SEARCH ENGINE REPORT
Group 1

Author and Contributions

- Đào Quang Dũng
- Võ Bùi Phú Hưng
- Phan Minh Duy
- Hoàng Hiệp

Table of contents:

INTRODUCTION

1. Definition of Search Engine

Search engines is a tool using in many websites for searching result on the database and it based on the keyword (query) that the user input from keyboard.

2. How do Search Engine works?

Search Engine finds the searching-word from database, use some special algorithm to find the final result. These algorithms can help us rank the priority of each websites or documents and give us the most suitable searching result. There are many searching – browser in the world, we can list some of the most popular such as: Chrome, Firefox, Safari, Edge, Bing... Chrome, Safari or Edge usually come with a default search engine set as a home page or starting page.

In this project, we will design Search Engine that we can search through many documents. Having a set of text file and queries, search engine operator can detect all documents involve the searching – word in the query. This problem can be solved by using data structures and algorithm that we learned at class.

3. Requirements of a search engine

As I concerned that there are 3 major reasons making these popular search – engine (Chrome, Safari, Firefox) become a good search engine:

- First, these programs can handle with extremely large amount of data.
- Second, these programs work very smooth and efficiency. Not only having fast run-time but also return the most relevant results for everyone.
- Third, these programs can rank information better than any another. In some cases, the words that we want to search can be found in many documents or many websites.

Depending on the popular and some other factors, they can give the closets results to the user. Many contributors can commit to different rankings in this engine. The most important things of these searching tool is giving the results to user from the best to worst.

There are also some non – functional requirements:

These programs must be simply understandable, adoptive, portable, enhanceable and reusable with good performance. It must be friendly, responsive and beautiful. Also the system can be developed in the near future.

DATA STRUCTURES AND ALGORITHM

1. Data Structures

In this project, we use vector <string> fileData[i] to store all characters in an document after normalizing these documents one by one. Pair structure and unordered_map in STL is using to process some operations in the least complexity time.

Trie was chosen due to its functionality as a dictionary. Using trie can help us to search, insert and delete some characters in logarithmic time. Firstly, every node will include a list of pairs, the first one contains characters or numerical and the second will contain pointer to the next nodes. With inserting operation, we will insert character one by one and there is the boolean to detect the "end of the word''. For deletion, we delete the child first as what we do in the binary search tree. If we get to the null node or final node, we can receive "end of world" signal and we can conclude that this word is not existed. On the other hands, the searching – work is completed and we can add all documents relative with the key to the result.

2. Algorithm

a. Indexing:

We store all documents from the database in the vector structure. To do this storing work, we will check the content line by line, word by word. If the word appears in case "stop words" we will skip it, otherwise we can store these word in the vector fileData[i] and inserting word into trie. Every insertion will take O(n) in time complexity where n is the number of characters. At the end of each word, there is a variable type unordered_map <int, int> will store the id file and counting time of this word in this file.

b. Searching:

In searching work, we will manipulate by vector structure in Standard library template. This follow the aforementioned approach, we will put all data from database into each vector fileData[i] and each of them will involve content of the file after normalizing. We can search for a key by performing hash table structure or trie. When we go down from the root to the leaf on trie, we can check whether there is "end of world" and show the list of documents containing it. If it hits the null node midway when we try to search or that is not "end of word", we can return null pointer. In some cases, we can use find operator in string STL to find a string in vector.

c. Deletion:

This function will delete all characters all words stored in this trie. However, when we build a trie as dictionary, the width of this trie can be very large, deletion can take approximately 1/15 indexing time. When we perform deleting process, we will go as the same as ordinary trees, where we will delete all children before deleting parent nodes.

d. Loading and Saving:

When we load the data from database, we will store the content of all documents into vector structure and also the content is normalizing into the normal case of a document, eliminating all colons (expect $ and #). With some queries, we can load the data from vector fileData into the trie and process easier than we both read and constructing operator from file.

RUNNING TIME

1. Load files

We split our data into two packages, the smaller and larger ones:

a. For the smaller:

It contains 171 files, consumes 800 KB storage, each file has from 100 to 3000 words, with 500 words on average. By importing this data 3 times, it tooks:

- The 1st : 201 ms
- The 2nd : 199 ms
- The 3rd : 200 ms.

So it is about 200 ms ~ 0.2 seconds on average.

b. For the larger:

It contains 11 271 files, consumes 65.6 MB, each file has from 100 to 4000 words, with 800 words on average.

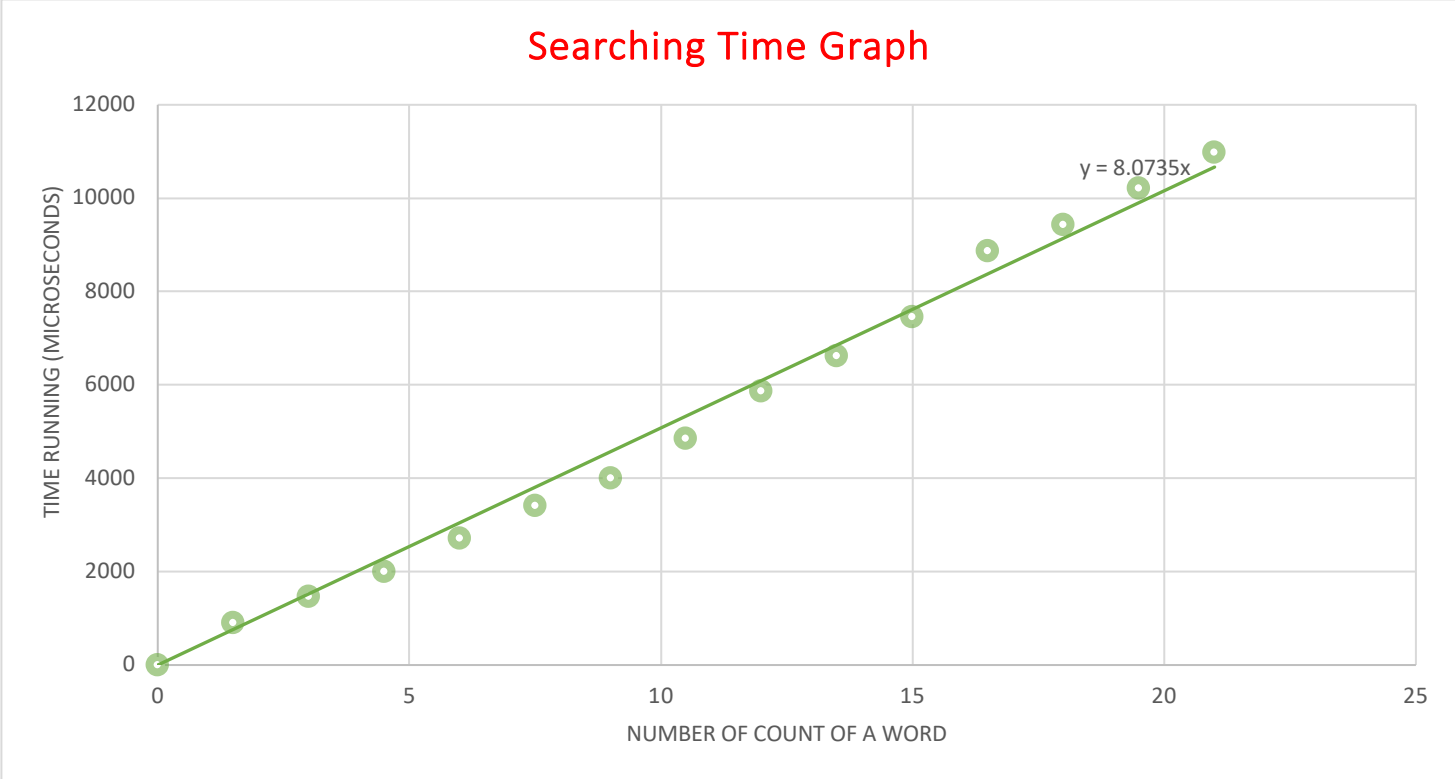
We have a time report relating to time loading of database in this project:

Loading Progress	Loaded files	Current time
25%	2 818	6.063s
50%	5 636	11.515s
75%	8 454	17.278s
100%	11 271	21.967s

2. Searching (Including search key word, rank and display result):

Here are some sample queries generated randomly, cover all required operators:

Word	Total count	Search time (Microseconds)
School	1084	2035
~University	3936	9975
New year	7500	16993
New AND year	1900	20910
Top * highest	4	6983
New -year	4278	14002
Opportunity	656	996
Job	812	1994
Job AND opportunity	76	2052
“come home”	16	17001
Vietnam	244	997
“Ho Chi Minh”	72	1000
1..10000000	9836	132154
$100	136	1
Samsung $1..$100000	1790	14006
“Michael Jackson”	4	1997
America AND intitle:fbi	4	2996
“Computer science”	40	5022
#MeToo	68	1
Working filetype:txt	1616	3991



3. **Comment:**
   - Search time depends mainly on the numbers of time that word occurs on the database.
   - Search time also depends on complexity of the query operation, as example, EXACT query often consumes more time than the normal one.
   - If the database is very large and the keywords occur 1.000.000 times, the average search time is 8.0735s by using linear trendline forecast. Also if the keywords occur 1 billion times, the search time will be 2 hours and 15 minutes and it can take approximately 4 hours in worst case (not practical in our daily problems).

OPTIMIZATION

1. **Indexing time and space**
   a. ***Affecting factors:***
      - The trie may get unnecessarily deep.
      - Many unused slots of character.
      - Files can contain many redundant words or many stop words.
      - Inserted words may not need to be traversed top-down again to add more documents to it.

   b. **Solutions:**
      - We can use hash table and vector to deal with memorable problem instead of using an array holding empty slots to store characters or numbers. Firstly, by using hash tables and vectors we can avoid creating empty slots of memories and hence less empty branches.

Secondly hash tables can be used in conjunction with other types of characters such as apostrophes or colons.

- We can see that filter time always costs O(n) time, which is unavoidable since we have to normalize the string as well as omitting weird symbols before insertion. However, we filter while inserting to avoid multiple O(n) operations.
- All data after normalizing are storing in the vector type can help us a lot when finding some elements or using algorithms.
- With a file that contains many already-inserted words or stop words, the best approach is to create another file containing "popular" words, which are present in about 25% of the files. We may also create a private list and documents for these "popular" words, so that once we detect them insertion shall be instant. Same for stop words detection. We can also use find in string STL which can minimize finding substring works in a large string.

2. **Searching time:**
   a. ***Affection factors:***
      - Too many unnecessary opening and reopening files for operators and this problem can lead to crash our program.
      - Taking a great deal of time to rank results (based on appearance of keywords and the frequency of it).
      - Taking time to extract related contents from each file.
      - Popular keywords may take a huge amount of time for both operators and ranking.
      - The program has to process the keyword string multiple times in order to get the right markers for each operator.

   b. **Solutions:**
      - We use many structure and own technique in this program to work in each file instead of working through each operator, meaning that for each file we could check the list of operators the keyword contains and see if each file fulfills their requirements. And also we use the algorithm and operator in Standard template library to minimize the code and we can look for it better.
      - The ranking shall be the last thing to do after all operators, the expected amount of files will be greatly reduced compared to the first search. At this time, we shall remove duplicates and search each file again for keywords and count their frequency, while also extracting contents for printing results.
      - The solution to "popular" keywords have already been given out in the indexing process, where we declare a private structure such as a set for "popular" words, where in their locations are stored and thus the lookup process takes minimum time.

PROBLEMS AND HOW TO SOLVE IT

1. **First approaching:**
   a. ***Firsty, we will consider 2 problems that first come to our mind when approaching the problem (index and search):***

- We notice that importing characters are not always saved right in the right file such as: ABC:XYZ and it can be saved as ABCXYZ. Memory in this project is also a big problem, when we have to save too many files (up to 10 thousand files or more) and some of them are no use at all (example as empty file) and this reason can increase running – time of our program. If we don't try to find the best way to store the file, we must have enormous space to store all data taken from importing files.
- Searching can lead to some hard problems as keywords need to read correctly and efficiency and also we can access to right files. We have to consider operators while looking after the searching to help the program run with ideal performance. We must optimize the program as possible because no one wants to see us searching or doing any operators for a string in an hour. To deal with this problem, we use the Trie structure, hash table and some functions and algorithm in Standard Template Library and we can minimize most operator into logarithmic time for each file, working smoothly with tons of files. In some cases, we also search only 5-6 characters in a word and this idea can help us save time.

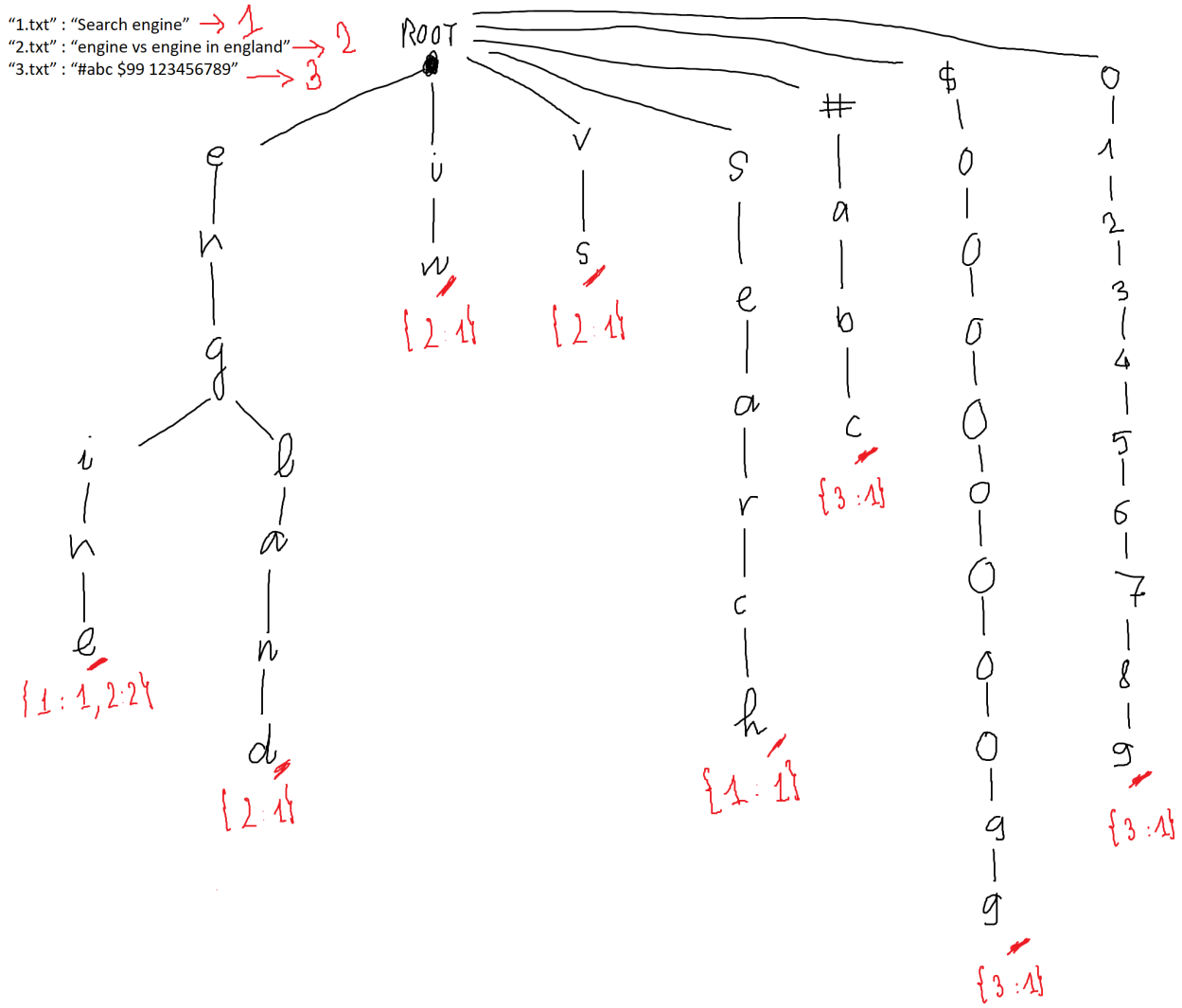
b. How to solve it?

We have a short outline to discuss what we do on this project:

All text paragraphs will be eliminated all punctuations, turning all characters into lower cases and we will remove all stop word in each file data. 2 characters '#' and '$' will be kept in the importing data.

These variables will be used in this project:

- fileName[i]: fileName[i] is the name of file with index i in the database. Example: fileData[1] = '1.txt'.
- vector fileData[i] will contain all data in a data file with index i after normalization
- All text paragraphs will be inserted into Trie, word by word. At the final node of a word, there is a variable unordered_map <int, int> storing id file and the times of existing of a word in this file. When consider about digits, we will set "0" at the beginning, it is satisfied that all numbers will contains 10 characters and this work will make the range query easier. Example: with 3 text file ("1.txt" : "Search engine", "2.txt" : "engine vs engine in england", "3.txt" : "#abc $99 123456789"), trie model will look like this:
- When we check some keys in text 2 that it existed in the text 1, we will do with the hash table method
- If this text is a normal text: For instance, the key "Search Engine" will be processed as "Search" OR "Engine". The priority of 2 separating word is equal to the number of existing of these words in the paragraph.
- All operators will be handled from left to right.



About operators:

- AND: This query will be processed: <result founding on the left> and <result founding on the right>, it is meaning that left characters and right characters will be looked up in this text. Solving: After finding all files are suitable with left characters and other file for right characters, we will union 2 set containing files and finding subset. The priority is same as the total number of existing of left characters and right characters.
- OR: Similar with normal search
- "-'' operator: This query is finding all texts not consisting "-''. Doing this operator, we will find all files consisting rest words and eliminating all file that it existed "-''.
- intitle:hammer nails: With this operator, after finding all texts included behind – words, we will choose all texts having name as "hammer". We should notice that we only process for string "hammer" on the title, not handling full string "hammer nails" in the title.
- "+" operator: We will do the same things with AND operator.
- filetype:txt operator: Same with AND query.

- Search for a price: We put "$" in front of a number. For example: camera $400, searching in a naive way.
- Search for a hashtag: We put "$" in front of a number. For example: camera $400, searching in a normal way.
- Search for an exact match. Put a word or phrase inside " ". For instance, "tallest building". We do filter – work for all text contains “tallest” AND “building”. After this work, we will use for loop to count the numbers of time that “tallest building” appeared in this text.
- Search for wildcards or unknown words. Put a "*" in your word or phrase where you want to leave a placeholder. For example, "largest * in the world". We will filter “largest” AND “in the world”. We should keep our eyes in this operation: taking words in this operation is different with naive searching. We will go from “*” and widen to 2 sides until we found another operation then we will use exact operation. We will find when “largest”, “in the world” existing in the text respectively.
- Search within a range of numbers. Put ".." between two numbers. For example, camera $50..$100. We will go down from trie and all moving steps must in the range that we search.
- Entering “~set” will bring back results that include words like “configure”, “collection” and “change” which are all synonyms of “set”: Finding a dictionary and we will do set OR configure OR collection OR…

About main function:

- unordered_map <int, int> merge (unordered_map <int, int> &A, unordered_map <int, int> &B): importing from A and B and finding union of A and B, same with OR operation.
- unordered_map <int, int> intersect (unordered_map <int, int> &A, unordered_map <int, int> &B): Importing from A and B, finding the intersection of A and B, same with AND operation.
- void exact (unordered_map <int, int> &A, string &key): Importing from A and show us all text consisting exact key.
- unordered_map <int, int> fileNameQuery (unordered_map <int, int> &A, string &fileNameRequire): At filenameRequire, we will process for filetype and intitle.
- void wildcard (unordered_map <int, int> &A, vector <string> &orderedKey): Receiving from A and finding all string all order properties of orderedKey.
- unordered_map <int, int> search(Trie* root, string &key): Showing all paragraphs contain key.
- unordered_map <int, int> range(string &left, string &right): Finding all paragraphs include from left to right in the database.

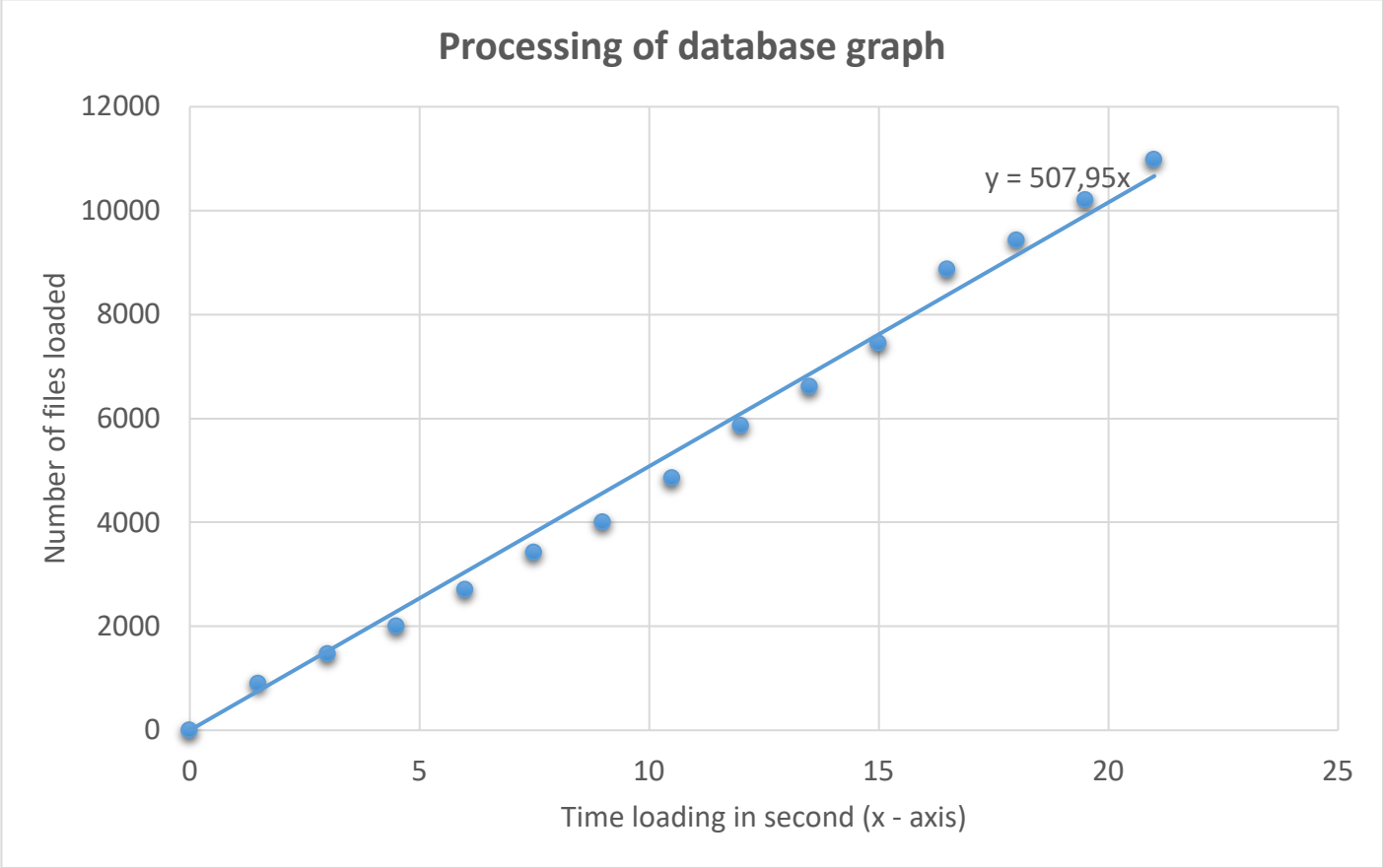
About files:

- trie.h: Containing all class relating trie structure.
- operator.h: Including all operators that we use in this project
- normalize.h: it is a standardize – file to process the all raw text into normal text.
- read.h: Input all data from database.
- screen.h: Table of choosing input and output.

SCALABILITY

We have an equation that predict the loading time, it can be followed as this table:

Time (seconds)	Number of Files (P(t))
1.5	895
3	1460
4.5	2001
6	2702
7.5	3412
9	4001
10.5	4850
12	5856
13.5	6612
15	7450
16.5	8870
18	9420
19.5	10203
21	10972



Based on linear trendline model, we have an equation that it can represent for the loading file:
$P(t) = 507.95t$ (when t represents for time in second)

From the function $P(t)$ we can find the root of the equation $P(t) = k$ (when k is the number of loading files)

$$t = \frac{k}{507.95}$$

It is easy to see from the graph that the running time is almost linear increasing.

- **Comment:**

With this predictive equation, we can calculate how much time taken when we run large number of text files:

- 1.000.000 files will be taken time about 1968s (approximately 33 minutes)
- 10.000.000 files will be taken time about 19680s (approximately 5.5 hours)
- 100.000.000 files will be taken time about 196800s (approximately 55 hours or 2 days and 7 hours)
- 1.000.000.000 files will be taken time about 22 days and 22 hours

We can see clearly that our program is not fully optimized when we deal with very extremely large database.

- **Other problems of scalability:**

1. ***Types of documents:***
   - Some popular searching browser such as Google, Bing, Firefox can optimize what the use looking for by using specialized method or algorithm. We can search to find many data types with only a string of key word. We can list some of these data types such as: Image, Audio, Maps, Magazine, product…. Especially, when we browse to buy something in a range (Camera $50..$100), these browser will recommend to us many option with image and always sort these products from high priority to less priority. Not only searching with the typing data but also with voice data.
   - Back to our search engine, we can only search with database as text file. Our program can't deal with image or audio searching. If we have more time to develop this program, we will find the solution for searching other type of data.

2. ***Priority Results:***
   Google uses a trademarked algorithm called PageRank to rank results and it depends on these factors:
      - The frequency and location of keywords: If the keyword appears only once within the body of a page, it will receive a low score for that keyword
      - How long the document has existed: People create new things on the Internet every day, and not all of them stick around for long. Google places more value on pages with an established history
      - The number of others that link to the page in question: Google looks at how many Web pages link to a particular site to determine its relevance.

   However, our program almost ranked the priority of results based on the frequency of key words in each documents. An innovation for our search engine is considering more factors and return the best results for the user than only browsing by frequency.

QUERIES

#metoo

```
 Searching ... #metoo
 Found 68 results in 0 microseconds.

File 4291: Data2852.txt
| #metoo: 4 |
... accused sexual harassment looks like vietnam #metoo moment three young women spoken vietnamese ...
-----------------------------------------------------

File 7334: Data7832.txt
| #metoo: 4 |
... accused sexual harassment looks like vietnam #metoo moment three young women spoken vietnamese ...
-----------------------------------------------------

File 2240: Data12093.txt
| #metoo: 4 |
... accused sexual harassment looks like vietnam #metoo moment three young women spoken vietnamese ...
-----------------------------------------------------

File 10574: Group18News63.txt
| #metoo: 4 |
... accused sexual harassment looks like vietnam #metoo moment three young women spoken vietnamese ...
-----------------------------------------------------

File 8541: Data943.txt
| #metoo: 3 |
... journalism politics accused sexual sacked stepped #metoo hashtag already used times twitter women ...
-----------------------------------------------------
```

~big

```
 Searching ... ~big
 Found 7728 results in 27927 microseconds.

File 12: 010.txt
| big: 4 | bad: 1 | grown: 3 | expected: 2 | large: 3 | heavy: 1 | expect: 1 | expects: 2 | great: 1 | largely: 1 | giving: 2 | liberal: 1 |
... meat year half world pork eaten liberal agricultural policies allowed farms produce 1961 ...
-----------------------------------------------------

File 7196: Data690.txt
| big: 4 | bad: 1 | grown: 3 | expected: 2 | large: 3 | heavy: 1 | expect: 1 | expects: 2 | great: 1 | largely: 1 | giving: 2 | liberal: 1 |
... meat year half world pork eaten liberal agricultural policies allowed farms produce 1961 ...
-----------------------------------------------------

File 5121: Data36.txt
| big: 4 | bad: 1 | grown: 3 | expected: 2 | large: 3 | heavy: 1 | expect: 1 | expects: 2 | great: 1 | largely: 1 | giving: 2 | liberal: 1 |
... meat year half world pork eaten liberal agricultural policies allowed farms produce 1961 ...
-----------------------------------------------------

File 5155: Data363.txt
| big: 4 | bad: 1 | grown: 3 | expected: 2 | large: 3 | heavy: 1 | expect: 1 | expects: 2 | great: 1 | largely: 1 | giving: 2 | liberal: 1 |
... meat year half world pork eaten liberal agricultural policies allowed farms produce 1961 ...
-----------------------------------------------------

File 7027: Data538.txt
| grown: 4 | liberal: 9 | largely: 1 | liberalism: 1 | big: 3 | liberals: 1 | expected: 1 | large: 1 |
... courts chart autocracy popular among less large majorities favouring leaders cannot openly admit ...
-----------------------------------------------------
```

1..10

```
 Searching ... 1..10
 Found 4604 results in 25928 microseconds.

File 4153: Data2728.txt
| 5: 16 | 10: 16 | 4: 3 |
... starts july also cut-off date season 4 means bought season battle pass already ...
--------------------------------------------------

File 6920: Data5218.txt
| 5: 16 | 10: 16 | 4: 3 |
... starts july also cut-off date season 4 means bought season battle pass already ...
--------------------------------------------------

File 423: Data10198.txt
| 5: 16 | 10: 16 | 4: 3 |
... starts july also cut-off date season 4 means bought season battle pass already ...
--------------------------------------------------

File 10449: Group17News38.txt
| 5: 16 | 10: 16 | 4: 3 |
... starts july also cut-off date season 4 means bought season battle pass already ...
--------------------------------------------------

File 2513: Data1251.txt
| 8: 3 | 2: 3 | 10: 2 | 1: 22 | 3: 1 | 7: 1 |
... tuesday comet episode 12 season episode 7 yuruyuri san episode 11 wednesday aria ...
--------------------------------------------------
```

New years

```
 Searching ... "new year"
 Found 8 results in 39953 microseconds.

File 10349: Group16News38.txt
| new year: 1 |
... 5 monday night underwent surgery wednesday new year tested fabric myers hard value ...
--------------------------------------------------

File 5983: Data4375.txt
| new year: 1 |
... workers areas financial losses may pick new year housing market continues weaken credit ...
--------------------------------------------------

File 6808: Data5117.txt
| new year: 1 |
... 5 monday night underwent surgery wednesday new year tested fabric myers hard value ...
--------------------------------------------------

File 4041: Data2627.txt
| new year: 1 |
... 5 monday night underwent surgery wednesday new year tested fabric myers hard value ...
--------------------------------------------------

File 3216: Data1885.txt
| new year: 1 |
... workers areas financial losses may pick new year housing market continues weaken credit ...
--------------------------------------------------
```

school

```
Searching ... school
Found 1084 results in 2034 microseconds.

File 8351: Data8759.txt
| school: 26 |
... teens sleep alert homework school starts preliminary findings new study middle ...
--------------------------------------------------

File 5320: Data3779.txt
| school: 26 |
... teens sleep alert homework school starts preliminary findings new study middle ...
--------------------------------------------------

File 2554: Data1289.txt
| school: 26 |
... teens sleep alert homework school starts preliminary findings new study middle ...
--------------------------------------------------

File 9011: Group03News19.txt
| school: 26 |
... teens sleep alert homework school starts preliminary findings new study middle ...
--------------------------------------------------

File 1914: Data1180.txt
| school: 19 |
... saigon high school students put slog graduation exams loom ...
--------------------------------------------------
```