**项目说明文档**

**数据结构课程设计**

**——关键字检索系统**

作 者 姓 名 刘淑仪

学 号 2251730

指 导 教 师 张 颖

学 院 专 业 软件学院 软件工程



二〇二三 年 十 月 二十五 日

目录

1 项目分析 1

1.1 项目背景分析 1

1.2 项目功能分析 1

2 项目设计 2

2.1 数据结构设计 2

2.2 类结构设计 3

2.3 成员与操作设计 4

2.4 系统设计 7

3 功能设计与项目实现 8

3.1 输入文本文件名和关键字功能的实现 8

3.1.1 输入文本文件名和关键字功能实现思路 8

3.1.2 输入文本文件名和关键字功能核心代码 8

3.1.3 } 输入文本文件名和关键字功能实现示例 9

3.2 输入和追加文本文件内容的实现 9

3.2.1 输入和追加文本文件内容实现思路 9

3.2.2 输入和追加文本文件内容核心代码 9

3.2.3 输入和追加文本文件内容实现示例 10

3.3 清空文本文件内容的实现 10

3.3.1 清空文本文件内容实现思路 10

3.3.2 清空文本文件内容核心代码 10

3.3.3 清空文本文件内容实现示例 11

3.4 查看文本文件内容的实现 12

3.4.1 查看文本文件内容实现思路 12

3.4.2 查看文本文件内容核心代码 12

3.4.3 查看文本文件内容实现示例 13

3.5 关键字检索算法的实现 13

3.5.1 关键字检索算法实现思路 13

3.5.1.1 BF算法 13

3.5.1.2 KMP算法 14

3.5.2 关键字检索算法核心代码 14

3.5.3 关键字检索算法实现示例 16

3.6 本项目涉及LString.h内容的主要实现 16

3.5.1 节点结构体和串类的定义 16

3.5.2 LString.h核心代码 17

4 项目测试 17

4.1 功能测试 17

4.1.1 基本功能测试 17

4.1.2 文件已存在且不为空测试 17

4.1.3 文件已存在且为空测试 18

4.1.4 关键词不存在测试 19

4.1.5 长/短文章测试 19

4.2 拓展测试 21

4.2.1 优化中文功能 21

4.3 Linux环境测试 22

5 集成开发环境与编译运行环境 22

# 项目分析

## 项目背景分析

在日常工作和学习中，我们经常需要处理文本文件，并需要查找其中特定关键字的出现次数。这可以是文本文档中的某个词汇、关键术语或其他信息。手动查找关键字在大量文本中的出现次数费时费力，容易出错，因此需要一个自动化的工具来完成这项任务。

项目的主要功能是创建、编辑和查看文本文件，并提供了关键字搜索的功能。用户可以输入文件名，将文本内容追加到文件中，查看文件内容，清空文件内容。用户可以输入关键字，程序会统计并输出关键字在文本文件中的出现次数。

项目使用了两种不同的算法来实现关键字搜索：BF算法和KMP算法。BF算法是一种简单但效率较低的算法，适用于小型文本文件。KMP算法是一种高效的字符串匹配算法，适用于大型文本文件，它通过部分匹配表提高了搜索效率。

这个项目可以应用于多种场景，包括文档编辑、文本分析、日志文件分析等领域。在文档编辑中，用户可以追踪特定关键字的使用情况。在文本分析中，可以统计某个词汇在大量文本数据中的频率。在日志文件分析中，可以查找特定事件或错误信息在日志文件中的出现次数。

## 项目功能分析

1. 文件处理功能

用户通过键盘输入文件名，程序创建一个文本文件或打开一个已存在的文本文件。用户可以将文本内容追加到文件中。用户可以查看文件的内容。用户可以选择清空文件内容。

1. 关键字搜索功能

用户通过键盘输入一个不含空格的关键字，用于搜索。使用BF（暴力搜索）算法实现关键字搜索。使用KMP算法（Knuth-Morris-Pratt算法）实现关键字搜索，以提高搜索效率。统计并输出关键字在文本文件中的出现次数。

1. 算法功能分析

·BF（暴力搜索）算法：

该算法用于对给定的文本内容中搜索关键字的出现次数。算法会逐个字符比较文本内容与关键字，当匹配失败时，会从下一个字符继续比较。如果找到了一个完全匹配的关键字，就增加计数器。该算法的时间复杂度为O(n \* m)，其中n是文本内容的长度，m是关键字的长度。可视化函数负责将迷宫的当前状态以文本形式显示出来，包括墙壁、路径、起点、终点和已探索的路径。在找到路径后，路径跟踪函数将打印出构成路径的坐标序列。

·KMP算法（Knuth-Morris-Pratt算法）

该算法用于改进关键字搜索的效率，特别适用于长文本中的多次搜索。首先构建部分匹配表（Next数组），用于在匹配失败时快速确定下一次比较的位置。算法会根据部分匹配表，在匹配失败时智能地滑动文本内容和关键字，以跳过已经匹配的部分。如果找到了一个完全匹配的关键字，就增加计数器。该算法的时间复杂度为O(n + m)，其中n是文本内容的长度，m是关键字的长度。

# 项目设计

## 数据结构设计

1. 文本文件和关键字的数据结构：

AString类：这是一个自定义的字符串类，用于处理文本文件名和关键字。它包含了存储字符串内容的char\* ch、int curLength用于记录字符串实际长度、int maxSize用于存放数组的最大长度等成员。这个类允许用户输入文件名和关键字，并进行字符串操作，如复制、连接等。

1. 文件处理和关键字搜索的数据结构：

std::ofstream ofs和std::ifstream file：这两个标准库的文件流用于创建、打开和操作文本文件。AString txtContent和AString txtFile：这两个字符串对象用于存储文本文件的内容，以及用户输入的关键字。AString KeyWord：用于存储用户输入的关键字。char\* inputLine：用于临时存储用户输入的文本内容。int next[maxTxtSize]：用于KMP算法中构建部分匹配表的数组。

1. 进行关键字搜索的算法数据结构：

countBF函数：用于BF算法的关键字搜索，通过比较文本内容和关键字的字符进行搜索，返回出现次数。Nextval函数和countKMP函数：用于KMP算法的关键字搜索，通过构建部分匹配表和快速滑动进行搜索，返回出现次数。

1. 用户输入和程序运行的数据结构：

int maxInputSize和int maxTxtSize：定义了用户输入的最大长度和文本文件的最大长度。枚举类型Status：用于表示函数结果状态代码，如OK、ERROR等。void Keymain()函数：项目的主要函数，用于执行文件处理和关键字搜索操作。

## 类结构设计

KeywordSolution类的设计充分体现了关键字搜索项目的实用性和教育价值，同时展示了字符串处理算法的实际应用。该类通过模块化的方式和高效的数据结构实现了用户友好的文本文件处理和关键字搜索功能。这种设计使得关键字搜索项目易于使用和理解，同时在代码的可维护性、扩展性和搜索效率方面表现出色，为处理文本数据提供了一个强大的工具。

在主要cpp文件中：

创建KeywordSolution类：用于存储AString实例和文件操作相关的参数作为私有成员，例如文本文件名、文本内容等。公有成员包括主要的项目功能函数，如Keymain()，以及其他辅助函数。文件处理和关键字搜索的功能函数：包括inputTxt()函数用于输入文本内容和关键字，ContainTxt()函数用于向文件追加文本内容，OutputTxt()函数用于查看文件内容，searchKeyword()函数用于搜索关键字，clearFile()函数用于清空文件内容，Exit()函数用于退出程序。关键字搜索的算法函数：包括countBF()函数用于BF算法的关键字搜索，Nextval()函数和countKMP()函数用于KMP算法的关键字搜索。

在LString.h中：

AString类：用于处理文本文件名和关键字。该类包含了存储字符串内容的char\* ch、int curLength用于记录字符串实际长度、int maxSize用于存放数组的最大长度等成员。它允许用户输入文件名和关键字，并进行字符串操作，如复制、连接等。

## 成员与操作设计

未注释的函数请见下文实现对应功能时代码的函数标签。

LString.h主要用于实现串：

class AString {

private:

char\* ch; //串存放数组

int curLength; //串的实际长度

int maxSize; //存放数组的最大长度

public:

AString(int sz = defaultSize); //构造函数，构造一个最大长度为sz，实际长度为0的字符串

AString(const char\* init); //构造函数，构造一个最大长度为maxSize，由init初始化的新字符串对象

AString(const AString& ob); //复制构造函数，由一个已有的字符串对象ob构造一个新的字符串

~AString() { delete[]ch; } //析构函数，释放动态分配的串空间并撤销该字符串对象

int Length()const { return curLength; } //返回串的实际长度

void setString(const char\* inputLine); //将字符串里的数据传入类里

void copyTo(char\* target, int targetSize) const;

//将串的内容复制到字符数组里

void clear() {

ch[0] = '\0'; // 设置第一个字符为终止字符，这将创建一个空字符串

curLength = 0; // 更新当前长度为0

}

AString& operator()(int pos, int len); //当0<=pos<maxSize且0<=len且pos+len<maxSize的时候，

//在串从pos所指出的位置开始连续取len个字符组成子串返回

int operator == (AString& ob)const { return strcmp(ch, ob.ch) == 0; }

//判断串是否相等。1等，0不等

int operator != (AString& ob)const { return strcmp(ch, ob.ch) != 0; }

//判断串是否不等。1不等，0等

int operator !()const { return curLength == 0; }

//判断串是否空。1空，0不空

AString& operator=(AString& ob); //串ob赋值给当前串

AString& operator +=(const AString& ob); //若合法，将串ob接在当前串后面

char& operator[](int i); //取当前串第i个字符

const char\* getString() const {return ch;} //得到字符串内容

// 将一个字符追加到字符串末尾

void append(char c) {

// 检查是否有足够的空间添加新字符

if (curLength + 1 < maxSize) {

ch[curLength] = c; // 将字符追加到末尾

curLength++; // 增加当前长度

ch[curLength] = '\0'; // 确保字符串以null字符结尾

}

}

friend int countBF(AString S, AString T);

friend void Nextval(const AString pattern, int next[]);

friend int countKMP(const AString text, const AString pattern);

friend std::ostream& operator<<(std::ostream& os, const AString& str);

};

主cpp用于实现关键字检索类以及BF、KMP算法：

class KeywordSolution {

private:

std::ofstream ofs;

std::ifstream file;

AString txtName;

AString txtContent;

AString txtFile;

AString KeyWord;

public:

void Keymain();

void menu();

void inputTxt(AString& file, int opt);

Status ContainTxt();

Status isExist();

Status OutputTxt();

Status searchKeyword();

Status clearFile();

void Exit() {

std::cout << "按任意键继续...\n";

// 清除输入缓冲区

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

// 等待用户按键

std::cin.get();

}

};

// 串的模式匹配算法

/\* ---- BF算法

将主串的第pos个字符和模式串的第一个字符比较，

若相等，则继续逐个比较后续字符；

若不等，从主串的下一字符起，重新与模式串的第一个字符比较 \*/

int countBF(AString pattern,AString text) {

int count = 0; // 子串出现的次数

int i = 0, j = 0;

while (i <= text.curLength - pattern.curLength) {

if (j < pattern.curLength && text.ch[i + j] == pattern.ch[j]) {

j++; // 继续匹配下一个字符

}

else {

if (j == pattern.curLength) {

count++; // 完全匹配，增加计数器

}

i++;

j = 0; // 重置子串索引，从主串的下一个字符开始新的匹配

}

}

return count; // 返回子串在主串中出现的总次数

}

/\* ---- KMP算法

利用已经部分匹配的结果而加快模式串的滑动速度

且主串的指针i不必回溯，可以提速到O(n+m)\*/

// 构造部分匹配表

void Nextval(const AString pattern, int next[])

{

int len = pattern.curLength;

next[0] = -1;//p的起始下标为0，所以这里设置为-1

int k = -1, j = 0;

while (j < len) {

if (k == -1 || pattern.ch[k] == pattern.ch[j]) {

j++;

k++;

next[j] = k;

}

else

k = next[k];

}

}

// KMP 算法

int countKMP(const AString text, const AString pattern) {

int next[maxTxtSize];

int lenText = text.curLength;

int lenPattern = pattern.curLength;

int i = 0, n = 0, k = 0;

Nextval(pattern, next);

while (i < lenText) {

if (k == -1 || text.ch[i] == pattern.ch[k]) {

++i;

++k;

}

else

k = next[k];

if (k == lenPattern) {

n++;

k = next[k];

}

}

return n;

}

## 系统设计

模块化：代码被组织成不同的函数，每个函数负责特定的任务，如初始化、生成迷宫、搜索路径和用户交互，使得代码易于理解和维护。

数据结构的选择：项目中使用了自定义的字符串类AString，它包含了字符数组、长度信息以及其他操作字符串所需的成员变量，这种数据结构在处理文本内容和关键字时非常灵活。为了文件处理，使用了C++标准库提供的文件流，如std::ofstream和std::ifstream，这些数据结构使得文件的读写操作更加便捷。

算法的选择：项目使用了两种不同的搜索算法，分别是BF算法和KMP算法。BF算法简单但效率较低，适用于小型文本文件；而KMP算法则通过构建部分匹配表提高了搜索效率，适用于大型文本文件。这种算法选择允许用户在不同场景下根据文本文件的大小和需求来选择合适的搜索方式。

错误处理机制：项目考虑了可能出现的错误情况，如文件打开失败、输入无效等，通过返回适当的错误状态代码来提示用户或进行错误处理。

扩展性和可维护性：代码采用了良好的命名规范和注释，使得代码易于阅读和维护。模块化的设计和数据结构的选择使得项目可以轻松扩展，例如添加新的搜索算法或改进用户界面。

效率考虑：项目中的KMP算法通过构建部分匹配表提高了搜索效率，尤其适用于处理大型文本文件。这种算法选择使得关键字搜索更加高效。

# 功能设计与项目实现

## 输入文本文件名和关键字功能的实现

### 输入文本文件名和关键字功能实现思路

这段代码是实现了用户文本输入的功能，其核心思路如下：首先，进入一个无限循环，允许用户多次输入文本。根据传入的参数 opt 的不同，确定了输入的最大长度 MaxSize，以适应不同情境下的输入需求。接着，代码动态分配了一个字符数组 inputLine，用于临时存储用户的文本输入。然后，使用 cin.getline(inputLine, MaxSize) 从标准输入流获取用户输入的文本，这个函数能够限制输入的长度不超过 MaxSize。获取到用户输入后，将其赋值给 AString 类对象 file，由 AString 类来管理这个字符串。最后，释放之前动态分配的内存，防止内存泄漏，并通过 break 退出循环，完成一次输入操作。

### 输入文本文件名和关键字功能核心代码

/\*

\* Function name : inputTxt

\* Function features : Enter the file

\* Input parameters : char\* file, int opt

\*/

void KeywordSolution::inputTxt(AString& file, int opt)

{

while (true) {

int MaxSize = 0;

if (!opt)

MaxSize = maxInputSize;

else

MaxSize = maxTxtSize;

/\* 创建一个字符串存储输入 \*/

char\* inputLine = nullptr;

inputLine = new char[MaxSize];

/\* 获取输入 \*/

cin.getline(inputLine, MaxSize);

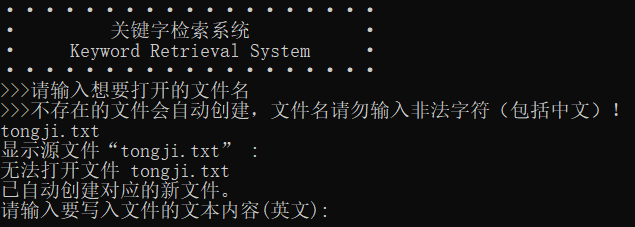
file.setString(inputLine);

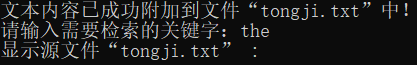
delete[] inputLine;

break;

}

### } 输入文本文件名和关键字功能实现示例





## 输入和追加文本文件内容的实现

### 输入和追加文本文件内容实现思路

这段代码实现了将用户输入的文本内容附加到指定的文本文件中的功能。首先，它通过打开文件以附加内容的方式来确保不会覆盖已有的文件内容，同时处理了文件打开失败的情况并返回错误状态。接着，它提示用户输入要写入文件的文本内容，并将用户输入的内容存储在名为txtContent的AString对象中。随后，将这个文本内容追加到已打开的文件中，并在每次追加后添加一个换行符，以确保每次写入的内容都位于新的一行。最后，关闭文件，释放相关资源，然后输出成功附加文本内容的提示信息，告知用户操作已完成。这段代码的实现思路清晰，通过AString对象管理文本内容，使用文件流进行文件操作，使得用户能够轻松将文本内容追加到指定文件中，为关键字检索系统的核心功能提供了文本内容的输入和管理支持。

### 输入和追加文本文件内容核心代码

/\*

\* Function name : ContainTxt

\* Function features : Add the input content to the attachment file.

\*/

Status KeywordSolution::ContainTxt()

{

char\* input = new char[maxInputSize];

txtName.copyTo(input, txtName.Length() + 1);

ofs.open(input, std::ios::app); // 打开文件以附加内容

if (!ofs.is\_open()) {

std::cerr << "无法打开文件 " << txtName << std::endl;

return ERROR; // 退出程序，发生错误

}

std::cout << "请输入要写入文件的文本内容(英文):\n";

inputTxt(txtContent, 1);

ofs << txtContent << std::endl; // 将文本内容写入文件

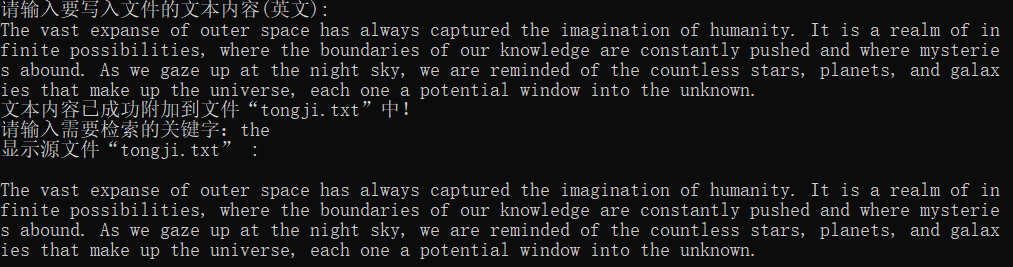
ofs.close(); // 关闭文件

std::cout << "文本内容已成功附加到文件“" << txtName << "”中！" << std::endl;

return OK;

}

### 输入和追加文本文件内容实现示例



## 清空文本文件内容的实现

### 3.3.1 清空文本文件内容实现思路

首先，代码创建了一个字符指针 input，并将文件名 txtName 的内容复制到 input 中，这是为了后续打开文件使用。

接着，使用 std::ofstream 类创建一个文件输出流 clearStream，并以输出（out）和截断（trunc）的模式打开文件，这意味着文件将被打开用于写入，并且在打开前将文件内容截断为空，即清空文件。

然后，代码检查文件是否成功打开，如果成功打开文件，则执行以下操作：关闭文件流，释放文件资源，确保文件操作的完整性；输出提示信息，告知用户文件已成功清空；返回操作成功的状态 OK。

如果无法成功打开文件，代码将输出错误信息，告知用户清空文件操作失败，并返回错误状态 ERROR。

### 3.3.2 清空文本文件内容核心代码

/\*

\* Function name : clearFile

\* Function features : Clear the content of the file.

\*/

Status KeywordSolution::clearFile()

{

char\* input = new char[maxInputSize];

txtName.copyTo(input, txtName.Length()+1);

std::ofstream clearStream(input, std::ofstream::out | std::ofstream::trunc);

if (clearStream.is\_open()) {

clearStream.close();

std::cout << "文件 " << txtName << " 已清空！" << std::endl;

return OK;

}

else {

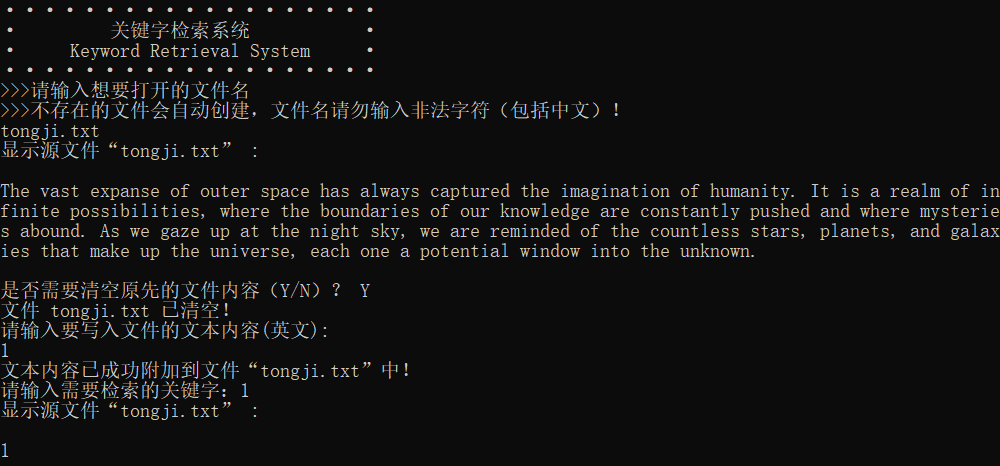
std::cerr << "无法清空文件 " << txtName << std::endl;

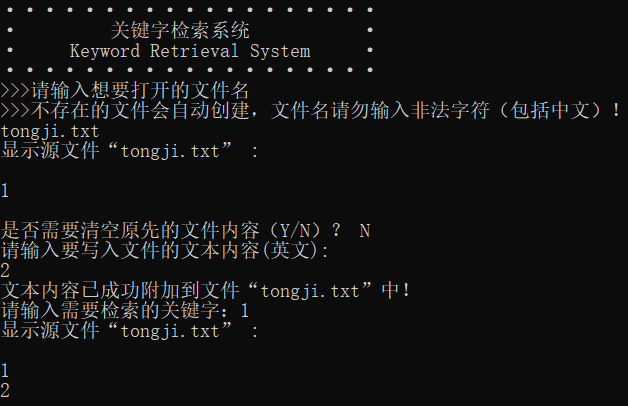
return ERROR;

}

}

### 3.3.3 清空文本文件内容实现示例





## 查看文本文件内容的实现

### 3.4.1 查看文本文件内容实现思路

首先，代码调用 txtFile.clear() 来清空存储文本文件内容的 txtFile 对象，以确保之前的内容被清空。输出提示信息，告知用户将要显示的源文件名称，这里使用 txtName.getString() 获取文件名。调用 isExist() 函数来检查文件是否存在，如果文件不存在，则输出提示信息说明已自动创建对应的新文件，并返回错误状态 ERROR。

如果文件存在，代码接下来检查文件是否为空，这是通过检查 file.get() == std::ifstream::traits\_type::eof() 是否为真来实现的。如果文件为空，输出相应提示信息，并关闭文件，最后返回错误状态 ERROR。

如果文件既存在又非空，代码将进入一个循环，使用 file.get(tmp) 逐字符读取文件内容，并将读取的字符追加到 txtFile 对象中，同时输出到控制台，以显示文件内容。

循环完成后，输出一个额外的换行符以分隔文件内容输出，并关闭文件。。

### 3.4.2 查看文本文件内容核心代码

/\*

\* Function name : OutputTxt

\* Function features : Output the file content.

\*/

Status KeywordSolution::OutputTxt()

{

txtFile.clear();

std::cout << "显示源文件“" << txtName.getString() << "” : \n";

if (!isExist()) {

std::cout << "已自动创建对应的新文件。\n";

return ERROR;

}

if (file.get() == std::ifstream::traits\_type::eof()) {

std::cout << "源文件为空" << std::endl;

file.close();

return ERROR;

}

else {

std::cout << '\n';

file.seekg(0, std::ios::beg);

char tmp;

while (file.get(tmp)) {

txtFile.append(tmp);

std::cout << tmp;

}

std::cout << "\n";

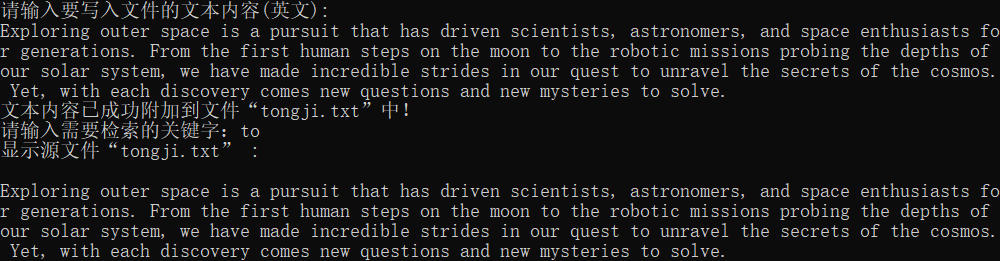
file.close(); // 关闭文件

return OK;

}

}

### 3.4.3 查看文本文件内容实现示例



## 关键字检索算法的实现

### 关键字检索算法实现思路

#### BF算法

首先，定义两个输入参数：AString pattern 表示模式串，即要搜索的关键字，以及 AString text 表示主串，即要在其中搜索关键字的文本。创建一个整数变量 count 用于记录模式串在主串中出现的次数，初始化为0。使用两个循环变量 i 和 j 来遍历主串和模式串。外层循环控制主串的遍历，从0到主串长度减去模式串长度，以保证不越界。内层循环用于比较主串和模式串中的字符。

在每次循环迭代中，首先检查 j 是否小于模式串的长度，并且检查当前主串位置 text.ch[i + j] 是否等于模式串位置 pattern.ch[j] 处的字符。如果相等，表示找到了一个匹配字符，将 j 自增，继续匹配下一个字符。

如果不相等，就意味着在当前位置字符不匹配，此时需要回溯。如果 j 正好等于模式串的长度，表示已经完全匹配，因此增加 count 计数器，然后将 i 向后移动一个位置，同时将 j 重置为0，开始新的匹配。

重复以上步骤，直到遍历完主串中的所有可能位置。最后，返回 count，即模式串在主串中出现的总次数。

#### KMP算法

首先，定义了两个函数：Nextval 用于构造部分匹配表，即计算模式串每个位置的最大匹配长度；countKMP 用于执行KMP算法的模式匹配操作。

在 Nextval 函数中，首先获取模式串的长度 len，初始化 next 数组。然后，进入循环，j 表示当前要计算的位置，k 表示前一个位置的最大匹配长度。在循环中，首先检查 k 是否等于-1 或者当前位置的模式串字符 pattern.ch[k] 是否等于当前位置的字符 pattern.ch[j]。如果相等，表示找到了一个匹配，将 j 和 k 同时递增，并将 next[j] 设置为 k。如果不相等，说明当前位置字符不匹配，此时需要回溯。将 k 更新为 next[k]，即回到前一个位置的最大匹配长度。继续循环，重复上述步骤，直到计算完整个 next 数组，这个数组存储了每个位置的最大匹配长度。

在 countKMP 函数中，首先创建 next 数组并初始化一些参数，包括主串长度 lenText，模式串长度 lenPattern，以及循环变量 i、n 和 k。调用 Nextval 函数来构造部分匹配表，并将结果存储在 next 数组中。进入主循环，循环变量 i 用于遍历主串。在每次循环迭代中，首先检查 k 是否等于-1 或者当前主串字符 text.ch[i] 是否等于模式串字符 pattern.ch[k]。如果相等，表示找到了一个匹配，将 i 和 k 同时递增。如果不相等，说明当前位置字符不匹配，此时根据部分匹配表回溯 k，将其更新为 next[k]。

每次回溯后，检查 k 是否等于模式串长度 lenPattern，如果相等，表示找到了一个完全匹配，将匹配次数 n 自增，并再次回溯 k。最后，返回匹配次数 n，即模式串在主串中出现的总次数。

### 关键字检索算法核心代码

// 串的模式匹配算法

/\* ---- BF算法

将主串的第pos个字符和模式串的第一个字符比较，

若相等，则继续逐个比较后续字符；

若不等，从主串的下一字符起，重新与模式串的第一个字符比较 \*/

int countBF(AString pattern,AString text) {

int count = 0; // 子串出现的次数

int i = 0, j = 0;

while (i <= text.curLength - pattern.curLength) {

if (j < pattern.curLength && text.ch[i + j] == pattern.ch[j]) {

j++; // 继续匹配下一个字符

}

else {

if (j == pattern.curLength) {

count++; // 完全匹配，增加计数器

}

i++;

j = 0; // 重置子串索引，从主串的下一个字符开始新的匹配

}

}

return count; // 返回子串在主串中出现的总次数

}

/\* ---- KMP算法

利用已经部分匹配的结果而加快模式串的滑动速度

且主串的指针i不必回溯，可以提速到O(n+m)\*/

// 构造部分匹配表

void Nextval(const AString pattern, int next[])

{

int len = pattern.curLength;

next[0] = -1;//p的起始下标为0，所以这里设置为-1

int k = -1, j = 0;

while (j < len) {

if (k == -1 || pattern.ch[k] == pattern.ch[j]) {

j++;

k++;

next[j] = k;

}

else

k = next[k];

}

}

// KMP 算法

int countKMP(const AString text, const AString pattern) {

int next[maxTxtSize];

int lenText = text.curLength;

int lenPattern = pattern.curLength;

int i = 0, n = 0, k = 0;

Nextval(pattern, next);

while (i < lenText) {

if (k == -1 || text.ch[i] == pattern.ch[k]) {

++i;

++k;

}

else

k = next[k];

if (k == lenPattern) {

n++;

k = next[k];

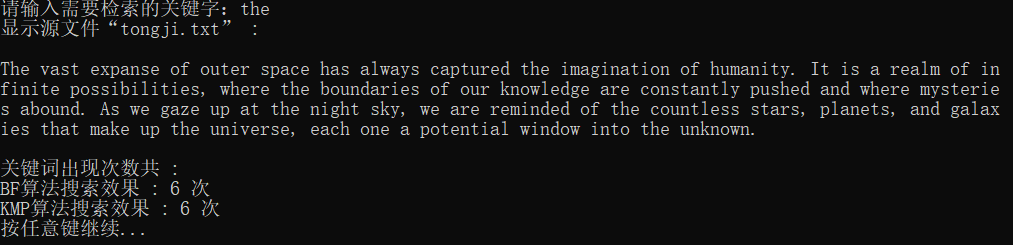
}

}

return n;

}

### 关键字检索算法实现示例



## 本项目涉及LString.h内容的主要实现

### 3.5.1 节点结构体和串类的定义

定义了一个名为 AString 的类，专门用于处理字符串。它通过私有成员变量存储字符串内容（char\* ch），记录当前字符串长度（curLength）和最大容量（maxSize）。类提供了多种构造函数，包括基于最大长度、现有 C 风格字符串和其他 AString 对象的初始化。析构函数负责释放分配的内存。类还包含多个成员函数，用于获取字符串长度、设置和复制字符串内容、清空字符串、提取子串、比较字符串是否相等或不等、判断字符串是否为空、赋值操作、字符串连接、访问特定字符和向字符串末尾追加字符。此外，还定义了几个友元函数，可能用于字符串的模式匹配和计数。整体上，AString 类提供了一套完整的工具，用于字符串的创建、存储、操作和析构，是一个字符串处理类。

### LString.h核心代码

详见类设计部分。

# 项目测试

## 功能测试

### 基本功能测试

**测试用例：**输入正常文件名->创建文件->输入关键词->查询出现次数

**用例说明：**正常功能测试

**预期结果：**程序应该创建文件并在文件中查询正确出现次数

**实验结果：**



### 4.1.2 文件已存在且不为空测试

**测试用例：**在已存在的文件中，测试清空后的搜索情况和不清空的搜索情况

**预期结果：**程序能正确运行，不崩溃

**实验结果：**





### 4.1.3 文件已存在且为空测试

**测试用例：**在已存在的文件中，内容为空，直接加入新文本再检索

**预期结果：**程序能正确运行，不崩溃

**实验结果：**

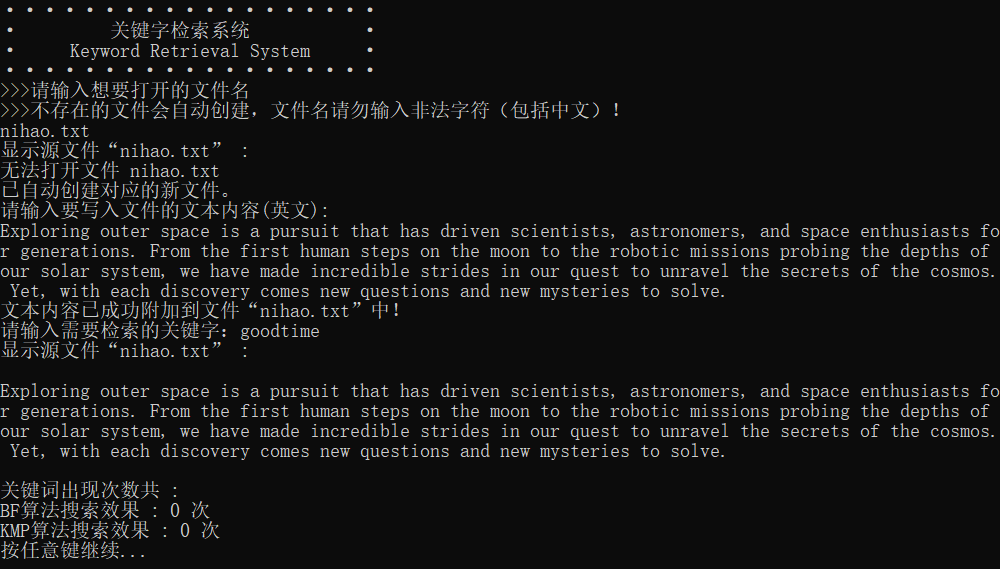


### 4.1.4 关键词不存在测试

**测试用例：**输入一个没有的词

**预期结果：**查询到的结果应该为0

**实验结果：**



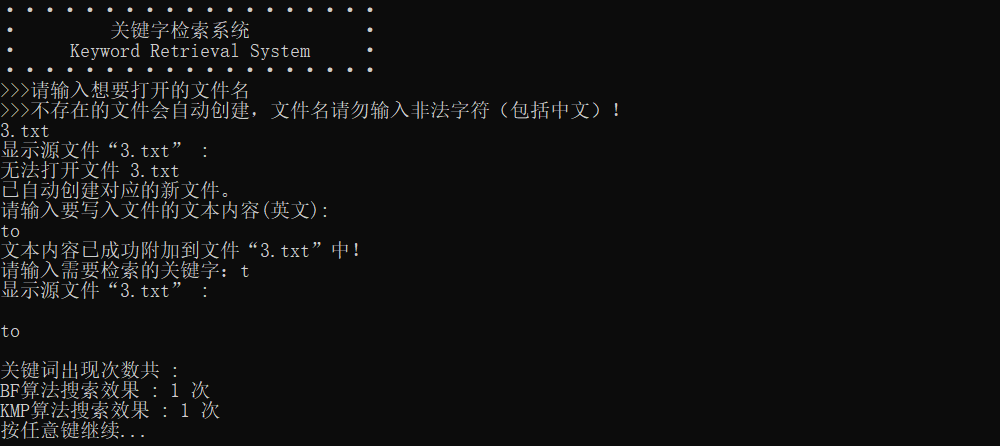
### 4.1.5 长/短文章测试

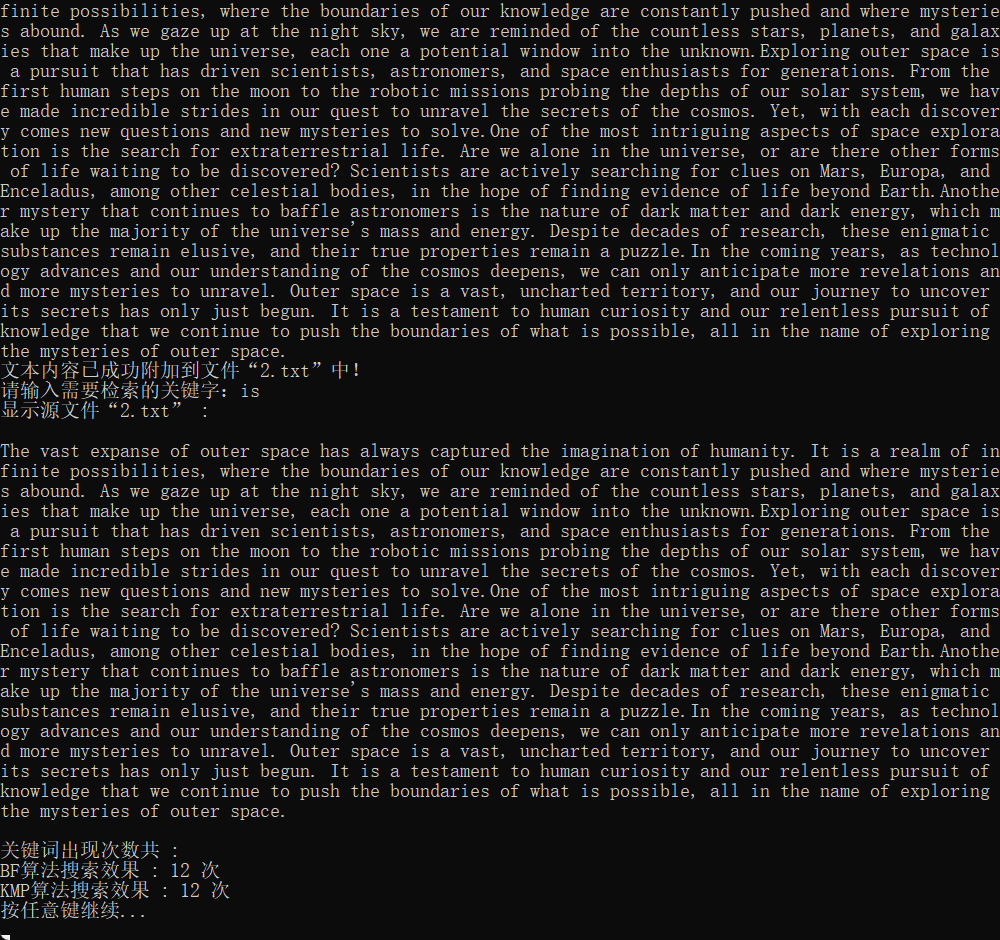
**测试用例：**输入长文章和短文章进行查询

**用例说明：**进行边界测试.

**预期结果：**程序应该在极端情况下也能正确运行，不崩溃

**实验结果：**





## 拓展测试

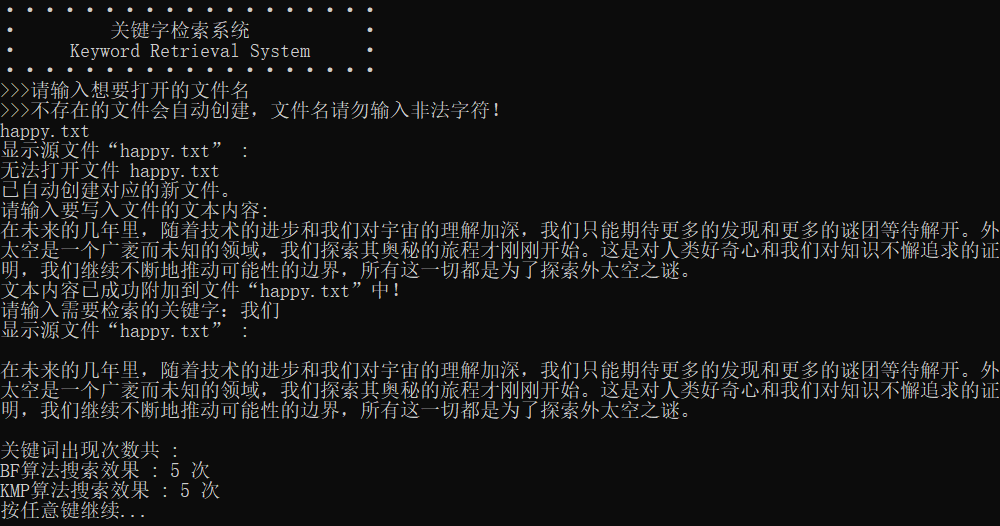
### 优化中文功能

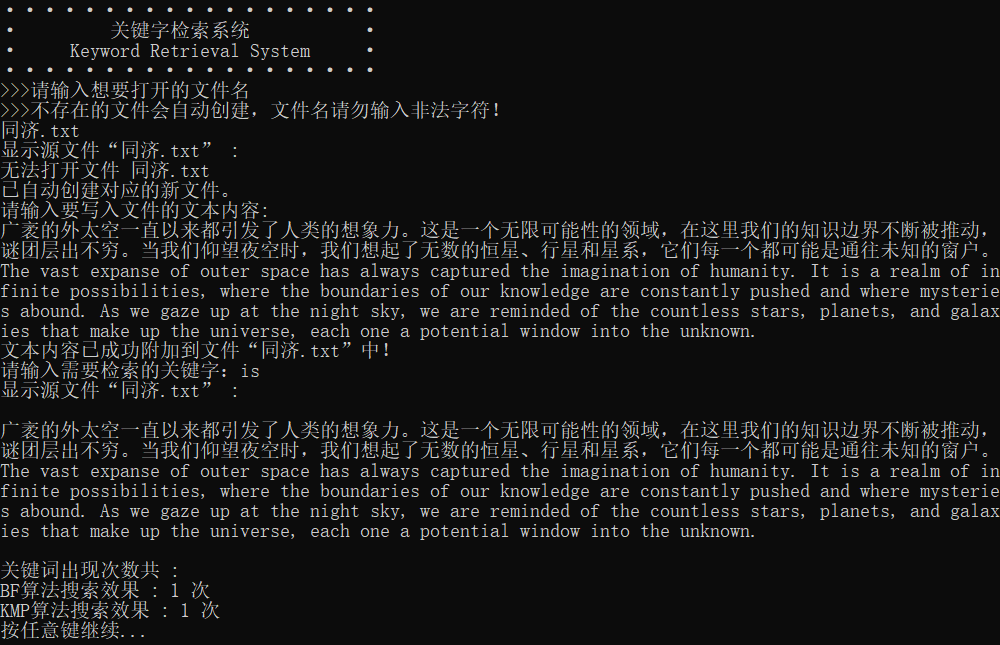
**测试用例：**将创建的文件名和插入的文件内容中夹杂中文

**用例说明：**创建中文名文件、添加中文内容，或者中英夹杂

**预期结果：**程序能正确运行，不崩溃

**实验结果：**



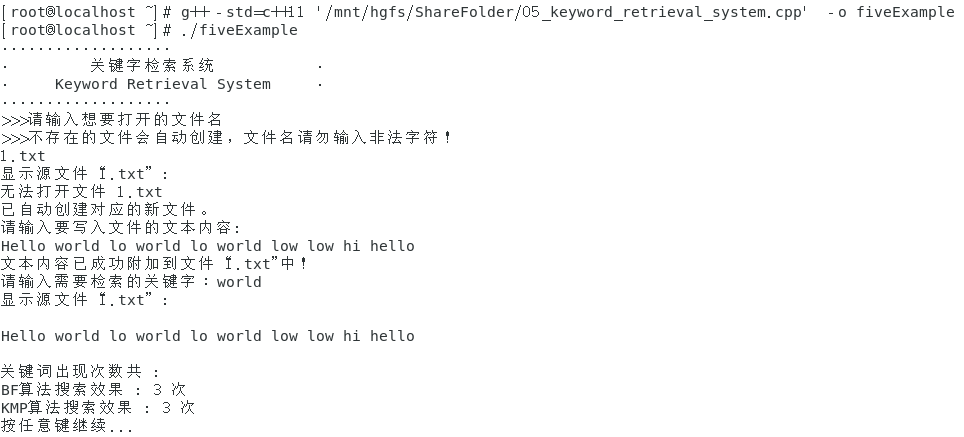


## Linux环境测试

**编译命令：** g++ -std=c++11 '/mnt/hgfs/ShareFolder/ 05\_keyword\_retrieval\_system' -o fiveExample

**运行命令：** ./fiveExample

**实验结果：**



1. **集成开发环境与编译运行环境**

Windows系统：Windows 11 x64

Windows集成开发环境：Microsoft Visual Studio 2022 (Release模式)

Windows编译运行环境：本项目适用于x86架构和x64架构

Linux系统：CentOS 7 x64