

**数**

**据**

**结**

**构**

**课**

**程**

**设**

**计**

**专业班级： 软件工程2003班**

**学 号： U202010851**

**姓 名： 侯皓斐**

**软件工程**

**目 录**

[1 任务分析 2](#_Toc90340204)

[1.1 任务描述 2](#_Toc90340205)

[1.2 模型分析 3](#_Toc90340206)

[2 思路分析与抽象数据类型的构建 7](#_Toc90340207)

[2.1 构建倒排索引 7](#_Toc90340208)

[2.2 searchVSM 15](#_Toc90340209)

[2.3 searchBool 16](#_Toc90340210)

[2.4 InvertedIndex数据结构类与Search功能类 17](#_Toc90340211)

[3 数据结构与算法简要介绍 18](#_Toc90340212)

[3.1 自定义数据类型 18](#_Toc90340213)

[3.2 链表 18](#_Toc90340214)

[3.3 栈与表达式求值 19](#_Toc90340215)

[3.4 堆 20](#_Toc90340216)

[3.5 字典树 22](#_Toc90340217)

[3.6 Splay平衡树 22](#_Toc90340218)

[3.7 排序 25](#_Toc90340219)

[4 测试用例与分析 26](#_Toc90340220)

[5 思考与感悟 31](#_Toc90340221)

[6 源代码 33](#_Toc90340222)

[6.1 LinkedList.cpp 34](#_Toc90340223)

[6.2 Set.cpp 37](#_Toc90340224)

[6.3 Heap.cpp 41](#_Toc90340225)

[6.4 TireTree.cpp 43](#_Toc90340226)

[6.5 InvertedIndex.cpp 46](#_Toc90340227)

[6.6 Search.cpp 55](#_Toc90340228)

[6.7 a.cpp 64](#_Toc90340229)

# 

# 1 任务分析

## 1.1 任务描述

用C语言实现一个基于命令行的英文文本搜索引擎，支持布尔查询、词组查询和基于向量空间模型的排序搜索，基于磁盘实现倒排索引及其相关操作。

## 1.2 模型分析

通过阅读相关资料，了解到为实现一个基于命令行的简单英文文本搜索引擎，需要完成倒排索引，词组查询，布尔查询，基于向量空间模型的排序搜索等部分。

1. 倒排索引

倒排索引的基本思想参见图1。左部称为词项词典。每个词项都有一个记录出现该词项的所有文档的列表，该表中的每个元素记录的是词项在某文档中的一次出现信息（为实现基于向量空间模型的排序搜索和词组查询，该信息中需要包含更多的信息，如出现的频率和位置信息），这个表中的每个元素通常称为倒排记录（posting）。每个词项对应的整个表称为倒排记录表（posting list）。所有词项的倒排记录表一起构成全体倒排记录表（postings）。图 1-3 中的词典按照字母顺序进行排序，而倒排记录表则按照文档ID号进行排序。

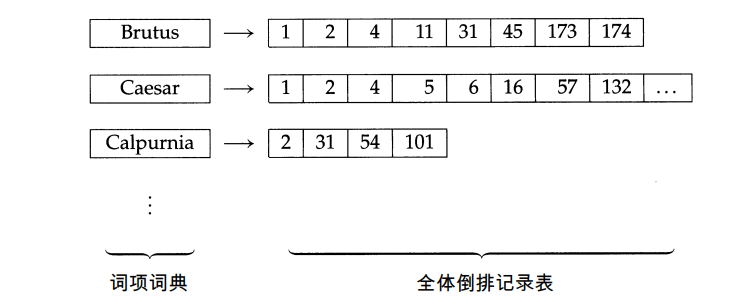


图1 倒排索引的两个组成部分

为获得检索速度的提升，就必须要事先建立索引。建立索引的主要步骤如下。

1. 收集需要建立索引的文档，如：



1. 将每篇文档转换成一个个词条（token）的列表，这个过程通常称为词条化 （tokenization），如：



1. 进行语言学预处理，产生归一化的词条来作为词项（由于时间限制且不是数据结构课程设计的重点，故未实现），如：



1. 对所有文档按照其中出现的词项来建立倒排索引，索引中包括一部词典和一个全体倒排记录表。这个过程基于排序方法，每篇文档的所有词项加上文档 ID（图左部）形成二元组，通过按照字母顺序排序（图中部）。然后，同一词项进行合并。最后，将词项和文档 ID 分开（图右部）。词项存储在词典中，每个词项有一个指针指向倒排记录表。词典中往往还会存储一些其他的概要信息。

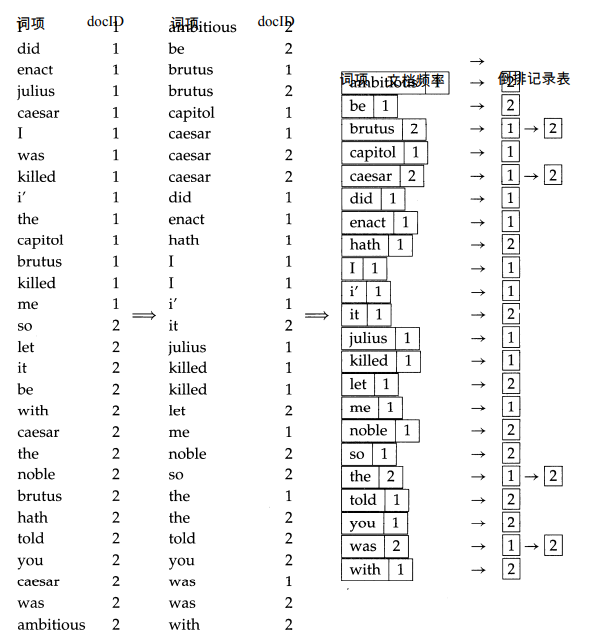


图2 过排序和合并建立倒排索引的过程

在最终得到的倒排索引中，词典和倒排记录表都有存储开销。前者往往放在内存中，而后者由于规模大得多，通常放在磁盘上。

具体实现中对上述流程进行了一定的修改，有利于时间复杂度的降低和代码的构建。

1. 词组查询

我们要在布尔查询中实现不仅可以查询单个词项所在的文档信息（直接取出对应的倒排记录即存在该信息），还能查询多个词项组成的词组所在的文档信息。

我们可以借助于k临近的类似思想解决该问题。若我们已经在倒排索引中维护了每个词项在文档中出现的位置信息，首先我们可以将词组中每个单词所在的文档集合全部求交，在每个词项的这些文档下的位置信息中，判断是否存在随单词在词组中位置递增，位置信息1临近递增的序列。

1. 布尔查询

布尔检索模型接受布尔表达式查询，即通过 AND、OR 及 NOT 等逻辑操作符将词项连接起来的查询。在该模型下，每篇文档只被看成是一系列词的集合。

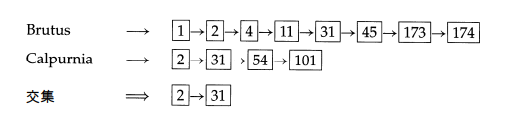


图3 最简单的布尔交查询的例子

（其中通过Brutus和Calpurnia所对应的倒排记录表进行求交计算查询结果）

我们也考虑了更一般情况下的布尔查询，如



这个过程中就需要通过栈来维护交，并，括号运算的优先级，从而实现表达式的求解。

1. 向量空间模型（**V**ector **S**pace **M**odel）

如下图，我们将VSM想象成一个高维空间，每一维代表不同的项(term)，我们假设所有的query和document都能够放在这个高维空间里。d4,d5,…,dM这些向量都代表着不同的document，从d1,d2两个向量我们能够看出，通过使用VSM我们能够捕捉到文档之间的差异。同理，我们也可以用相同的方法来表示Query q。我们可以用数学化的方式定义每一个向量，然后再用两个向量间的点积结果表示两个文档，或查询和文档间的相似度。

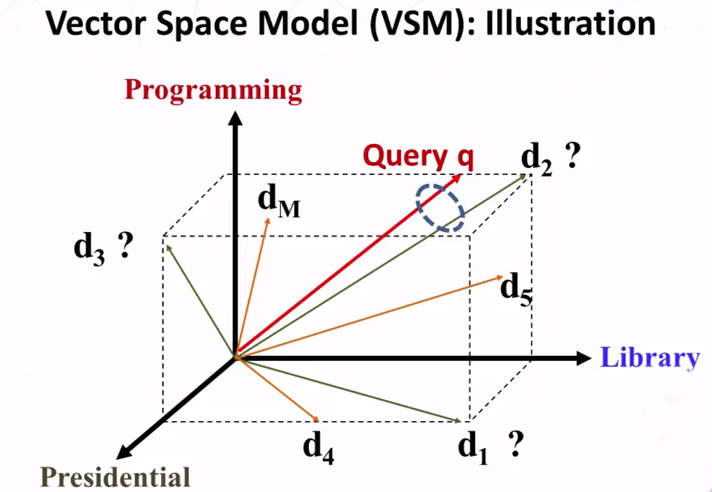
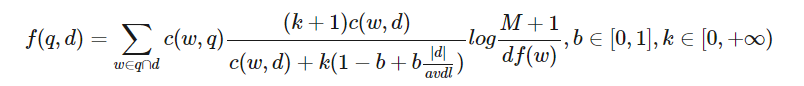


图4 向量空间模型图解

在综合考量多种模型下，最终选定使用BM25（Best Match25）模型进行对每个文档进行向量的构建。



其中：

c(w,q) c(w,d)代表单词w在查询q（文档d）内出现的次数。

k为严格控制一个出现了很多次的词对最终结果的贡献所需要的系数。

|d| 为文档d的长度。avld 为文档的平均长度。

b常数用以控制文本长度归一化函数来惩罚(penalization)一个长文本[[1]](#footnote-1)。

M为总文档的数目，df(w)为w出现的总文档数。

在数学上，我们认为f(q,d)越高，查询q与文档d更相似，成为被搜索到的文本的概率更大。

# 2 思路分析与抽象数据类型的构建

为提高代码的运行效率和简化问题的分析，需要在程序需要的时候设计和使用合理的数据结构。

经过分析，程序主要流程应如下表示：

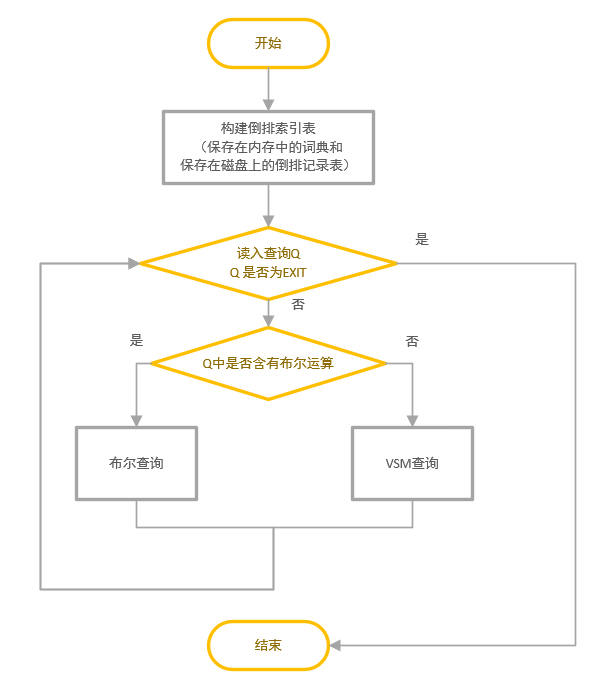


图5 程序基本流程示意图

## 2.1 构建倒排索引

通过分析我们可以看到，倒排索引中我们要维护如下信息：

文档（document）的相关信息：

编号（int），名称（char\* Name），总词数（int）。

单词（string）到它的编号（int）的映射

单词编号（int）到它在倒排记录表中的位置指针（int）的映射

倒排记录表（dictionary.data）中维护每个词项（term）的信息：

单词编号（id，int）,

它所在文档总数（totDoc，int）,

它出现的总频率（totFrequency，int），

它出现的文档信息（docID，vector<int>），

它在不同文档中的频率（docFrequency，vector<int>），

它在不同文档中位置信息（posinfo，vector<LinkedList>）；

前三项应存储在内存中，但也应该在索引构建完成后输出至磁盘，待下一次调用该程序，便可直接读入信息，而不用花费较多时间再度构倒排索引表。

1. 对文档进行编号处理，可以构建以下数据类型：
2. **struct** Document {
3. **char** name[100];
4. **int** WordNumber;
5. };

在内存中Document应存储与一个变长数组（动态数组）中，由于C++泛型相关知识理解的不深入，且由于时间原因，不得不以C++STL库中的vector动态数组加以替代。

构建完倒排索引表后，将所有信息输出到docIndex.data二进制文件中。

（2）建立单词（string）到它的编号（int）的映射：

为避免来回比较单词字符串产生的复杂度，应对每一个单词进行编号分析，建立由word到整型数的映射关系。由于对Hash表随机查找的担忧，且出于减少内存上的负担的考虑，选择字典树（TrieTree）作为字典的存储结构。

1. #define MAX 26
2. **typedef** **struct** TrieNode {                   //Trie结点声明
3. **bool** isWord;                            //标记该结点处是否构成单词
4. **int** id;
5. **struct** TrieNode \*next[MAX];             //儿子分支
6. } Trie;
7. **class** TrieTree {
8. **private**:
9. **int** totWordnum = 0;
10. TrieNode \*root;
11. ofstream fout;
12. **public**: TrieTree() {}
13. **public**: **int** gettotWordnum() {}
14. **public**: **void** settotWordnum(**int** num) {}
15. **public**: **void** insert(string s) {}  //插入单词
16. **public**: **void** insert(string s,**int** id) {}  //在已经建好的表基础上插入
17. **public**: **int** getId(string s) {}  //获取单词的Id，单词不存在或单词为空时返回0
18. **private**: **void** dfs(TrieNode \*p, **char**\* word, **int** len) {}
19. **public**: **void** printWords(string path) {}
20. };

字典树（TrieTree）又称单词查找树、前缀树，是一种树形结构，是一种哈希树的变种。在统计、排序和保存大量的字符串（但不仅限于字符串）是具有更小的时间复杂度，因此可以应用于搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。

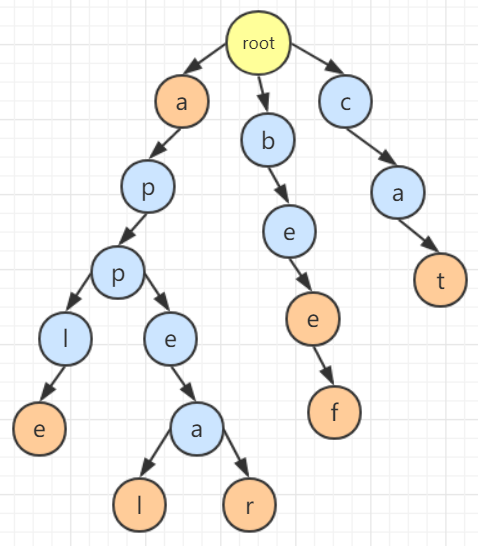


图6 字典树结构示意图

字典树有如下性质：

* 1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符；
  2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串；
  3. 每个节点的所有子节点包含的字符都不相同。

字典树实现了如上描述的功能。

int gettotWordnum() {} 获取总的不同的单词的数目

void settotWordnum(int num) {} 在非首次创建时设立单词的数目

void insert(string s) {} 在首次创建时在字典树中插入单词

void insert(string s,int id) {} 在非首次创建时在字典树中插入单词

int getId(string s) {} 获取一个单词的Id，单词不存在或单词为空时返回0

void dfs(TrieNode \*p, char\* word, int len) {}

void printWords(string path) {} 上述两个函数实现信息向二进制文件WordsIdList.data的输出。

完成docIndex.data，WordsIdList.data

1. 构建倒排记录表（dictionary.data）维护每个词项（term）的信息

为了结构化数据，便于二进制文件的读写，且便于代码的实现，构建term数据类型如下：

1. **struct** TERM {
2. **int** id, totDoc, totFrequency;
3. vector<**int**> docID;
4. vector<**int**> docFrequency;
5. vector<LinkedList> posinfo;
6. };
7. **void** out(ofstream &fout, TERM &term) {}
8. **void** in(ifstream &fin, TERM &term) {}

其中Linkedlist是整型数的有序单链表，能在O（n+m）的时间复杂度内合并(求交或求并)两个有序单链表，与严蔚敏教材定义的带尾指针的单链表别无二致。其具体抽象数据类型如下：

1. **typedef** **struct** Node {
2. **int** num;
3. **struct** Node \*next;
4. } Node;
5. **class** LinkedList {
6. **private**:
7. Node \*start;
8. Node \*end;
9. **public**: LinkedList() {}
10. **public**: **void** init() {}
11. **public**: **void** Free() {}
12. **public**: **void** push\_back(**int** num) {}
13. **public**: Node\* getBegin() {}
14. **public**: Node\* getNext(Node \*p) {}
15. **public**: **void** intersextion(LinkedList a) {}  //有序链表求交集
16. **public**: **void** merge(LinkedList a) {}  //有序链表求并集
17. };

接下来是整个倒排索引构建的核心。dictionary.data构建的流程大致如下：

1. 初始化每一个文档，遍历每一个文档
   1. 将文档中的每一个单词得到id，与当前的单词数的位置信息一起，插入平衡树Splay。
   2. 中序遍历输出平衡树Splay节点上的信息，构建0+docId形式的temp文件。
2. 定义最大合并路K = 200，当前合并深度deep = 1，当前未合并的文档数nowTotDocNumber = totDocNumber， nextTotDocNumber = 0；
3. 若nowTotDocNumber = 1代表已经合并完毕，跳转4，否则继续执行
   1. 枚举i=0；i< nowTotDocNumber;i+=k
   2. 合并（deep-1）+（i）后的min（K, nowTotDocNumber – i）个文档
      1. 将这些中间表的term个数读入，防止读入越界
      2. 将每个中间表的元素读取一个（注意防止读入越界），形成（termId，docId）二元组压入heap堆（首先以termId作为第一关键字排序，由小到大；然后在termId相同的情况下判断docId，docId小则二元组靠前。）
      3. 只要堆不空，弹出堆首元素，进行合并与输出工作，输出至（deep）+(i/K).temp文件中
      4. nextTotDocNumber ++；
   3. deep ++; nowTotDocNumber = nextTotDocNumber; nextTotDocNumber = 0;
4. 将deep+0.temp复制进入dictionary.data文件，注意应加入0元素作为不存在单词id的编号，在此过程中记录每一个term的读入位置，压入一个变长数组WordPostionList。
5. 将WordPostionList输出至二进制文件WordPostionList.data

由于单词的数目较多，而可能会有较多重复的单词，若使用快速排序算法会将大量无关的信息来回搬动，降低程序运行效率，若使用平衡树则可随着当前单词位置递增插入平衡树指定的节点，而且节点处便于维护一个链表，随时递增的存储和维护位置信息，每次插入的复杂度小于快速排序每个元素的平均移动次数。

1. #define Maxn 20000+10
2. **class** Set {
3. **private**:
4. **int** lc[Maxn],rc[Maxn],fa[Maxn],size[Maxn],num[Maxn];
5. **int** data[Maxn];
6. LinkedList posList[Maxn];
7. **int** root = 0,tot = 0,pos = 0, differentNum = 0;
8. ofstream out;
9. **public**: Set () {}
10. **private**: **void** updata (**int** u) {}
11. **private**: **void** zig (**int** x) {}
12. **private**: **void** zag (**int** x) {}
13. **private**: **void** splay (**int** x) {}  //将节点Splay至跟
14. **private**: **void** insert (**int** u,**int** key,**int** pos) {}
15. **private**: **void** MidOrderDfs(**int** root, **int** nowDoc) {}  //中序遍历
16. **public**: **void** Insert (**int** key,**int** pos) {}  //外界插入节点的接口
17. **public**: **void** Search(string path,**int** nowDoc) {}  //有序输出这个文档合并结果
18. **public**: **void** getDifferentNumber(**int** root) {}
19. };

其中Splay函数是Splay平衡树的核心。Splay平衡树中文名为伸展树，由丹尼尔·斯立特Daniel Sleator和罗伯特·恩卓·塔扬Robert Endre Tarjan在1985年发明的。它的主要思想是：对于查找频率较高的节点，通过Splay函数（不断左旋或右旋）使其处于离根节点相对较近的节点。

可以证明，当每次操作后，均将该节点Splay至根节点，每次操作的**均摊时间复杂度**为O（logn）[[2]](#footnote-2)。而Splay代码清晰易调试的特点，使其成为了本次代码中的较优选择。

若按照模型分析中数量的假设，可知初始化文档的复杂度为

逐层合并时，应按照类似外部排序的方式进行合并，同时为了降低合并过程中的时间复杂度，应维护一个堆，每次弹出termId最小的词项（同时保证termId相同时docId最小），然后压入从它所在的文档里下一个termId与docId组成的二元组。其具体抽象数据类型和实现与教材所写基本一致。

1. #define MaxK 230
2. **struct** MergeNode {
3. **int** termId;
4. **int** thisDoc;
5. };
6. **bool** operator < (MergeNode a, MergeNode b) {
7. **if**(a.termId < b.termId)
8. **return** 1;
9. **else** **if**(a.termId == b.termId) {
10. **if**(a.thisDoc < b.thisDoc)
11. **return** 1;
12. **else**
13. **return** 0;
14. }
15. **else**
16. **return** 0;
17. }
18. **class** Heap {
19. **private**:
20. MergeNode a[MaxK];
21. **int** size;
22. **public**: Heap() {}
23. **public**: **int** Size() {}
24. **public**: **void** push(MergeNode key) {}  //压入堆
25. **public**: MergeNode pop() {}  //弹出堆
26. };

合并过程需要进行文件读写操作，而且需要遍历几乎每一个词项多次，复杂度较高。

合并一层，每一个term都要被读入，被压入堆，弹出，合并，然后输出。而我们需要合并约次。故合并形成倒排索引表的时间复杂度应为

1. 总结在内存和外存中建立倒排索引的过程，并进行分析。

可构建算法流程图大致如下：

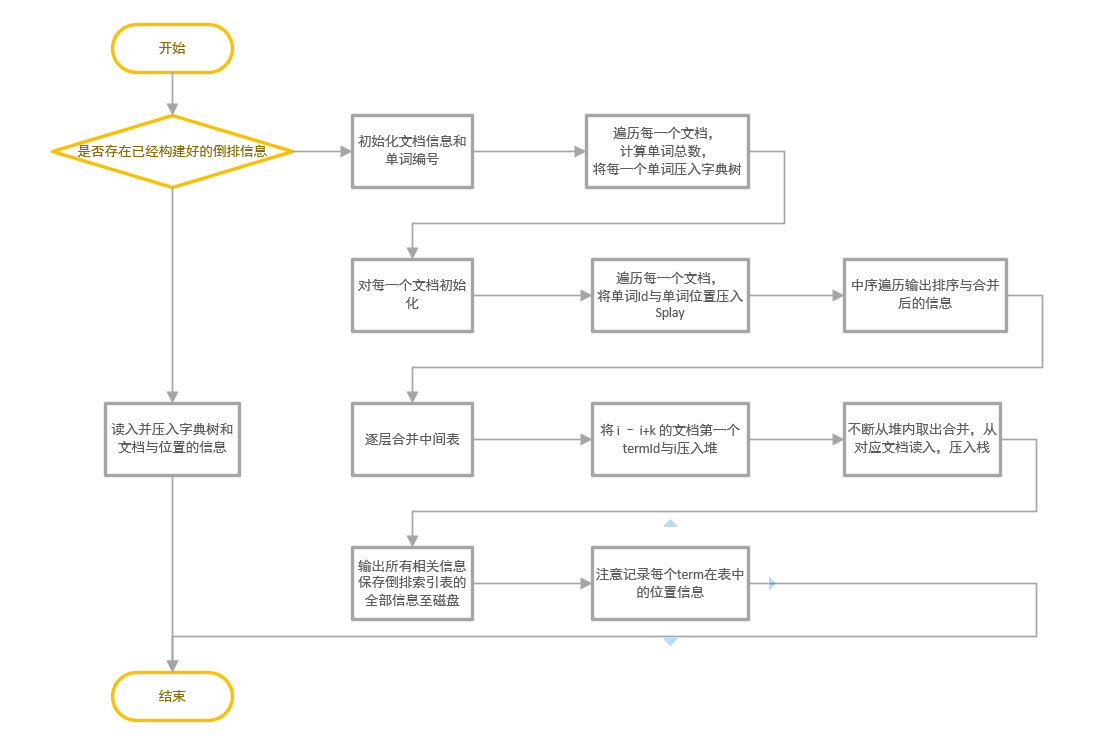


图6 倒排索引表构建整体流程示意图

整体分析其时间复杂度应为：

本次测试集选取为CGTN上11月19日共1327篇文章与莎士比亚的《哈姆雷特》第三幕第一场[[3]](#footnote-3)。其大约计算量应在PC短时间可计算能力之内。

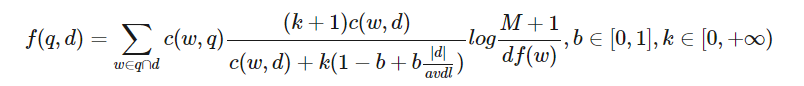
其最坏空间复杂度应为：

估算后，其占用内存绝不会超过2G，不会出现内存爆炸问题。

而最多同时使用的句柄数不会超过，程序能够正常输入输出。[[4]](#footnote-4)

## 2.2 searchVSM

基于向量空间模型进行排序搜索，我们对于一个查询操作query应该提出其中的单词，然后拉出它的倒排索引记录，该单词所在的文档的评分结果应按照BM25模型加上如下值



可抽象出如下数据类型，便于存贮文档的评分信息和排序后查找出文档编号。

1. **class** VSMcross {//叉积结果
2. **public**:
3. **int** docId;
4. **double** crossans;
5. **void** operator = (**const** VSMcross &D) {
6. docId = D.docId;
7. crossans = D.crossans;
8. }
9. };

然后使用堆排序，先使用大根堆的构建算法，然后取出前k个评分较大的VSMcross节点，输出其对应的文件名。该过程时间复杂度为

但在合并时我们就使用了堆这一数据结构，而由于C++泛型机制使用不熟练，被迫改使用归并排序的将整个结果数组排序，然后取出前k大评分的文档输出。两者只有当M较大时才会产生较大的差别。

整个过程时间复杂度约为

## 2.3 searchBool

布尔检索需要实现在单个词查询与词组查询的基础上。

单个词汇的查询较为简单，取出其倒排索引记录，将所有docId压入一个集合（链表）即可获得检索结果文档集合。

有p个词汇组成的词组不仅时所有词汇所在文档集合的交集，更要满足在这个文档中存在，（），随i增大pos以1的速度增大的序列。因此可以使用如下算法求解词组查询。

1.取出词组中所有单词的Id，从而取出其倒排索引中的所有信息。

2.将其每个单词的docId集合求交得到答案候补集合。

3.遍历这个答案候补集合，取出每一个可能的文档，

a)取出每个文档对应的位置信息集合，然后取出头指针

b)为保证位置信息递增，若第i个的单词位置小于第i-1个单词位置，指针向后遍历。

c)若出现逐个+1的序列，答案压入此文档。否则一直未出现则不存在该词组。

该算法的时间复杂度应为

我们这就将Bool检索式，转化为了简单的集合的Bool运算。我们为实现集合的交并运算与括号，应使用栈（LIFO）维护中缀表达式的求解。（此处设置较运算优先级高于并运算）。此处计算中缀表达式的算法设计可以类似于严蔚敏《数据结构》教材52页的算法设计。定义表达式节点如下：

1. **struct** ExpressionNode {
2. **int** n; // 0 运算符 1 词 n>1 长度为n的短语
3. **char** operation;
4. LinkedList keyword;
5. LinkedList Set;
6. };

整体一次查询的复杂度最坏为

## 2.4 InvertedIndex数据结构类与Search功能类

回顾与总结，倒排索引本质也是数据存储和分析的一种形式，其也为一种数据结构。而我们也要实现Search的功能类，其内部聚合InvertedIndex类。

1. **class** InvertedIndex {
2. **private**:
3. string path = "newsDoc";
4. string TEMPpath = "TEMP";
5. **int** totDocsNumber = 0;
6. **double** aveBookWordsNumber = 0;
7. vector<Document> Books;
8. vector<**int**> WordPostionList;
9. TrieTree Words;
10. **private**: **void** initDocsandWords() {}  //初始化文档信息和字典树
11. **private**: **void** initEveryDoc() {}  //将每个文档合并
12. **private**: **void** mergeDoc(**int** startDoc, **int** k, **int** deep) {}  //合并k个文档
13. **private**: **int** Merge() {}  //合并所有文档，返回合并的深度
14. **private**: **void** outInfomation(**int** finalDeep) {}  //输出内存保存的信息
15. **Private: void** ReadWordsIdList() {}  //若构建好直接读入压入内存
16. **public**: **void** Build() {}  //构建倒排索引
17. **public**: **int** getWordId(string word) {}
18. **public**: **int** gettotDocsNumber() {}
19. **public**: **int** getReadPosion(**int** wordid) {}
20. **public**: **int** getBookWordsNumber(**int** bookid) {}
21. **public**: **double** getaveBookWordsNumber() {}
22. **public**: **char**\* getDocName(**int** docid) {}
23. };
24. **class** Search {
25. **private**:
26. InvertedIndex a;
27. **double** k = 1.5;  //重要常数
28. **double** b = 0.75;  //重要常数
29. VSMcross tempstore[MAXDOCXNUM];
30. **int** precede[6][6] = {};
31. **private**: **void** Sort(VSMcross \*a, **int** l, **int** r) {}  //归并排序
32. **private**: **void** searchVSM(string str) {}
33. **private**: **int** getOptId(**char** a) {}  //得到运算符的编号，用以计算优先级
34. **private**: **int** Precede(**char** a, **char** b) {}   //计算优先级
35. **private**: LinkedList getDocSet(ExpressionNode expresionnnode) {}
36. **private**: **void** searchBool(string str) {}
37. **public**: **void** main() {}  //外界调用
38. };

# 3 数据结构与算法简要介绍

## 3.1 自定义数据类型

本项目为更加合理的结构化数据和实现算法，设计了如下数据类型：

1. **struct** Document {
2. **char** name[100];
3. **int** WordNumber;
4. };
5. **struct** TERM {
6. **int** id, totDoc, totFrequency;
7. vector<**int**> docID;
8. vector<**int**> docFrequency;
9. vector<LinkedList> posinfo;
10. };
11. **class** VSMcross {//叉积结果
12. **public**:
13. **int** docId;
14. **double** crossans;
15. **void** operator = (**const** VSMcross &D) {
16. docId = D.docId;
17. crossans = D.crossans;
18. }
19. };
20. **struct** ExpressionNode {
21. **int** n; // 0 运算符 1 词 n>1 长度为n的短语
22. **char** operation;
23. LinkedList keyword;
24. LinkedList Set;
25. };

本项目也为了程序实现更优秀的时间复杂度和空间利用率综合使用了链表，变长数组，堆，栈，平衡树，字典树等数据结构和外部排序，归并排序等经典算法。

## 3.2 链表

本代码中使用的便是记录尾指针的单向链表。而其实现的功能如上文抽象数据类型定义所定义。且本项目可以保证链表内均为有序递增数据，故可以实现求交求并的运算。具体算法和代码实现可参阅整体代码。

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。 相比于线性表顺序结构，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而线性表和顺序表相应的时间复杂度分别是O(logn)和O(1)。

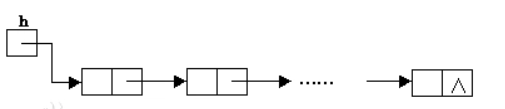


图7 链表的结构示意图

使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。链表最明显的好处就是，常规数组排列关联项目的方式可能不同于这些数据项目在记忆体或磁盘上顺序，数据的存取往往要在不同的排列顺序中转换。链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。

## 3.3 栈与表达式求值

本项目使用数组实现栈，令top指针指向栈内元素的上一个，故空栈时top指向0。栈开的足够大，保证其不会溢出。而本项目中的栈并未被显性的实现，形成抽象数据类型。但在计算集合交并运算中发挥了巨大作用。

栈（stack）又名堆栈，它是一种运算受限的线性表。限定仅在表尾进行插入和删除操作的线性表。这一端被称为栈顶，相对地，把另一端称为栈底。向一个栈插入新元素又称作进栈、入栈或压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素；从一个栈删除元素又称作出栈或退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素。

算法思想如下：

设： s1----操作数栈,存放暂不运算的数和中间结果

s2----算符栈,存放暂不运算的算符

1.置s1,s2为空栈；开始符#进s2；

2.从表达式读取“单词”w----操作集合/算符

3.当w!=‘#’ || s2的顶算符!=‘#’时, 重复：

3.1 若w为操作集合，则w进s1,读取下一“单词”w；

3.2 若w为算符，则：

3.2.1 若 prio(s2的顶算符(θ1)) < prio(w(θ2)),则:

w进s2；读取下一“单词”w；

3.2.2 若 prio(s2的顶算符(θ1))=prio(w(θ2)) ，且=“)”

去括号， pop(s2)； 读取下一“单词”w；

3.2.3 若 prio(s2的顶算符(θ1)) > prio(w(θ2))，则：

{ pop(s1,a)；pop(s1,b)；pop(s2,op)；

c=b op a； push(s1,c)； /\*op为θ1\*/

}

4. s1的栈顶元素为表达式的值。

## 3.4 堆

堆在本项目中对动态维护最小值起到了非常重要的作用，极大的加快了外部排序的速度，降低了时间复杂度。

堆（heap）也被称为优先队列（priority queue）。队列中允许的操作是先进先出（FIFO），在队尾插入元素，在队头取出元素。而堆也是一样，在堆底插入元素，在堆顶取出元素，但是堆中元素的排列不是按照到来的先后顺序，而是按照一定的优先顺序排列的。这个优先顺序可以是元素的大小或者其他规则。最大堆中堆顶的元素是整个堆中最大的，并且每一个分支也可以看成一个最大堆。同样的，我们可以定义最小堆。

堆可以看成一个完全二叉树，每次总是先填满上一层，再在下一层从左往右依次插入。

堆的插入步骤：

1.将新元素增加到堆的末尾

2.按照优先顺序，将新元素与其父节点比较，如果新元素小于父节点则将两者交换位置。

3.不断进行第2步操作，直到不需要交换新元素和父节点，或者达到堆顶

4.最后通过得到一个最小堆

通过将新元素与父节点调整交换的操作叫做上滤(percolate up)。

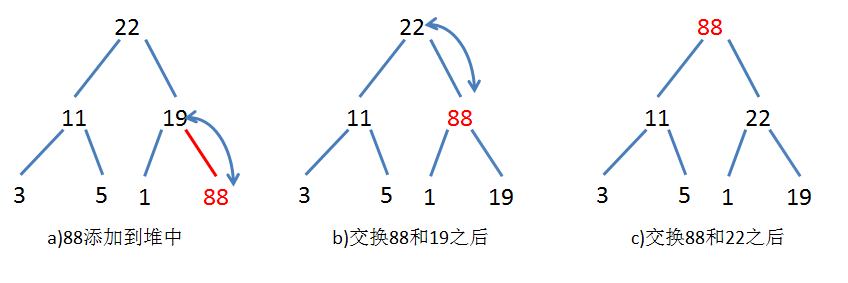


图8 堆的插入操作示意图

堆的删除操作与插入操作相反，插入操作从下往上调整堆，而删除操作则从上往下调整堆。

1.删除堆顶元素（通常是将堆顶元素放置在数组的末尾）

2.比较左右子节点，将小的元素上调。

3.不断进行步骤2，直到不需要调整或者调整到堆底。

上述调整的方法称为下滤（percolate down）。

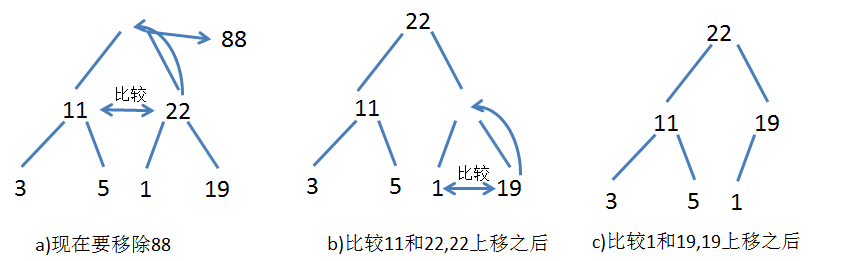


图8 堆的插入操作示意图

## 3.5 字典树

字典树及其实现的功能在上文已经有过初步描述，本项目为在内存中以较好的空间复杂度和时间复杂读保存单词到整型数的映射关系，构建字典树。相较于Hash表其稳定的表现令人满意。

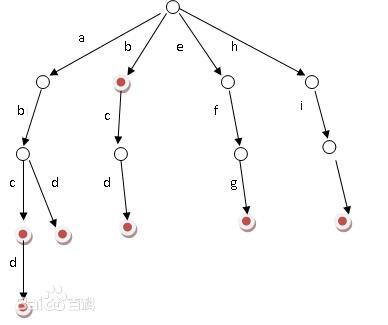


图9 字典树的结构示意图

搜索字典树内某个单词的方法为：

1.从根结点开始一次搜索；

2.取得要查找关键词的第一个字母，并根据该字母选择对应的子树并转到该子树继续进行检索；

3.在相应的子树上，取得要查找关键词的第二个字母,并进一步选择对应的子树进行检索。

4.迭代2，3过程

5.在某个结点处，关键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找。判断是否已打上is\_word的标记。若是，则查询成功。若中途退出或最后节点未被标记，则查找失败。

其他操作类似处理。

## 3.6 Splay平衡树

Splay是一种特殊的二叉查找树，它保证了每次插入和查找的均摊复杂度为O（logn）。为了使 Splay 保持平衡而进行旋转操作，旋转的本质是将某个节点上移一个位置。在 Splay 中旋转分为两种：左旋和右旋。

旋转需要保证：整棵 Splay 的中序遍历不变（不能破坏二叉查找树的性质）。受影响的节点维护的信息依然正确有效。root 必须指向旋转后的根节点。

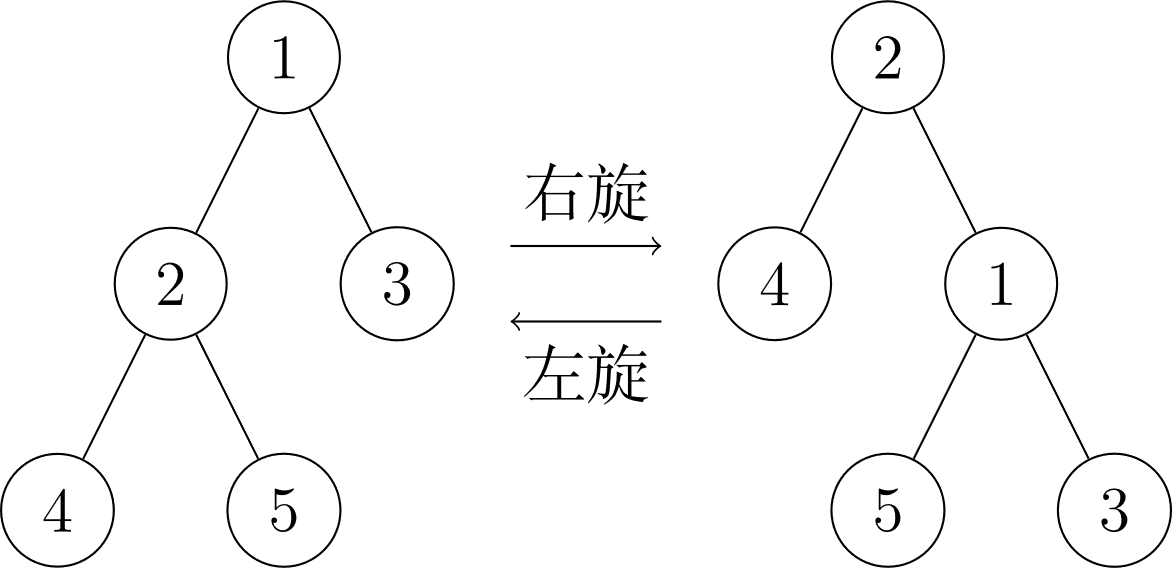


图10 Splay的左旋与右旋

Splay 规定：每访问一个节点后都要强制将其旋转到根节点。此时旋转操作具体分为6种情况讨论（其中x为需要旋转到根的节点）正是Splay操作保证了单词操作的均摊复杂度为O（logn）！而其他二叉搜索树操作基本未变。

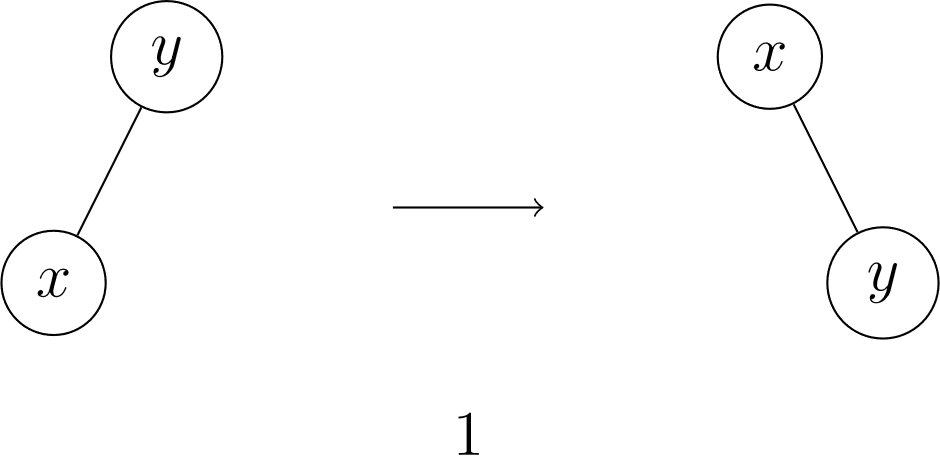
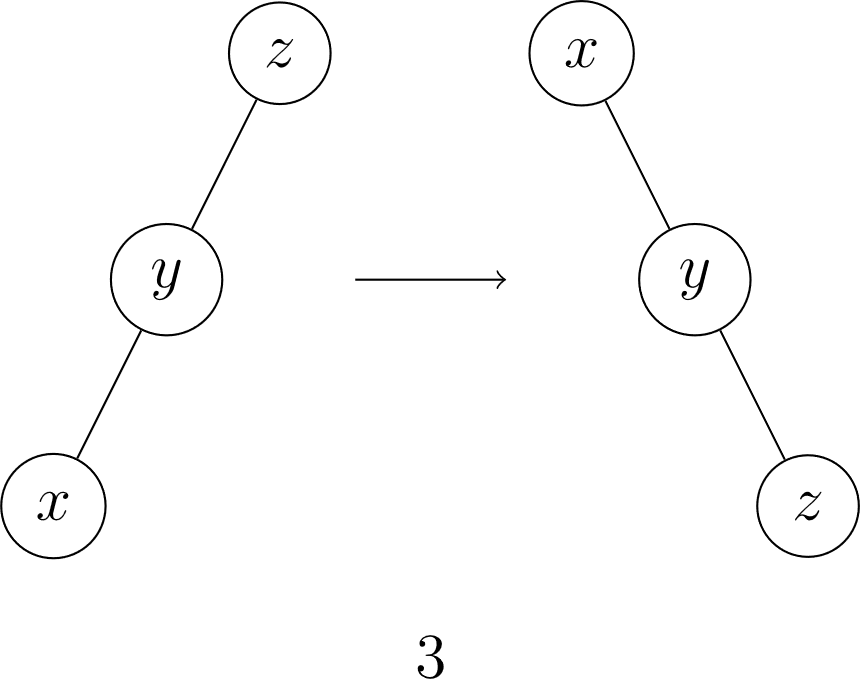
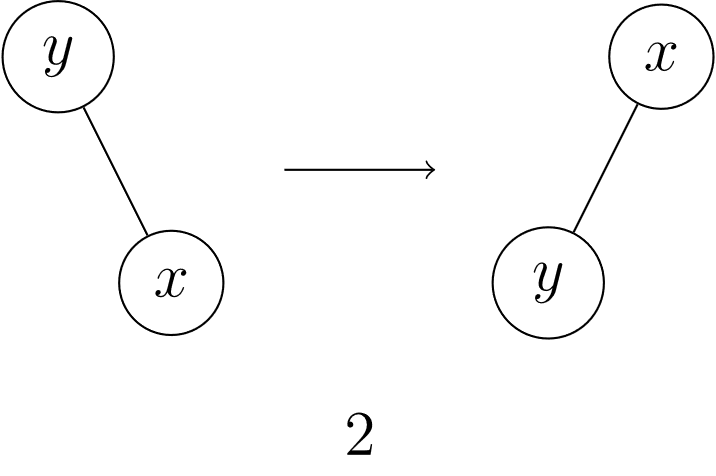
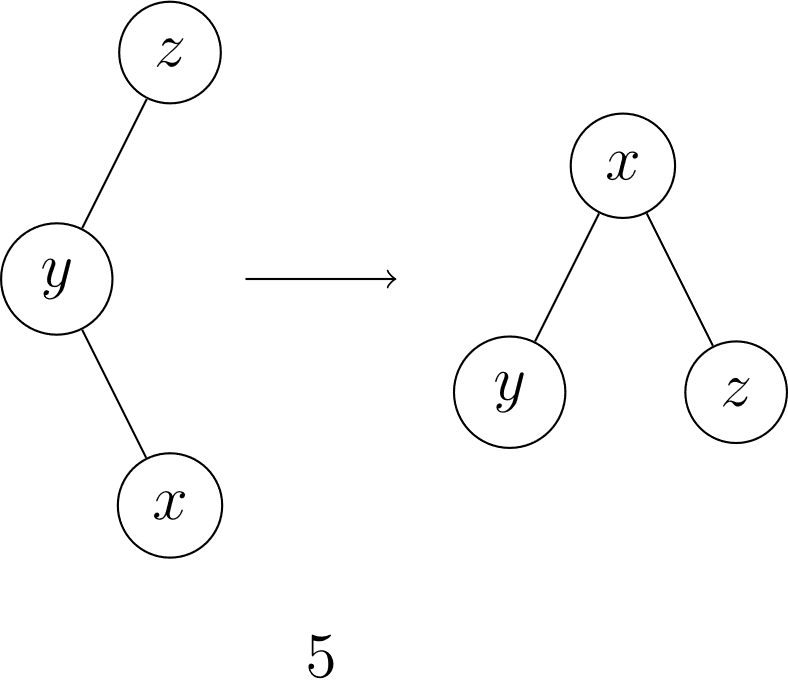
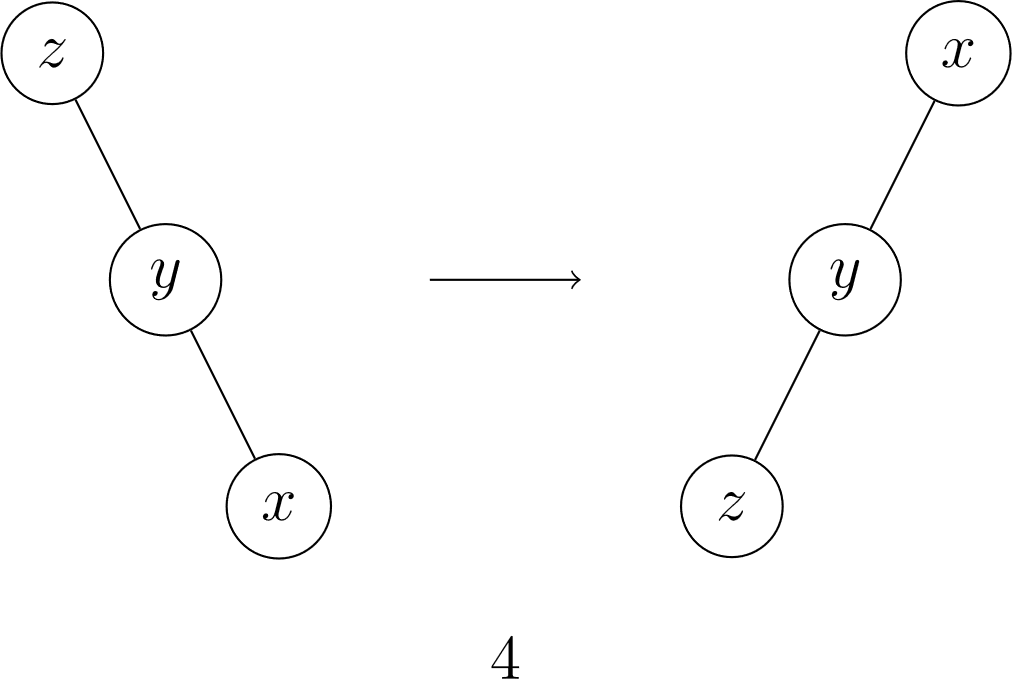
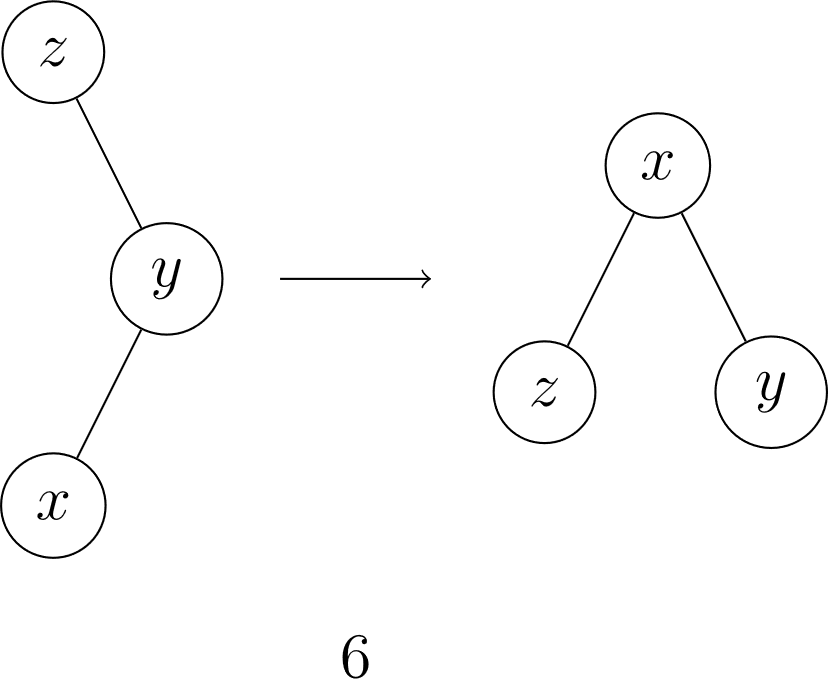
   

图10 Splay到根节点过程中可能会出现的六种情况

如果x的父亲是根节点，直接将x左旋或右旋。

如果x的父亲不是根节点，且x和父亲的儿子类型相同，首先将其父亲左旋或右旋，然后将x右旋或左旋。

如果x的父亲不是根节点，且x和父亲的儿子类型不同，将x左旋再右旋、或者右旋再左旋。

1. **private**: **void** updata (**int** u) {
2. size[u]=size[lc[u]]+size[rc[u]]+num[u];
3. **return** ;
4. }
5. **private**: **void** zig (**int** x) {  //右旋
6. **int** y=fa[x],z=fa[y];
7. fa[x]=z;fa[y]=x;fa[rc[x]]=y;
8. **if**(lc[z]==y)        lc[z]=x;
9. **else**                rc[z]=x;
10. lc[y]=rc[x];rc[x]=y;
11. updata(y);
12. }
13. **private**: **void** zag (**int** x) {  //左旋
14. **int** y=fa[x],z=fa[y];
15. fa[x]=z;fa[y]=x;fa[lc[x]]=y;
16. **if**(lc[z]==y)        lc[z]=x;
17. **else**                rc[z]=x;
18. rc[y]=lc[x];lc[x]=y;
19. updata(y);
20. }
21. **private**: **void** splay (**int** x) {
22. **while**(fa[x]) {
23. **int** y=fa[x],z=fa[y];
24. **if**(z) {
25. **if**(lc[z]==y)
26. **if**(lc[y]==x)        zig(x),zig(x);
27. **else**                zag(x),zig(x);
28. **else**
29. **if**(lc[y]==x)        zig(x),zag(x);
30. **else**                zag(x),zag(x);
31. }
32. **else**
33. **if**(lc[y]==x)            zig(x);
34. **else**                    zag(x);
35. }
36. updata(x);
37. root=x;
38. }

## 3.7 排序

外部排序的具体思想在上文思路分析中已得到完整地表述。下面介绍归并排序的具体思想。

归并排序是用分治（divide and conquer）思想，分治模式在每一层递归上有三个步骤：

分解（Divide）：将n个元素分成个含n/2个元素的子序列。

解决（Conquer）：用合并排序法对两个子序列递归的排序。

合并（Combine）：合并两个已排序的子序列已得到排序结果。两个子序列已有序，可以通过指针来回比较划过一边合并。

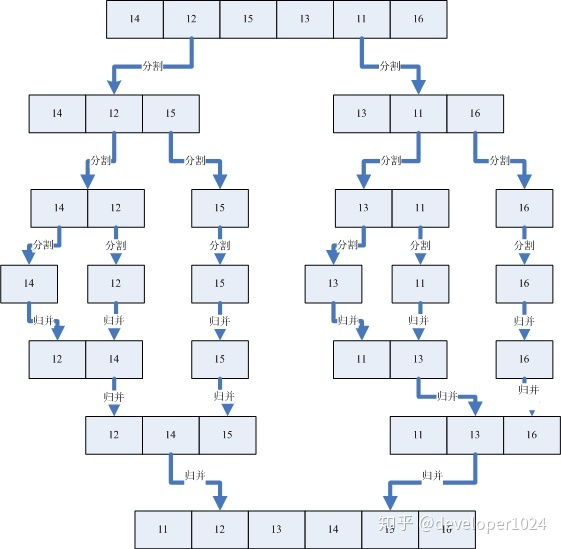


图11 归并排序的例子

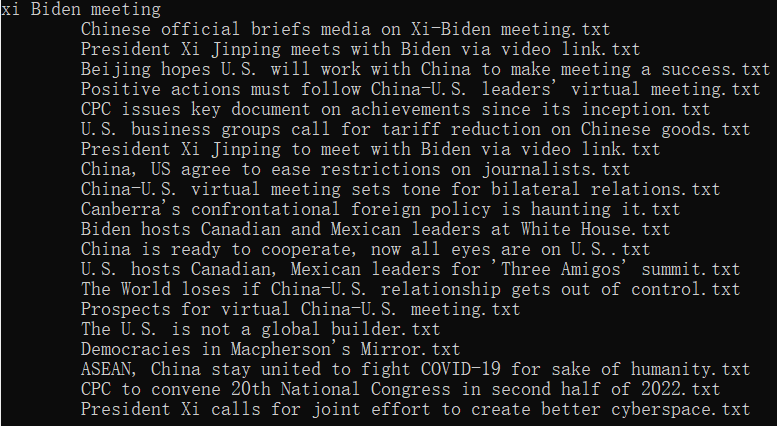
它将一个T(n)的问题，转换为两个T(n/2)的问题，花费了O（n）的复杂度。由主定理容易知道其时间复杂度为O（nlogn）。

# 4 测试用例与分析

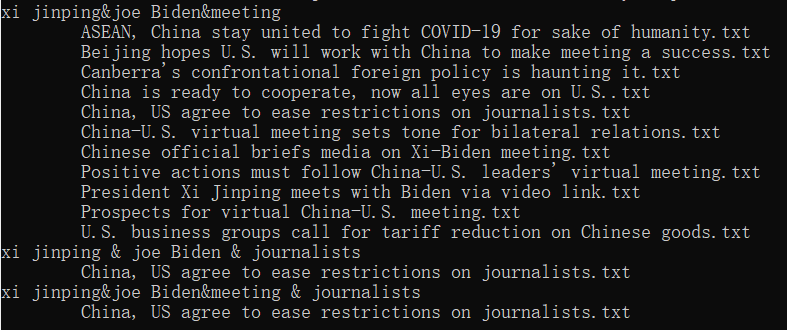
正如前文所说，使用python与开源newspaper库[[5]](#footnote-5)爬取CGTN11月19日新闻1327篇文章并结合莎士比亚的《哈姆雷特》第三幕第一场组成搜索的文档集合。以此建立倒排索引。

11月19日主要发生的头条事件为习近平同志与美国总统视频会晤，就两国记者问题达成一致。

当我们搜索“习拜会”时，搜索引擎返回如下：



而我们使用Bool检索时：



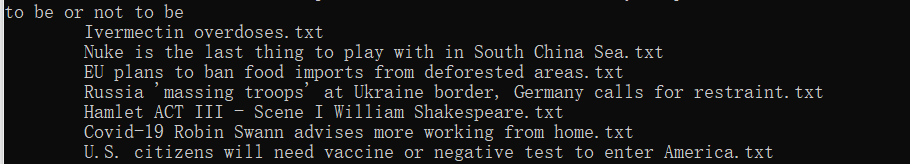
我们初步验证了两种搜索方式和倒排索引建立没有较大问题。对于post的相关query两者均返回了极为有价值和意义的文件。我们可以通过这些文件了解到“习拜会”的前因后果，主要问题。

我们也可以初步看到这样一个问题，VSM排序搜索模型返回的文档可能在数学上相似度较高，但实际与我们想要的信息可能仍有差距。如返回的文档The U.S. is not a global builder.txt和“习拜会”并无直接关系。可以通过调整k，b的常数值得到部分修正。而Bool检索的交运算过于强大，将文本直接限定，只可能越来越小，可能失去了一些我们想要的信息。当我们搜索xi jinping & joe Biden & journalists时，返回值竟然只有一个[[6]](#footnote-6)。



当我们提交含有更多干扰信息的查询时，由于BM25良好的限制，将the，of，and等常见词的权重压低。而实际上他们也确实对于搜索结果产生的贡献可以说是微乎其微。

而我突然想到如果当我们搜索“to be or not to be”这类均由常见词汇组成的语句，VSM能否正常发挥？

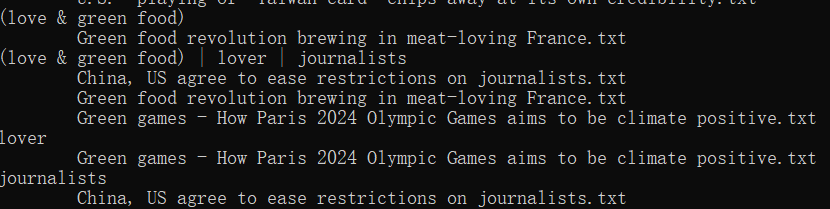


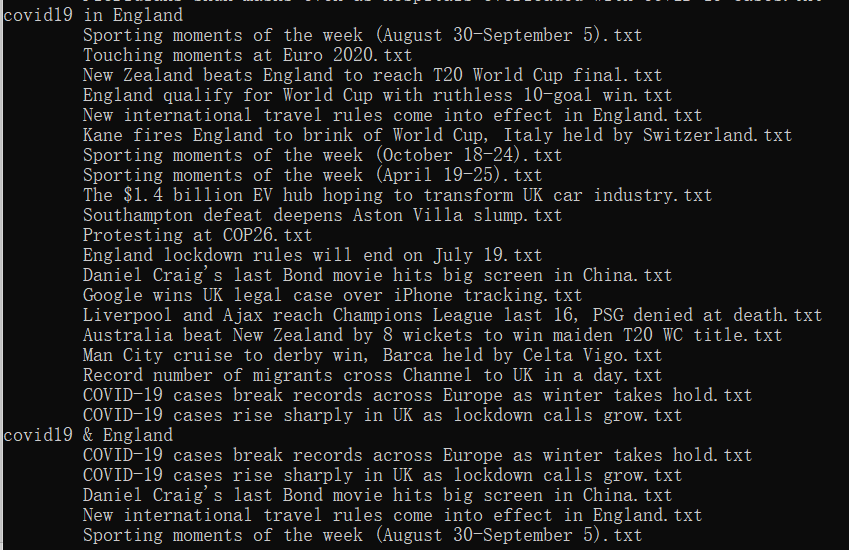
搜索返回了我们想要的值，但是排序并不靠前。而当我们使用Bool检索：



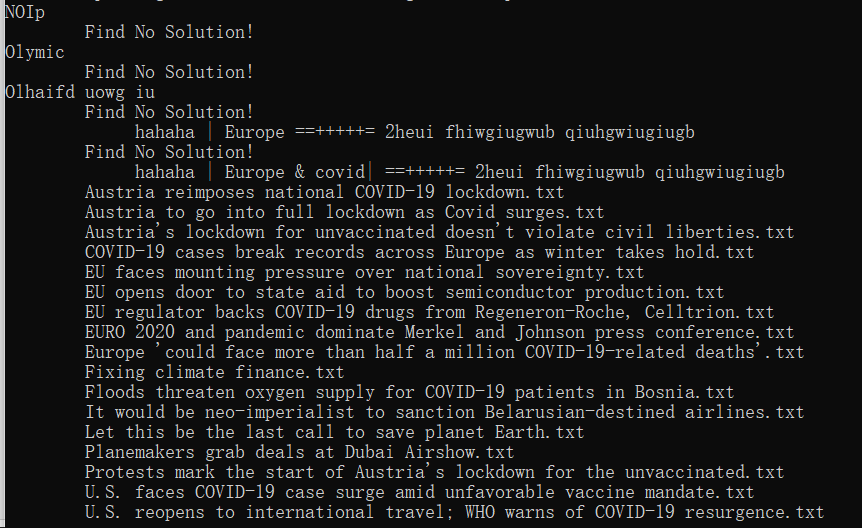
Bool检索使用词组搜索精准的得到了我们想要的文档。

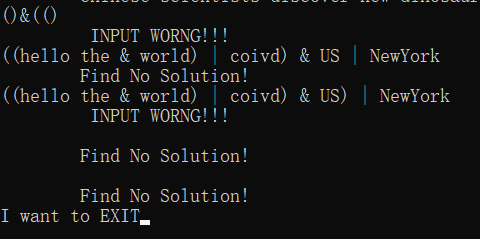
我们接下来通过几组查询继续寻找程序可能存在的问题。

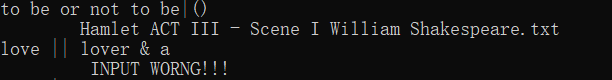




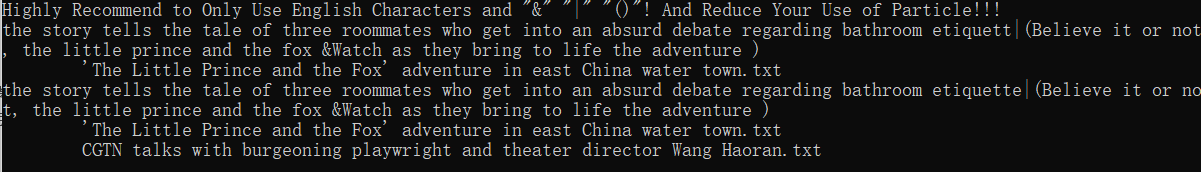
明显上面一组关于新冠肺炎的查询，VSM搜索表现很差。可能是由于covid19本身为常见词，被相关函数压低了权值。



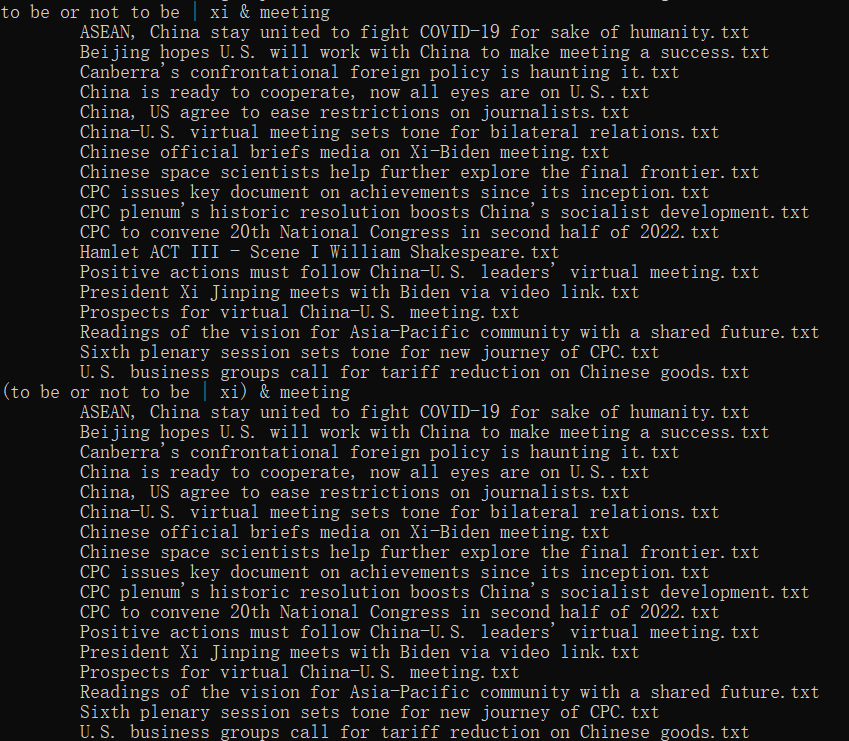




有上三组测试可看出，程序具有较好的鲁棒性。（设定回车后程序关闭）



经过测试可以看出词组查询和布尔查询内应该没有Bug存在。



可以看到程序可以严格按照优先级计算，第二次搜索中搜不到《哈姆雷特》。

程序到目前为止的测试中尚未发现重大运行时异常，也没有发现输出错误等Bug存在。可以基本认定该算法与数据结构的实现已经满足第三层次的正确性：对于精心选择的，苛刻的，带有刁难性的几组输入数据能够得出满足规格说明要求的结果。

# 5 思考与感悟

当我写下这段文字的时候，正是武汉凌晨一点的夜晚。但是我并不疲惫，我又想起了那些年，“轻敲键盘落指有声”揪着头发写算法题的那些日子。Splay，Tire树这些当年经典的算法的记忆被唤醒，那段冲击省队的辉煌和落寞又在脑海中一幕幕上演。

话题拉回到我们的课程设计，在这次数据结构课程设计中我学会了很多：

首先，是从分析问题到解决问题，建立模型的能力。从实际问题出发考虑数据结构和算法的落地，这是信息学竞赛中少有的思维。而当视野开阔后，我能发现数据结构更加深层次的那些东西。数据结构的根本意义不是“用空间换时间”，“保证优秀的时间复杂度”，而是合理的组织，存储数据，在具体应用中获得更好的效果，并不是越高级的数据结构越好。

第二点，我也增加了自己的学习能力，短短的时间内，学习《信息检索导论》又完成了1200+行的代码，属实是保肝加时间紧凑。

第三点．我也更加体会到了老师所讲的“抽象化”的更深层的含义。竞赛中的代码最多不超过300行，基本不可复用，写一套程序费一套程序。完全没有必要去分析函数如何定义，数据结构如何定义能让程序更加简洁，更加易懂，若是能压到一个函数中便就这样做了。但是在这次代码中确实明确感受到若能合理的抽象化数据可以很好的帮助我们，例如term可以有in，out函数，不需要每次写一大堆读入输出。

本次项目进行过一次痛苦的重构，其主要原因有两点。第一点我不了解系统能同时开启的文件数是有限的，在合并时暴力M路合并，程序运行异常无法解决。第二点，我没有采用二进制文件随机读写，将所用的词条分别存储在不同的acsii文件中，文件检索，读取时间过慢！也有着抽象化做的不好，程序函数普遍较大，数据结构不明晰的原因。但最后这个原因现在也并没有完全解决好，重构时间过于紧张，周末奋战重构，时间堪堪赶上。

程序本能实现的更好，由于个人能力和时间等种种原因进行了不少阉割。未进行语言预处理。BM25模型中k，b两个参数的值来自于我的大致估算，没有能结合后续测试继续调节参数以达到更好的评分效果。

下面将是我实现的源代码，正如前文所言，时间仓促，未能做好完整的备注，但结合前面的思路分析是不难理解的。说来惭愧，函数过于臃肿，理解较难的问题有所改进，但仍是本项目的一个重大缺点，老师也请多多理解。

而且老师要求使用C语言，由于我的疏忽第一版使用了C++编写代码，由于惯性和时间原因，不得不保留现在这个状态。而且我也使用了C++STL中的vector代替了本应自己实现的变长数组，这是一个重大错误。老师也请多多理解。

特别感谢辅导课程设计的刘小锋老师为我提供了这样的机会，也特别感谢李国徽院长在《数据结构》这门课程上给我们的敦敦教诲。前面的思路分析是我这次课程设计思考的浓缩性总结，虽然已经费尽心血，可能会出现纰漏，代码亦是如此，请老师多多指教。

本项目支持开源社区，现已将项目完整开源至

[git@github.com:yuechuhaoxi020609/searchVSM.git](mailto:git@github.com:yuechuhaoxi020609/searchVSM.git)

2021.12.14

侯皓斐

于武汉华中科技大学

# 6 源代码

项目文件夹下包含两个子目录，newsDoc，TEMP。分别用以存放被搜索的文档，TEMP存放临时表文件。

项目文件夹下有七个程序：

* 1. cpp.
  2. Heap.cpp
  3. InvertedIndex.cpp
  4. LinkedList.cpp
  5. Search.cpp
  6. Set.cpp
  7. TireTree.cpp

现将其完整展示如下：

## 6.1 LinkedList.cpp

1. //LinkedList.cpp
2. #ifndef \_LINKEDLIST\_
3. #define \_LINKEDLIST\_
4. #include <cstdio>
5. #include <cstdlib>
6. **using** **namespace** std;
7. **typedef** **struct** Node { //节点的定义
8. **int** num;
9. **struct** Node \*next;
10. } Node;
11. **class** LinkedList {
12. **private**:
13. Node \*start;
14. Node \*end;// 记录尾指针，便于在末尾O（1）插入元素
15. **public**: LinkedList() {
16. start = end = (Node \*)malloc(**sizeof**(Node));
17. start->next = NULL;
18. }
19. **public**: **void** init() {
20. start = end = (Node \*)malloc(**sizeof**(Node));
21. start->next = NULL;
22. }
23. **public**: **void** Free() { //释放内存
24. Node \*p = start;Node \*q = p->next;
25. **while**(q != NULL) {
26. free(p);
27. p = q;
28. q = q->next;
29. }
30. **return** ;
31. }
32. **public**: **void** push\_back(**int** num) { //在链表最后插入值
33. Node \*p = (Node\*)malloc(**sizeof**(Node));
34. p->num = num; p->next = NULL;
35. end->next = p; end = p;
36. **return** ;
37. }
38. **public**: Node\* getBegin() { //获得首元素指针
39. **return** start->next;
40. }
41. **public**: Node\* getNext(Node \*p) {
42. **return** p->next;
43. }
44. **public**: **void** intersextion(LinkedList a) { //两有序链表求交集
45. Node \*q = a.getBegin();
46. Node \*p = start->next;
47. Node \*newstart, \*newend;
48. newstart = newend = (Node \*)malloc(**sizeof**(Node));
49. newstart->next = NULL;
50. **while**(p != NULL && q != NULL) {
51. **if**((p->num) < (q->num)) {
52. p = p->next;
53. }
54. **else** **if**((p->num) == (q->num)) {
55. Node\* r = (Node \*)malloc(**sizeof**(Node));
56. r->num = p->num; r->next = NULL;
57. newend->next = r; newend = r;
58. p = p->next; q = q->next;
59. }
60. **else** **if**((p->num) > (q->num)) {
61. q = q->next;
62. }
63. }
64. Free(); a.Free();
65. start = newstart;
66. end = newend;
67. **return** ;
68. }
69. **public**: **void** merge(LinkedList a) { //两有序链表求并集
70. Node \*q = a.getBegin();
71. Node \*p = start->next;
72. Node \*newstart, \*newend;
73. newstart = newend = (Node \*)malloc(**sizeof**(Node));
74. newstart->next = NULL;
75. **while**(p != NULL && q != NULL) {
76. **if**((p->num) < (q->num)) {
77. Node\* r = (Node \*)malloc(**sizeof**(Node));
78. r->num = p->num; r->next = NULL;
79. newend->next = r; newend = r;
80. p = p->next;
81. }
82. **else** **if**((p->num) == (q->num)) {
83. Node\* r = (Node \*)malloc(**sizeof**(Node));
84. r->num = p->num; r->next = NULL;
85. newend->next = r; newend = r;
86. p = p->next; q = q->next;
87. }
88. **else** **if**((p->num) > (q->num)) {
89. Node\* r = (Node \*)malloc(**sizeof**(Node));
90. r->num = q->num; r->next = NULL;
91. newend->next = r; newend = r;
92. q = q->next;
93. }
94. }
95. **while**(p != NULL) {
96. Node\* r = (Node \*)malloc(**sizeof**(Node));
97. r->num = p->num; r->next = NULL;
98. newend->next = r; newend = r;
99. p = p->next;
100. }
101. **while**(q != NULL) {
102. Node\* r = (Node \*)malloc(**sizeof**(Node));
103. r->num = q->num; r->next = NULL;
104. newend->next = r; newend = r;
105. q = q->next;
106. }
107. Free(); a.Free();
108. start = newstart;
109. end = newend;
110. **return** ;
111. }
112. };
113. #endif

## 6.2 Set.cpp

1. #ifndef SET
2. #define SET
3. #include <string>
4. #include <vector>
5. #include <dirent.h>
6. #include <fstream>
7. #include <sstream>
8. #include <iostream>
9. #include "LinkedList.cpp"
10. #define Maxn 20000+10
11. **class** Set {
12. **private**:
13. **int** lc[Maxn],rc[Maxn],fa[Maxn],size[Maxn],num[Maxn];
14. **int** data[Maxn];
15. LinkedList posList[Maxn];
16. **int** root = 0,tot = 0,pos = 0, differentNum = 0;
17. ofstream out;
18. **public**: Set () {
19. **for**(**int** i = 0; i < Maxn; i++)
20. lc[i] = rc[i] = fa[i] = size[i] = num[i] = 0;
21. }
22. **private**: **void** updata (**int** u) {
23. size[u]=size[lc[u]]+size[rc[u]]+num[u];
24. **return** ;
25. }
26. **private**: **void** zig (**int** x) { //右旋
27. **int** y=fa[x],z=fa[y];
28. fa[x]=z;fa[y]=x;fa[rc[x]]=y;
29. **if**(lc[z]==y)        lc[z]=x;
30. **else**                rc[z]=x;
31. lc[y]=rc[x];rc[x]=y;
32. updata(y);
33. }
34. **private**: **void** zag (**int** x) { //左旋
35. **int** y=fa[x],z=fa[y];
36. fa[x]=z;fa[y]=x;fa[lc[x]]=y;
37. **if**(lc[z]==y)        lc[z]=x;
38. **else**                rc[z]=x;
39. rc[y]=lc[x];lc[x]=y;
40. updata(y);
41. }
42. **private**: **void** splay (**int** x) { //将x节点绕至根节点
43. **while**(fa[x]) {
44. **int** y=fa[x],z=fa[y];
45. **if**(z) {
46. **if**(lc[z]==y)
47. **if**(lc[y]==x)        zig(x),zig(x);
48. **else**                zag(x),zig(x);
49. **else**
50. **if**(lc[y]==x)        zig(x),zag(x);
51. **else**                zag(x),zag(x);
52. }
53. **else**
54. **if**(lc[y]==x)            zig(x);
55. **else**                    zag(x);
56. }
57. updata(x);
58. root=x;
59. }
60. **private**: **void** insert (**int** u,**int** key,**int** pos) {
61. **if**(key < data[u]) {
62. **if**(lc[u])
63. insert(lc[u], key, pos);
64. **else** {
65. lc[u]=++tot;
66. data[tot]=key;
67. fa[tot]=u;
68. size[tot]=1;
69. num[tot]=1;
70. posList[tot].init();
71. posList[tot].push\_back(pos);
72. splay(tot);
73. }
74. }
75. **else** **if**(key == data[u]) {
76. num[u]++;
77. posList[u].push\_back(pos);
78. splay(u);
79. **return** ;
80. }
81. **else** {
82. **if**(rc[u])
83. insert(rc[u], key, pos);
84. **else** {
85. rc[u]=++tot;
86. data[tot]=key;
87. fa[tot]=u;
88. size[tot]=1;
89. num[tot] = 1;
90. posList[tot].init();
91. posList[tot].push\_back(pos);
92. splay(tot);
93. }
94. }
95. }
96. **private**: **void** MidOrderDfs(**int** root, **int** nowDoc) { //中序遍历，输出文档初始化结果
97. **if**(!root)
98. **return** ;
99. MidOrderDfs(lc[root], nowDoc);
100. out.write((**char**\*)&data[root], **sizeof**(**int**));
101. **int** totDocFrequency = 1;
102. out.write((**char**\*)&totDocFrequency, **sizeof**(**int**));
103. out.write((**char**\*)&num[root], **sizeof**(**int**));
104. out.write((**char**\*)&nowDoc, **sizeof**(**int**));
105. out.write((**char**\*)&num[root], **sizeof**(**int**));
106. Node \*p = posList[root].getBegin();
107. **while**(p != NULL) {
108. out.write((**char**\*)&(p->num), **sizeof**(**int**));
109. p = p->next;
110. }
111. //cout << endl;
112. MidOrderDfs(rc[root], nowDoc);
113. }
114. **public**: **void** Insert (**int** key,**int** pos) { //外部插入调用接口 其调用insert函数
115. **if**(!root) {
116. root=++tot;data[root]=key;size[root]=num[root]=1;
117. posList[root].init();
118. posList[root].push\_back(pos);
119. }
120. **else**
121. insert(root, key, pos);
122. }
123. **public**: **void** Search(string path,**int** nowDoc) { //外部输出调用接口
124. out.open(path, ios::out | ios::binary);
125. getDifferentNumber(root);
126. out.write((**char**\*)&differentNum, **sizeof**(**int**));
127. MidOrderDfs(root, nowDoc);
128. }
129. **public**: **void** getDifferentNumber(**int** root) {
130. **if**(!root)
131. **return** ;
132. getDifferentNumber(lc[root]);
133. differentNum++;
134. getDifferentNumber(rc[root]);
135. }
136. };
137. #endif

## 6.3 Heap.cpp

1. #ifndef HEAP
2. #define HEAP
3. #define MaxK 230
4. **using** **namespace** std;
5. **struct** MergeNode { //堆维护外部排序要合并的节点
6. **int** termId;
7. **int** thisDoc;
8. };
9. **bool** operator < (MergeNode a, MergeNode b) { //以termId为第一关键字，thisDoc为第二关键字
10. **if**(a.termId < b.termId)
11. **return** 1;
12. **else** **if**(a.termId == b.termId) {
13. **if**(a.thisDoc < b.thisDoc)
14. **return** 1;
15. **else**
16. **return** 0;
17. }
18. **else**
19. **return** 0;
20. }
21. **bool** operator > (MergeNode a, MergeNode b) {
22. **return** !(a<b);
23. }
24. **class** Heap {
25. **private**:
26. MergeNode a[MaxK];
27. **int** size;
28. **public**: Heap() {
29. size = 0;
30. }
31. **public**: **int** Size() {
32. **return** size;
33. }
34. **public**: **void** push(MergeNode key) { //压入节点
35. size++;
36. **int** i;
37. **for** (i = size; i > 1 && a[i>>1] > key; i >>= 1)
38. a[i]=a[i>>1];
39. a[i]=key;
40. }
41. **public**: MergeNode pop() { //弹出节点
42. MergeNode p, ans = a[1];
43. swap(a[1], a[size]);
44. p = a[1];
45. **int** s = 1; size = size - 1;
46. **for**(**int** j = 2\*s; j <= size; s = j, j = j\*2) {
47. **if**(j+1 <= size && a[j+1] < a[j])
48. j++;
49. **if**(p < a[j])
50. **break**;
51. a[s] = a[j];
52. }
53. a[s] = p;
54. **return** ans;
55. }
56. };
57. #endif

## 6.4 TireTree.cpp

1. #ifndef TIRETREE
2. #define TIRETREE
3. #include <cstdlib>
4. #include <string>
5. #include <fstream>
6. #include <sstream>
7. #include <iostream>
8. **using** **namespace** std;
9. #define MAX 26
10. **typedef** **struct** TrieNode {                   //Trie结点声明
11. **bool** isWord;                            //标记该结点处是否构成单词
12. **int** id;
13. **struct** TrieNode \*next[MAX];             //儿子分支
14. } Trie;
15. **class** TrieTree {
16. **private**:
17. **int** totWordnum = 0;
18. TrieNode \*root;
19. ofstream fout;
20. **public**: TrieTree() {
21. root = (TrieNode\*) malloc (**sizeof**(TrieNode));
22. **for**(**int** i = 0; i < MAX ; i++)
23. root->next[i] = NULL;
24. root->id = root->isWord = 0;
25. }
26. **public**: **int** gettotWordnum() { //获取总的不同的单词的数目
27. **return** totWordnum;
28. }
29. **public**: **void** settotWordnum(**int** num) { //在非首次创建时设立单词的数目
30. totWordnum = num;
31. }
32. **public**: **void** insert(string s) { //在首次创建时在字典树中插入单词
33. TrieNode \*p = root;
34. **for**(**int** i = 0; i < s.size(); i++) {
35. **if**(p->next[s[i] - 'a'] == NULL) {
36. TrieNode \*q = (TrieNode\*) malloc (**sizeof**(TrieNode));
37. **for**(**int** j = 0; j < MAX; j++)
38. q->next[j] = NULL;
39. q->isWord = **false**;
40. q->id = 0;
41. p->next[s[i] - 'a'] = q;
42. p = p->next[s[i] - 'a'];
43. }
44. **else**
45. p = p->next[s[i] - 'a'];
46. }
47. p->isWord = **true**;
48. **if**(!(p->id))
49. p->id = ++ totWordnum;
50. }
51. **public**: **void** insert(string s,**int** id) { //在非首次创建时在字典树中插入单词
52. TrieNode \*p = root;
53. **for**(**int** i = 0; i < s.size(); i++) {
54. **if**(p->next[s[i] - 'a'] == NULL) {
55. TrieNode \*q = (TrieNode\*) malloc (**sizeof**(TrieNode));
56. **for**(**int** i = 0; i < MAX; i++)
57. q->next[i] = NULL;
58. q->isWord = **false**;
59. q->id = 0;
60. p->next[s[i] - 'a'] = q;
61. p = p->next[s[i] - 'a'];
62. }
63. **else**
64. p = p->next[s[i] - 'a'];
65. }
66. p->isWord = **true**;
67. **if**(!(p->id))
68. p->id = id;
69. }
70. **public**: **int** getId(string s) { //获取一个单词的Id，单词不存在或单词为空时返回0
71. **if**(s.length() == 0)
72. **return** 0;
73. TrieNode\* p = root;
74. **for** (**int** i = 0; i < s.length(); i++)
75. **if**(p->next[s[i] - 'a'] != NULL)
76. p = p->next[s[i] - 'a'];
77. **else**
78. **return** 0;
79. **if**(p != NULL && p->isWord)
80. **return** p->id;
81. **else**
82. **return** 0;
83. }
84. **private**: **void** dfs(TrieNode \*p, **char**\* word, **int** len) {
85. **if**(p->isWord) {
86. fout.write(word, 30\***sizeof**(**char**));
87. fout.write((**char**\*)&(p->id), **sizeof**(**int**));
88. }
89. **for**(**int** i = 0; i < MAX; i++) {
90. word[len++] = 'a' + i;
91. word[len] = '\0';
92. **if**(p->next[i])
93. dfs(p->next[i], word, len);
94. word[--len] = '\0';
95. }
96. }
97. **public**: **void** printWords(string path) { //上述两个函数实现信息向二进制文件WordsIdList.data的输出。
98. fout.open(path, ios::out | ios::binary);
99. **char** word[30]; **int** len = 0;
100. fout.write((**char**\*)&totWordnum, **sizeof**(**int**));
101. dfs(root, word, len);
102. }
103. };
104. #endif

## 6.5 InvertedIndex.cpp

1. #ifndef INVERTEDINDEX
2. #define INVERTEDINDEX
3. #include <string>
4. #include <vector>
5. #include <dirent.h>
6. #include <fstream>
7. #include <sstream>
8. #include <iostream>
9. #include "TireTree.cpp"
10. #include "Set.cpp"
11. #include "Heap.cpp"
12. **using** **namespace** std;
13. **struct** Document { //文档信息的结构化定义
14. **char** name[100];
15. **int** WordNumber;
16. };
17. **struct** TERM { //term信息的定义
18. **int** id, totDoc, totFrequency;
19. vector<**int**> docID;
20. vector<**int**> docFrequency;
21. vector<LinkedList> posinfo;
22. };
23. **void** out(ofstream &fout, TERM &term) { //输出term于指定文件，指定位置
24. fout.write((**char**\*)&term.id, **sizeof**(**int**));
25. fout.write((**char**\*)&term.totDoc, **sizeof**(**int**));
26. fout.write((**char**\*)&term.totFrequency, **sizeof**(**int**));
27. **for**(**int** i = 0; i < term.totDoc; i++) {
28. fout.write((**char**\*)&term.docID[i], **sizeof**(**int**));
29. fout.write((**char**\*)&term.docFrequency[i], **sizeof**(**int**));
30. Node \*p = term.posinfo[i].getBegin();
31. **while**(p != NULL) {
32. fout.write((**char**\*)&(p->num), **sizeof**(**int**));
33. p = p->next;
34. }
35. }
36. }
37. **void** in(ifstream &fin, TERM &term) { //输入term于指定文件，指定位置
38. fin.read((**char**\*)&term.id, **sizeof**(**int**));
39. fin.read((**char**\*)&term.totDoc, **sizeof**(**int**));
40. fin.read((**char**\*)&term.totFrequency, **sizeof**(**int**));
41. **for**(**int** i = 0; i < term.totDoc; i++) {
42. **int** docId, docFre, pos; LinkedList poslist;
43. poslist.init();
44. fin.read((**char**\*)&docId, **sizeof**(**int**));
45. term.docID.push\_back(docId);
46. fin.read((**char**\*)&docFre, **sizeof**(**int**));
47. term.docFrequency.push\_back(docFre);
48. **for**(**int** j = 0; j < docFre; j++) {
49. fin.read((**char**\*)&(pos), **sizeof**(**int**));
50. poslist.push\_back(pos);
51. }
52. term.posinfo.push\_back(poslist);
53. }
54. }
55. **class** InvertedIndex {
56. **private**:
57. string path = "newsDoc";
58. string TEMPpath = "TEMP";
59. **int** totDocsNumber = 0;
60. **double** aveBookWordsNumber = 0;
61. vector<Document> Books;
62. vector<**int**> WordPostionList;
63. TrieTree Words;
64. **private**: **void** initDocsandWords() { // 初始化文档信息和字典树
65. DIR \*dir;
66. **struct** dirent \*ptr;
67. **if** ((dir = opendir(path.c\_str())) == NULL) {
68. perror("Open dir error...");
69. **return**;
70. }
71. **while** ((ptr = readdir(dir)) != NULL) {
72. **if** (strcmp(ptr->d\_name, ".") == 0 || strcmp(ptr->d\_name, "..") == 0)    ///current dir OR parrent dir
73. **continue**;
74. string aNewBook = ptr->d\_name;
75. ifstream newfin;
76. newfin.open(path + "/" + aNewBook);
77. stringstream buffer;
78. buffer << newfin.rdbuf();
79. string str(buffer.str());
80. **int** thisDocWordNumber = 0;
81. **for**(**int** i = 0; i < str.size(); i++)
82. **if**(str[i] <= 'Z' && str[i] >= 'A')
83. str[i] = str[i] - 'A' + 'a';
84. **for**(**int** i = 0; i < str.size(); i++) {
85. **char** aNewWord[100];**int** top = 0;
86. **if**(str[i] >= 'a' && str[i] <= 'z') {
87. **int** j = i;
88. **for**(; j < str.size(); j++)
89. **if**(str[j] >= 'a' && str[j] <= 'z')
90. aNewWord[top++] = str[j];
91. **else**
92. **break**;
93. i = j;
94. aNewWord[top] = '\0';
95. top = 0;
96. thisDocWordNumber++;
97. string NewWord = aNewWord;
98. Words.insert(NewWord);
99. }
100. }
101. Document \*aBook;
102. aBook = (Document\*)malloc(**sizeof**(Document));
103. strcpy(aBook->name, aNewBook.c\_str());
104. aBook->WordNumber = thisDocWordNumber;
105. Books.push\_back(\*aBook);
106. totDocsNumber++;
107. }
108. closedir(dir);
109. ofstream fout("docIndex.data", ios::out | ios::binary);
110. **for**(**int** i = 0; i < totDocsNumber; i++)
111. aveBookWordsNumber += (**double**)Books[i].WordNumber/totDocsNumber;
112. fout.write((**char** \*)&totDocsNumber, **sizeof**(**int**));
113. fout.write((**char** \*)&aveBookWordsNumber, **sizeof**(**double**));
114. **for**(**int** i = 0; i < totDocsNumber; i++) {
115. fout.write((**char** \*)&Books[i], **sizeof**(Document));
116. }
117. fout.close();
118. **return** ;
119. }
120. **private**: **void** initEveryDoc() { //每个文档进行初始化
121. **for**(**int** nowDoc = 0; nowDoc < totDocsNumber; nowDoc++) {
122. **int** thisDocWordNumber = 0;
123. Set tempList;
124. ifstream newfin;
125. newfin.open(path + "/" + Books[nowDoc].name);
126. stringstream buffer;
127. buffer << newfin.rdbuf();
128. string str(buffer.str());
129. //cout << str;
130. **for**(**int** i = 0; i < str.size(); i++)
131. **if**(str[i] <= 'Z' && str[i] >= 'A')
132. str[i] = str[i] - 'A' + 'a';
133. **for**(**int** i = 0; i < str.size(); i++) {
134. string aNewWord;
135. **if**(str[i] >= 'a' && str[i] <= 'z') {
136. **int** j = i;
137. **for**(; j < str.size(); j++)
138. **if**(str[j] >= 'a' && str[j] <= 'z')
139. aNewWord.push\_back(str[j]);
140. **else**
141. **break**;
142. i = j;
143. **int** Id = Words.getId(aNewWord);
144. thisDocWordNumber++;
145. tempList.Insert(Id, thisDocWordNumber);
146. }
147. }
148. tempList.Search(TEMPpath + "/" + "0" + to\_string(nowDoc) + ".temp",nowDoc);
149. }
150. }
151. **private**: **void** mergeDoc(**int** startDoc, **int** k, **int** deep) { // 合并从startDoc开始的k个深度为deep-1的临时表，
152. ifstream fin[MaxK];
153. **for**(**int** i = 0; i < k; i++)
154. fin[i].open(TEMPpath + "/" + to\_string(deep-1) + to\_string(startDoc + i) + ".temp", ios::in | ios::binary);
155. ofstream fout;
156. fout.open(TEMPpath + "/" + to\_string(deep) + to\_string(startDoc/k) + ".tmp", ios::out | ios::binary);
157. **int** totTERMnum[MaxK], nowTERMnum[MaxK]; TERM nowTERM[MaxK];
158. Heap heap;
159. **for**(**int** i = 0; i < k; i++) {
160. fin[i].read((**char**\*)&totTERMnum[i], **sizeof**(**int**));
161. nowTERMnum[i] = 0;
162. **if**(nowTERMnum[i] + 1 <= totTERMnum[i]) {
163. fin[i].read((**char**\*)&nowTERM[i].id, **sizeof**(**int**));
164. fin[i].read((**char**\*)&nowTERM[i].totDoc, **sizeof**(**int**));
165. fin[i].read((**char**\*)&nowTERM[i].totFrequency, **sizeof**(**int**));
166. nowTERMnum[i] ++;
167. heap.push((MergeNode){nowTERM[i].id, i});
168. }
169. }
170. **int** PreMinTermId = 0;
171. TERM aNewTerm;
172. aNewTerm.id = 0;
173. aNewTerm.totDoc = 0;
174. aNewTerm.totFrequency = 0;
175. aNewTerm.docID.clear();
176. aNewTerm.docFrequency.clear();
177. aNewTerm.posinfo.clear();
178. **int** totDifferentWordNum = 0;
179. **while**(heap.Size()) {
180. MergeNode nowMinNode = heap.pop();
181. **if**(PreMinTermId == nowMinNode.termId) {
182. aNewTerm.totDoc += nowTERM[nowMinNode.thisDoc].totDoc;
183. aNewTerm.totFrequency += nowTERM[nowMinNode.thisDoc].totFrequency;
184. }
185. **else** **if**(PreMinTermId == 0) {
186. aNewTerm.id = nowMinNode.termId;
187. aNewTerm.totDoc += nowTERM[nowMinNode.thisDoc].totDoc;
188. aNewTerm.totFrequency += nowTERM[nowMinNode.thisDoc].totFrequency;
189. PreMinTermId = nowMinNode.termId;
190. }
191. **else** {
192. out(fout, aNewTerm);
193. totDifferentWordNum++;
194. aNewTerm.id = nowMinNode.termId;
195. aNewTerm.totDoc = nowTERM[nowMinNode.thisDoc].totDoc;
196. aNewTerm.totFrequency = nowTERM[nowMinNode.thisDoc].totFrequency;
197. aNewTerm.docID.clear();
198. aNewTerm.docFrequency.clear();
199. aNewTerm.posinfo.clear();
200. }
201. **for**(**int** i = 0; i < nowTERM[nowMinNode.thisDoc].totDoc; i++) {
202. **int** nowReadDocId, nowReadDocFre;
203. fin[nowMinNode.thisDoc].read((**char**\*)&nowReadDocId, **sizeof**(**int**));
204. fin[nowMinNode.thisDoc].read((**char**\*)&nowReadDocFre, **sizeof**(**int**));
205. aNewTerm.docID.push\_back(nowReadDocId);
206. aNewTerm.docFrequency.push\_back(nowReadDocFre);
207. LinkedList nowReadDocPosList;
208. nowReadDocPosList.init();
209. **for**(**int** j = 0; j < nowReadDocFre; j++) {
210. **int** pos;
211. fin[nowMinNode.thisDoc].read((**char**\*)&pos, **sizeof**(**int**));
212. nowReadDocPosList.push\_back(pos);
213. }
214. aNewTerm.posinfo.push\_back(nowReadDocPosList);
215. }
216. PreMinTermId = nowMinNode.termId;
217. **if**(nowTERMnum[nowMinNode.thisDoc] + 1 <= totTERMnum[nowMinNode.thisDoc]) {
218. nowTERMnum[nowMinNode.thisDoc] = nowTERMnum[nowMinNode.thisDoc] + 1;
219. fin[nowMinNode.thisDoc].read((**char**\*)&nowTERM[nowMinNode.thisDoc].id, **sizeof**(**int**));
220. fin[nowMinNode.thisDoc].read((**char**\*)&nowTERM[nowMinNode.thisDoc].totDoc, **sizeof**(**int**));
221. fin[nowMinNode.thisDoc].read((**char**\*)&nowTERM[nowMinNode.thisDoc].totFrequency, **sizeof**(**int**));
222. heap.push((MergeNode){nowTERM[nowMinNode.thisDoc].id, nowMinNode.thisDoc});
223. }
224. }
225. **if**(aNewTerm.id != 0) {
226. out(fout, aNewTerm);
227. totDifferentWordNum++;
228. }
229. fout.close();//关键
230. ifstream finCopy;
231. finCopy.open(TEMPpath + "/" + to\_string(deep) + to\_string(startDoc/k) + ".tmp", ios::in | ios::binary);
232. ofstream foutCopy;
233. foutCopy.open(TEMPpath + "/" + to\_string(deep) + to\_string(startDoc/k) + ".temp", ios::out | ios::binary);
234. foutCopy.write((**char**\*)&totDifferentWordNum, **sizeof**(**int**));
235. **for**(**int** i = 0; i < totDifferentWordNum; i++) {
236. TERM term;
237. in(finCopy, term);
238. out(foutCopy, term);
239. }
240. }
241. **private**: **int** Merge() { //合并文档，形成最终的倒排索引
242. **const** **int** K = 200;
243. **int** nowTotDocNumber = totDocsNumber, deep = 1;
244. **int** nextDeepTotDocNumber = 0;
245. **while**(nowTotDocNumber != 1) {
246. **for**(**int** i = 0; i < nowTotDocNumber; i+=K) {
247. mergeDoc(i, min(K, nowTotDocNumber - i), deep);
248. nextDeepTotDocNumber++;
249. }
250. nowTotDocNumber = nextDeepTotDocNumber;
251. nextDeepTotDocNumber = 0;
252. deep = deep + 1;
253. }
254. **return** deep - 1;
255. }
256. **private**: **void** outInfomation(**int** finalDeep) { //输出内存中全部信息
257. ofstream foutWordLish, foutDictionary;
258. Words.printWords("WordsIdList.data");
259. foutWordLish.open("WordPostionList.data", ios::out | ios::binary);
260. foutDictionary.open("dictionary.data", ios::out | ios::binary);
261. ifstream fin;
262. fin.open(TEMPpath + "/" + to\_string(finalDeep) + to\_string(0) + ".temp", ios::in | ios::binary);
263. **int** totTERMnum;
264. fin.read((**char**\*)&totTERMnum, **sizeof**(**int**));
265. **int** newtotTERMnum = totTERMnum + 1;
266. foutDictionary.write((**char**\*)&(newtotTERMnum), **sizeof**(**int**));
267. WordPostionList.push\_back(foutDictionary.tellp());
268. **int** zero = 0;
269. foutDictionary.write((**char**\*)&(zero), **sizeof**(**int**));
270. foutDictionary.write((**char**\*)&(zero), **sizeof**(**int**));
271. foutDictionary.write((**char**\*)&(zero), **sizeof**(**int**));
272. **for**(**int** i = 1; i <= totTERMnum; i++) {
273. TERM aNewTERM;
274. in(fin, aNewTERM);
275. WordPostionList.push\_back(foutDictionary.tellp());
276. out(foutDictionary, aNewTERM);
277. }
278. foutWordLish.write((**char**\*)&newtotTERMnum, **sizeof**(**int**));
279. **for**(**int** i = 0; i <= totTERMnum; i++)
280. foutWordLish.write((**char**\*)&WordPostionList[i], **sizeof**(**int**));
281. }
282. **void** ReadWordsIdList() { // 读入内存中应有的信息
283. ifstream fin1;
284. fin1.open("docIndex.data", std::ifstream::binary);
285. fin1.read((**char**\*)&totDocsNumber, **sizeof**(**int**));
286. fin1.read((**char**\*)&aveBookWordsNumber, **sizeof**(**double**));
287. **for**(**int** i = 0; i < totDocsNumber; i++) {
288. Document doc;
289. fin1.read((**char**\*)&doc, **sizeof**(Document));
290. Books.push\_back(doc);
291. }
292. fin1.close();
293. ifstream fin2;
294. fin2.open("WordsIdList.data", std::ifstream::binary);
295. **int** totWordNum = 0;
296. fin2.read((**char**\*)&totWordNum, **sizeof**(**int**));
297. Words.settotWordnum(totWordNum);
298. **for**(**int** i = 1; i <= totWordNum; i++) {
299. **char** word[30];**int** id;string Word;
300. fin2.read((**char**\*)word, 30\***sizeof**(**char**));
301. fin2.read((**char**\*)&id, **sizeof**(**int**));
302. Word = word;
303. Words.insert(Word, id);
304. }
305. **int** totTERMnum = 0;
306. ifstream fin3;
307. fin3.open("WordPostionList.data", ios::binary);
308. fin3.read((**char**\*)&totTERMnum, **sizeof**(**int**));
309. **for**(**int** i = 0; i < totTERMnum; i++) {
310. **int** dictionarypos = 0;
311. fin3.read((**char**\*)&dictionarypos, **sizeof**(**int**));
312. WordPostionList.push\_back(dictionarypos);
313. }
314. }
315. **public**: **void** Build() {
316. ifstream fin1;
317. fin1.open("WordsIdList.data", std::ofstream::binary);
318. **if**(fin1.is\_open()) {
319. ReadWordsIdList();
320. }
321. **else** {
322. initDocsandWords();//为每一个新闻编号
323. //构建字典树
324. initEveryDoc();
325. //收集所有的(termID, docID, freq)的元组
326. //本地对收集到的元组按照termID进行排序，
327. //然后把结果写回到本地磁盘文件文件
328. **int** finalDeep = Merge();//对第二步的结果进行归并排序(merge)，就可以创建一个简单的倒排索引
329. outInfomation(finalDeep);
330. }
331. }
332. **public**: **int** getWordId(string word) {
333. **return** Words.getId(word);
334. }
335. **public**: **int** gettotDocsNumber() {
336. **return** totDocsNumber;
337. }
338. **public**: **int** getReadPosion(**int** wordid) {
339. **return** WordPostionList[wordid];
340. }
341. **public**: **int** getBookWordsNumber(**int** bookid) {
342. **return** Books[bookid].WordNumber;
343. }
344. **public**: **double** getaveBookWordsNumber() {
345. **return** aveBookWordsNumber;
346. }
347. **public**: **char**\* getDocName(**int** docid) {
348. **return** Books[docid].name;
349. }
350. };
351. #endif

## 6.6 Search.cpp

1. #ifndef SEARCH
2. #define SEARCH
3. #include <cstdio>
4. #include <iostream>
5. #include <cstring>
6. #include <string>
7. #include <vector>
8. #include <cmath>
9. #include <algorithm>
10. #define MAXSEARCHWORDNUM 1000
11. #define MAXDOCXNUM 2000
12. #include "InvertedIndex.cpp"
13. **using** **namespace** std;
14. **class** VSMcross {//叉积结果
15. **public**:
16. **int** docId;
17. **double** crossans;
18. **void** operator = (**const** VSMcross &D) {
19. docId = D.docId;
20. crossans = D.crossans;
21. }
22. };
23. **struct** ExpressionNode {
24. **int** n; // 0 运算符 1 词 n>1 长度为n的短语
25. **char** operation;
26. LinkedList keyword;
27. LinkedList Set;
28. };
29. **class** Search {
30. **private**:
31. InvertedIndex a;
32. **double** k = 1.5;
33. **double** b = 0.75;
34. VSMcross tempstore[MAXDOCXNUM];
35. **int** precede[6][6] = {
36. {0  },
37. {0, 1, -1, -1, 1, 1},
38. {0, 1,  1, -1, 1, 1},
39. {0,-1, -1, -1, 0, 0},
40. {0, 1,  1,  0, 1, 1},
41. {0,-1, -1, -1, 0, 0},
42. } ;
43. **private**: **void** Sort(VSMcross \*a, **int** l, **int** r) {//将搜索结果按相关程度用归并排序排序
44. **if**(l >= r)
45. **return** ;
46. **int** mid = (l + r) >> 1;
47. Sort(a, l, mid);Sort(a, mid+1, r);
48. **int** lpos = l, rpos = mid+1, temp = l;
49. **while**(lpos <= mid && rpos <= r) {
50. **if**(a[lpos].crossans > a[rpos].crossans)
51. tempstore[temp++] = a[lpos++];
52. **else**
53. tempstore[temp++] = a[rpos++];
54. }
55. **while**(lpos <= mid)
56. tempstore[temp++] = a[lpos++];
57. **while**(rpos <= r)
58. tempstore[temp++] = a[rpos++];
59. **for**(**int** i = l; i <= r; i++)
60. a[i] = tempstore[i];
61. **return** ;
62. }
63. **private**: **void** searchVSM(string str) {//利用向量空间模型搜索
64. vector<**int**> keyword;
65. **int** len = 0;
66. **while**(str[len] != '\0')
67. len++;
68. **for**(**int** i = 0; i < len; i++)
69. **if**(str[i] <= 'Z' && str[i] >= 'A')
70. str[i] = str[i] - 'A' + 'a';
71. **for**(**int** i = 0; i < len; i++) {
72. string aNewWord;
73. **if**(str[i] >= 'a' && str[i] <= 'z') {
74. **int** j = i;
75. **for**(; j < len; j++)
76. **if**(str[j] >= 'a' && str[j] <= 'z')
77. aNewWord.push\_back(str[j]);
78. **else**
79. **break**;
80. i = j;
81. **if**(a.getWordId(aNewWord) != 0)
82. keyword.push\_back(a.getWordId(aNewWord));
83. }
84. }
85. VSMcross result[MAXDOCXNUM];
86. **for**(**int** i = 0; i < a.gettotDocsNumber(); i++) {
87. result[i].docId = i;
88. result[i].crossans = 0;
89. }
90. **for**(**int** i = 0; i < keyword.size(); i++) {
91. ifstream fin;TERM term;
92. fin.open("dictionary.data", ios::in | ios::binary);
93. fin.seekg(a.getReadPosion(keyword[i]), ios::beg);
94. in(fin, term);
95. //cout << term.id << " " << term.totDoc << " " << term.totFrequency << endl;
96. **for**(**int** j = 0; j < term.totDoc; j++) {
97. result[term.docID[j]].crossans +=
98. (term.docFrequency[j]\*(k+1)\*log((**double**)(a.gettotDocsNumber()+1) /term.totDoc)) /
99. (term.docFrequency[j] + (k\*(1-b+b\*a.getBookWordsNumber(term.docID[j])/a.getaveBookWordsNumber())));
100. }
101. }
102. Sort(result, 0, a.gettotDocsNumber()-1);
103. keyword.clear();
104. **int** totansnum = 0;
105. **for**(**int** i = 0; i < min(a.gettotDocsNumber(), 20); i++)
106. **if**(result[i].crossans > 0) {
107. cout << "\t" << a.getDocName(result[i].docId) << endl;
108. totansnum++;
109. }
110. **else**
111. **break**;
112. **if**(totansnum == 0)
113. printf("\tFind No Solution!\n");
114. }
115. **private**: **int** getOptId(**char** a) {
116. **if**(a=='|')      **return** 1;
117. **if**(a=='&')      **return** 2;
118. **if**(a=='(')      **return** 3;
119. **if**(a==')')      **return** 4;
120. **if**(a=='#')      **return** 5;
121. **else**            **return** 0;
122. }
123. **private**: **int** Precede(**char** a, **char** b) {
124. **int** idA = getOptId(a), idB = getOptId(b);
125. **return** precede[idA][idB];
126. }
127. **private**: LinkedList getDocSet(ExpressionNode expresionnnode) { //单词查询和词组查询
128. **int** n = expresionnnode.n;
129. **if**(n == 1) { //单词查询
130. LinkedList ans;
131. ans.init();
132. ifstream fin;TERM term;
133. fin.open("dictionary.data", ios::in | ios::binary);
134. fin.seekg(a.getReadPosion(expresionnnode.keyword.getBegin()->num), ios::beg);
135. in(fin, term);
136. **for**(**int** i = 0; i < term.totDoc; i++)
137. ans.push\_back(term.docID[i]);
138. **return** ans;
139. }
140. **else** { // 词组查询
141. LinkedList DocSet[MAXSEARCHWORDNUM], ans;
142. ans.init();
143. Node \*p = expresionnnode.keyword.getBegin();
144. **for**(**int** i = 0; i < n; i++) {
145. DocSet[i].init();
146. ifstream fin;TERM term;
147. fin.open("dictionary.data", ios::in | ios::binary);
148. fin.seekg(a.getReadPosion(p->num), ios::beg);
149. in(fin, term);
150. **for**(**int** j = 0; j < term.totDoc; j++)
151. DocSet[i].push\_back(term.docID[j]);
152. fin.close();
153. p = p->next;
154. }
155. **for**(**int** i = 1; i < n; i++)
156. DocSet[0].intersextion(DocSet[i]);
157. ifstream fin;
158. TERM term[MAXSEARCHWORDNUM];
159. p = expresionnnode.keyword.getBegin();
160. **for**(**int** i = 0; i < n; i++) {
161. fin.open("dictionary.data", ios::binary);
162. fin.seekg(a.getReadPosion(p->num), ios::beg);
163. in(fin, term[i]);
164. fin.close();
165. p = p->next;
166. }
167. p = DocSet[0].getBegin();
168. LinkedList poslist[MAXSEARCHWORDNUM];
169. Node \*posNode[MAXSEARCHWORDNUM];
170. **for**(**int** i = 0; i < n; i++)
171. poslist[i].init();
172. **while**(p != NULL) {
173. **int** nowDocId = p->num;
174. **for**(**int** i = 0; i < n; i++) {
175. poslist[i].Free();
176. poslist[i].init();
177. **int** L = 0, R = term[i].totDoc-1, mid, ans;
178. **while**(L <= R) {
179. mid = (L + R) >> 1;
180. **if**(term[i].docID[mid] == nowDocId) {
181. ans = mid;
182. **break**;
183. }
184. **else** **if**(term[i].docID[mid] < nowDocId)
185. L = mid + 1;
186. **else**
187. R = mid - 1;
188. }
189. Node \*q = term[i].posinfo[ans].getBegin();
190. **while**(q != NULL) {
191. poslist[i].push\_back(q->num);
192. q = q->next;
193. }
194. }
195. **for**(**int** i = 0; i < n; i++)
196. posNode[i] = poslist[i].getBegin();
197. **int** flag = 1;
198. **while**(posNode[0] != NULL) {
199. **for**(**int** i = 1; i < n; i++) {
200. **while**(posNode[i] && posNode[i-1] && (posNode[i]->num < posNode[i-1]->num + 1))
201. posNode[i] = posNode[i]->next;
202. }
203. flag = 1;
204. **for**(**int** i = 1; i < n; i++)
205. **if**(posNode[i] == NULL || (posNode[i]->num != posNode[i-1]->num + 1)) {
206. flag = 0;
207. **break**;
208. }
209. **if**(flag) {
210. ans.push\_back(nowDocId);
211. **break**;
212. }
213. **else**
214. posNode[0] = posNode[0]->next;
215. }
216. p = p->next;
217. }
218. **return** ans;
219. }
220. }
221. **private**: **void** searchBool(string str) { //布尔查询
222. ExpressionNode expression[MAXSEARCHWORDNUM];
223. **int** totExpressionNode = 0;
224. **int** len = 0;
225. **while**(str[len] != '\0')
226. len++;
227. str[len++] = ' ';
228. str[len++] = '\0';
229. **for**(**int** i = 0; i < len; i++)
230. **if**(str[i] <= 'Z' && str[i] >= 'A')
231. str[i] = str[i] - 'A' + 'a';
232. **for**(**int** i = 0; i < len; i++) {
233. string aNewWord;
234. **if**(str[i] == '&' || str[i] == '|' || str[i] == '(' || str[i] == ')') {
235. totExpressionNode++;
236. expression[totExpressionNode].n = 0;
237. expression[totExpressionNode].operation = str[i];
238. }
239. **if**(str[i] >= 'a' && str[i] <= 'z') {
240. totExpressionNode++;
241. expression[totExpressionNode].n = 0;
242. expression[totExpressionNode].keyword.init();
243. **int** j = i;**int** flag = 1;//指示现在是否是由单词到
244. **for**(; j < len; j++) {
245. **if**(str[j] >= 'a' && str[j] <= 'z') {
246. aNewWord.push\_back(str[j]);
247. flag = 1;
248. }
249. **else** **if**(flag) {
250. expression[totExpressionNode].n++;
251. expression[totExpressionNode].keyword.push\_back(a.getWordId(aNewWord));
252. aNewWord.clear();
253. flag = 0;
254. }
255. **if**(str[j] == '&' || str[j] == '|' || str[j] == '(' || str[j] == ')') {
256. j = j-1;
257. **break**;
258. }
259. }
260. i = j;
261. }
262. }
263. /\*for(int i = 1; i <= totExpressionNode; i++) {//(alibaba & work culture) | (COVID19 and its vaccine)
264. if(expression[i].n == 0) {
265. cout << "opt" << " " << expression[i].operation << endl;
266. }
267. else {
268. cout << expression[i].n << " ";
269. Node \*p = expression[i].keyword.getBegin();
270. while(p != NULL) {
271. cout << p->num << " ";
272. p = p->next;
273. }
274. cout << endl;
275. }
276. }\*/
277. **for**(**int** i = 1; i <= totExpressionNode; i++)
278. **if**(expression[i].n > 0)
279. expression[i].Set = getDocSet(expression[i]);
280. **char** OPTR[MAXSEARCHWORDNUM];**int** topOPTR = 0;OPTR[topOPTR++] = '#';
281. LinkedList OPND[MAXSEARCHWORDNUM];**int** topOPND = 0;
282. expression[totExpressionNode+1].n = 0;
283. expression[totExpressionNode+1].operation = '#';
284. **for**(**int** i = 2; i <= totExpressionNode; i++)
285. **if**(expression[i-1].n == 0 && expression[i].n == 0 &&
286. expression[i-1].operation != '(' && expression[i-1].operation != ')' &&
287. expression[i].operation != '(' && expression[i].operation != ')') {
288. cout << "\t INPUT WORNG!!!" << endl;
289. **return** ;
290. }
291. **for**(**int** i = 1; i <= totExpressionNode+1; i++) {
292. **if**(expression[i].n > 0) {
293. OPND[topOPND++] = expression[i].Set;
294. }
295. **else** {
296. **switch** (Precede(OPTR[topOPTR-1], expression[i].operation)) {
297. **case** -1:
298. OPTR[topOPTR++] = expression[i].operation;
299. **break**;
300. **case** 0:
301. topOPTR--;
302. **break**;
303. **case** 1:
304. **char** OP = OPTR[--topOPTR];
305. LinkedList b, a;
306. **if**(topOPND >= 1)    b = OPND[--topOPND];
307. **else**                b.init();
308. **if**(topOPND >= 1)    a = OPND[--topOPND];
309. **else**                a.init();
310. **if**(OP == '&') {
311. a.intersextion(b);
312. OPND[topOPND++] = a;
313. }
314. **else** **if**(OP == '|') {
315. a.merge(b);
316. OPND[topOPND++] = a;
317. }
318. i --;
319. **break**;
320. }
321. }
322. }
323. **if**(topOPTR != 1 && topOPND != 1)
324. cout << "\t INPUT WORNG!!!" << endl;
325. **else** {
326. **int** totansnum = 0;
327. Node \*p = OPND[topOPND-1].getBegin();
328. **while**(p != NULL) {
329. cout << "\t" << a.getDocName(p->num) << endl;
330. totansnum++;
331. p = p->next;
332. }
333. **if**(totansnum == 0)
334. printf("\tFind No Solution!\n");
335. }
336. }
337. **public**: **void** main() {
338. a.Build();
339. cout << "Hello!We Have Built A InvertedIndex And We Have Prepared for you!" << endl;
340. cout << "To Use Boolean Search, Please Use \"&\" \"|\" \"()\"!" << endl;
341. cout << "input EXIT to exit this program!" << endl;
342. cout << "Highly Recommend to Only Use English Characters and \"&\" \"|\" \"()\"! And Reduce Your Use of Particle!!!" << endl;
343. cout << "Created by Hou haofei 2021.12.12 in Wuhan" << endl;
344. **while**(1) {
345. **char** str[MAXSEARCHWORDNUM];**int** flag = 1, stopflag = 0;
346. gets(str);
347. **for**(**int** i = 0; str[i] != '\0'; i++)
348. **if**(str[i]=='&' || str[i]=='|' || str[i]=='(' || str[i]==')')
349. flag = 0;
350. **for**(**int** i = 0; str[i+3] != '\0'; i++)
351. **if**(str[i]=='E' && str[i+1]=='X' && str[i+2]=='I' && str[i+3]=='T')
352. stopflag = 1;
353. **if**(stopflag)
354. **break**;
355. **if**(flag)
356. searchVSM(str);
357. **else**
358. searchBool(str);
359. }
360. }
361. };
362. #endif

## 6.7 a.cpp

1. #ifndef MAIN
2. #define MAIN
3. #include <iostream>
4. #include "InvertedIndex.cpp"
5. #include "Search.cpp"
6. **using** **namespace** std;
8. **int** main () {
9. Search a;
10. a.main();
11. **return** 0;
12. }
13. #endif

1. 不能过度惩罚是因为一个文档很长可能有下面两个原因：

   使用了很多的词 ⇒ 更多的惩罚

   文本有很多内容 ⇒ 尽量少的惩罚 [↑](#footnote-ref-1)
2. 关于伸展树的时间复杂度分析较为复杂，详情可参阅《算法导论》。 [↑](#footnote-ref-2)
3. 经典台词”to be or not to be, that’s a question.”出自本选段 [↑](#footnote-ref-3)
4. 代码经过一版重构，其主要原因之一在于第一版同时开启文件句柄过多，导致程序无法正常运行。而对于更详细的分析，可见后文。 [↑](#footnote-ref-4)
5. 本项目支持开源项目：https://github.com/codelucas/newspaper [↑](#footnote-ref-5)
6. 布尔搜索的一个普遍问题就是采用 AND 操作符产生的结果正确率虽高但是召回率偏低，而采用 OR 操作符召回率高但是正确率低，很难或者说不可能找到一个令人满意的折衷方案。《信息检索导论》 [↑](#footnote-ref-6)