****

实践教学

****

兰州理工大学

计算机与通信学院

2024 年秋季学期

算法与数据结构 课程设计

题　　目：任务书-02

专业班级：计算机科学与技术 1 班

姓　　名：

学　　号：

指导教师：

成　　绩：

目　录

摘　要

本项目开发了一个基于 GTK3 的图形化应用程序，展示算法与数据结构课程中的四个关键问题：集合运算、递归替换、哈夫曼编码系统和排序重构。应用程序采用模块化设计，主程序通过`src/main.c`文件实现，负责窗口的初始化、背景图片的动态更新和字体大小的调整。各功能模块分别在独立的源文件中实现，并通过 GTK 的 Notebook 组件集成展示。集合运算模块实现了集合的基本操作和幂集求解，递归替换模块支持 C/C++代码中的`#include`指令递归展开，哈夫曼编码系统通过输入文本构建最优前缀编码以实现文本压缩与解码，排序重构模块则通过特定差集数组推导可能的原始数列。项目深入分析了每个问题的逻辑和物理存储结构，设计并实现了高效算法，经过测试验证了其正确性与效率，充分展示了数据结构与算法在解决复杂问题中的重要应用价值。关键词： 集合运算；递归替换；哈夫曼编码；排序重构；算法复杂度;图形化；

一．集合运算问题

设计一个程序，实现两个集合的并集、交集、差集、显示输出等，要求结果集合中的元素不重复；实现一个集合的幂集的求解。（1）

1. 集合运算模块采用类语言定义相关的数据类型

1.1 类图表示

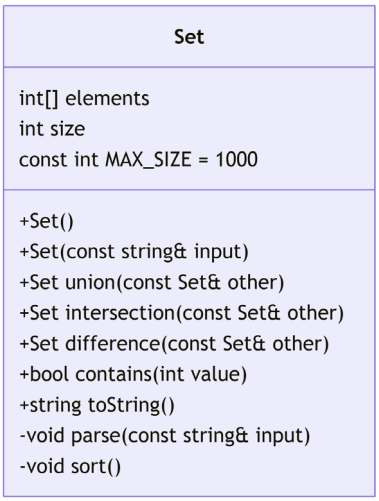

图 1-UML 集合运算类图

1.2 类图说明

- **Set 类**：这是集合运算模块的核心类，负责管理和操作集合数据。
  - **数据成员**：
    - elements：一个整数数组，用于存储集合中的元素。
    - size：当前集合中元素的个数。
    - MAX_SIZE：集合的最大容量，常量值为 1000。
  - **构造方法**：
    - Set()：默认构造函数，创建一个空集合。
    - Set(const string& input)：从字符串创建集合，字符串格式为以空格或逗号分隔的整数。

- **成员方法：**
  - union(const Set& other)：计算当前集合与另一个集合的并集，返回一个新的集合。
  - intersection(const Set& other)：计算当前集合与另一个集合的交集，返回一个新的集合。
  - difference(const Set& other)：计算当前集合与另一个集合的差集，返回一个新的集合。
  - contains(int value)：检查集合中是否包含指定的元素。
  - toString()：将集合转换为字符串表示，格式化输出集合中的元素。
- **私有方法：**
  - parse(const string& input)：解析输入字符串并填充集合。
  - sort()：对集合元素进行排序。

1.3 类语言定义说明

```
模块名：SetOperations（集合运算模块）

类名：Set（集合类）
----------------------------------------
数据成员：
    - elements: int[]        // 存储集合元素的数组
    - size: int              // 当前集合中的元素个数
    - MAX_SIZE: const int    // 集合的最大容量（1000）

构造方法：
    + Set()
        功能：创建一个空集合
        参数：无
        返回：Set 对象
```

```
15.
16.     + Set(const string& input)
17.         功能：从字符串创建集合
18.         参数：input - 以空格或逗号分隔的整数字符串
19.         返回：Set 对象
20.         异常：当输入格式错误时抛出异常
21.
22. 成员方法：
23.     1. 集合运算方法
24.     + Set union(const Set& other)
25.         功能：计算当前集合与另一个集合的并集
26.         参数：other - 另一个集合对象
27.         返回：包含并集结果的新集合
28.         时间复杂度：O(n log n)
29.
30.     + Set intersection(const Set& other)
31.         功能：计算当前集合与另一个集合的交集
32.         参数：other - 另一个集合对象
33.         返回：包含交集结果的新集合
34.         时间复杂度：O(n log n)
35.
36.     + Set difference(const Set& other)
37.         功能：计算当前集合与另一个集合的差集
38.         参数：other - 另一个集合对象
39.         返回：包含差集结果的新集合
40.         时间复杂度：O(n log n)
41.
42.     2. 工具方法
43.     + bool contains(int value)
```

```
        功能：检查集合中是否包含指定元素
        参数：value - 要查找的整数值
        返回：如果包含返回 true，否则返回 false
        时间复杂度：O(n)

    + string toString()
        功能：将集合转换为字符串表示
        参数：无
        返回：格式化的集合字符串，如 "1, 2, 3"
        时间复杂度：O(n)

私有方法：
    - void parse(const string& input)
        功能：解析输入字符串并填充集合
        参数：input - 输入字符串
        返回：无
        异常：当输入格式无效时抛出异常

    - void sort()
        功能：对集合元素进行排序
        参数：无
        返回：无
        时间复杂度：O(n log n)
```

这个类语言定义和类图展示了集合运算模块的结构和功能，使得面向过程的 C 通过上述结构模拟了面向对象的设计。

2. 集合运算算法设计

代码实现了集合的并集、交集和差集运算，并与 GTK 图形用户界面（GUI）集成，以便用户输入集合并查看运算结果。代码功能完整，结构合理，具有以下特点：

2.1 模块化设计：

- union.h 声明了必要的全局变量和函数接口。
- union.c 包含具体实现，并与 GUI 组件交互。

2.2 错误处理：

- 使用 ErrorContext 进行异常捕获和处理，确保在运行过程中即使发生错误也不会导致程序崩溃。

2.3 集合操作算法：

- 包括解析输入集合的算法、检测数组中是否包含某值的函数、将数组转化为字符串的函数，以及集合并集、交集和差集运算逻辑。

2.4 用户友好的 GUI：

- 提供输入区域、运算按钮和输出区域，方便用户操作。
- 使用 GtkTextView 获取输入数据，并将结果显示在输出框中。

2.5 优化与建议

1. **代码复用：**
   - 解析输入、错误处理等部分代码存在重复，可进一步封装成独立函数以提高复用性。
2. **性能改进：**
   - 当前使用线性搜索检查数组中是否包含某值，性能为 (O(n))。如果集合较大，可考虑使用哈希表或二分搜索树以提高性能。
3. **GUI 美化：**

- 增加一些提示文本，如 “请用逗号或空格分隔输入”。
- 在结果框中加入颜色区分或滚动条以提升用户体验。

4. **支持动态输入：**
   - 当前输入集合的大小由 MAX_SET_SIZE 限制。如果需要更灵活的操作，可动态分配内存，并处理用户可能输入的更大集合。

3. 系统结构操作流程图和函数的调用关系图

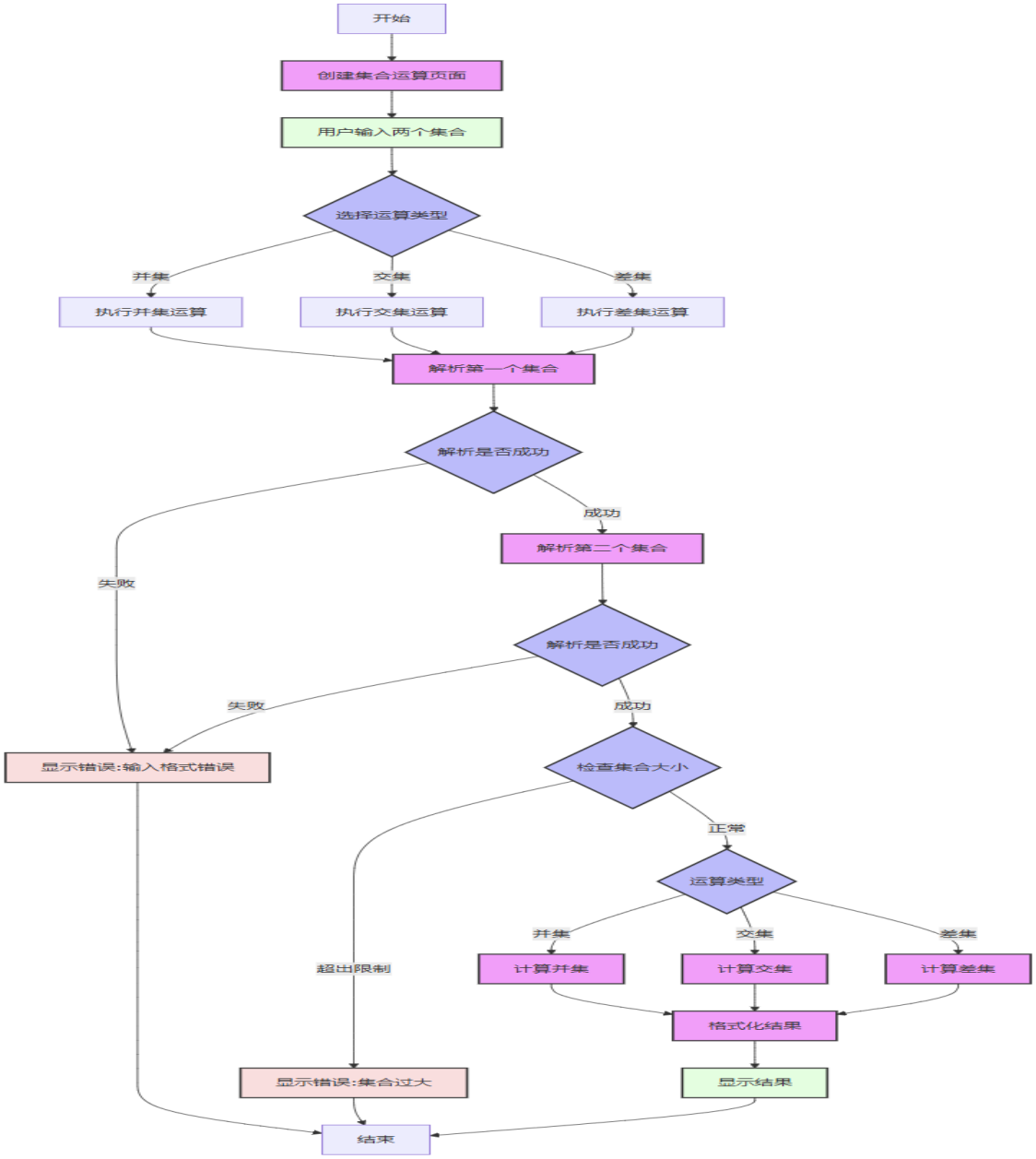

图 2-集合运算模块系统结构操作流程图

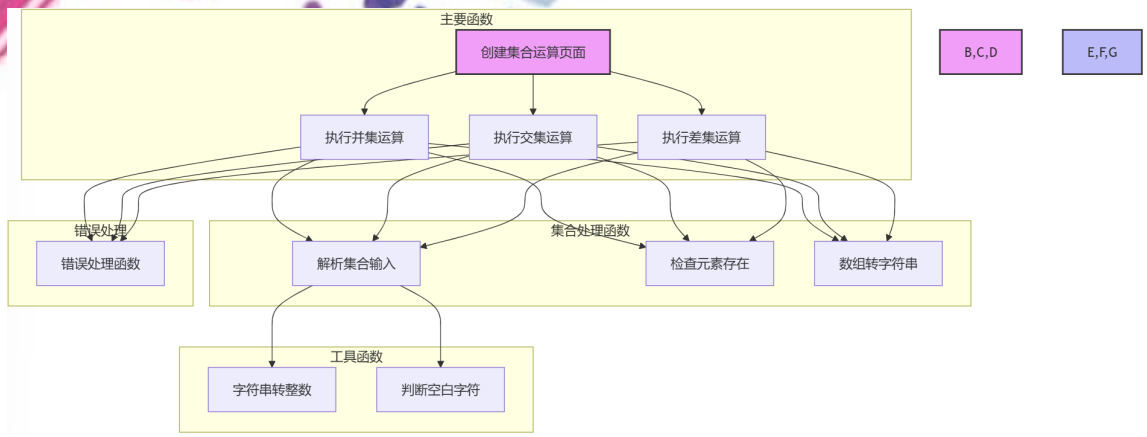

图 3-集合运算模块函数调用关系图

4. 调试分析

a、调试中遇到的问题及对问题的解决方法

I 内存泄漏问题

```
1.  FINALLY({
2.  g_free(set1_text);
3.  g_free(set2_text);
4.  free(set1);
5.  free(set2);
6.  free(result_str);
7.  g_free(display);
8.  });
```

- 问题：在错误处理时可能发生内存泄漏
- 解决：使用 TRY-CATCH-FINALLY 机制，在 FINALLY 块中统一释放所有分配的内存

II 输入验证问题

```
1. int size1 = parse_union_set(set1_text, set1);
2. int size2 = parse_union_set(set2_text, set2);
3.     if (size1 < 0 || size2 < 0) {
4.         THROW(&error_ctx, ERROR_INVALID_INPUT, "无效输入 - 请输入有效的集合");
```

```
5.     }
6.
```

- 问题：用户输入格式不规范导致程序崩溃
- 解决：添加输入验证，对无效输入进行错误提示

III 集合大小限制

```
define MAX_SET_SIZE 1000
```

- 问题：需要限制集合大小防止内存溢
- 解决：定义 MAX_SET_SIZE 常量限制集合大小

b、算法的时间复杂度和空间复杂度

I 算法的时间复杂度和空间复杂度

i 并集运算

```
1. int result_size = 0;
2. for (int i = 0; i < size1; i++) {
3. result[result_size++] = set1[i];
4. }
5. for (int i = 0; i < size2; i++) {
6. if (!contains(result, result_size, set2[i])) {
7. result[result_size++] = set2[i];
8. }
9. }
10.
```

- 时间复杂度：$O(n^2)$，因为需要检查重复元素
- 空间复杂度：O(n)，需要存储结果集合

ii 交集运算

```
1. int result_size = 0;
2. for (int i = 0; i < size1; i++) {
3. if (contains(set2, size2, set1[i])) {
4. result[result_size++] = set1[i];
```

```
5. }
6. }
```

- 时间复杂度：$O(n^2)$，需要在第二个集合中查找元素
- 空间复杂度：O(n)，存储结果集合

iii 差集运算

```
1. // 计算 A-B
2. int result_size_ab = 0;
3. for (int i = 0; i < size1; i++) {
4. if (!contains(set2, size2, set1[i])) {
5. result_ab[result_size_ab++] = set1[i];
6. }
7. }
8.
```

```
1.     // 计算 B-A
2.     int result_size_ba = 0;
3.     for (int i = 0; i < size2; i++) {
4.         if (!contains(set1, size1, set2[i])) {
5.             result_ba[result_size_ba++] = set2[i];
6.         }
7.     }
8.
```

- 时间复杂度：$O(n^2)$，需要计算两个方向的差集
- 空间复杂度：O(n)，需要存储两个差集结果

II 优化建议：

可以使用哈希表将查找操作优化到 O(1)，但会增加空间复杂度

可以先对输入集合排序，然后使用双指针法优化交集和差集运算

考虑使用位图（bitmap）存储集合，可以减少内存使用并提高运算速度。

5. 测试结果

输入示例：

```
集合 A：1 2 3

集合 B：2 3 4
```

预计输出示例：

```
并集：{1,2,3,4}

交集：{2,3}

差集：A-B:{1}B-A :{4}
```

图 4-集合运算模块并集运算结果

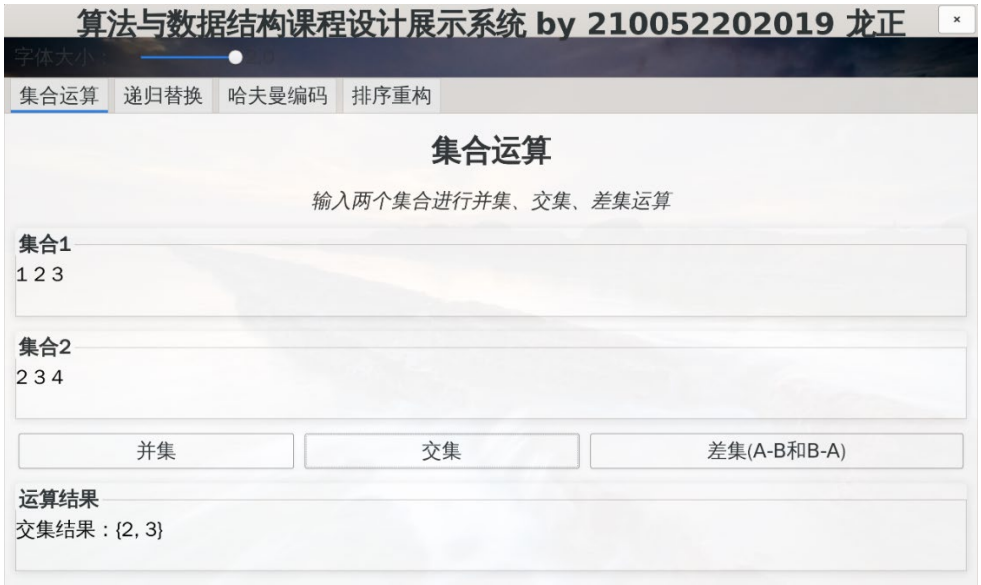

图 5-集合运算模块交集运算

图 6-集合运算差集运算

6. 源程序（带注释）

```
1. /**
2.  * 集合元素查找函数
3.  * 功能：在给定数组中查找指定值
4.  * 参数：arr-目标数组，size-数组大小，value-要查找的值
5.  * 返回：布尔值，表示是否找到
6.  */
7. static bool contains(const int arr[], int size, int value) {
8.     // 参数检查：如果数组为空或大小为 0，返回 false
9.     if (!arr || size <= 0) {
10.         return false;
11.     }
12.
13.     // 遍历数组的每个元素
14.     for (int i = 0; i < size; i++) {
```

```
        // 比较当前元素与目标值
        if (arr[i] == value) {
            // 找到匹配元素，返回 true
            return true;
        }
    }
    // 遍历结束未找到匹配元素，返回 false
    return false;
}

/**
 * 集合并集运算函数
 * 功能：计算两个集合的并集，结果保存在全局 result 数组中
 * 参数：widget-GTK 部件，data-用户数据（未使用）
 */
void perform_set_union(GtkWidget *widget, gpointer data) {
    // 声明错误处理上下文
    ErrorContext error_ctx;
    // 初始化错误处理上下文
    init_error_context(&error_ctx);

    // 声明所需的临时变量
    char *set1_text = NULL;    // 第一个集合的输入文本
    char *set2_text = NULL;    // 第二个集合的输入文本
    int *set1 = NULL;          // 第一个集合的数组
    int *set2 = NULL;          // 第二个集合的数组
    char *result_str = NULL;   // 结果字符串

    // 使用 TRY-CATCH 进行错误处理
```

```
    TRY(&error_ctx) {
        // 获取第一个输入框的文本缓冲区
        GtkTextBuffer *buffer1 =
gtk_text_view_get_buffer(GTK_TEXT_VIEW(text_view_input1));
        // 获取第二个输入框的文本缓冲区
        GtkTextBuffer *buffer2 =
gtk_text_view_get_buffer(GTK_TEXT_VIEW(text_view_input2));

        // 获取第一个文本缓冲区的起始和结束位置
        GtkTextIter start1, end1;
        gtk_text_buffer_get_bounds(buffer1, &start1, &end1);
        // 获取第二个文本缓冲区的起始和结束位置
        GtkTextIter start2, end2;
        gtk_text_buffer_get_bounds(buffer2, &start2, &end2);

        // 获取输入文本内容
        set1_text = gtk_text_buffer_get_text(buffer1, &start1, &end1, FALSE);
        set2_text = gtk_text_buffer_get_text(buffer2, &start2, &end2, FALSE);

        // 检查输入是否为空
        if (!set1_text || !set2_text) {
            THROW(&error_ctx, ERROR_INVALID_INPUT, "输入不能为空");
        }

        // 为集合分配内存
        set1 = malloc(MAX_SET_SIZE * sizeof(int));
        set2 = malloc(MAX_SET_SIZE * sizeof(int));

        // 检查内存分配是否成功
```

```
71.         if (!set1 || !set2) {
72.             THROW(&error_ctx, ERROR_MEMORY_ALLOCATION, "内存分配失败");
73.         }
74.
75.         // 解析输入文本为整数数组
76.         int size1 = parse_union_set(set1_text, set1);
77.         int size2 = parse_union_set(set2_text, set2);
78.
79.         // 检查解析结果
80.         if (size1 < 0 || size2 < 0) {
81.             THROW(&error_ctx, ERROR_INVALID_INPUT, "无效的输入格式");
82.         }
83.
84.         // 计算并集
85.         int result_size = 0;
86.         // 首先复制第一个集合的所有元素
87.         for (int i = 0; i < size1; i++) {
88.             result[result_size++] = set1[i];
89.         }
90.
91.         // 添加第二个集合中不重复的元素
92.         for (int i = 0; i < size2; i++) {
93.             // 检查元素是否已存在于结果集中
94.             if (!contains(result, result_size, set2[i])) {
95.                 // 不存在则添加到结果集
96.                 result[result_size++] = set2[i];
97.             }
98.         }
99.
```

```
        // ... 后续处理代码 ...
    }
    CATCH(&error_ctx) {
        // 错误处理：显示错误对话框
        handle_error(gtk_widget_get_toplevel(widget),
                     error_ctx.code,
                     error_ctx.message);
    }
    FINALLY({
        // 清理资源：释放所有分配的内存
        g_free(set1_text);
        g_free(set2_text);
        free(set1);
        free(set2);
        free(result_str);
    });
}

```

二．递归替换问题

编写程序，扩展 C/C++源文件中的#include 指令（以递归的方式）。请以文件名的内容替换形如下面的代码行。

#include “filename”

1. 递归替换模块采用类语言定义相关的数据类型

1.1 类图展示

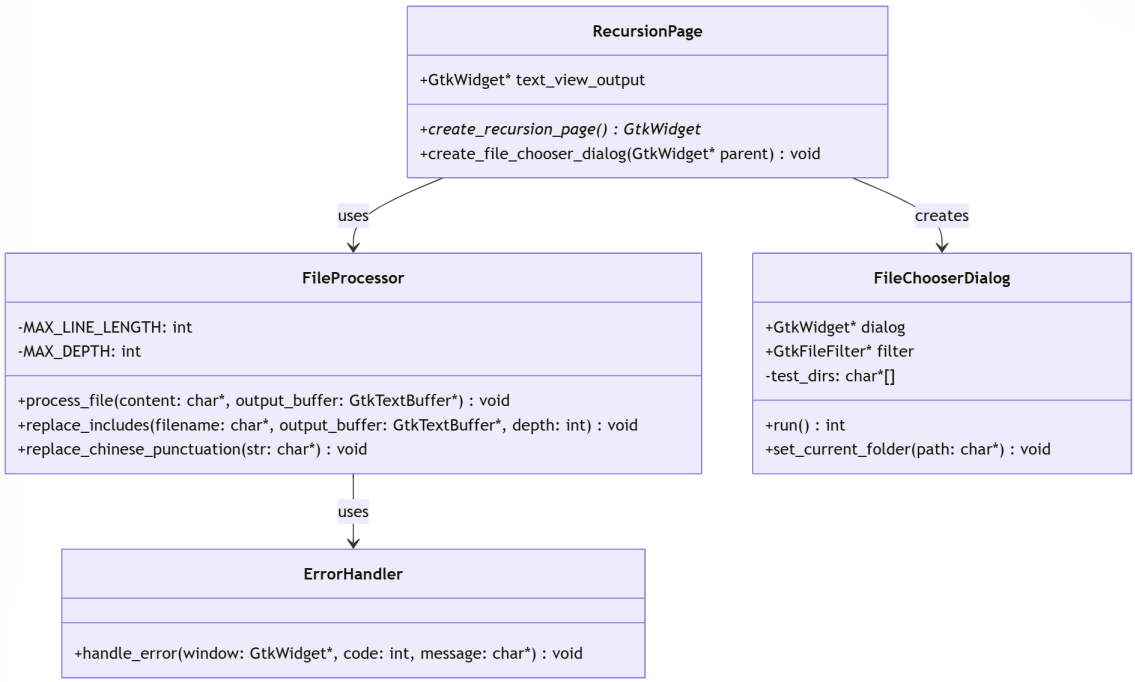

图 7-UML 递归展开#include 类图

类的主要职责：

- RecursionPage：
- 负责创建和管理 GUI 界面
- 处理用户交互
- FileProcessor：
- 处理文件内容
- 执行递归替换操作
- 处理中文标点符号转换
- FileChooserDialog：
- 提供文件选择对话框
- 管理文件过滤器
- 处理目录导航
- ErrorHandler：
- 处理错误情况
- 显示错误消息

这些类之间的关系主要是组合和依赖关系，共同实现了递归替换功能。

1.2 要采用的数据类型

✧ 基本数据类型：

char：用于存储文件内容和字符处理

int：用于记录递归深度和缓冲区大小

size_t：用于数组索引和长度计算

常量定义：

```
#define MAX_LINE_LENGTH 2048

#define MAX_DEPTH 10
```

定义了最大行长度和递归深度限制。

✧ 字符串缓冲区：

```
char line[1024];
```

用于存储文件的每一行内容。

✧ 文件路径数组：

```
1. const char search_paths[] = {

2. “.”,

3.  “./resources/test_files”,

4.  “../resources/test_files”,

5.  “../../resources/test_files”,

6.  “../../../resources/test_files”,

7.  NULL

8. };

9.
```

用于存储可能的文件搜索路径。

✧ *GTK 界面组件：*

```
static GtkWidget text_view_output;
```

用于显示处理结果的文本视图组件。

这些数据类型共同构成了递归替换问题的数据结构体系，主要用于：

- 文件内容的读取和存储
- 路径管理和文件查找
- 递归深度控制
- 用户界面交互
- 结果输出显示

通过这些数据类型的组合使用，实现了对源文件中 include 指令的递归展开处

理。

2. 递归替换模块算法设计

为递归展开模块设计算法，主要包含以下几个关键算法：

2.1 递归展开算法（主算法）

- 算法：RecursiveExpand(filename, depth)
- 输入：文件名 filename，当前递归深度 depth
- 输出：展开后的文件内容

IF depth > MAX_DEPTH THEN

返回错误：”递归深度超限”

以下是算法伪代码：

```
1. 打开文件 filename
2. IF 打开失败 THEN
3. 返回错误：”文件无法打开”
4. 初始化输出缓冲区 output
5. WHILE 未到文件末尾 DO
6. 5.1 读取一行内容 line
7. 5.2 IF line 以”#include”开头 THEN
8. a) 提取引号中的文件名 included_file
9. b) 构造完整路径 full_path
10. c) 输出包含文件开始标记
11. d) 递归调用 RecursiveExpand(full_path, depth + 1)
12. e) 输出包含文件结束标记
13. 5.3 ELSE
14. 处理普通行（调用 ChinesePunctuationReplace）
15. 将处理后的行添加到输出缓冲区
```

```
16. 关闭文件
17. 返回输出缓冲区内容
18.
```

2.2 中文标点替换算法

- 算法：ChinesePunctuationReplace(text)
- 输入：包含中文标点的文本 text
- 输出：替换后的文本

```
1. 初始化输出缓冲区 output
2. FOR i = 0 TO text.length - 1 DO
3. 2.1 IF text[i]是 UTF-8 编码的中文标点 THEN
4. CASE text[i] OF
5.   "：" : output += ":"
6.   "（" : output += "("
7.   "）" : output += ")"
8.   其他 : output += text[i]
9. 2.2 ELSE
10. output += text[i]
11. 返回 output
12.
```

2.3 文件路径处理算法

- 算法：ResolvePath(base_path, include_path)
- 输入：基础路径 base_path，包含文件路径 include_path
- 输出：完整的文件路径

```
1. IF include_path 是绝对路径 THEN
```

```
2. 返回 include_path
3. 获取 base_path 的目录部分 dir
4. 拼接路径：dir + "/" + include_path
5. 规范化路径（处理"./"和"../"）
6. 返回完整路径
7.
```

2.4 算法复杂度分析

➢ 时间复杂度：

- 主算法：O(n × d)，其中 n 是文件总行数，d 是递归深度
- 标点替换：O(m)，m 是文本长度
- 路径处理：O(p)，p 是路径长度

➢ 空间复杂度：

O(n × d)，主要来自递归调用栈和输出缓冲区

➢ 关键数据结构：

- 字符串缓冲区：用于存储处理后的文本
- 递归调用栈：用于处理嵌套包含
- 文件路径缓冲区：用于路径处理

3. 系统结构操作流程图和函数的调用关系图

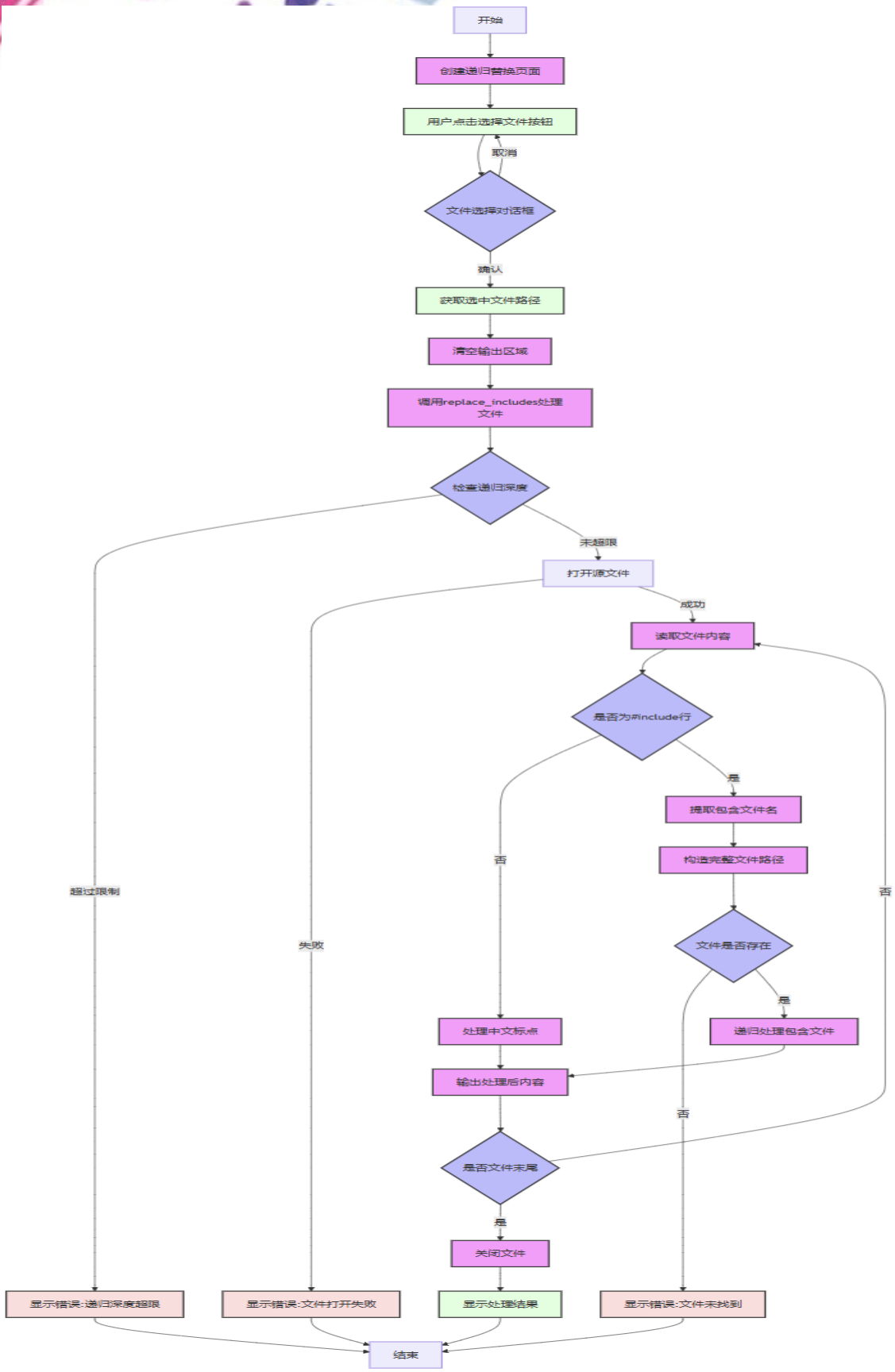

图 8-递归展开模块系统结构操作流程图

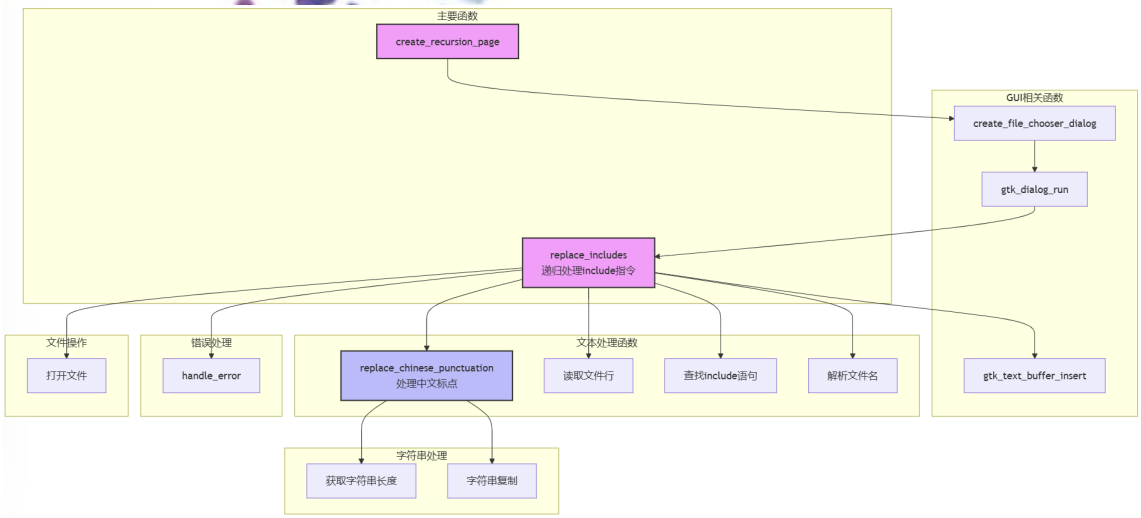

图 9-递归展开函数调用关系

4. 调试分析

a、调试中遇到的问题及对问题的解决方法

I 问题：文件路径错误

- **现象**：在选择文件时，可能会出现无法找到文件或路径错误的情况。
- **解决方法**：确保文件选择对话框正确配置，并检查文件路径是否正确。可以通过在调试时打印出选择的文件路径来验证。

II 问题：内存泄漏

- **现象**：在处理文件内容时，可能会出现内存泄漏，导致程序占用内存不断增加。
- **解决方法**：使用工具(如 Valgrind)检测内存泄漏，确保在使用 g_free 和 free 时正确释放所有动态分配的内存。

III 问题：递归深度过大

- **现象**：在递归展开#include 指令时，可能会因为递归深度过大导致栈溢出。
- **解决方法**：增加递归深度限制，或者使用迭代方法替代递归。

IV 问题：文件读取错误

- **现象**：在读取文件内容时，可能会出现读取不完整或读取失败的情况。

- **解决方法**：检查文件是否存在并具有读取权限，确保文件指针正确使用。

b、算法的时间复杂度和空间复杂度

- **时间复杂度：**
  - 递归展开#include 指令的时间复杂度主要取决于文件的层次结构和每个文件中包含的#include 指令数量。假设每个文件平均包含 k 个#include 指令，递归深度为 d，则时间复杂度为 O(k^d)。
- **空间复杂度：**
  - 空间复杂度主要由递归调用栈和存储文件内容的内存决定。假设每个文件的平均大小为 k 字节，递归深度为 d，则空间复杂度为 O(n * d)。

5. 测试结果

5.1 对.cpp 文件进行递归展开

- test.cpp 文件内容：

```cpp
#include <iostream>
#include "header1.h"
#include "header2.h"

int main() {
    std::cout << "This is main file" << std::endl;
    test_func1();
    test_func2();
    return 0;
}
```

```
11.
```

- header1.h 内容如下：

```
1. #ifndef HEADER1_H
2. #define HEADER1_H
3.
4. #include "header3.h"
5.
6. void test_func1() {
7.     std::cout << "This is function 1" << std::endl;
8.     test_func3();
9. }
10.
11. #endif
12.
```

- header2.h 内容如下：

```
1. #ifndef HEADER2_H
2. #define HEADER2_H
3.
4. void test_func2() {
5.     std::cout << "This is function 2" << std::endl;
6. }
7.
8. #endif
9.
```

- header3.h 内容如下：

```
1. #ifndef HEADER3_H
```

```
2. #define HEADER3_H

3.

4. void test_func3() {

5.     std::cout << "This is function 3" << std::endl;

6. }

7.

8. #endif

9.
```

5.2cpp 文件展开运行效果：

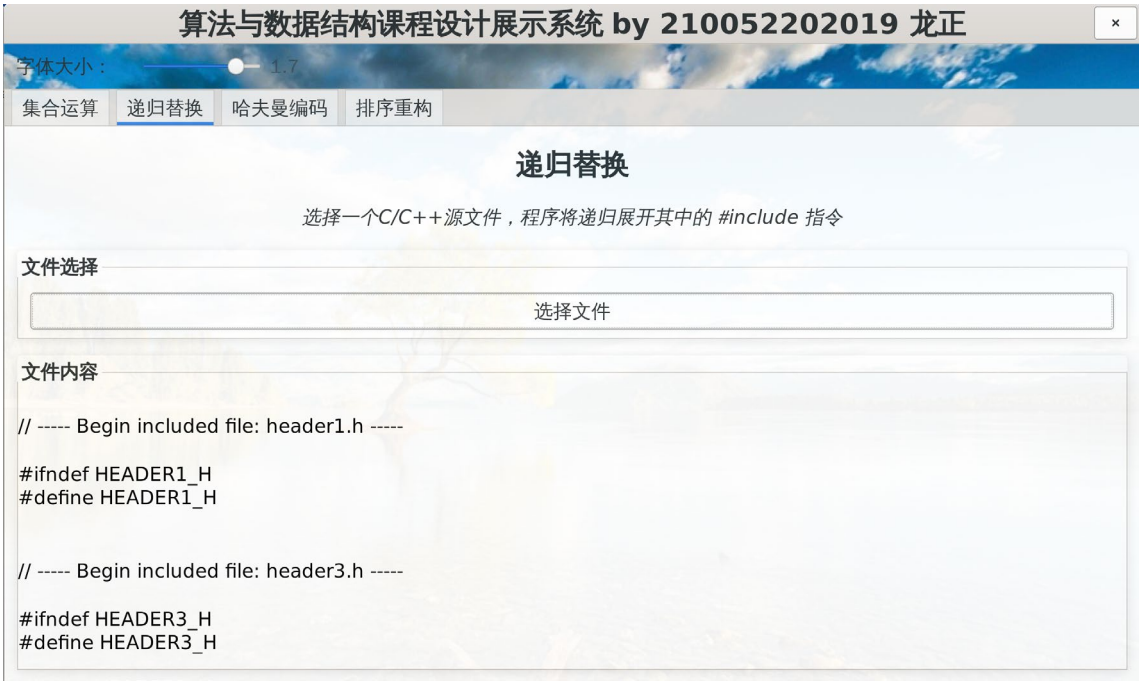

图 10-递归展开 CPP 文件展开效果演示

➢ 展开文件内容（详情）：

```
1. // ----- Begin included file: header1.h -----

2.

3. #ifndef HEADER1_H

4. #define HEADER1_H

5.
```

```
6.
7. // ----- Begin included file: header3.h -----
8.
9. #ifndef HEADER3_H
10. #define HEADER3_H
11.
12. void test_func3() {
13.     std::cout << "This is function 3" << std::endl;
14. }
15.
16. #endif
17. // ----- End included file: header3.h -----
18.
19.
20. void test_func1() {
21.     std::cout << "This is function 1" << std::endl;
22.     test_func3();
23. }
24.
25. #endif
26. // ----- End included file: header1.h -----
27.
28.
29. // ----- Begin included file: header2.h -----
30.
31. #ifndef HEADER2_H
32. #define HEADER2_H
33.
34. void test_func2() {
```

```
35.     std::cout << "This is function 2" << std::endl;
36. }
37.
38. #endif
39. // ----- End included file: header2.h -----
40.
41.
42. int main() {
43.     std::cout << "This is main file" << std::endl;
44.     test_func1();
45.     test_func2();
46.     return 0;
47. }
48.
```

- 对.c 文件展开演示效果：

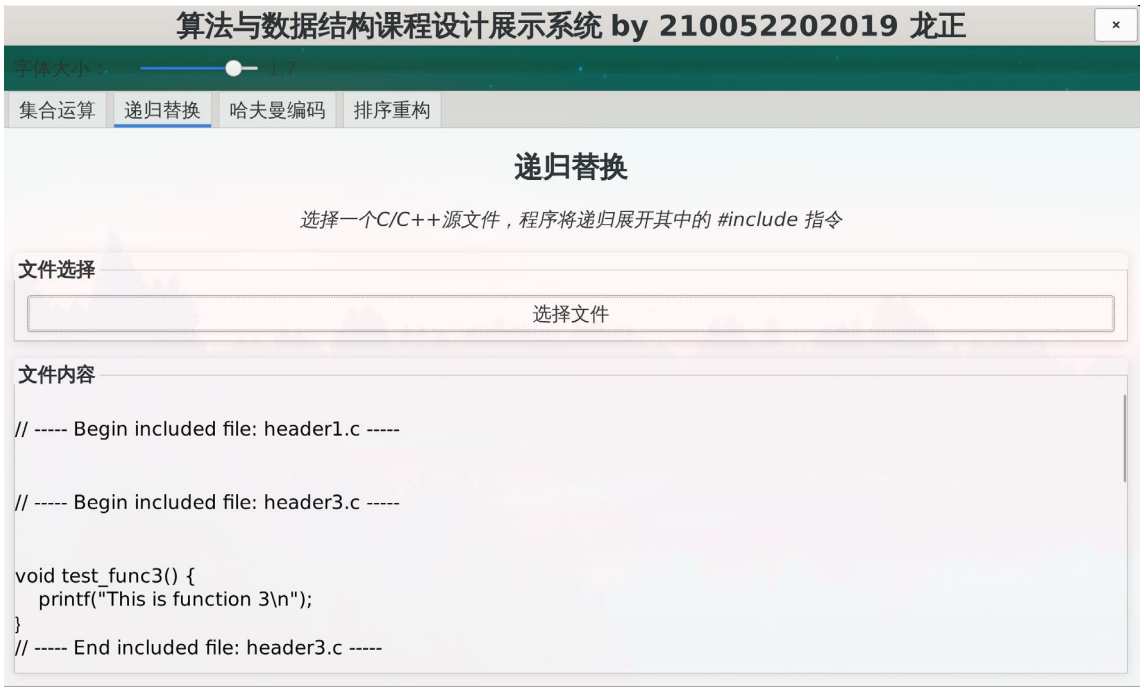

图 11-递归展开 C 文件演示

由于同理，所以此处省略输入具体文件和具体输出文件，只展示运行效果图。

5. 源程序（带注释）

- recursion.h 代码：

```
#ifndef RECURSION_H

#define RECURSION_H

#include <gtk/gtk.h>

// 函数声明
GtkWidget* create_recursion_page(void); // 创建递归页面的函数
void replace_includes(const char* filename, GtkTextBuffer *output_buffer, int depth); // 递归替换 #include 的函数

#endif // 结束头文件保护
```

- recursion.c 代码：

```
#define MAX_LINE_LENGTH 2048 // 定义每行的最大长度
#define MAX_DEPTH 10 // 定义递归的最大深度

// 递归处理 include 指令
void replace_includes(const char* filename, GtkTextBuffer *output_buffer, int depth) {
    GtkTextIter end; // 定义文本迭代器，用于在文本缓冲区中插入文本
    GtkWidget *window = gtk_widget_get_toplevel(gtk_widget_get_parent(gtk_widget_get_parent(GTK_WIDGET(text_view_output))));

```

```
    // 检查递归深度，防止无限递归导致栈溢出
    if (depth > MAX_DEPTH) {
        handle_error(window, ERROR_INVALID_OPERATION, "包含文件嵌套层数过多");
        return;
    }

    // 打开文件以读取内容
    FILE *file = fopen(filename, "r");
    if (!file) {
        // 如果文件打开失败，尝试在其他目录中查找文件
        const char* search_paths[] = {
            ".",
            // 其他可能的搜索路径可以在这里添加
        };

        for (size_t i = 0; i < sizeof(search_paths)/sizeof(search_paths[0]); i++) {
            char full_path[PATH_MAX];
            snprintf(full_path, sizeof(full_path), "%s/%s", search_paths[i],
filename);
            file = fopen(full_path, "r");
            if (file) {
                break; // 如果成功打开文件，退出循环
            }
        }

        if (!file) {
            // 如果仍然无法打开文件，报告错误
            char error_msg[256];
            snprintf(error_msg, sizeof(error_msg), "无法打开文件: %s", filename);
```

```
37.             handle_error(window, ERROR_FILE_NOT_FOUND, error_msg);
38.             return;
39.         }
40.     }
41.
42.     char line[1024]; // 用于存储从文件中读取的每一行
43.     while (fgets(line, sizeof(line), file)) {
44.         // 检查缓冲区溢出
45.         if (strlen(line) >= sizeof(line) - 1) {
46.             handle_error(window, ERROR_BUFFER_OVERFLOW, "行内容过长");
47.             fclose(file);
48.             return;
49.         }
50.
51.         // 检查是否为#include 指令
52.         if (strstr(line, "#include") == line) {
53.             char included_file[MAX_LINE_LENGTH];
54.             char *start = strchr(line, '"') + 1; // 找到第一个引号后的字符
55.             char *end_quote = strrchr(line, '"'); // 找到最后一个引号
56.             if (start && end_quote && end_quote > start) {
57.                 size_t len = end_quote - start;
58.                 if (len >= MAX_LINE_LENGTH - 1) {
59.                     handle_error(window, ERROR_BUFFER_OVERFLOW, "包含的文件路径过长
");
60.                     return;
61.                 }
62.                 strncpy(included_file, start, len); // 提取包含文件的名称
63.                 included_file[len] = '\0';
64.
```

```
65.                 // 构造包含文件的完整路径
66.                 char *dirname = g_path_get_dirname(filename);
67.                 char full_path[PATH_MAX];
68.                 size_t path_len = strlen(dirname) + strlen(included_file) + 2;  //
+2 for '/' and '\0'
69.                 if (path_len >= PATH_MAX) {
70.                     handle_error(window, ERROR_BUFFER_OVERFLOW, "路径太长");
71.                     return;
72.                 }
73.                 snprintf(full_path, PATH_MAX, "%s/%s", dirname, included_file);
74.                 g_free(dirname);
75.                 // 输出包含文件的开始标记
76.                 gtk_text_buffer_get_end_iter(output_buffer, &end);
77.                 gtk_text_buffer_insert(output_buffer, &end,
78.                     "\n// ----- Begin included file: ", -1);
79.                 gtk_text_buffer_insert(output_buffer, &end, included_file, -1);
80.                 gtk_text_buffer_insert(output_buffer, &end, " -----\n\n", -1);
81.                 // 递归处理包含文件
82.                 replace_includes(full_path, output_buffer, depth + 1);
83.                 // 输出包含文件的结束标记
84.                 gtk_text_buffer_get_end_iter(output_buffer, &end);
85.                 gtk_text_buffer_insert(output_buffer, &end,
86.                     "\n// ----- End included file: ", -1);
87.                 gtk_text_buffer_insert(output_buffer, &end, included_file, -1);
88.                 gtk_text_buffer_insert(output_buffer, &end, " -----\n\n", -1);
89.             }
90.         } else {
91.             // 处理普通行
92.             char formatted_line[MAX_LINE_LENGTH * 2];  // 双倍缓冲区以防万一
```

```
            const char* src = line;
            char* dst = formatted_line;

            // 格式化处理
            while (*src) {
                if (*src == '：') *dst++ = ':'; // 替换中文冒号
                else if (*src == '（') *dst++ = '('; // 替换中文左括号
                else if (*src == '）') *dst++ = ')'; // 替换中文右括号
                else *dst++ = *src; // 复制其他字符
                src++;
            }
            *dst = '\0'; // 终止字符串

            gtk_text_buffer_get_end_iter(output_buffer, &end);
            gtk_text_buffer_insert(output_buffer, &end, formatted_line, -1); // 插入格式化后的行
        }
    }

    fclose(file); // 关闭文件
}

```

三．哈夫曼码的编/译码系统

方程 $A^5+B^5+C^5+D^5+E^5=F^5$ 刚好有一个满足 $0 \leqslant A \leqslant B \leqslant C \leqslant D \leqslant E \leqslant F \leqslant 75$ 的整数解。请编写一个求出该解的程序。

1. 哈夫曼码编/译码模块采用类语言定义相关的数据类型

下面是哈夫曼编码/解码系统的类语言定义说明和类图。

1.1 类图展示

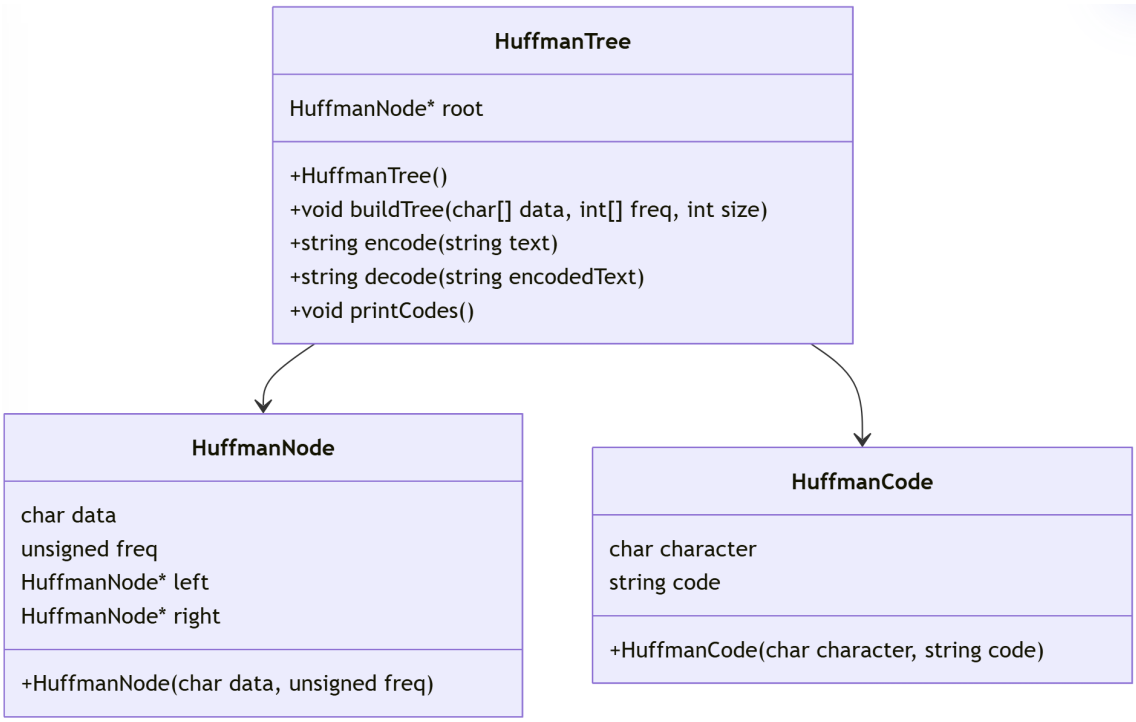

图 12-UML 哈夫曼编码模块类图

1.2 类图说明

- **HuffmanNode 类**：表示哈夫曼树的节点。
  - **数据成员**：
    - data：节点存储的字符。
    - freq：字符出现的频率。
    - left：指向左子节点的指针。

        - right：指向右子节点的指针。
    - **构造方法：**
        - HuffmanNode(char data, unsigned freq)：初始化节点的字符和频率。
- **HuffmanTree 类：**管理哈夫曼树的构建和编码/解码操作。
    - **数据成员：**
        - root：指向哈夫曼树根节点的指针。
    - **构造方法：**
        - HuffmanTree()：初始化一个空的哈夫曼树。
    - **成员方法：**
        - buildTree(char[] data, int[] freq, int size)：根据字符和频率数组构建哈夫曼树。
        - encode(string text)：将输入文本编码为哈夫曼码。
        - decode(string encodedText)：将哈夫曼码解码为原始文本。
        - printCodes()：打印所有字符的哈夫曼编码。
- **HuffmanCode 类：**存储字符及其对应的哈夫曼编码。
    - **数据成员：**
        - character：字符。
        - code：字符对应的哈夫曼编码。
    - **构造方法：**
        - HuffmanCode(char character, string code)：初始化字符和编码。

1.3 类语言定义说明

```
1. 模块名：HuffmanCodingSystem（哈夫曼编码系统）
2.
3. 类名：HuffmanNode（哈夫曼节点类）
4. ----------------------------------------
```

```
5. 数据成员：
6.     - data: char          // 节点存储的字符
7.     - freq: unsigned      // 字符出现的频率
8.     - left: HuffmanNode*  // 左子节点指针
9.     - right: HuffmanNode* // 右子节点指针
10.
11. 构造方法：
12.     + HuffmanNode(char data, unsigned freq)
13.         功能：初始化节点的字符和频率
14.         参数：data - 字符，freq - 频率
15.         返回：HuffmanNode 对象
16.
17. 类名：HuffmanTree（哈夫曼树类）
18. ----------------------------------------
19. 数据成员：
20.     - root: HuffmanNode*  // 哈夫曼树根节点指针
21.
22. 构造方法：
23.     + HuffmanTree()
24.         功能：初始化一个空的哈夫曼树
25.         参数：无
26.         返回：HuffmanTree 对象
27.
28. 成员方法：
29.     + void buildTree(char[] data, int[] freq, int size)
30.         功能：根据字符和频率数组构建哈夫曼树
31.         参数：data - 字符数组，freq - 频率数组，size - 数组大小
32.         返回：无
33.
```

```
    + string encode(string text)
        功能：将输入文本编码为哈夫曼码
        参数：text - 输入文本
        返回：编码后的字符串

    + string decode(string encodedText)
        功能：将哈夫曼码解码为原始文本
        参数：encodedText - 编码字符串
        返回：解码后的文本

    + void printCodes()
        功能：打印所有字符的哈夫曼编码
        参数：无
        返回：无

类名：HuffmanCode（哈夫曼编码类）
----------------------------------------
数据成员：
    - character: char       // 字符
    - code: string          // 字符对应的哈夫曼编码

构造方法：
    + HuffmanCode(char character, string code)
        功能：初始化字符和编码
        参数：character - 字符，code - 编码
        返回：HuffmanCode 对象
```

这个类语言定义和类图展示了哈夫曼编码/解码系统的结构和功能，模拟了面向对象的设计

2. 哈夫曼编译码模块算法设计

2.1 哈夫曼树构建算法

2. 1 算法设计

I 统计字符频率：

- 输入：文本字符串
- 输出：字符数组和对应的频率数组

伪代码：

```
1. function count_frequency(text):
2.     initialize count array of size MAX_CHAR to 0
3.     for each character c in text:
4.         increment count[c]
5.     initialize data and freq arrays
6.     for each character i in count:
7.         if count[i] > 0:
8.             add i to data array
9.             add count[i] to freq array
10.    return data, freq
11.
```

II 构建哈夫曼树：

- 输入：字符数组和频率数组
- 输出：哈夫曼树的根节点

伪代码：

```
1. function build_huffman_tree(data, freq, size):
2.     create a min heap of capacity size
3.     for each character in data:
4.         create a new node with character and frequency
5.         insert node into min heap
6.     while min heap size is not 1:
7.         extract two nodes with minimum frequency
8.         create a new node with frequency equal to the sum of the two nodes
9.         set extracted nodes as left and right children of the new node
10.        insert the new node into min heap
11.    return the remaining node in min heap
12.
```

III 生成哈夫曼编码：

- 输入：哈夫曼树的根节点
- 输出：字符的哈夫曼编码

伪代码：

```
1. function print_codes(root, code_str, top):
2.     if root is a leaf node:
3.         store code_str as the code for root.data
4.         print root.data and code_str
5.     else:
6.         if root.left is not null:
7.             code_str[top] = '0'
8.             print_codes(root.left, code_str, top + 1)
9.         if root.right is not null:
10.            code_str[top] = '1'
11.            print_codes(root.right, code_str, top + 1)
```

```
12.
```

IV 编码文本：

- 输入：文本字符串
- 输出：哈夫曼编码字符串

伪代码：

```
1. function encode_text(text):
2.     initialize encoded string
3.     for each character c in text:
4.         find code for c
5.         append code to encoded string
6.     return encoded string
7.
```

V 解码文本：

- 输入：哈夫曼编码字符串
- 输出：解码后的文本字符串

伪代码：

```
1. function decode_text(encoded_text, root):
2.     initialize decoded string
3.     set current node to root
4.     for each bit in encoded_text:
5.         if bit is '0':
6.             move to left child
7.         else:
8.             move to right child
9.         if current node is a leaf:
```

```
            append current node's data to decoded string
            reset current node to root
    return decoded string
```

2.2 算法复杂度分析

I 时间复杂度：

- 统计字符频率：O(n)，其中 n 是输入文本的长度。
- 构建哈夫曼树：O(d log d)，其中 d 是不同字符的数量（最多为 MAX_CHAR）。
- 生成哈夫曼编码：O(d)，因为每个字符最多访问一次。
- 编码文本：O(n)，因为需要遍历每个字符。
- 解码文本：O(m)，其中 m 是编码文本的长度。

II 空间复杂度：

- 统计字符频率：O(d)，用于存储字符和频率。
- 构建哈夫曼树：O(d)，用于存储哈夫曼树节点。
- 生成哈夫曼编码：O(d)，用于存储编码。
- 编码文本：O(n)，用于存储编码结果。
- 解码文本：O(m)，用于存储解码结果。

3. 系统结构操作流程图和函数的调用关系图

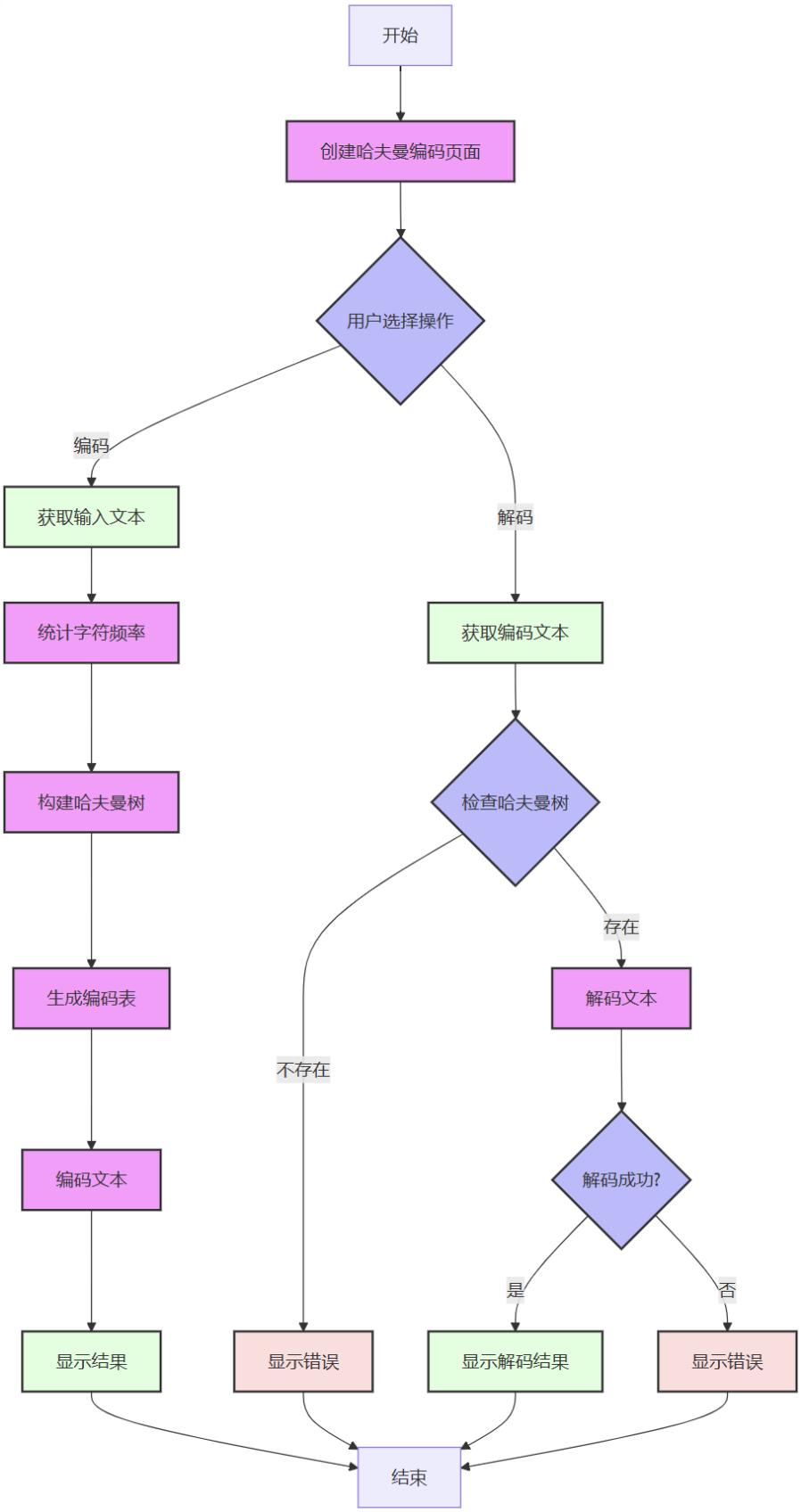

图 13-哈夫曼编码模块系统结构操作流程图

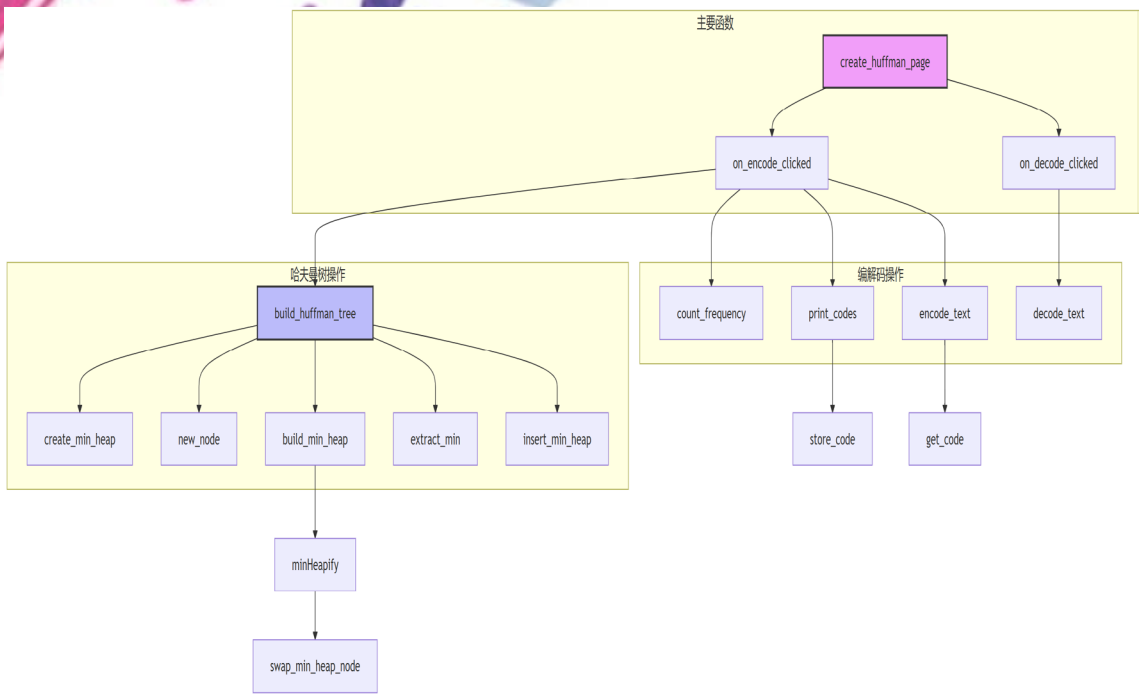

图 14-哈夫曼编码模块函数调用关系图

4. 调试分析

a、调试中遇到的问题及对问题的解决方法

I 问题：字符频率统计错误

- **现象**：编码结果不正确，可能是因为字符频率统计不准确。
- **解决方法**：检查 `count_frequency` 函数，确保所有字符的频率都被正确统计。可以通过在调试时打印出每个字符的频率来验证。

II 问题：哈夫曼树构建失败

- **现象**：程序崩溃或无法生成编码，可能是因为哈夫曼树构建过程中出现错误。
- **解决方法**：检查 `build_huffman_tree` 函数，确保最小堆的操作（如插入、提取最小值）正确无误。可以通过在调试时打印出每个节点的频率和结构来验证。

III 问题：编码表生成错误

- **现象**：生成的编码表不完整或错误。
- **解决方法**：检查 `print_codes` 函数，确保所有叶子节点的编码都被正确生成和存储。可以通过在调试时打印出每个字符的编码来验证。

IV 问题：内存管理问题

- **现象**：内存泄漏或非法访问，可能是因为动态内存管理不当。

- **解决方法**：使用工具（如 `Valgrind`）检测内存泄漏，确保在使用 `g_free` 和 `free` 时正确释放所有动态分配的内存。

V 问题：解码失败

- **现象**：解码结果不正确，可能是因为编码字符串不匹配哈夫曼树。
- **解决方法**：检查 `decode_text` 函数，确保解码过程正确遍历哈夫曼树。可以通过在调试时打印出解码过程中的每一步来验证。

b、算法的时间复杂度和空间复杂度

- **时间复杂度**：
  - **统计字符频率**：O(n)，其中 n 是输入文本的长度。
  - **构建哈夫曼树**：O(d log d)，其中 d 是不同字符的数量。
  - **生成哈夫曼编码**：O(d)，因为每个字符最多访问一次。
  - **编码文本**：O(n)，因为需要遍历每个字符。
  - **解码文本**：O(m)，其中 m 是编码文本的长度。
- **空间复杂度**：
  - **统计字符频率**：O(d)，用于存储字符和频率。
  - **构建哈夫曼树**：O(d)，用于存储哈夫曼树节点。
  - **生成哈夫曼编码**：O(d)，用于存储编码。
  - **编码文本**：O(n)，用于存储编码结果。
  - **解码文本**：O(m)，用于存储解码结果。

5. 测试结果

5.1 哈夫曼编码

输入示例：

```
210052202019 Long Zheng
```

实际运行效果：

图 15-哈夫曼编码演示

编码表（完整展示）：

```
0: 00

Z: 0100

e: 0101

空格: 011

L: 1000

9: 10010

o: 10011

5: 10100

h: 10101

1: 1011

2: 110

n: 1110

g: 1111
```

输出结果：

```
110 1011 00 00 10100 110 110 00 110 00 1011 10010 011 1000 10011 1110 1111 011 0100 10101
0101 1110 1111
```

5.2 哈夫曼解码

输入示例：

```
110 1011 00 00 10100 110 110 00 110 00 1011 10010 011 1000 10011 1110 1111 011 0100 10101
0101 1110 1111
```

实际运行效果：

图 16-哈夫曼解码演示

编码表：

同上，此处省略。

输出结果：

```
210052202019 Long Zheng
```

6. 源程序（带注释）

huffman.h 代码：

```
#ifndef HUFFMAN_H
#define HUFFMAN_H


// 基本函数声明
GtkWidget* create_huffman_page(void); // 创建哈夫曼编码页面的函数
MinHeapNode* new_node(char data, unsigned freq); // 创建新的哈夫曼树节点
MinHeap* create_min_heap(unsigned capacity); // 创建最小堆
void swap_min_heap_node(MinHeapNode** a, MinHeapNode** b); // 交换最小堆中的两个节点
void minHeapify(MinHeap* minHeap, int idx); // 对最小堆进行堆化
int is_size_one(MinHeap* minHeap); // 检查最小堆是否只剩一个节点
MinHeapNode* extract_min(MinHeap* minHeap); // 从最小堆中提取最小节点
void insert_min_heap(MinHeap* minHeap, MinHeapNode* minHeapNode); // 向最小堆中插入节点
void build_min_heap(MinHeap* minHeap); // 构建最小堆
MinHeapNode* build_huffman_tree(char data[], int freq[], int size); // 构建哈夫曼树
void print_codes(MinHeapNode* root, char* code_str, int top, GtkTextBuffer* buffer); // 打印哈夫曼编码
void count_frequency(const char* text, char* data, int* freq, int* size); // 统计字符频率
void encode_text(const char* text, GtkTextBuffer* output_buffer); // 编码文本
char* decode_text(const char* encoded_text, MinHeapNode* root); // 解码文本
void store_code(char data, char* code); // 存储字符的哈夫曼编码
char* get_code(char c); // 获取字符的哈夫曼编码
void clear_huffman_codes(void); // 清除哈夫曼编码

#endif // 结束头文件保护
```

```
26.
```

huffman.c 代码：

```
1. // 前向声明所有静态函数
2. static void free_huffman_tree(MinHeapNode* node); // 释放哈夫曼树的内存
3. static void on_encode_clicked(GtkWidget *widget, gpointer user_data); // 编码按钮的回调函数
4. static void on_decode_clicked(GtkWidget *widget, gpointer data); // 解码按钮的回调函数
5.
6. // 全局变量
7. static GtkWidget *text_view_input; // 输入文本视图
8. static GtkWidget *text_view_output; // 输出文本视图
9. static GtkWidget *text_view_codes; // 显示编码的文本视图
10. static MinHeapNode* huffman_tree = NULL; // 哈夫曼树的根节点
11.
12. static HuffmanCode huffman_codes[MAX_CHAR]; // 存储字符的哈夫曼编码
13. static int code_count = 0; // 已存储编码的数量
14.
15. // 释放哈夫曼树的内存
16. static void free_huffman_tree(MinHeapNode* node) {
17.     if (node == NULL) return; // 如果节点为空，直接返回
18.     free_huffman_tree(node->left); // 递归释放左子树
19.     free_huffman_tree(node->right); // 递归释放右子树
20.     free(node); // 释放当前节点
21. }
22.
23. // 存储编码
24. void store_code(char data, char* code) {
```

```
25.     huffman_codes[code_count].character = data; // 存储字符
26.     huffman_codes[code_count].code = g_strdup(code); // 存储编码
27.     code_count++; // 增加编码计数
28. }
29.
30. // 获取字符的编码
31. char* get_code(char c) {
32.     for (int i = 0; i < code_count; i++) { // 遍历已存储的编码
33.         if (huffman_codes[i].character == c) { // 如果找到匹配的字符
34.             return huffman_codes[i].code; // 返回对应的编码
35.         }
36.     }
37.     return NULL; // 如果未找到，返回 NULL
38. }
39.
40. // 统计字符频率
41. void count_frequency(const char* text, char* data, int* freq, int* size) {
42.     int count[MAX_CHAR] = {0}; // 初始化字符计数数组
43.     int len = strlen(text); // 获取文本长度
44.     *size = 0; // 初始化字符种类计数
45.
46.     // 统计频率
47.     for (int i = 0; i < len; i++)
48.         count[(unsigned char)text[i]]++; // 增加对应字符的计数
49.
50.     // 收集非零频率的字符
51.     for (int i = 0; i < MAX_CHAR; i++) {
52.         if (count[i] > 0) { // 如果字符出现过
53.             data[*size] = (char)i; // 存储字符
```

```
            freq[*size] = count[i]; // 存储频率
            (*size)++; // 增加字符种类计数
        }
    }
}

// 构建哈夫曼树
MinHeapNode* build_huffman_tree(char data[], int freq[], int size) {
    MinHeapNode *left, *right, *top; // 定义左右子节点和顶节点
    MinHeap* minHeap = create_min_heap(size); // 创建最小堆

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = new_node(data[i], freq[i]); // 为每个字符创建节点

    minHeap->size = size; // 设置最小堆的大小
    build_min_heap(minHeap); // 构建最小堆

    while (!is_size_one(minHeap)) { // 当最小堆中不止一个节点时
        left = extract_min(minHeap); // 提取最小节点作为左子节点
        right = extract_min(minHeap); // 提取次小节点作为右子节点
        top = new_node('$', left->freq + right->freq); // 创建新节点，频率为左右子节点之和
        top->left = left; // 设置左子节点
        top->right = right; // 设置右子节点
        insert_min_heap(minHeap, top); // 将新节点插入最小堆
    }

    return extract_min(minHeap); // 返回根节点
}
```

```
82.
83. // 打印哈夫曼编码
84. void print_codes(MinHeapNode* root, char* code_str, int top, GtkTextBuffer*
buffer) {
85.     if (root->left) { // 如果存在左子节点
86.         code_str[top] = '0'; // 在编码字符串中添加'0'
87.         print_codes(root->left, code_str, top + 1, buffer); // 递归处理左子节点
88.     }
89.
90.     if (root->right) { // 如果存在右子节点
91.         code_str[top] = '1'; // 在编码字符串中添加'1'
92.         print_codes(root->right, code_str, top + 1, buffer); // 递归处理右子节点
93.     }
94.
95.     if (!(root->left) && !(root->right)) { // 如果是叶子节点
96.         GtkTextIter end;
97.         gtk_text_buffer_get_end_iter(buffer, &end); // 获取文本缓冲区的结束迭代器
98.         char temp[256];
99.         code_str[top] = '\0'; // 终止编码字符串
100.
101.         // 存储编码
102.         store_code(root->data, code_str);
103.
104.         if (root->data == ' ')
105.             sprintf(temp, "空格: %s\n", code_str); // 特殊处理空格字符
106.         else if (root->data == '\n')
107.             sprintf(temp, "换行: %s\n", code_str); // 特殊处理换行字符
108.         else
109.             sprintf(temp, "%c: %s\n", root->data, code_str); // 处理普通字符
```

```
110.         gtk_text_buffer_insert(buffer, &end, temp, -1); // 插入编码信息
111.     }
112. }
113.
```

四．排序重构问题

方程 $A^5+B^5+C^5+D^5+E^5=F^5$ 刚好有一个满足 $0 \leqslant A \leqslant B \leqslant C \leqslant D \leqslant E \leqslant F \leqslant 75$ 的整数解。请编写一个求出该解的程序。

1. 排序重构模块采用类语言定义相关的数据类型

1.1 类图

下面是排序重构问题的类语言定义说明和类图。

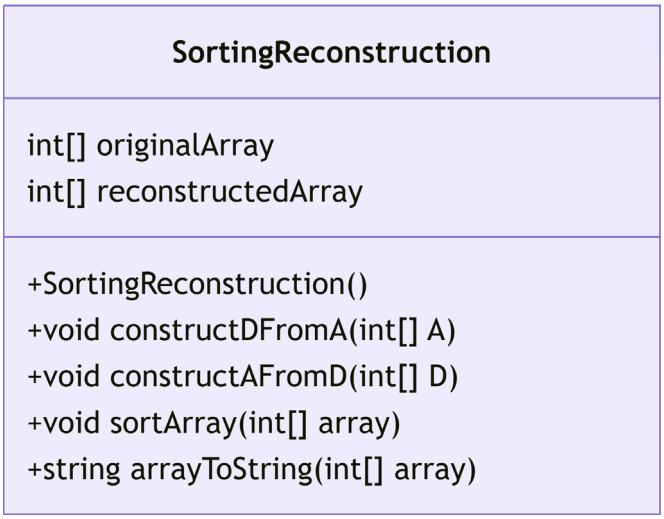

图 17-UML 排序重构模块类图

1.2 类图说明

- **SortingReconstruction 类**：负责排序重构问题的处理，包括从数组 A 构造数组 D，以及从数组 D 重构数组 A。
  - **数据成员**：
    - originalArray：存储原始数组 A。
    - reconstructedArray：存储重构后的数组 D。
  - **构造方法**：
    - SortingReconstruction()：初始化一个排序重构对象。
  - **成员方法**：

- constructDFromA(int[] A)：从数组 A 构造数组 D。
- constructAFromD(int[] D)：从数组 D 重构数组 A。
- sortArray(int[] array)：对给定数组进行排序。
- arrayToString(int[] array)：将数组转换为字符串表示。

1.3 类语言定义说明

```
模块名：SortingReconstructionSystem（排序重构系统）

类名：SortingReconstruction（排序重构类）
----------------------------------------
数据成员：
    - originalArray: int[]        // 存储原始数组 A
    - reconstructedArray: int[]   // 存储重构后的数组 D

构造方法：
    + SortingReconstruction()
        功能：初始化一个排序重构对象
        参数：无
        返回：SortingReconstruction 对象

成员方法：
    + void constructDFromA(int[] A)
        功能：从数组 A 构造数组 D
        参数：A - 原始数组
        返回：无

    + void constructAFromD(int[] D)
        功能：从数组 D 重构数组 A
        参数：D - 差分数组
```

```
        返回：无

    + void sortArray(int[] array)
        功能：对给定数组进行排序
        参数：array - 需要排序的数组
        返回：无

    + string arrayToString(int[] array)
        功能：将数组转换为字符串表示
        参数：array - 需要转换的数组
        返回：格式化的字符串

```

这个类语言定义和类图展示了排序重构问题的结构和功能，模拟了面向对象的设计。

2. 排序重构模块算法设计

2.1 算法设计

I 从数组 A 构造数组 D：

- 输入：数组 A，大小 N
- 输出：数组 D，大小 D_size
- 伪代码：

```
function construct_D_from_A(A, N):
    if A[0] != 0:
        throw "A 序列的第一个数必须为 0"
    initialize D as empty list
    for i from 0 to N-1:
        for j from i+1 to N-1:
            D.append(A[j] - A[i])
```

```
    return D

```

II 从数组 D 重构数组 A：

- 输入：数组 D，大小 D_size
- 输出：数组 A，大小 A_size
- 伪代码：

```
function construct_A_from_D(D, D_size):
    sort D
    initialize A as empty list
    A.append(0)
    for each value in D:
        if value not in A:
            A.append(value)
    return A

```

III 数组排序：

- 输入：数组
- 输出：排序后的数组
- 伪代码：

```
function sort_array(array):
    use quicksort to sort array

```

2.2 算法复杂度分析

1. **时间复杂度：**
   - **从数组 A 构造数组 D：**O(N^2)，因为需要遍历数组 A 的所有可能的(i, j)对。
   - **从数组 D 重构数组 A：**O(D_size log D_size)，因为需要对数组

D 进行排序。

- **数组排序**：O(n log n)，使用快速排序。

2. **空间复杂度**：
   - **从数组 A 构造数组 D**：O(D_size)，用于存储结果数组 D。
   - **从数组 D 重构数组 A**：O(A_size)，用于存储结果数组 A。
   - **数组排序**：O(1)，如果使用原地排序算法。

3．系统结构操作流程图和函数的调用关系图

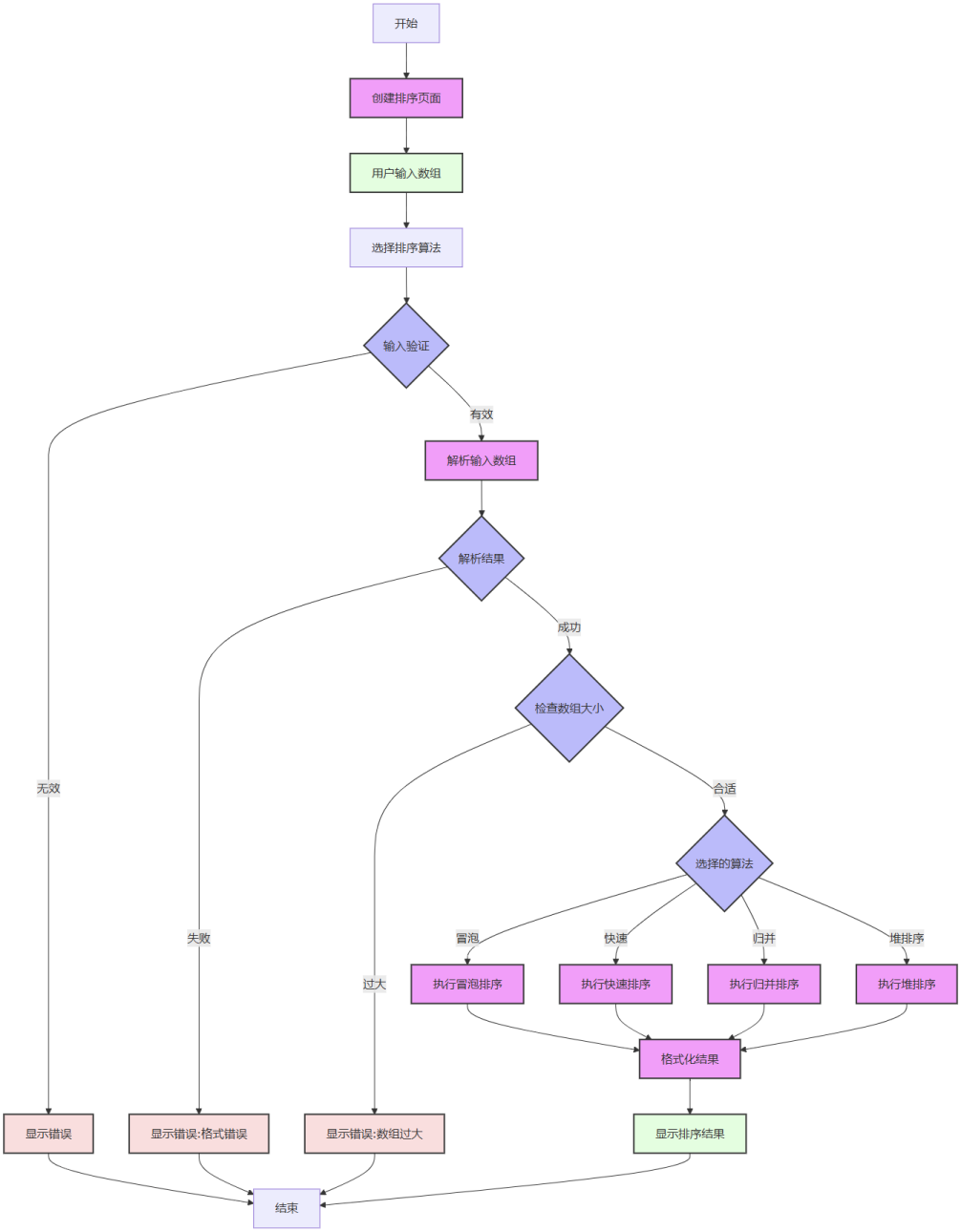

图 18-排序重构模块系统结构操作流程图

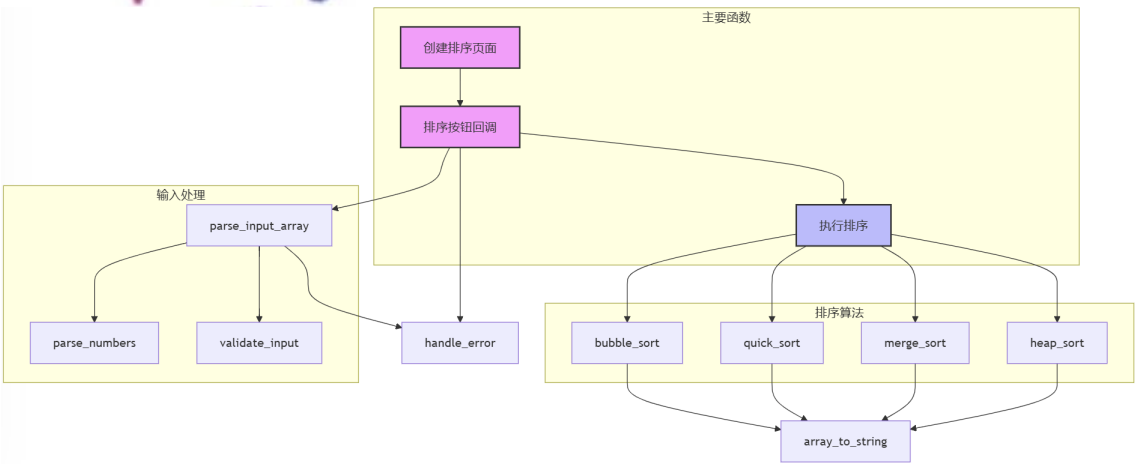

图 19-排序重构模块函数关系调用图

4. 调试分析

a、调试中遇到的问题及对问题的解决方法

I 问题：输入验证不足

- **现象**：程序在处理不符合预期格式的输入时崩溃或产生错误结果。
- **解决方法**：在处理输入之前，增加输入验证步骤，确保输入数组的格式和内容符合预期。可以通过在调试时打印输入数组来验证。

II 问题：数组越界

- **现象**：程序在访问数组元素时崩溃，可能是因为数组越界。
- **解决方法**：检查数组访问的索引，确保不超过数组的实际大小。可以通过在调试时打印数组大小和访问索引来验证。

III 问题：内存管理问题

- **现象**：内存泄漏或非法访问，可能是因为动态内存管理不当。
- **解决方法**：使用工具（如 Valgrind）检测内存泄漏，确保在使用 free 时正确释放所有动态分配的内存。

IV 问题：排序算法错误

- **现象**：排序结果不正确，可能是因为排序算法实现有误。
- **解决方法**：检查 sort_array 函数，确保排序算法（如快速排序）实现正确。可以通过在调试时打印排序前后的数组来验证。

V 问题：构造 D 或 A 时逻辑错误

- **现象**：构造的数组 D 或 A 不正确，可能是因为算法逻辑错误。
- **解决方法**：检查 construct_D_from_A 和 construct_A_from_D 函数，确保算法逻辑正确。可以通过在调试时打印中间结果来验证。

b、算法的时间复杂度和空间复杂度

- **时间复杂度**：
  - **从数组 A 构造数组 D**：O(N^2)，因为需要遍历数组 A 的所有可能的(i, j)对。
  - **从数组 D 重构数组 A**：O(D_size log D_size)，因为需要对数组 D 进行排序。
  - **数组排序**：O(n log n)，使用快速排序。
- **空间复杂度**：
  - **从数组 A 构造数组 D**：O(D_size)，用于存储结果数组 D。
  - **从数组 D 重构数组 A**：O(A_size)，用于存储结果数组 A。
  - **数组排序**：O(1)，如果使用原地排序算法。

5. 测试结果

输入示例（用于 A 来构造 D）：

```
0 2 5
```

预测结果：

```
2 3 5
```

实际结果

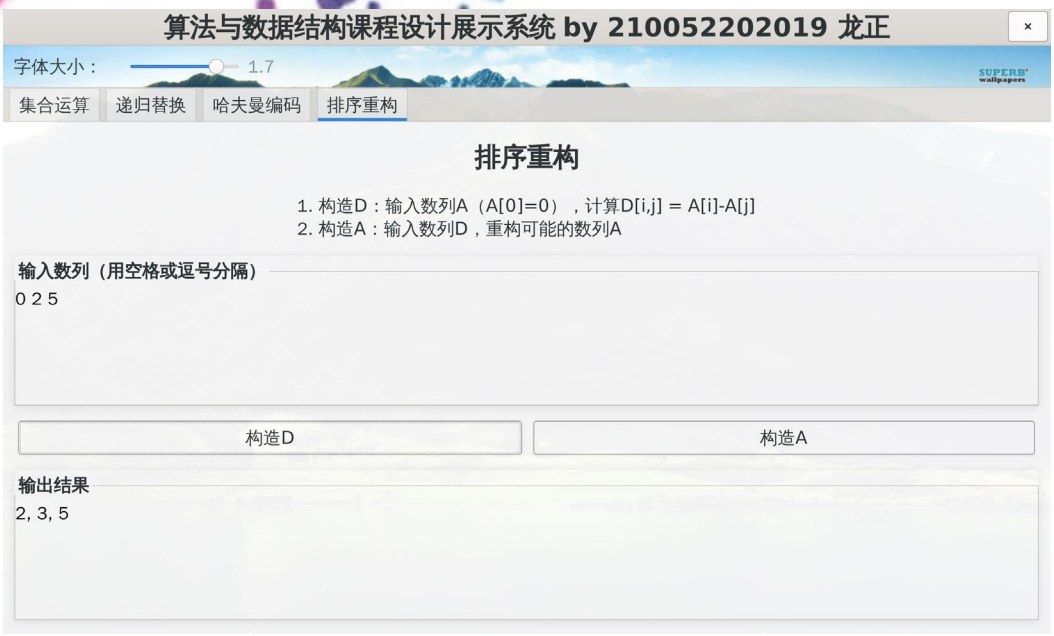

图 20-排序重构模块 A 构造 D 演示

输入示例（用于 D 来构造 A）

```
2 3 5
```

预测示例：

```
0 2 5
```

实际结果：

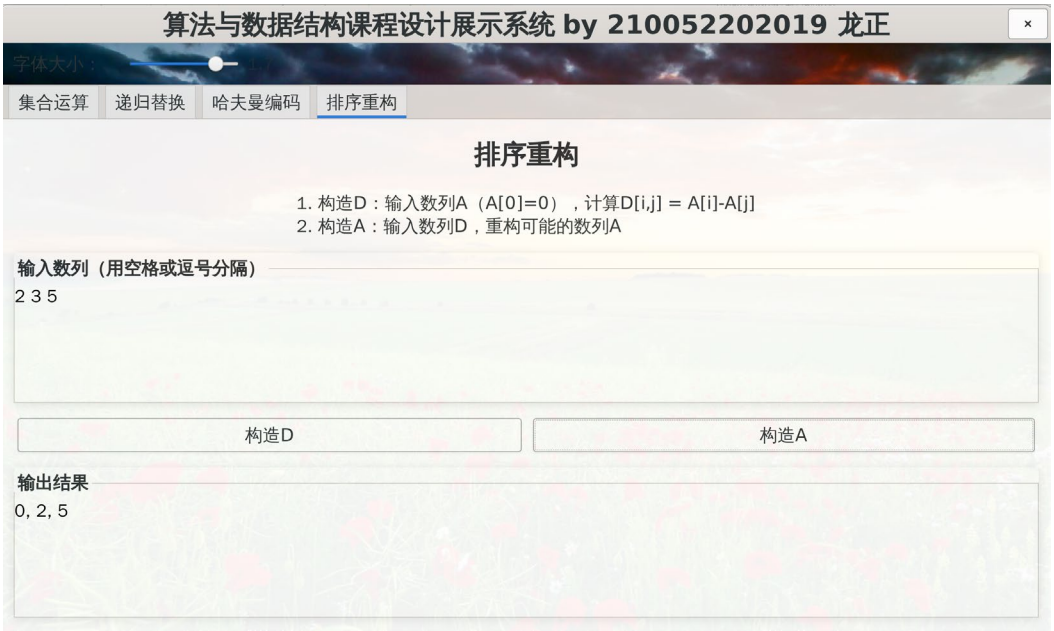

图 21-排序重构模块 D 构造 A 演示

6. 源程序（带注释）

sorting.h 代码：

```
1. #ifndef SORTING_H
2. #define SORTING_H
3.
4. #define MAX_ARRAY_SIZE 100 // 定义数组的最大大小
5. #define MAX_NUMBERS 100  // 定义最大数字数量
6.
7. // 函数声明
8. GtkWidget* create_sorting_page(void); // 创建排序页面的函数
9. void construct_D_from_A(const int* A, int N, int* D, int* D_size); // 从数组 A 构造数组 D
10. void construct_A_from_D(const int* D, int D_size, int* A, int* A_size); // 从数组 D 重构数组 A
11. void sort_array(int* arr, int size); // 对数组进行排序
12.
13. #endif // 结束头文件保护
14.
```

sorting.c

```
1. #define _GNU_SOURCE
2.
3. static GtkWidget *text_view_input; // 输入文本视图
4. static GtkWidget *text_view_output; // 输出文本视图
5.
6. // 比较函数用于 qsort
7. static int compare_ints(const void* a, const void* b) {
```

```
    return (*(int*)a - *(int*)b); // 返回两个整数的差值，用于排序
}

// 从 A 构造 D
void construct_D_from_A(const int* A, int N, int* D, int* D_size) {
    if (!A || !D || !D_size || N <= 0) { // 检查输入参数的有效性
        handle_error(NULL, ERROR_INVALID_INPUT, "无效的输入参数");
        return;
    }

    // 检查 A[0]是否为 0
    if (A[0] != 0) {
        handle_error(NULL, ERROR_INVALID_INPUT, "A 序列的第一个数必须为 0");
        return;
    }

    if (N > MAX_ARRAY_SIZE) { // 检查数组大小是否超过最大限制
        handle_error(NULL, ERROR_BUFFER_OVERFLOW, "输入数组过大");
        return;
    }

    // 计算 D 序列的大小
    int expected_size = (N * (N - 1)) / 2; // 计算组合数 C(N, 2)
    if (expected_size > MAX_ARRAY_SIZE) { // 检查结果数组大小是否超过最大限制
        handle_error(NULL, ERROR_BUFFER_OVERFLOW, "结果数组将超出最大限制");
        return;
    }

    *D_size = 0; // 初始化 D 的大小
```

```
    for (int i = 0; i < N; i++) { // 遍历数组 A
        for (int j = i + 1; j < N; j++) { // 遍历 A 中每个元素之后的元素
            D[(*D_size)++] = A[j] - A[i]; // 计算差值并存储在 D 中
        }
    }
}

// 从 D 构造 A
void construct_A_from_D(const int* D, int D_size, int* A, int* A_size) {
    if (!D || !A || !A_size || D_size <= 0) { // 检查输入参数的有效性
        handle_error(NULL, ERROR_INVALID_INPUT, "无效的输入参数");
        return;
    }

    // 对 D 进行排序
    qsort(D, D_size, sizeof(int), compare_ints); // 使用快速排序对 D 进行排序

    *A_size = 0; // 初始化 A 的大小
    A[(*A_size)++] = 0;  // A[0]必须为 0

    for (int i = 0; i < D_size; i++) { // 遍历数组 D
        if (!contains(A, *A_size, D[i])) { // 检查 D[i]是否已在 A 中
            A[(*A_size)++] = D[i]; // 如果不在，则添加到 A 中
        }
    }
}

// 数组排序
void sort_array(int* arr, int size) {
```

```
66.     if (!arr || size <= 0) { // 检查输入参数的有效性
67.         handle_error(NULL, ERROR_INVALID_INPUT, "无效的输入参数");
68.         return;
69.     }
70.
71.     qsort(arr, size, sizeof(int), compare_ints); // 使用快速排序对数组进行排序
72. }
73.
```

五．其他模块 （GUI 和异常处理机制模块）

用于统一管理程序错误，基于 GTK+3.0 实现的图形用户界面和采用 TRY-CATCH-FINALLY 机制的异常错误处理系统，提供了统一的界面交互和错误处理框架，包含背景图片自动切换、标签页管理、错误对话框显示和日志记录等功能，确保了程序运行的稳定性和用户体验的友好性。

1. 采用类语言定义相关的数据类型

1.1 类图

下面是 GUI 和异常处理机制模块的类语言定义说明。

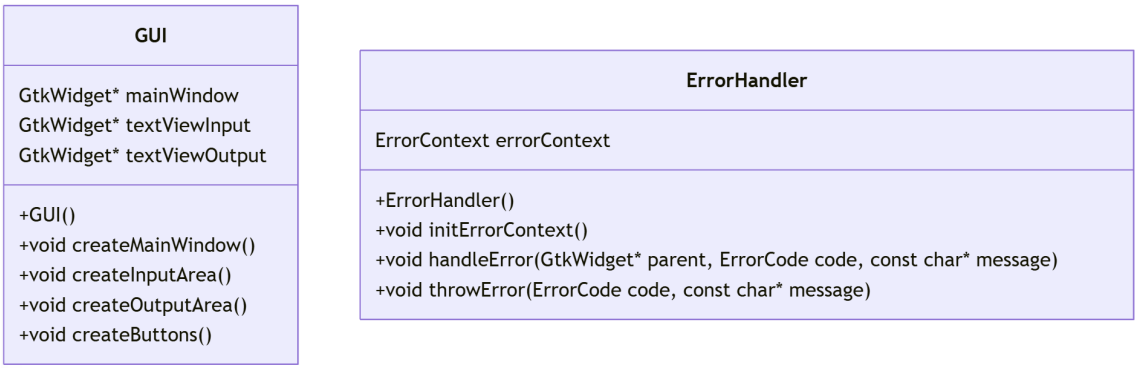

图 22-UML 其他模块类图

1.2 类图说明

- **GUI 类**：负责创建和管理图形用户界面。
  - **数据成员**：
    - mainWindow：主窗口的指针。
    - textViewInput：输入文本视图的指针。
    - textViewOutput：输出文本视图的指针。
  - **构造方法**：
    - GUI()：初始化 GUI 对象。
  - **成员方法**：
    - createMainWindow()：创建主窗口。
    - createInputArea()：创建输入区域。
    - createOutputArea()：创建输出区域。
    - createButtons()：创建操作按钮。
- **ErrorHandler 类**：负责异常处理和错误管理。
  - **数据成员**：
    - errorContext：错误上下文，用于管理错误状态。
  - **构造方法**：
    - ErrorHandler()：初始化异常处理对象。
  - **成员方法**：
    - initErrorContext()：初始化错误上下文。
    - handleError(GtkWidget* parent, ErrorCode code, const char* message)：处理错误并显示错误信息。
    - throwError(ErrorCode code, const char* message)：抛出错误。

1．3 类语言定义说明

```
1. 模块名：GUISystem（图形用户界面系统）
2.
```

```
3. 类名：GUI（图形用户界面类）
4. ----------------------------------------
5. 数据成员：
6.     - mainWindow: GtkWidget*       // 主窗口指针
7.     - textViewInput: GtkWidget*    // 输入文本视图指针
8.     - textViewOutput: GtkWidget*   // 输出文本视图指针
9.
10. 构造方法：
11.     + GUI()
12.         功能：初始化 GUI 对象
13.         参数：无
14.         返回：GUI 对象
15.
16. 成员方法：
17.     + void createMainWindow()
18.         功能：创建主窗口
19.         参数：无
20.         返回：无
21.
22.     + void createInputArea()
23.         功能：创建输入区域
24.         参数：无
25.         返回：无
26.
27.     + void createOutputArea()
28.         功能：创建输出区域
29.         参数：无
30.         返回：无
31.
```

```
    + void createButtons()
        功能：创建操作按钮
        参数：无
        返回：无

模块名：ErrorHandlingSystem（异常处理系统）

类名：ErrorHandler（异常处理类）
----------------------------------------
数据成员：
    - errorContext: ErrorContext   // 错误上下文

构造方法：
    + ErrorHandler()
        功能：初始化异常处理对象
        参数：无
        返回：ErrorHandler 对象

成员方法：
    + void initErrorContext()
        功能：初始化错误上下文
        参数：无
        返回：无

    + void handleError(GtkWidget* parent, ErrorCode code, const char* message)
        功能：处理错误并显示错误信息
        参数：parent - 父窗口指针，code - 错误代码，message - 错误信息
        返回：无
```

```
+ void throwError(ErrorCode code, const char* message)
    功能：抛出错误
    参数：code - 错误代码，message - 错误信息
    返回：无

```

根据以上内容，对 C 语言进行类似面向对象编程的模拟操作。

2. GUI 和异常处理机制模块的算法设计

2.1GUI 主程序算法

- 算法：InitializeGUI()
- 输入：无
- 输出：GUI 应用程序窗口

```
1. 初始化 GTK 环境
2. 创建主窗口并设置属性（标题、大小、位置）
3. 创建标签页容器（Notebook）
4. FOR 每个功能模块 DO 4.1 创建对应的页面 4.2 添加到标签页容器
5. 加载并设置背景图片
6. 注册信号处理函数
7. 显示窗口
8. 进入 GTK 主循环
9.
```

2.2 异常处理机制算法

- 算法：ErrorHandler()
- 输入：错误类型、错误信息、错误上下文输出：错误处理结果

```
1. 初始化错误上下文结构
```

```
2. 设置异常捕获点
3. IF 发生异常 THEN
4.获取错误类型和信息
5. CASE 错误类型 OF
6. GUI 错误：显示错误对话框
7. 系统错误：写入错误日志
8. 致命错误：终止程序
9. 3.3 执行资源清理
10. 返回错误处理结果
11.
```

2.3 资源管理算法

- 算法：ResourceManager()
- 输入：资源类型、操作类型
- 输出：资源操作结果

```
1. 初始化资源管理器
2. CASE 操作类型 OF
3. 分配：分配请求的资源
4. 释放：释放指定的资源
5. 清理：清理所有资源
6. 记录资源状态
7. 返回操作结果
8.
```

算法特点：

- 模块化设计
- 事件驱动
- 统一的错误处理
- 自动的资源管理
- 可扩展的界面结构

3．系统操作流程图和系统函数调用关系图

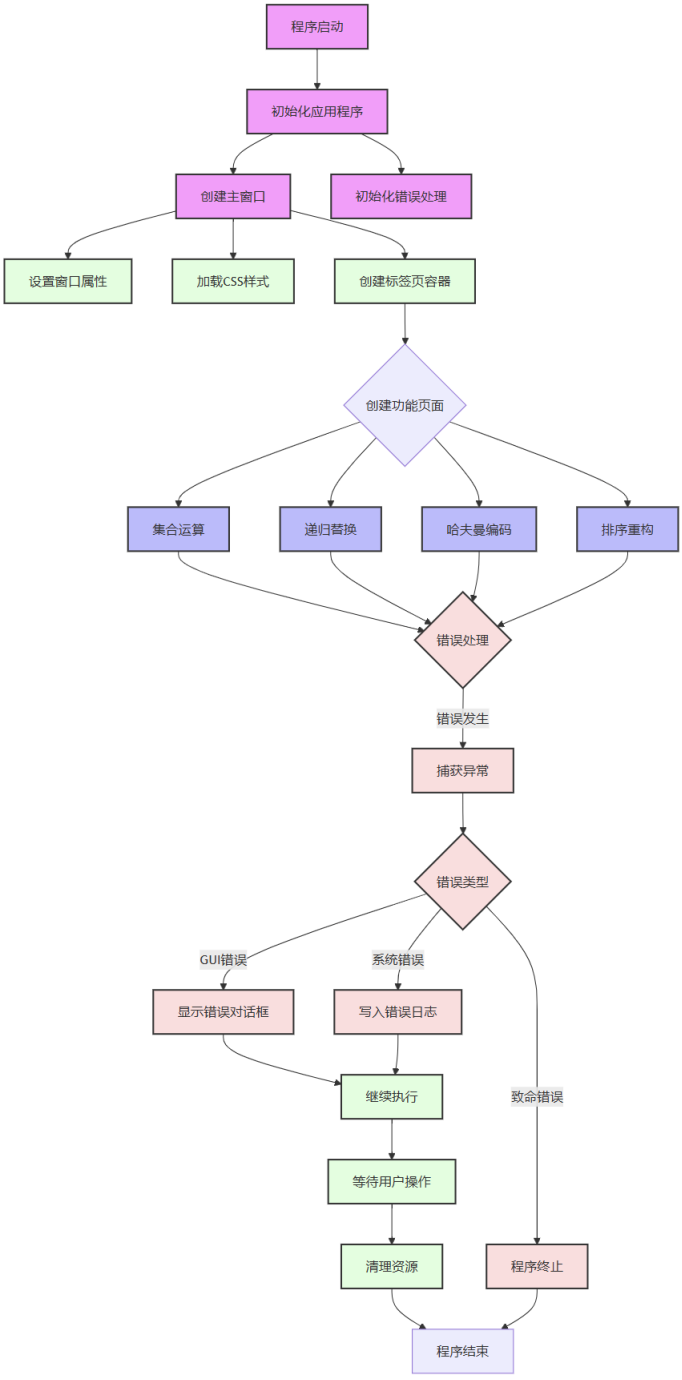

图 23-其他模块系统操作流程图

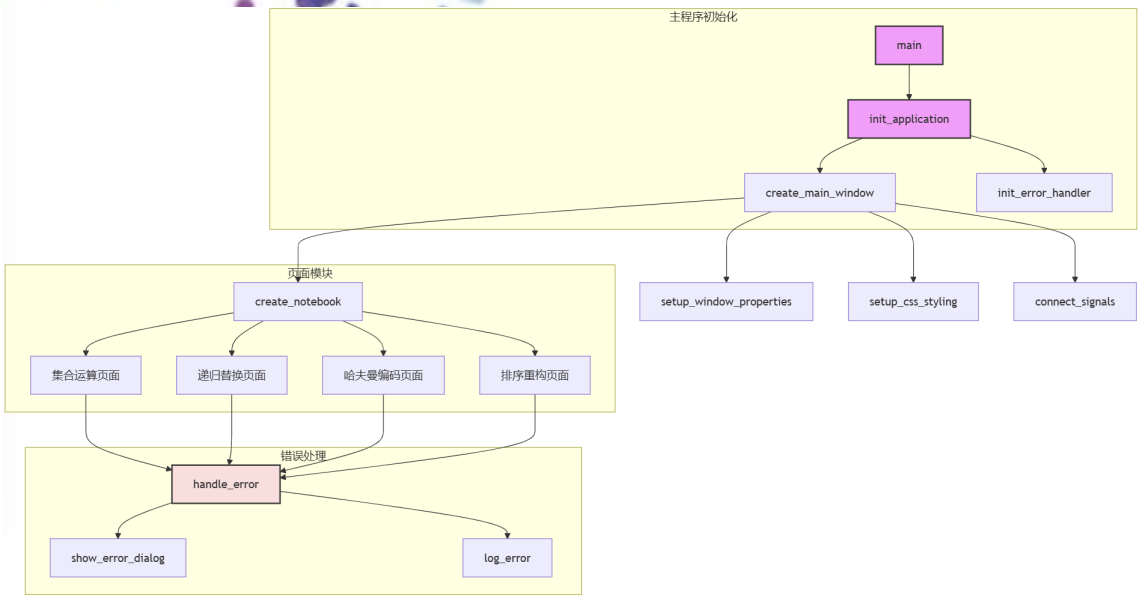

图 24-其他模块函数调用关系图

4. 测试结果

输入不符合要求的类型，会弹出错误提示，而不会导致程序中断。

图 25-异常处理机制错误演示

GUI 界面的背景正常自动切换，同时字体大小能全局调节；而且调整窗口尺寸时，

背景图片以及字体会跟随一起缩放。

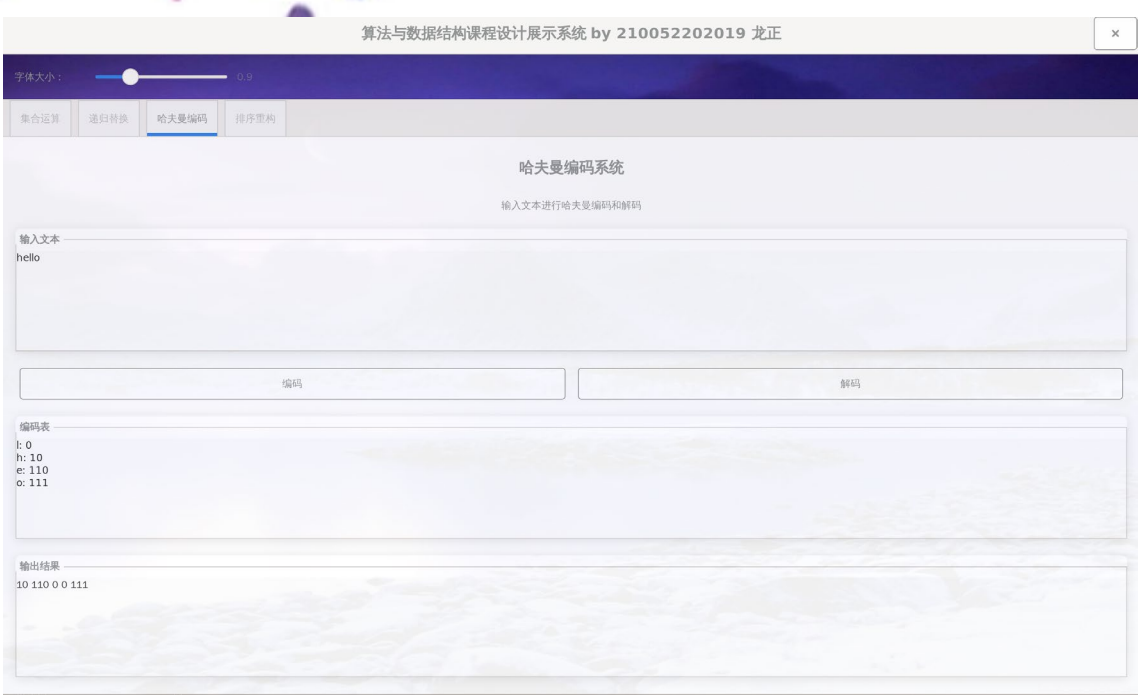

图 26-GUI 字体大小调节演示

5. 源程序（带注释）

main.c 中的 GUI 代码：

```
1. // 定义应用程序上下文结构体
2. typedef struct {
3.     GtkWidget *window; // 主窗口
4.     GtkWidget *background_image; // 背景图片
5.     GtkWidget *content_box; // 内容盒子
6.     GtkWidget *notebook; // 选项卡控件
7.     GtkWidget *font_scale; // 字体缩放控件
8.     char **image_paths; // 背景图片路径数组
9.     int image_count; // 背景图片数量
10.    double scale_factor; // 缩放因子
11.    GdkPixbuf *current_pixbuf; // 当前显示的图片
12.    int base_font_size; // 基础字体大小
```

```
    double font_scale_factor; // 字体缩放因子
} AppContext;

static AppContext app_ctx; // 全局应用程序上下文

// 更新背景图片
static gboolean update_background(gpointer data) {
    if (app_ctx.image_count <= 0) { // 检查是否有背景图片
        g_print("No background images found!\n");
        return G_SOURCE_CONTINUE; // 继续定时器
    }

    // 获取窗口当前大小
    int width, height;
    gtk_window_get_size(GTK_WINDOW(app_ctx.window), &width, &height);

    // 选择随机背景图片
    int index = rand() % app_ctx.image_count;
    GdkPixbuf *pixbuf = gdk_pixbuf_new_from_file_at_scale(
        app_ctx.image_paths[index], width, height, FALSE, NULL);

    if (pixbuf) {
        gtk_image_set_from_pixbuf(GTK_IMAGE(app_ctx.background_image), pixbuf);
        g_object_unref(pixbuf); // 释放 pixbuf 资源
    }

    return G_SOURCE_CONTINUE; // 继续定时器
}

```

main.c 中异常处理代码：

```c
// 错误处理示例
void some_function() {
    ErrorContext error_ctx;
    init_error_context(&error_ctx); // 初始化错误上下文

    TRY(&error_ctx) {
        // 可能抛出错误的代码
        if (some_condition) {
            THROW(&error_ctx, ERROR_INVALID_INPUT, "Invalid input detected");
        }
    }
    CATCH(&error_ctx) {
        handle_error(app_ctx.window, error_ctx.code, error_ctx.message); // 处理错误
    }
    FINALLY({
        // 清理代码
    });
}

```

error_handler.h 代码

```c
#ifndef ERROR_HANDLER_H
#define ERROR_HANDLER_H

// 错误代码枚举
typedef enum {
    ERROR_NONE,
    ERROR_INVALID_INPUT,
```

```
    ERROR_MEMORY_ALLOCATION,
    ERROR_FILE_NOT_FOUND,
    ERROR_INVALID_OPERATION,
    ERROR_BUFFER_OVERFLOW
} ErrorCode;

// 错误上下文结构体
typedef struct {
    ErrorCode code; // 错误代码
    char message[256]; // 错误信息
} ErrorContext;

// 初始化错误上下文
void init_error_context(ErrorContext *ctx) {
    ctx->code = ERROR_NONE; // 初始化为无错误
    ctx->message[0] = '\0'; // 清空错误信息
}

// 处理错误
void handle_error(GtkWidget *parent, ErrorCode code, const char *message) {
    GtkWidget *dialog = gtk_message_dialog_new(GTK_WINDOW(parent),
        GTK_DIALOG_DESTROY_WITH_PARENT,
        GTK_MESSAGE_ERROR,
        GTK_BUTTONS_CLOSE,
        "Error: %s", message);
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}

```

```
#endif

```

error_handler.c 代码

```c
// 初始化错误上下文
void init_error_context(ErrorContext *ctx) {
    ctx->code = ERROR_NONE; // 将错误代码初始化为无错误
    ctx->message[0] = '\0'; // 清空错误信息字符串
    ctx->has_error = FALSE; // 初始化错误标志为 FALSE
}

// 处理错误
void handle_error(GtkWidget *parent, ErrorCode code, const char *message) {
    // 创建一个错误消息对话框
    GtkWidget *dialog = gtk_message_dialog_new(GTK_WINDOW(parent),
        GTK_DIALOG_DESTROY_WITH_PARENT, // 关闭对话框时销毁
        GTK_MESSAGE_ERROR, // 设置消息类型为错误
        GTK_BUTTONS_CLOSE, // 使用关闭按钮
        "Error: %s", message); // 显示错误信息
    gtk_dialog_run(GTK_DIALOG(dialog)); // 运行对话框，等待用户响应
    gtk_widget_destroy(dialog); // 销毁对话框
}

// 获取错误代码对应的字符串
const char* get_error_string(ErrorCode code) {
    switch (code) {
        case ERROR_NONE:
            return "No error";
        case ERROR_INVALID_INPUT:
            return "Invalid input";
```

```
        case ERROR_MEMORY_ALLOCATION:
            return "Memory allocation failed";
        case ERROR_FILE_NOT_FOUND:
            return "File not found";
        case ERROR_INVALID_OPERATION:
            return "Invalid operation";
        case ERROR_BUFFER_OVERFLOW:
            return "Buffer overflow";
        default:
            return "Unknown error";
    }
}

// 记录错误信息
void log_error(ErrorCode code, const char *message) {
    // 打印错误信息到标准错误输出
    fprintf(stderr, "Error [%d]: %s - %s\n", code, get_error_string(code),
message);
}

```

总　结

通过本次课程设计的规划与实施，在杜玉红老师的悉心指导下，我成功开发了一个功能完善且用户友好的算法与数据结构展示系统。整个项目从需求分析到最终测试，不仅实现了预期的功能目标，还让我在编程能力和软件工程素养方面得到了显著提升。

在开发过程中，我深入学习并应用了 GTK 图形界面库。这让我掌握了如何创建复杂且响应式的用户界面，并熟悉了窗口布局、事件处理以及 CSS 样式的应用。在杜老师的点拨下，我顺利完成了动态背景管理和实时字体大小调整的功能，通过灵活运用 GTK 提供的 API，不仅提升了界面的美观性，也增强了用户体验设计方面的敏感度。

项目的核心围绕算法与数据结构展开，解决了几个关键问题。其中，集合运算模块可以实现集合的并集、交集、差集以及幂集求解，确保结果集合中元素无重复；递归替换模块通过递归扩展#include 指令，有效解决了复杂嵌套文件的解析问题；哈夫曼编码与解码模块建立了高效的哈夫曼树，实现了文本信息的自动编码和解码；排序重构模块则能够通过数学分析实现从数列 A 构造差分数列 D 以及从差分数列 D 重构数列 A 的功能。杜老师的悉心指导在这些模块的逻辑分析和实现过程中发挥了重要作用，每一个模块的设计都让我对算法逻辑有了更深入的理解，同时增强了工程实践能力。

整个系统采用了模块化设计，将功能划分为主模块、错误处理模块及各个算法演示模块。这样的架构设计提高了代码的可维护性和扩展性，同时也让我充分体会到高内聚、低耦合设计的优越性。在杜老师的建议下，我设计了统一的错误管理机制，将所有的错误处理集中到专门的模块中，不仅提高了代码的清晰度，也为后续功能扩展打下了良好的基础。各个算法演示模块则是独立开发和测试的，既保障了模块的稳定性，也确保了它们之间的互不干扰。

开发过程中，我遇到了不少技术难题，比如背景图片的动态加载与缩放、内存管理以及错误捕获机制的实现等。在杜老师的鼓励和指导下，我查阅了大量技术文档，参考开源项目，并积极参与社区讨论，最终找到了解决方案。比如，在实现背

景图片的动态加载功能时，我充分理解了 GdkPixbuf 对象的引用计数机制，并结合 gdk_pixbuf_scale_simple 函数，实现了图片的高效缩放和内存管理，避免了内存泄漏问题。统一的错误处理机制则通过 setjmp 和 longjmp 函数模拟了类似高级语言中的异常捕获功能，显著提升了系统的健壮性。

为了管理项目，我采用了 Git 进行版本控制。这种方式让我能够清晰地跟踪代码的变更记录，随时恢复到历史版本，减少了因误操作带来的困扰。在杜老师的建议下，通过详细的提交记录，我对开发进度有了更清晰的掌握，这种习惯在后续的开发工作中无疑将大有裨益。

在测试阶段，我编写了详尽的单元测试代码，确保每个模块的功能都能正确运行。通过模拟各种使用场景，我验证了系统的可靠性，并针对性地优化了性能问题，比如解决了界面响应迟缓和内存泄漏等问题。反复的测试和调试，让系统在美观性和实用性上都达到了更高的标准。

这次课程设计让我更加深刻地理解了算法与数据结构的理论与应用，也在实践中锻炼了模块化设计、系统集成以及资源管理等技能。在杜玉红老师的指导下，整个开发过程从零开始设计并实现完整的系统，我不仅完成了一个复杂的项目，还积累了宝贵的经验，学会了如何面对和解决技术难题。

此次课程设计的完成，不仅实现了教学目标，更让我对软件开发的全流程有了更清晰的认识。这次独立完成的项目，不仅展示了算法与数据结构的实际应用，还让我在综合素质和专业能力上取得了长足进步，为未来的学习和职业发展打下了坚实基础。这次宝贵的实践经历，也让我更加坚定了未来深耕技术领域的信心和决心。

参考文献

（注：参考文献的格式如下：

[1]梁路宏，艾海舟，何克忠.基于多模板匹配的单人脸检测[J].中国图象图形学报，1999，4A(10):823-830.

[2] 胡珊珊.基于遗传算法的人脸检测研究[M].青岛:青岛大学出版社,2006）

[3] 严蔚敏，吴伟民.《数据结构（C 语言版）》.清华大学出版社.

[4] 严蔚敏，吴伟民.《数据结构题集（C 语言版）》.清华大学出版社.

[5]《DATA STRUCTURE WITH C++》. William Ford,William Topp .清华大学出版社（影印版）.

[6]谭浩强.《c 语言程序设计》. 清华大学出版社.

[7]数据结构与算法分析（Java 版） ，A Practical Introduction to Data Structures and Algorithm Analysis Java Edition Clifford A. Shaffer ， 张铭,刘晓丹译 电子工业出版社 2001 年 1 月

致　谢

本次课程设计的顺利完成离不开杜玉红老师的悉心指导和无私帮助。在项目的每一个阶段，从需求分析到系统实现，再到测试优化，杜老师都给予了我极大的关心和支持。她渊博的学识和严谨的治学态度不仅让我受益匪浅，也为我树立了学习和研究的榜样。在设计过程中，无论遇到技术难题还是方法选择上的困惑，杜老师总能给予耐心的解答和启发性的建议，帮助我不断突破自我，顺利完成了这一复杂的课程设计项目。

同时，我还要感谢课程组提供的学习和实践机会，为我们创造了良好的开发环境和条件。在学习 GTK 图形界面库以及实现动态背景管理、递归替换等模块功能的过程中，我深刻感受到理论联系实际的重要性，特别是在解决问题的过程中积累了宝贵的经验。这些都离不开课程组精心安排的教学资源和丰富的课外指导。

此外，在项目开发过程中，我查阅了大量技术文档并参考了许多开源项目，也从相关技术社区获得了很多灵感和帮助。感谢这些开源贡献者和社区成员，他们的知识分享让我在开发中受益良多。

最后，我要感谢我的家人和朋友在这一阶段给予我的鼓励和支持。正是他们的陪伴与关怀，让我在完成课程设计的过程中充满信心与动力。

衷心感谢所有帮助过我的人！这次课程设计不仅是一次职业技能上的提升，更是我个人成长中的一段重要经历。