Roll Your Own Mini Search Engine

Group 1 王柯棣 赵伊蕾 沈韵泓

Date : 2022-03-19

Chapter 1: Introduction

Problem description:

In this project, we will create a mini search program handling with the works of Shakespeare. In this program, we need to deal with stopwords and do word-stemming work. Because only the root of word will show the meaning. The program is supposed to create inverted index over all the articles with word-stemming. And it can find out all the documents where the word/phrase comes from.

Data Structure or Algorithm:

In the foundation of inverted index, we can use a couple of ways.

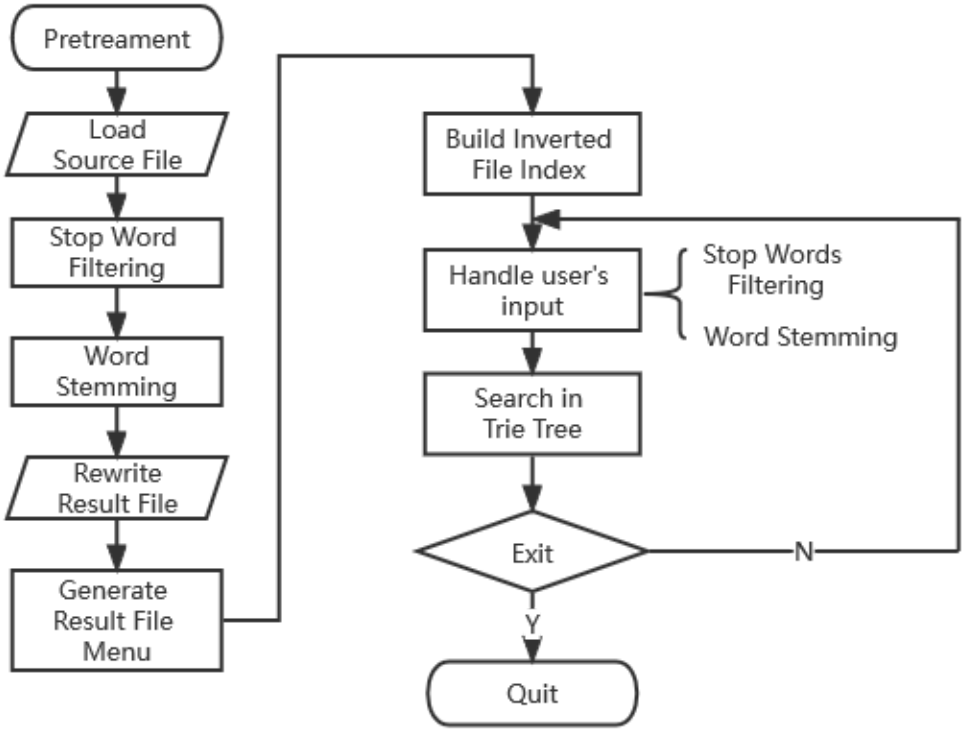
Firstly, **hash tables** are available. The word will be put into the hash table according to the key values. When the collision happens, there are a lot of ways to modify, which is depend on the frequency of collisions.

Secondly, we can use **trie** or some other tree like data structures, such as **AVL tree**. **BST** has the best advantage for comparison, so we can just put in all the words in a tree, so it will be quite easy to find the users' input words. Whatever the character of the tree is, the time complexity will be O(log n). But for the **AVL tree**, it can be faster because it will make the height of left tree and right tree at almost the same level.

Purpose of this project:

We can learn a lot from this project. Firstly, we will have an deep understanding of the concept of **index**. Just like the view in databases, index is something that shows user the things they want to know. And according to this project, we can master the tree data structure well.

Chapter 2: Data Structure / Algorithm Specification



2.1 Stop Words Filtering

2.1.1 Description

In this part, we are supposed to eliminate those stop words (also called noisy words sometimes) from the source file. As some words like 'Shakespeare' also has a high word frequency, we downloaded the stop words list (including all the words that appear frequently but meaningless for searching) form github, built a BST for the stop words and searched the words in source file with the list one by one to judge whether it is a stop word or not.

After finishing this part, we would have a result file free of all the stop words appearing in the given list, for example, if the original sentence is "We should go out tomorrow", there would be only "tomorrow" left in the result file.

Besides, this part would transfer all the capital letters into their lowercase.

2.1.2 Data Structure

In this part, we use **Binary Search Tree** to store all the stop words listed in *stopwrods.txt*, here comes the struct definition of the tree node:

```
struct tnode
{
    char *word;                 //point to the word stored in this
node
    struct tnode *left, *right; //point to the left/right child of
current node
};
```

2.1.3 Pseudo Code

We can roughly divide the stop word filtering into 2 parts - BST building and searching whether the word appears in the BST, the pseudo codes are as follows:

```
struct tnode *buildstoptree(char *word, struct tnode *p)
{
    if(p is empty)
    {
        initialize a new tree node, make it as p
        insert word into node p;
    }
    else if(word == the word stored in p) end the process;
    else if(word < the word stored in p) search the left child of node
p;
    else if(word > the word stored in p) search the right child of
node p;
}
```

After building the BST of stop words, if we want to judge whether a word is a stop word, we just have to find whether it is in the BST, if yes, then it's a stop words, otherwise it isn't a stop words, the pseudo code would be:

```
struct tnode*findstopword(char *word, struct tnode*p)
{
    //if foud return the pointer to the node, else return NULL
    if(p is NULL) return NULL, the word is not a stopword;
    if(w == the word  stored in p) return p, the word is a stop word;
    else if(word < the word stored in p) search the left child of node
p;
    else if(word > the word stored in p) search the right child of
node p;

    return NULL;
}
```

After doing so, we can figure out the stop words in the original file and eliminate it.

In the end, we should free the space, and the pseudo code would be:

```
void freetree(struct tnode* p)
{
    if(p is not NULL)
    {
        free p's left child;
        free p's right child;

        free the word stored in p;
        free p itself;
    }
}
```

2.2 Word Stemming

2.2.1 Description

It is a way to process a word so that only its stem or root form is left. For example, the word (process, processing,processes, processed) are going to get back to the word "process". And we also need to deal with the noun, adjective words and so on.

```
*   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
```

From the table we can find that there are lot of situations in this problem. And some words are unregualr, such as better -> good strike -> struck. So the final object of the algorithm of stemming words is to handle all the regular situations.

2.2.2 Pseudo Code

In this part, we use 5 steps to deal with the word stemming problem. In each step, we will make a judgement for the word and find the correct way to modify it.

```
void step1()
{
  case end with 's':
    if(end with'ies')
      change it into'i;
    else if(end with 'sses')
      change it into 'ss';
    else    delete's'
  case end with ('ed'||'ing') && !isConsonant:
        if(end with 'ated'||'ating')
      change it into 'ate';
    else if(end with 'bling'||'bled')
      change it into 'ble';
    else. if(end with 'izing'||'ized')
      change it into 'ize';
  case have some consonant letter:
    add 'e';
}
```

After the step 1 operation, we can deal with some simple transformation.

```
realization ->      realize
feed        ->      feed
agreed      ->      agree
```

The way to identify the different transformation way of words "feed" and "agreed" is according to the consonant judgment.

```
void step2()
{
  case end with 'ational'||'tional'||'enci'||'anci'||
    'izer'||'ble'||'bli'||'entli'||'alli'||'ent'......
    replace by 'ate','tion','ence','ance','ble','al'....
}
```

In the step 2, we made a lot of work on noun and adjective words. It is because they have typical affix formation. For example, noun words often end with 'tion', and adjective words often end with 'ive', 'tional'…… Then we can use these regular rules to handle.

```
void step3()
{
     case end with
'icate'||'ative'||'alize'||'iciti'||'ful'||'ness'......
    replace by 'ic','','al','ance','ble'....
}
```

In the step3 algorithm, the core is the same as step2 algorithm. But there are some different. The words in step3 sometimes need to be cut off in the end, but they will be replaced by in step2.

```
void step5()
{
    deal with the end 'e'.
  To make things easy, we delete it!
}
```

To make things easy, we decide to delete the word ending 'e', because for regular rules can not judge whether the word should or should not have an 'e' in the end.

2.3 Inverted File Index

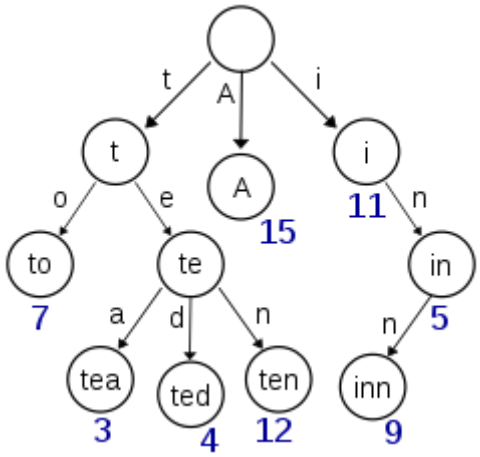
2.3.1 Word Counting & Index Generation

2.3.1.1 Data Structure - Trie

Specification:

Trie, which is called prefix tree, is a kind of k-ary tree which is often used for storing strings. In trie, all the children of a node have the same prefix, and the children node stores the key (char if we store strings in trie) following the prefix. So if we want to get a get the string, we only need to do a simple DFS, which can be known from the following picture. (In practice, we don’t need to store the whole string but the single char into the node, and the picture shows the whole string only to

show the principle. For example, in node 5 we only need to store ‘n’ inside, and travel root->11->5 to get string “in”.)



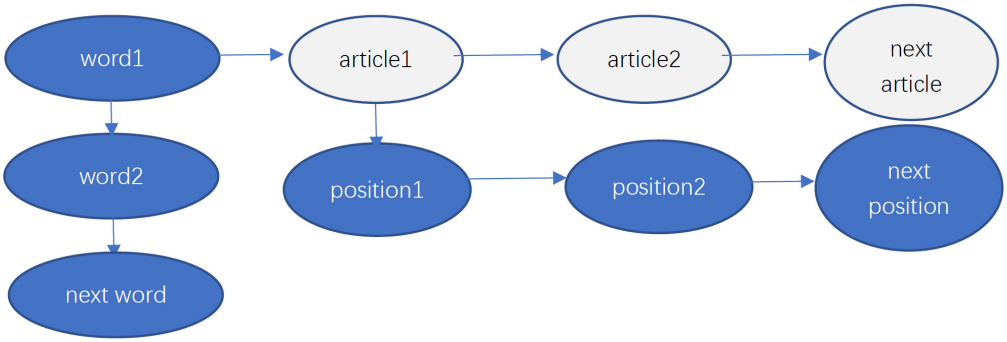
Graph 1. A sample of trie. (Picture from Wikipedia)

Compared with other data structures such as B-tree of Red Black tree, which stores the whole string into the node, trie obviously saves tons of space and is very easy to search the word. What’s more, it is exactly much easier to program. Meanwhile, in the node we can also store the index when encountering the last letter of a string, which enables the generation and storage of index.

In this program, the trie is 26-ary. But to save space, we use first_children-next_sibling pattern to implement the data structure.

2.3.1.2 Data Structure - Inverted file index / Linked list

In trie, we have given all the words a distinct index. Here we have an array wordList[] with length 1<<20, large enough to hold all English words in this word. The wordList[] is an array of series of dummy head pointers to linked lists which store the corresponding article containing the word with its position. For example, if word “ZJU” indexed 1 appeared in the first article as its first word, then wordList[1] will have a node marking the first article, and the article is also a head with its node marking its first word. All these shown below:



Pseudo Code:

Because we use firstchildren-nextsibling, there’s some change in the real program.

```
int searchInTrie(trie, word)
{
    ptr = trie;
    letter = word[0];
    while(letter != ‘\0’)
    {
        search the children of trie for letter;
```

```
        if successful:
            letter = the following letter, ptr = that children we
found;
        else return false;
    }
    if(letter == '\0')
        return ptr->index;
}
```

Insert is just like search, where the only difference is that we link new nodes when we fail to search a single letter, and continue to do it until we insert all letters. So we won't waste much space here to specify the procedure.

```
void indexGeneration(trie)
{
    static index = 0;
    while(word = inputting a new word)
    {
        search the word in the trie;
        if (the word already exists)
            continue;
        else
            insert the word into trie with ++index;
        insert the word into wordList[index of the word];
    }
}
```

2.3.2 Queue Processing

Data Structure - The linked list shown before

Specification:

Since our engine aims to provide more precise search, we'll not only test whether the inputting words exist or not but also test whether their positions are adjacent.

In this part the user will input a string, which will split into several words, with their index stored in an array. Once a word didn't take place in any article, we return with "Not Found". Following that, we will load their corresponding linked list storing the article they took place. Since the article are read in order, the linked lists are ranked in descending order automatically when they are built. So each time we search, we choose the minimum article index of all, and travel all the linked list until their index are all no larger than it. After that, if they're pointing at the same article, then we found an article which contains all the words, and go to check whether their positions are adjacent, and then move to test the next article. Else, if they're pointing at different articles, choose the minimum index continually until a pointer is NULL, which means all positions of a certain word have been searched thoroughly.

When it comes to testing whether the positions are adjacent, the progress is just like the search for common article, where the only difference is that in the previous we check whether their index equals, but now whether their positions are adjacent. Meanwhile, this time we will record the number of results and return it.

After all of them, if we succeed in finding the phrase, we'll insert it into the search result list in descending order, which is easy to implement.

Pseudo Code:

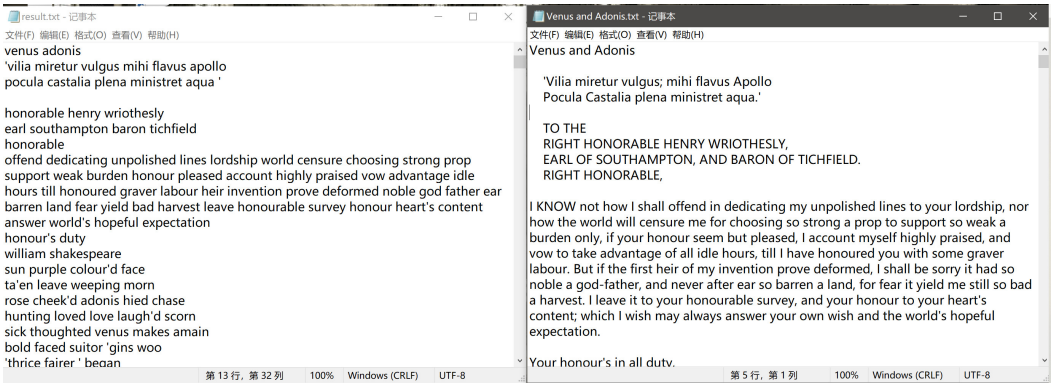
```
bool findCommonArticle(string)
{
    split the string into words;
    if(any word doesn't exist)
        return false;
    load the pointers toward the words into array list[];
    while(all pointers of list[] is not NULL)
    {
        min_index = the minimum index of all pointers;
        for all pointers in the list[], while(list[i]->index >
min_index)
            list[i] = list[i]->next;
        if(they point towards the same article)
        {
            find adjacent Position in that article;
            make all of them point toward the next article;
        }
         else continue;
    }
}
```

Since the process of finding adjacent is just like it, we won't specify it again.

Chapter 3: Testing Results

3.1 Stop Word Filtering Part

Purpose: Delete all the stop words & transfer all the capital letters into lowercase.
Status: Pass.
Result:



3.2 Word Stemming Part

```
1 warning generated.
realize
realiz
realizing
realiz
realization
realiz
realize
realiz
exit
```

```
careness
care
caring
care
cats
cat
meeting
meet
exit
```

Although some words are not exactly the same as the prototype of the word after processing, they are the same as the result of processing the prototype of the word. So if we process the operation of word-stemming for the words that the user enters to be retrieved, we can complete the search.

3.3 Word Count Part

```
+-------------------------------------------------------------------------+
|                        [SYSTEM] Word Count Result                       |
+-------------------------------------------------------------------------+
| [SYSTEM] With Stop Words  --- There is a total of 398676 words.         |
| [SYSTEM] Without Stop Words - There is a total of 181833 words.         |
| [SYSTEM] Distinct Words  ---- There is a total of 13080 words.          |
+-------------------------------------------------------------------------+
```

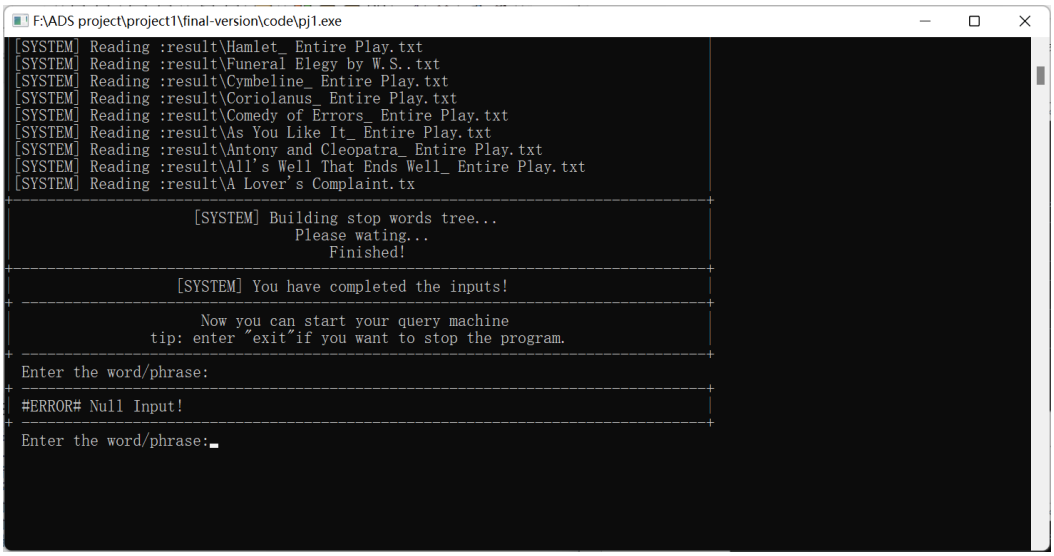
3.4 Other Test Results

Test 1: Special Null input.

Purpose: Test the program in the special minimum case.

Status: Pass.

Result:

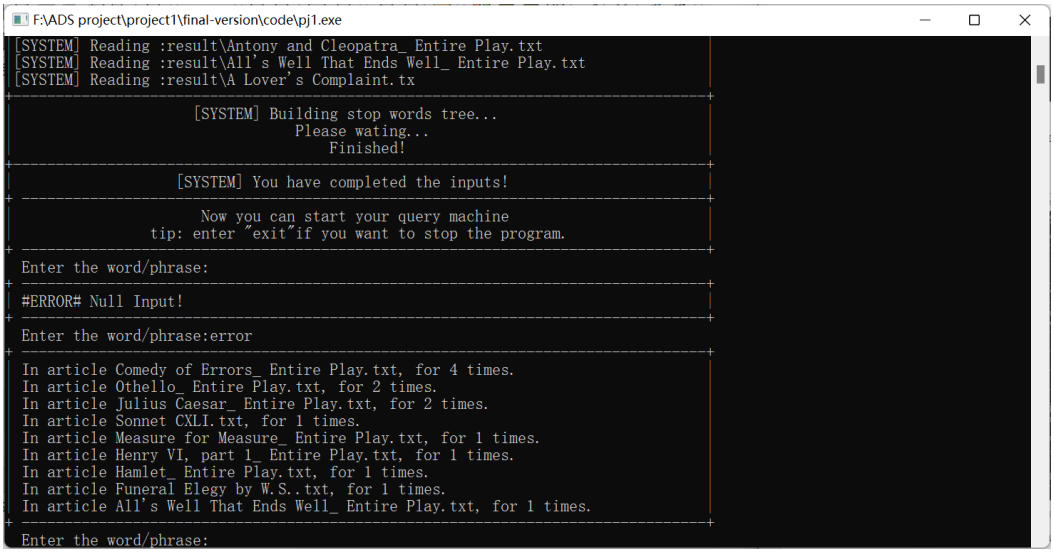


Test 2: Minimum size: a word.

Purpose: Test the program in the real minimum case.

Status: Pass.

Result:

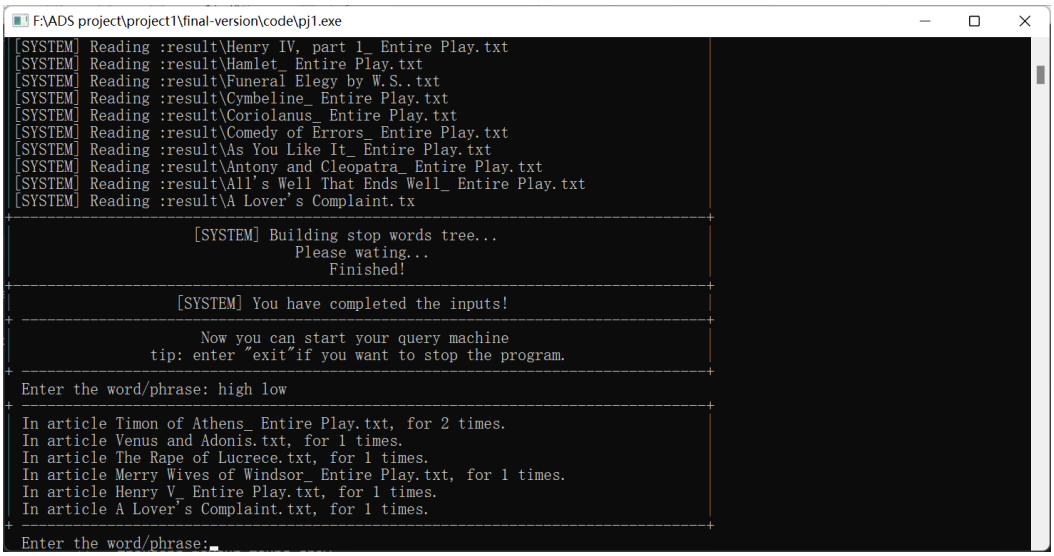


Test 3: Phrase search.

Purpose: Test one of the basic requirements of the program.

Status: Pass.

Result:

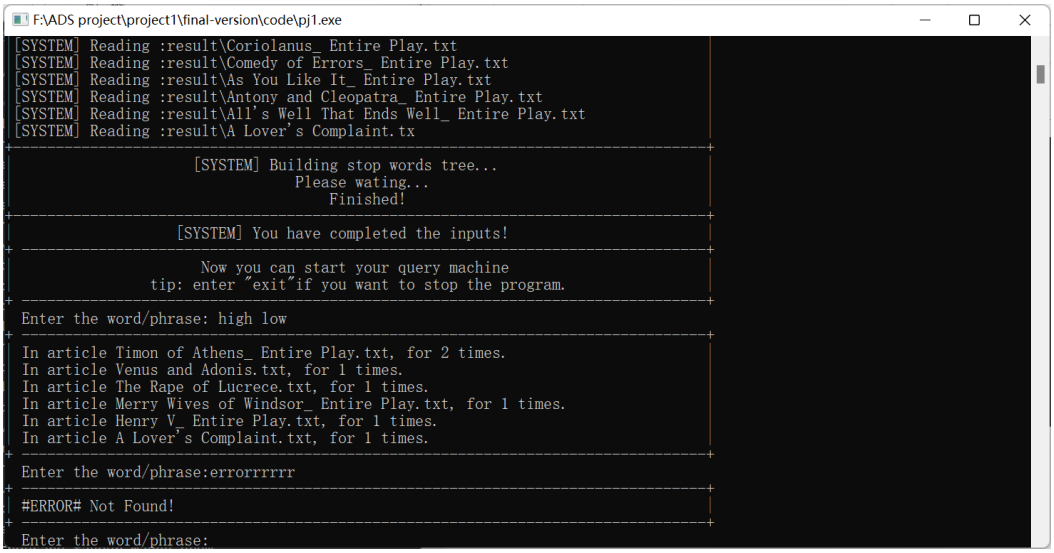


Test 4: The case “Not found”

Purpose: Check whether the program will output results wrongly.

Status: Pass.

Result:

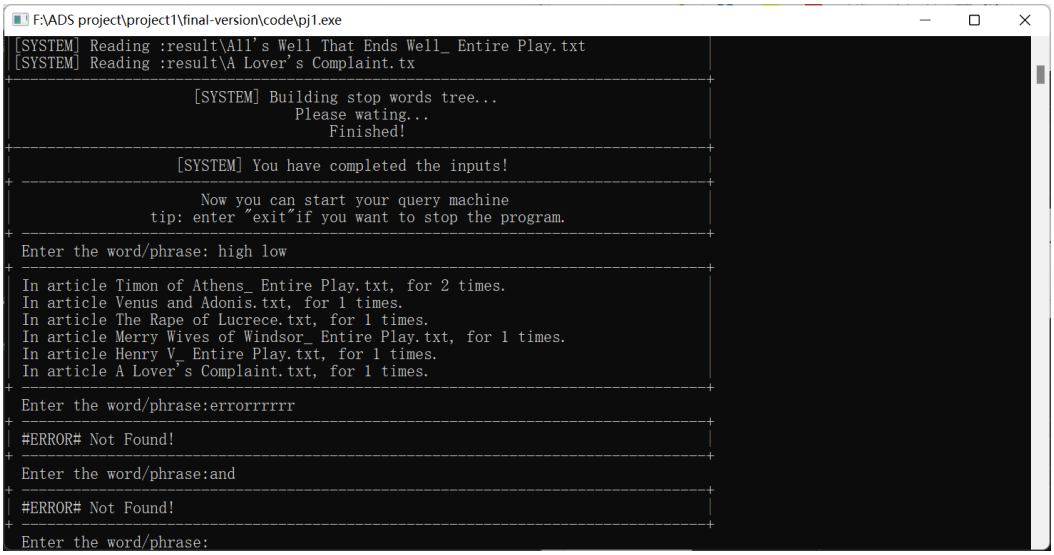


Test 5: The case when directly inputting a “stop word”

Purpose: Check whether the stop words are deleted from our database.

Status: Pass.

Result:

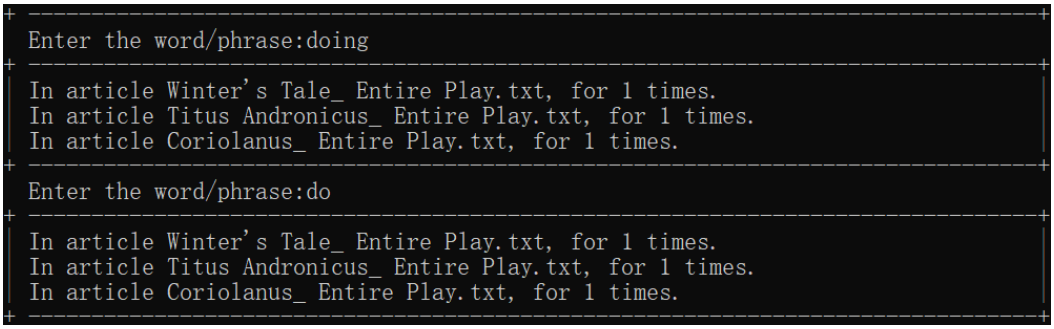


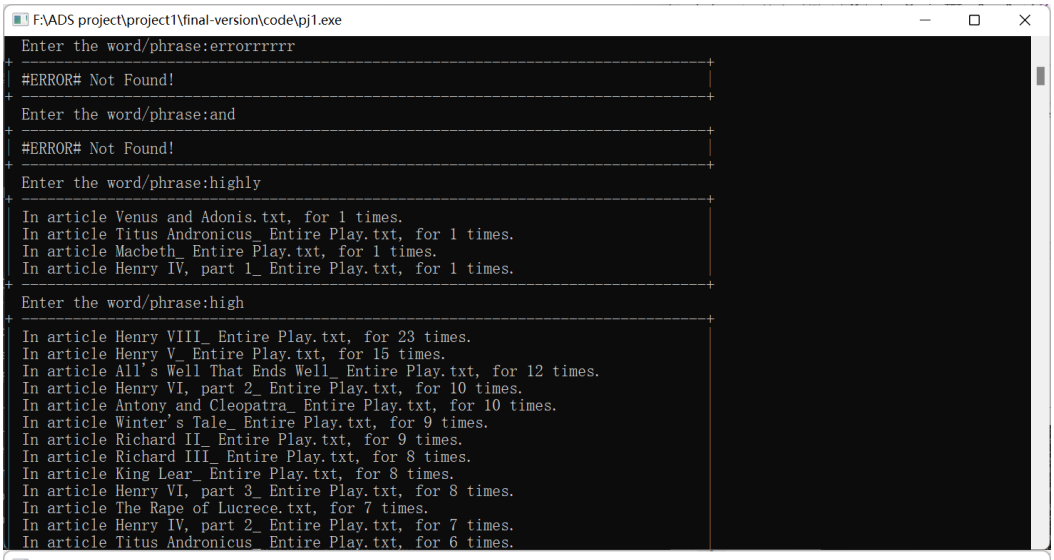
Test 6: Word stemming.

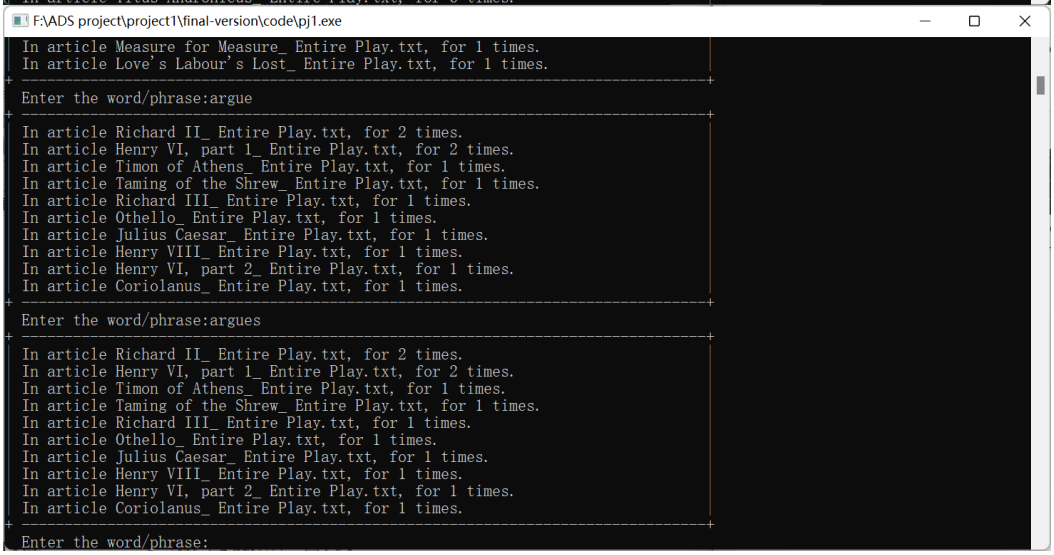
Purpose: Check whether the stemming works.

Status: Denying.

Result:





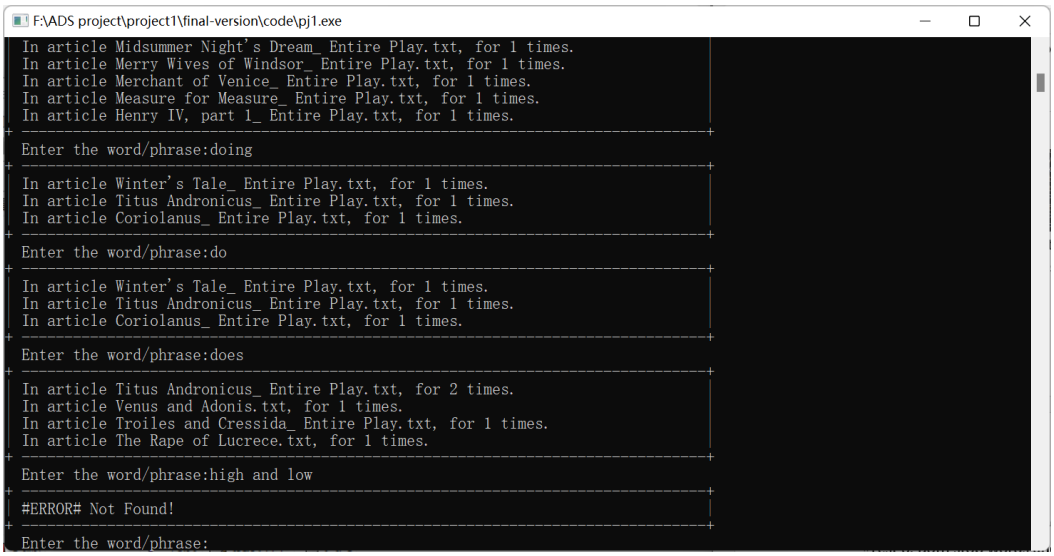


Test 7: Input phrase with stop words

Purpose: Check whether the stop words in the input will be deleted as well.

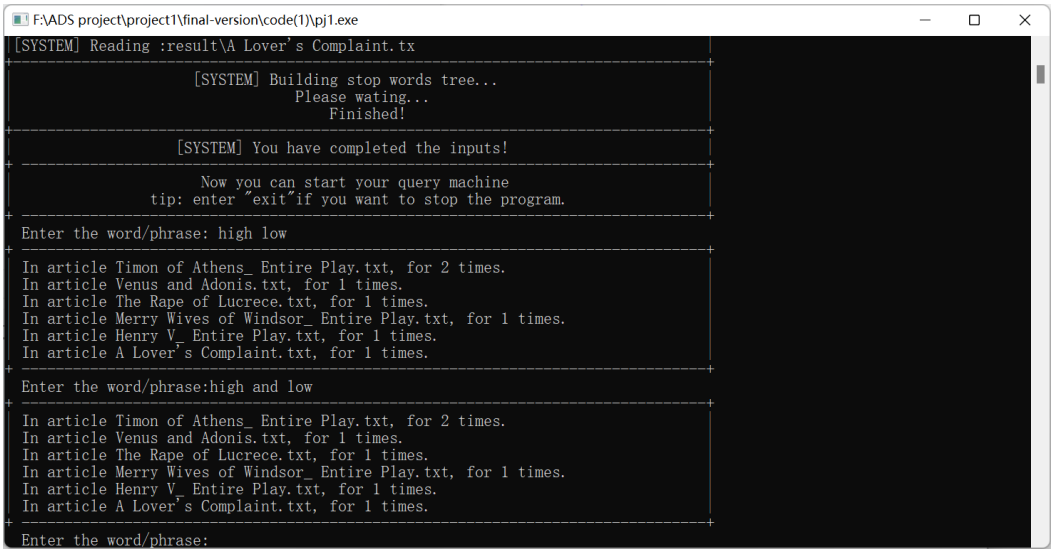
Status: Pass.

Result:



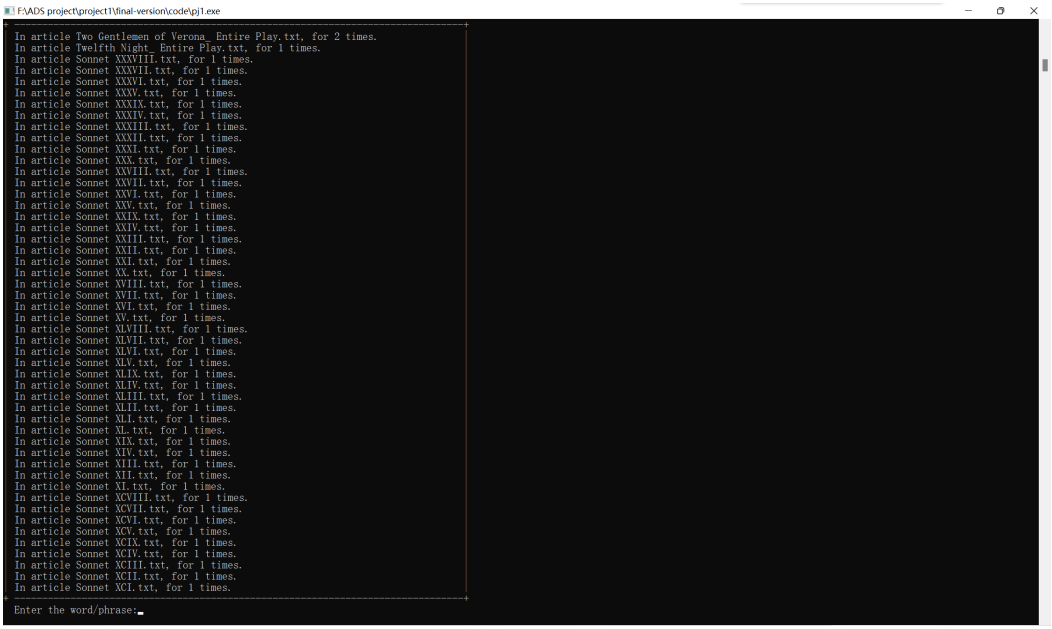
Test 8: Both stop words and stemming

Purpose: Test in an more complicated environment
Status: pass
Result:



Test 9: Test for thresholding

Purpose: Test when the output is more than 50.
Status: pass (Originally the poetry of sonnets has more than 100 articles, not to mention others)
Result:



Chapter 4: Analysis and Comments

4.1 Stop Word Filtering

4.1.1 Building BST

Time Complexity

As the time complexity is according to the insert sequence of stop words(it is obvious that the BST in this project is neither the best case - complete binary tree, or the worst case - a straight line), we just analyze the best case and worst case here.

- Best Case: Every time we insert a word into the binary tree, it is complete

  Assuming we are insert the n^{th} word, there are already n-1 nodes in the tree, and its height is floor(log(n-1)).

  Thus O(T(n)) = floor(log(2-1)) + floor(log(3-1)) + ... +floor(log(n-1)) = log(n!) <= $\log(n^n)$ = n*log(n)

- Worst Case: Every time we insert a word into the binary tree, it looks like a straight line

  Assuming we are insert the n^{th} word, there are already n-1 nodes in the tree, and its height is (n-1).

  Thus O(T(n)) = 1 + 2 + 3 + ... + (n-1) = n*(n-1)/2 = $O(n^2)$

Space Complexity

Assuming there are a total of n stop words we should stored in the tree, each word should be stored in a single tree node, thus the space complexity should be O(n).

4.1.2 Searching Stop Word

As we use the recursive algorithm to search the word, the space complexity of this part equals to its time complexity.

Assuming we have to n1 words to search and n0 stop words in BST:

- Best Case:

  As the height of the BST is floor(log(n0)), a single search would take O(floor(log(n0))) time and space.

  By adding them up, the total Time & Space should be O(n1*log(n0)).

- Worst Cse:

  As the height of the BST is n0, a single search would take O(n0) time and space.

  By adding them up, the total Time & Space should be O(n0*n1).

4.2 Inverted File Index Part

4.2.1 Word Counting & Index Generation

Time Complexity

Here we read all the words in all articles one by one. Each time we read a word, search it in trie, insert it into trie if needed, and mark its article and position. Now we'll analysis them.

First is the operation of trie. Since our search are all based on English words, the length of a single word can't be as large as it wants, so it can be seem as a constant. For each search, we'll search at most constant number of letter. And concerning each letter of the word, there's const number of distinct chars to choose from, so the time complexity of insert&search is O(1).

Next, mark the word's article and position. Because the articles are read in order, the linked lists are descending order. So if we want to find the article, it is either the article which the root is pointing at, or it didn't happen before, and so is the position in the article. Therefore, the insertion is also O(1).

Consequently, suppose there're n words across all articles, the time complexity is n*O(1) = O(n).

Space Complexity

There's not much to say about that. As shown before, we know the max length of all words is a constant, and there's 26 distinct letters, So the space of trie is O(1) in this project. Meanwhile, there's no denying that once there's a word, there's a position to record accordingly, so the space is O(n) where n is the number of total words. Therefore, total space complexity is O(n).

4.2.2 Queue Processing

Time Complexity

In this part, there's constant number of words inputted, each of them takes up space of O(n). In the worst case, their articles are all common and positions are all adjacent, which will take O(n) time to search them. For instance, there's a worst case when all the articles are composed of "so hard so hard so hard…", and we want a query of "so hard", all the positions will be searched. So the time complexity is obviously O(n).

However, in practice the inputs are much more sparse, so the query will be much faster.

Space Complexity

We don't need extra space in this process, only an array to store inputs, an array to store their index, and an array to load their according linked list. So space complexity here is O(1).

Totally, the Time Complexity is O(n) + O(n) = O(n), while the space complexity is O(n) + O(1) = O(n).

Appendix: Source Code

```
//#include "head.h"
#include "stopwords.h"
#include "indexWord.h"

static int n_total_word_st = 0;//total word number (with stopwords,
with duplicates)
static int n_total_word = 0;   //total word number (without stopwords,
with duplicates)
static char b[100];
static int k;
static int k0;
/* j is a general offset into the string */
static int j;
struct tnode *root =
buildstoptree("..\\documents//testfile&output//stopwords.txt", root);
void stem(int index,int position);
int main()
{
    //Doing pretreatment - stop words filtering & word stemming
```

```
    pretreatment();

    FILE *fp;
    int i,n;
    char * title_print;
    fp = fopen("..\\documents//testfile&output//resultlist.txt", "r");
    if(!fp)
    {
        printf("| #ERROR# Failed to open resultlist.txt
                            |\n");
        printf("+-------------------------------------------------------
---------------------------+\n");
        system("pause");
        return -1;
    }
    Trie = (Tree) malloc (sizeof(trie_node));
    Trie->ID = 0;;
    Trie->FirstChild = Trie->NextSibling = NULL;
    ReadTitle(fp);
    fclose(fp);

    char title[MAX_TITLE_LENGTH];

    FILE *fpTitle;
    fpTitle = fopen("..\\documents//testfile&output//resultlist.txt",
"r");
    printf("+-------------------------------------------------------------
------------------------+\n");
    printf("|                              [SYSTEM] Loading files...
                        |\n");
    printf("+-------------------------------------------------------------
------------------------+\n");

    while(!feof(fpTitle))
    {
        fgets(title, MAX_TITLE_LENGTH, fpTitle);
        if(title[strlen(title)-1] == '\n')
            title[strlen(title)-1] = '\0';
        printf("|[SYSTEM] Reading :");
        fp = fopen(title, "r");
        titlePrint(title);


        ReadArticle(fp);
        fclose(fp);
    }
    int len_blank;
    int len_figure = 0;
    int tmp;
    //output word count
    printf("+-------------------------------------------------------------
------------------------+\n");
    printf("|                              [SYSTEM] Word Count Result
                         |\n");
```

```
    printf("+-------------------------------------------------------------
------------------------+\n");
    printf("| [SYSTEM] With Stop Words  --- There is a total of %d
words.", n_total_word_st);
    tmp = n_total_word_st;
    while(tmp > 0)
    {
        len_figure++;
        tmp /= 10;
    }
    len_blank  = 23 - len_figure;
    for(int i = 0; i < len_blank; i++) printf(" ");
    printf("|\n");
    printf("| [SYSTEM] Without Stop Words - There is a total of %d
words.", n_total_word);
    len_figure = 0;
    tmp = n_total_word;
    while(tmp > 0)
    {
        len_figure++;
        tmp /= 10;
    }
    len_blank  = 23 - len_figure;
    for(int i = 0; i < len_blank; i++) printf(" ");
    printf("|\n");
    printf("| [SYSTEM] Distinct Words  ---- There is a total of %d
words.", wordIndex);
    len_figure = 0;
    tmp = wordIndex;
    while(tmp > 0)
    {
        len_figure++;
        tmp /= 10;
    }
    len_blank  = 23 - len_figure;
    for(int i = 0; i < len_blank; i++) printf(" ");
    printf("|\n");


    //Build BST of stop words to handle user's input
    printf("+-------------------------------------------------------------
------------------------+\n");
    printf("|                       [SYSTEM] Building stop words tree...
                      |\n");
    printf("|                                    Please wating...
                      |\n");
    //Reading stop words from stopwords.txt
    printf("|                                         Finished!
                       |\n");
    printf("+-------------------------------------------------------------
------------------------+\n");
    printf("|                     [SYSTEM] You have completed the
inputs!                       |\n");
    printf("+ -------------------------------------------------------------
------------------------+\n");
```

```
    printf("|                        Now you can start your query
machine                          |\n");
    printf("|                tip: enter \"exit\"if you want to stop
the program.                |\n");
    printf("+ ------------------------------------------------------------
------------------------+\n");
    char word[MAX_WORD_LENGTH];
    printf("  Enter the word/phrase: ");
    gets(b);
    printf("+ ------------------------------------------------------------
------------------------+\n");
    while(strcmp(b,"@exit"))
    {
        findPositionOfArray(b);
        printf("+ ------------------------------------------------------
----------------------------+\n");
        printf("  Enter the word/phrase:");
        gets(b);
        printf("+ ------------------------------------------------------
----------------------------+\n");
    }

    //When exit, delete stop words tree and print information
    printf("|                       [SYSTEM] Deleting stop words tree...
                      |\n");
    printf("|                                    Please wating...
                      |\n");
    freetree(root);
    printf("|                                       Finished!
                      |\n");
    printf("+-----------------------------------------------------------
------------------------+\n");
    printf("|                              [SYSTEM] Thank you for using!
                       |\n");
    printf("+ ------------------------------------------------------------
------------------------+\n");
    system("pause");
    return 0;
}
void titlePrint(char * title)
{
    char s[MAX_TITLE_LENGTH];
    int len_s;
    int len_blank;
    strcpy(s,title);
    int i=0,n=strlen(s);
    for(i=0;i<n;i++)
        if(s[i]=='\\')
            break;
    for(j=i;j<=n;j++)
        s[j-i]=s[j+1];
    len_s = strlen(s);
    s[len_s] = '\0';
    printf("%s",s);
        len_blank = 63 - len_s;
```

```
    for(i = len_blank; i > 0; i--) printf(" ");
    printf("|\n");
    return;
}

void ReadTitle(FILE *fp)
{
    if(feof(fp))    return ;
    char input[MAX_TITLE_LENGTH];
    char title[MAX_TITLE_LENGTH];
    int n;
    while(!feof(fp))    //copy from the last char until we get the
first '\'
    {
        fgets(input, MAX_TITLE_LENGTH, fp);
        n = strlen(input);
        if(input[n-1] == '\n')
            input[--n] = '\0';
        while(input[n] != '\\')
        {
            title[n] = input[n];
            n--;
        }
        strcpy(articleList[++articleCount], title+n+1);
    }
    articleCount = 0;
}

void ReadArticle(FILE* fp)
{
    if(feof(fp))    return ;
    wordCount = 0;  //a new article, and no word has been read
    char word[MAX_WORD_LENGTH];
    articleCount++;

    int index;
    while(!feof(fp))    //while the article is not read thoroughly
    {
        fscanf(fp, "%s", word);
        wordCount++;    //we have read a new word
        n_total_word++;
        index = findIndexOfWord(word);
        if(!index)  //the word haven't appeared in the trie database
        {
            insertIntoTrie(word);
            index = wordIndex;
        }
        insertIntoArticle(word, index);
    }
}

//return the index of the word if found
//if not found, return 0
int findIndexOfWord(char word[])
{
```

```
    int i;
    Tree P = Trie->FirstChild;  //node Trie is dummy
    Tree parent = P;
    for(i = 0; word[i]; i++)
    {
        while(P && P->key != word[i])
            P = P->NextSibling;     //search whether this letter
exists
        if(!P)   return 0;  //when the word's postfix lacks, P is NULL

        parent = P;     //update P's parent
        P = P->FirstChild;  //go to search the next letter
    }

    if(word[i] == '\0') //when the word is a prefix of the other word,
P is behind the last letter
        return parent->ID;  //since P moves as we search, it is parent
that points towards the last letter
    else return 0;  //not found
}

//insert a NEW word into the trie
void insertIntoTrie(char word[])
{
    Tree P = Trie->FirstChild;  //node Trie is dummy
    Tree ptr, pre, parent = Trie;
    int i, j;
    for(i = 0; word[i]; i++)
    {
        pre = P;
        //search word[i]
        while(P && P->key != word[i])
        {
            pre = P;
            P = P->NextSibling;
        }
        //word[i] doesn't exist
        if(!P)  //the letter not exist in the trie
        {
            ptr = (Tree) malloc (sizeof(trie_node));
            ptr->ID = 0;
            ptr->key = word[i];
            ptr->FirstChild = ptr->NextSibling = NULL;

            if(pre) //add a new letter in a single line
                pre->NextSibling = ptr;
            else parent->FirstChild = ptr;   //we come into a new line

            P = ptr;    //go to insert the next letter
        }
        //go to search the next line
        parent = P;
        P = P->FirstChild;
    }
    parent->ID = ++wordIndex;   //update the index
```

```
}

void insertIntoArticle(char word[], int index)
{
    article particle;
    position pposition;
    if(wordList[index] == NULL) //the word haven't appeared
    {
        //malloc the dummy root
        wordList[index] = (article) malloc (sizeof(article_node));
        wordList[index]->firstPosition = NULL;
        wordList[index]->nextArticle = NULL;
        wordList[index]->frequency = 0; //in dummy head, we count the
number of articles
        wordList[index]->indexOfArticle = 0;
        //malloc the new article
        particle = (article) malloc (sizeof(article_node));
        particle->firstPosition = NULL;
        particle->frequency = 0;
        particle->nextArticle = NULL;
        particle->indexOfArticle = articleCount;

        wordList[index]->nextArticle = particle;
        wordList[index]->frequency++;
    }
    else    //the word has appeared, so we need to search whether it
appeared in this article
    {
        //travel the linkedlist to test whether the word has appeared
in this article
        //since the article is read in order, the article to insert
into is always the first node except the dummy head
        if(wordList[index]->nextArticle->indexOfArticle !=
articleCount)    //the article didn't have this word before
        {
            //insert a new article
            particle = (article) malloc (sizeof(article_node));
            particle->firstPosition = NULL;
            particle->frequency = 0;
            particle->nextArticle = NULL;
            particle->indexOfArticle = articleCount;
            particle->nextArticle = wordList[index]->nextArticle;

            wordList[index]->nextArticle = particle;
            wordList[index]->frequency++;
        }
        else    //the article already has this word
            particle = wordList[index]->nextArticle;
    }
    //now we insert the word into the article
    particle->frequency++;
    pposition = (position) malloc (sizeof(position_node));
    pposition->place = wordCount;
    pposition->next = particle->firstPosition;
    particle->firstPosition = pposition;
```

```
}

void findPositionOfArray(char word[])
{
    int i = 0;
    while(word[i] == ' ' || word[i] == '\n')    //delete the empty
chars
        i++;
    if(word[i] == '\0')
    {
        printf("| #ERROR# Null Input!
                            |\n");
        return ;
    }

    //turning all the words into lowercase
    i = 0;
    while(word[i] != '\0')
    {
        if(word[i]>='A' && word[i]<='Z') word[i]+=32;
        i++;
    }

    int index[MAX_SEARCH_WORD]; //store the index of the inputting
words
    article arrayWordList[MAX_SEARCH_WORD]; //pointers towards the
word records

    int count = 0;  //mark the position after the last word

    //call iteratively and store the index of word into array index
    char* pword = word;
    bool isStopWord;
    while(pword)
    {
        pword = split(index+(count++), pword, &isStopWord);
        if(isStopWord)  count--;
    }

    if(count == 0)
    {
        printf("| #ERROR# The input are all STOP WORDS!
                            |\n");
        return;
    }

    for(i = 0; i < count; i++)
    {
        if(!index[i])   //if some words are not in any article, return
not found directly
        {
            printf("| #ERROR# Not Found!
                                |\n");
            return;
        }
```

```
        arrayWordList[i] = wordList[index[i]]->nextArticle;     //load
the words
    }

    bool found = getCommonArticle(arrayWordList, count);    //find the
phrase
    if(!found)
    {
        printf("| #ERROR# Not Found!
                            |\n");    //specially deal with not found
    }
}

//store the index of the first word in arrat into index[0], and return
the pointer to the first letter of the next word
//if no word anymore after split, return NULL
char* split(int index[], char word[], bool *isStopWord)
{
    int i = 0;
    char result[MAX_WORD_LENGTH];
    while(word[i] != ' ' && word[i] == '\n')
        i++;
    //copy the first word
    while(word[i] != ' ' && word[i] != '\0')
    {
        result[i] = word[i];
        i++;
    }
    result[i] = '\0';
    struct tnode *querry = NULL;
    querry = findstopword(root, result);
    if(querry == NULL)
    {
        *isStopWord = 0;
        int n = strlen(result), t;
        strcpy(b, result);
        stem(0, n-1);
        for(t = 0; t <= k; t++)
            result[t] = b[t];
        result[t] = '\0';
        index[0] = findIndexOfWord(result);
    }
    else
    {
    //  printf("%s",result);
        *isStopWord = 1;
    }
    if(word[i] == '\0') return NULL;    //the search array has been
fully splited
    else return word+i+1;   //return the pointer to the next word

}

//search the phrase and return whether we succeed
bool getCommonArticle(article arrayWordList[], int count)
```

```
{
    article result = NULL;
    bool found = false; //not found at first
    int min_fileindex;
    int i;
    int times_in_this_article;

    while(!isArticleNull(arrayWordList, count))    //in case the last
article of a certain word is a common article
    {
        //if the words are not in the same article now, loop until
they have the same article
        while(!allArticleEqual(arrayWordList, count))
        {
            min_fileindex = findMinArticleIndex(arrayWordList, count);
 //find the minimum one to save time
            for(i = 0; i < count; i++)
            {
                while(arrayWordList[i] && arrayWordList[i]-
>indexOfArticle > min_fileindex)
                    arrayWordList[i] = arrayWordList[i]->nextArticle;
                //when one word's appearances has been searched
thoroughly, we exit the function
                if(!arrayWordList[i])
                {
                    printResults(result);
                    return found;       //the search of one word is
finished, and no common article following
                }
            }
        }
        //if all the words are in the same article, we go to search
whether their position are adjacent
        times_in_this_article = findAdjPosition(arrayWordList, count);
 //update the found status

        found = found == true ? true : times_in_this_article;
        if(times_in_this_article > 0)
            result = findNewArticle(result, arrayWordList[0]-
>indexOfArticle, times_in_this_article);

        for(i = 0; i < count; i++)
            arrayWordList[i] = arrayWordList[i]->nextArticle;
//search whether there's next article having the word array
    }
    printResults(result);   //the memory has already been freed
    return found;
}

//when we find a new article, insert it into the result linked list in
descending order
article findNewArticle(article result, int index, int
times_in_this_article)
{
    if(result == NULL)  //null list
```

```
    {
        result = (article) malloc (sizeof(article_node));
        result->firstPosition = NULL;
        result->nextArticle = NULL;
        result->frequency = 0;  //used to record the number of search
results
    }
    article ptr;    //new article node
    ptr = (article) malloc (sizeof(article_node));
    ptr->frequency = times_in_this_article;
    ptr->nextArticle = NULL;
    ptr->firstPosition = NULL;
    ptr->indexOfArticle = index;
    article p = result;
    //now search where to insert into
    while(p->nextArticle && p->nextArticle->frequency > ptr-
>frequency)
        p = p->nextArticle;
    ptr->nextArticle = p->nextArticle;
    p->nextArticle = ptr;
    result->frequency++;    //find a new article
}

//thresholding, print top at most 50 results
void printResults(article result)
{
    int i = 0;
    int printCount = 0;
    int len_blank;
    int len_figure;
    article ptr = result->nextArticle;

    while(printCount++ < 50 && ptr) //have results remaining and no
more than 50 results
    {
        if(ptr->frequency > 0)
        {
            printf("| In article %s, for %d times.", articleList[ptr-
>indexOfArticle], ptr->frequency);
            len_figure = 0;
            while(ptr->frequency)
            {
                len_figure++;
                ptr->frequency /= 10;
            }
            len_blank = 56 - strlen(articleList[ptr->indexOfArticle])
- len_figure;
            for(i = len_blank; i > 0; i--) printf(" ");
            printf("|\n");
        }

        result->nextArticle = ptr->nextArticle;
        free(ptr);
        ptr = result->nextArticle;
    }
```

```
    while(ptr)  //free the remaining search results
    {
        result->nextArticle = ptr->nextArticle;
        free(ptr);
        ptr = result->nextArticle;
    }
    free(result);   //free the root
}

//judge whether the article array has NULL value
bool isArticleNull(article arrayWordList[], int count)
{
    int i;
    for(i = 0; i < count; i++)
        if(!arrayWordList[i])
            return true;
    return false;
}

//judge whether the article array has the same article
bool allArticleEqual(article arrayWordList[], int count)
{
    int i;
    int index = arrayWordList[0]->indexOfArticle;
    for(i = 0; i < count; i++)
        if(arrayWordList[i]->indexOfArticle != index)
            return false;
    return true;
}

//find the min article index of all
int findMinArticleIndex(article arrayWordList[], int count)
{
    int min = arrayWordList[0]->indexOfArticle;
    int i;
    for(i = 1; i < count; i++)
        min = arrayWordList[i]->indexOfArticle < min ?
arrayWordList[i]->indexOfArticle : min;
    return min;
}

//search whether the words have adjacent position
int findAdjPosition(article arrayWordList[], int count)
{
    position positionList[count];
    int i, min_index, times = 0;
    for(i = 0; i < count; i++)
        positionList[i] = arrayWordList[i]->firstPosition;
    //check in case one word's position is the last one
    while(!isPositionNull(positionList, count))
    {
        while(!allPositionEqual(positionList, count))
        {
            min_index = findMinPosition(positionList, count);
            for(i = 0; i < count; i++)
```

```
            {
                while(positionList[i] && positionList[i]->place-i >
min_index)
                    positionList[i] = positionList[i]->next;
                //one word's position is over, and return the result
                if(!positionList[i])
                    return times;
            }
        }
        //finally we found one……
        times++;
        for(i = 0; i < count; i++)
            positionList[i] = positionList[i]->next;
    }
    return times;
}

//judge whether the position array has NULL value
bool isPositionNull(position positionList[], int count)
{
    int i;
    for(i = 0; i < count; i++)
        if(!positionList[i])
            return true;
    return false;
}

//judge whether the position array has the adjacent position
bool allPositionEqual(position positionList[], int count)
{
    int i;
    int index = positionList[0]->place;
    for(i = 0; i < count; i++)
        if(positionList[i]->place != index+i)
            return false;
    return true;
}

//find the min position index of all words
int findMinPosition(position positionList[], int count)
{
    int min = positionList[0]->place;
    int i;
    for(i = 1; i < count; i++)
        min = positionList[i]->place-i < min ? positionList[i]->place-
i : min;
    return min;
}


/*It's used to judge whether the letter is consonant, because the word
stemming operations have regular rules
which has some relation with whether the letter is consonant.
If the letter is 'a' 'e' 'i' 'o' 'u' then return FALSE
else return TRUE
```

```
*/
static int isConsonant(int index)
{
  switch (b[index])
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
      return false;
    case 'y':
      return (index == k0) ? true : !isConsonant(index - 1);
    default:
      return true;
  }
}

/* Measure 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 getMeasure()
{
  int position;
  int index;

  position = 0;
  index = k0;

  while (true)
  {
    if (index > j)
      return position;
    if (!isConsonant(index))
      break;
    index++;
  }

  index++;
  while(true)
  {
    while (true)
    {
      if (index > j)
        return position;
      if (isConsonant(index))
        break;
```

```
      index++;
    }
    index++;
    position++;

    while (true)
    {
      if (index > j)
        return position;
      if (!isConsonant(index))
        break;

      index++;
    }
    index++;
  }
}

/* `TRUE` when `k0, ... j` contains a vowel. */
static int vowelInStem()
{
  int index;

  index = k0 - 1;
  while (++index <= j)
  {
    if (!isConsonant(index))
    {
      return true;
    }
  }
  return false;
}

/* `TRUE` when `j` and `(j-1)` are the same consonant. */
static int isDoubleConsonant(int index)
{
  if (b[index] != b[index - 1])
  {
    return false;
  }
  return isConsonant(index);
}

/* `TRUE` when `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.
 *
 * `cav(e)`, `lov(e)`, `hop(e)`, `crim(e)`, but `snow`,
 * `box`, `tray`.
 */
static int cvc(int index)
{
  int character;
```

```c
  if (index < k0 + 2 || !isConsonant(index) || isConsonant(index - 1)
|| !isConsonant(index - 2))
    return false;
  character = b[index];
  if (character == 'w' || character == 'x' || character == 'y')
    return false;
  return true;
}

/* `ends(s)` is `TRUE` when `k0, ...k` ends with `value`. */
static int ends(const char *value)
{
  int length = value[0];
  /* Tiny speed-up. */
  if (value[length] != b[k])
    return false;
  if (length > k - k0 + 1)
    return false;
  if (memcmp(b + k - length + 1, value + 1, length) != 0)
    return false;
  j = k - length;

  return true;
}

/* `setTo(value)` sets `(j + 1), ...k` to the characters in
 * `value`, readjusting `k`. */
static void setTo(const char *value)
{
  int length = value[0];
  memmove(b + j + 1, value + 1, length);
  k = j + length;
}

/* Set string. */
static void replace(const char *value)
{
  if (getMeasure() > 0)
    setTo(value);

}

/* `step1ab()` gets rid of plurals, `-ed`, `-ing`.
 *
 * Such as:
 *
 *   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()
{
  int character;

  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 (getMeasure() > 0)
      k--;
      return;
  }
  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 (isDoubleConsonant(k))
    {
      k--;
      character = b[k];
```

```
      if (character == 'l' || character == 's' || character == 'z'||
character == 'r') {
        k++;
      }
    }
    else if (getMeasure() == 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
 * getMeasure() > 0. */
static void step2()
{
  switch (b[k - 1])
  {
    case 'a':
      if (ends("\07" "ational"))
      {
        replace("\03" "ate");
        break;
      }
      if (ends("\06" "tional"))
      {
        replace("\04" "tion");
        break;
      }
      break;
    case 'c':
      if (ends("\04" "enci")) {
        replace("\04" "ence");
        break;
      }

      if (ends("\04" "anci"))
      {
        replace("\04" "ance");
        break;
      }

      break;
```

```
    case 'e':
      if (ends("\04" "izer"))
      {
        replace("\03" "ize");
        break;
      }
      break;
    case 'l':
      /* --DEPARTURE--: To match the published algorithm,
       * replace this line with:
       *
       * ```
       * if (ends("\04" "abli")) {
       *     replace("\04" "able");
       *
       *     break;
       * }
       * ```
       */
      if (ends("\03" "bli")) {
        replace("\03" "ble");
        break;
      }

      if (ends("\04" "alli")) {
        replace("\02" "al");
        break;
      }

      if (ends("\05" "entli")) {
        replace("\03" "ent");
        break;
      }

      if (ends("\03" "eli")) {
        replace("\01" "e");
        break;
      }

      if (ends("\05" "ousli")) {
        replace("\03" "ous");
        break;
      }

      break;
    case 'o':
      if (ends("\07" "ization")) {
        replace("\03" "ize");
        break;
      }

      if (ends("\05" "ation")) {
        replace("\03" "ate");
        break;
      }
```

```
      if (ends("\04" "ator")) {
        replace("\03" "ate");
        break;
      }

      break;
    case 's':
      if (ends("\05" "alism")) {
        replace("\02" "al");
        break;
      }

      if (ends("\07" "iveness")) {
        replace("\03" "ive");
        break;
      }

      if (ends("\07" "fulness")) {
        replace("\03" "ful");
        break;
      }

      if (ends("\07" "ousness")) {
        replace("\03" "ous");
        break;
      }

      break;
    case 't':
      if (ends("\05" "aliti")) {
        replace("\02" "al");
        break;
      }

      if (ends("\05" "iviti")) {
        replace("\03" "ive");
        break;
      }

      if (ends("\06" "biliti")) {
        replace("\03" "ble");
        break;
      }

      break;
    /* --DEPARTURE--: To match the published algorithm, delete this
line. */
    case 'g':
      if (ends("\04" "logi")) {
        replace("\03" "log");
        break;
      }
  }
}
```

```
/* `step3()` deals with -ic-, -full, -ness etc.
 * similar strategy to step2. */
static void step3()
{
  switch (b[k])
  {
    case 'e':
      if (ends("\05" "icate")) {
        replace("\02" "ic");
        break;
      }

      if (ends("\05" "ative")) {
        replace("\00" "");
        break;
      }

      if (ends("\05" "alize")) {
        replace("\02" "al");
        break;
      }

      break;
    case 'i':
      if (ends("\05" "iciti")) {
        replace("\02" "ic");
        break;
      }

      break;
    case 'l':
      if (ends("\04" "ical")) {
        replace("\02" "ic");
        break;
      }

      if (ends("\03" "ful")) {
        replace("\00" "");
        break;
      }

      break;
    case 's':
      if (ends("\04" "ness")) {
        replace("\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;
      }

      /* takes care of -ous */
      if (ends("\02" "ou")) {
        break;
      }

      return;
    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 (getMeasure() > 1)
  {
    k = j;
  }
```

```
}

/* `step5()` removes a final `-e` if `getMeasure()` is
 * greater than `1`, and changes `-ll` to `-l` if
 * `getMeasure()` is greater than `1`. */
static void step5()
{
  int a;

  j = k;

  if (b[k] == 'e') {
    a = getMeasure();

    if (a > 1 || (a == 1 && !cvc(k - 1))) {
      k--;
    }
  }

  if (b[k] == 'l' && isDoubleConsonant(k) && getMeasure() > 1) {
    k--;
  }
}

/* In `stem(i, j)`
 * Typically, `i` is zero and `j` is the offset to the
 * last character of a string, `(b[j + 1] == '\0')`.
 * The stemmer adjusts the characters `b[i]` ... `b[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. */
void stem(int index, int position)
{
  k = position;
  k0 = index;
  /* 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();
  }
}
```

```
/* read stoplist into binary tree, expects one entry per line */
struct tnode *buildstoptree(char *fname, struct tnode *p)
{
    FILE *fp = {0};
    char line[MAXLINE];
    int len = 0, lcount = 0;

    fp = fopen(fname, "r");
    if(fp == NULL)
    {
        printf("+-------------------------------------------------------
---------------------------+\n");
        printf("| #ERROR# Fail to open stopwords.txt
                            |\n");
        printf("+-------------------------------------------------------
---------------------------+\n");
        return NULL;
    }
    while(fgets(line, MAXLINE, fp) != NULL)
    {
        len = strlen(line);
        if(len < STMINLEN)  continue;
        else                lcount++;

        if(line[len - 1] == '\n')  line[--len] = '\0';

        p = addtree(p, line);
    }
    if(lcount == 0)
    {
        printf("+-------------------------------------------------------
---------------------------+\n");
        printf("| #ERROR# Zero stopwords..
                            |\n");
        printf("+-------------------------------------------------------
---------------------------+\n");
        return NULL;
    }

    fclose(fp);
    return p;
}

/* split string into tokens, return token array */
char **split(char *string, char *delim)
{
    //split the whole line into single words(tokens)
    char **tokens = NULL;
    char *working = NULL;
    char *token = NULL;
    int idx = 0;

    tokens  = (char**)malloc(sizeof(char *) * MAXTOKENS);
    if(tokens == NULL)  return NULL;
```

```
    working = (char*)malloc(sizeof(char) * strlen(string) + 1);
    if(working == NULL)  return NULL;

    /* to make sure, copy string to a safe place */
    strcpy(working, string);

    //transfer all the capital letters in string "working" into its
lowercase
    int lenwork = strlen(working);
    for(int i = 0 ; i < lenwork; i++)
    {
        if(working[i] >= 'A' && working[i] <= 'Z') working[i] += 32;
    }

    for(idx = 0; idx < MAXTOKENS; idx++)  tokens[idx] = NULL;
    //split string "working" into single words, string "delim" is the
punctuations omitted
    token = strtok(working, delim);
    idx = 0;

 /* always keep the last entry NULL terminated */
    while((idx < (MAXTOKENS - 1)) && (token != NULL))
    {
        tokens[idx] = (char*)malloc(sizeof(char) * strlen(token) + 1);
        if(tokens[idx] != NULL)
        {
            strcpy(tokens[idx], token);
            idx++;
            token = strtok(NULL, delim);
        }
    }

    free(working);
    return tokens;
}

/* install word in binary tree */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if(p == NULL)
    {
        p = talloc();
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    }
    else if((cond = strcmp(w, p->word)) == 0)  p->count++;
    else if(cond < 0)                          p->left = addtree(p-
>left, w);
    else                                       p->right = addtree(p-
>right, w);

    return p;
```

```
}

/* make new tnode */
struct tnode *talloc(void)
{
    return(struct tnode *)malloc(sizeof(struct tnode));
}

/* find value w in binary tree */
struct tnode *findstopword(struct tnode *p, char *w)
{
    struct tnode *temp;
    int cond = 0;

    temp = p;

    while(temp != NULL)
    {
        if((cond = strcmp(temp->word, w)) == 0)  return temp;
        else if(cond > 0)                        temp = temp->left;
        else                                     temp = temp->right;
    }

    return NULL;
}

/* free binary tree */
void freetree(struct tnode *p)
{
    if(p != NULL)
    {
        free(p->left);
        free(p->right);

        free(p->word);
        free(p);
    }
}

/* return the result of buidling filelist 0 - failed, 1 -successful*/
int BuildFileList()
{
    FILE*fp;
    //the relative path form code to source files
    char path[MAX_LENPATH] = "..\\documents\\source";
    char *dir = path;
    printf("| [SYSTEM] Begin building filelist...
                        |\n");

    Search(dir);

    //write menu in fileliset.txt
    fp = fopen("..\\documents\\testfile&output\\filelist.txt", "w");
    if (fp == NULL)
    {
```

```
        return 0;
    }
    while (n_file > -1)
    {
        fprintf(fp, "..\\%s\n", stack[n_file].dir+3);
        n_file--;
    }

    fclose(fp);

    return 1;
}

/* return the result of buidling resultlist 0 - failed, 1 -
successful*/
int BuildResultList()

{
    FILE*fp;
    //the relative path form code to result files
    char path[MAX_LENPATH] = "result";
    char *dir = path;

    printf("| [SYSTEM] Begin building resultlist...
                        |\n");
    Search(dir);

    //write menu in resultliset.txt
    fp = fopen("..\\documents\\testfile&output\\resultlist.txt", "w");
    if (fp == NULL)
    {
        return 0;
    }
    while (n_file > 0)
    {
        fprintf(fp, "..\\documents\\%s\n", stack[n_file].dir);
        n_file--;
    }
    fprintf(fp, "..\\documents\\%s", stack[n_file].dir);

    fclose(fp);

    return 1;
}

/* scan the folder pointed by dir recurssively*/
int Search(const char *dir)
{
    long handle;
    struct _finddata_t findData;
    char dirNew[MAX_LENPATH];

    strcpy(dirNew, dir);
    strcat(dirNew, "\\*.*");
```

```
    if ((handle = _findfirst(dirNew, &findData)) == -1L)
    {
        printf("| #ERROR# Failed to find 'findfirst' file!
                            |\n");
        printf("+-------------------------------------------------------
---------------------------+\n");
    }

    while (!_findnext(handle, &findData))
    {
        if (findData.attrib & _A_SUBDIR)
        {
            //exclude .. & .
            if (strcmp(findData.name, ".") == 0 ||
strcmp(findData.name, "..") == 0) continue;

            strcpy(dirNew, dir);
            strcat(dirNew, "\\");
            strcat(dirNew, findData.name);
            // scan recurssively
            Search(dirNew);
        }
        else
        {
            //push current path into the stack
            if (++n_file < MAX_DATA)
            {
                strcpy(stack[n_file].dir , dir);
                strcat(stack[n_file].dir, "\\");
                strcat(stack[n_file].dir, findData.name);
            }
        }
    }
    _findclose(handle);
}

void pretreatment()
{
    printf("+-------------------------------------------------------
-----------------------+\n");
    printf("|                              [SYSTEM] Doing pretreatment ...
                        |\n");
    printf("|                                      Please wating...
                       |\n");
    printf("+-------------------------------------------------------
-----------------------+\n");

    //delete those char when sccaning
    char *delim = ".,:;`\"+-_(){}[]<>*&^%$#@!?~/|\\=
\t\r\n1234567890";
    char **tokens = NULL;
    struct tnode *querry = {0};
    char line[MAXLINE];
    int i = 0;
```

```
    //build filelist
    int build = BuildFileList();
    if(!build)
    {
        printf("| #ERROR# Failed to build filelist!
                            |\n");
        printf("+-----------------------------------------------------
---------------------------+\n");
    }
    else printf("| [SYSTEM] Build filelist successfully!
                            |\n");
         printf("+-----------------------------------------------------
-----------------------------+\n");

    //build the BST of stop words

    if(root == NULL)
    {
        printf("| #ERROR# Empty stop words tree!
                           |\n");
        printf("+-----------------------------------------------------
---------------------------+\n");
    }


    // handle files in FOLDER 'source' one by one, output the handled
files in FOLDER 'result'
    //change the current work path to FOLDER project
    char path[MAX_LENPATH] = "\0";
    getcwd(path, MAX_LENPATH);
    int lenpath = strlen(path);
    while(path[lenpath-1] != '\\') lenpath--;
    lenpath--;
    path[lenpath] = '\0';
    strcat(path, "\\documents");
    chdir(path);

    //create FOLDER 'result'
    int status = mkdir("result");
    if(status)
    {
        printf("| #WARNNING# FOLDER 'result' already exists! Delete
before loading new files.     |\n");
        printf("+-----------------------------------------------------
---------------------------+\n");
    }
    else
    {
        printf("| [SYSTEM] Create FOLDER 'result' successfully!
                            |\n");
        printf("+-----------------------------------------------------
-----------------------------+\n");
    }

    // handle those files in the order printed in filelist.txt
```

```
    FILE *list = fopen("..\\documents//testfile&output//filelist.txt",
"r");
    if(!list)
    {
        printf("| #ERROR# Failed to open filelist.txt!
                            |\n");
        printf("+-------------------------------------------------------
---------------------------+\n");
    }
    fgets(path, MAXLINE, list);
    while(!feof(list))
    {
        // delete '\n' at the end of the line
        lenpath = strlen(path);
        path[lenpath-1] = '\0';
        lenpath--;

        //open source file
        FILE * fp1 = fopen(path, "r");
        if(!fp1)
        {
            printf("| #ERROR# Failed to open source file!
                                |\n");
            printf("+-------------------------------------------------
--------------------------------+\n");
        }

        //create the result file which has the same name as the source
file
        //resul file name - res_name
        i = lenpath-1;
        while(path[i] != '\\') i--;
        char res_name[80] = "\0";
        char res_addr[80] = "\0";
        strcpy(res_name, path+i+1);
        strcpy(res_addr,"result//");
        strcat(res_addr, res_name);

        //write result file
        FILE * fp2 = fopen(res_addr, "w");
        if(!fp2)
        {
            printf("| #ERROR# Failed to build resultlist!
                                |\n");
            printf("+-------------------------------------------------
--------------------------------+\n");
        }



        //handle fp1 (when the filelist is not at the end)
        while(!feof(fp1))
        {
            fgets(line, MAXLINE, fp1);
            if(strlen(line) < MINLEN) continue;
            //spilt line into single tokens(words)
```

```
            tokens = split(line, delim);
            for(i = 0; tokens[i] != NULL; i++)
            {
                n_total_word_st++;
                querry = findstopword(root, tokens[i]);

                //if the token is not a stop word, then do word
stemming, print it in result file
                if(querry == NULL)
                {
                    strcpy(b,tokens[i]);
                    int t = 0, n = strlen(b);
                    stem(0,n-1);
                    for(t=0;t<=k;t++)
                        fprintf(fp2,"%c",b[t]);

                    fprintf(fp2,"%c",32);
                }
            }

            for(i = 0; tokens[i] != NULL; i++) free(tokens[i]);

            fprintf(fp2, "\n");
            fgets(line, MAXLINE, fp1);
        }
        fclose(fp1);
        fclose(fp2);
        fgets(path, MAXLINE, list);
    }
    printf("| [SYSTEM] Stop words has been filtered!
                        |\n");
    printf("| [SYSTEM] Word stemming is done!
                         |\n");
    printf("+-----------------------------------------------------------
------------------------+\n");

    //freetree(root);

    //build resultlist
    build = BuildResultList();
    if(!build)
    {
        printf("| #ERROR# Failed to build resultlist!
                              |\n");
        printf("+-----------------------------------------------------------
----------------------------+\n");
    }
    else
    {
        printf("| [SYSTEM] Build resultlist successfully!
                              |\n");
    }

    getcwd(path, MAX_LENPATH);
    strcat(path, "//code");
```

```
    chdir(path);
}
```

References

[1] Kingston Chan, "stemmer.c"https://github.com/kingston-chan/top_words/blob/main/stemmer.c

[2] Happy Codings, "Filter Text with a Stop Word List" https://c.happycodings.com/small-programs/code16.html

[3] Sebleier, "NLTK's list of english stopwords" https://gist.github.com/sebleier/554280

[4] William Shakespeare, “The Complete Works of William Shakespeare” http://shakespeare.mit.edu/

Author List

Name	Task
王柯棣	Build Inverted File Index & Implement of Searching Algorithm corresponding section of Chapter 2,3,4 in Report Download most works of William Shakespeare
赵伊蕾	Word Stemming Part & part of Output Format / Pretreatment corresponding section of Chapter 2,3,4 & whole Chapter 1 in Report, Find the Stop Words List
沈韵泓	Stop Word Filtering Part & Path List Generation & part of Output Format / Pretreatment corresponding section of Chapter 2,3,4 in Report, Merge three parts

Declaration

We hereby declare that all the work done in this project titled "pj1" is of our independent effort as a group.

Signatures

Each author must sign his/her name here: