数据结构课程设计

项目说明文档

关键字检索系统

|  |  |
| --- | --- |
| 作者姓名： | 杨烜赫 |
| 学 号： | 2252709 |
| 指导教师： | 张 颖 |
| 学院专业： | 软件学院 软件工程 |



同济大学

Tongji University

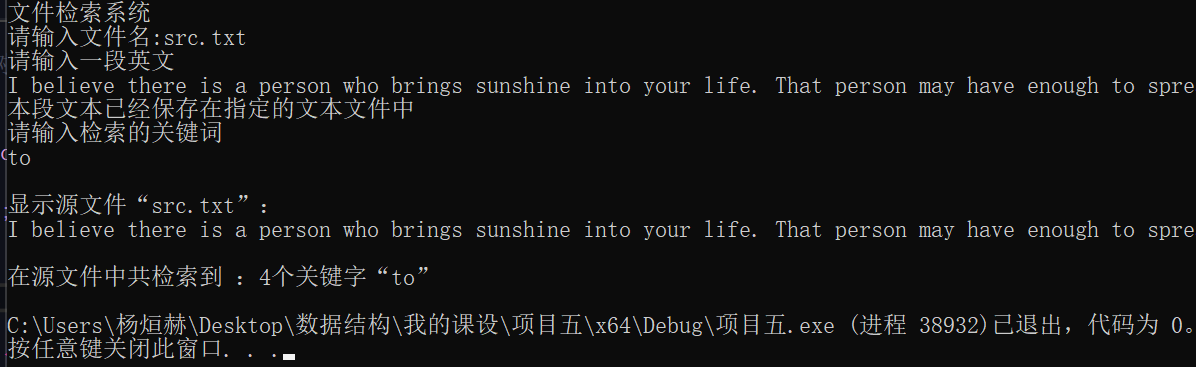
1. **题目简介**

**1.实验背景**

建立一个文本文件，文件名由用户用键盘输入，输入一个不含空格的关键字，统计输出关键字在文本中的出现次数。

**2.实验功能**

本项目的设计要求可以分成两个部分实现：首先建立一个文本文件，文件名由用户用键盘输入；然后输入一个不含空格的关键字，统计输出该单词在文本中的出现次数。

**3.项目示例**

1. **项目设计**

## 1.数据结构设计

**字典树节点 (node 结构体)**

节点属性：

int cnt：用于存储单词在文本中出现的频率。每当在字典树中插入相应单词时，该计数器递增。

node\* next[26]：数组，表示26个可能的后续节点，每个元素对应英文字母表中的一个字母。例如，next[0] 对应字母 'a' 的节点，next[25] 对应字母 'z' 的节点。

节点构造：

在构造函数中，cnt 初始化为0，表示单词尚未出现。

next 数组的每个元素初始化为 nullptr，表示开始时没有任何后续节点。

**字典树 (Trie 类)**

根节点：node\* root：字典树的起始节点。它不对应任何字母，但其后续节点表示字典树中的第一个字母。

搜索结果：int find：用于存储搜索单词的频率。当单词在树中被找到时，find 将更新为该单词的频率。

构造与析构：构造函数中创建根节点。

析构函数中调用 deleteTree 方法递归删除所有字典树节点，以释放分配的内存。

## 2.类结构设计

**字典树 (Trie 类)**

私有成员变量：

node\* root：根节点，字典树的起始点。

公有成员变量：

int find：用于存储搜索到的特定单词的频率。

构造函数：初始化根节点。

析构函数：递归释放字典树内存。

成员函数：包括插入单词、搜索单词和释放字典树内存的函数。

## 3.成员与操作设计

**插入单词 (insert )**

接收一个字符数组 word，逐字符构建字典树。如果遇到不存在的字符节点，则创建新节点。

**搜索单词 (search )**

接收一个字符数组 word，在字典树中搜索该单词，并更新 find 变量以表示找到的单词频率。

**释放字典树内存 (deleteTree )**

递归地释放字典树的所有节点，防止内存泄漏。

**文件读写与处理**

在 main 函数中，程序首先要求用户输入文件名和文本内容，并将内容写入文件。

然后，程序读取文件内容，并逐词将其插入到字典树中。

用户输入搜索关键词后，程序在字典树中搜索该词，并输出词频。

## 4.算法设计

**字典树构建算法 (Trie Construction)**

初始化：构建一个空的字典树，从一个空的根节点开始。

单词插入：

遍历待插入的单词的每个字符。

对于每个字符，计算其在字典树中的索引（例如，'a' 对应索引0，'z' 对应索引25）。

检查当前字符对应的字典树节点是否存在，如果不存在则创建新节点。

移动到下一个字符对应的节点，重复该过程直到单词结束。

在单词的最后一个字符对应的节点上，增加单词频数计数。

**单词搜索算法**

从根节点开始：从字典树的根节点开始搜索。

遍历单词的每个字符：

对于单词的每个字符，计算其在字典树中的索引。

检查当前字符是否有对应的字典树节点。如果没有，说明单词不在字典树中，搜索失败。

移动到下一个字符对应的节点。

返回结果：如果单词的所有字符都在字典树中找到对应节点，返回该单词的频数。

**递归删除字典树**

递归基：如果当前节点为空，返回。

递归删除子节点：对于当前节点的每一个可能的子节点，递归调用删除函数。

删除当前节点：删除当前节点，释放其占用的内存。

**文本处理和字典树应用**

文件读写：

读取用户输入的文本，将其写入指定的文件。

重新打开文件，读取文件内容以便后续处理。

分词和字典树插入：

遍历文件内容的每个字符，使用字母检查（isalpha）和转换（tolower）将文本分割为单词。

对每个提取的单词，使用插入算法将其加入字典树。

关键词搜索：

用户输入关键词，对其进行同样的字母检查和转换。

使用字典树的搜索算法找到关键词的频数。

1. **项目实现**

**3.1. 字典树节点的定义**

1. **struct** node {
2. **int** cnt; // 单词频数
3. node\* next[26]; // 26个可能的后续节点
4. node() {
5. cnt = 0;
6. **for** (**int** i = 0; i < 26; i++) {
7. next[i] = nullptr;
8. }
9. }
10. };

这部分代码定义了字典树的基本结构，每个节点包含一个表示单词出现次数的计数器 cnt 和一个数组 next，存储可能的后续节点。

**3.2. 字典树树类的实现**

1. // 建立字典树类
2. **class** Trie {
3. **private**:
4. node\* root; // 根节点
5. **public**:
6. **int** find;
7. Trie() {
8. root = **new** node();
9. find = 0;
10. }
11. ~Trie() {
12. deleteTree(root); // 递归删除所有节点
13. }
14. **void** insert(**char**\* word); // 构造插入函数
15. **void** search(**char**\* word); // 构造搜索函数
16. **void** deleteTree(node\* root); // 释放字典树内存
17. };

这部分代码实现了一个字典树，包括插入、搜索和删除树的功能。

**3.3. 插入函数与搜索函数实现**

1. **void** Trie::insert(**char**\* word) {
2. node\* current = root;
3. **for** (**int** i = 0; word[i] != '\0'; i++) {
4. **int** index = word[i] - 'a';
5. **if** (current->next[index] == nullptr) {
6. current->next[index] = **new** node();
7. }
8. current = current->next[index];
9. }
10. current->cnt++;
11. }
13. **void** Trie::search(**char**\* word) {
14. node\* current = root;
15. **for** (**int** i = 0; word[i] != '\0'; i++) {
16. **int** index = word[i] - 'a';
17. **if** (current->next[index] == nullptr) {
18. find = 0;
19. **return**;
20. }
21. current = current->next[index];
22. }
23. find = current->cnt;
24. }

在这个代码段中，提供了字典树（Trie）的两个基本功能：插入（insert）和搜索（search）。

**void Trie::insert(char\* word)**

功能：将一个单词插入字典树。

初始化: 从字典树的根节点开始。

遍历单词中的每个字符:

计算字符的索引（假设只处理小写字母，'a' 的索引为 0，'b' 的索引为 1，依此类推）。

如果当前字符对应的节点不存在，则创建一个新节点。

移动到下一个节点。

增加单词频率: 遍历结束后，当前节点的 cnt 属性增加 1，表示这个单词在字典树中的出现次数。

**void Trie::search(char\* word)**

功能：在字典树中搜索一个单词，并记录其出现频率。

初始化: 从字典树的根节点开始。

遍历单词中的每个字符:

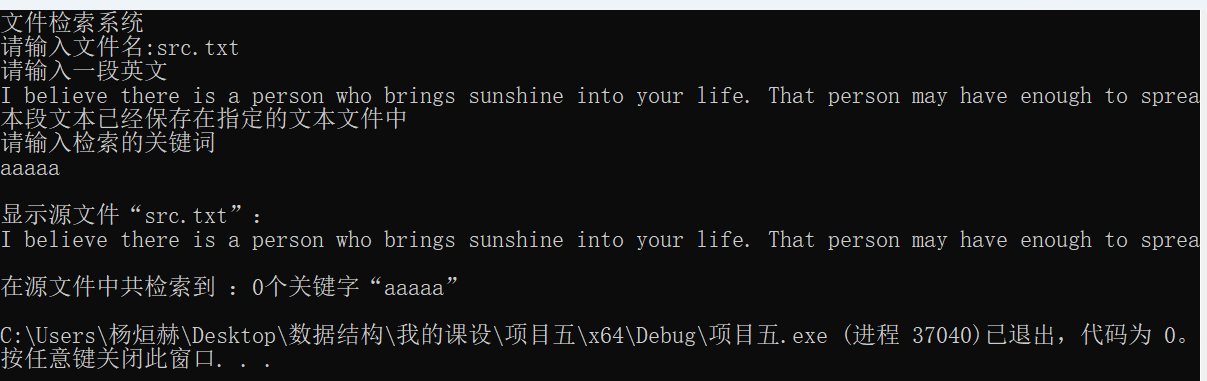
计算字符的索引。

如果当前字符对应的节点不存在，表示该单词不存在于字典树中，将 find 设置为 0 并返回。

否则，继续遍历到下一个节点。

返回单词频率: 遍历结束后，使用 find 属性记录当前节点的 cnt 值，即单词在字典树中的出现频率。

**健全性测试：**

****

**内存管理:**

程序中有对动态分配的内存进行清理的逻辑。deleteTree 函数递归地释放字典树的所有节点，这有助于防止内存泄漏。

文件操作的错误处理:

在尝试打开文件时，程序检查文件指针是否为 nullptr。这是一个很好的做法，可以防止对无效文件指针的操作。

**输入处理:**

程序处理了字母以外的字符，确保只有字母被插入到字典树中。这通过 isalpha 函数实现，帮助确保数据的一致性和准确性。

在插入单词时，程序将所有大写字母转换为小写，这有助于保持数据的一致性，并且使搜索不区分大小写。

**程序的鲁棒性:**

通过在 main 函数中进行用户交互，程序处理了用户输入，包括文件名和文本内容，增强了用户体验。

**面向对象的设计:**

程序采用类（Trie）来封装字典树的逻辑，使得代码更加模块化，易于维护和扩展。

1. **项目创新点**

本项目的创新之处在于实现了一个基于字典树（Trie）的文本检索系统，该系统能够高效地处理和查询大量文本数据。字典树的应用提高了单词查找的速度，尤其是在处理大量数据时。此外，通过将所有输入转换为小写，系统能够实现大小写不敏感的搜索，增强了用户体验。程序中还包含了对非字母字符的处理，保证了数据的一致性和准确性。这些特点使得该项目在文本处理和检索领域具有创新性和实用价值。

**五、实验小结**

通过本次实验，我们成功地构建了一个基于字典树的文本检索系统，验证了字典树在文本处理中的有效性和效率。实验过程中，我们重点关注了内存管理、错误处理和用户交互的重要性，从而确保了程序的健全性和鲁棒性。虽然在实验中确实展现了字典树的高效检索能力，但也发现了一些潜在的改进空间，如对内存分配的进一步检查和对异常输入的更全面处理。总体来说，这个项目不仅提升了我们对字典树数据结构的理解，也为未来在更复杂的文本处理和数据结构设计项目中奠定了坚实的基础。