How you implement your code

```
bool exists(const std::string& name)
{
    struct stat buffer;
    return(stat(name.c_str(),&buffer)==0);
}
```

先利用exists()，確認資料夾中的 txt 檔案有幾個。並把所有檔案中的 title 存起來，同時把所有內文 parse 和 split 後 insert 進兩個 Trie 裡。其中一個 Trie 正著存，另一個則先把字串 reverse 後再 insert (用來 suff-search)。

之後打開 query.txt，每次讀入一行並把 split 的結果建成一個 expression tree。經過 expression tree 把各個 node 的答案 intersection 或是 union，就可以把得到答案並輸出到 output 裡。

以下是我 trie 的建造方法。Who 是用來表示是從哪個檔案 insert 的單字，如果他有經過那個 node，就把他 insert 到那個 node 的 whose 中，這樣就能知道這個檔案中有沒有這串字。如果是這個單字的結尾，則額外 insert “who”到 flag 裡，表示在這個文檔中這個詞已經到結尾了。這樣 exact-search 才能知道這個詞是不是跟要找的詞完全一樣。

```
struct Trie {
    set<int> flag;
    set<int> whose;
    struct Trie* son[26]={NULL};
    Trie()
    {
        flag = {};
        whose = {};
    }
};
```

```
void Insert(Trie* root, string s, int who) {
    Trie* temp = root;
    for(char ss:s)
    {
        if(ss<96) ss = ss+32;
        if (temp->son[ss - 'a'] == NULL)
        {
            temp->son[ss - 'a'] = NewTrie();

            temp->son[ss - 'a']->whose.insert(who);
        }
        else temp->son[ss - 'a']->whose.insert(who);
        temp = temp->son[ss - 'a'];

    }
    temp->flag.insert(who);
}
```

Pre-search 跟 suff-search 很像，都是檢測完要的字串後，返回所有那個節點中的 whose，就能知道哪些 data 中有內文經過這個節點。只是 suffix-search 要用反著存的 trie 和反著的關鍵字來搜索。

```
set<int> findTrie_exat(string str)
{
    Trie* p = trie;

    for (char ss:str)
    {
        if (p->son[ss-'a'] == NULL)
            return {};
        p = p->son[ss - 'a'];
    }

    return p->flag;  //此串是字符集中某
}
```

```
set<int> findTrie_suff( string suff)
{
    Trie* p = R_trie;

    for (int i = suff.length()-1;i>=0;i--)
    {
        if (p->son[suff[i]-'a'] == NULL)  //
            return {};
        p = p->son[suff[i] - 'a'];
    }
    return p->whose;  //此串是字符集中某串的字
}
```

```
set<int> findTrie_pre( string pre)
{
    Trie* p = trie;
    for(char c : pre)
    {
        if(p->son[c-'a']==NULL) return {};
        p = p->son[c-'a'];
    }
    return p->whose;
}
```

而 exact-search 則是返回最後節點的 flag，才能保證這是這個單字的最後一個字母。

最後計算結果的部分，則是用 query_tree 和 recursion。將左右 sub_tree 的值回傳後，選擇將兩邊結果並集或是交集並回傳。

Challenges you encounter in this project

1. 時間跑太久。
   原本使用的方法是把每個 data 都建造兩個的 trie，並在查完後直接刪掉，沒有把 trie 存起來，這樣發現跑一千筆資料需要 120 秒。後來嘗試把跑過的 trie 存起來而不 delete 掉，這樣還是需要四到五秒時間。所以最後我嘗試將所有 data 合起來存成兩個 trie，所需時間才大幅下降。
2. Suff-search 不會做。
   本來不知道 suff-search 怎麼使用一個 trie 做出來，但後來上討論區看到很多人使用兩個 trie 反著 insert 才會做。
3. Compile 問題。
   用助教給的 command compile 一直報錯，最後只好借同學的電腦 compile。

References that give you the idea (github/paper…)

https://iter01.com/563709.html 字典樹詳解