

《数据结构》实验

期末报告

班 级： 信息221

学 号： 19122120

姓 名： 侯兆晗

任课教师： 刘浏

**实验目的：**

使用存储结构来存储字符及其对应的频率。

对从文件中读取的字符进行频率统计。

对统计出的频率进行排序并打印结果。

分析不同数据结构和排序算法下时间复杂度和空间复杂度的不同。

**程序设计思路：**

**方法一（代码在文件夹first中）**

方法一的程序设计思路遵循了数据存储结构选择、排序算法选择、功能分解、内存管理、灵活性和简洁性等原则。选择链表作为数据结构，链表的选择使得插入、删除等操作更为高效。同时使用冒泡排序算法进行排序。通过函数进行操作，从文本文件中读取字符并进行统计。代码分解成多个函数，每个函数执行特定任务，确保程序的可读性和可维护性。

**方法二（代码在文件夹second中）**

方法二思路概括为，读取文本文档中的字符并统计每个字符的频率，然后将结果存储在一个哈希表中。程序首先尝试打开文本文档，如果文件不存在或无法打开，则输出错误消息并退出。然后，它创建一个哈希表，并逐个读取文件中的字符。对于每个字符，会在哈希表中查找该字符的频率，如果找到了就增加频率，如果没有找到就插入一个新节点并设置频率为1。最后，打印哈希表中的所有节点，并关闭文件。

**实验内容：**

**方法一（代码在文件夹first中）**

**算法与代码分析：**

头文件

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

定义一个结构体Node，用于表示一个链表中的结点。每个结点包含三个字段：

character：用于存储一个字符。

frequency：用于存储该字符的频率。

next：是一个指向下一个结点的指针。

// 结点结构体

typedef struct Node {

char character;// 用于存储汉字

int frequency;// 用于存储该汉字的频率

struct Node\* next; //指向下一个结点的指针

} Node;

使用一个函数createNode，它的作用是创建一个新的结点，并初始化字符和频率。这个函数接受一个字符作为参数，并为这个新结点动态分配内存。之后，将传入的字符赋值给新结点的character字段，将新结点的frequency字段初始化为1（表示该字符首次出现），将新结点的next字段初始化为NULL（表示当前结点没有下一个结点）。最后，返回这个新结点。

时间复杂度：createNode(char c): O(1)

空间复杂度：createNode(char c): O(1)，分配一个结点的大小，所以空间复杂度是O(1)。

// 创建结点

Node\* createNode(char c) {

Node\* newNode = (Node\*)malloc(sizeof(Node)); // 为新结点动态分配内存

newNode->character = c;// 将传入的字符赋值给新结点的character字段

newNode->frequency = 1; // 新结点的频率初始化为1，表示该汉字首次出现

newNode->next = NULL;// 新结点的next字段初始化为NULL，表示当前结点没有下一个结点

return newNode; // 返回新结点

}

定义一个函数insertOrUpdate，它的作用是插入或更新一个结点的频率。如果给定的字符在链表中已经存在，那么它就更新该字符的频率；如果给定的字符在链表中不存在，那么它就插入一个新的结点。

函数的参数有两个：

head：链表的头部结点。

c：要插入或更新的字符。

函数的实现逻辑：

定义一个指针current，并让它指向链表的头部结点head。

使用一个while循环遍历链表。如果当前结点的下一个结点的字符与要插入的字符相同，那么更新该结点的频率并立即返回。

如果在链表中遍历完毕后仍然没有找到与要插入的字符相同的结点，那么插入一个新的结点。

时间复杂度：对于每个字符，遍历链表一次。最坏情况下，需要遍历整个链表，因此时间复杂度为O(n)。

空间复杂度：最多为每个字符分配一个结点，所以空间复杂度为O(n)。

// 插入或更新结点的频率

void insertOrUpdate(Node\* head, char c) {

Node\* current = head;// 从链表头部开始查找

// 如果链表不为空并且当前结点的下一个结点的字符与要插入的字符相同

while (current->next != NULL) {

if (current->next->character == c) {

current->next->frequency++;// 更新该结点的频率

return;

}

current = current->next;// 继续查找下一个结点

}

current->next = createNode(c); // 如果链表为空或者已经遍历完链表但未找到匹配的字符，则插入新的结点

}

定义一个函数bubbleSort，用于对链表中的结点进行冒泡排序。冒泡排序重复地遍历要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。

函数的参数是一个链表的头部结点head。

函数的实现逻辑：

定义一个指针ptr1，让它指向链表的第一个结点。

定义一个指针lptr，初始值为NULL。lptr用于记录已经遍历过的结点的位置。

使用一个do-while循环进行冒泡排序。

在每次循环中，首先将swapped变量设置为0，表示没有发生交换。

然后使用一个while循环遍历链表。遍历的范围是从当前结点到lptr之间的结点。

在每次遍历中，如果当前结点的频率小于下一个结点的频率，那么就交换这两个结点的频率和汉字。

交换完成后，将swapped变量设置为1，表示发生了交换。

继续遍历下一个结点，直到遍历完整个链表。

在每次循环结束后，将lptr更新为当前遍历到的最后一个结点。

如果在循环中没有发生交换，说明链表已经完成排序了，可以提前退出循环。

时间复杂度：冒泡排序的时间复杂度是O(n2)。由于只对链表的长度进行一次冒泡排序，所以实际上为O(n)。

空间复杂度：不需要额外的空间，空间复杂度为O(1)。

// 冒泡排序

void bubbleSort(Node\* head) {

int swapped;

Node\* ptr1;

Node\* lptr = NULL;

do {

swapped = 0;

ptr1 = head->next;

while (ptr1->next != lptr) {

if (ptr1->frequency < ptr1->next->frequency) {

// 交换频率和汉字

int tempFrequency = ptr1->frequency;

char tempCharacter = ptr1->character;

ptr1->frequency = ptr1->next->frequency;

ptr1->character = ptr1->next->character;

ptr1->next->frequency = tempFrequency;

ptr1->next->character = tempCharacter;

swapped = 1;

}

ptr1 = ptr1->next;

}

lptr = ptr1;

} while (swapped);

}

定义一个函数printSortedFrequency，用于打印排序后的链表中每个结点的字符和频率。

函数的参数是一个链表的头部结点head。

函数的实现逻辑：

定义一个指针current，让它指向链表的第二个结点（跳过头结点）。

使用一个while循环遍历链表。如果当前结点不为空，那么就打印当前结点的字符和频率。

继续遍历下一个结点，直到遍历完整个链表。

时间复杂度：O(n)。

空间复杂度：不需要额外的空间，空间复杂度为O(1)。

// 打印排序后的频率

void printSortedFrequency(Node\* head) {

Node\* current = head->next; // 从链表的第二个结点开始打印（跳过头结点）

while (current != NULL) {

printf("字符： %c 频率： %d\n", current->character, current->frequency);

current = current->next;

}

}

主函数main，实现以下功能：

打开文件：

使用fopen函数尝试打开名为"wenben.txt"的文件，以只读模式（"r"）。

如果文件打开失败（例如文件不存在），则输出错误信息并返回1。

初始化链表：

创建一个头结点，使用字符\0进行初始化。这个头结点不存储实际的字符和频率信息，只是作为链表的起始点。

读取文件并更新链表：

使用fgetc函数从文件中逐个读取字符，直到遇到文件结束标记（EOF）。

对于读取到的每一个非0字符，调用insertOrUpdate函数，将其插入到链表中或更新其在链表中的频率。

关闭文件：

使用fclose函数关闭已打开的文件。

排序链表：

调用bubbleSort函数对链表进行冒泡排序，按照字符的频率从低到高进行排序。

打印排序后的链表：

调用printSortedFrequency函数，打印排序后的链表中每个结点的字符和频率。

时间复杂度：O(n)，其中n是文件的字符数。

空间复杂度：文件读取操作需要额外的空间来存储文件的内容，所以空间复杂度为O(n)。

int main() {

FILE\* file = fopen("wenben.txt", "r");

if (file == NULL) {

printf("Unable to open file.\n");

return 1;

}

Node\* head = createNode('\0'); //创建一个以'\0'为初始字符的结点作为头结点

// 从文件中读取字符，直到遇到文件结束标记（EOF）

char c;

while ((c = fgetc(file)) != EOF) {

if (c!=0) {

insertOrUpdate(head, c); // 插入或更新结点频率

}

}

fclose(file);

bubbleSort(head); // 排序

printSortedFrequency(head); // 打印排序后的频率

return 0;

}

**方法二（代码在文件夹second中）**

**算法与代码分析：**

头文件

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

定义哈希表的最大大小

#define MAX\_SIZE 1000 // 定义哈希表的最大大小

Node结构体，这个结构体代表哈希表中的一个结点。它有三个成员：

character: 存储字符。

frequency: 存储字符的频率。

next: 指向下一个结点的指针，用于链表结构。

typedef struct Node {

char character; // 存储字符

int frequency; // 存储字符的频率

struct Node \*next; // 指向下一个Node的指针，用于处理哈希冲突

} Node;

HashTable结构体，这个结构体代表整个哈希表。它有两个成员：

table: 是一个指向指针的指针，用于存储哈希表中的所有结点。使用一个动态分配的二维数组，其中每一行都是一个链表的头结点。

size: 哈希表的大小。

typedef struct HashTable {

Node \*\*table; // 二维指针，指向Node指针的数组，即哈希表的主体部分

int size; // 哈希表的大小

} HashTable;

hash函数，使用取模操作来计算给定键的哈希值。

时间复杂度：O(1)。

空间复杂度：O(1)。

int hash(int key, int size) {

return key % size; // 使用取余法计算哈希值

}

createHashTable函数，创建一个指定大小的哈希表。首先分配一个HashTable结构的内存，并初始化其大小。然后，它分配一个Node指针的数组，并将其所有元素初始化为NULL。

HashTable \*createHashTable(int size) {

HashTable \*hashTable = (HashTable \*)malloc(sizeof(HashTable));// 为HashTable结构体分配内存

hashTable->size = size;// 设置哈希表的大小

hashTable->table = (Node \*\*)malloc(size \* sizeof(Node \*));//为Node指针数组分配内存

for (int i = 0; i < size; i++)

{

hashTable->table[i] = NULL; // 初始化Node指针数组的每一个元素为NULL

}

return hashTable;// 返回新创建的哈希表的指针

}

insert函数，将一个具有特定字符和频率的新结点插入哈希表。首先计算新结点的索引，然后分配一个新的Node结构的内存，并将字符和频率存储在其中。最后将新结点插入到哈希表的相应索引位置。

时间复杂度：O(1)。

空间复杂度：O(1)。为新结点分配内存

void insert(HashTable \*hashTable, char character, int frequency) {

int index = hash(hash(character, hashTable->size), hashTable->size);// 使用双重哈希函数计算索引

Node \*node = (Node \*)malloc(sizeof(Node));// 为新的节点动态分配内存

node->character = character;// 设置节点的字符和频率

node->frequency = frequency;

node->next = hashTable->table[index];// 如果哈希表在该索引处已经有节点，则将新节点插入到链表的开头

hashTable->table[index] = node;// 将新节点添加到哈希表的指定索引位置

}

search函数，在哈希表中查找一个具有特定字符的结点。首先计算要查找的字符的索引，然后从哈希表的相应索引位置开始遍历链表，直到找到具有正确字符的结点或达到链表的末尾。

时间复杂度：O(n)（n 是哈希表中的结点数量）。遍历链表查找结点

空间复杂度：该函数只返回一个指针，没有明显的空间消耗。

Node \*search(HashTable \*hashTable, char character) {

int index = hash(hash(character, hashTable->size), hashTable->size);// 使用双重哈希函数计算索引

Node \*current = hashTable->table[index];// 开始从哈希表的指定索引位置查找

while (current != NULL && current->character != character)// 如果当前节点为空或其字符与要查找的不匹配，则继续查找下一个节点

{

current = current->next;

}

return current;// 返回找到的节点或如果未找到则返回NULL

}

printHashTable函数，遍历哈希表并打印所有结点的字符和频率。首先遍历哈希表的每个索引，然后遍历每个索引上的链表，并打印每个结点的字符和频率。

时间复杂度：O(mn)。遍历哈希表中的每个位置：O(m)（m 是哈希表中的结点数量），对于每个位置，遍历链表打印结点：O(n)（n 是该位置的链表长度）。因此，总的时间复杂度为 O(mn)。

空间复杂度：O(mn)。insert 函数具有最佳的时间复杂度（O(1)），而 search 和 printHashTable 函数的时间复杂度与哈希表中的结点数量有关（分别为 O(n) 和 O(mn)）。在空间复杂度方面，通常与哈希表中的结点数量有关（O(m) 和 O(mn)）。

void printHashTable(HashTable \*hashTable) {

for (int i = 0; i < hashTable->size; i++)// 遍历哈希表的每个索引位置

{

Node \*current = hashTable->table[i];

while (current != NULL)// 从当前索引位置开始遍历链表，并打印每个节点的字符和频率

{

printf("字符： %c 频率： %d\n", current->character, current->frequency);

current = current->next;

}

}

}

主函数使用了哈希表来存储每个汉字的频率，查找和更新频率的操作都可以在常数时间内完成。主要实现以下几个功能：

文件打开：使用fopen函数尝试打开名为"wenben.txt"的文件以进行读取。如果文件打开失败（例如，文件不存在或没有读取权限），则会输出"无法打开文件"的消息，并以返回值1结束程序。

创建哈希表：使用createHashTable函数创建一个哈希表，其最大大小为MAX\_SIZE。

读取文件并统计频率：使用fscanf函数从文件中逐个读取字符，直到文件结束（EOF）。对于每个读取的字符：如果字符是汉字，则在哈希表中查找该字符的频率。如果找到了相应的字符，则增加其频率。如果未找到相应的字符，则使用insert函数在哈希表中插入一个新结点，并将频率设置为1。

打印哈希表：使用printHashTable函数打印哈希表中的所有结点。

关闭文件：使用fclose函数关闭文件。

时间复杂度：O(n) 或 O(m)。打开文件：这一步的时间复杂度是O(1)，因为打开文件是一个常数时间的操作；创建哈希表：创建哈希表通常是一个常数时间的操作，所以这一步的时间复杂度也是O(1)；读取文件中的每个字符并统计频率：这是一个循环操作，直到文件结束。所以，如果文件的大小为n，这一步的时间复杂度是O(n)；在哈希表中查找字符的频率：这个操作的时间复杂度取决于哈希表的设计和实现。在最坏的情况下，如果哈希表中的所有元素都需要被检查（例如，哈希函数质量差或者数据分布不均匀），那么查找的时间复杂度可能是O(n)。在平均情况下，如果哈希函数数据分布均匀，那么查找的时间复杂度可能是O(1)；插入新结点或更新结点频率：这个操作的时间复杂度也取决于哈希表的设计和实现。在最坏的情况下，插入或更新一个结点的频率可能需要O(1) 时间。但是，如果哈希表中的所有元素都需要被检查，那么插入或更新一个结点的频率可能需要O(n) 时间；打印哈希表中的所有结点：这个操作的时间复杂度取决于哈希表的大小。如果哈希表的大小为 m，那么打印所有结点的时间复杂度是O(m)。时间复杂度取决于文件的大小和哈希表的大小，在最好的情况下，时间复杂度是 O(1)；在最坏的情况下，时间复杂度是 O(n) 或 O(m)。

空间复杂度：O(n) 或 O(m)。打开文件不需要额外的空间；创建哈希表需要额外的空间来存储哈希表的结构和结点。假设哈希表的大小为 m，那么创建哈希表的空间复杂度是 O(m)；读取文件中的每个字符并统计频率需要额外的空间来存储字符和频率。这个空间的大小取决于文件的大小，所以空间复杂度是 O(n)；在哈希表中查找字符的频率需要额外的空间来存储哈希表的结构和结点。这个空间的大小取决于哈希表的大小，所以空间复杂度是 O(m)；插入新结点或更新结点频率需要额外的空间来存储新的结点或更新结点的频率。这个空间的大小取决于哈希表的大小，所以空间复杂度是 O(m)；打印哈希表中的所有结点需要额外的空间来存储打印的结果。这个空间的大小取决于哈希表的大小，所以空间复杂度是 O(m)。空间复杂度取决于文件的大小和哈希表的大小。在最好的情况下，空间复杂度是 O(1)；在最坏的情况下，空间复杂度是 O(n) 或 O(m)。

int main()

{

FILE \*file = fopen("wenben.txt", "r");// 打开文件

if (file == NULL)

{

printf("无法打开文件\n");

return 1;

}

HashTable \*hashTable = createHashTable(MAX\_SIZE);// 创建哈希表

char character;

int frequency = 0;

while (fscanf(file, "%c", &character) != EOF)// 读取文件中的每个字符，并统计频率

{

// 如果遇到字符，则统计频率，并在哈希表中插入新节点或更新节点频率

if (character)

{

Node \*node = search(hashTable, character);// 在哈希表中查找该字符的频率，如果找到则增加频率，否则插入新节点并设置频率为1

if (node == NULL)

{

insert(hashTable, character, 1);// 插入新节点并设置频率为1

}

else

{

node->frequency++;// 如果找到则增加频率

}

}

}

printHashTable(hashTable);// 打印哈希表中的所有节点

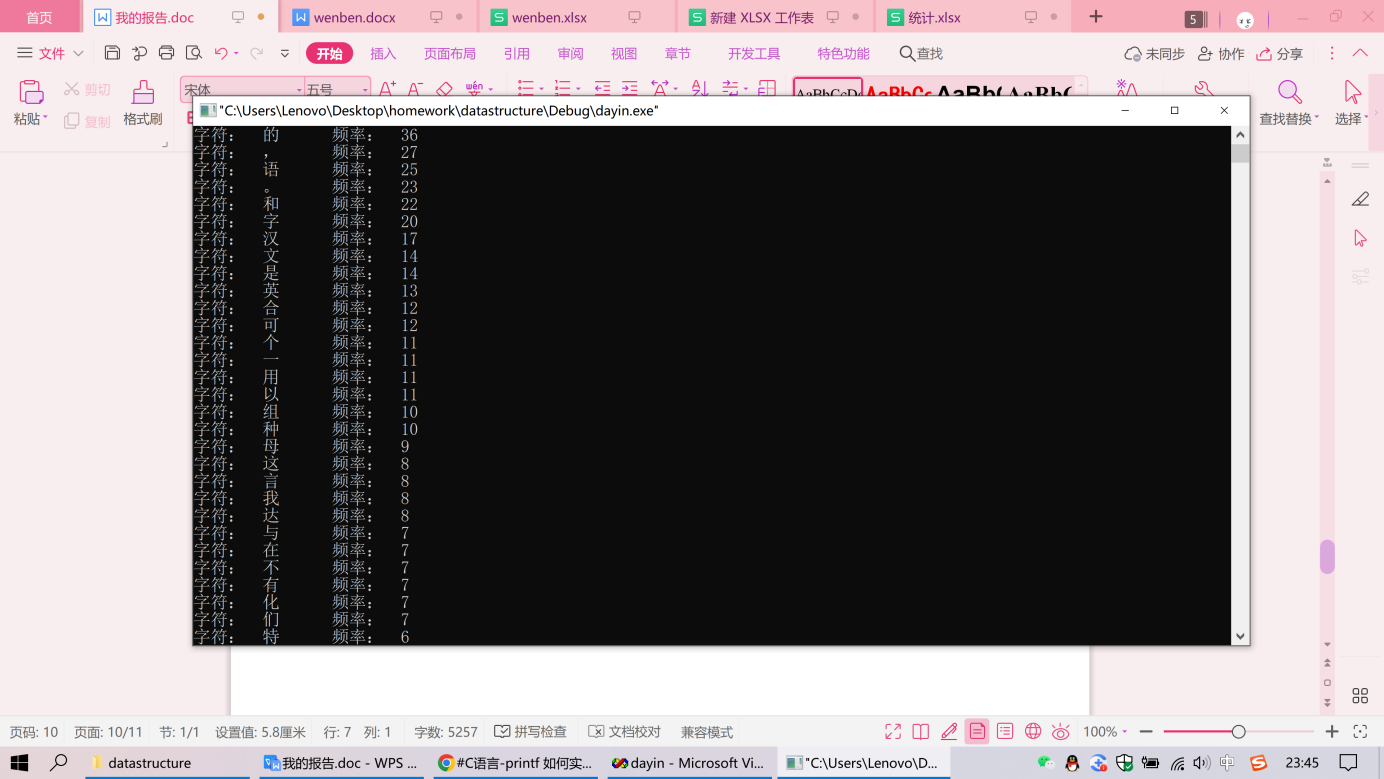
fclose(file);// 关闭文件

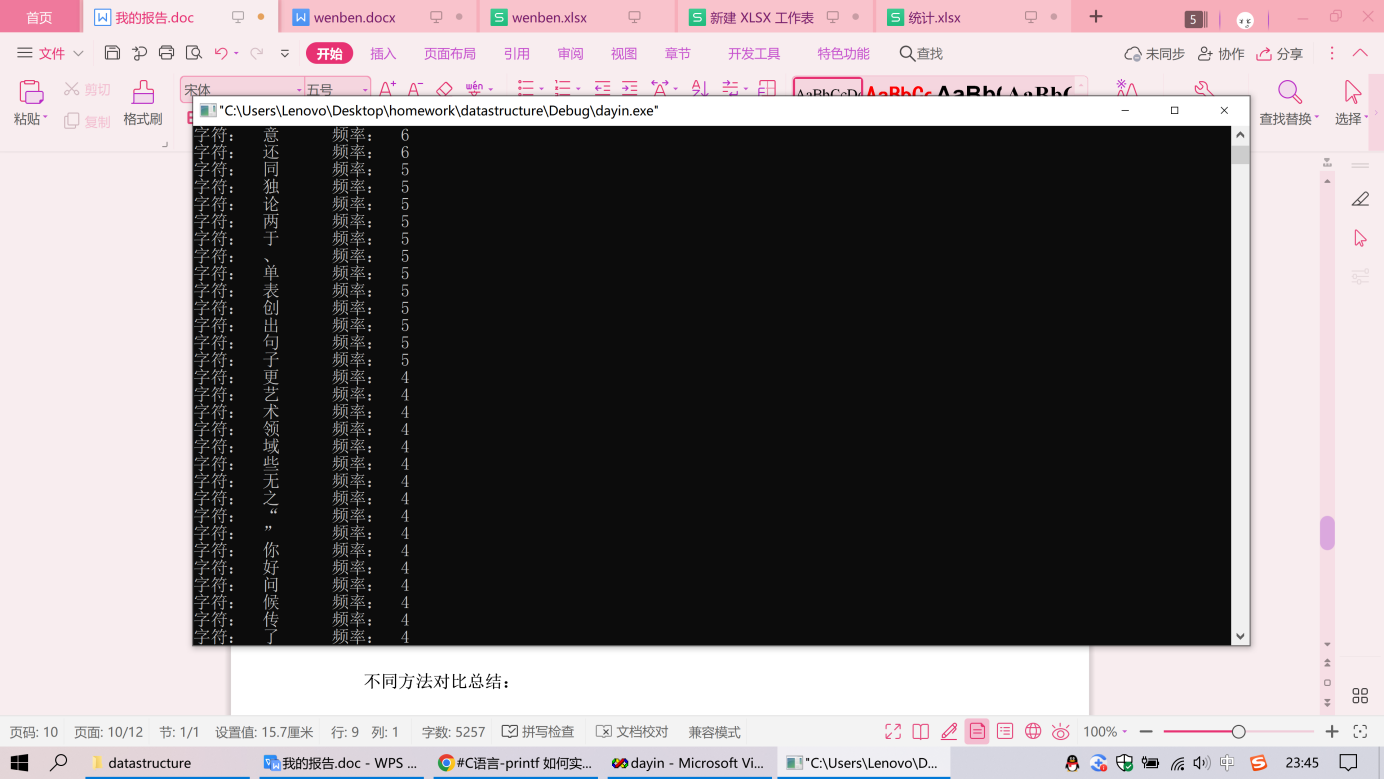
return 0;

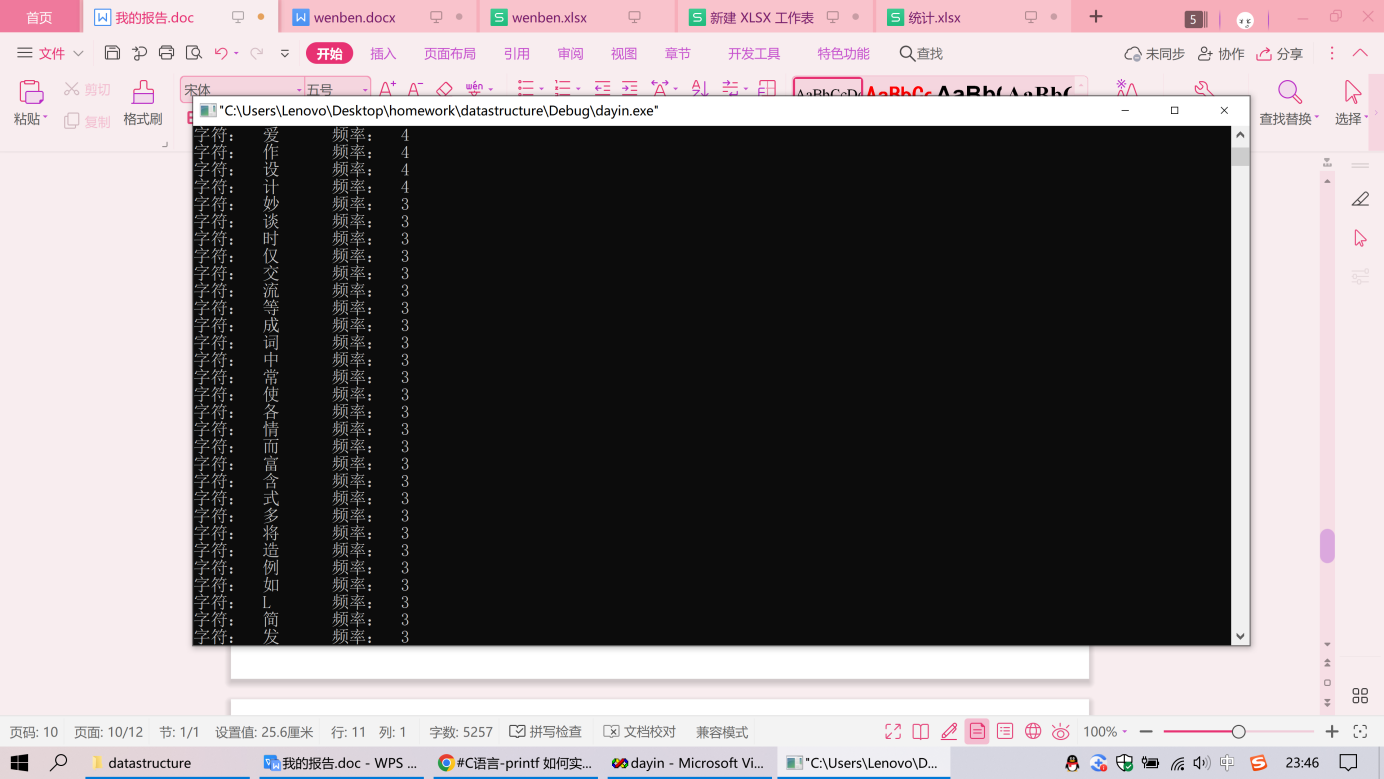
}

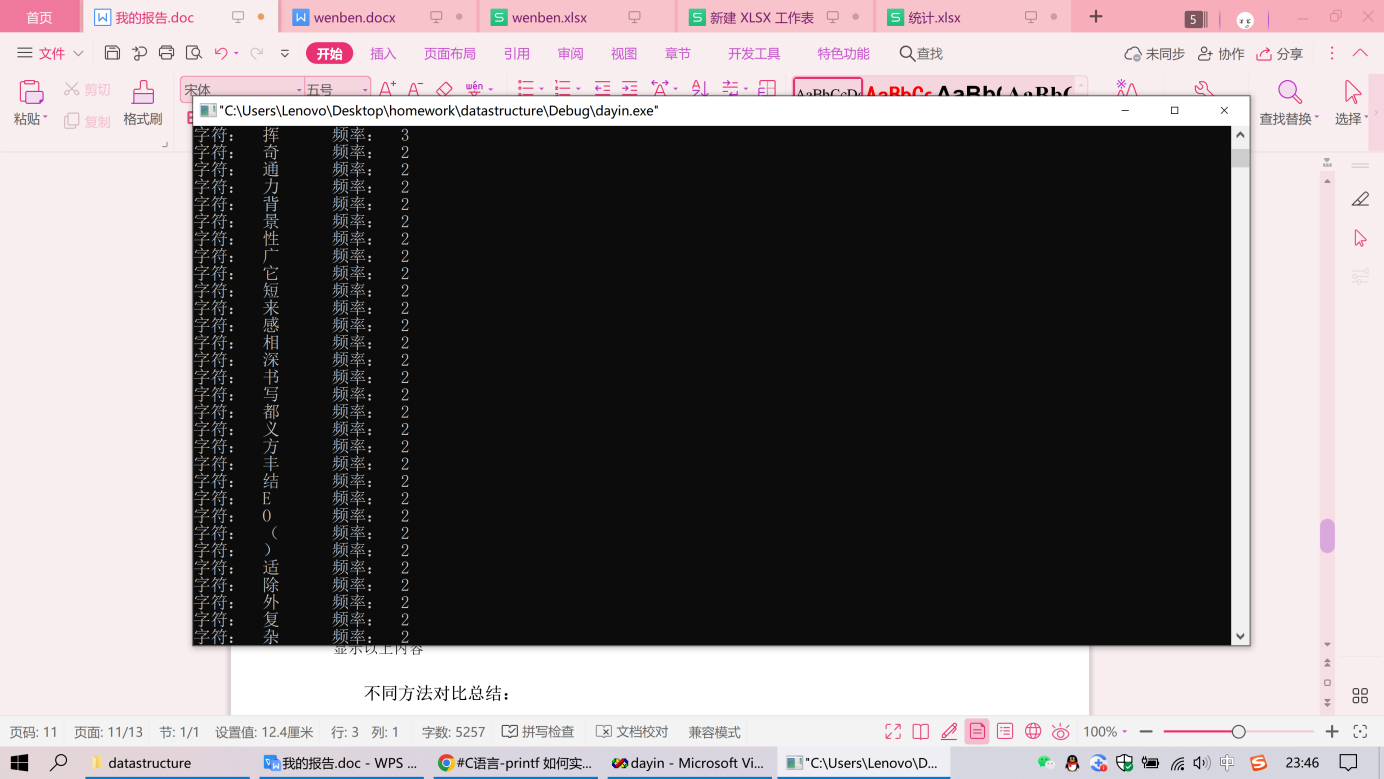
**实验结果：**

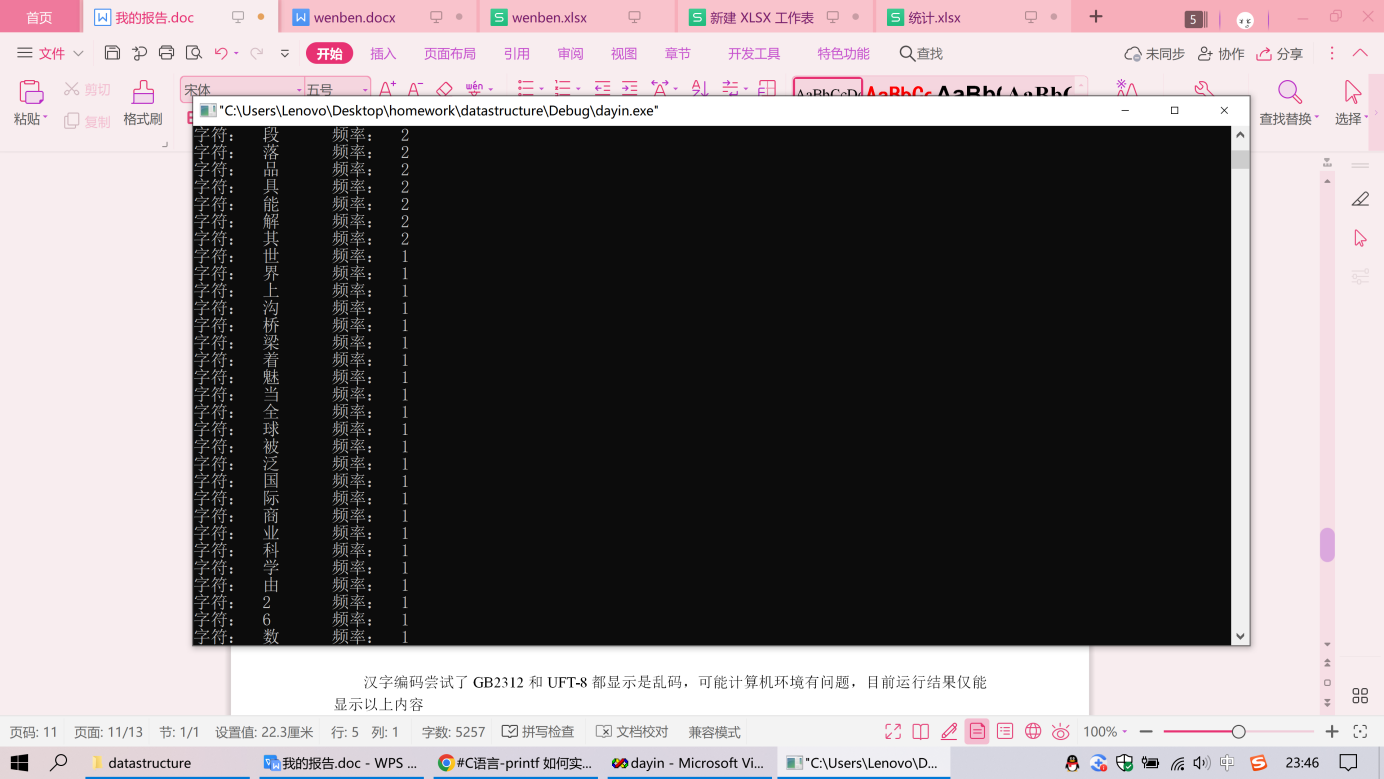
运行结果为以下图片。

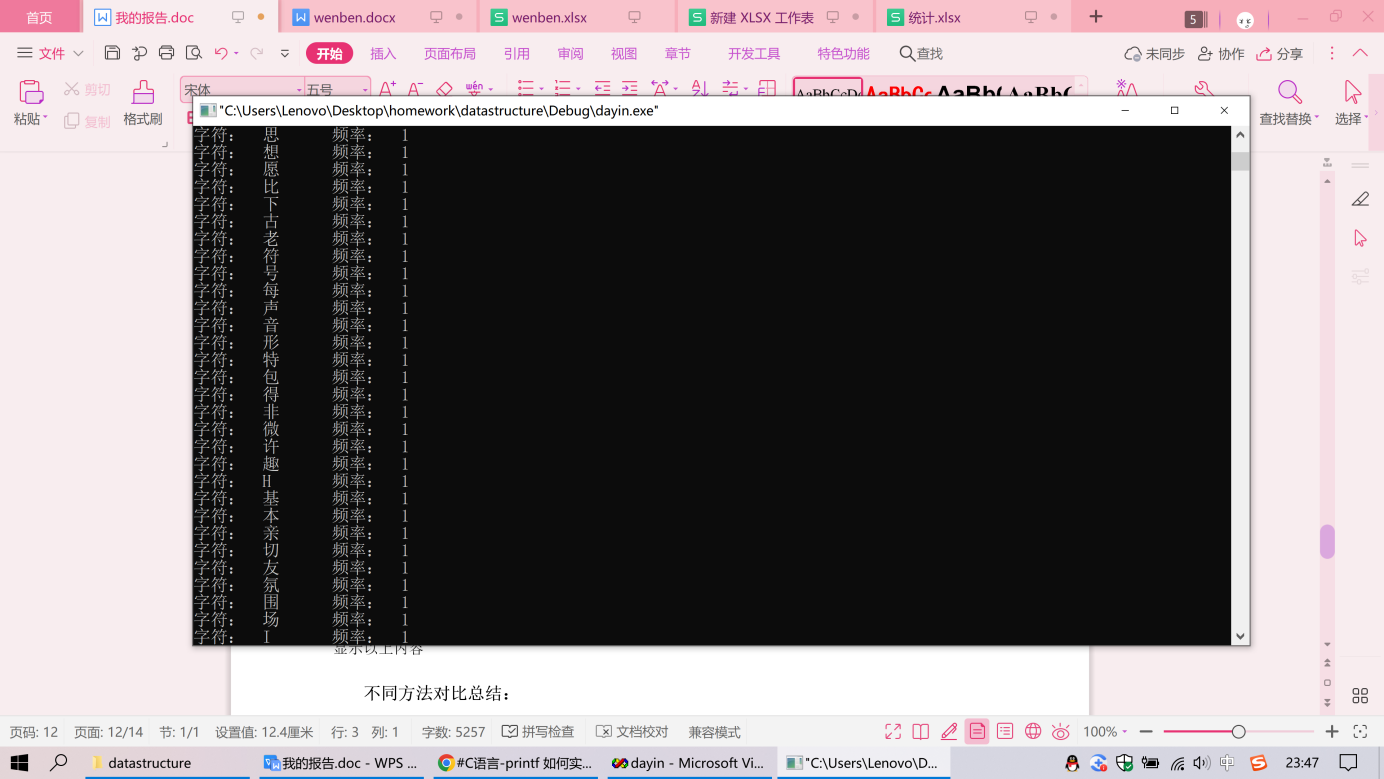


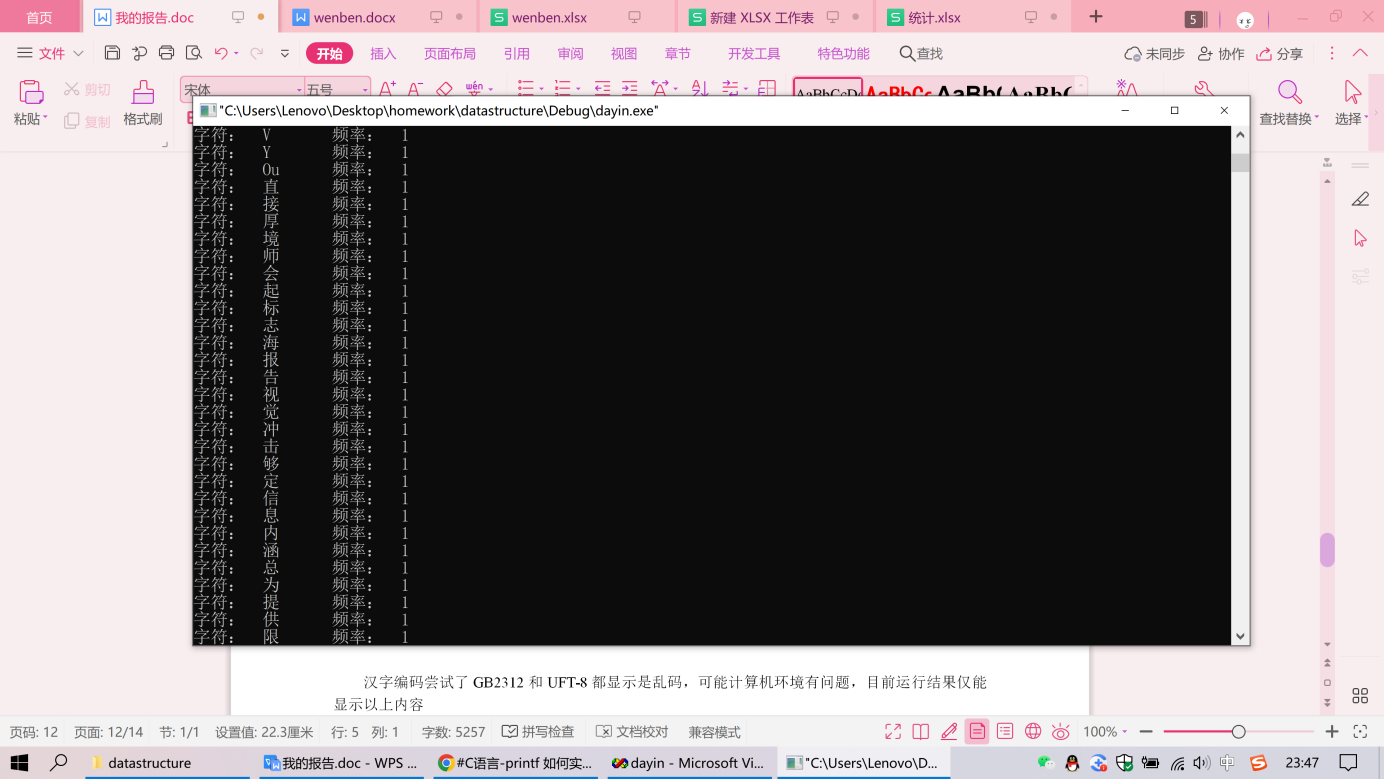


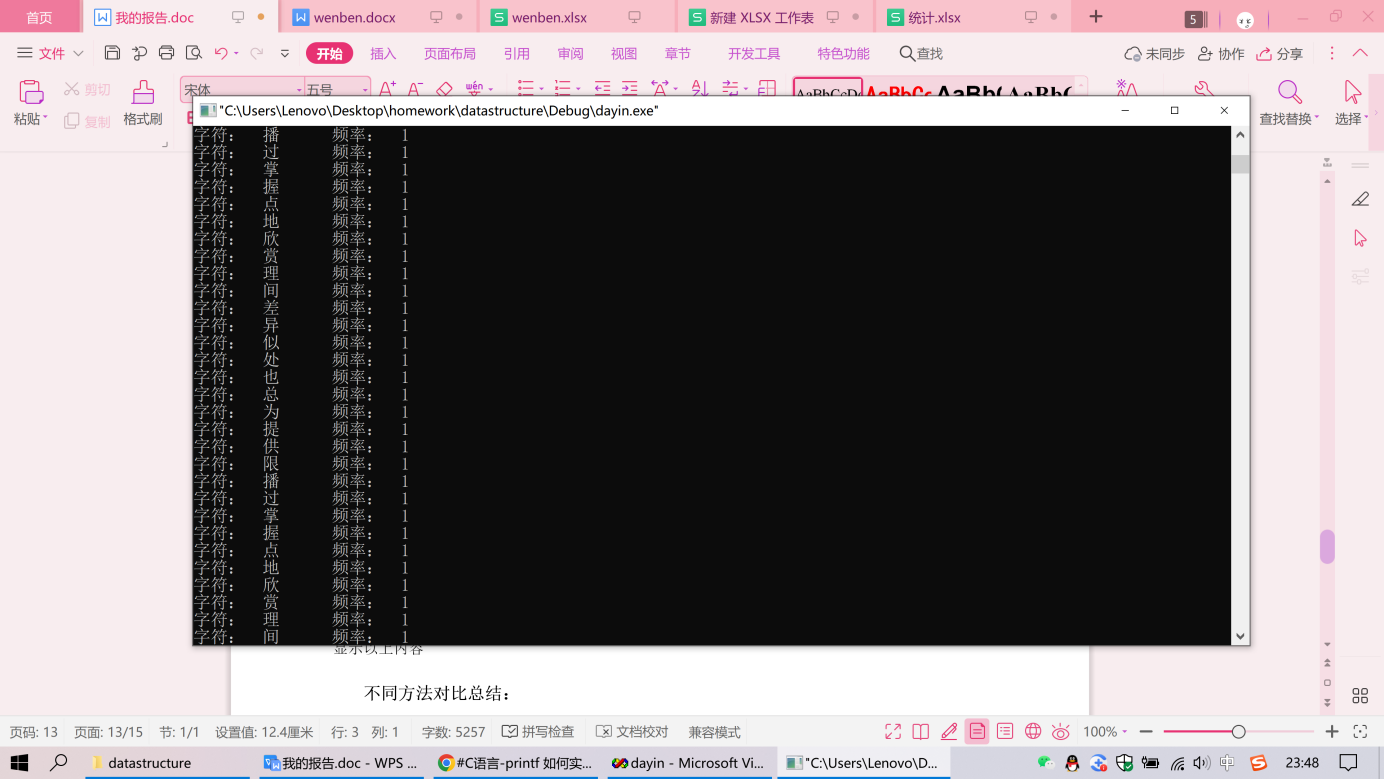


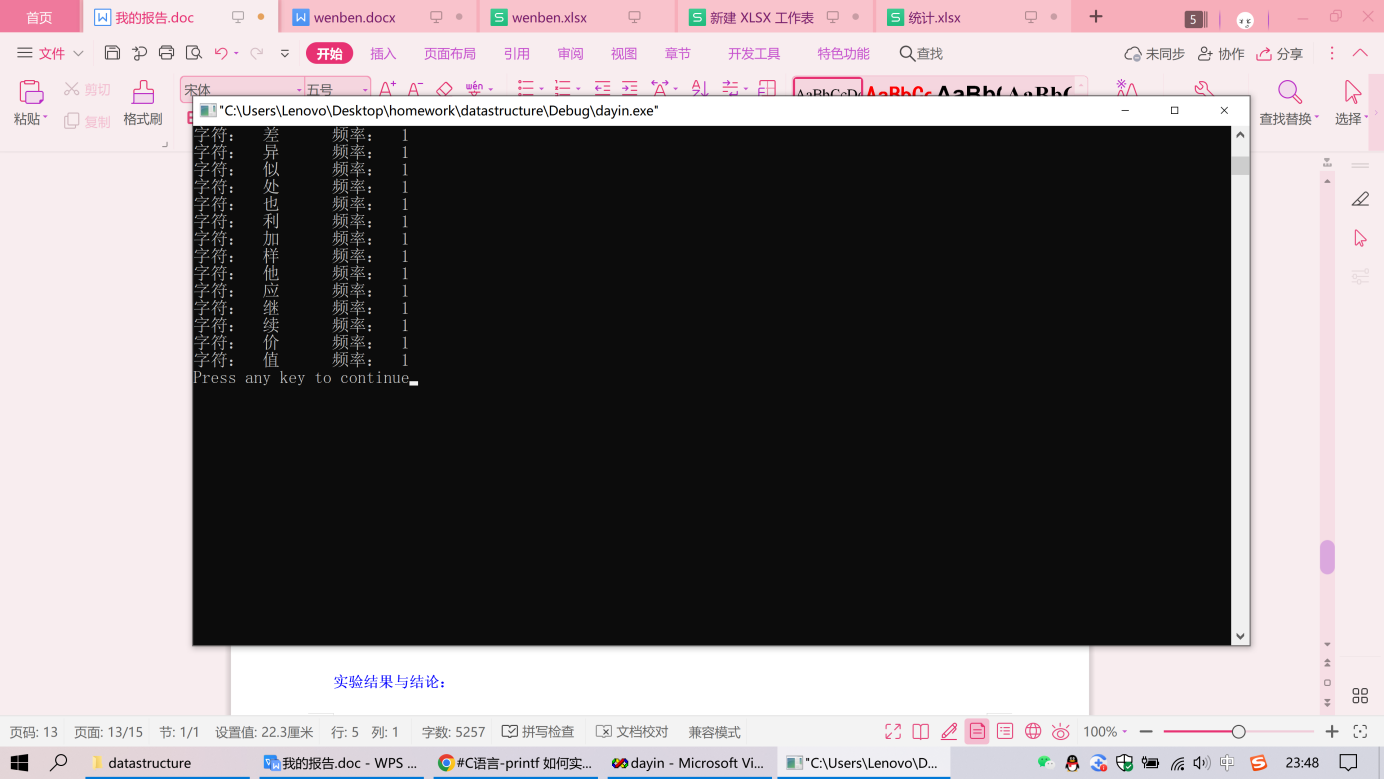












通过运行方法一中的代码，可以从wenben.txt文档中读取字符，并将所读取的字符按照频率进行排序，可显示出以上结果。

**不同方法对比总结：**

在方法一中通过运行上述代码，成功地实现了对链表的创建、插入、更新和排序操作。通过读取文件中的字符，成功地构建了一个包含字符频率信息的链表。冒泡排序算法成功地对链表进行了排序，按照字符的频率从低到高进行了排列。成功地打印了排序后的链表，显示了每个字符及其对应的频率。

方法一中整段代码时间复杂度取决于冒泡排序法，所以为为O(n2)；空间复杂度取决于最多为每个字符分配一个结点，所以为O(n)。

在方法二中通过运行上述代码，成功地实现了链表和哈希表的创建，可完成对字符的插入和更新和排序操作，通过调用函数打印出排序后的频率。

方法二中整段代码时间复杂度取决于printHashTable函数，函数遍历哈希表中的每个位置：O(m)（m 是哈希表中的结点数量），对于每个位置，遍历链表打印结点：O(n)（n 是该位置的链表长度）。因此，整段代码时间复杂度为 O(mn)。空间复杂度同样取决于printHashTable函数，insert 函数具有最佳的时间复杂度（O(1)），而 search 和 printHashTable 函数的时间复杂度与哈希表中的结点数量有关（分别为 O(n) 和 O(mn)）。在空间复杂度方面，通常与哈希表中的结点数量有关（O(m) 和 O(mn)）。因此，整段代码空间复杂度为 O(mn)。

对比方法一和方法二发现，方法一的逻辑相对于方法二来讲比较简单高效，方法一和方法二的代码在输入输出数据和运算数据方面的时间复杂度是相同的。在空间复杂度方面来看，方法一在处理非常大的文件时会消耗大量内存，方法二使用哈希表可能会节省存储空间，但定义哈希表大小时需要注意文本的大小，以免数据处理空间不足。

两种方法都可以进行汉字频率排序，从而达到本次实验的目的。

南京农业大学课程考核情况表

2023 —2024 学年第 一 学期

学 院： 信息管理学院

课程名称： 数据结构实验 学分： 2

学号姓名： 19122120侯兆晗

考核方式：

考核内容：

评 语：

成绩：

教师签名：

时间：