

《数据结构》上机报告

2022 年 11 月 19 日

姓名：郑博远 学号：2154312 班级：计科 1 班 得分：_____

实验题目	查找的应用(字典树)	
实验目的	1. 理解字典树的概念； 2. 掌握字典树的存储结构和基本操作；	
问题描述	给一篇超过 100000 字的英文文章，你能统计出里面每一个单词出现的次数吗？ 相信聪明的你，一定不屑于使用 <code>std::map</code> 这种走捷径的手段，并想到使用字典树可以很好地解决这个问题。	
基本要求	输入： 1. 一个纯 ASCII 字符组成的文本文件，统计里面每个单词出现的次数； 2. 单词仅由大、小写英文字母组成，处理时将所有大写字母转换为小写字母； 3. 当出现连接符 '-' 连接两个单词时，如果 '-' 后跟的是换行符，可以视为 '-' 前后是一个因换行而被拆开的单词（即删除 '-' 并将 '-' 前后连接起来成为一个单词），否则视为两个单词； 4. 使用字典树作为基本数据结构。 输出： 1. 输出为若干形式为<key,value>的键值对，其中键为文本中出现的单词，值为这个单词出现的次数； 2. 每一行输出一个键值对，输出按键的字典序升序排序。	
选做要求		
	已完成选做内容（序号）	

<p>数据结构设计</p>	<p>本次实验涉及到的数据结构为字典树，具体实现如下：</p> <pre> /* Trie的结点 */ struct node{ //以当前节点为结束的单词出现次数 int times; //单词的下一个字符 当前结点的子结点 node* nextNode[26]; }; /* 维护的前top_num频次数组元素 */ struct info { //字符串内容 string str; //出现次数 int times; }; /* Trie的类 */ class Trie { private: //字典树的根节点 node* root; //存放文件内容 string file_content; //统计前n的词频（目前top_num） const unsigned int top_num = 10; //存放前top_num词频的词 vector<info> top_frequency; }; </pre>
<p>功能(函数)说明</p>	<pre> /** * @brief 创建空的字典树 */ Trie::Trie() /** * @brief 销毁字典树(调用Delete_dfs) */ Trie::~~Trie() /** * @brief 从文件读入单词构建字典树 </pre>

```

    * @param filename 读入的文件名
    */
    bool Trie::ReadFromFile(const char* filename)

    /**
    * @brief 递归删除所有结点(析构函数调用)
    * @param cur 当前遍历的结点
    */
    void Trie::Delete_dfs(node* cur)

    /**
    * @brief 在字典树中插入新单词
    * @param cur 当前的结点位置
    * @param word 新插入的单词
    * @return 该单词目前的出现频次
    */
    int Trie::InsertWord(node* cur, string word)

    /**
    * @brief 深度优先搜索递归遍历
    * @param cur 当前搜索的结点
    * @param q 存储字符串的队列
    * @param out 输出流
    */
    void Trie::Traverse_dfs(node* cur, queue <char> q,
    ostream& out)

    /**
    * @brief 遍历字典树输出所有单词及频数
    * @param out 输出流
    */
    void Trie::Traverse(ostream& out)

    /**
    * @brief 维护前top_num个出现频次最多的字符串
    * @param str 当前新字符串
    * @param times 新字符串出现频数
    */
    void Trie::MaintainTop(string str, int times)

    /**
    * @brief 打印出现频次为前top_num的单词
    * @param out 输出流

```

	<pre>*/ void Trie::PrintTop(ostream& out)</pre>
界面设计和使用说明	<p>本次的程序界面为命令行形式。进入程序后，若命令行方式传入的参数错误，则程序给出功能提示如下方图 1 所示。</p>  <p>图 1 程序的功能提示</p> <p>程序包含两个功能，“--count”模式按照字典序输出统计的每个单词词频；“--top10”模式输出按词频从高到低排序的词频前 10 单词（不满前 10 则输出前 n 个）。</p>  <p>图 2 测试程序对于连字符的处理</p> <p>选择“count”模式，并输入文件名从文件中读入单词。上方图 2 展示了程序对连字符的处理方式：当出现连接符“-”连接两个单词时，如果“-”后跟的是换行符，可以视为“-”前后是一个因换行而被拆开的单词（即删除“-”并将“-”前后连接起来成为一个单词），否则视为两个单词。根据题目要求，程序将所有</p>

单词中的大写字母统一转为小写字母进行统计。

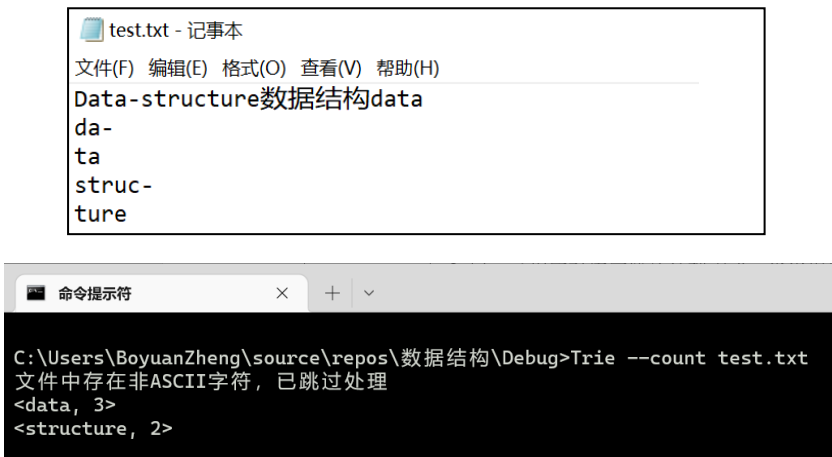


图 3 测试程序对非 ASCII 字符的处理

若程序中出现非 ASCII 字符，程序将给出错误提示，同时在统计词频时跳过非 ASCII 的字符，照常统计其他的单词。如上方图 3 所示，在文件中加入中文字符“数据结构”，则程序给出提示“文件中存在非 ASCII 字符，已跳过处理”。

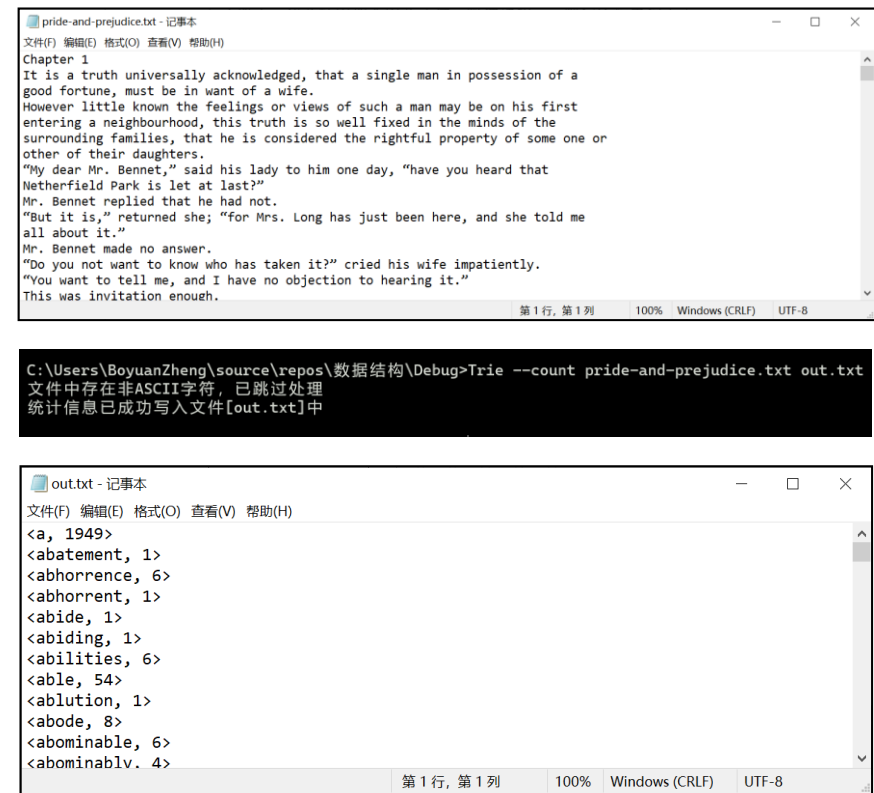
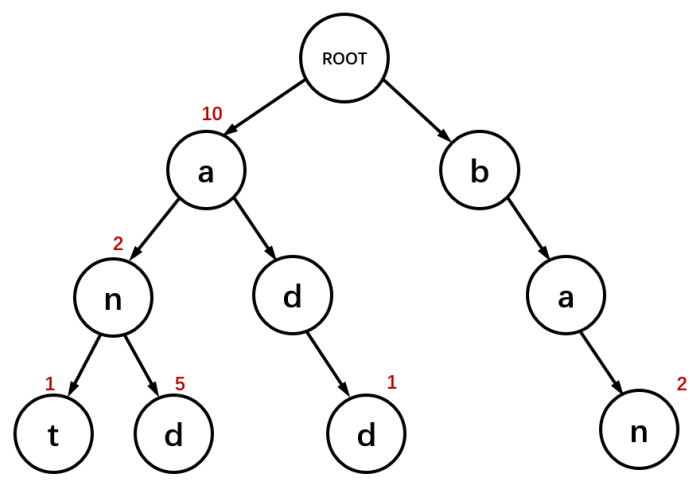


图 4 测试程序对大文件处理以及输出到文件

	<p>下面测试程序对超过 100000 词的文本的处理。如上方图 4 所示：使用《傲慢与偏见》的英文文本作为输入，词数超过十二万；将统计词频输入到 out.txt 文件中。程序正确读入后给出统计成功的信息，并将结果成功写入文件。</p> <p>选择“top10”模式，程序读入文件后能够统计输出出现词频前 10 的单词，按照频数从大到小排列。如上方图 5 中所示，《傲慢与偏见》一书中最高词频的单词分别为“the”（4331 次）、“to”（4163 次）、“of”（3611 次）等。</p>  <p>图 5 统计文本中出现频数前 10 的单词</p> <p>下面测试程序对于输入错误的处理。若输入的文件不存在，则程序能够给出错误提示如下：</p>  <p>图 6 输入文件不存在的错误提示</p>
调试分析	<p>1. 将每个单词出现的次数记录在单词的最后一个字母的结点的 times 变量上。在插入新单词时，若其本身不存在原先的字典树中，则需要依次建立不存在的旧结点，否则则在原有的单词末尾结点让 times 变量自增。在建立旧结点时，我先是忘记在单词末尾结点的 times 应该是 1，而在建立结点时统一 times 赋为 0，导致单词统计频数统一少 1；后来误改为将所有新建的结点 times 均赋值为 1，导致路径上不是完整单词的结点也被错误计数。修</p>

	<p>改后，让新建的结点中只有单词末尾自增即可；</p> <p>2. 遍历字典树进行查找时，我通过队列的方式将遍历过的结点进行存储，利用队列先进先出的特性来打印单词。但打印时要将整个队列不断 pop 弹出直至队列 empty，这会导致之后的单词缺少前缀。因此需要复制构造一个新队列再进行打印，使得每一个相同前缀的单词都能够被完整显示。</p>
心得体会	<p>本次的词频统计问题是对字典树这一数据结构的基础应用。字典树的典型应用是用于统计和排序大量的字符串（如本题中统计每个单词的出现次数），经常被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。其核心思想是空间换时间，利用字符串的公共前缀来降低查询时间开销以提高效率。</p> <p>字典树包含如下三个性质：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。</p> <div data-bbox="542 1332 1236 1814"></div> <p>图 7 字典树存储方式示意图</p> <p>在本题中，我将每个单词的出现次数记录在字典树的最后一</p>

个结点上。如上方图 7 所示，以“a”、“n”、“t”这一条路径为例，单词“a”出现 10 次，“an”出现 2 次，“ant”出现 1 次。由于题目中只涉及到小写英文字母的存储，因此每个结点直接用一个大小 26 的指针数组存放后继的字符，数组下标+小写字母 a 的 ASCII 码即对应存储的字母。

对于字典树的建立，若文件长为 k ，则易得时间复杂度为 $O(k)$ 。而查找某个字符串的时间复杂度则取决于串长，即若字符串长度为 1，则查找的时间复杂度为 $O(1)$ 。字典树的建立与查找时间复杂度都很低，这得益于字典树以空间换时间的思想。

与此同时，我也在本次作业中实践了研讨课上统计前 10 频率字符串的研讨题，额外在程序中加入了统计前 10 单词频数的功能。我在建立字典树时始终维护一个长度为 10（或者未满时小于 10）的数组，存放出现词频最高的前 10 个单词。每次新在字典树插入单词时，都比较是否需要更新前 10 单词列表。由于 10 个的表长较小且固定，可以视作时间复杂度为 $O(1)$ 。最终输出时，也能够直接遍历数组输出，而无需再在树中进行搜索。

附.完整代码

1.字典树的头文件(Trie.h)

```
#pragma once
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <queue>
using namespace std;

/* Trie的结点 */
struct node{
    //以当前节点为结束的单词出现次数
    int times;
    //单词的下一个字符 当前结点的子结点
    node* nextNode[26];
public:
    //建立新的字典树结点
    node();
};

/* 维护的前top_num频次数组元素 */
struct info {
    //字符串内容
    string str;
    //出现次数
    int times;
};

/* Trie的类 */
class Trie {
private:
    //字典树的根节点
    node* root;
    //存放文件内容
    string file_content;
    //统计前n的词频（目前top_num）
    const unsigned int top_num = 10;
    //存放前top_num词频的词
```

```

vector<info> top_frequency;
//递归释放所有节点
void Delete_dfs(node* cur);
//递归遍历字典树输出所有单词及频数
void Traverse_dfs(node* cur, queue<char> q, ostream& out);
//维护前top_num的数组
void MaintainTop(string str, int times);
public:
    //构造空字典树
    Trie();
    //销毁字典树
    ~Trie();
    //从文件读入单词构造字典树
    bool ReadFromFile(const char* filename);
    //在字典树插入新单词
    int InsertWord(node* cur, string word);
    //遍历字典树输出所有单词及频数
    void Traverse(ostream& out);
    //打印前top_num个出现频次的字符
    void PrintTop(ostream& out);
};

```

2.字典树的源文件(Trie.cpp)

```

#include "../Trie.h"
#include <iostream>
#include <fstream>
#include <queue>
#include <cstring>
using namespace std;

/**
 * @brief 创建新的字典树结点
 */
node::node()
{
    times = 0;
    memset(nextNode, 0, sizeof(nextNode));
}

/**

```

```

    * @brief 创建空的字典树
*/
Trie::Trie()
{
    root = new(nothrow) node;
    if (!root)
        exit(-1);
}

/**
 * @brief 递归删除所有结点(析构函数调用 )
 * @param cur 当前遍历的结点
*/
void Trie::Delete_dfs(node* cur)
{
    for (int i = 0; i < 26; i++)
        if (cur->nextNode[i])
            Delete_dfs(cur->nextNode[i]);
    delete cur;
}

/**
 * @brief 销毁字典树(调用Delete_dfs)
*/
Trie::~Trie()
{
    Delete_dfs(root);
}

/**
 * @brief 在字典树中插入新单词
 * @param cur 当前的结点位置
 * @param word 新插入的单词
 * @return 该单词目前的出现频次
*/
int Trie::InsertWord(node* cur, string word)
{
    int index = word[0] - 'a';
    // 如果没有该节点就新建
    if (cur->nextNode[index] == NULL) {
        cur->nextNode[index] = new(nothrow) node;
        if (!cur->nextNode[index])
            exit(-1);
    }
}

```

```

    }
    //单词结束了
    if (word.size() == 1)
        return (++cur->nextNode[index]->times);
    else {
        word.replace(0, 1, "");
        return InsertWord(cur->nextNode[index], word);
    }
}

/**
 * @brief 深度优先搜索递归遍历
 * @param cur 当前搜索的结点
 * @param q 存储字符串的队列
 * @param out 输出流
 */
void Trie::Traverse_dfs(node* cur, queue<char> q, ostream& out)
{
    queue<char> q1(q), q2;

    if (cur->times > 0) {
        out << "<";
        while (!q1.empty()) {
            out << q1.front();
            q1.pop();
        }
        out << ", " << cur->times << ">" << endl;
    }

    for (int i = 0; i < 26; i++) {
        if (!cur->nextNode[i])
            continue;
        q2 = q;
        q2.push(i + 'a');
        Traverse_dfs(cur->nextNode[i], q2, out);
    }
}

/**
 * @brief 遍历字典树输出所有单词及频数
 */
void Trie::Traverse(ostream& out)
{

```

```

    queue<char> q;
    Traverse_dfs(root, q, out);
}

/**
 * @brief 维护前top_num个出现频次最多的字符串
 * @param str 当前新字符串
 * @param times 新字符串出现频数
 */
void Trie::MaintainTop(string str, int times)
{
    unsigned int i = top_frequency.size();
    while (i > 0 && times >= top_frequency[i - 1].times){
        if (top_frequency[i - 1].str == str) {
            top_frequency.erase(top_frequency.begin() + i - 1);
        }
        i--;
    }
    top_frequency.insert(top_frequency.begin() + i, info{str, times});
    if (top_frequency.size() > top_num)
        top_frequency.pop_back();
}

/**
 * @brief 打印出现频次为前top_num的单词
 * @param out 输出流
 */
void Trie::PrintTop(ostream& out)
{
    for (unsigned int i = 0; i < top_frequency.size(); i++) {
        out << top_frequency[i].str << "\t" << top_frequency[i].times
        << endl;
    }
}

/**
 * @brief 从文件读入单词构建字典树
 * @param filename 读入的文件名
 */
bool Trie::ReadFromFile(const char* filename)
{
    ifstream infile(filename, ios::in);
    stringstream filestream;

```

```

if (!infile.is_open()) {
    cout << "文件[" << filename << "]不存在!" << endl;
    return false;
}

filestream << infile.rdbuf();
infile.close();
file_content = filestream.str();

//处理换行连字符
while (file_content.find("-\n") != string::npos) {
    file_content.replace(file_content.find("-\n"), strlen("-\n"),
    "");
}

//将剩余连字符转空格
while (file_content.find("-") != string::npos) {
    file_content.replace(file_content.find("-"), strlen("-"), "
");
}

bool invalid = false;
//大写转小写
for (unsigned int i = 0; i < file_content.size(); i++) {
    if (file_content[i] >= 'a' && file_content[i] <= 'z')
        continue;
    else if (file_content[i] >= 'A' && file_content[i] <= 'Z')
        file_content[i] += ('a' - 'A');
    else {
        if (file_content[i] < 0)
            invalid = true;
        file_content[i] = ' ';
    }
}

if(invalid)
    clog << "文件中存在非ASCII字符，已跳过处理" << endl;

/* 文件读入部分结束 以下开始建树 */

filestream.clear();
filestream.str("");
filestream << file_content;

```

```

while (filestream.good()) {
    string newword;
    filestream >> newword;
    if (newword == "")
        break;
    MaintainTop(newword, InsertWord(root, newword));
}

return true;
}

```

3.主程序的源文件(main.cpp)

```

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
#include <iomanip>
#include <stack>
#include "../Trie.h"
using namespace std;

/* void usage(const char* const procname, const int args_num)
 * @brief 打印函数提示信息
 * @param procname 可执行文件名称
 * @param args_num 参数个数
 */
void usage(const char* const procname, const int args_num)
{
    const int wkey = 7 + strlen(procname) + 1;
    const int wopt = 7 + strlen(procname) + 4;

    cout << endl;
    cout << "Usage: " << procname << " { --count filename1 (filename2) }"
    << endl;
    cout << setw(wkey) << ' ' << "{ --top10 filename1 (filename2) }" <<
    endl;
    cout << endl;

    cout << setw(wkey) << ' ' << "必选项: 指定程序功能(二选一)" << endl;
    cout << setw(wopt) << ' ' << "--count : 统计词频 [输入文件] [输出文

```

```

件(未指定则为屏幕)]" << endl;
    cout << setw(wopt) << ' ' << "--top10 : 输出出现频率最高的前top_num
单词 [输入文件] [输出文件(未指定则为屏幕)]" << endl;
    cout << endl;
}

int main(int argc, char** argv)
{
    Trie trie;
    ofstream outfile;
    if ((argc == 3 || argc == 4) && (strcmp(argv[1], "--count") == 0 ||
strcmp(argv[1], "--top10") == 0)) {
        if (strcmp(argv[1], "--count") == 0){
            if (!trie.ReadFromFile(argv[2]))
                return -1;
            switch (argc) {
                case 3:
                    trie.Traverse(cout);
                    break;
                case 4:
                    outfile.open(argv[3], ios::out);
                    if (!outfile.is_open()) {
                        cout << "文件[" << argv[3] << "]无法写入!" << endl;
                        return -1;
                    }
                    trie.Traverse(outfile);
                    cout << "统计信息已成功写入文件[" << argv[3] << "]中" <<
endl;

                    outfile.close();
                    break;
            }
        }
        else if (strcmp(argv[1], "--top10") == 0) {
            if (!trie.ReadFromFile(argv[2]))
                return -1;
            switch (argc) {
                case 3:
                    trie.PrintTop(cout);
                    break;
                case 4:
                    outfile.open(argv[3], ios::out);
                    if (!outfile.is_open()) {
                        cout << "文件[" << argv[3] << "]无法写入!" << endl;

```



```
        return -1;
    }
    trie.PrintTop(outfile);
    cout << "统计信息已成功写入文件[" << argv[3] << "]中" <<
endl;

    outfile.close();
    break;
    }
}
else
    usage(argv[0], argc);

return 0;
}
```