一、问题分析

1.1 处理的对象（数据）

处理的数据是英文文本文件的单词。英文文本文件中除了英文字母和空格，还可能有标点符号等，除英语外的字符均视为分隔符，只统计单词频率。

1.2 实现的功能

最终功能：英文文本文件的词频统计。

具体功能：

1) 分词：以非英文字符作为分隔符,且将大写转化为小写
2) 存储：使用 Trie 字典树储存
3) 建堆：以单词频率排序
4) 输出：输出堆前 100 单词

1.3 结果显示

输出频率前 100 的单词

1.4 样例求解过程

样例输入：

I will give you some advice about life.
Eat more roughage;
Do more than others expect you to do and do it pains;
Remember what life tells you;
do not take to heart every thing you hear.
do not spend all that you have.
do not sleep as long as you want.

样例输出：

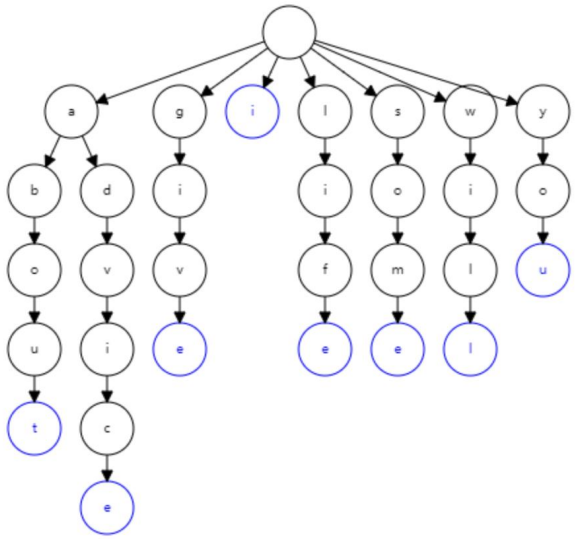
do 6
you 6
not 3
as 2
life 2
more 2
to 2
about 1
advice 1
all 1
and 1
eat 1
every 1
expect 1
give 1
have 1
hear 1
heart 1
i 1
it 1

long 1
others 1
pains 1
remember 1
roughage 1
sleep 1
some 1
spend 1
take 1
tells 1
than 1
that 1
thing 1
want 1
what 1
will 1

求解过程：

1）逐个字符扫描以分词和转化为小写，如第一行可分为
i,will,give,you,some,advice,about,life 其中忽略了’.’，修改了 I

2）将每个单词存入 Trie，无对应位置字符则新建，字符末尾加标记。第一行示例如图所示。



3）中序遍历整个 Trie，将有单词标记结尾的放入优先队列中

4）输出优先队列前 100 个单词，由于不满 100 个，所以全部输出。

二、数据结构和算法设计

题目要求使用 Trie。Trie 是一棵多叉树，采用子节点为指针链表的动态结点表示法。为了方便查找访问，链表改为 STL 中的 map。

1.抽象数据类型设计

```
class Trie{
    private:
        Node *root;
        priority_queue<...>frequency;//优先队列/堆存储单词频率，自定义规则
    public:
        Trie();
        ~Trie();
        void saveWord(string s);//存储单词
        void traverse();//遍历建堆
        void printHeap();//输出堆
};
//ADT 只显示功能接口和重要数据
```

2.物理数据类型设计

```
// 定义 Trie 树的节点结构
struct Node {
    char alpha=0; // 节点存储的字符
    bool isEnd=false; // 是否为单词结尾
    int time=0; // 单词出现的次数
    map<char,Node*> next; // 子节点
    Node(){}
    Node(char c):alpha(c){}
};
// 自定义比较函数，用于优先队列
class compare{
    public:
    bool operator() (const pair<int,string>&x,const pair<int,string>&y){
        if(x.first==y.first)
            return x.second>y.second;
        else return x.first<y.first;
    }
};
// Trie 树类
class Trie {
    private:
        Node *root; // 根节点
        priority_queue<pair<int,string>,vector<pair<int,string>>,compare>
frequency; // 优先队列，用于存储单词及其出现次数

        // 递归遍历 Trie 树，将单词及其出现次数存入优先队列
        void traverseHelp(Node* curr,string path);
        // 递归删除 Trie 树的所有节点
        void deleteTrie(Node*curr);
```

```
    public:
        Trie() {
            root=new Node();
        }
        ~Trie(){
            deleteTrie(root);
        }

        // 保存单词到Trie树中
        void saveWord(string s);

        // 遍历Trie树，将单词及其出现次数存入优先队列
        void traverse() {
            traverseHelp(root,"");
        }

        // 打印优先队列中的前100个单词及其出现次数
        void printHeap();
};
```

3.算法思想的设计

1) Trie树采用子节点为指针链表的动态结点表示法。
2) Trie树的建立/单词的储存为查找后直接迭代的方法。
3) Trie树的遍历建堆为前序遍历、深度优先搜索。
4) Trie树的删除为后续遍历。
5) 排序单词的频率用优先队列，自定义排序方式，效率高。

4.关键功能的算法步骤

1) 保存单词void saveWord(string s);函数：

取根节点为当前结点,然后不断在子节点中查找是否有字符串中当前字符,没有则新建。然后当前结点指向下一结点。单词结束则加上结束标志和次数加一。

2) Trie树的遍历建堆void traverseHelp(Node* curr,string path);函数

接口是traverseHelp,但由于涉及递归深搜和root的访问,所以真正实现设为private。

采用前序遍历的递归方式。访问方式为若为非根节点则将当前字符加入路径，若有单词结尾标志则将频率和单词加入堆。

三、算法性能分析

1.时间复杂度

读取文件和分词为O(n),n为字符数量。
单个单词查找/存储为O(w),w为单词长度，因此所有单词存储和遍历为O(n)。
堆时间复杂度为O(mlogm),m为不重复的单词数量。
因此时间复杂度为O(n+mlogm)。通常来说n比mlogm大得多，因为英文小说中没有无意义的单词，且很多单词会重复。
综上，时间复杂度为O(n)。

2.空间复杂度

空间复杂度取决于每个节点子节点的个数，以及单词的最大长度。最坏情况下 O(S^L),其中 S 为子节点的个数，L 为单词的最大长度。

3.总结

时间复杂度为 O(n)

空间复杂度 O(S^L)

4.不足之处的思考

第一次提交代码时出现了第一个测试数据运行时间过长。根据优化提示，可以使用多线程的并行化计算，使得运行速度更快。因此本人进行了自学后尝试。根据四核机器，可以同时进行 8 个线程。实际上出现了若读文件多线程进行，会导致单词储存出现问题。如果只存单词时多线程进行，可能需要涉及线程池，C++没有内部支持需要自行书写，超出了我目前的知识理解。试图从其他方面优化算法前重新提交了一遍之前未修改的代码居然通过了，不明是何环境因素影响。因此线程方面的知识和实现目前欠缺。