"
    , "so", "some", "somebody", "someone", "something", "somewhere", "state",
    "states", "still", "still", "such", "sure",
    "t", "take", "taken", "than", "that", "the", "their", "them", "then", "there",
    "therefore", "these", "they", "thing", "things", "think", "thinks", "this",
    "those", "though", "thought", "thoughts", "three", "through", "thus", "to"
    , "today", "together", "too", "took", "toward", "turn", "turned", "turning"
    , "turns", "two", "u", "under", "until", "up", "upon", "us", "use", "used",
    "uses", "v", "very", "w", "want", "wanted", "wanting", "wants", "was", "way",
    "ways", "we", "well", "wells", "went", "were", "what", "when", "where",
    "whether", "which", "while", "who", "whole", "whose", "why", "will", "with",
    "within", "without", "work", "worked", "working", "works", "would", "x", "y"
    , "year", "years", "yet", "you", "young", "younger", "youngest", "your",
    "yours", "z"
};
```

```
/*
Structure declaration.
*/
typedef struct FileNode* File;
struct FileNode {
    FILE* Location; //Pointer to where the file is.
    int FileNumber; //Position of file in a database.
};

typedef struct  OffsetNode* Offset;
struct OffsetNode {
    int OffsetInFile; //Location in the file.
    Offset NextOffset; //Pointer to next Location of the term.
};

typedef struct RecordNode* Record;
struct RecordNode {
    int LocalFrequency; //Occurrence number of this term in the file.
    int HostFile; //Numeber of the file it belongs to.
    Offset FileOffset; //Location in the file.
    Record NextRecord; //Pointer to next record of the term.
};

typedef struct TermNode* Term;
struct TermNode {
    char Content[WordLength]; //Term as string.
    int TotalFrequency; //Occurrence number of this term in the database.
    int TotalFiles;           //Occurrence number of files which contain this term
    Record RecordList; //Pointer to a list of its appearance records.
};

typedef struct TableNode* Table;
struct TableNode {
    int State; //Mark whether it is empty or not.
    Term Word; //Term in this node.
};

/*
Functions declaration.
*/
extern int stem(char * p, int i, int j); //Get the stem of a word.
char* WordToTerm(char* word); //Pre-process word into term
int IsStopWord(char* word); //Determine whether it is a stop word
```

```
int Hashing(char* word, int* TotalTableSize); //Hash function.
void Insert(Table** Table_1, char* word, int fileNo, int lineNo, int* CurrentTableSize, int*
TotalTableSize, int* WhichPrime);  //Insert an input.
int Find(char* word, Table** Table_1, int* TotalTableSize, int* WhichPrime); //Find the
position of a word.
void Rehashing(Table** Table_1, int* TotalTableSize, int* WhichPrime); //Rehashing when
half of the table is full.
void my_strcpy(char* s1, char* s2, int m, int n);//string copy operation
void sort(Record *List, int num);  //sort the document ID by the frequency of the term
Term Query_word(char *word, Table** Table_1, int* TotalTableSize, int* WhichPrime);
    //find the proper position of a term in the data base
void Query(char* word, float threshold, Table** Table_1, int* TotalTableSize, int*
WhichPrime); //solve 2 different situations of query for both word and phrases
void WriteOut(Table** Table_1, int* TotalTableSize,char* filename);
void DeleteTable(Table** Table_1, int* TotalTableSize);

/*
Pre-process word into term.
Functions as follow:
1、If it is upper case, transform it to lower case.
2、Filter stop words.
3、Get the stem of a word.
*/
char* WordToTerm(char* word) { //Pre-process word into term
    int i;

    for (i = 0; (word[i] != '\0') && (i < 30); i++) {
        if ((word[i] >= 'A') && (word[i] <= 'Z')) {
            word[i] = word[i] - 'A' + 'a'; //upper case->lower case.
        }
    }

    if (IsStopWord(word)) { //Filter stop words.
        strcpy(word," ");
        return word;
    }
    else{
       i--;
        i = stem(word, 0, i);
        word[i + 1] = '\0'; //Get the stem of a word.

        return word;
    }
}
```

```
/*
Find out whether it is a stop word.
Way to do that:
Search the word in stopWordTable by binary search since the stopWordTable is sorted.
*/
int IsStopWord(char* word) {
    int low, high, middle;
    low = 0, high = StopWordNumber - 1;
    middle = (low + high) / 2;
    while (low <= high) {
        middle = (low + high) / 2;
        if (strcmp(StopWordTable[middle], word)<0) { //stopWordTable[middle]<word
            low = middle + 1;
        }
        else if (strcmp(StopWordTable[middle], word)>0) { //stopWordTable[middle]>word
            high = middle - 1;
        }
        else {
            return 1;
        }
    }
    return 0;
}

/*
Get the key of hash.
Way to do that:
Using quadratic probing.
*/
int Hashing(char* word, int* TotalTableSize) {
    int result = 0, i;

    //the algorithm to calculate the hash value
    for (i = 0; (word[i] != '\0') && (i < 30); i++) {
        result = 32 * result + word[i];//Using 32 as weight of a char,
                                       //and multipling 32 is indeed shifting bits, which
is fast in hardware.
    }
    result = result % (*TotalTableSize);
    if (result < 0) { //Overflow may make the result negative, so here is a makeup.
        result = result + (*TotalTableSize);
    }
    return result;
```

```
}

/*
Record one time ocurrence of word.
Possible situations:
1、It is a new word.
2、It is already in table but first time in this file.
3、It is already in table and also in this file.
*/
void Insert(Table** Table_1, char* word, int fileNo, int WordOffset, int* CurrentTableSize,
int* TotalTableSize, int* WhichPrime) {
    int pos,flag = 1;
    Record CurrentRecord;
    Term NewTerm = NULL;
    Record NewRecord = NULL;
    Offset NewOffset=NULL;

    TotalWordCount++; //Increase number of total words.

    word=WordToTerm(word); //Make word standardization.
    if (strcmp(word," ")==0){ //If it is stop word,
        return; //don't need to be inserted.
    }
    else {
        pos = Find(word, Table_1, TotalTableSize, WhichPrime);  //Get the position.
    }

    if ((*Table_1)[pos]->State != Legitimate) { //It is a new word.
        (*CurrentTableSize)++;
        (*Table_1)[pos]->State = Legitimate;
        NewTerm = (Term)malloc(sizeof(struct TermNode));
        (*Table_1)[pos]->Word = NewTerm;
        strcpy(NewTerm->Content,word);
        NewTerm->TotalFrequency = 1;
        NewTerm->TotalFiles = 1;
        NewRecord = (Record)malloc(sizeof(struct RecordNode));
        NewTerm->RecordList = NewRecord;
        NewRecord->LocalFrequency = 1;
        NewRecord->NextRecord = NULL;
        NewRecord->HostFile = fileNo;
        NewOffset = (Offset)malloc(sizeof(struct OffsetNode));
        NewRecord->FileOffset = NewOffset;
        NewOffset->OffsetInFile = WordOffset;
        NewOffset->NextOffset = NULL;
```

```
    }
    else if ((*Table_1)[pos]->State == Legitimate) { //It is already in hashtable.
        ((*Table_1)[pos]->Word->TotalFrequency)++;
        CurrentRecord = (*Table_1)[pos]->Word->RecordList;
        while (CurrentRecord != NULL) {
            if (CurrentRecord->HostFile == fileNo) {
                flag = 0;
                break;
            }
            CurrentRecord = CurrentRecord->NextRecord;
        }
        if (flag) { //It is the first time it appears in this file.
            ((*Table_1)[pos]->Word->TotalFiles)++;
            NewRecord = (Record)malloc(sizeof(struct RecordNode));
            NewRecord->NextRecord = (*Table_1)[pos]->Word->RecordList;
            (*Table_1)[pos]->Word->RecordList = NewRecord;
            NewRecord->LocalFrequency = 1;
            NewRecord->HostFile = fileNo;
            NewRecord->FileOffset = NULL;
            NewOffset = (Offset)malloc(sizeof(struct OffsetNode));
            NewOffset->NextOffset = NewRecord->FileOffset;
            NewRecord->FileOffset = NewOffset;
            NewOffset->OffsetInFile = WordOffset;
        }
        else {// It has appeared in this file.
            (CurrentRecord->LocalFrequency)++;
            NewOffset = (Offset)malloc(sizeof(struct OffsetNode));
            NewOffset->NextOffset = CurrentRecord->FileOffset;
            CurrentRecord->FileOffset = NewOffset;
            NewOffset->OffsetInFile = WordOffset;
        }
    }
    return;
}

/*
Find whether and where is the word in hash table.
Way to deal with collision:
Open Addressing & Quadratic Probing.
*/
int Find(char* word, Table** Table_1, int* TotalTableSize, int* WhichPrime) {
    int CurrentPos;   //Key,
    int CollisionNum = 0;
    int test=0;
```

```
    CurrentPos = Hashing(word, TotalTableSize); //Initial hash key.
    while (((*Table_1)[CurrentPos]->State==Legitimate) &&
((strcmp((*Table_1)[CurrentPos]->Word->Content,word)!=0))) {
        CurrentPos += 2 * ++CollisionNum - 1;  //Equal to square operation.
        CurrentPos = CurrentPos % (*TotalTableSize);
        if (CollisionNum > ((*TotalTableSize) / 2 + 1)) { //Load density > 0.5
            Rehashing(Table_1, TotalTableSize, WhichPrime);
            CollisionNum = 0;  //Reset CollisionNum.
            CurrentPos = Hashing(word, TotalTableSize);  //Reset CurrentPos.
        }
    }
    return CurrentPos;
}

/*
Rehashing.
Trigger: Load density of hash table > 0.5
*/
void Rehashing(Table** Table_1, int* TotalTableSize, int* WhichPrime) {
    int newpos, i,j, temp = *TotalTableSize; //Temp keeps the last size.
    Table* temptable;

    temptable=*Table_1;
    *WhichPrime = *WhichPrime + 1; //Next prime.
    *TotalTableSize = PrimeNumberTable[*WhichPrime]; //New tablesize.
    (*Table_1) = (Table*)malloc((*TotalTableSize)*sizeof(Table)); //Establish new table.
    for(j=0;j<(*TotalTableSize);j++){
        (*Table_1)[j] = (Table)malloc(sizeof(struct TableNode));
    }
    for(j=0;j<(*TotalTableSize);j++){  //Initialize.
        (*Table_1)[j]->State=Empty;
    }
    for (i = 0; i < temp; i++) {
        if (temptable[i]->State == Legitimate) {  //If it has a term, put it to newtable
            newpos = Find(temptable[i]->Word->Content,
Table_1,TotalTableSize,WhichPrime);
            (*Table_1)[newpos] = temptable[i];
        }
    }
    free(temptable);

    return;
}
```

```
/*copy s2[m]~s2[n-1] to s1*/
void my_strcpy(char* s1, char* s2, int m, int n){
    int i;
    for (i = 0; i + m < n; i++)
        s1[i] = s2[m + i];
    s1[i] = '\0';
}

Term Query_word(char *word, Table** Table_1, int* TotalTableSize, int* WhichPrime){
    int pos;
    /*tunrns to lower case, solve stemming and noisy word */
    word = WordToTerm(word);
    if (strcmp(word," ")==0){ //it's a noisy word
        printf("Noisy Word!\n");
        return NULL;
    }
        /*find the position in the hash table*/
    else{
        pos = Find(word, Table_1, TotalTableSize, WhichPrime);

        /*if it's not in the hash table*/
        if ((*Table_1)[pos]->State != Legitimate){
            printf("Not Found!\n");
            return NULL;
        }
        else
            return (*Table_1)[pos]->Word;  //return the position in the hash table
    }
}

/*Bubble sort to sort the record list by local frequency decreasingly*/
void sort(Record *List, int num){
    int i, j;
    for (i = 0; i < num; i++){
        for (j = i; j < num; j++){
            if (List[i]->LocalFrequency < List[j]->LocalFrequency){
                Record newRecord = List[j];
                List[j] = List[i];
                List[i] = newRecord;
            }
        }
    }
```

```
}

void Query(char* word, float threshold, Table** Table_1, int* TotalTableSize, int*
WhichPrime){
    char word_temp[max_num][max_length];
    int flag = 0;// flag marks whether the input is a word or a phrase
    int i = 0, j = 0, k=0;
    int initial=-1;
    int num = 0; // num represent the number of words in the phrase
    Term Foundterm;
    Term *Foundterms;
    Record PresentRecord;
    Record* List;
    Record* CurrentRecord;// use to find whether the documents contain the phrase
    Offset* CurrentOffset;//// use to find whether the documents contain the phrase
    int count;
    int NotRightFile=0;
    int NotFound=1;
    int wordnum=0;

    /* if the input is a phrase, copy all the words into word_temp[][] */

    j = -1;  // j represent the last index of space
    while (word[i] != '\0'){
        if (word[i] == ' '||word[i]==' ' ||word[i]=='\'' ||word[i]=='-' ||word[i]=='['
||word[i]== ']' ||word[i]== '|' ||word[i]== '\"'||word[i]== '.' ||word[i]== '\?'
||word[i]== '!' ||word[i]== ':' ||word[i]== ';'||word[i]== ',' ){      // i represent the
present index of space
            if (i - j>1){
                flag = 1;
                /*copy a word into s_temp[num][] */
                my_strcpy(word_temp[num++], word, j + 1, i);
                word[i] = '\0';
            }
            j = i;
        }
        i++;
    }
    if (i - j>1)
        my_strcpy(word_temp[num++], word, j + 1, i);

    /* if the input is a phrase */
    if (flag == 1){
        /*set aside some space for Foundterms[], CurrentRecord[] and CurrentOffset[] */
```

```
        /*num represent the number of words in a phrase*/
        Foundterms = (Term *)malloc(sizeof(Term)* num);
        CurrentRecord = (Record *)malloc(sizeof(Record)*num);
        CurrentOffset = (Offset *)malloc(sizeof(Offset)*num);
        for (i = 0; i<num; i++){
            Foundterms[i] = Query_word(word_temp[i], Table_1, TotalTableSize,
WhichPrime);
            /*deal with noisy word or not found word*/
            if (Foundterms[i] == NULL){
                CurrentRecord[i] = NULL;
                CurrentOffset[i] = NULL;
                continue;
            }
            /*CurrentRecord[i] represent the current record of the ith word in the phrase*/
            /*CurrentOffset[i] represent the current offset of the ith word in the phrase*/
            else {
                /*avoid starting from noisy word or not found word*/
                if(initial == -1)
                    initial =i;
                CurrentRecord[i] = Foundterms[i]->RecordList;
                CurrentOffset[i] = CurrentRecord[i]->FileOffset;
                wordnum++;
            }
        }
        if(wordnum == 0)
            return;
        count = 0;    // count the number of documents
        for (j = 0; j < Foundterms[initial]->TotalFiles; j++){    // go through the
recordlist of the first word in the phrase
            for (k = 0; k < CurrentRecord[initial]->LocalFrequency; k++){ // go through
the position of the first word in a specific file
                for (i = 0; i < num; i++){// go through the rest words in the phrase
                    /*the records are stored by documentID decreasingly*/
                    while (CurrentRecord[i] != NULL && CurrentRecord[i]->HostFile >
CurrentRecord[initial]->HostFile){
                            CurrentRecord[i] = CurrentRecord[i]->NextRecord;
                            if(CurrentRecord[i] !=NULL)
                                CurrentOffset[i] = CurrentRecord[i]->FileOffset;
                    }
                    /*could not find a word in the database anymore, all the documents
contain the phrase has been printed out */
                    if (CurrentRecord[i] == NULL){
                        //if(NotFound ==1)    // no documents ID has been printed out
before
```

```
                    //   printf("Not Found!");
                    //   return ;
                    continue;
                }
                /*find the file contains both word[0] and word[i]*/
                else if (CurrentRecord[i]->HostFile ==
CurrentRecord[initial]->HostFile){

                    /*the offset are stored decreasingly*/
                    while (CurrentOffset[i] != NULL &&
CurrentOffset[i]->OffsetInFile > (CurrentOffset[initial]->OffsetInFile + i - initial ))
                        CurrentOffset[i] = CurrentOffset[i]->NextOffset;

                    /*could not find the phrase in this file*/
                    if (CurrentOffset[i] == NULL){
                        NotRightFile = 1;     //mark in order to jump out 2 level
of loops
                        break;
                    }
                    /*find the offset that offset(word[i]) = offset(word[0])+i*/
                    else if (CurrentOffset[i]->OffsetInFile !=
CurrentOffset[initial]->OffsetInFile + i-initial )
                        break;

                }
                else{
                    /*could not find the phrase in this file*/
                    NotRightFile = 1;
                    break;
                }
            }
            /*the file contains the phrase*/
            if (i == num && count < threshold){
                NotFound =0;
                printf("%7d", CurrentRecord[initial]->HostFile);
                count++;
                /*output 10 files in a line*/
                if (count % 10 == 0)
                    printf("\n");
                NotRightFile = 1;
            }
            /*jump out the second level of loops*/
            if(NotRightFile ==1 ){
                NotRightFile =0;
```

```
                    break;
                }
                /*search the next offset of word[0]*/
                else CurrentOffset[initial] = CurrentOffset[initial]->NextOffset;
            }
            /*search the next file of word[0]*/
            CurrentRecord[initial] = CurrentRecord[initial]->NextRecord;
            if(CurrentRecord[initial] !=NULL)
                CurrentOffset[initial] = CurrentRecord[initial]->FileOffset;
        }
        if(NotFound ==1 )
            printf("Not Found!\n");
    }

    /* the input is a word */
    else{
        Foundterm = Query_word(word, Table_1, TotalTableSize, WhichPrime);
        if (Foundterm == NULL)
            return;
        if( threshold < 1 )
            threshold = threshold * Foundterm->TotalFiles;
        /*store the record list into an array List[]*/
        List = (Record*)malloc(sizeof(Record)* Foundterm->TotalFiles);
        PresentRecord = Foundterm->RecordList;
        for (i = 0; i<Foundterm->TotalFiles; i++){
            List[i] = PresentRecord;
            PresentRecord = PresentRecord->NextRecord;
        }
        /*sort the record list by the local frequency decreasingly*/
        sort(List,i);
        /*output 8 files in a line*/
        PresentRecord = Foundterm->RecordList;
        for (count = 0; count <  Foundterm->TotalFiles && count < threshold; count++){
            if (count % 8 == 0)
                printf("\n");
                    //  print document ID and frequency in the document
                printf("%4d(%3d)", List[count]->HostFile,
List[count]->LocalFrequency);
        }
    }
}

void WriteOut(Table** Table_1, int* TotalTableSize, char* filename){
    int i;
```

```
    Record currentrecord;
    Offset currentoffset;
    FILE *fp;

    fp=fopen(filename, "w+");
    /* Output current table. */
    for(i=0;i<(*TotalTableSize);i++){
        if((*Table_1)[i]->State==Legitimate){
            fprintf(fp,"Term:%s\n",(*Table_1)[i]->Word->Content);
            fprintf(fp,"TotalFrequency:%d\n",(*Table_1)[i]->Word->TotalFrequency);
            currentrecord=(*Table_1)[i]->Word->RecordList;
            while(currentrecord!=NULL){
                fprintf(fp,"HostFile:%d\n",currentrecord->HostFile);
                fprintf(fp,"LocalFrequency:%d\n",currentrecord->LocalFrequency);
                currentoffset=currentrecord->FileOffset;
                while(currentoffset!=NULL){
                    fprintf(fp,"FileOffset:%d\n",currentoffset->NextOffset);
                    currentoffset=currentoffset->NextOffset;
                }
                currentrecord=currentrecord->NextRecord;
            }
        }
    }
    /* Output ending. */
    fclose(fp);
    return;
}

void DeleteTable(Table** Table_1, int* TotalTableSize){
    int i;
    Record currentrecord,freerecord;
    Offset currentoffset,freeoffset;

    for(i=0;i<(*TotalTableSize);i++){
        if((*Table_1)[i]->State==Legitimate){
            currentoffset=(*Table_1)[i]->Word->RecordList->FileOffset;
            while(currentoffset!=NULL){
                freeoffset=currentoffset;
                currentoffset=currentoffset->NextOffset;
                free(freeoffset);
            }
            currentrecord=(*Table_1)[i]->Word->RecordList;
            while(currentrecord!=NULL){
                freerecord=currentrecord;
```

```
                currentrecord=currentrecord->NextRecord;
                free(freerecord);
            }
        }
        free((*Table_1)[i]);
    }
    free(*Table_1);

    return;
}

int main() {
    int a=0,b,c=0;
    int *WhichPrime = &a, *TotalTableSize=&b, *CurrentTableSize=&c;
    Table** Table_1;
    Table* TableArray;
    int FileNo = 0, WordOffset = 0; //Postion of file and word.
    char InputWord[WordLength];
    long file;
    struct _finddata_t find;
    struct FileNode data[MaxFileNumber];
    char temp;
    int i,j, flag = 1;
    char s[max_length]; // store the input word or phrase.
    float threshold;
    FILE* fp; //Pointer to output file.
    char filename[FILENAMELENGTH]; //store output filename.
    int index=0,digit=1;

    /*Initialize output filename.*/
    strcpy(filename,"table0");
    strcat(filename,".txt");


    /*begin to build the inverted file index*/
    *TotalTableSize = PrimeNumberTable[*WhichPrime];
    TableArray = (Table*)malloc((*TotalTableSize)*sizeof(Table));
    Table_1=&TableArray;
    for(j=0;j<(*TotalTableSize);j++){
        (*Table_1)[j] = (Table)malloc(sizeof(struct TableNode)); //Establish table.
    }
    for(j=0;j<(*TotalTableSize);j++){
        (*Table_1)[j]->State=Empty;
    }
```

```
    _chdir(DATAPATH);  //Change it if you put data in other directory.
    if ((file = _findfirst("*.txt", &find)) == -1L)
    {
        printf("Directory is empty!\n");
        exit(0);
    }
    data[FileNo].FileNumber = FileNo; //Number each file.
    data[FileNo].Location = fopen(find.name, "r"); //Open file.
    fseek(data[FileNo].Location, 0L, SEEK_SET); //Make file pointer points to the beginning
of file.
    //Get words from file.
    WordOffset = 0;
    while (flag) {
        i = 0;
        while (temp = fgetc(data[FileNo].Location)) {
            if (temp == EOF) {  //Reach the end of file.
                flag = 0;
                break;
            }
            //Ignore certain symbols.
            if (temp == ' ' ||temp == '\'' ||temp == '-' ||temp == '[' ||temp == ']' ||temp
== '|' ||temp == '\"'|| temp == '.' || temp == '\?' || temp == '!' || temp == ':' || temp
== ';'|| temp == ',' || temp == '\n') {
                break;
            }
            else {
                InputWord[i++] = temp;
            }
        }
        if (flag&&(i!=0)) {
            InputWord[i] = '\0'; //Mark the ending of a word.
            Insert(Table_1, InputWord, FileNo, WordOffset, CurrentTableSize,
TotalTableSize, WhichPrime);
            WordOffset++;
        }
    }
    fclose(data[FileNo].Location);//Close file.
    FileNo++;
    while (_findnext(file, &find) == 0) {
        flag=1;
        data[FileNo].FileNumber = FileNo;
        data[FileNo].Location = fopen(find.name, "r"); //Open file.
        fseek(data[FileNo].Location, 0L, SEEK_SET); //Make file pointer points to the
```

```
beginning of file.
        //Get words from file.
        WordOffset = 0;
        while (flag) {
            i = 0;
            while (temp = fgetc(data[FileNo].Location)) {
                if (temp == EOF) {  //Reach the end of file.
                    flag = 0;
                    break;
                }
                //Ignore certain symbols.
                if (temp == ' ' ||temp == '\'' ||temp == '-' ||temp == '[' ||temp == ']'
||temp == '|' ||temp == '\"'|| temp == '.'|| temp == ',' || temp == '\?' || temp == '!' ||
temp == ':' || temp == ';'|| temp == ',' || temp == '\n') {

                    break;
                }
                else {
                    InputWord[i++] = temp;
                }
            }
            if (flag) {
                InputWord[i] = '\0'; //Mark the ending of a word.
                Insert(Table_1, InputWord, FileNo, WordOffset, CurrentTableSize,
TotalTableSize, WhichPrime);
                WordOffset++;
            }
        }
        fclose(data[FileNo].Location);//关闭文件
        FileNo++;

        /*In order to handle huge data, write out table into hard disk every 1000 data
files.*/
        if(FileNo%1000==0){
            WriteOut(Table_1,TotalTableSize,filename);
            /*Prepare for the next output filename.*/
            index++;
        if(index%((int)pow(10,digit))==0){
            strncpy(filename,filename,5+digit);
            filename[6+digit]='\0';
            digit++;
            filename[3+digit]='1';
            filename[4+digit]='0'-1;
            strcat(filename,".txt");
```

```
            }
            filename[4+digit]++;
            /*Prepare ending.*/

            DeleteTable(Table_1,TotalTableSize);

            /*Initialize new table.*/
            *WhichPrime=0;
            *TotalTableSize = PrimeNumberTable[*WhichPrime];
            TableArray = (Table*)malloc((*TotalTableSize)*sizeof(Table));
            Table_1=&TableArray;
            for(j=0;j<(*TotalTableSize);j++){
                (*Table_1)[j] = (Table)malloc(sizeof(struct TableNode)); //Establish
table.
            }
            for(j=0;j<(*TotalTableSize);j++){
                (*Table_1)[j]->State=Empty;
            }
        }
    }
    _findclose(file);

    /*begin to do the query program*/
    /*input the word and threshold*/
    while(1){
        if(flag!=0){
            printf("\ninput 1 for continue, 0 for exit\n");
            scanf("%d",&j);
            if(!j)
                break;
        }
        printf("Please input a word or a phrase: ");
        for(i=0;i<max_length;i++)
            s[i]=0;
        if(flag!=0)
            getchar();
        gets(s);
        printf("Please input an integer or a floating point (percentage) as the threshold:
");
        scanf("%f", &threshold);
        Query(s,threshold, Table_1, TotalTableSize, WhichPrime);
        flag=1;
    }
    system("PAUSE");
```

```
    return 0;
    /*begin to do the query program*/
}
```

Version 3: Stemming.c

```
/* This is the Porter stemming algorithm, coded up in ANSI C by the
author. It may be be regarded as canonical, in that it follows the
algorithm presented in

Porter, 1980, An algorithm for suffix stripping, Program, Vol. 14,
no. 3, pp 130-137,

only differing from it at the points marked --DEPARTURE-- below.

See also http://www.tartarus.org/~martin/PorterStemmer

The algorithm as described in the paper could be exactly replicated
by adjusting the points of DEPARTURE, but this is barely necessary,
because (a) the points of DEPARTURE are definitely improvements, and
(b) no encoding of the Porter stemmer I have seen is anything like
as exact as this version, even with the points of DEPARTURE!

You can compile it on Unix with 'gcc -O3 -o stem stem.c' after which
'stem' takes a list of inputs and sends the stemmed equivalent to
stdout.

The algorithm as encoded here is particularly fast.

Release 1: was many years ago
Release 2: 11 Apr 2013
fixes a bug noted by Matt Patenaude <matt@mattpatenaude.com>,

case 'o': if (ends("\03" "ion") && (b[j] == 's' || b[j] == 't')) break;
==>
case 'o': if (ends("\03" "ion") && j >= k0 && (b[j] == 's' || b[j] == 't')) break;

to avoid accessing b[k0-1] when the word in b is "ion".
Release 3: 25 Mar 2014
fixes a similar bug noted by Klemens Baum <klemensbaum@gmail.com>,
that if step1ab leaves a one letter result (ied -> i, aing -> a etc),
step2 and step4 access the byte before the first letter. So we skip
```

```
steps after steplab unless k > k0.
*/

#include <string.h>  /* for memmove */
//#include "stdafx.h"

#define TRUE 1
#define FALSE 0

/* The main part of the stemming algorithm starts here. b is a buffer
holding a word to be stemmed. The letters are in b[k0], b[k0+1] ...
ending at b[k]. In fact k0 = 0 in this demo program. k is readjusted
downwards as the stemming progresses. Zero termination is not in fact
used in the algorithm.

Note that only lower case sequences are stemmed. Forcing to lower case
should be done before stem(...) is called.
*/

static char * b;       /* buffer for word to be stemmed */
static int k, k0, j;     /* j is a general offset into the string */

                         /* cons(i) is TRUE <=> b[i] is a consonant. */

static int cons(int i)
{
    switch (b[i])
    {
    case 'a': case 'e': case 'i': case 'o': case 'u': return FALSE;
    case 'y': return (i == k0) ? TRUE : !cons(i - 1);
    default: return TRUE;
    }
}

/* m() measures the number of consonant sequences between k0 and j. if c is
a consonant sequence and v a vowel sequence, and <..> indicates arbitrary
presence,

<c><v>       gives 0
<c>vc<v>     gives 1
<c>vcvc<v>   gives 2
<c>vcvcvc<v> gives 3
....
*/
```

```
static int m()
{
    int n = 0;
    int i = k0;
    while (TRUE)
    {
        if (i > j) return n;
        if (!cons(i)) break; i++;
    }
    i++;
    while (TRUE)
    {
        while (TRUE)
        {
            if (i > j) return n;
            if (cons(i)) break;
            i++;
        }
        i++;
        n++;
        while (TRUE)
        {
            if (i > j) return n;
            if (!cons(i)) break;
            i++;
        }
        i++;
    }
}

/* vowelinstem() is TRUE <=> k0,...j contains a vowel */

static int vowelinstem()
{
    int i; for (i = k0; i <= j; i++) if (!cons(i)) return TRUE;
    return FALSE;
}

/* doublec(j) is TRUE <=> j,(j-1) contain a double consonant. */

static int doublec(int j)
{
    if (j < k0 + 1) return FALSE;
```

```
    if (b[j] != b[j - 1]) return FALSE;
    return cons(j);
}

/* cvc(i) is TRUE <=> i-2,i-1,i has the form consonant - vowel - consonant
and also if the second c is not w,x or y. this is used when trying to
restore an e at the end of a short word. e.g.

cav(e), lov(e), hop(e), crim(e), but
snow, box, tray.

*/

static int cvc(int i)
{
    if (i < k0 + 2 || !cons(i) || cons(i - 1) || !cons(i - 2)) return FALSE;
    {  int ch = b[i];
    if (ch == 'w' || ch == 'x' || ch == 'y') return FALSE;
    }
    return TRUE;
}

/* ends(s) is TRUE <=> k0,...k ends with the string s. */

static int ends(char * s)
{
    int length = s[0];
    if (s[length] != b[k]) return FALSE; /* tiny speed-up */
    if (length > k - k0 + 1) return FALSE;
    if (memcmp(b + k - length + 1, s + 1, length) != 0) return FALSE;
    j = k - length;
    return TRUE;
}

/* setto(s) sets (j+1),...k to the characters in the string s, readjusting
k. */

static void setto(char * s)
{
    int length = s[0];
    memmove(b + j + 1, s + 1, length);
    k = j + length;
}
```

```
/* r(s) is used further down. */

static void r(char * s) { if (m() > 0) setto(s); }

/* step1ab() gets rid of plurals and -ed or -ing. e.g.

caresses  ->  caress
ponies    ->  poni
ties      ->  ti
caress    ->  caress
cats      ->  cat

feed      ->  feed
agreed    ->  agree
disabled  ->  disable

matting   ->  mat
mating    ->  mate
meeting   ->  meet
milling   ->  mill
messing   ->  mess

meetings  ->  meet

*/

static void step1ab()
{
    if (b[k] == 's')
    {
        if (ends("\04" "sses")) k -= 2; else
            if (ends("\03" "ies")) setto("\01" "i"); else
                if (b[k - 1] != 's') k--;
    }
    if (ends("\03" "eed")) { if (m() > 0) k--; }
    else
        if ((ends("\02" "ed") || ends("\03" "ing")) && vowelinstem())
        {
            k = j;
            if (ends("\02" "at")) setto("\03" "ate"); else
                if (ends("\02" "bl")) setto("\03" "ble"); else
                    if (ends("\02" "iz")) setto("\03" "ize"); else
                        if (doublec(k))
                        {
```

```
                    k--;
                    {  int ch = b[k];
                    if (ch == 'l' || ch == 's' || ch == 'z') k++;
                    }
                }
                else if (m() == 1 && cvc(k)) setto("\01" "e");
        }
}

/* step1c() turns terminal y to i when there is another vowel in the stem. */

static void step1c() { if (ends("\01" "y") && vowelinstem()) b[k] = 'i'; }


/* step2() maps double suffices to single ones. so -ization ( = -ize plus
-ation) maps to -ize etc. note that the string before the suffix must give
m() > 0. */

static void step2() {
    switch (b[k - 1])
    {
    case 'a': if (ends("\07" "ational")) { r("\03" "ate"); break; }
              if (ends("\06" "tional")) { r("\04" "tion"); break; }
              break;
    case 'c': if (ends("\04" "enci")) { r("\04" "ence"); break; }
              if (ends("\04" "anci")) { r("\04" "ance"); break; }
              break;
    case 'e': if (ends("\04" "izer")) { r("\03" "ize"); break; }
              break;
    case 'l': if (ends("\03" "bli")) { r("\03" "ble"); break; } /*-DEPARTURE-*/

                                                                /* To match the
published algorithm, replace this line with

                                                                case 'l': if
(ends("\04" "abli")) { r("\04" "able"); break; } */

              if (ends("\04" "alli")) { r("\02" "al"); break; }
              if (ends("\05" "entli")) { r("\03" "ent"); break; }
              if (ends("\03" "eli")) { r("\01" "e"); break; }
              if (ends("\05" "ousli")) { r("\03" "ous"); break; }
              break;
    case 'o': if (ends("\07" "ization")) { r("\03" "ize"); break; }
              if (ends("\05" "ation")) { r("\03" "ate"); break; }
              if (ends("\04" "ator")) { r("\03" "ate"); break; }
```

```
                break;
    case 's': if (ends("\05" "alism")) { r("\02" "al"); break; }
                if (ends("\07" "iveness")) { r("\03" "ive"); break; }
                if (ends("\07" "fulness")) { r("\03" "ful"); break; }
                if (ends("\07" "ousness")) { r("\03" "ous"); break; }
                break;
    case 't': if (ends("\05" "aliti")) { r("\02" "al"); break; }
                if (ends("\05" "iviti")) { r("\03" "ive"); break; }
                if (ends("\06" "biliti")) { r("\03" "ble"); break; }
                break;
    case 'g': if (ends("\04" "logi")) { r("\03" "log"); break; } /*-DEPARTURE-*/

                                                                /* To match the
published algorithm, delete this line */

    }
}

/* step3() deals with -ic-, -full, -ness etc. similar strategy to step2. */

static void step3() {
    switch (b[k])
    {
    case 'e': if (ends("\05" "icate")) { r("\02" "ic"); break; }
                if (ends("\05" "ative")) { r("\00" ""); break; }
                if (ends("\05" "alize")) { r("\02" "al"); break; }
                break;
    case 'i': if (ends("\05" "iciti")) { r("\02" "ic"); break; }
                break;
    case 'l': if (ends("\04" "ical")) { r("\02" "ic"); break; }
                if (ends("\03" "ful")) { r("\00" ""); break; }
                break;
    case 's': if (ends("\04" "ness")) { r("\00" ""); break; }
                break;
    }
}

/* step4() takes off -ant, -ence etc., in context <c>vcvc<v>. */

static void step4()
{
    switch (b[k - 1])
    {
    case 'a': if (ends("\02" "al")) break; return;
```

```
    case 'c': if (ends("\04" "ance")) break;
        if (ends("\04" "ence")) break; return;
    case 'e': if (ends("\02" "er")) break; return;
    case 'i': if (ends("\02" "ic")) break; return;
    case 'l': if (ends("\04" "able")) break;
        if (ends("\04" "ible")) break; return;
    case 'n': if (ends("\03" "ant")) break;
        if (ends("\05" "ement")) break;
        if (ends("\04" "ment")) break;
        if (ends("\03" "ent")) break; return;
    case 'o': if (ends("\03" "ion") && j >= k0 && (b[j] == 's' || b[j] == 't')) break;
        if (ends("\02" "ou")) break; return;
        /* takes care of -ous */
    case 's': if (ends("\03" "ism")) break; return;
    case 't': if (ends("\03" "ate")) break;
        if (ends("\03" "iti")) break; return;
    case 'u': if (ends("\03" "ous")) break; return;
    case 'v': if (ends("\03" "ive")) break; return;
    case 'z': if (ends("\03" "ize")) break; return;
    default: return;
    }
    if (m() > 1) k = j;
}

/* step5() removes a final -e if m() > 1, and changes -ll to -l if
m() > 1. */

static void step5()
{
    j = k;
    if (b[k] == 'e')
    {
        int a = m();
        if (a > 1 || a == 1 && !cvc(k - 1)) k--;
    }
    if (b[k] == 'l' && doublec(k) && m() > 1) k--;
}

/* In stem(p,i,j), p is a char pointer, and the string to be stemmed is from
p[i] to p[j] inclusive. Typically i is zero and j is the offset to the last
character of a string, (p[j+1] == '\0'). The stemmer adjusts the
characters p[i] ... p[j] and returns the new end-point of the string, k.
Stemming never increases word length, so i <= k <= j. To turn the stemmer
into a module, declare 'stem' as extern, and delete the remainder of this
```

```
file.
*/

int stem(char * p, int i, int j)
{
    b = p; k = j; k0 = i; /* copy the parameters into statics */
    if (k <= k0 + 1) return k; /*-DEPARTURE-*/

                               /* With this line, strings of length 1 or 2 don't go
through the
                               stemming process, although no mention is made of this in
the
                               published algorithm. Remove the line to match the
published
                               algorithm. */

    step1ab();
    if (k > k0) {
        step1c(); step2(); step3(); step4(); step5();
    }
    return k;
}
```

References

[1] Mark Allen Weiss,"Data Structures and Algorithm Analysis in C (Second Edition)", Addison-Wesley,1996.

[2]Onix,"Stop Word List1",

http://www.lextek.com/manuals/onix/stopwords1.html.

[3]MartinPorter,"The Porter Stemming Algorithm",

http://tartarus.org/martin/PorterStemmer/index.html.

Author List:

ZiMin Chen:Write query program and chapter 1.2 of the program

Pu Yang:do the test , write the chapter 3.4 of the program and wrap document and source file.

Changlin Zhang: write index generation program and presentation materials.

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.