Group 11

Data structure and Algorithms for Mini Search Engine

Members : Tô Tuấn An, Từ Khai Hoài, Bùi Duy Bảo, Lê Quý Khôi, Nguyễn Minh Uyên

+Data structure: We choose a <string,Trie> unordered_map to store every file so that we have a map containing 1001 slots for 1001 files. The first index string is the filename and the second is a Trie structure which stores all the words that appear in the file. We choose Trie for searching to implement this project because it is an efficient information retrieval data structure, as its complexity is O (key length).

+Algorithms: every TrieNode will contain these things:

-an array of children (0 to 9 is numbers, 10 to 35 is letter, 36 is @, 37 is _, 38 is $, 39 is %, 40 is #, 41 is -, a bool variable isLeaf(to mark a word), vector<int> order (to store the position of the word in file for wildcard and exact match searching operator).

Inserting data:

First, we initialize an unordered_map<string, Trie> data = null;

To read 1001 files from top (based on the __index.txt file), we use an InputListFile function to read file by file. In every single file, we implement a InputFile function in order to open each file and read the file line by line with the SenFilter function to split sentences into a vector of words.

```
vector<string> SenFilter(string sen){
        vector<string> v;
        string word;
        for(int i = 0; i < sen.length(); i++){
                if (sen[i] == ' '){ //if there is a space in between two words
                        vector v push back every word;
                }else{
                        store the word in a string;
                }
        }
}
```

```
void InputListFile (string filename, unordered_map<string, Trie> &data){
        open( filename); // filename is __index.txt
        string line; int n = 0;
        while(filename is opened){
                while(n < 1001){
                        read every "line" in file __index.txt  to to the filename;
                        InputFile (line, data);
```

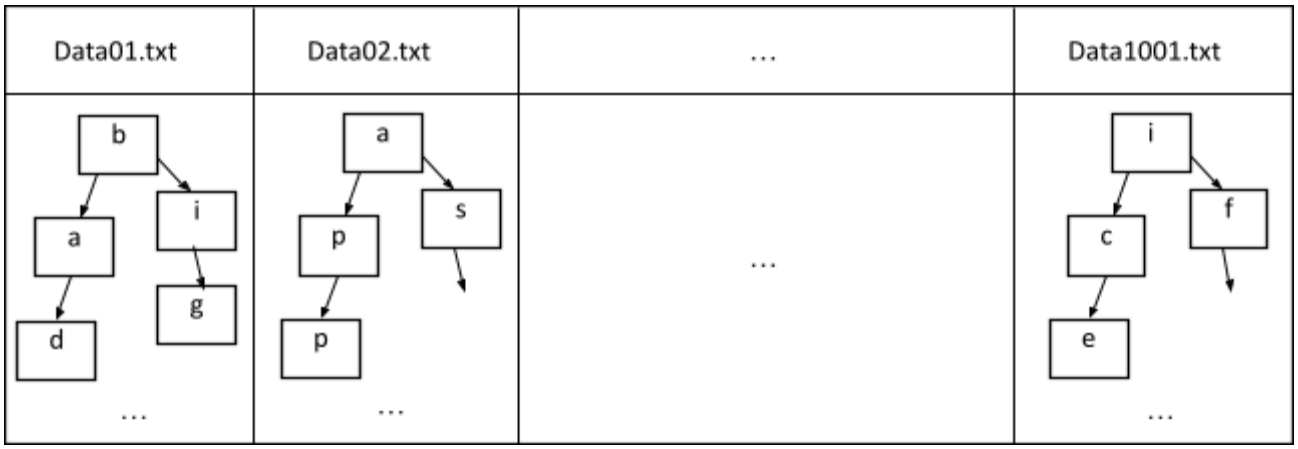
```
                        ++n;
                }
        }
        close(filename);
}
```

```
void InputFile(string filename, unordered_map <string,Trie> &data){
        open(dataset\\filename);
        int place = 1; string sen;
        if(filename is opened){
                while(not end of file){
                        read every line in the file;
                        vector<string> v = SenFilter(sen);
                        if(!data[filename].root) data[filename].root = getNode(); //if there is no
such filename index in the map, we create one;
                                for(int i = 0; I < v.size(); ++i){
                                   insert(data[filename].root, v[i], place); ++place;
                                        //we build a Trie inside the map with the index of filename
by inserting word into it with function implemented in Trie.cpp, where v[i] is the word in the file
and place is the appeared order of the word;
                                }
                   }
        }
}
```

The pseudocode of the insert function will be:

```
void insert(TrieNode* root, string key, int place){
        TrieNode* cur = root;
        Int id;
        for(int i = 0; i < key.length(); ++i){
                id = get_index(key[i]) //get the index of the key;
                if(id == -1) continue;
                if(!cur->child) cur->child[id] = getNode() // if no such index create a new node
for it
                cur = cur->child;
        }
        cur->isLeaf = true; //make a signal for the end of the words
        cur->order.push_back(place) // store the order of the word appeared in file
}
```

After running the InputListFile("__index.txt", data); the unodered_map<string,Trie> data



becomes an array with the index is filename, and the value is the Trie contains all words from the file:For example: the data structure is illustrated as below:

Searching data:

Fundamentally, we build up a function to search a keyword, (before that, we need to check if the keyword is a stopword, by searching it in the pre-built stopword Trie) if it is, we will erase the keyword.

We also implement a function to hash the character into an index to fit the array in Trie (0 to 9 is numbers, 10 to 35 is letter, 36 is @, 37 is _, 38 is $, 39 is %, 40 is #, 41 is -)

To search a word in the Trie, we compare the character and move down. The search can be stopped due to the end of string or lack of key in Trie. If the boolean value isLeaf of the last node is true then the key exists in the Trie else the search will end since the key does not exist. The pseudocode will be:

```
TrieNode* searchWord(TrieNode* root, string key){
        TrieNode* cur = root;
        int id;
        for(int i = 0; i < key.length(); ++i){
                id = get_index(key[i]) //convert the key to its own index
                if(id == -1) continue;
                if(!cur->child[id]) return nullptr; //if no such key return null
                cur = cur ->child[id] //if there is such key pointer move down
        }
        //After searching the whole key length we check if the boolean isLeaf is true return the
pointer
        if(cur){
                If(cur->isLeaf) return cur;
        }
        return nullptr;
```

```
}
```

Analyse the query:

(Our search engine supports 12/12 types of query)

Normally, if no special operator is between two keywords, we consider it as an AND operator. Ex, “fish chip” = “ fish AND chip”

We implement a function called checkOperator() to check the operators in the query with the help of stringstream to read the word from left to right.

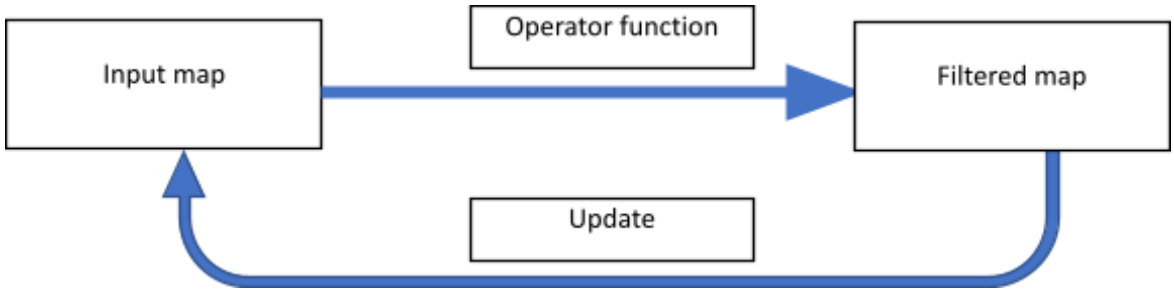
```
void checkOperator(string query, unordered_map<string,Trie> data,
unordered_map<string,Trie> &imap, unordered_map <string,Trie>  &omap, …){
        query = query + “END!”; //add an ending signal for the query
        stringstream ss(query)
        while(ss >> tmp){ //read the query word by word
                if(the word is AND){
                        skip it and read the next word;
                  }else if(the word is OR or END!){
                        do the orOperator(data, imap, omap) function;
                  }else if(the first eight letters is intitle:){
                              do the inTitle_search(imap, key) function;
                 }else if(the first letter is ‘-‘){
                        do the minus_Search(imap, key) function;
                   }else if(the first nine letters is “filetype:”){
                        do the filetypeOperator(imap, key) function;
                   }else if(the first letter if ‘ “ ‘){
                        get all of the word in between those “ “;
                        if there is an asterisk, we count it, and store the first position of the
asterisk in a start value;
                           do the wildCardOperator(start, ast, key, imap) function;
                  }else if(the is .. is the word){
                        do the rangeOperator(imap, tmp, line) function;
                   }else if(the first letter is ‘~’){
                        do the synonymsSearch(key, imap, tableKey, synonyms, line) function;
                  }else{
                        this is the case when we have the $, the #, the +, and normal keywords;
                         do the andOperator(key, imap) function;
                  }
        }
}
```

*AndOperator, inTitleSearch, minus_Search, filetypeOperator

Most of the functions like andOperator, orOperator, inTitle_Search, minus_Search, filetypeOperator they just need two arguments - the map and the keywords.

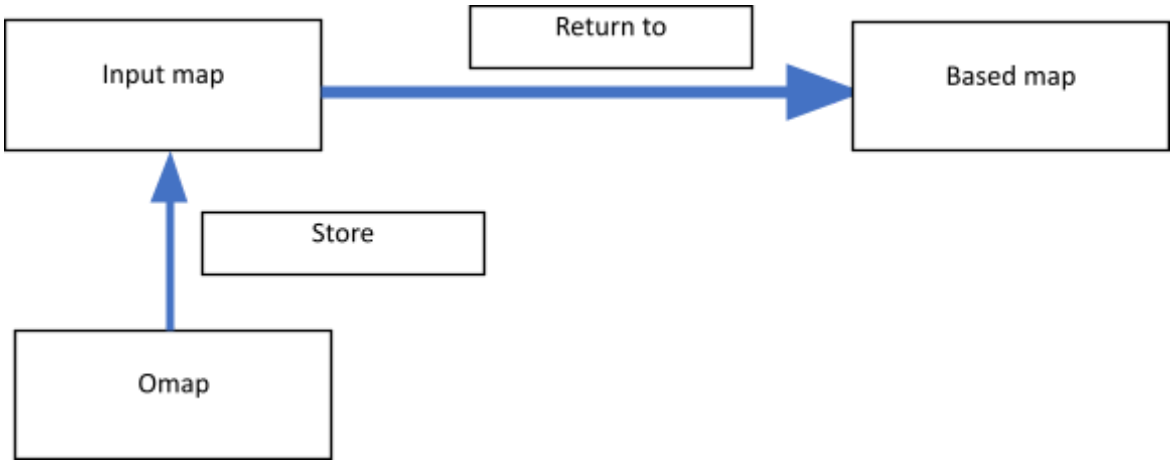
The logic behind this is that first, we input an unordered_map<> data (with full data of 1001) and the keyword into these operators:

If it is andOperator, we will search through the Trie in the map[filename] with the given keyword and after that filter out the index of filename the contains the keyword and update the input map with the filtered map, the minusOperator do the same but instead of filtering out the file that contains the keyword, we filter out the ones does not. The inTitle_Search and minus_Search work based on the same concept of andOperator but we do the search only in the filename (the index of the map) but the Trie.



*OrOperator

This type of operator has a union logic. For example: “shirt OR skirt”, the final map will end up containing files that have “shirt” and the files that have “skirt”. So when we read the word OR in the query, the omap appends all the data in the input map and then the input map returns to the first built map (contains all 1001 files). It waits until the end of the query that the input map will union all the files in the omap.



*WildCardOperator, ExactPhraseOperator

In order to search for exact phrases and wildcards, we need to make sure that the words are adjacent to each other or they are separated by a number of asterisks. To do so, we build up a function called searchWordpos() to return a list of positions of a certain word in a text file:

```
vector<int> searchWordpos(TrieNode* root, string key){
        TrieNode* cur = root;
        int id;
        for(int  i = 0; i < key.length(); i++){
                id = get_index(key[i]) //convert the character to its id
                if(!cur->child[id]) return vector<int>(); // if no such id return null
                cur = cur ->child[id]; // if yes move down
        }
        if(cur){
                if(cur -> isLeaf) return cur->order; // is it has such keyword the position vector
        }
        return vector<int>();
}
```

After that, we implement a wildCardOperator() function and apply the searchWordpos(). Before going into the details, let's take a query for example like "climate change".

The prerequisite point for a file to contain that phrase is that it contains both two words. Hence, the first step is to filter out the file that contains those two keywords from the input map. This helps to minimize the number of maps for searching.

The second step is to compare the positions of those words to meet the demand of the query. For example, "climate" appears twice in that file with the position array is {12,25} and "change" appear only once in tha file with the position array is {26}. Since the order of appearance is always increasing, the array is also in an increasing order. Our algorithm to compare is that we run through the array of the first appeared word (this case is "climate"), for every single element in the array we conduct a binary search of that element plus (N + 1) (N is the number of asterisk if "climate * change" N is 1) in the subsequent array of the next word (this case is "change"). We do that until there is no more subsequent array then we will accept that map[filename] or if we run through the whole array of the first word without any matching we will not accept that map[filename]. The total searching time complexity is (NlogN) (since the function is quite long, you can check the code in the HandlingQuery.cpp file).

*RangeOperator

The function is implemented to search for a range of numbers. In the example: $50..$100, searches are returned for the query from "$50" to "$100", but not for "$101" and beyond.

We need to use *atoi( )* to convert string to integer types for comparisons, at the end, we receive the starting value (e.g “$50”) and end value (e.g “$100”). Then we use a For loop to traverse all numbers between lower-bound and upper-bound above; before searching in dataset, we use *to_string( )* convert it to string type.

In detail, we pass by reference imap (which is an *unordered_map<string, Trie>* type) containing all our dataset. After that, initialize an temporary *unordered_map<string, Trie>* tmpmap, use it to hold all satisfied values after searching. Eventually, we assign imap = tmpmap, and delete tmpmap.

*SynonymSearch

For the synonym search, we store the synonym table in a CSV file, each line contains words that have the same meaning. Then we will read that file to the search program. In the input process, we use a technique called “Double hash table” to make the synonym search more efficient:

- We use the first hash table to store each word as a key, and its value will be used as a key in the second hash table. Synonym words will have the same value.

- Then, in the second table, for each key, it will have an array of synonym word as value. This hash table works like a chaining hash table.

In the synonym search, when we input a word, the first hash table will return a value and the second table will use that value and return the synonyms.

Here is a demonstration of the synonym search. We will convert the data of synonyms below into two hash tables:

1: danger, hazard, insecurity, jeopardy, risk

2: doubt, distrust, mistrust, surmise, suspect

Key	**Value**	**Key**	**Value**
danger	1	doubt	2
hazard	1	distrust	2
insecurity	1	mistrust	2
jeopardy	1	surmise	2
risk	1	suspect	2

Key	Value
1	danger, hazard, insecurity, jeopardy, risk
2	doubt, distrust, mistrust, surmise, suspect

Suppose we want to search synonyms for word ***danger.*** In the first table, we see that it has value 1, so we will use value 1 as a key to search in the second table and it will return ***danger, hazard, insecurity, jeopardy, risk***, which are the synonyms of ***danger***.

Other features:

Efficiency:

Our search engine can do multiple queries like:

Query: Manchester AND city OR blue -united “Raheem Sterling” …

Ranking system:

The ranking system of the files is based on their scores. The scores are collected randomly by the default setting of our team which means every files has its own score to be set.

History:

After searching any query, the queries are stored in a local History.txt file. Whenever the user wants to check the history for references, the system will use a STACK data structure to recommend the last five searching queries.

Output Result:

We implement two modes of printing results: get top 5 results (with preview mode) and get all results (with no preview mode).

First, we push all the data maps into a priority_queue to sort every map[filename] based on its score.

In the “get all results” mode, we solely print all the filename that satisfy the query respectively. However, in the “get top 5 results”" mode, we also want to preview the text file and highlight the keywords inside that file, so we decide to build up a OpenFile function for that purpose:

```
void OpenFile(string key, unordered_map<string, Trie> data, vector<string> line){
        vector<int> highlight; //implement a vector<int> to store position of words in that file
        for(int i = 0; i < line.size(); ++i){ //line is query
```

```
                vector<int> v = searchWordpos(data[key].root, line[i]); //return the position of
words in query to v
                for(int i = 0; i < v.size(); i++){
                  highlight. push_back(v[i]); // highlight store all the positions
                }
                sort(highlight) // sort
                while(file is opened){
                        read all the words until it meet the positions of keywords
                        (called it stop_point);
                        if(it is stop_point){
                                store next 100 words after the stop_point to preview
                        }
                }
                while(preview is opened){
                        read words by words in preview;
                        if(the word position is matched with the ones in highlight){
                                print it out in cyan color
                        }else{
                                print it out in black color
                        }
                }
        }
}
```

Conclusion:

Finishing our project, we successfully implemented a super cool mini search engine which is multifunctional, user-friendly and super efficient. However, there remain some small problems to take into account in the future.

Our projects can run all types of queries and search for keywords in a small amount of time, averagely less than one second for every query search. The data structure is easy to construct and update, meanwhile we need a little bit of time to wait for structure building every time running (in this case we run 1001 files which cost roughly 22 seconds).

Moreover, our wildcard operators only works for searching the query in this form “a ***...* b” but not in “ a * b * c” which can be updated in the future. We also try to think up a more accurate ranking system for our search engine to present the top 5 results effectively.

----THE END----