一、必做题

题目：编程实现希尔、快速、堆排序、归并排序算法。要求随机产生 10000 个数据存入磁盘文件，然后读入数据文件，分别采用不同的排序方法进行排序，并将结果存入文件中。

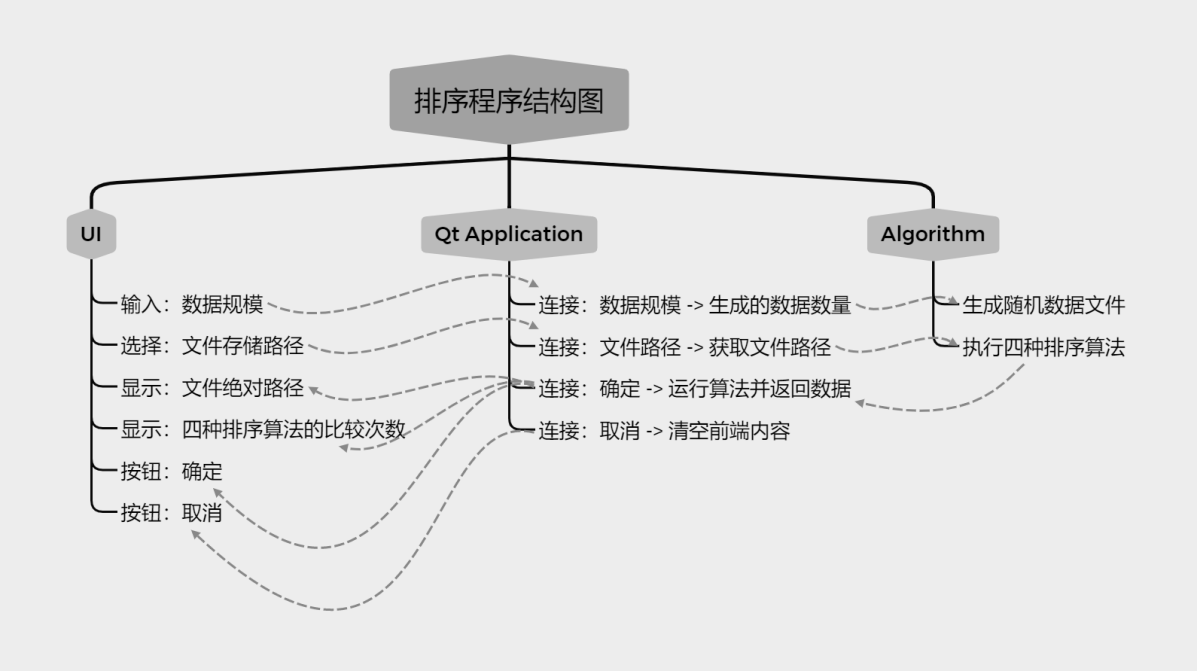
1.1 数据结构设计与算法思想

本程序涉及到四种排序算法，其中：

- 希尔排序：通过倍增方法逐渐扩大选择插入数据量的规模，而达到减小数据比较的次数，时间复杂度优化到近似 $O(n^{1.65})$
- 快速排序：通过分治的算法策略，不断确定每一个数的最终位置，从而达到近似 $O(n \log n)$ 的时间复杂度
- 堆排序：通过堆的树形结构，减小两数的比较次数，最终通过不断维护堆结构来获得有序序列，时间复杂度为 $O(n \log n)$
- 归并排序：通过分治排序回溯时左右两支有序的特点，进行归并，使算法整体的时间复杂度为固定的 $O(n \log n)$

1.2 程序结构

为了更好的了解排序算法内部的机制，我统计了每一种排序算法内部的比较次数,并结合了 Qt 的可视化框架进行编写。将四种排序作为算法内核嵌入 GUI 界面，程序结构如下：



UI

```
// 窗口 继承窗口库组件
<widget class="QWidget" name="SortWidget">
    // 标题 标签组件
    <widget class="QLabel" name="titleLabel"></widget>
    // 输入 网格布局组件
    <widget class="QGridLayout" name="inputGridLayout"></widget>
    // 输出 网格布局组件
    <widget class="QGridLayout" name="outputGridLayout"></widget>
    // 交互 水平布局组件
    <widget class="QHBoxLayout" name="buttonHLayout"></widget>
</widget>
```

Qt Application

```
#ifndef DATASTRUCTURECLASSDESIGN_SORTWIDGET_H
#define DATASTRUCTURECLASSDESIGN_SORTWIDGET_H

#include <QWidget>


QT_BEGIN_NAMESPACE
namespace Ui {
    class SortWidget;
}
QT_END_NAMESPACE

class SortWidget : public QWidget {
Q_OBJECT

private:
    Ui::SortWidget* ui;          // 窗口对象指针
    QString path;                // 文件存储路径

private slots:
    void pushFolderButton();     // 槽函数 - 触发事件：获取存储路径的窗口对话
    void pushCommitButton();     // 槽函数 - 触发事件：根据输入数据量执行算法
    void pushCancelButton();     // 槽函数 - 触发事件：清空窗口所有标签的数据

public:
    explicit SortWidget(QWidget* parent = nullptr); // 窗口构造函数

    ~SortWidget() override;                         // 窗口析构函数
};
```

Algorithm

```
#include <iostream>
#include <vector>
#include <fstream>

class sortAlgorithm {
private:
```

```
    int Size, Range;
    std::string Path;
    std::vector<int> arr;

    void Generate(int num, int range);                  // 数据生成

    int ShellSort(std::vector<int> a);                  // 希尔排序
    int QuickSort(std::vector<int> a);                  // 快速排序
    int HeapSort(std::vector<int> a);                   // 堆排序
    int MergeSort(std::vector<int> a);                  // 归并排序

    void WriteToFile(std::string path, std::vector<int>& a);    // 写入文件

public:
    sortAlgorithm(int _Size, int _Range, std::string _Path);    // 构造函数

    int ShellSort();                                    // 用户调用希尔排序
    int QuickSort();                                    // 用户调用快速排序
    int HeapSort();                                     // 用户调用堆排序
    int MergeSort();                                    // 用户调用归并排序
};
```

1.3 实验数据与测试结果分析

原始数据序列生成算法使用时间种子的除留余数法，在数据规模为 10000 的情况下，四种算法的比较次数如下：

	shell	heap	quick	merge
1	129419	231824	39061	120488
2	128051	232204	38723	120487
3	134873	231945	39052	120251
4	135719	232096	39158	120510
5	128821	232051	38747	120434
average	131376	232024	38948	120434

可以发现：在数据规模在 $1e4$ 的情况下，堆排序的比较次数最多，而快速排序的比较次数最少

1.4 程序清单

- `/Src/Algorithm/sortAlgorithm.cpp`：内核排序算法
- `/Src/Forms/SortWidget.ui`：ui 界面
- `/Src/Headers/SortWidget.h`：排序组件逻辑头文件
- `/Src/Post/SortWidget.cpp`：排序组件逻辑源文件
- `/Src/Test/TestSort.cpp`：程序测试文件

其中，除了 ui 界面使用 Qt Designer 设计师工具进行编写，其余文件均为与文件名同名的 C++ 类，便于维护与扩展

二、选做题

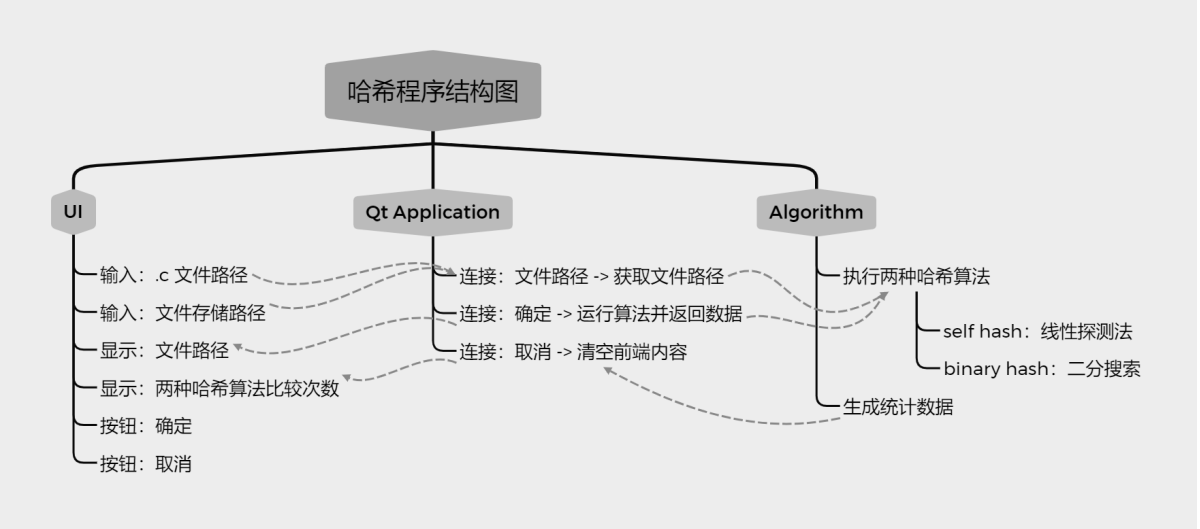
题三：扫描一个 C 源程序，利用两种方法统计该源程序中的关键字出现的频度，并比较各自查找的比较次数。

1. 用 Hash 表存储源程序中出现的关键字，利用 Hash 查找技术统计该程序中的关键字出现的频度。用线性探测法解决 Hash 冲突。设 Hash 函数为 Hash(key) = [(key 的第一个字母序号) × 100 + (key 的最后一个字母序号)] % 41
2. 用顺序表存储源程序中出现的关键字，利用二分查找技术统计该程序中的关键字出现的频度。

2.1 数据结构设计与算法思想

- 第一问：
  - 首先用关键字序列构造哈希表，哈希函数使用题中所给，容易算出哈希表的最大容量为 $25 \times 100 + 40$，我们设置为 3000，哈希表数据结构为：`int: <string, int>` 的键值对形式，可以使用动态数组容器 `std::vector<std::pair>` 进行存储。其中键设为 `int` 可以使得对于一个 `int` 类型的哈希值进行 $O(1)$ 的存储与查找，获得 `<word, cnt>`，即 `<std::string, int>`
  - 然后对于文件中的每一个单词采用哈希搜索。利用哈希函数计算出每一个单词的哈希值，然后通过线性探测法进行搜索比对。关键在于如何解析出一个完整的单词，对于流读入的字符串（不包含空格、换行符、制表符等空白符），我们删除其中的符号后将剩余部分进行合并，之后进行异常处理，排除空串后进行哈希查找。查找次数包含成功和失败的比较次数
- 第二问：将关键字存储于顺序表中，排序后，利用二分查找技术进行搜索统计。查找次数包含成功和失败的比较次数

2.2 程序结构



UI

```
// 窗口 继承窗口库组件
<widget class="QWidget" name="SortWidget">
    // 标题 标签组件
    <widget class="QLabel" name="titleLabel"></widget>
    // 输入 网格布局组件
    <widget class="QGridLayout" name="inputGridLayout"></widget>
    // 输出 网格布局组件
    <widget class="QGridLayout" name="outputGridLayout"></widget>
    // 交互 水平布局组件
    <widget class="QHBoxLayout" name="buttonHLayout"></widget>
</widget>
```

Qt Application

```
#ifndef DATASTRUCTURECLASSDESIGN_HASHWIDGET_H
#define DATASTRUCTURECLASSDESIGN_HASHWIDGET_H

#include <QWidget>


QT_BEGIN_NAMESPACE
namespace Ui {
    class HashWidget;
}
QT_END_NAMESPACE

class HashWidget : public QWidget {
Q_OBJECT

private:
    Ui::HashWidget* ui;                         // 窗口对象指针
    QString inputFilePath, outputFolderPath;  // 输入输出路径

private slots:
    void pushInputFileButton();     // 槽函数 - 触发事件：获取输入文件路径的窗口对话
    void pushOutputFolderButton();  // 槽函数 - 触发事件：获取存储文件路径的窗口对话
    void pushCommitButton();        // 槽函数 - 触发事件：根据输入的文件开始执行算法
    void pushCancelButton();        // 槽函数 - 触发事件：清除当前窗口所有标签的内容

public:
    explicit HashWidget(QWidget* parent = nullptr); // 窗口构造函数

    ~HashWidget() override;                         // 窗口析构函数

};

#endif //DATASTRUCTURECLASSDESIGN_HASHWIDGET_H
```

Algorithm

```
#include <iostream>
#include <unordered_map>
#include <fstream>
#include <vector>
#include <algorithm>


class hashAlgorithm {
private:
    std::vector<std::string> keywords = {
            "auto", "break", "case", "char", "const",
            "continue", "default", "do", "double", "else",
            "enum", "extern", "float", "for", "goto",
            "if", "int", "long", "register", "return",
            "short", "signed", "sizeof", "static", "struct",
            "switch", "typedef", "union", "unsigned", "void",
            "volatile", "while"
    };

    std::string inputPath, outputPath;
    int hashSize;

    // std 检验
    void writeToFile(std::unordered_map<std::string, int>& keywordCount,
std::string path);

    // 自定义哈希表检验
    void writeToFile(std::vector<std::pair<std::string, int>>& keywordCount,
std::string path);

public:
    // 构造函数
    hashAlgorithm(std::string _inputPath, std::string _outputPath);

    void stdHash();                 // 标准哈希 - 用于测试检验
    int selfHash();                 // 自定义哈希
    int binaryHash();               // 二分哈希
};
```

2.3 实验数据与测试结果分析

代码文件从 github 上下载而来，其中 100 行与 500 行的代码文件为完整程序，2500 行代码文件为手动构造程序，三种文件数据规模的查找比较次数如下：

	self hash	binary hash
100 lines	120	775
500 lines	1546	10300
2500 lines	5561	39097

可以发现：使用线性探测法进行哈希的比较次数远小于二分搜索的比较次数

2.4 程序清单

- `/Src/Algorithm/hashAlgorithm.cpp`：内核哈希算法
- `/Src/Forms/HashWidget.ui`：ui 界面
- `/Src/Headers/HashWidget.h`：哈希组件逻辑头文件
- `/Src/Post/HashWidget.cpp`：哈希组件逻辑源文件
- `/Src/Test/TestHash.cpp`：程序测试文件

其中，除了 ui 界面使用 Qt Designer 设计师工具进行编写，其余文件均为与文件名同名的 C++ 类，便于维护与扩展

三、收获与体会

算法思维。四种排序算法总共也就 100 行，加上堆排序采用树形迭代，快排与归并排序采用递归，整体编码难度并不算高。即使加上了文件 I/O 操作也不过 150 行。而第二道题也是 hash 查找与 binary 查找加上文件 I/O 操作，整体算法难度也十分有限。但是对于我来说也加强了 hash 构建与查找的逻辑。

前后端开发。为了锻炼设计模式思维与软件工程思维，我加入了 Qt 集成的 GUI 界面，不仅体会到了算法对于软件的核心作用所在，也体会到了窗口程序的数据交互与数据接口的开发过程。比如 Qt 中信号与槽的概念，其实就是一种数据交互的 API 格式。同时在体验开发过程的同时，也感受到了用户体验的前端板块，通过使用 Qt Designer 的设计师开发界面，可以通过拖动组件的形式进行可视化设计界面，提高了开发效率，如果可以多人组队，也实现了前后端分离的同步开发模式。

异常处理。与传统 OJ 不同的是，一个软件还需要考虑更多的异常处理，没有标准的 std 数据，加上用户输入、操作的无穷性，注定了软件的异常处理不是一个简单的事，比如在解析用户输入的字符串以及解析用户按钮操作的过程中，需要添加很多的字符串处理结构以及事件触发处理结构，这对于我强化软件开发的健壮性很有帮助。

版本管理。通过 Git 版本管理与 Github 的云端同步功能，也更能体会到开发留痕与 bug 检测的优势。

当然美中不足的也有很多，比如单人全栈开发并不利于锻炼团队协作的能力，同时对于程序的架构设计也没有经过深度考虑，仅仅有界面，数据正常交互就戛然而止。希望在未来的算法与开发路上可以走的更远、更坚定。

仓库地址

github: https://github.com/Explorer-Dong/DataStructureClassDesign

gitee: https://gitee.com/explorer-dong/DataStructureClassDesign