实验报告：基于Trie树的英文词频统计

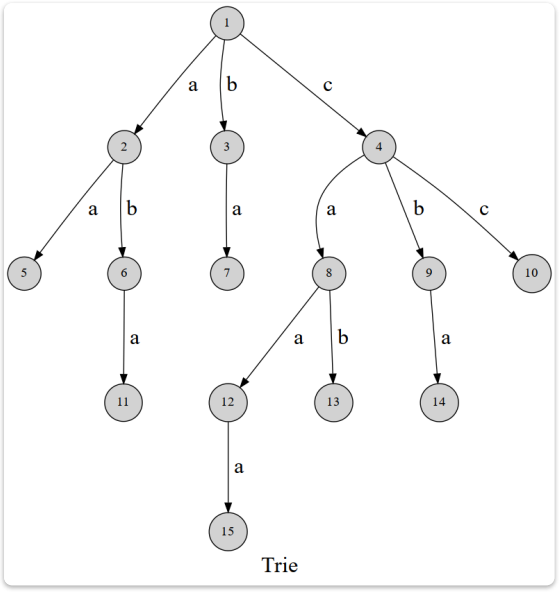
一、实验目的

1. 掌握Trie树（字典树）的基本原理与实现方法。
2. 实现英文文本中单词的词频统计，并按要求排序输出。
3. 学习文件读写、字符串处理及高效数据结构的设计。

二、实验原理

Trie树是一种多叉树结构，用于高效存储和检索字符串集合。其核心特点为：

- 每个节点表示一个字符，从根到节点的路径构成一个单词。
- 子节点通过字符索引快速定位（如`a-z`对应下标`0-25`）。
- 叶子节点（或标记节点）记录单词出现次数。



可以发现，这棵字典树用边来代表字母，而从根结点到树上某一结点的路径就代表了一个字符串。举个例子，
$1 \to 2 \to 4 \to 8 \to 12$表示的就是字符串`caa`

Trie树的应用：

1. 字符串检索：
   从根节点开始一个一个字符进行比较，如果发现不同的字符，表示该字符串在集合中不存在
2. 词频统计：（如本实验）
   Trie树常被搜索引擎系统用于文本词频统计。为了实现词频统计，我们修改了节点结构，用一个整型变量`count`来计数。对每一个关键字执行插入操作，若已存在，计数加1，若不存在，插入后`count`置1。
3. 字符串排序：
   Trie树可以对大量字符串按字典序进行排序，思路也很简单：遍历一次所有关键字，将它们全部插入trie树，树的每个结点的所有儿子很显然地按照字母表排序，然后先序遍历输出Trie树中所有关键字即可。
4. 前缀匹配：
   - 例如：找出一个字符串集合中所有以`ab`开头的字符串。我们只需要用所有字符串构造一个trie树，然后输出以`a->b->`开头的路径上的关键字即可。
   - trie树前缀匹配常用于搜索提示。如当输入一个网址，可以自动搜索出可能的选择。当没有完全匹配的搜索结果，可以返回前缀最相似的可能。

词频统计流程：

1. 读取文件内容，按非字母字符分割单词。
2. 将单词统一为小写后插入Trie树，统计出现次数。
3. 遍历Trie树收集所有单词及其频率。
4. 按频率降序（同频按字典序升序）排序后输出结果。

三、具体问题

一、处理对象

1. **输入对象**：当前目录下的文本文件`in.txt`，内容为英文文本。
2. **单词定义**：仅由字母组成，非字母字符（如标点、数字、空格）作为分隔符。单词需统一转换为小写形式。
3. **输出对象**：前100个高频单词及其出现次数（不足则全输出），按频率降序排列，同频按字典序升序。

二、功能实现

1. **文件读取**：逐字符读取文件内容。
2. **单词分割**：通过非字母字符分割连续字母序列为单词。
3. **大小写转换**：将所有字母转换为小写。
4. **词频统计**：使用Trie树存储单词及其出现次数。
5. **数据收集**：遍历Trie树，收集所有单词及其频率。
6. **排序与输出**：按频率降序和字典序升序排序后输出结果。

三、结果显示

- **输出格式**：每行一个单词及其频率，形如`word count`。
- **输出限制**：最多输出前100个高频单词。
- **输入样例**：

```
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.
```

- **输出结果**：（将词频最高的100个词和出现次数输出）

```
    do 6
    you 6
    not 3
    as 2
    life 2
    more 2
    to 2
    about 1
    advice 1
    all 1
    ...
```

四、ADT（抽象数据类型）

1. **Trie树节点（`TrieNode`）**：
   - **数据成员**：
     - `children[26]`：指向子节点的指针数组（对应字母`a-z`）。
     - `count`：记录当前节点是否为单词结尾及其出现次数。
   - **操作**：初始化、析构（递归释放内存）。
2. **Trie树（`Trie`）**：
   - **数据成员**：
     - `root`：根节点指针。
   - **操作**：
     - `insert(word)`：插入单词并更新计数。
     - `collect()`：深度优先遍历收集所有单词及其频率。

五、算法思想

1. **Trie树插入**：
   - 从根节点开始，逐字符向下构建路径。
   - 每个字符对应一个子节点索引（`a-z`映射为`0-25`）。
   - 单词结束时，对应节点的`count`自增。
2. **数据收集**：
   - 深度优先遍历（DFS）Trie树，递归记录路径上的字符组合。
   - 遇到`count > 0`的节点时，保存当前路径为单词及其频率。
3. **排序算法**：
   - 使用快速排序（`std::sort`），自定义比较函数：
     - 频率降序。
     - 同频率时，按字典序升序（如`do`在`you`之前）。

六、具体代码实现

Trie节点的具体实现

```
struct TrieNode {
    TrieNode* children[26];//可能会有26个字母数量的子树
    int count;//这里用count来代表从根节点到该节点的单词出现的次数
    TrieNode() : count(0) {
        for (int i = 0; i < 26; ++i) children[i] = nullptr;
    }
    ~TrieNode() {
        for (int i = 0; i < 26; ++i) {
            if (children[i]) delete children[i];
        }
    }
};
```

DFS

```
// 深度优先遍历收集单词
```

```
    void dfs(TrieNode* node, string current_word) {
        if (node->count > 0) {
            words.emplace_back(current_word, node->count);
        }
        for (int i = 0; i < 26; ++i) {
            if (node->children[i]) {
                dfs(node->children[i], current_word + char('a' + i));
            }
        }
    }
```

单词的插入

```
void insert(const string& word) {
        TrieNode* node = root;
        for (char c : word) {
            int index = tolower(c) - 'a';
            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }
            node = node->children[index];
        }
        node->count++;
    }
```

字符串分割为单词并插入到Trie树

```
 while (file.get(c)) {
        if (isalpha(c)) {
            current_word += tolower(c);
        } else if (!current_word.empty()) {
            trie.insert(current_word);
            current_word.clear();
        }
    }
```

七、算法性能分析

1. **时间复杂度**：
   - **插入阶段**：每个单词长度为L，插入时间复杂度为$O(L)$。总共有N个单词，插入总时间为$O(N \cdot L)$。
   - **收集阶段**：遍历Trie树的时间复杂度为$O(M)$，其中MM为Trie节点总数（通常接近$N \cdot L$）。
   - **排序阶段**：快速排序时间复杂度为$O(NlogN)$。
   - **总时间复杂度**：$O(N \cdot L + NlogN)$。
2. **空间复杂度**
   - **Trie树**：每个节点最多有26个子节点，空间复杂度为$O(26L)$（理论最坏情况，实际接近线性）。
   - **单词列表**：存储所有单词及其频率，空间复杂度为$O(N)$。
3. **优化方向**：
   - **多线程处理**：将文件分块读取，并行构建多个Trie子树，最后合并结果。
   - **内存管理**：使用智能指针（如`unique_ptr`）避免手动释放内存的复杂性。

八、实验总结

通过本实验，掌握了Trie树在词频统计中的应用，熟悉了文件处理与排序算法。实验代码时间复杂度为$O(N \cdot L)$（N为单词数，L为平均长度），空间复杂度为$O(M \cdot 26)$（M为节点总数）。未来可进一步探索并行化优化以提升大规模文本处理效率。