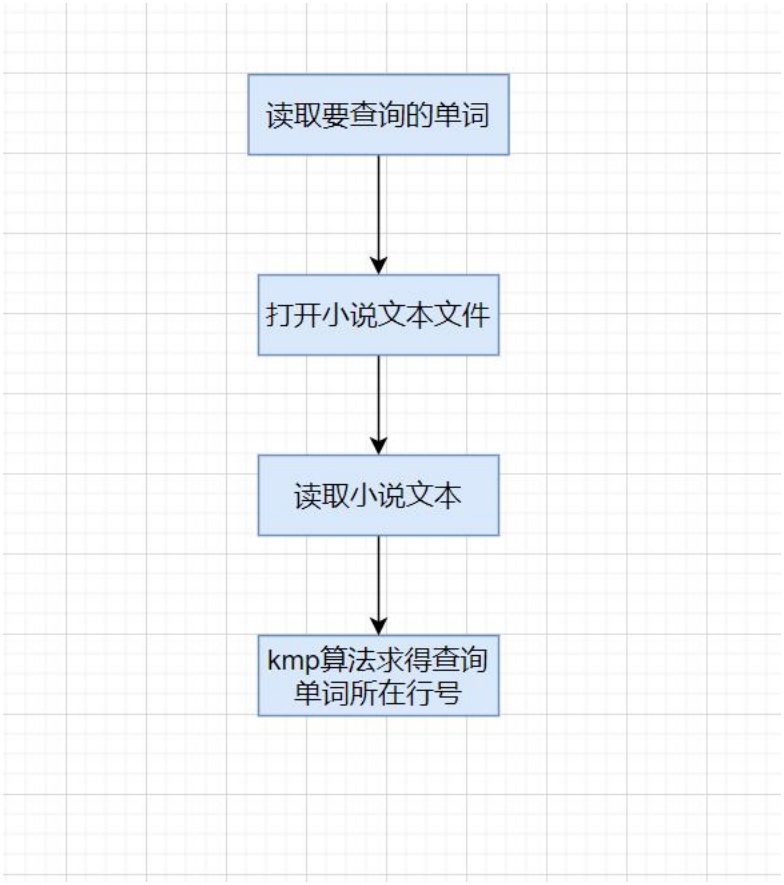
1. 题目分析

说明程序设计的任务，强调的是程序要做什么，此外列出各成员分工

- 任务:设计一个文字统计系统“文学研究助手”,统计文本文件中输入的词汇出现次数和位置
- 成员分工：

2. 数据结构设计

该程序主要用到的数据结构是动态数组，主程序的流程如下：



3. 程序设计

实现概要设计中的数据类型，对主程序、模块及主要操作写出伪代码，画出函数的调用关系

各模块伪代码如下：

获取单词数量：

```
# 获取单词数 word_count
get_word_count():
    word_count = input("Enter word count: ")
    return word_count
```

存储要查询的单词：

```
# 为 word_count 个单词分配内存,存储在 words 数组中
allocate_words(count):
    words = []
    for i in range(count):    # 循环读取每个单词
        word = input("Enter word: ")
        words.add(word)  # 添加到 words 数组
    return words
```

读取文本文件：

```
# 读取文本文件内容
read_text(file):
    text = ""
    line_numbers = []
    size = 100      # 初始化内存大小
    while !file.end():    # 读取文件直到末尾
        if file.next() == '\n':  # 如果是换行符
            line_count++         # 行数加 1
        else:
            line_numbers.add(line_count)  # 添加行号
            text += file.next()    # 添加字符
        if len(text) > size:    # 如果超过内存,扩容2倍并拷贝原数据
            size *= 2
            tmp = text
            text = ""
            for char in tmp: # 拷贝原数据
                text += char
```

KMP 算法：

```
# KMP 算法查找单词 word 在 text 中的行号,存储在 lines 数组
kmp_search(text, word, line_numbers):
    lines = []
    # KMP查找算法,省略细节
    if 找到匹配:
        lines.add(line_numbers[i - j])  # 添加匹配行号
    return lines
```

Main 函数：

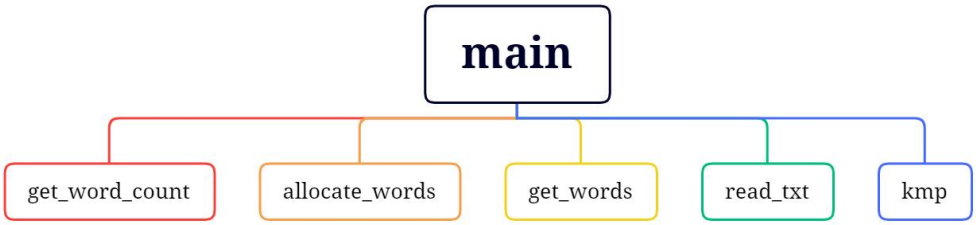
```
# 主函数
main():
    word_count = get_word_count()  # 获取单词数
    words = allocate_words(word_count)  # 分配内存存储单词
    get_words(words, word_count)      # 读取单词输入

    fp = open("novel.txt")           # 打开文本文件
    text, line_numbers = read_text(fp) # 读取文本内容和行号

    for word in words:
        lines = kmp_search(text, word, line_numbers) # KMP查找单词
        print(word, "exist in lines:", lines)      # 打印所在行号

    free(words)     # 释放words内存
    free(text)      # 释放text内存
    free(line_numbers) # 释放line_numbers内存
    close(fp)        # 关闭文本文件
```

各函数层次关系图：



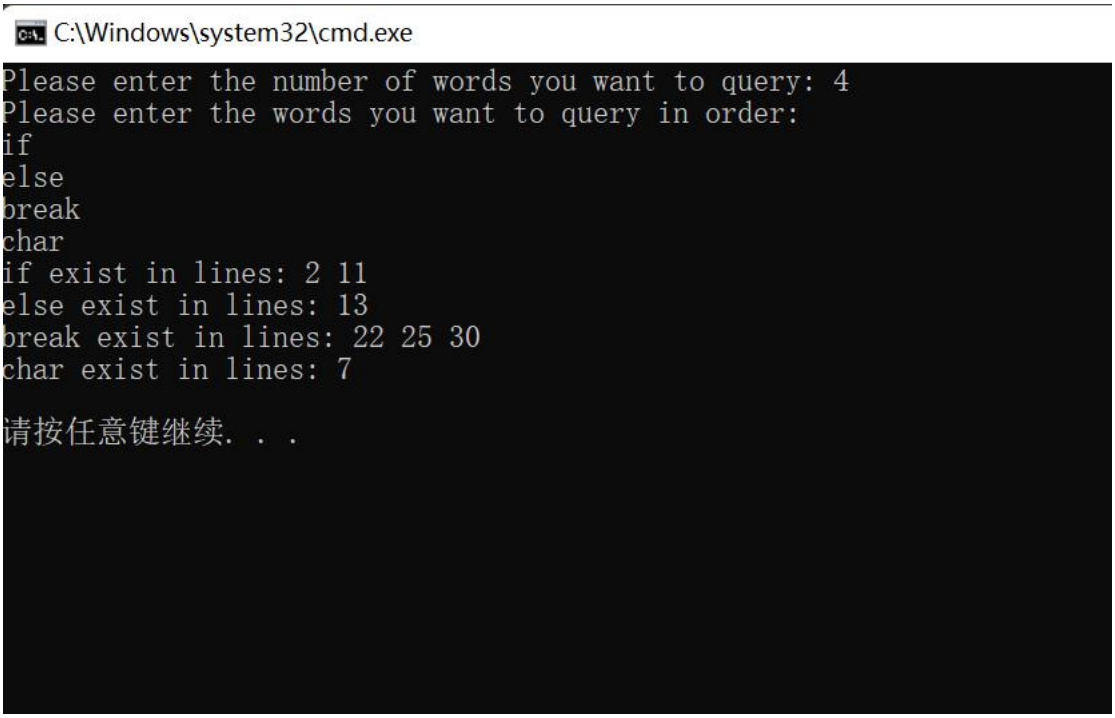
4. 调试分析

- 问题 1:内存管理,如何选择初始空间大小,扩容时机
- 解决:选取较小初始空间,当空间使用率过高时扩容,每次增加一倍空间
- 问题 2:如何让 KMP 算法在匹配成功后继续匹配而不重复记录行号
- 解决:记录最后匹配行号,仅当匹配行号改变时更新并打印

时间复杂度 O(文本长度 * 查询单词数量)，空间复杂度 O(文本长度 + 查询单词数量)

5. 测试结果

输入还包括 novel.txt，文本较长，可见附件



6. 用户使用说明

首先输入你想查询的单词的数量，然后一次输入你想要查询的单词，然后程序会打印查询的单词所在的行号

7. 选作内容

实现了(1) (2) 和 (4)

(1) 模式匹配要基于 KMP 算法。

(2) 整个统计过程中只对小说文字扫描一遍以提高效率。

(3) 假设小说中的每个单词或者从行首开始，或者前置以一个空格符。利用单词匹配特点另写一个高效的统计程序，与 KMP 算法统计程序进行效率比较。

(4) 推广到更一般的模式集匹配问题，并设待查模式串可以跨行。

8. 附录

literary_search_assistance.c 程序

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

// 获取需要查找的单词数
int get_word_count() {
    int word_count;
    scanf("%d", &word_count);
    return word_count;
}

// 为 word_count 个单词分配空间,并检查内存分配是否成功
char **allocate_words(int count) {
    char **words = (char **)malloc(sizeof(char *) * count);
    if (words == NULL) {
        printf("Memory allocation failed!\n");
        exit(1); // 内存分配失败则退出程序
    }
    return words;
}

// 读取 word_count 个单词到 words 中,并检查每个单词的内存分配是否成功
void get_words(char **words, int count) {
```

```
    for (int i = 0; i < count; ++i) {
        char *word = (char *)malloc(sizeof(char) * 50);
        if (word == NULL) {
            printf("Memory allocation failed!\n");
            exit(1); // 内存分配失败则退出程序
        }
        scanf("%s", word);
        words[i] = word;
    }
}

// 动态读取文本内容到*text 中,记录每行开始位置到*line_numbers
// 使用扩容机制,每次内存不足时多分配一倍的空间,直到文件末尾
void read_text(FILE *fp, char **text, int **line_numbers, int *char_count, int *line_count)
{
    int size = 100;
    *text = (char *)malloc(size * sizeof(char));
    *line_numbers = (int *)malloc(size * sizeof(int));
    if (*text == NULL || *line_numbers == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
```

```
// 读取文本内容
while (!feof(fp)) {
    // 扩容
    if (*char_count + 1 >= size) {
        size *= 2;
        char *tmp = (char *)malloc(size * sizeof(char));
        int *line_numbers_tmp = (int *)malloc(size * sizeof(int));
        if (tmp == NULL || line_numbers_tmp == NULL) {
            printf("Memory allocation failed!\n");
            exit(1);
        }
        strcpy(tmp, *text);
        free(*text);
        *text = tmp;
        memcpy(line_numbers_tmp, *line_numbers, (*char_count) * sizeof(int));
        free(*line_numbers);
        *line_numbers = line_numbers_tmp;
    }

    char next_char = fgetc(fp);
    if (next_char == '\n')
```

```
            (*line_count)++; // 记录行数
        else {
            (*line_numbers)[*char_count] = *line_count; // 记录当前行号
            (*text)[*char_count] = next_char;             // 添加字符
            (*char_count)++;                              // 统计字符数
        }
    }
}

// KMP 算法在 text 中查找 pattern,打印所在行号
void kmp(char *text, char *pattern, int *line_numbers, int char_count, int line_count) {
    int text_len = strlen(text);
    int pat_len = strlen(pattern);
    if (text_len < pat_len) {
        return;
    }
    int last_line = -1;
    int *next = (int *)malloc(pat_len * sizeof(int));
    next[0] = 0;
    // 构建 next 数组
    for (int i = 1, j = 0; i < pat_len;) {
        if (pattern[i] == pattern[j]) { // 如果当前字符匹配前缀
```

```
            j++;
            next[i] = j;
            i++; // 继续匹配下一个字符
        } else if (j > 0) {
            j = next[j - 1];
        } else {
            next[i] = 0;
            i++; // 继续匹配下一个字符
        }
    }

    // KMP 匹配算法
    for (int i = 0, j = 0; i < text_len;) {
        if (text[i] == pattern[j]) { // 如果当前字符匹配
            i++;                    // 文本下标右移
            j++;                    // 模式下标右移
        } else if (j != 0) {
            j = next[j - 1]; // 模式下标回退到 next[j-1]
        } else {
            i++; // 文本下标右移
        }
        if (j == pat_len) { // 如果找到匹配
```

```
            if (line_numbers[i - j] != last_line) {
                last_line = line_numbers[i - j];
                printf("%d ", last_line); // 打印行号
            }
        }
    }
    printf("\n");
    free(next); // 释放 next 数组空间
}

int main() {
    // 获取需要查找的单词数
    int word_count = get_word_count();

    // 为 word_count 个单词分配空间,并检查内存分配是否成功
    char **words = allocate_words(word_count);

    // 读取 word_count 个单词到 words 中,并检查每个单词的内存分配是否成功
    get_words(words, word_count);

    // 打开文本文件,检查文件打开是否成功
    FILE *fp = fopen("./novel.txt", "r");
```

```
if (fp == NULL) {
    printf("File open failed!\n");
    exit(1); // 文件打开失败则退出程序
}

// 定义变量,记录文本内容、文本长度、行号数组、字符数、行数
char *text;
int *line_numbers;
int char_count = 0;
int line_count = 1;

// 读取文本
read_text(fp, &text, &line_numbers, &char_count, &line_count);

// 查找每个单词并精准打印行号
for (int i = 0; i < word_count; ++i) {
    printf("%s exist in lines: ", words[i]);
    kmp(text, words[i], line_numbers, char_count, line_count);
}

// 释放所有内存空间,避免内存泄露
for (int i = 0; i < word_count; ++i)
```

```
        free(words[i]);
    }
    free(words);
    free(text);
    free(line_numbers);
}
```