Roll Your Own Mini Search Engine

Group 5
(Zhao Bingqian, Lin Xiaojun, Lei Lulu)

Date: 2010-12-6

Chapter 1: introduction

1.1 Problem description

We all use search engines such as google and baidu. Sometimes it seems that we can't live without them. So this time we will create our own search engine which is supposed to handle 100,000 inquires over 100 files in 10 seconds.

1.2 Background Algorithm and Data Structures

1.2.1 Invert file index

An inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents. Here in this project, we solve the problem with inverted file index which is used by almost all main search engines to speed up the searching task. We store the words' information in the inverted file which includes the articles and lines it appears. Its main advantage is when dealing with queries of more than one keyword, we may finish inquires by accomplish intersection and union logic operations in an inverted table, get the result and then store it. In this way we convert records queries to operation on address set, so as to improve the speed of search!

1.2.2 AVL Tree

An AVL tree is a self-balancing binary search tree, and it was the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

AVL trees are often compared with red-black trees because they support the same set of operations and because red-black trees also take O(log n) time for the basic operations. AVL trees perform better than red-black trees for lookup-intensive applications.

1.2.3 Trie

A trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.

Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.

It is not necessary for keys to be explicitly stored in nodes.

Though it is most common, tries need not be keyed by character strings. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes. In particular, a bitwise trie is keyed on the individual bits making up a short, fixed size of bits such as an integer number or pointer to memory.

Chapter 2: data structure & algorithm specification

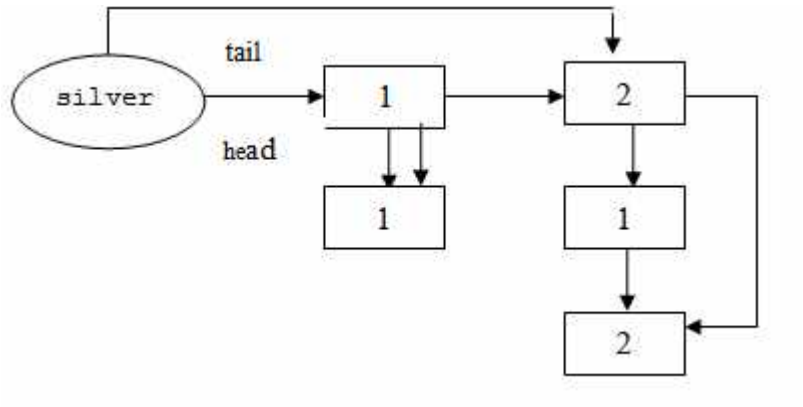
2.1 Inverted file index

Our idea is simple. We want to search a word, so we should know each word's accurate address in the articles given. Inverted file is just for this purpose. In the chart below, the second column is to record address for each word.

Doc	Text
1	Gold silver truck
2	Delivery of silver arrived in a silver truck

Term	Documents Words
silver	<(1;1)(2;1,2)>
truck	<(1;1),(2;2)>

To get the chart, we scan all the articles and record the address. Here we apply a collection of lists to realize it. The lists are illustrated below.



Every word has such a structure which includes two kinds of lists.

One is article list used to record the articles the word appears. Its node is a user-defined structure called ArticleNode, which is displayed as follows.

```
struct ArticleNode{
    int order;
    LineNodePtr head,tail;
    ArticleNodePtr next;
};
```

The other is line node used to record the lines of a article the word appears , as is refered above.

Notice that in the two lists head and tail pointers are used. Tail stores the last node in a list. When we need to insert a node, we adopt it., we will rely it. Head points the first node of a list. When we need to search a word and print out the information about it in the order it input. They are used

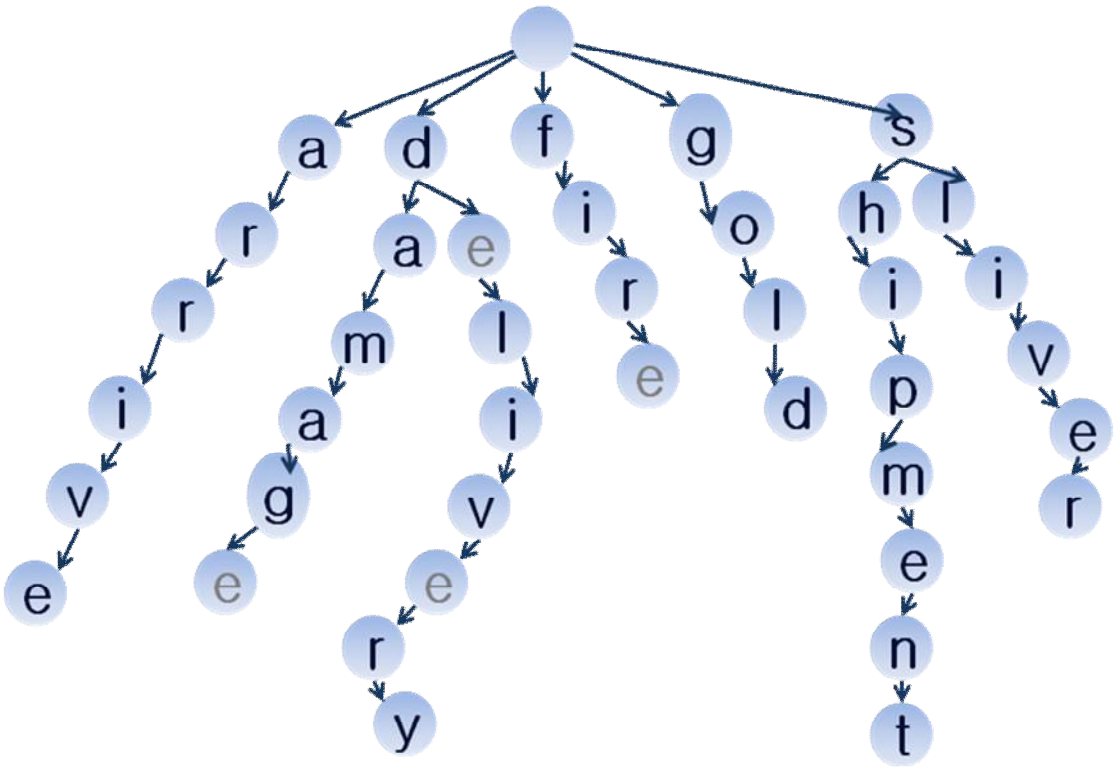
just to satisfy the output specification. Otherwise we may just insert a node as the first node in a list and that operation doesn't need pointers.

2.2 AVL Tree

It is to solve the problem that how we store those word node with that struture. We store them in an AVL Tree just in the dictional order. It's very easy so we won't display it here.

2.3 Trie

It's an alternative solution to AVL Tree which is illustrated below.



Now the node stores letter rather than words. Each node has at most

26 children. We use a flag to mark the end of a word. For example, we insert cat, t is marked, then we know the word cat is in the tree. It is simple to insert.

2.4 Answer list

A list is for an inquiry. Its node is ArticleNode introduced above. For every word, we look for it in the AVL Tree first. If it exits, we copy its information to the answer list. Note that no line is printed more than once for the same inquiry, so we merge it with the answer list instead of simply copy it.

Chapter 3: Testing Results

3.1 Test Cases

Our tester created 10 cases to test if the two algorithms solve the problem correctly. And all the cases and the expected results are given below.

3.1.1 See if the program work correctly when the file content changes.

Case number	Purpose	Input	Expected output
Case 1	See if the program works correctly when some files are empty.	3 A00 # A01 What will	1 A02 Shipment of gold arrived in

		happen ? I don’t know . # A02 Shipment of gold arrived in a truck # 1 Shipment 0	
Case 2	See if the program works correctly when there are two or more same lines in a file	3 A00 # A01 What will happen ? I don’t know . # A02 Shipment of gold arrived in Shipment of gold arrived in a truck # 1 Shipment gold 0	1 A02 Shipment of gold arrived in Shipment of gold arrived in 0
Case3	There are symbols in the files	2 A00 Gold. "how are you?" silver,truck # A01 Shipment of gold, arrived? "Truck?" damaged in a fire # 4 truck arrive what ever silver truck 0	2 A00 silver,truck A01 "Truck?" 1 A01 Shipment of gold, arrived? "Truck?" 0 Not Found 1 A00 silver,truck

3.1.2 See if the program will work correctly when the file content is certain and the inquiries changes.

In those cases, all the files are as below:

A00 Gold silver truck #	A01 Shipment of gold damaged in a fire #	A02 Delivery of silver arrived in a silver truck #	A03 Shipment of gold arrived in a truck #

Case number	Purpose	Input	Expected output
Case 4	See if the program works correctly when the inquiry contains word does not appear in the file	2 Hello What ever 0	0 Not Found 0 Not Found
Case 5	See if the program works correctly when one inquiry contains only one word	2 gold turck 0	3 A00 Gold A01 Shipment of gold A03 Shipment of gold 3 A00 silver truck A02 truck A03 a truck
Case 6	See if the program works correctly when the inquiry contains stop words (ignore the stop words)	2 A shipment Of gold 0	2 A01 Shipment of gold A03 Shipment of gold 3 A00 Gold A01 Shipment of gold A03 Shipment of gold
Case 7	See if the program works correctly when the inquiry consists of only stop words	2 In a Of a 0	0 Not Found 0 Not Found
Case 8	See if the program works correctly when the inquiry contains word need to be stemmed	2 Damaged in Trucks	1 A01 damaged 3 A00 silver truck A02 truck A03

			a truck
Case 9	See if the program works correctly when a inquiry contains more than one word, there are some files contains all of them and some contains only part.	1 silver truck 0	2 A00 silver truck A02 of silver a silver truck
Case 10	See if the program works correctly when a inquiry contains more than one word and they appear in the same line in a file.	1 Shipment gold 0	2 A01 Shipment of gold A03 Shipment of gold
Case 11	See if the program works correctly when a inquiry contains more than one word and they appear in different lines in a file.	1 Gold truck 0	2 A00 Gold silver truck A03 Shipment of gold a truck

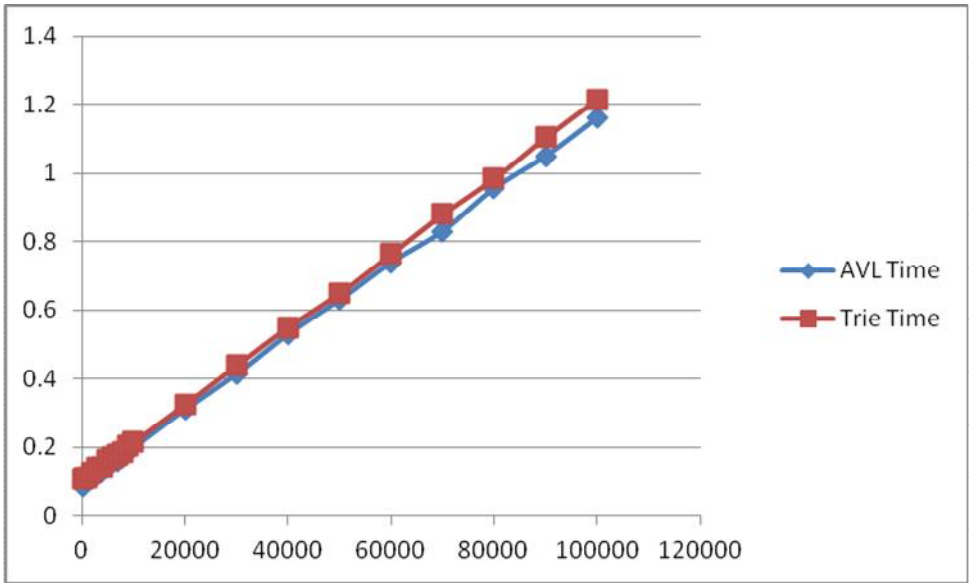
Program case	1	2	3	4	5	6	7	8	9	10	11
AVL tree	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
Trie	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass

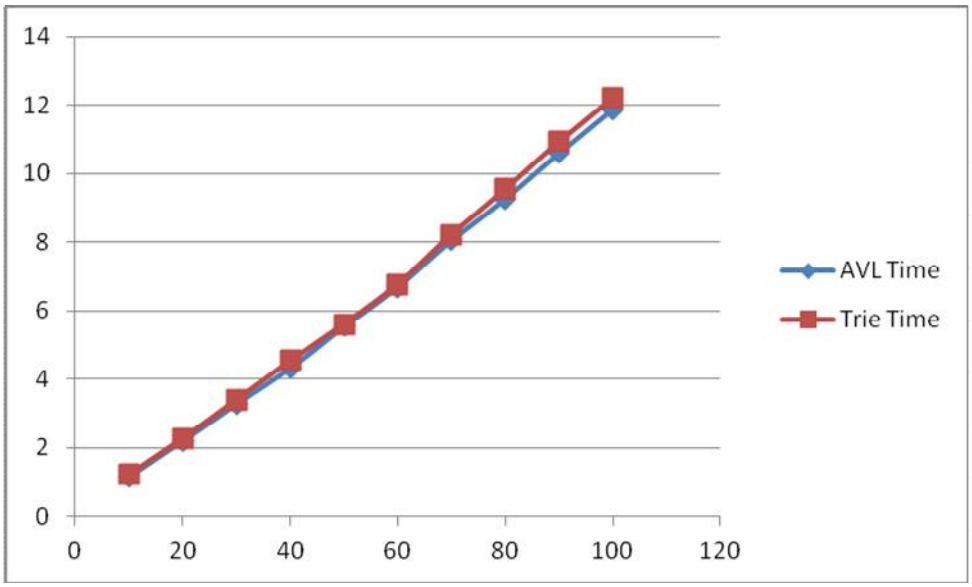
3.2 Running time

The tester write a program – createfile ，to create a file contains all the data ,and the tester add some lines to the program to count time.

The AVL tree:

File number	Lines number	Column number	Inquiry	AVL Time	Trie Time
10	100	20	100	0.085000	0.109000
10	100	20	500	0.120000	0.110000
10	100	20	1000	0.100000	0.110000
10	100	20	2000	0.115000	0.125000
10	100	20	3000	0.120000	0.140000
10	100	20	4000	0.130000	0.140000
10	100	20	5000	0.145000	0.165000
10	100	20	6000	0.155000	0.170000
10	100	20	7000	0.160000	0.180000
10	100	20	8000	0.170000	0.185000
10	100	20	9000	0.185000	0.205000
10	100	20	10000	0.200000	0.215000
10	100	20	20000	0.310000	0.325000
10	100	20	30000	0.415000	0.440000
10	100	20	40000	0.530000	0.550000
10	100	20	50000	0.630000	0.650000
10	100	20	60000	0.740000	0.765000
10	100	20	70000	0.830000	0.880000
10	100	20	80000	0.955000	0.985000
10	100	20	90000	1.050000	1.105000
10	100	20	100000	1.160000	1.215000
20	100	20	100	0.205000	0.220000
20	100	20	1000	0.220000	0.240000
20	100	20	10000	0.410000	0.425000
20	100	20	100000	2.215000	2.280000
20	50	40	100000	2.235000	2.250000
20	25	80	100000	2.175000	2.235000
20	20	100	100000	2.185000	2.255000
30	100	20	100000	3.290000	3.400000
40	100	20	100000	4.330000	4.535000
50	100	20	100000	5.545000	5.615000
60	100	20	100000	6.695000	6.770000
70	100	20	100000	8.055000	8.225000
80	100	20	100000	9.270000	9.545000
90	100	20	100000	10.595000	10.925000
100	100	20	100000	11.880000	12.205000
100	100	19	100000	11.375000	11.245000
100	100	18	100000	10.900000	11.165000
100	100	17	100000	9.945000	10.480000





Chapter 4: Analysis and Comments

From all the testing results, both AVL tree and Trie are able to do the work, and there is not much difference in the time token.

4.1 Time Complexity :

The AVL tree takes O(logN)to insert and O(logN)to find, N is the

number of index word in the files. So the time complexity of the program using AVL tree is O((N+M)logN).

For the other program, Trie takes O(P) to insert and O(P) to find, P is the length of the longest word. So the time complexity of the program using Trie is O(PN+PM).

In this project we don't need to do deletion, but the AVL tree has to do rotation to keep the tree easy for search, which will take some time.

4.2 Space Complexity :

In AVL tree, there are exactly N nodes. So the space complexity is O(N). In Trie, the nodes is more than N and less than PN. So the space complexity is O(PN).

4.3 Possible Improvements

In the result answer, we merge the answer lists one by one, which can be improve by divide and conquer merging. But the words in an inquiry is no more than 10, which means this improvements isn't very efficient.

If the data scale is increased to 500,000 files with 400,000,000 distinct words, our program can't answer in 10s. For the program using AVL tree, the time for each insertion and finding is O(logN), which is about 28 if N equals 400,000,000. For the program using Trie, the time for each insertion and finding is O(P), which is supposed less than 20. So as

long as there is enough space to build the inverted index tree, our programs can give the right answer in the end.

But if the data scale is increased much larger, it's impossible to store the inverted file in the RAM. So the inverted file should be stored in the disk separated. In this case, B+ tree is useful, which reduces the number of I/O operations required to find an element in the tree.

Appendix: Source Code (if required)

```
/*
* cy_G5_P2_AVL.c - the program for Research Project 2 of Advanced Data Structures
and Algorithm Analysis.
* Copyright (C) Zhao Bingqian of Group 5. All rights reserved.
* Purpose: search words in different articles.
* Algorithm: Using AVL Tree to store the inverted file.
* Notice: the maximum number of articles, lines in a article and characters in
a line is 100, 200 and 250.
* Instructor: Chen Yue.
* Version: 2010/12/5
*/
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define WordLength 10       /*The longest length of a word.*/
#define LineLength 250      /*The longest length of a line.*/
#define LineNumber 200      /*The maximum number of lines in a article.*/
#define ArticleNumber 100   /*The maximum number of articles.*/
#define EmptyNode -1        /*A flag of an empty node.*/
#define TRUE 1
#define FALSE 0
#define AlphaNumber 26      /*The number of english alpha.*/

typedef struct WordNode *WordNodePtr;
typedef struct ArticleNode *ArticleNodePtr;
typedef struct LineNode *LineNodePtr;
```

```
/*
*The Trie stores the stop words.
*Member 'f' marks it's a word from the root to this node.
*Member 'Next[]' points to its 26 children.
*/
typedef struct TrieNode *Trie;
struct TrieNode{
    char f;
    Trie Next[AlphaNumber];
};

/*
*The AVL tree node stores the index word.
*Member 'word[]' stores the index word.
*Member 'head' and 'tail' points to the head and tail of the Article list.
*Member 'left' and 'right' points to its left and right child.
*Member 'Height' stores the height of the node in the AVL tree.
*/
struct WordNode{
    char word[WordLength+1];
    ArticleNodePtr head,tail;
    WordNodePtr left,right;
    int Height;
};

/*
*The list node stored the article number which contains the index word.
*Member 'order' stores the article number.
*Member 'head' and 'tail' points to the head and tail of the line list.
*Member 'next' points to  the next node of the Article list.
*/
struct ArticleNode{
    int order;
    LineNodePtr head,tail;
    ArticleNodePtr next;
};

/*
*The list node stored the line number of the article which contain the index word.
*Member 'order' stores the line number in the article.
*Member 'next' points the next node of the list.
*/
struct LineNode{
    int order;
```

```
    LineNodePtr next;
};

/*The prior declaration of the AVL tree handling functions.*/
WordNodePtr NewWordNode(void);
WordNodePtr AddWord(WordNodePtr t,char *word,int ArticleNum,int LineNum);
WordNodePtr SingleRotateWithLeft(WordNodePtr K2);
WordNodePtr SingleRotateWithRight(WordNodePtr K2);
WordNodePtr DoubleRotateWithLeft(WordNodePtr K3);
WordNodePtr DoubleRotateWithRight(WordNodePtr K3);
WordNodePtr FindWord(WordNodePtr p,char *word);
int Height(WordNodePtr p);
void PrintTree(WordNodePtr);
void EmptyTree(WordNodePtr);

/*The prior declaration of the list handling functions.*/
ArticleNodePtr NewArticleNode(void);
LineNodePtr NewLineNode(void);
ArticleNodePtr AddArticle(int ArticleNum,int LineNum);
LineNodePtr AddLine(int LineNum);
void CopyArticle(ArticleNodePtr p,ArticleNodePtr q);
void MergeArticle(ArticleNodePtr p,ArticleNodePtr q);
void CopyLine(LineNodePtr p,LineNodePtr q);
void EmptyArticle(ArticleNodePtr p);
void EmptyLine(LineNodePtr p);
void PrintArticle(ArticleNodePtr p);
void PrintLine(int ArticleNum,LineNodePtr p);

/*The prior declaration of the Trie handling functions.*/
Trie NewTrieNode();
void InsertTrie(Trie StopWordsList,char *str);
int StopWord(Trie StopWordsList,char *str);
void PrintStopWords(Trie p,char *str,int k);

/*The prior declaration of stemming functions.*/
int Stemming(char * p, int i, int j);

/*The prior declaration of the Maximum and FatalError functions.*/
int Max(int x,int y);
void FatalError(char *str);

/*The declaration of the array which stores the stop words list.*/
const char *Stop_Words_List[] = {
    "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", "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",
    NULL
};

/*The declaration of variables in Stemmning functions.*/
static char * b;
static int k,k0,j;

/*
```

```
*Article[i] stores the articles' information.
*Article[][0] stores the article title. Article[][i] stores the ith line of the
article.
*/
char *Article[ArticleNumber][LineNumber+1];

int main(void){
    int n,m,k;
    int LineNum,i,j;
    char line[LineLength],word[LineLength];

    WordNodePtr t,pw;              /*t is the inverted index tree.*/
    ArticleNodePtr Ans,pf;         /*Ans stores the inquiry answer result.*/
    Trie StopWordsList;            /*StopWordsList stores the stop words in the
Trie.*/

    /*Creat a Trie node for the head of the StopWordsList Trie.*/
    StopWordsList=NULL;
    StopWordsList=NewTrieNode();
    if(StopWordsList==NULL)FatalError("Out of space!");

    /*Insert each stop word into the StopWordsList Trie.*/
    i=0;
    while(Stop_Words_List[i]!=NULL)
        InsertTrie(StopWordsList,Stop_Words_List[i++]);

    /*
    *Read in the number n and
    *build up the AVL inverted index tree t
    *until n is equal to 0.
    */
    while(scanf("%d",&n),getchar(),n!=0){
        /*Initialize the AVL inverted index tree t.*/
        t=NULL;

        for(k=0;k<n;k++){
            /*Get the article title of the kth article.*/
            gets(line);
            Article[k][0]=(char*)malloc(sizeof(char)*(strlen(line)+1));
            strcpy(Article[k][0],line);

            /*Initialize the total line number of the article.*/
            LineNum=0;
            /*Get the (LineNum+1)th line of the article until it's equal to "#".*/
```

```
            while(gets(line),strcmp(line,"#")){
                /*Increase the total line number of the article.*/
                LineNum++;
                /*Store the whole line in the array Article[][].*/

    Article[k][LineNum]=(char*)malloc(sizeof(char)*(strlen(line)+1));
                strcpy(Article[k][LineNum],line);

                i=0;
                /*Scan the whole line and pick out the index words.*/
                while(line[i]){
                    /*Ingore the non-alpha in the line.*/
                    while(line[i]&&!isalpha(line[i]))i++;
                    if(!line[i])break;

                    /*Copy the lower case of index word to word[].*/
                    j=0;
                    while(isalpha(line[i]))
                        word[j++]=tolower(line[i++]);
                    word[j]='\0';

                    /*Ingore the words longer than 10.*/
                    if(strlen(word)<=WordLength){
                        /*Ingore the stop words.*/
                        if(!StopWord(StopWordsList,word)){
                            /*Stemming the word.*/
                            word[Stemming(word,0,strlen(word)-1)+1]='\0';
                            /*Add the information of the index word into the tree
t.*/
                            t=AddWord(t,word,k,LineNum);
                        }
                    }
                }
            }
        }

        /*Read in the number of inquiries m.*/
        scanf("%d",&m);getchar();
        /*Respond each inquiry.*/
        for(k=0;k<m;k++){
            /*Get the whole inquiry line.*/
            gets(line);

            /*New an answer list to store the inquire result.*/
```

```
            Ans=NewArticleNode();
            Ans->order=EmptyNode;
            Ans->head=Ans->tail=NULL;
            Ans->next=NULL;

            /*Scan the whole inquiry line and pick out the inquiry words.*/
            i=0;
            while(line[i]){
                /*Ingore the blank spaces in the line.*/
                while(line[i]==' ')i++;
                if(!line[i])break;

                /*Copy the lower case of inquiry word to word[].*/
                j=0;
                while(line[i]&&line[i]!=' ')
                    word[j++]=tolower(line[i++]);
                word[j]='\0';

                /*Ingore the words longer than 10.*/
                if(strlen(word)<=WordLength){
                    /*Ingore the stop words.*/
                    if(!StopWord(StopWordsList,word)){
                        /*Stemming the word.*/
                        word[Stemming(word,0,strlen(word)-1)+1]='\0';

                        /*Find the inquiry word in the inverted index tree.*/
                        pw=FindWord(t,word);
                        if(pw!=NULL){
                            pf=pw->head;

                            /*
                            *If it's the first inquiry word, copy the result to
Ans list.
                            *Otherwise, merge the result to the Ans list, leaving
the common information.
                            */
                            if(Ans->next==NULL)
                                CopyArticle(Ans,pf);
                            else
                                MergeArticle(Ans,pf);
                        }
                    }
                }
            }
```

```
            /*First, print out the number of articles containing the inquiry
word.*/
            printf("%d\n",-(Ans->order+1));
            /*
            *If no article found, print out "Not Found".
            *Otherwise, print out the articles' title and the lines containing
the inquiry word.
            */
            if(Ans->next==NULL)
                puts("Not Found");
            else
                PrintArticle(Ans);

            /*Empty the space of the current answer list.*/
            EmptyArticle(Ans);
        }
        /*Empty the space of the current inverted index tree.*/
        EmptyTree(t);
    }

    return 0;
}

/*Return the maximum element of x and y.*/
int Max(int x,int y){
    if(x>y)
        return x;
    else
        return y;
}

/*Print out the error information and quit.*/
void FatalError(char *str){
    puts(str);
    exit(1);
}

/*
*Add the new index word information into the inverted index tree.
*/
WordNodePtr AddWord(WordNodePtr t,char *word,int ArticleNum,int LineNum){
    /*
    *If the index word isn't found,
```

```
    *creat a new tree node to store the word.
    */
    if(t==NULL){
        t=NewWordNode();
        strcpy(t->word,word);
        t->Height=0;
        /*Add the article information to the node.*/
        t->head=t->tail=AddArticle(ArticleNum,LineNum);
        t->left=t->right=NULL;
    }
    /*
    *If the index word is smaller than the node word,
    *add the word information to the left subtree of the node.
    */
    else if(strcmp(word,t->word)<0){
        t->left=AddWord(t->left,word,ArticleNum,LineNum);

        /*Adjust the tree to keep it as an AVL tree.*/
        if(Height(t->left)-Height(t->right)==2)
            if(strcmp(word,t->left->word)<0)
                t=SingleRotateWithLeft(t);
            else
                t=DoubleRotateWithLeft(t);
    }
    /*
    *If the index word is larger than the node word,
    *add the word information to the right subtree of the node.
    */
    else if(strcmp(word,t->word)>0){
        t->right=AddWord(t->right,word,ArticleNum,LineNum);

        /*Adjust the tree to keep it as an AVL tree.*/
        if(Height(t->right)-Height(t->left)==2)
            if(strcmp(word,t->right->word)>0)
                t=SingleRotateWithRight(t);
            else
                t=DoubleRotateWithRight(t);
    }
    /*
    *If the index word is found,
    *add the new information to the node.
    */
    else
        /*
```

```
        *If the article is in the article list of the word, add the new line.
        *Otherwise, creat an article node to the word.
        */
        if(t->tail->order==ArticleNum)
            t->tail->tail=t->tail->tail->next=AddLine(LineNum);
        else
            t->tail=t->tail->next=AddArticle(ArticleNum,LineNum);

    /*Adjust the height of the node.*/
    t->Height=Max(Height(t->left),Height(t->right))+1;
    return t;
}

/*
*Rotate the left nodes.
*/
WordNodePtr SingleRotateWithLeft(WordNodePtr K2){
    WordNodePtr K1;

    K1=K2->left;
    K2->left=K1->right;
    K1->right=K2;

    /*Update the height of the nodes changed.*/
    K2->Height=Max(Height(K2->left),Height(K2->right))+1;
    K1->Height=Max(Height(K1->left),K2->Height)+1;

    return K1;
}

/*
*Rotate the right nodes.
*/
WordNodePtr SingleRotateWithRight(WordNodePtr K2){
    WordNodePtr K1;

    K1=K2->right;
    K2->right=K1->left;
    K1->left=K2;

    /*Update the height of the nodes changed.*/
    K2->Height=Max(Height(K2->left),Height(K2->right))+1;
    K1->Height=Max(K2->Height,Height(K1->right))+1;
```

```
    return K1;
}

/*
*Rotate the right nodes of the left subtree.
*/
WordNodePtr DoubleRotateWithLeft(WordNodePtr K3){
    WordNodePtr K1,K2;

    K2=K3->left;
    K1=K2->right;
    K2->right=K1->left;
    K3->left=K1->right;
    K1->left=K2;
    K1->right=K3;

    /*Update the height of the nodes changed.*/
    K2->Height=Max(Height(K2->left),Height(K2->right))+1;
    K3->Height=Max(Height(K3->left),Height(K3->right))+1;
    K1->Height=Max(K2->Height,K3->Height)+1;

    return K1;
}

/*
*Rotate the left nodes of the right subtree.
*/
WordNodePtr DoubleRotateWithRight(WordNodePtr K3){
    WordNodePtr K1,K2;

    K2=K3->right;
    K1=K2->left;
    K3->right=K1->left;
    K2->left=K1->right;
    K1->left=K3;
    K1->right=K2;

    /*Update the height of the nodes changed.*/
    K2->Height=Max(Height(K2->left),Height(K2->right))+1;
    K3->Height=Max(Height(K3->left),Height(K3->right))+1;
    K1->Height=Max(K2->Height,K3->Height)+1;

    return K1;
}
```

```
/*
*Return the height of the AVL tree node.
*/
int Height(WordNodePtr p){
    if(p==NULL)
        return -1;
    else
        return p->Height;
}

/*
*Find the index word in the inverted index tree.
*/
WordNodePtr FindWord(WordNodePtr p,char *word){
    while(p!=NULL)
        if(strcmp(word,p->word)<0)
            p=p->left;
        else if(strcmp(word,p->word)>0)
            p=p->right;
        else return p;

    /*Not find, return NULL.*/
    return p;
}

/*
*Creat a new tree node to store the the word.
*/
WordNodePtr NewWordNode(void){
    WordNodePtr p;

    p=NULL;
    p=malloc(sizeof(struct WordNode));
    if(p==NULL)
        FatalError("Out of space!");
    else
        return p;
}

/*
*Free the space of the inverted index tree.
*/
void EmptyTree(WordNodePtr p){
```

```
    if(p==NULL)return 0;

    /*Empty the left subtree recursively.*/
    if(p->left!=NULL)
        EmptyTree(p->left);

    /*Empty the right subtree recursively.*/
    if(p->right!=NULL)
        EmptyTree(p->right);

    /*Empty the Article list.*/
    EmptyArticle(p->head);

    free(p);
}

/*
*Creat a new article list node to store the new information.
*/
ArticleNodePtr AddArticle(int ArticleNum,int LineNum){
    ArticleNodePtr p;

    p=NewArticleNode();
    p->order=ArticleNum;
    p->head=p->tail=AddLine(LineNum);
    p->next=NULL;

    return p;
}

/*
*Creat a new line list node to store the new information.
*/
LineNodePtr AddLine(int LineNum){
    LineNodePtr p;

    p=NewLineNode();
    p->order=LineNum;
    p->next=NULL;

    return p;
}

/*
```

```
*Copy the whole article list from q to p.
*/
void CopyArticle(ArticleNodePtr p,ArticleNodePtr q){
    ArticleNodePtr head,r,nq;
    LineNodePtr nl;

    head=p;
    r=p->next;
    while(q!=NULL){
        /*'head->order stores the number of aricles in the list, from -1 to
-(n+1).*/
        head->order--;

        /*'nl' is the new line list head to be added to the new article list node.*/
        nl=NewLineNode();
        nl->order=EmptyNode;
        nl->next=NULL;
        /*Copy the whole line list from node q->head to node nl.*/
        CopyLine(nl,q->head);

        /*'nq' is the new article node to be added in the article list of p.*/
        nq=NewArticleNode();
        nq->order=q->order;
        nq->head=nl;
        nq->tail=NULL;
        nq->next=r;

        /*Add the new article node nq to p.*/
        p->next=nq;
        p=nq;
        q=q->next;
    }
}

/*
*Merge the the article list p and q.
*Remain the common part of p and q and delete others in p.
*/
void MergeArticle(ArticleNodePtr p,ArticleNodePtr q){
    ArticleNodePtr head,r,nq;
    LineNodePtr nl;

    head=p;
    r=p->next;
```

```
    while(r!=NULL&&q!=NULL)
        /*The article in list q isn't in list p. Delete the article node in p.*/
        if(r->order<q->order){
            head->order++;
            r=r->next;
            EmptyLine(p->next->head);
            free(p->next);
            p->next=r;
        }
        /*The article in list p isn't in list q. Ignore the node.*/
        else if(r->order>q->order)
            q=q->next;
        /*The artilcle is in both list. Merge the line list from q to p.*/
        else{
            CopyLine(r->head,q->head);
            p=r;
            r=r->next;
            q=q->next;
        }
    /*If there are more articles in list p except q, delete them.*/
    while(r!=NULL){
        head->order++;
        r=r->next;
        EmptyLine(p->next->head);
        free(p->next);
        p->next=r;
    }
}

/*
*Copy the whole line list from q to p without repetition.
*/
void CopyLine(LineNodePtr p,LineNodePtr q){
    LineNodePtr r,nq;

    r=p->next;
    while(r!=NULL&&q!=NULL)
        if(r->order<q->order){
            p=r;
            r=r->next;
        }
        /*Node q isn't in list p. Copy it from q to p.*/
        else if(r->order>q->order){
            nq=NewArticleNode();
```

```
            nq->order=q->order;
            nq->next=r;

            p->next=nq;
            p=nq;
            q=q->next;
        }
        else{
            p=r;
            r=r->next;
            q=q->next;
        }
    /*If there are more nodes in q, copy them all to list p.*/
    while(q!=NULL){
        nq=NewArticleNode();
        nq->order=q->order;
        nq->next=r;

        p->next=nq;
        p=nq;
        q=q->next;
    }
}

/*
*Creat a new empty article list node.
*/
ArticleNodePtr NewArticleNode(void){
    ArticleNodePtr p;

    p=NULL;
    p=malloc(sizeof(struct ArticleNode));
    if(p==NULL)
        FatalError("Out of space!");
    else
        return p;
}

/*
*Creat a new empty line list node.
*/
LineNodePtr NewLineNode(void){
    LineNodePtr p;
```

```
    p=NULL;
    p=malloc(sizeof(struct LineNode));
    if(p==NULL)
        FatalError("Out of space!");
    else
        return p;
}

/*
*Free the space of the whole article list.
*/
void EmptyArticle(ArticleNodePtr p){
    ArticleNodePtr q;

    while(p!=NULL){
        /*Store the next node of p in q.*/
        q=p->next;
        /*Empty the line list of p.*/
        EmptyLine(p->head);
        /*Free the space of current article node.*/
        free(p);
        /*Turn to the next node.*/
        p=q;
    }
}

/*
*Free the space of the whole line list.
*/
void EmptyLine(LineNodePtr p){
    LineNodePtr q;

    while(p!=NULL){
        /*Store the next node of p in q.*/
        q=p->next;
        /*Free the space of current line node.*/
        free(p);
        /*Turn to the next node.*/
        p=q;
    }
}

/*
*Print out the whole article list.
```

```
*/
void PrintArticle(ArticleNodePtr p){
    while(p){
        /*Ignore the dummy head node of the list.*/
        if(p->order>=0){
            /*Print out the article title first.*/
            puts(Article[p->order][0]);
            /*Print out the whole lines which contain the inquiry words.*/
            PrintLine(p->order,p->head);
        }
        p=p->next;
    }
}

/*
*Print out the whole line list.
*/
void PrintLine(int ArticleNum,LineNodePtr p){
    while(p){
        /*Ignore the dummy head node of the list.*/
        if(p->order!=EmptyNode)
            /*Print out the line of the article which contains the inquiry words.*/
            puts(Article[ArticleNum][p->order]);
        p=p->next;
    }
}

/*
*Creat a new Trie node where stores the stop words.
*/
Trie NewTrieNode(){
    int i;
    Trie p;

    p=malloc(sizeof(struct TrieNode));
    p->f=0;
    for(i=0;i<AlphaNumber;i++)
        p->Next[i]=NULL;
    return p;
}

/*
*Insert a new stop word 'str' into the 'StopWordsList' Trie.
*Stop insert the word if it contains non-lower-case alpha characters.
```

```
*/
void InsertTrie(Trie StopWordsList,char *str){
    Trie p,q;
    int i,k;

    /*Scan the Trie from the root.*/
    p=StopWordsList;
    /*Search the word from the first letter.*/
    i=0;
    /*Search the word until it ends or it isn't in the Trie.*/
    while(islower(str[i])){
        k=str[i]-'a';
        if(!p->Next[k])break;
        p=p->Next[k];
        i++;
    }
    /*Insert the word from the letter where search loop breaks out.*/
    while(islower(str[i])){
        k=str[i]-'a';
        /*Creat the new Trie node to store the letter of the word.*/
        q=NULL;
        q=NewTrieNode();
        if(q==NULL)FatalError("Out of space!");
        p->Next[k]=q;
        p=q;
        i++;
    }
    /*Mark the last letter node that there is a word.*/
    if(!str[i]) p->f=1;
}

/*
*Search the stop word in the StopWordsList Trie.
*Return 1 if found, 0 not.
*/
int StopWord(Trie StopWordsList,char *str){
    int i,k;
    Trie p;

    p=StopWordsList;
    i=0;
    while(str[i]){
        k=str[i]-'a';
        /*Break out the loop if the letter doesn't exist.*/
```

```
        if(p->Next[k]==NULL)return 0;
        p=p->Next[k];
        i++;
    }
    /*If the last letter node refers to a word, return that the word is found.*/
    if(p->f) return 1;
    /*Otherwise, return that the word is not found.*/
    else return 0;
}

/*
*The following codes is the stemming function which we find from the Internet.
*This function is using The Porter Stemming Algorithm.
*More information about the algorithm, please refer to the References [6] listed
in our report.
*/
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;
   }
}

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++;
   }
}

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

static int doublec(int j)
{  if (j < k0+1) return FALSE;
   if (b[j] != b[j-1]) return FALSE;
   return cons(j);
}

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;
}

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;
}

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

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

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");
   }
}

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

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-*/

              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-*/

} }

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;
} }

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") && (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;
}

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--;
}



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

   step1ab(); step1c(); step2(); step3(); step4(); step5();
   return k;
}
```

For test

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<string.h>
4   #include<time.h>
5   #define WORDNUM 4000
6   #define WORDLEN 30
7   //#define FILENUM 50         /*change them to */
8   //#define COLUMN 30          /*define the size*/
9   //#define WORDINACOLUMN 100    /* of a file*/
10  //#define INQUIRY 1000           /*change to define the number of inqueries*/
11  int main(){
12      int i,k,j,m,n=0,q,l;
13      char *wordnum[WORDNUM],str[WORDLEN];
14      int FILENUM,COLUMN,WORDINACOLUMN,INQUIRY;
```

```
    FILE *fp1;
    FILE *fp2;

    printf("File Number:");scanf("%d",&FILENUM);
    printf("Line Number:");scanf("%d",&COLUMN);
    printf("Column Number:");scanf("%d",&WORDINACOLUMN);
    printf("Inquiry Number:");scanf("%d",&INQUIRY);
    fp1=fopen("wordlist.txt","r");
    fscanf(fp1,"%s",str);
    n=0;
    /*read in the words in "wordlist.txt" and store them in the array*/
    while(!feof(fp1)){
        wordnum[n]=(char*)malloc(sizeof(char)*(strlen(str)+1));
        strcpy(wordnum[n],str);
        n++;
        fscanf(fp1,"%s",str);
    }

    srand( (unsigned)time( NULL ) );
    fp2=fopen("input.txt","w+");/*open  the  "input.txt",if  it  doesnot
exist,create a new one*/
    fprintf(fp2,"%d\n",FILENUM);/*write in the number of files*/
    /*write in the title of files*/
    for(j=0;j<FILENUM;j++){
        fputc('A',fp2);
        if (j<10){
            fputc('0',fp2);
            fputc(j+'0',fp2);
            fputc('\n',fp2);
        }
        else{
            fputc(j/10+'0',fp2);
            fputc(j%10+'0',fp2);
            fputc('\n',fp2);
        }
        for(m=0;m<COLUMN;m++){     /*the number of columns in a file*/
            for(i=0;i<WORDINACOLUMN;i++){     /*the number of words in a
column */
                k=rand()%n;   /*write a word in the "input.txt" randomly*/
                fputs(wordnum[k],fp2);
                fputc(' ',fp2);/* blank between two words*/
            }
            fputc('\n',fp2);
        }
```

```
59          fputc('#',fp2); /*end of a file*/
60          fputc('\n',fp2);
61      }
62      fprintf(fp2,"%d\n",INQUIRY); /*create the inquiry list*/
63      for(q=0;q<INQUIRY;q++){
64          l=rand()%n;
65          fputs(wordnum[l],fp2);
66          fputc('\n',fp2);
67      }
68      fputc('0',fp2); /*end of the file*/
69      fclose(fp1);
70      fclose(fp2);
71      return 0;
72  }
```

References

[1] Mark Allen Weiss, “Data Structure and Algorithm Analysis in C(Second Edition)”, POSTS&TELECOM PRESS 2005

[2] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, "Introduction to Information Retrieval", Cambridge University Press. 2008

[3] Wikipedia,"Trie",http://en.wikipedia.org/wiki/Trie

[4] Wikipedia,"Stemming",http://en.wikipedia.org/wiki/Stemming

[5] Wikipedia,"Stop words",http://en.wikipedia.org/wiki/Stop_word

[6]Martin Porter,"The Porter Stemming Algorithm",http://tartarus.org/~martin/PorterStemmer/

[7] Lextek International,"Stopword list 1",http://www.lextek.com/manuals/onix/stopwords1.html

Author list：

we discuss the algorithm and the realization details together

赵冰骞 The programmer of the Stop Words function, AVL algorithm and Trie algorithm.

林小钧 Tester and ppt maker.

雷露露 report writer.

Declaration

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

Signatures

赵冰骞 林小钧 雷露露