Group 12: Project Report

Design issues:

This project requires a search engine for a set of texts in folder, so there are 2 main problems and some small problems:

+ Read all the text file and search key words in it.
+ Dealing with queries.
+ Standardize input words and text files.
+ For better performance, we need to load all the text file first.

Our group's solution:

+ Using a specific tree structure to store all files.
+ Using <vector> to store keyword(s) from console input and pass them to the suitable query.

+ Parsing the files for words before loading to memory and memorize their positions.

Data structures:

After looking from the internet all the types of trees suggested in the pdf file, we decided to use ***Trie*** to store the files.

```
struct node
{
    vector <int> title; // index for article that have keyword in title
    vector < pair<int, int> > position; // article ; position in article
    node* pNext[38];//a->z, 0->9, #, $
    node(); // constructor that sets each element in array pNext to NULL
};
```

We decided to store all 1600 files in 1 tree, so this is what we do:

+ Each node of letter represents a letter from a word.
+ Letters are linked to each other to form words.
+ The node contains a vector which store data of the word.

For example:

ROOT ->A->C->E

// the last node E store data for the whole word “ACE”

->Q->U->I->R->E

// this E node (for “ACQUIRE”) is different node from the one above

->B->E->C->A->U->S->E

……

->Z->E->R->O

Algorithms:

+ Parsing data:

Load up the files, parse each line for words and position of words and insert them into the Trie.

```
vector<string> ParseStream(string & line, vector<int> & posinart, int &linestart)
{
    transform(line.begin(), line.end(), line.begin(), ::tolower); // convert string to lowercase
    size_t prev = 0, pos; // maintain position in string
    vector<string> wordVector;
    string tmp;
    size_t x;
    while ((pos = line.find_first_of(" \t\n\"`~!@%^&*()-_=+|{}[];:,'<.>/\\?", prev)) != string::npos)
    {
        if (pos > prev)
        {
            tmp = line.substr(prev, pos - prev);
            wordVector.push_back(tmp);
            posinart.push_back(prev + linestart);
        }
        prev = pos + 1;
    }
    x = line.length();
    if (prev < x)
    {
        tmp = line.substr(prev, string::npos);
        wordVector.push_back(tmp);
        posinart.push_back(prev + linestart);
    }
    linestart += (x + 1);
    return wordVector;
}
```

+Tree insertion:

```
while (cnt<length) {
    if ((int)(s[cnt] - '0') >= 0 && (int)(s[cnt] - '0') <= 9) {
        tmp = (int)(s[cnt] - '0');
        if (cur->pNext[tmp + 26] == NULL)
            cur->pNext[tmp + 26] = new node;
        cur = cur->pNext[tmp + 26];
    }
    else if ((int)(s[cnt] - 'a') >= 0 && (int)(s[cnt] - 'a') <= (int)('z' - 'a'))
    {
        tmp = (int)(s[cnt] - 'a');
        if (cur->pNext[tmp] == NULL)
            cur->pNext[tmp] = new node;
        cur = cur->pNext[tmp];
    }
    cnt++;
}
if (intitle)cur->title.push_back(article);
cur->position.push_back(make_pair(article, posinart));
return;
```

The core of the insertion process is going through each letter of the words and traversing through the tree to the corresponding nodes and create the data.

+ User input:

When end-user input key words to search, we try to delete all the express mark that we don't need or mistyping, then we try to split the phrase into many small meaning words (English, query-keywords, names, …) and push them to the vector separately.

For example: Input

Christ Pratt	-> <Chris, Pratt>
Tom AND Jerry intitle:toon	-> <Tom, AND, Jerry, intitle:, toon>
Book $100	-> <Book,$100>
Pen $10..$100	-> <Pen,$10..$100>
m,eaningful se,”ntences	-> <meaningful,sentences>
m, eaningful se, “ntences	-> <m, eaningful, se, ntences>
thrill*	-> <thirll*>
john -oliver	-> <john,-,oliver>
#wonderwoman	-><#wonderwoman>

+Query call:

After parsing the input words, we push them to the function query call to decide what to do with these parts of the vector.

If it is a normal word, search it on trie, return back the result.

If it is a query-keyword, push the next elements of vector and the previous result to the suitable query function, and again return back the result.

+Queries:

- Basic: basic search of keywords is considered OR query.
- AND : basic idea is to take intersection from search results.
- OR : same as AND but take union instead.
- INTITLE : check in the “title” vector for results.
- “-“ : find set subtraction between previous results with search results from the query.
- “$” : same as basic search, with some checking conditions.
- “#” : same as basic search, with some checking conditions.
- INRANGE: parse input for start value and end value then search between range accordingly.
- “*” : use BFS in Trie to find incomplete matches.

+Searching on trie:

```
    while (cnt<length) {
        if ((int)(s[cnt] - '0')<10 && (int)(s[cnt] - '0') >= 0)
            tmp = (int)(s[cnt] - '0') + 26;
        else if ((int)(s[cnt] - 'a') >= 0 && (int)(s[cnt] - 'a') <= (int)('z' - 'a'))
            tmp = (int)(s[cnt] - 'a');
        else {
            cnt++;
            continue;
        }
        if (cur->pNext[tmp] == NULL)return NULL;
        cur = cur->pNext[tmp];
        cnt++;
    }
    return cur;
```

Traverse same as insertion but instead of creating data, we return the data present.

+History suggestion:

Use a different Trie to store user input and then apply BFS for current string everytime user type a character. Afterwards output the suggestion results in appropriate manners.

```
if (ch == '\b')
{
    dbg.clear();
    dbg = search_string.substr(0, search_string.length() - 1);
    search_string.clear();
    search_string = dbg;
    cnt_ch -= 2;
}
```

Handles the backspace character

```
else
    search_string.push_back(ch);
system("CLS");
cout << search_string;
res.clear();
res = historySearch(history_root, search_string);
coord.X = 0;
coord.Y = y + cnt + 1;
SetConsoleCursorPosition(h_std_out, coord);
while (!res.empty()) {
    tmp = res.back();
    res.pop_back();
    cout << tmp << endl;
}
++cnt_ch;
coord.X = cnt_ch;
coord.Y = y;
SetConsoleCursorPosition(h_std_out, coord);
ch = _getch();
```

Handles suggestion search and output

+Output results:

Results of search queries is a vector of index of articles. The user interface is designed to output results into pages, each page has 5 articles along with the article's titile and file name. The user can use keyboard input to manuver between pages or continue to search a new query.

```
if (ch == 13)
    break;
system("CLS");
cout << search_string << "\n\nFound " << sz << " results in " << stime << " ms" << "\n\n\n";
if (move_page + 5 >= total) {
    cout << "[" << move_page + 1 << "] - [" << total << "]\n\n";
    copy(page.begin() + move_page, page.end(), screen);
    cout << "\n<< Press <1> to continue previous page\n\n";
}
else if (move_page == 0) {
    cout << "[1] - [5]\n\n";
    copy(page.begin() + move_page, page.begin() + move_page + 5, screen);
    cout << "\nPress <2> to continue next page >>\n\n";
}
else {
    cout << "[" << move_page + 1 << "] - [" << move_page + 5 << "]\n\n";
    copy(page.begin() + move_page, page.begin() + move_page + 5, screen);
    cout << "\n\n<< Press <1> to continue previous page\t\t||\t\tPress <2> to continue next page >>\n\n";
}
cout << "Press <ENTER> to continue searching..." << endl;
```

+Running time:

The average load time of data files is 1220 miliseconds and average load time of queries is below 1 milisecond for simple search, and about 10 miliseconds for short combined queries. Although very long queries such as "a OR the OR you OR i OR but OR because OR have" can take up to 20 miliseconds. Combinations of incomplete match queries "*" migh take longer than 25 miliseconds.

+Optimization issues:

The searching and handling of queries is relatively fast but the data loading process is not optimal. The use of system libaries and a better parsing function could help me improve the load time.

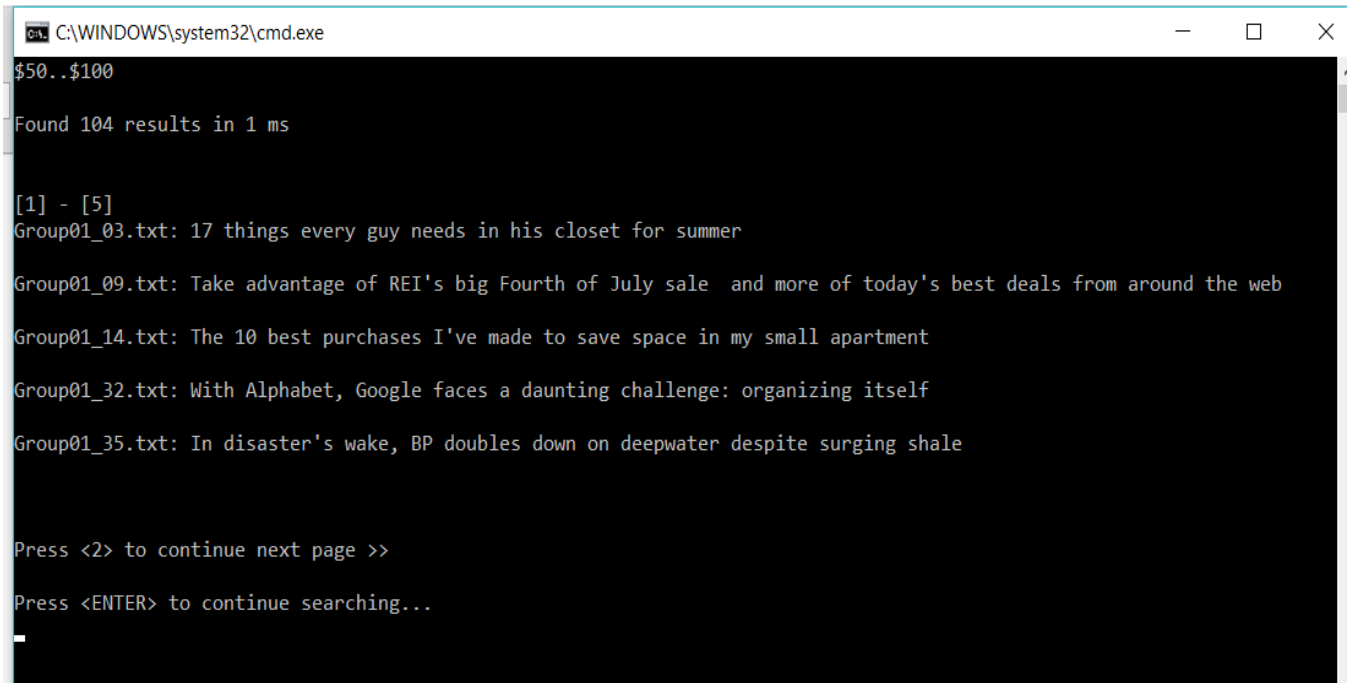
+Scalability:

Our implementation of storing and searching is efficient enough for extremely large data collections but the loading time issues remains.

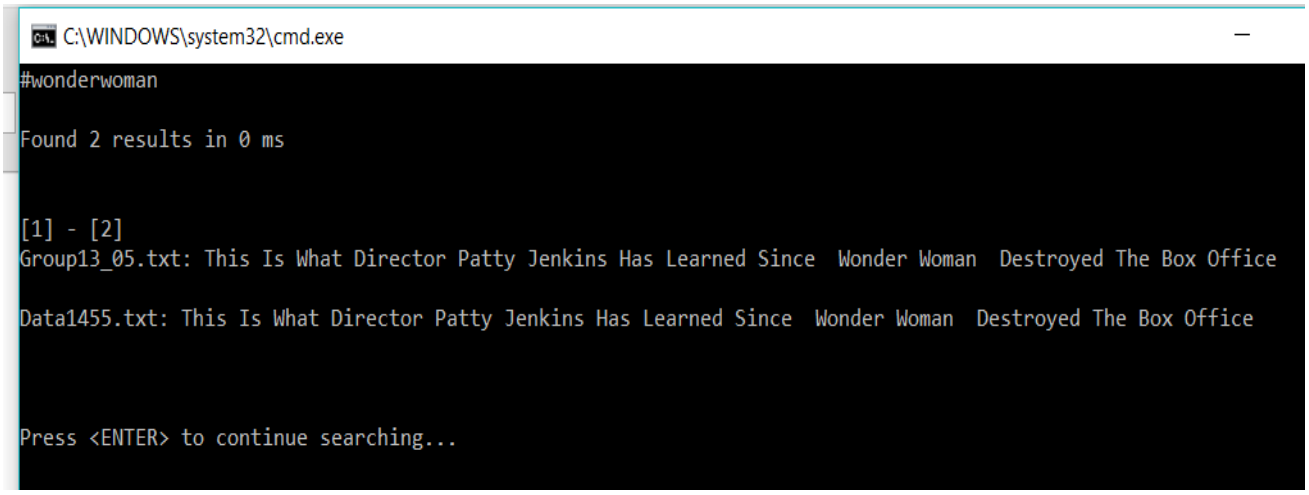
+Examples of some querries:

1. a* a a a a a a a a a a*

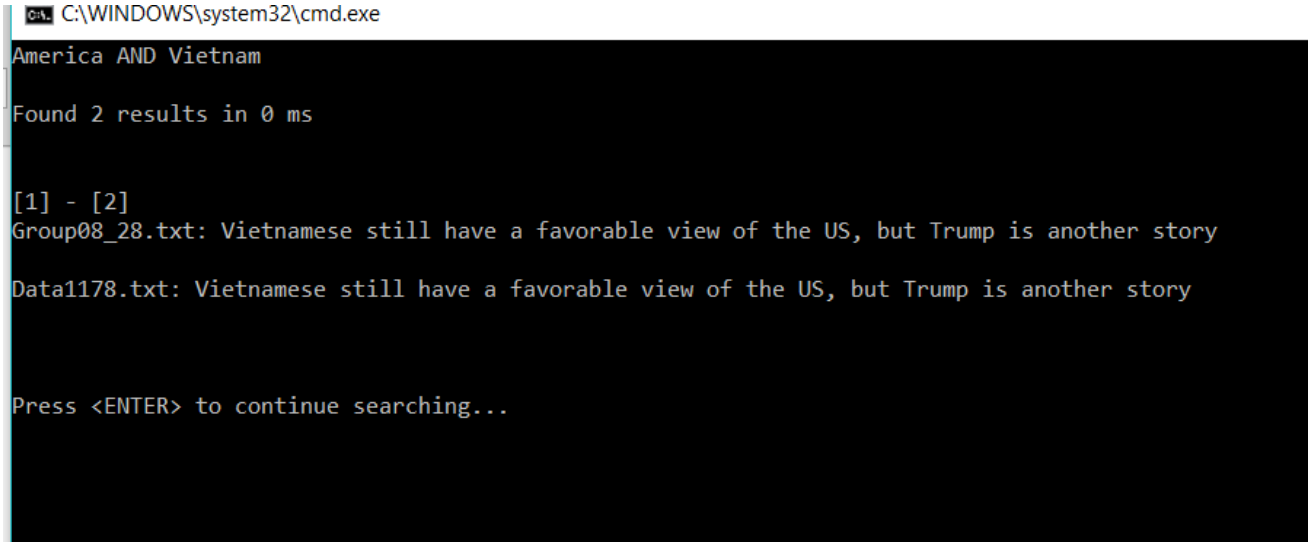
2. $50..$100



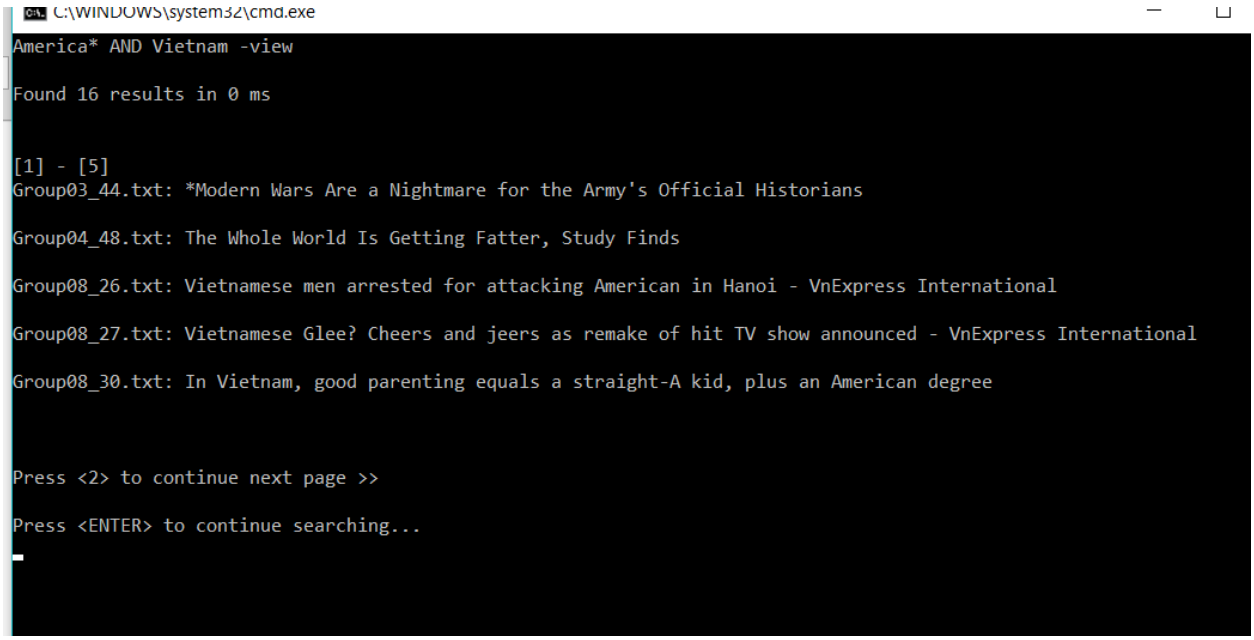
3. #wonderwonman

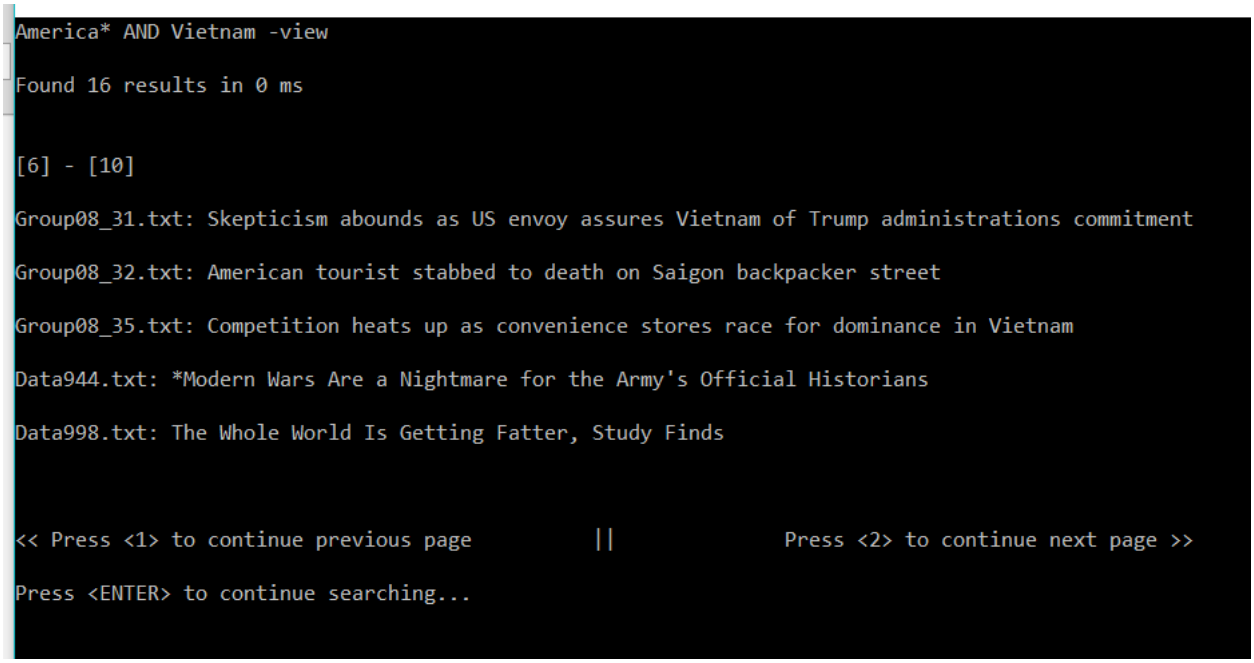


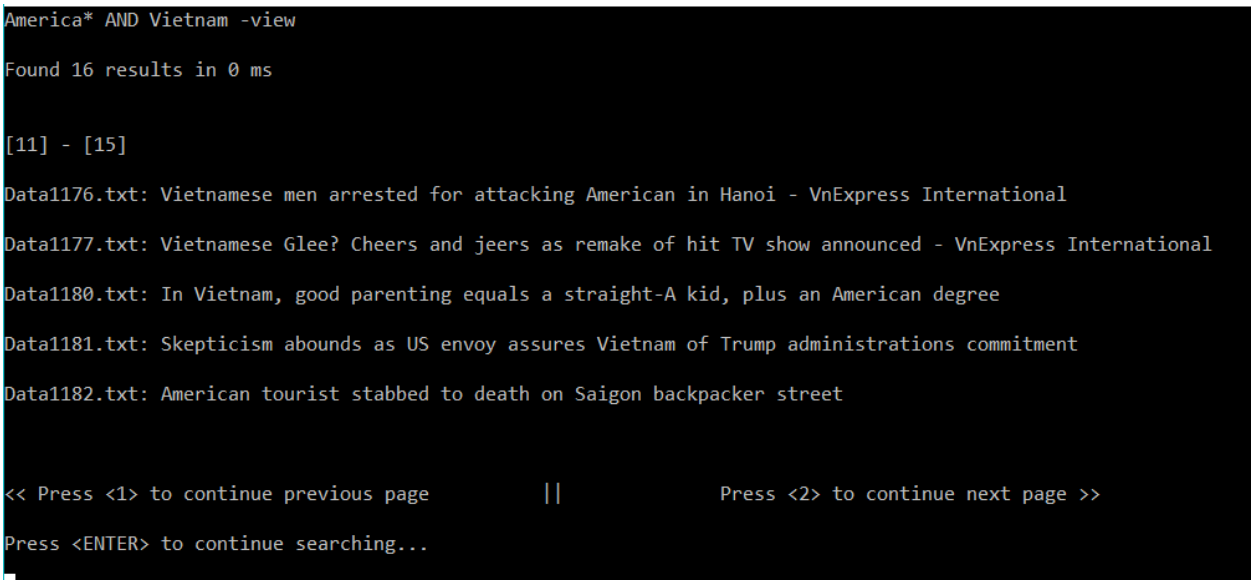
4. America AND Vietnam

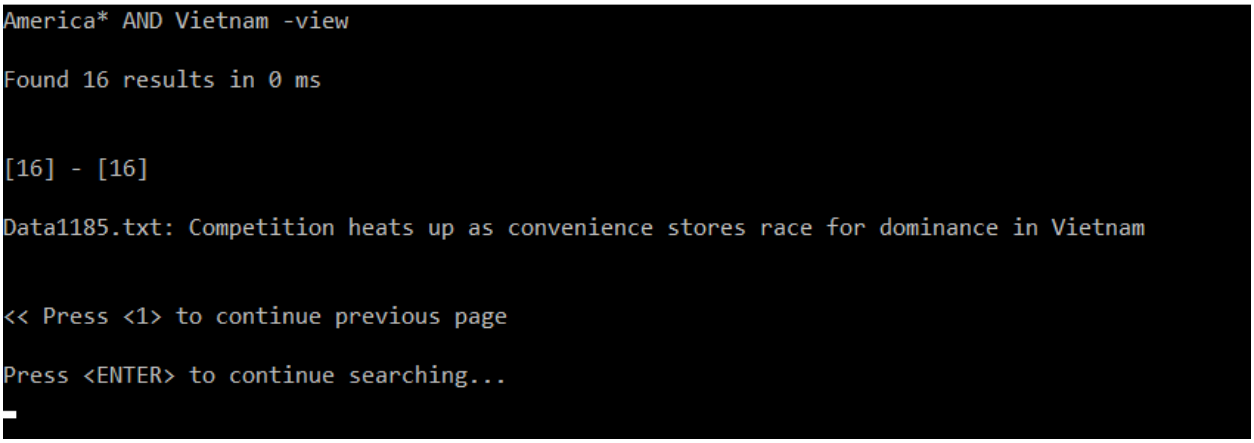


5. America* AND Vietnam -view









STT	Họ và tên	%	List
1651034	Huỳnh Hà Mai Trinh	25	History suggestion, Design UI, Incomplete matches
1651045	Hoàng Đình Hiếu	25	Data structure, In range query, Compile and debug
1651055	Nguyễn Võ Hồng Thắng	25	Query handling, Query combining, Parsing input
1651069	Nguyễn Quốc Việt	25	Parsing data, And, or, intitle Query, Compile and debug