

南京审计大学计算机学院

计算机科学与技术系

数据结构课程设计报告

课程设计题目： 铁路票务管理

姓 名： 金太宇

学 号： 222090218

指 导 教 师： 陈一飞

完 成 时 间： 2023 年 12 月 29 日星期五

目 录

1 需求分析.....	4
1.1 课程设计内容.....	4
1.2 系统功能需求分析.....	4
2 系统设计.....	6
2.1 数据结构及文件设计.....	6
2.2 功能模块的设计.....	8
2.2.1 DataEntry.cpp/h 模块.....	8
2.2.2 LinkList.h/tpp 模块.....	8
2.2.3 LinkQueue.cpp/h 模块.....	9
2.2.4 Login.cpp/h 模块	9
2.2.5 Search.cpp/h 模块.....	9
2.2.6 Ticket.cpp/h 模块.....	9
2.2.7 Tree.h/tpp 模块	9
2.2.8 数据文件夹 (data/).....	9
3 系统调试与测试.....	11
3.1 调试过程遇到问题及解决方案.....	11
3.2 测试功能展示.....	14
4 分析优化与改进.....	15
4.1 算法性能分析.....	15
4.1.1 PurchaseTicket 函数	16
4.1.2 RefundTicket 函数.....	16
4.1.3 Timetable 函数.....	17
4.2 优化与改进.....	19
4.2.1 Timetable 函数.....	19
4.2.2 SqList 泛型模板	19
4.2.3 未来优化目标.....	20

5	总结.....	20
附	录.....	21

1 需求分析

1.1 课程设计内容

本项目旨在综合应用数据结构知识，如线性表、队列、树、查找、排序和图等，以提高程序的分析、设计、实现及测试能力。项目的核心为开发一个铁路票务管理系统，要求如下：

1. 所有数据使用文件存储。
2. 至少实现一种树形结构。
3. 至少使用一种高效的排序算法。

1.2 系统功能需求分析

本项目的目标是开发一个全面、高效且用户友好的铁路票务管理系统。它将涉及复杂的数据结构和算法来确保数据的有效存储和快速检索，同时提供一个直观、易用的界面给最终用户。系统将充分利用树形结构来组织车次和车站数据，同时利用图结构来管理城市间的交通网络。此外，项目还将包含高效的排序和搜索算法来优化查询性能。根据要求设计的功能需求如下：

1. 登录

普通用户

- 提供注册功能，包括邮箱验证或手机短信验证。
- 登录时支持忘记密码功能，通过绑定的邮箱或手机重置密码。
- 可以通过社交媒体账号（如 Facebook, Google 等）登录。

管理员

- 专门的登录界面，提供更高安全性，如两步验证。
- 管理员操作日志记录，以追踪系统更改历史。

2. 录入/修改

用户个人信息

- 支持上传个人头像，增强用户体验。
- 提供紧急联系人信息录入，用于紧急情况。

车辆类别信息

- 能够根据市场变化或季节性需求动态调整车辆类别。
- 提供车辆类别的详细描述，如座位配置、设施等。

车次信息

- 车次信息维护应包括临时调整、取消或增加车次的能力。
- 提供车次实时状态更新，如延迟、取消等信息。

城市信息

- 包括城市旅游指南或推荐，提高用户体验。
- 实时更新交通状况，如施工、交通管制信息。

3. 查询

时刻表查询

- 提供实时时刻表更新。
- 支持语音搜索和自然语言处理。

票价查询

- 提供不同优惠方案，如学生票、老年票。

换乘时间查询

- 集成本地交通信息，提供更准确的换乘时间。

车站经过车次查询

- 支持地图视图，直观显示车站及经过车次。

个人订单查询

- 提供历史订单查询和打印功能。

4. 站间最优查询

- 提供基于预算、时间或舒适度的多重选择。
- 集成天气信息，提供出行建议。

5. 购票

当天车票

- 提供实时座位选择和视图。

购票后支付

- 支持多种支付方式，如信用卡、支付宝等。

预售车票

- 提供提醒服务，如购票成功通知。

6. 退票

- 提供在线自助退票服务。
- 清晰显示退票费用和退款时间。

7. 改签

- 实现快速改签功能，减少用户等待时间。
- 提供改签费用预估。

8. 管理员功能

客户资料管理

- 提供高级搜索和筛选功能。

车次信息管理

- 实现批量操作，提高工作效率。

综合上述分析，我们根据项目题目的具体要求，精心设计了一套全面且详尽的需求规划。该规划不仅涵盖了基础功能，还考虑了系统的扩展性和用户体验。这些需求被系统地整合和展现在以下图表中（见图 1），其中详细阐述了各项功能的细节和实现机制，确保项目的成功实施。

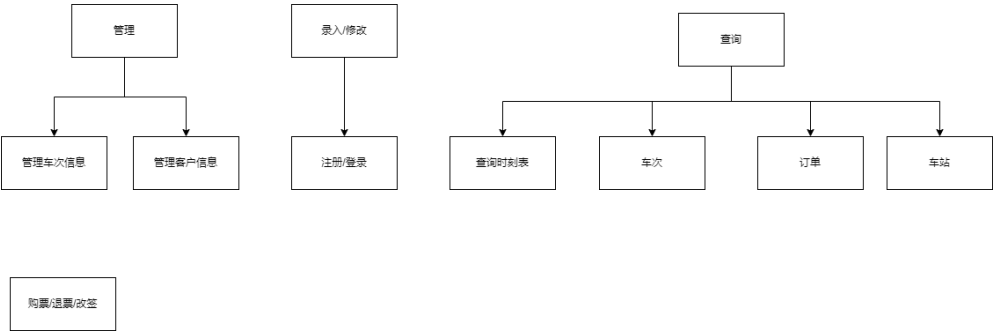


图 1

2 系统设计

2.1 数据结构及文件设计

数据结构的选择及设计反映了系统的核心功能和操作的需要。以下是对各个

数据结构的设计和分析：

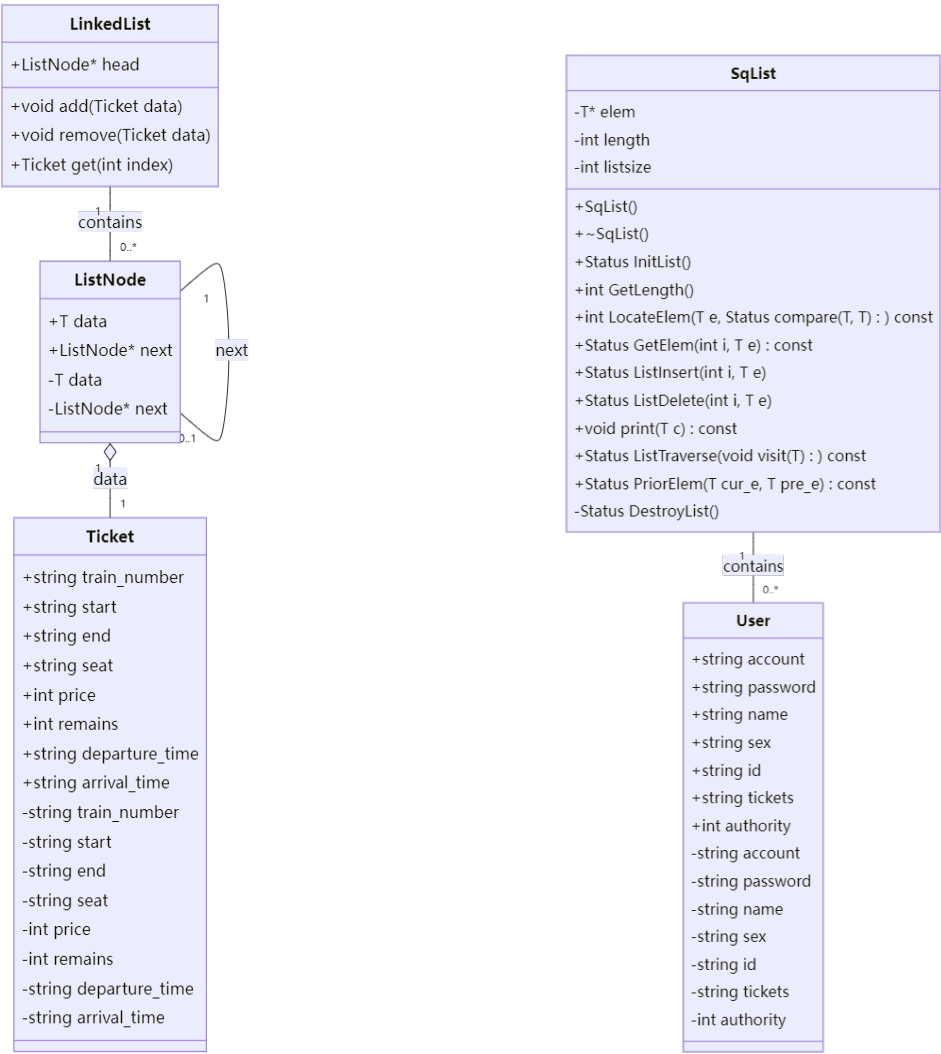
我们采用顺序表来存储用户信息，即 `SqList<User> UserList`;顺序表（`SqList`）提供快速的随机访问，便于查找和更新用户信息。适合于用户数量不是非常大的情况下。

同样使用顺序表 `SqList<Train> TrainList`;存储列车基本信息。类似于用户列表，顺序表在处理固定大小的数据元素时效率较高，适合于快速访问列车数据。

顺序表 `SqList<TrainNumber> TrainNumberList`;用于管理具体的车次信息。方便进行车次的增加、删除和访问。适用于频繁的车次信息更新。

使用非线性数据结构图 `Graph CityGraph`;管理城市间的交通网络。图结构非常适合表示复杂的城市网络，便于实现最优路径查询和城市间连接的可视化。

使用单链表 `LinkedList<Ticket> TicketList`;管理用户的车票信息。链表结构便于动态地添加和删除票务信息，灵活处理车票销售和退票等操作。



如上图所示，我们展示了比较具有代表性的链表和顺序表数据结构图。值得一提的是，我们采用了 C++ 中的泛型模板类来解决数据结构问题，泛型模板类是 C++ 语言的一个高级特性，允许程序员编写与数据类型无关的代码。

例如 `LinkedList<T>` 类，定义的 `LinkedList` 类是一个模板类，表示可以用于任何数据类型 `T` 的链表。这种灵活性允许使用同一链表代码来存储不同类型的数据，例如整数、字符串或者自定义类型，如 `User` 或 `Ticket`。`ListNode` 是链表中的节点，包含数据 `data` 和指向下一个节点的指针 `next`。

构造函数允许创建新节点并初始化其数据和下一个节点指针。

使用这样的方式我们获得了类型独立性：模板类使得 `LinkedList` 可以用于任何类型 `T`，提高了代码的复用性；灵活性：链表结构允许动态地增加和删除元素，无需预先分配固定大小的内存空间；封装性：将链表的实现细节隐藏在类内部，提供清晰的接口给外部使用。

同时，我们使用结构体定义实体数据类型，并采用与泛型类结合的方式方便快捷的创建数据结构，使得系统的延展性提高。

2.2 功能模块的设计

在铁路票务管理系统（`TrainTicketing`）中，我们采用了模块化的设计方法。这种方法不仅有助于代码的组织和维护，而且也使得各个部分的测试和优化变得更加容易。以下是详细模块设计：

2.2.1 `DataEntry.cpp/h` 模块

- 功能：此模块专注于数据的录入和修改。它处理包括车次、用户信息、城市详情等在内的多种数据类型的录入。此外，它还允许修改现有数据，以维持信息的最新状态。
- 重要性：作为系统中数据处理的核心，此模块对于维护数据的完整性和准确性至关重要。它是系统其他所有数据依赖功能的基础，确保了整个系统的数据一致性和可靠性。

2.2.2 `LinkList.h/tpp` 模块

- 功能：该模块实现了泛型链表，为各种类型的数据提供了灵活的存储和管理方式。这在处理动态数据集，如用户票务信息、车次列表时尤为重要。
- 重要性：通过提供动态的数据结构，这个模块增强了系统的灵活性和扩展性。它使得数据的添加和删除操作更加高效，对于需要频繁更新数据的系统尤为关键。

2.2.3 LinkQueue.cpp/h 模块

- 功能: 这个模块提供了链式队列的实现, 专门用于处理需要先进先出逻辑的场景, 如候补订单管理。
- 重要性: 在提高用户体验和系统实用性方面至关重要。例如, 候补机制能够优化票务分配, 提高资源利用率。

2.2.4 Login.cpp/h 模块

- 功能: 负责管理用户登录过程, 包括普通用户和管理员。它处理身份验证、权限分配等关键任务。
- 重要性: 此模块对于系统的安全性至关重要。它不仅保护系统免受未经授权访问, 还为不同类型的用户提供了定制化的服务体验。

2.2.5 Search.cpp/h 模块

- 功能: 负责所有搜索功能, 包括时刻表、票价和车站查询。
- 重要性: 快速准确的搜索功能, 提高用户满意度和系统效率。

2.2.6 Ticket.cpp/h 模块

- 功能: 票务相关操作, 如购票、退票、改签。
- 重要性: 系统的核心功能之一, 直接关系到用户体验。

2.2.7 Tree.h/tpp 模块

- 功能: 实现树形结构, 用于存储和检索复杂的数据关系, 如车次信息。
- 重要性: 提供高效的数据组织方式, 支持复杂查询和操作。

2.2.8 数据文件夹 (data/)

- 内容: 包含 City.csv, Ticket.csv, Train.csv, TrainNumber.csv, User.csv 等数据文件。
- 重要性: 存储系统运行所需的所有静态数据, 是系统运行的基础。

我们的模块设计如(图 2)所示, **DataEntry** 模块是系统数据处理的核心。它负责所有关于数据录入和修改的操作, 涉及车次、用户和城市等信息。这个模块是系统数据管理的基础, 确保了数据的准确性和及时更新。**LinkList** 和 **LinkQueue** 模块提供了灵活的数据结构支持。**LinkList** 实现了一个泛型链表, 用于灵活地管理各种类型的数据, 如用户的票务列表, 而 **LinkQueue** 则实现了链式队列, 主要用于处理需要先进先出逻辑的场景, 例如候补订单的管理。

Login 模块处理用户登录逻辑, 这包括对普通用户和管理员的不同处理。这个模块是系统安全的关键, 确保了系统的安全性。**Search** 模块承担着系统内所有搜索功能的实现, 包括对时刻表、票价和车站的查询。这个模块的设计注重于

搜索的准确性和速度，直接影响到用户对系统的总体满意度。

在票务处理方面，Ticket 模块是非常关键的。它涵盖了购票、退票和改签等核心功能，直接关系到用户的直接体验。系统的用户友好性和效率在很大程度上依赖于这个模块的表现。而 Tree 模块则是我们对高效数据处理的另一个体现，它通过实现树形结构，有效地存储和检索复杂的数据关系，如车次信息，从而支持系统中的复杂查询和高效操作。

在文件层面，系统通过一个专门的数据文件夹来组织和管理所有必要的数据文件，如城市信息、用户数据、车次信息等。这些数据文件的存在是系统运行的基础，确保了数据的持久化和稳定性。此外，系统还包括文档和资源文件夹，其中包含系统架构图和可执行文件等，不仅为系统提供了文档支持，还提供了直接的运行入口，增强了系统的可访问性和易用性。

模块的设计尤为重要，需要满足特定的功能需求，并共同工作，形成一个协调一致、高效运行的整体。模块化的设计方法不仅有助于代码的组织和维护，而且也使得各个部分的测试和优化变得更加容易，从而确保了整个系统的高性能和可靠性。

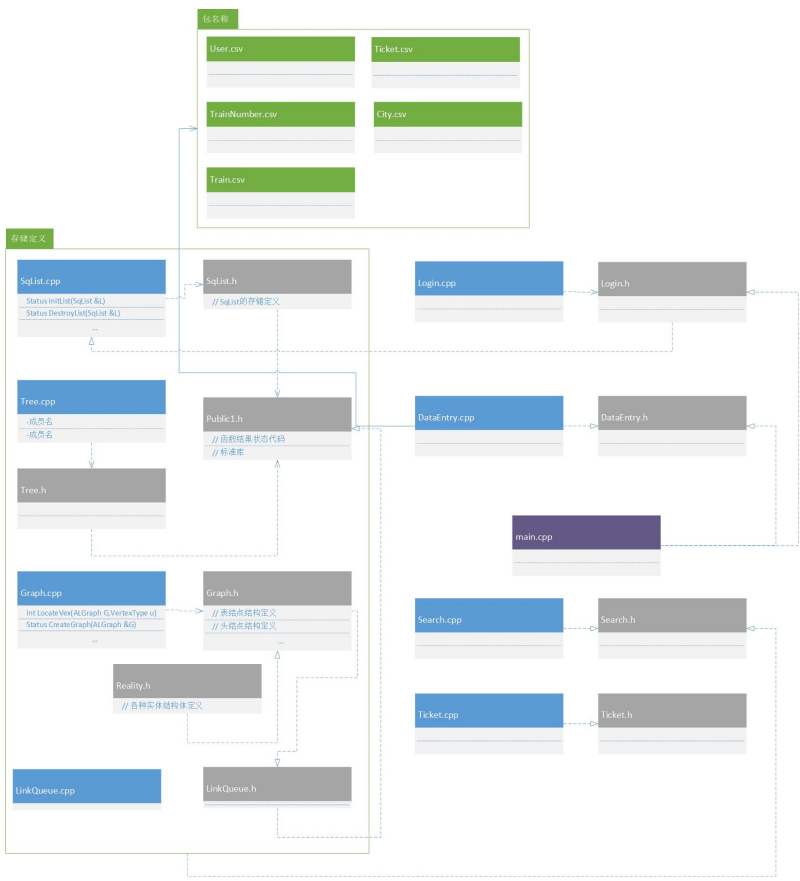


图 2

本人负责相关票务模块的设计和实现，包括票的结构体定义、票的数据结构定义、票的算法等，业务逻辑如（图 3）所示，相关票的算法主要包括购票、退票和改签等操作，我根据业务逻辑留下函数接口以供主函数调用。

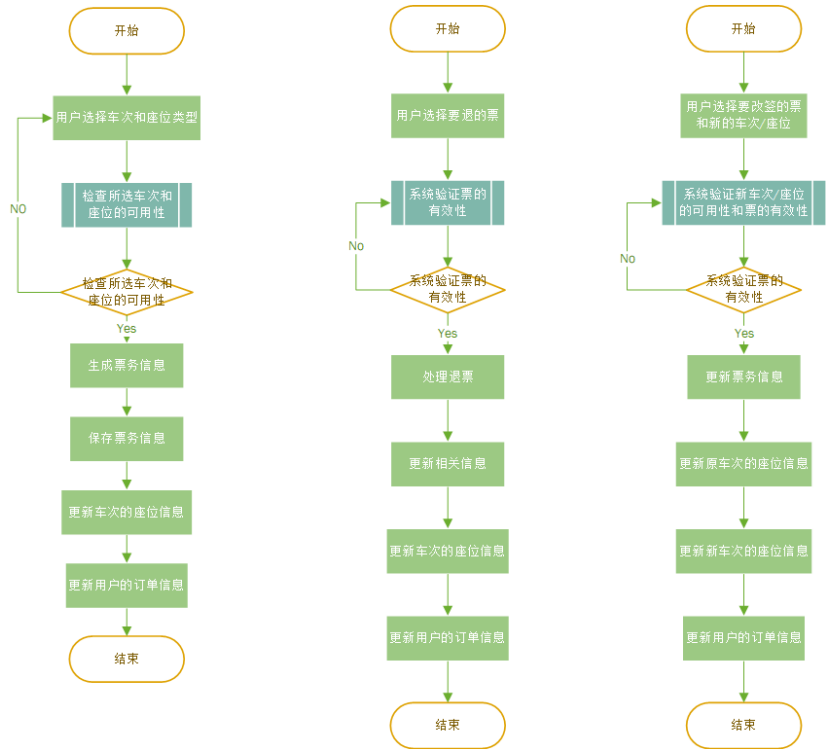


图 3

函数接口如下：

```
Status PurchaseTicket(User &user, Ticket &ticket);
Status EnqueueWaitingList(User &user, Ticket &ticket);
Status RefundTicket(User &user, Ticket &ticket);
```

3 系统调试与测试

3.1 调试过程遇到问题及解决方案

如（图 4）的 Git Graph 展示了我们的开发路径和调试过程。我们遇到的问题颇多，也一直在尝试解决。

宏观问题：

- **初始连接问题：**项目初期遇到头文件和源文件连接问题，出现了重复定义和未定义等问题。解决方案是建立头文件和源代码的连接规范，并启用头文件保护，以避免这些问题。
- **数据结构冗余：**多种实体需要共享同一种数据结构，导致需要维护多组类

似的数据结构，从而产生了函数和参数的冗余。为此，团队采用了泛型类，使数据结构管理更加高效和简洁。

- **代码规范不一**：团队成员之间在代码规范上存在较大差异。解决方案是统一代码规范，确保代码的一致性，提高代码质量和团队协作效率。

微观问题（这里仅描述有代表性的问题，截至 2023/12/29）：

Author: 金太宇 <222090218@stu.nau.edu.cn>

Date: Sat Dec 23 2023 11:21:39 GMT+0800 (Singapore Standard Time)

bugfix(主函数):修复了异常读取的问题

问题：

数据格式化问题，`fscanf` 中的格式字符串 `"%99[^\,],%99[^\,],%99[^\,],%99[^\,],%99[^\,],%d\n"` 假定了数据行以特定格式出现，但如果文件中的数据格式与此不匹配，可能会导致错误的数据读取。需要确保格式字符串与实际数据格式相匹配。

解决方案：

使用 `std::getline` 函数来逐行读取文件中的数据，并使用 `std::istringstream` 来将每行数据解析为不同的字段。这种方法更灵活，不再依赖于固定的数据格式字符串，能够处理不同格式的数据。

Author: 金太宇 <222090218@stu.nau.edu.cn>

Date: Fri Dec 29 2023 10:37:32 GMT+0800 (Singapore Standard Time)

bugfix(主函数): 修复了买的车票无法写入的问题

问题：

在主函数中，存在一个问题，即购票操作后无法成功写入数据。

解决方案：

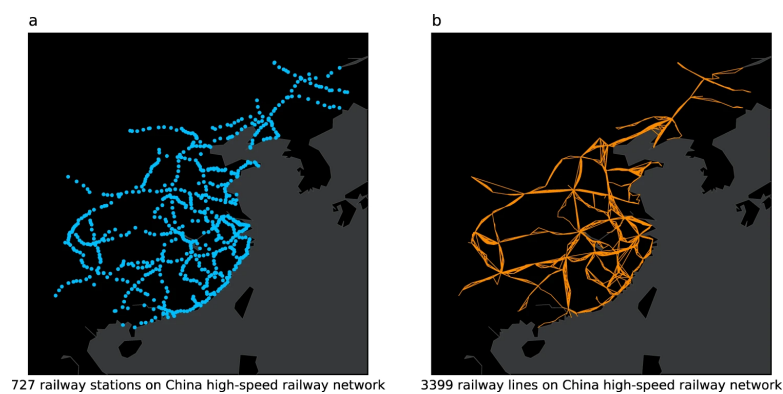
原问题的根本原因在于程序使用了 ``exit()'` 函数，导致程序直接退出，忽略了数据写入的过程。为了解决这个问题，我将原来的 ``exit()'` 调用替换为 ``return'` 语句，以便程序继续执行，而不会立即退出。这样，购票操作完成后，程序会继续执行后续的数据写入和其他必要的操作，确保数据被正确写入。



图 4

3.2 测试功能展示

关于测试数据，我们使用了[1]中的数据集来构建我们的站点图数据结构。该数据集可在 figshare 网站找到，包括了截至 2020 年 1 月 27 日的 3399 条列车运营线路上相邻站点之间的里程信息。此数据集也包含了车次数据，但由于其体量过大，我们选择了使用随机生成的方法模拟数据，从而产生了 13631 条模拟车票信息。



最终形成了如（图 5）如所示的车票格式，并且数据量可观。并且建立了较完善的城际高铁交通网络图，可通过查询最短路径展示如（图 6）。

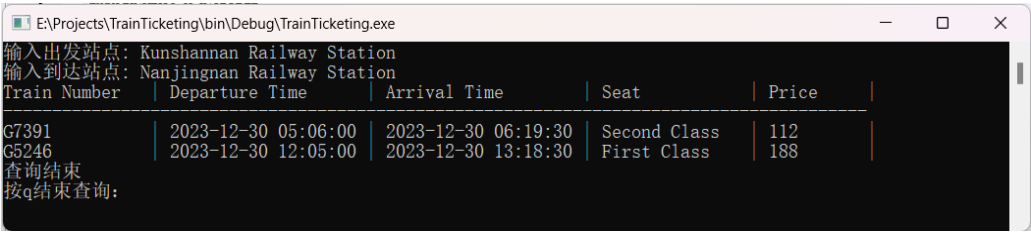
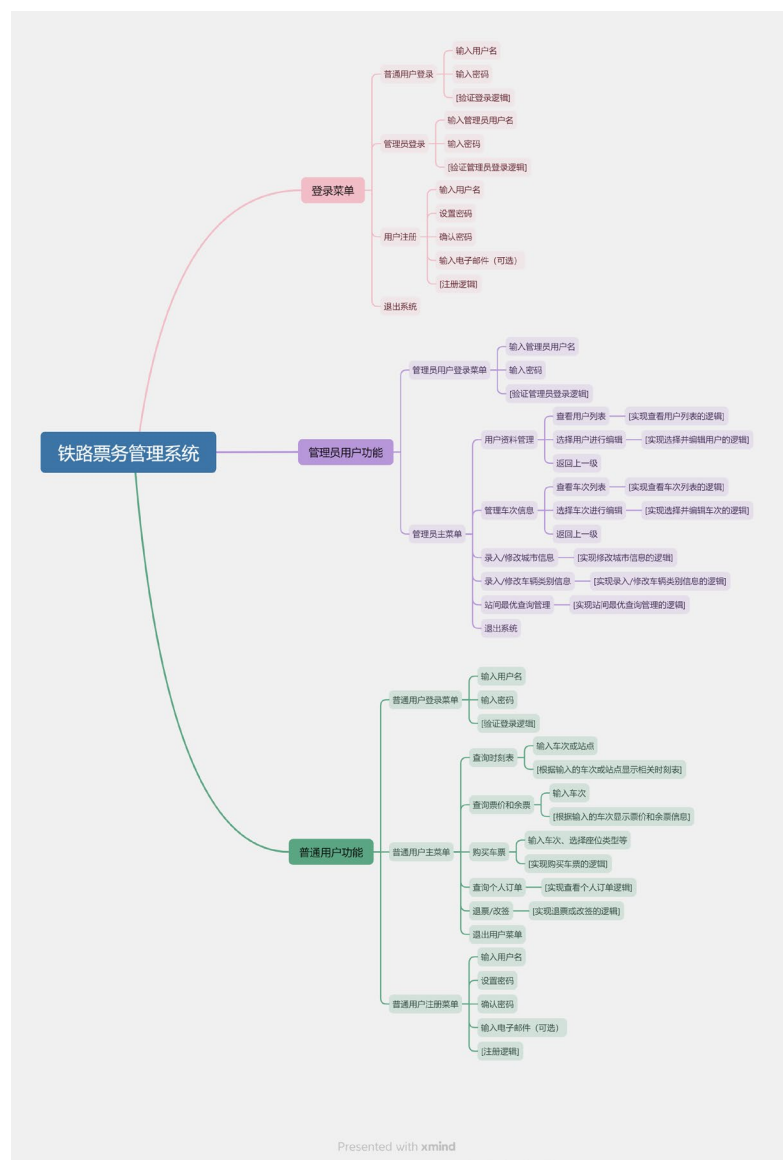


图 5



图 6

此外，设计主菜单以及用户的交互逻辑如下图：



4 分析优化与改进

4.1 算法性能分析

性能良好的算法能够快速处理大量数据，提供及时的响应。这对于用户体验至关重要，特别是在需要实时或近实时反馈的应用中，如在线交易系统、实时通信等，并且可以更好地利用系统资源，如处理器时间、内存和存储空间，有助于降低运行成本系统可以处理更大的数据集或更多的并发用户，而不会显著降低性能。这对于快速增长的应用或数据量大的系统尤其重要。我们也一直在尝试优化系统的算法性能。

4.1.1 PurchaseTicket 函数

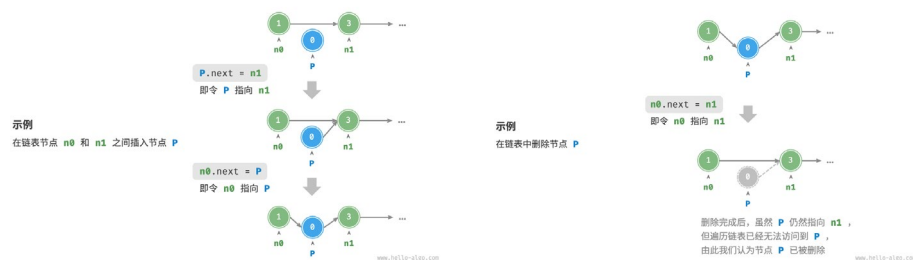
时间复杂度:

链表查找操作: `TicketList.GetElemPtr(i)` 和 `TicketList.GetElem(i)` 在循环中被调用, 这表明查找操作是线性的, 即 $O(n)$, 其中 n 是链表 `TicketList` 的长度。这是因为要找到特定的车票, 函数需要遍历整个链表。

链表删除和插入操作: 如下图, `TicketList.ListDelete(i, tempTicketForDel)` 和 `TicketList.ListInsert(i, tempTicket)` 都在循环之后执行一次。这些操作的复杂度通常也是 $O(n)$, 因为可能涉及到节点的移动。

更新用户票记录: 这部分操作的复杂度取决于 `user.tickets` 字符串的长度。
+= 操作是在字符串末尾追加, 其复杂度一般是 $O(1)$ 。

综上所述, 整个 `PurchaseTicket` 函数的时间复杂度大致是 $O(n)$, 主要取决于链表的长度。



空间复杂度:

局部变量: 函数使用了几个局部变量 (如 `tempTicket`, `tempTicketForDel` 等), 但这些不随链表大小变化, 所以对空间复杂度的贡献是 $O(1)$ 。

链表操作: 由于链表操作不需要额外的空间 (除了上述局部变量), 所以这部分的空间复杂度也是 $O(1)$ 。

用户票记录更新: 更新 `user.tickets` 字符串不会显著增加空间复杂度, 除非字符串在追加过程中需要扩展。这通常是由底层实现管理的, 不应该对算法的空间复杂度产生重大影响。

因此, 如果不算上链表本身的空间复杂度, `PurchaseTicket` 函数的空间复杂度是 $O(1)$, 即常量级别。

4.1.2 RefundTicket 函数

时间复杂度:

链表搜索操作：该函数首先通过一个循环遍历 TicketList 链表来查找匹配的 trainNumber。这个操作的时间复杂度是 $O(n)$ ，其中 n 是链表的长度。由于需要逐个检查链表中的每个元素，所以这是一个线性搜索。

链表的删除和插入操作：在找到对应的 trainNumber 后，函数执行一次删除 (ListDelete) 和插入 (ListInsert) 操作。这些操作通常也具有 $O(n)$ 的时间复杂度，尽管实际的复杂度取决于链表的具体实现。

字符串搜索和修改：接着函数在 user.tickets 字符串中搜索 trainNumber。这个操作的时间复杂度取决于字符串的长度，若采用 KMP 算法，如（图 7），则通常为线性阶 $O(m)$ ，其中 m 是字符串的长度。接着进行字符串的删除操作，这通常也是 $O(m)$ 。

总体来说，RefundTicket 函数的时间复杂度大体上是 $O(n + m)$ ，其中 n 是链表的长度， m 是用户票记录字符串的长度。

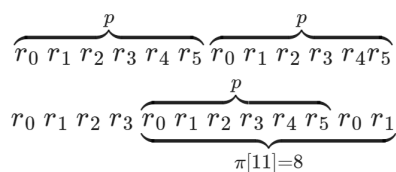


图 7

空间复杂度：

局部变量：函数使用了几个局部变量（如 tempTicket, tempTicketForDel 等），这些占用的空间是固定的，因此贡献的空间复杂度为 $O(1)$ 。

链表操作：链表的删除和插入操作通常不需要额外的空间，所以这部分的空间复杂度也是 $O(1)$ 。

字符串操作：字符串的搜索和修改操作不会增加额外的空间复杂度，除非涉及到字符串扩展或缩减。然而，这些操作通常由底层字符串管理机制处理，不会对算法整体的空间复杂度产生显著影响。

因此，RefundTicket 函数的空间复杂度是 $O(1)$ ，即常量级别。

4.1.3 Timetable 函数

时间复杂度：

链表遍历：函数遍历整个 TicketList 链表以找到匹配的车票。这个遍历过程的时间复杂度是 $O(n)$ ，其中 n 是链表的长度。

快速排序：如（图 8）如果找到匹配的车票，函数使用快速排序算法对它们进行排序。快速排序的平均时间复杂度是 $O(m \log m)$ ，其中 m 是匹配票务的数量。在最坏的情况下，这个复杂度可以达到 $O(m^2)$ 。

打印输出：最后，函数遍历 `matchingTickets` 向量并打印每张票的信息。这个过程的时间复杂度是 $O(m)$ 。

综上，总体时间复杂度是 $O(n + m \log m + m)$ 。在最坏情况下，如果所有票都匹配，那么 $n = m$ ，时间复杂度可以近似为 $O(n \log n)$ 。

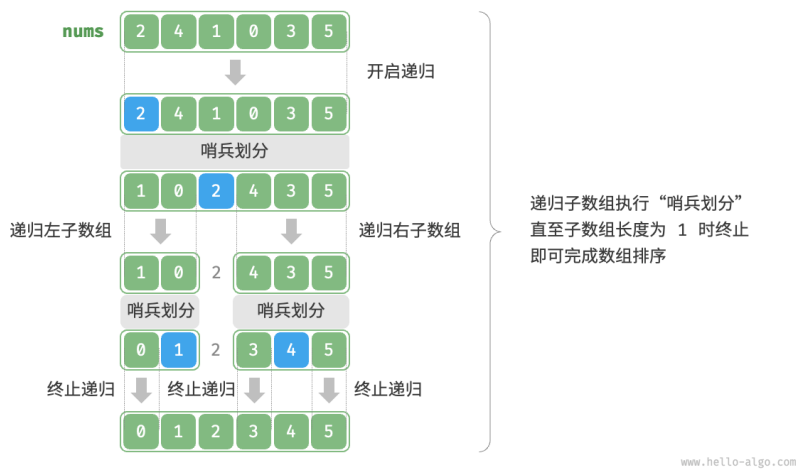


图 8

空间复杂度：

匹配票务向量：`matchingTickets` 向量的空间复杂度是 $O(m)$ ，其中 m 是匹配的票务数量。这是因为它需要存储所有匹配的票务。

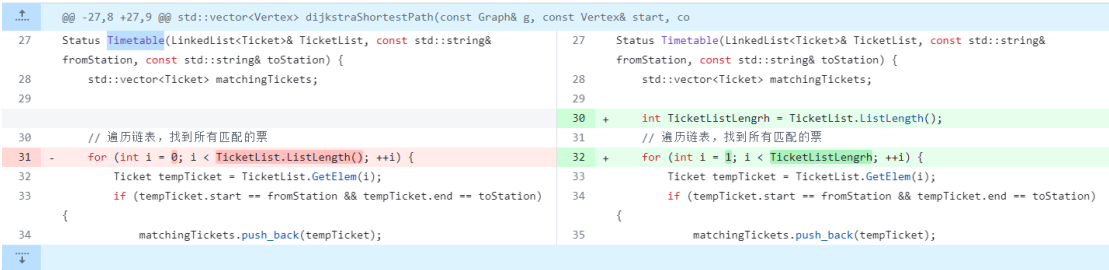
局部变量和快速排序：快速排序在递归过程中会占用额外的栈空间。在最坏情况下，这个空间复杂度可以达到 $O(m)$ 。但在平均情况下，它是 $O(\log m)$ 。

其他局部变量：函数中使用的其他局部变量（如 `tempTicket`）占用的空间是固定的，因此贡献的空间复杂度为 $O(1)$ 。

因此，整个函数的空间复杂度大致为 $O(m)$ ，主要由匹配的票务向量决定。

4.2 优化与改进

4.2.1 Timetable 函数



```
@@ -27,8 +27,9 @@ std::vector<Vertex> dijkstraShortestPath(const Graph& g, const Vertex& start, co
27 Status Timetable(LinkedList<Ticket>& TicketList, const std::string&
   fromStation, const std::string& toStation) {
28     std::vector<Ticket> matchingTickets;
29
30     // 遍历链表，找到所有匹配的票
31 -   for (int i = 0; i < TicketList.ListLength(); ++i) {
32         Ticket tempTicket = TicketList.GetElem(i);
33         if (tempTicket.start == fromStation && tempTicket.end == toStation)
34         {
35             matchingTickets.push_back(tempTicket);
36     }
37 }
38
39 Status Timetable(LinkedList<Ticket>& TicketList, const std::string&
   fromStation, const std::string& toStation) {
28     std::vector<Ticket> matchingTickets;
29
30 +   int TicketListLength = TicketList.ListLength();
31     // 遍历链表，找到所有匹配的票
32 +   for (int i = 0; i < TicketListLength; ++i) {
33         Ticket tempTicket = TicketList.GetElem(i);
34         if (tempTicket.start == fromStation && tempTicket.end == toStation)
35         {
36             matchingTickets.push_back(tempTicket);
37     }
38 }
```

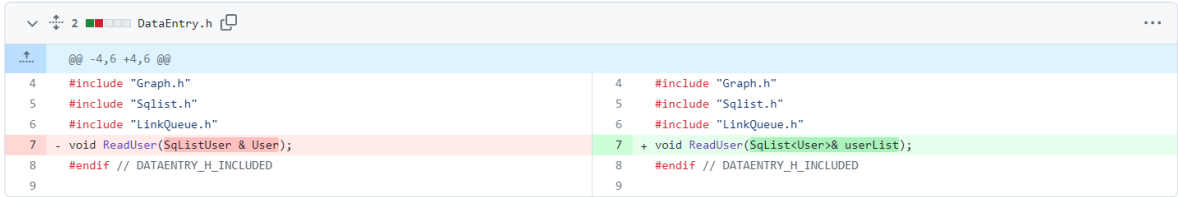
图 9

如（图 9），在优化前，代码在循环中多次调用 `TicketList.ListLength()`。这种做法可能会引起性能问题，特别是如果 `ListLength()` 函数每次调用都要遍历整个链表来计算长度的话。在链表很长的情况下，这会导致每次迭代都带来不必要的性能开销。

在优化后，通过在循环外部调用一次 `ListLength()` 并将结果存储在 `TicketListLength` 变量中解决了这个问题。这意味着无论链表有多长，长度计算只会执行一次，从而减少了循环中的计算量。

当然，仍然有优化空间，我们可以优化链表操作，如果可能的话，使用更高效的数据结构来管理票务信息，例如哈希表或平衡二叉树，这些结构可以提供更快的查找和更新操作。

4.2.2 SqlList 泛型模板



```
@@ -4,6 +4,6 @@
4 #include "Graph.h"
5 #include "SqlList.h"
6 #include "LinkQueue.h"
7 - void ReadUser(SqlListUser & user);
8 #endif // DATAENTRY_H_INCLUDED
9
4 #include "Graph.h"
5 #include "SqlList.h"
6 #include "LinkQueue.h"
7 + void ReadUser(SqlList<User>& userList);
8 #endif // DATAENTRY_H_INCLUDED
9
```

图 10

对 `ReadUser` 函数的参数进行了修改。原来的版本是接受一个 `SqlListUser` 对象的引用，而在新版本中，更改参数类型为 `SqlList<User>&` 的引用。使用了模板编程来使 `ReadUser` 函数更加通用。通过这种方式，`SqlList` 类现在可以处理任何类型的 `User` 对象，而不仅仅是特定类型。这意味着 `SqlList` 已经变成了一个模板类，允许用户定义存储在 `SqlList` 中的数据类型。

泛型模板类允许代码重用，提高了代码的可维护性和可扩展性。现在，相同的 `SqList` 数据结构可以用于不同的数据类型，而不需要为每种类型重写代码。

模板通常是在编译时实例化的，这意味着没有运行时的类型检查或转换开销，这可能会提高性能。使用模板可以使得代码更加清晰和易于理解，尤其是在处理多种数据类型的时候。

优化代码更加泛化和模块化，为未来可能的变化和扩展打下基础。这是现代 C++ 编程中一个常见的实践，它促进了更高级别的抽象和代码重用。

4.2.3 未来优化目标

优化数据结构：如果 `TicketList` 的使用频繁且其大小很大，考虑使用更高效的数据结构，如哈希表或平衡二叉树，以提高搜索效率。

避免不必要的复制：在遍历链表时，可以考虑使用指针或引用来避免不必要的票务复制。

考虑排序的提升能力：可以采用更加先进的排序方式，比如组合排序，结合插入排序和快速排序等，确保使用最适合当前数据集的排序算法。

5 总结

通过这次课程设计，我深刻体会到了数据结构在实际应用中的重要性。在开发铁路票务管理系统的过程中，我应用了学到的理论知识，提高了我的编程技能和问题解决能力。

这个项目使我对数据结构的应用有了更深的理解。在设计系统时，我意识到每种数据结构都有其独特的优势，不仅要考虑系统的功能需求，还要设计适当的数据结构来存储和管理数据。例如，我使用顺序表来存储用户信息，以便于快速访问和更新；而链表则用于管理用户的车票信息，因其在动态数据处理中的高效性。这些选择不仅提高了系统的效率，也优化了存储空间的使用。

我学会了如何在实际项目中解决问题。在开发过程中，我遇到了许多挑战，如头文件和源文件的连接问题，数据结构的冗余问题等。这些问题的解决过程教会了我如何有效地使用工具和技术，比如使用泛型类和模板来优化代码，提高复用性和灵活性。这些都是现代软件工程实践中不可或缺的元素。

此外，我也意识到了模块化设计的重要性。通过将系统划分为不同的模块，如登录、票务处理、查询等，我能更有效地管理和维护代码。这种方法不仅有助于代码的组织和调试，还增强了系统的可扩展性。通过使用实际数据集来测试系统，进一步提高了项目的实用性和可靠性。

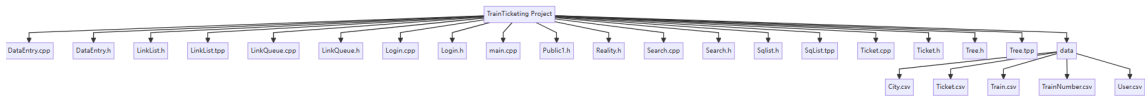
我对团队合作有了更深的认识。在这个项目中，我与我的同学们一起工作，共同解决问题。这个过程教会了我沟通和协作的重要性，以及如何在团队环境中有效地贡献我的技能。

这次课程设计不仅加深了我对数据结构的理解，也提升了我的编程技能和团队协作能力。我相信这些经验和技能将对我的未来学习和职业生涯大有裨益。

附录

[1]. Zhang, D., Peng, Y., Xu, Y. et al. A high-speed railway network dataset from train operation records and weather data. Sci Data 9, 244 (2022).

文件目录展示图



文件目录

根目录：

- |
- |—— 源文件 (.cpp, .h, .tpp):
 - |—— DataEntry.cpp - 数据录入功能实现。
 - |—— DataEntry.h - 数据录入功能头文件。
 - |—— LinkedList.h - 链表数据结构头文件。
 - |—— LinkedList.tpp - 链表模板实现。
 - |—— LinkQueue.cpp - 基于链表的队列实现。
 - |—— LinkQueue.h - 链接队列头文件。
 - |—— Login.cpp - 登录功能实现。
 - |—— Login.h - 登录功能头文件。
 - |—— main.cpp - 程序主入口。
 - |—— Public.h - 公共工具函数头文件。
 - |—— README.md - 项目文档的 Markdown 文件。
 - |—— Reality.h - 实体结构体定义。

- | |—— Search.cpp - 搜索功能实现。
- | |—— Search.h - 搜索功能头文件。
- | |—— Sqlist.h - 顺序列表数据结构头文件。
- | |—— SqlList.tpp - 顺序列表模板实现。
- | |—— Ticket.cpp - 票务处理实现。
- | |—— Ticket.h - 票务处理头文件。
- | |—— TrainTicketing.cbproj - Code::Blocks 项目文件。
- | |—— TrainTicketing.depend - Code::Blocks 依赖文件。
- | |—— TrainTicketing.layout - Code::Blocks 布局文件。
- | |—— Tree.h - 树形数据结构头文件。
- | |—— Tree.tpp - 树形数据结构模板实现。
- | |
- | |—— 二进制文件 (bin):
- | |—— Debug:
- | |—— TrainTicketing.exe - 项目可执行二进制文件。
- | |
- | |—— 数据文件 (data):
- | |—— City.csv - 城市数据 CSV 文件。
- | |—— Ticket.csv - 票务数据 CSV 文件。
- | |—— Train.csv - 火车数据 CSV 文件。
- | |—— TrainNumber.csv - 火车号数据 CSV 文件。
- | |—— User.csv - 用户数据 CSV 文件。
- | |
- | |—— Markdown 文件 (md):
- | |—— Modules.png - 展示模块架构的图片。
- | |—— Structs.png - 展示所用数据结构的图片。
- | |—— TicketFlow.png - 展示票务流程的图片。
- | |
- | |—— 对象文件 (obj):
- | |—— Debug:
- | |—— DataEntry.o - DataEntry 的编译对象文件。
- | |—— LinkQueue.o - LinkQueue 的编译对象文件。
- | |—— Login.o - Login 的编译对象文件。
- | |—— main.o - main 的编译对象文件。
- | |—— Search.o - Search 的编译对象文件。
- | |—— Ticket.o - Ticket 的编译对象文件。

<https://github.com/Hydrbonytttrium/TrainTicketing.git>