第一部分 算法实现说明

1.1 题目

给定一个有向图，实现以下操作并实时展示过程：
(1)建立并显示出它的邻接链表；
(2)对该图进行拓扑排序，显示拓扑排序的结果，并随时显示入度域的变化情况；
(3)计算关键路径，显示事件最早发生时间 Ve，事件最晚发生时间 Vl，活动最早开始时 间 E，活动最晚开始时间 L，时间余量 L-E。

1.2 软件功能

1.2.1 图的建立以及邻接链表可视化

功能描述：用户在输入框以指定格式输入有向图内容，软件给出相应的可视化图结构以及邻接链表。
界面操作说明：
1、在顶部的输入框中输入想要构建的图结构，包括起点，终点，对应边权重。
2、点击生成/清空图按钮，即可生成对应的图，右侧基本信息 Tab 显示出了邻接链表
实现方式：
解析文本框中的输入内容，提取起点终点以及边权重，在右侧按照指定格式打印邻接链表。根据点的字典序进行排序，以画板中心为圆心，计算各个点的位置，然后先绘制边，再绘制结点，保证节点不被盖住。

1.2.2 进行拓扑排序

功能描述：用户可以选择自动拓扑排序或者单步拓扑排序。
界面操作说明：
点击拓扑排序（自动）或拓扑排序单步，已排序的边将通过高亮黄色表示，节点将通过高亮绿色表示并更新对应点的入度标签。拓扑排序结果将在底部文本框中显示。若有向图存在环，则提示错误。
实现方式：
根据拓扑排序计算方式，将入度域为 0 的节点放入队列中，并更新相应终点的入度域，重复上述步骤直到节点入度域都为 0,并调用 canvas 类方法更改对应边和点的设置，从而实现可视化。

1.2.3 计算关键路径

功能描述：用户可以选择点击计算关键路径，实现关键路径的可视化标注，以及对应显示事件最早发生时间 Ve，事件最晚发生时间 Vl，活动最早开始时 间 E，活动最晚开始时间 L，时间余量 L-E。
界面操作说明：点击关键路径，关键路径上的边将高亮为红色，关键路径上的节点将高亮为红色。在关键路径的 Tab 中将显示关键路径的分析结果，包括最早发生时间 Ve，事件最晚发生时间 Vl，活动最早开始时 间 E，活动最晚开始时间 L，时间余量 L-E。
实现方式：根据关键路径算法，先进行正向和逆向遍历，计算每个节点的最早开始时间和最晚开始时间，找出所有关键活动（即最早和最晚时间相等的活动），并调用 canvas 类方法更改对应边和点的设置，实现可视化。

1.3 设计思想

本项目遵循 MVC（Model View Controller）模式的设计思路。由于体量较小，因此将 controller

和 View 界面集成在了 gui.py 文件中。Model 集成在 Grap.py 文件中，包括存储结构以及相应的方法。

本次程序设计主要分为三个部分。首先，是算法的核心设计与实现，包括满足题目要求的所有内部算法和数据结构。)，但不包含任何与图形界面相关的内容。图形显示的设计，专门负责处理界面的呈现，而不涉及具体的算法。通过 controller 将二者进行连接。

1.3.1 Model 层

Graph.py 中定义了数据结构以及相应的拓扑排序方法和关键路径计算方式，使用结合 python 中的语法，使用集合存放节点，字典存放边，Key 为起点，value 为终点和代价的集合。

```
self.adjacency_list = defaultdict(list)
self.nodes = set()
```

Graph 类中的 topological_sort 和 critical_path 只负责计算结果，不涉及如何可视化展示。

topological_sort 除了计算拓扑排序的结果，还要记录每一步的操作以便在后续实现单步可视化。

```
steps.append({
    'processing_node': u,
    'in_degrees': in_degree_map.copy()
})
```

1.3.2 View 层

gui.py 中使用 tkinter 构建整个程序的布局格式，主要分为控制部分和画布部分，控制部分包括输入文本，以及相应的按钮。画布部分负责可视化有向图。

MainApplication 类，负责整个程序界面，初始化时调用 def _create_widgets(self):，负责构建起所有程序的可视化框架。提供了可交互的按钮，用于可视化有向图的画布，以及相应的 Tab 用于显示相应结果。

1.3.3 Controller 层

通过控制器将原有的 View 和 Model 层相关连，主要负责对应结果的可视化展示和作为槽函数链接按钮。

```
def _generate_graph(self):
```

负责生成相应的有向图

```
def _display_adjacency_list(self):
```

负责在基础信息 Tab 中给出邻接链表

```
def _run_topo_sort_step(self, auto_run=False):
```

负责拓扑排序的单项演示

```
def _run_topo_sort_auto(self):
```

负责拓扑排序的自动演示

```
def _calculate_critical_path(self):
```

负责将计算好的关键路径结果可视化

1.4 逻辑结构与物理结构

1.4.1 逻辑结构

使用图状关系表示数据之间的关系，图状关系是一种非线性的数据结构，由节点（或顶点）和边组成，能够有效地表达数据元素之间的复杂关联和多对多关系。有向图是一种图状结构，其中边（也称为弧）具有方向性，即边从一个节点（称为起点或尾节点）指向另一个节点（称为终点或头节点）。这种方向性表示了节点间关系的单向性或依赖性

1.4.2 物理结构

核心物理结构邻接表 (Adjacency List)。

self.adjacency_list 是一个 defaultdict(list)，字典的键（Key）是节点 u。这个键在内存中代表一个节点对象。字典的值（Value）是一个 Python 列表（list）。这个列表存储在内存中一块连续的区域。列表中的每一个元素是一个元组 (Tuple) (v, weight)。这个元组存储了从节点 u 出发的一条边的两个属性：

1、目标节点（v）： 表示边的方向。

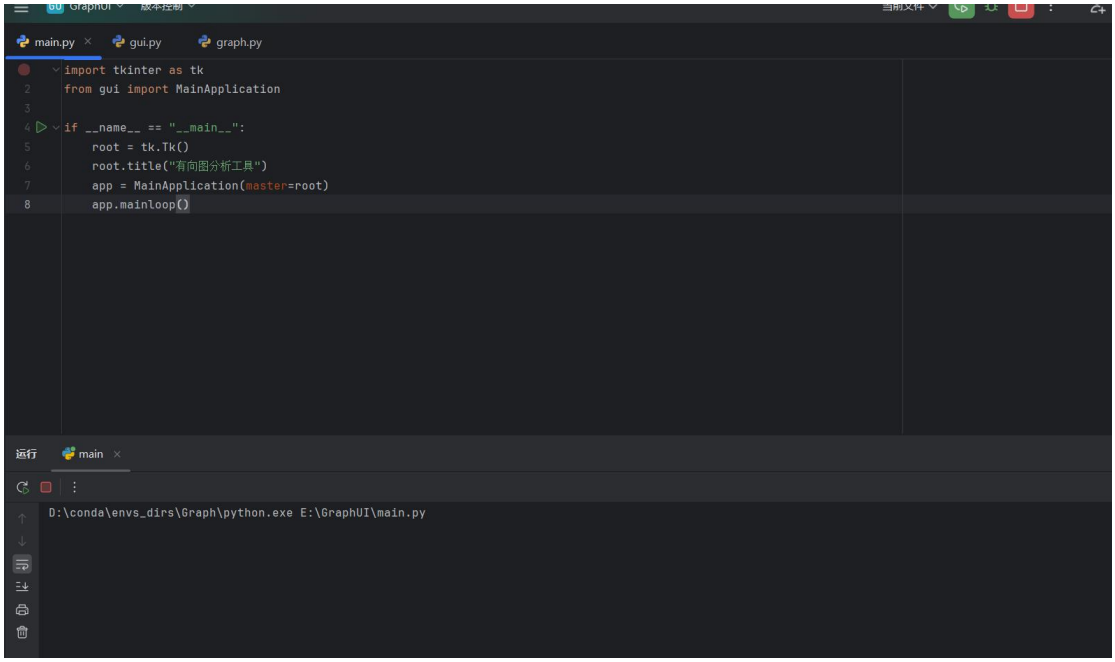
2、权重（weight）： 表示边的属性（如活动耗时）。

1.5 开发平台

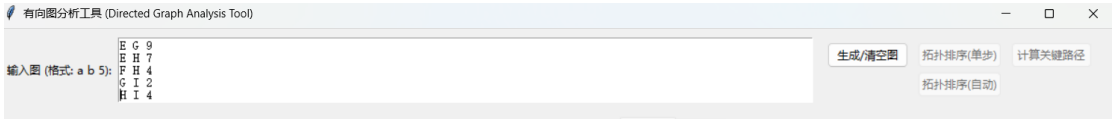
操作系统：Window 11 ，集成开发环境（IDE）：PyCharm ，库和框架：tkinter

1.6 系统的运行结果分析说明

运行 main 函数



为防止用户不知道格式，这里在文本框中给出了运行示例



鲁棒性分析：

输入阶段检查输入格式，没有写路径代价默认为 1

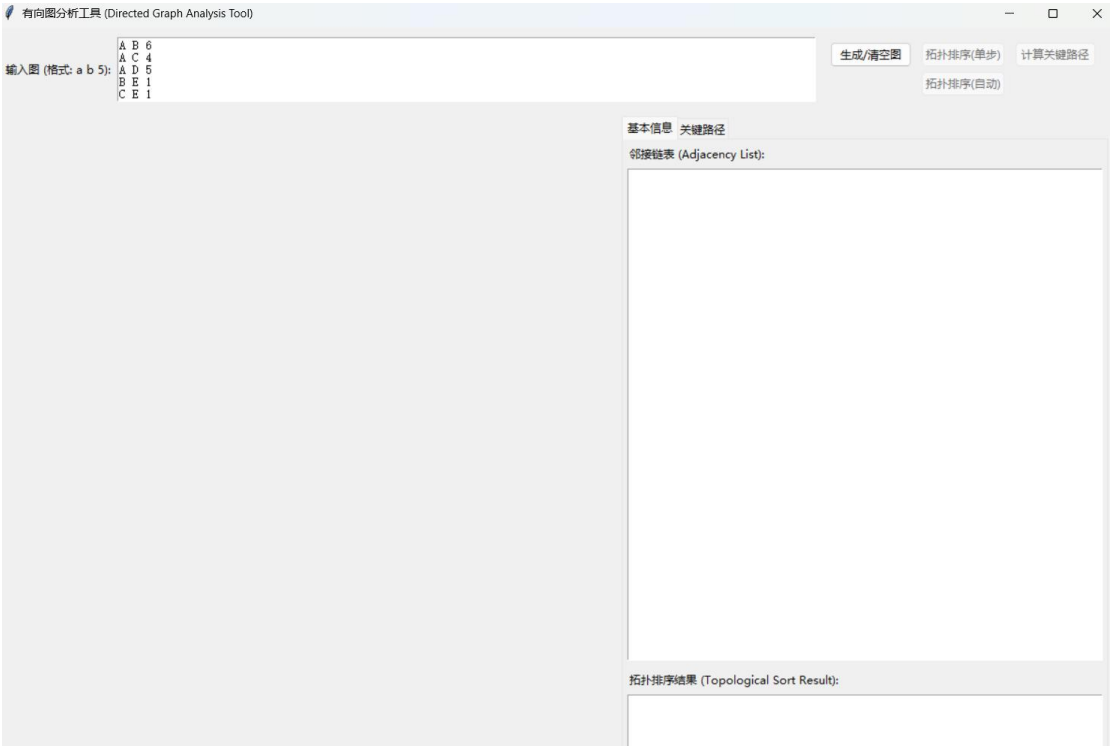
```
try:
    edges = input_data.split('\n')
    for edge in edges:
        parts = edge.strip().split()
        if len(parts) == 3:
            u, v, w = parts[0], parts[1], int(parts[2])
            self.graph.add_edge(u, v, w)
        elif len(parts) == 2:  # Default weight 1 if not provided
            u, v = parts[0], parts[1]
            self.graph.add_edge(u, v, weight: 1)
except (ValueError, IndexError) as e:
    messagebox.showerror( title: "输入错误", message: f"输入格式错误: {e}\n请遵循 '起点 终点 权重' 格式.")
    return
```

拓扑排序时，先检查有向图是否为环，若为环，则提示错误信息。

```
topo_order, self.topo_steps, is_dag = self.graph.topological_sort()
if not is_dag:
    messagebox.showerror( title: "错误", message: "图中存在环，无法进行拓扑排序!")
    self.status_var.set("错误：检测到环")
    self.topo_steps = []
    return
```

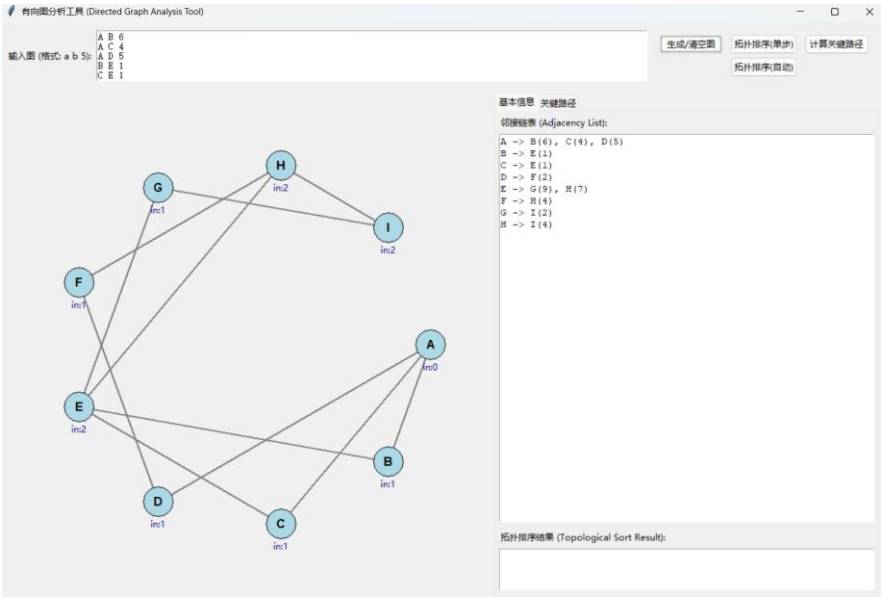
1.7 操作说明

双击 exe 文件打开

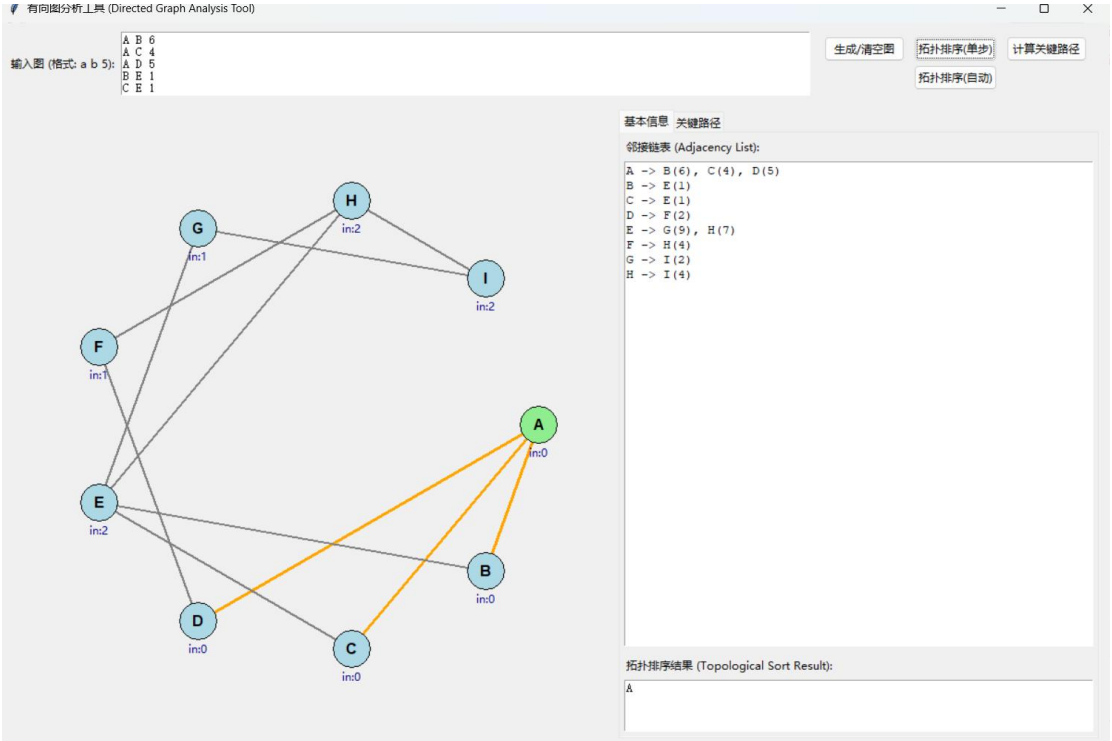


默认情况下已经给出示例图

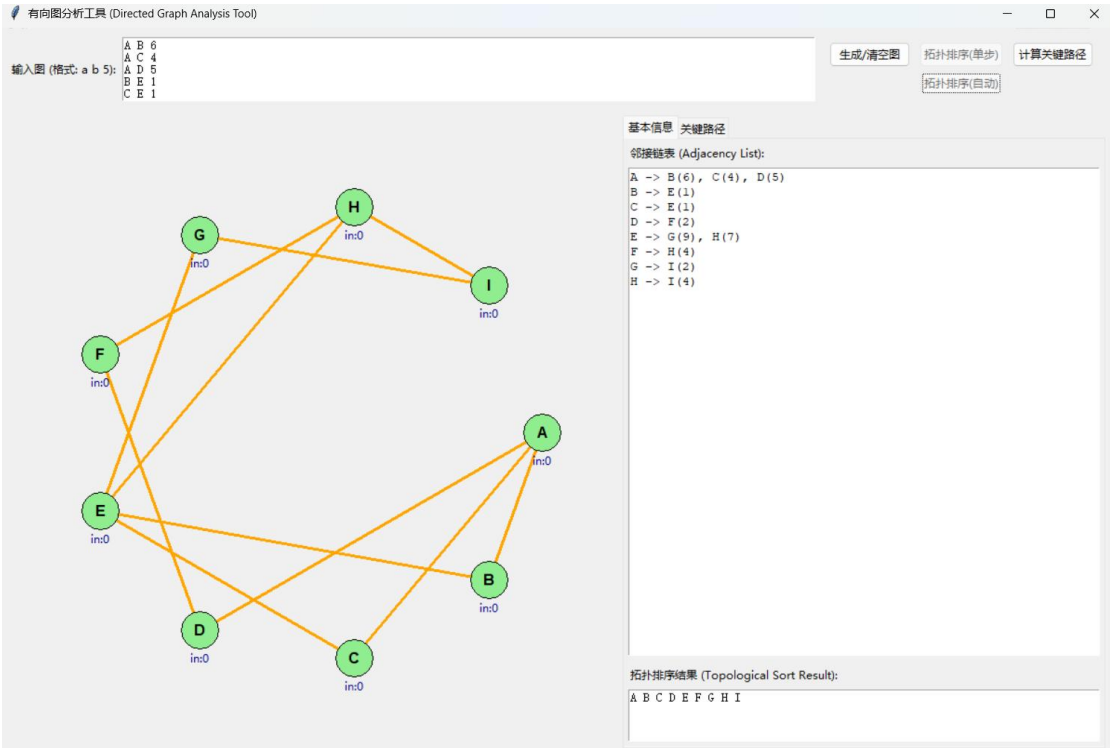
点击生成/清空图，绘制有向图，右侧基本信息中显示了邻接链表



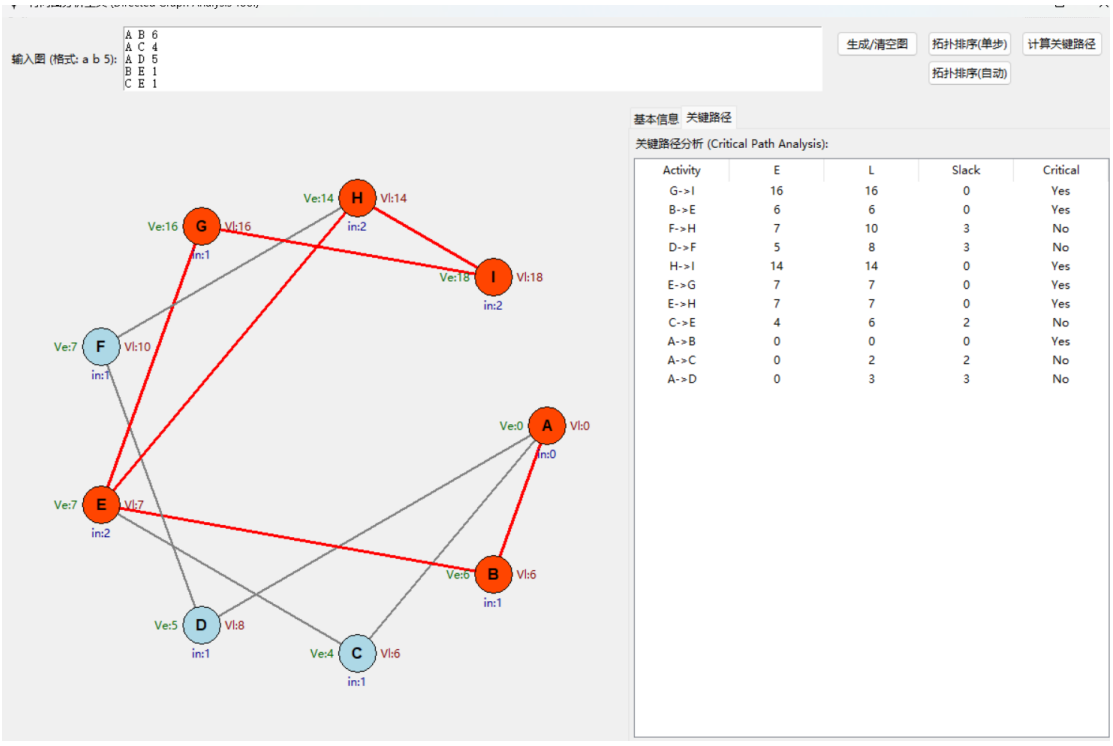
点击拓扑排序（单步）实现单步示意，下侧拓扑排序结果中显示当前结果



点击拓扑排序（自动）实现自动示意



点击计算关键路径，有向图中进行可视化显示，包括高亮和标签，右侧的 Tab 中有相应计算表格



第二部分 综合应用设计说明

2.1 题目

某高校，教职工 9000 多人，其组织机构较为复杂，顶层分为学院、党群组织、行政 机构、直属单位、附属单位等（可参考同济大学主页中介绍）。每一组织部门又分为多 个层次，学校各教职工依据职位和角色可以隶属于多个组织部门。现需设计一个组织机构管理系统，系统需管理各级部门及各部门教职工，各部门需设立主管职位一名、主管 副职多名、其他人员多名。

（1）动态建立组织结构，即可插入、删除部门等。

（2）在部门中添加各类型人员，可定义职位。注意一个人可以在多个部门工作，因此人 员需用线性表单独存储。

（3）根据输入的人员名，查找其所在的部门、职位等信息。

2.2 软件功能

2.2.1 组织机构架构可视化

功能描述：

通过目录层次结构显示高校组织架构，点击相应的父节点可以查看该父节点下的子节点

界面操作说明：

每一层组织分为组织唯一 id 和组织结构名，右侧以主管，副职、其他人员的层次结构显示当前机构的人员名单

实现方式：

通过 ttk.Treeview(tree_frame, columns=("name",), show="tree headings")建立树组织结构，对于每一树节点，都是一个 Department 类，该类包含了下属部门以及该部门的所有员工。

2.2.2 添加部门

功能描述：

在指定父部门下添加子部门

界面操作说明：

在树结构中选定节点，点击添加部门按钮，即可在指定父节点下添加子节点

实现方式：

选择指定的 Department 类实例，然后调用 def add_child(self, child)方法，在该部门下添加子节点，刷新树结构，显示新结果

2.2.3 添加新人员

功能描述：

添加新的人员信息

界面操作说明：

点击添加人员按钮，在对话框中输入人员的基本信息，点击 OK 即可添加人员

实现方式：

创建 Person 类的新实例，保存在线性表中

2.2.4 分配职位

功能描述：

对人员分配部门，职位

界面操作说明：

在树结构中选定部门，点击分配职位，在弹出的对话框中选择人员，选择职位类别，输入 u
职位名称
实现方式：
获取当前选择的部门节点，选定 Person 类实例，调用 add_assigment(self, department_object, position_title)方法，添加所属部门和职位名称

2.2.4 查找人员

功能描述：
通过姓名查找人员名单，以及对应的所有详细信息
界面操作说明：
点击查找人员，默认情况下显示线性表中所有的人员名单，双击相应的条目，即可查看详细信息。除员工 ID 外，所有信息均可更新。选中的职务，点击卸任该职位，即可卸任职位。
实现方式：
通过姓名在线性表中进行查找，找到后双击跳转相应信息界面，点击信息进行更新。
卸任职位通过删除 person 类中的属性，以及删除对应 department 类中的属性。

2.2.5 删除部门

功能描述：
删除树结构中的某一节点
界面操作说明：
选定节点，点击删除部门按钮，递归删除该节点下的所有部门，包括该节点
实现方式：
选定节点，定义删除 def delete_department(self, department_id: str) -> bool 方法，使用递归的方法，删除下面所有后代节点，释放内存

2.2.6 删除人员

功能描述：
删除人员
界面操作说明：
点击删除人员按钮，输入删除人员的 ID，即可删除
实现方式：
通过 ID 查找人员，若该人员存在，则释放实例，从线性表中删除，并且遍历所有 Department 类实例，更新对应信息。

2.2.7 保存/加载数据

功能描述：
将数据保存为 json 格式，或者读取 json 文件，初始化数据
界面操作说明：
点击保存/加载数据按钮，从文件夹中选择相应文件
实现方式：
调用数据模型管理器 OrgModel 类中的
def save_to_file(self, filepath: str)方法，将线性表结构和树结构进行保存。
调用 def load_from_file(self, filepath: str)方法，将 json 文件中的数据进行重建

2.3 设计思想

该程序遵循 MVC（Model-View-Controller）设计模式，以更好地组织和分离系统的不同层次。Model 层为 OrgModel.py 用于实现不涉及可视化的功能。View 层为 gui.py 和

PersonDetailDialog.py 主要负责主页面以及对话框的可视化实现。Controller 层为 controller.py 用于实现用户操作和具体函数的链接。

1. Model 层

设计 person 类和 Department 类，分别代表人员和部门，人员类中包含人员基本信息，部门类中除了该部门的信息之外还设计 parent 和 children 用于存储子节点。得益于 python 的强大功能，可以使用列表不受限制存储子节点，更符合实际的组织结构。

设计 OrgModel.类用于管理这两种类，OrgModel 包含树结构根节点 root，和一个字典充当线性表，是主要的存储结构。此外核心功能的实现也都在 OrgModel 类中。

2. View 层

View 层主要体现在 gui.py 中，负责主页面的构建，其余页面以功能为线索进行串联，每一个功能按钮都额外调出对话框。在对话框中重新设计该功能需要的界面。而无需考虑主页面的影响。

3. controller 层

Controller 类中包含 view（gui.py）和 model（OrgModel），负责实现界面与底层方法的链接交互。

当点击按钮时，调用 Controller 类中的方法，该方法通过调用 Model 层中的方法得到结果，然后调用 view 层中的方法在界面上进行展示。

2.4 逻辑结构与物理结构

2.4.1 逻辑结构

部门结构抽象为树形结构，根节点为该高校，后代节点设置为层级的部门。每一个节点为 Department 类的实例。

由于一个人可以在多个部门工作，因此通过线性结构对所有人员进行存储。线性表中的每一项是一个 Person 类实例。

2.4.2 物理结构

树形结构，定义 Department 类如下：

```
class Department:
def __init__(self, department_id: str, name: str, parent=None):
    """
    构造函数，用于初始化 Department 对象的属性。
    :param department_id: 部门 ID
    :param name: 部门名称
    :param parent:  父部门引用
    """
    self.department_id: str = department_id
    self.name: str = name
    self.parent = parent
    self.children = []
    # roles 字典用于存储担任具体职务的人员
    self.roles = {}
    # role_categories 列表用于定义部门内职务的分类
    self.role_categories = {'主管': [], '副主管': [], '其他人员': []}
```

parent 用于存放父节点，根节点的父节点为 None, children 列表负责存放所有子节点的引用，

在 OrgModel 类中，存储根节点

```
def __init__(self):
    """初始化模型，建立一个根部门和空的人员花名册"""
    self.root_department: Department = Department("root", "同济大学")
    self.personnel_roster: dict[str, Person] = {}
```

线性结构，利用字典实现，可以根据人员 id 快速查找

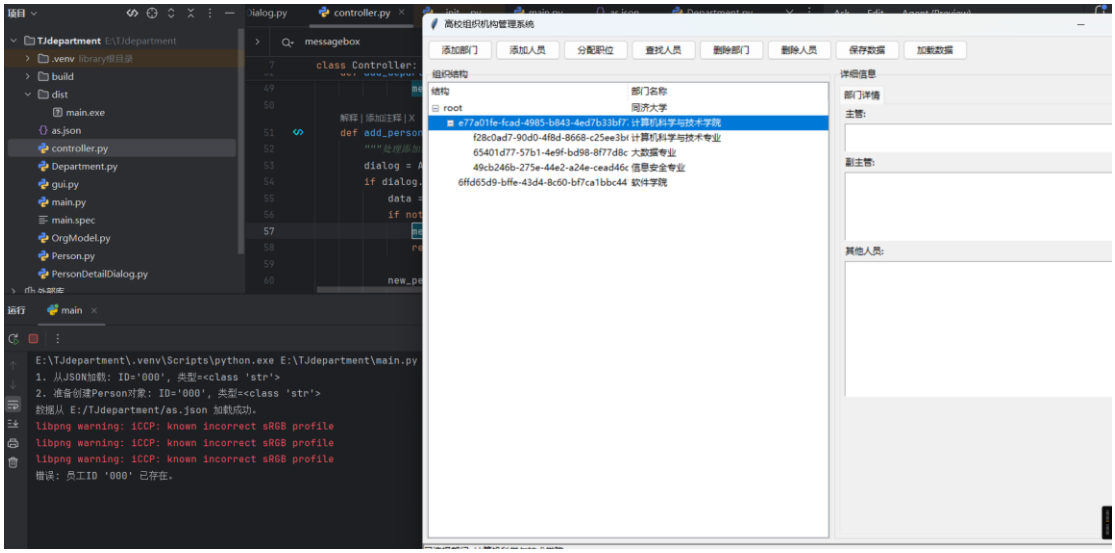
```
self.personnel_roster: dict[str, Person] = {}
```

2.5 开发平台

操作系统：Window 11 ，集成开发环境（IDE）：PyCharm ，库和框架：tkinter

2.6 系统的运行结果分析说明

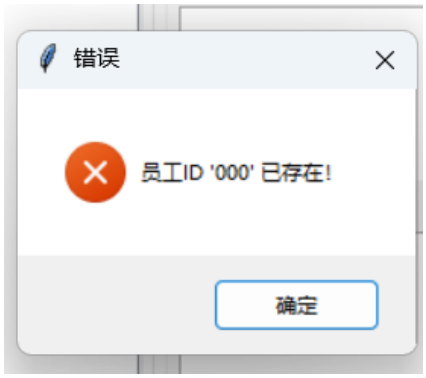
2.6.1 运行 main 函数



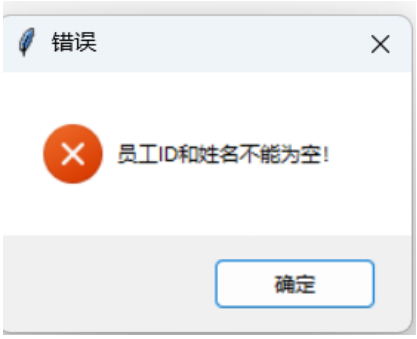
能够正确记录结果，并且在控制台输出相关调试信息

2.6.2 软件鲁棒性和正确性：

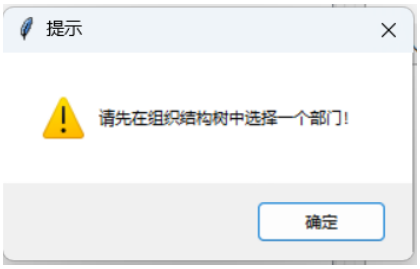
1.添加已存在员工 id



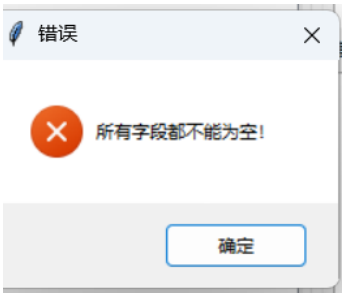
2.添加人员为空

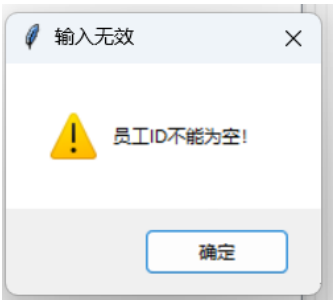


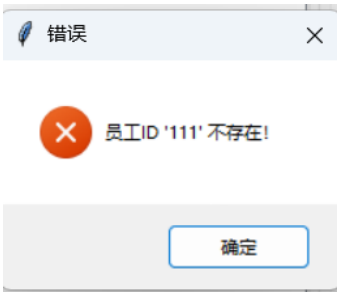
3.未选择部门



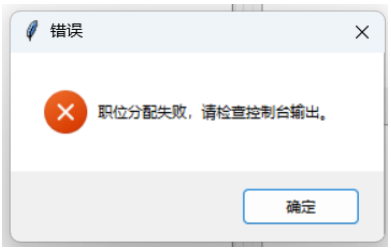
4、分配/删除职务错误



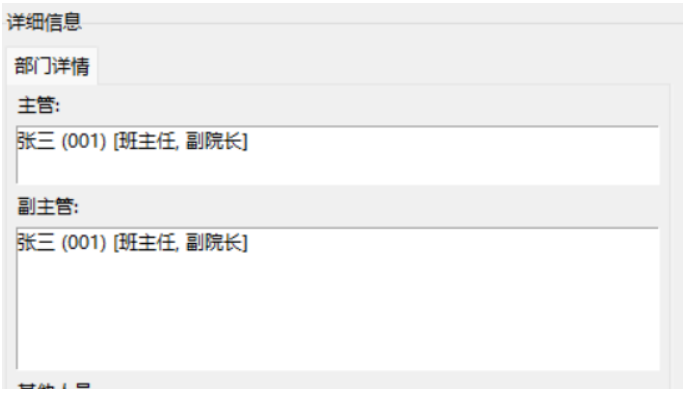




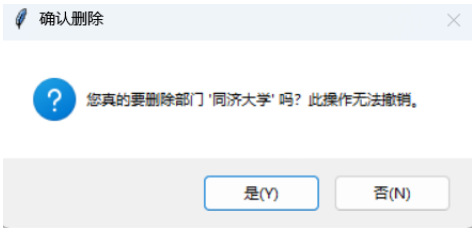
5.主管只能设置一位



6.同一人在同一部门的不同职业全部显示，在不同部门正常显示

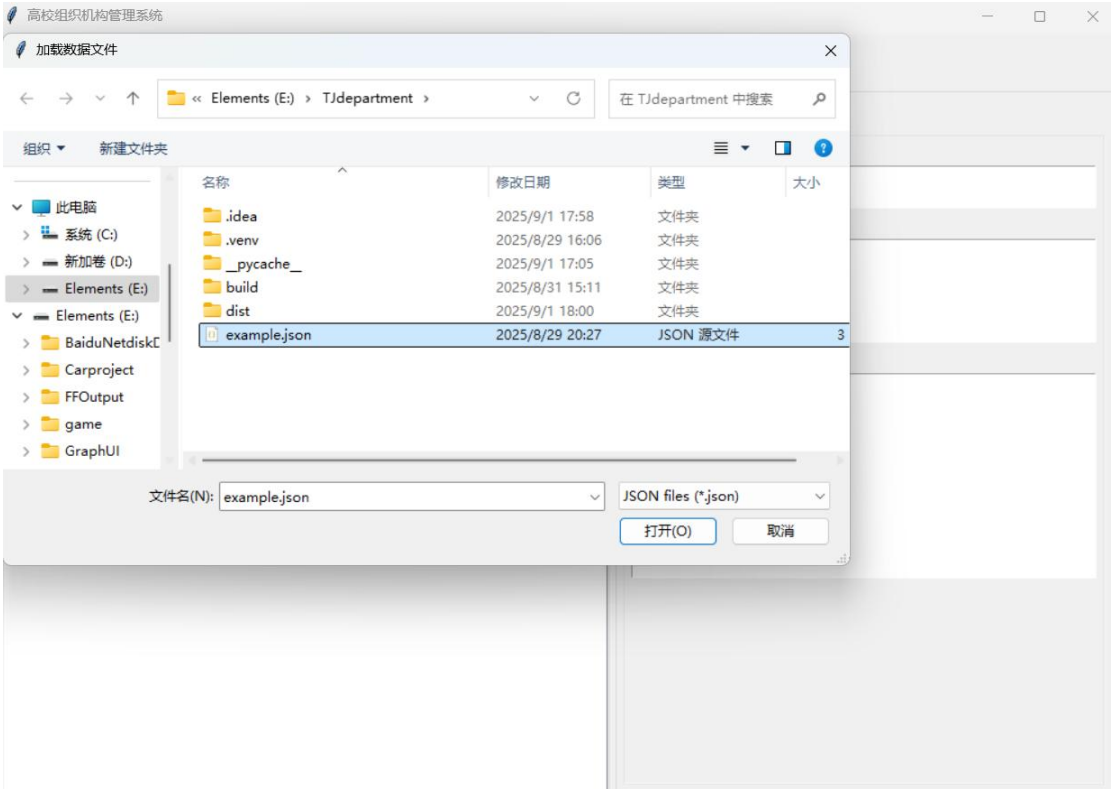


此外对于删除等不可逆操作，还设计了确认对话，防止误操作。



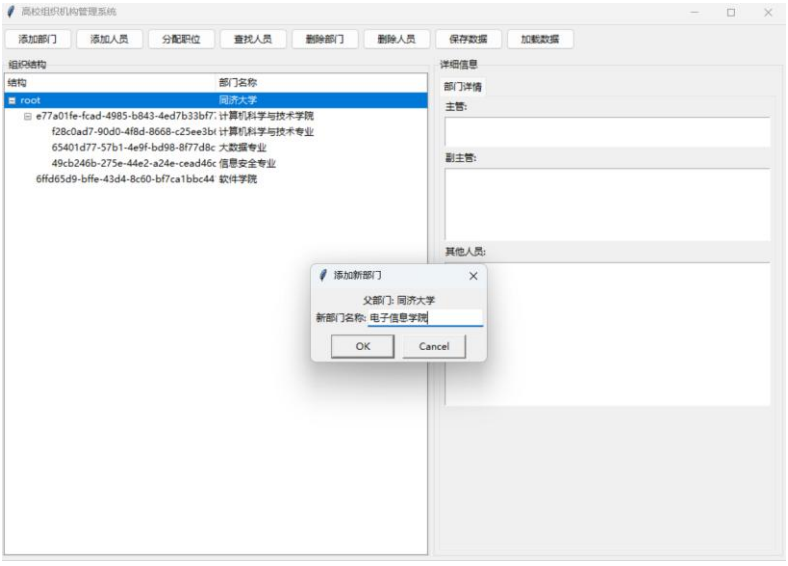
2.7 运行案例

双击 exe 文件打开程序

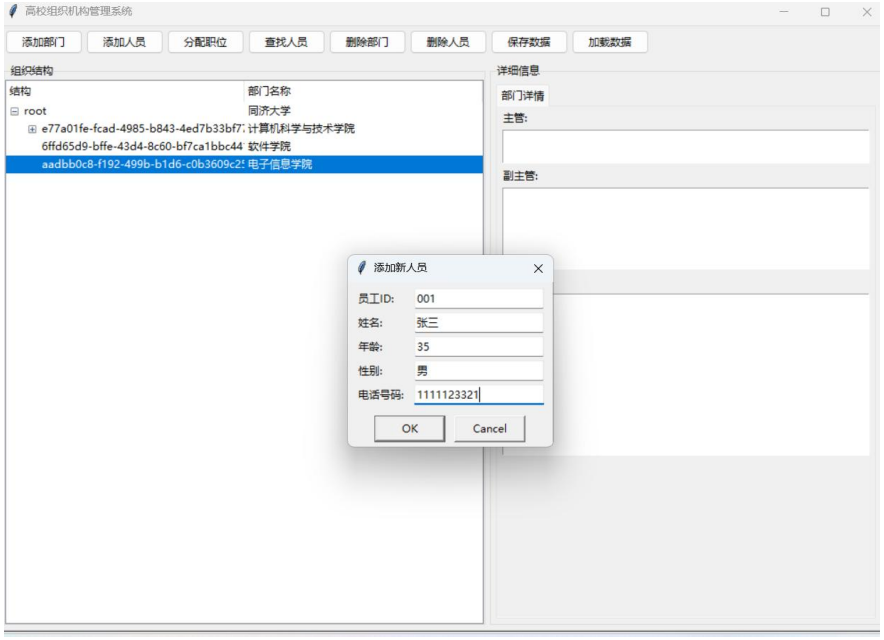


加载准备好的文件

添加部门

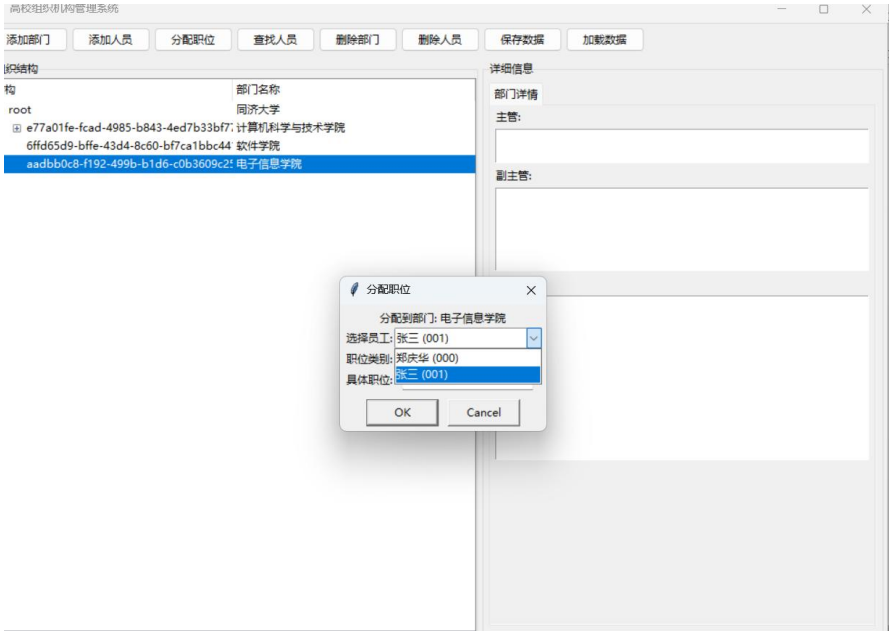


添加人员

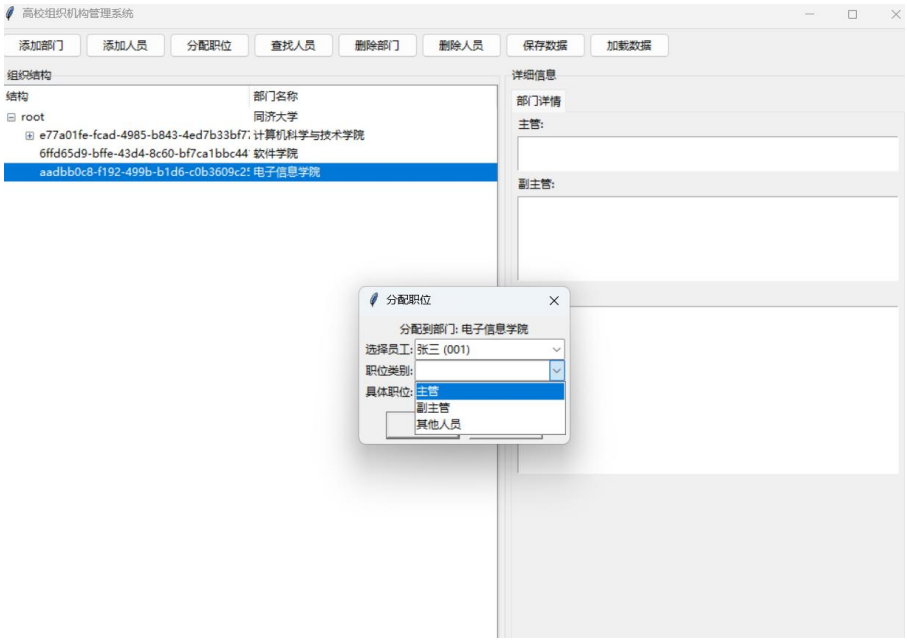


分配职务

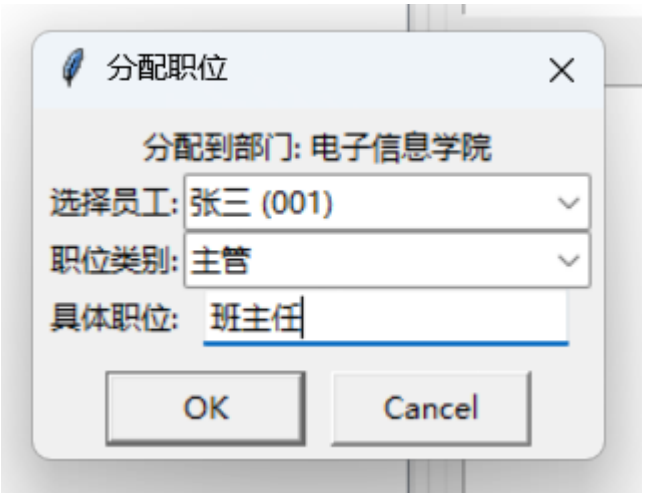
选择人员



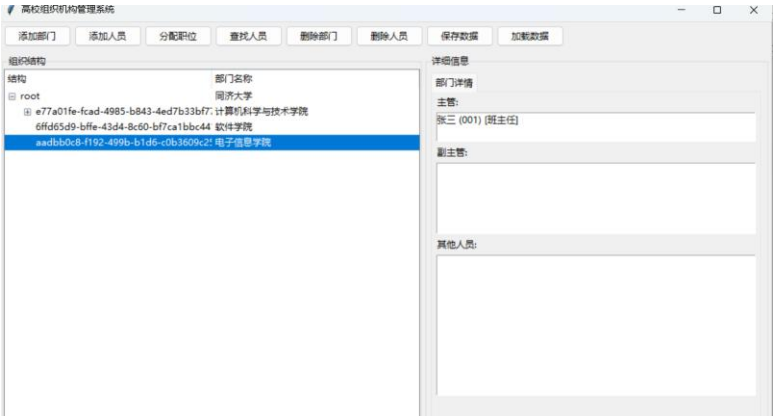
选择职务类别



输入职位



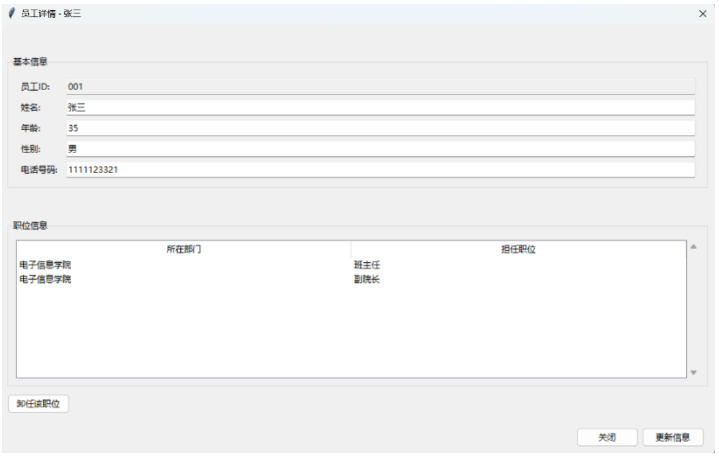
再次点击树结构进行刷新，右图即可出现相应人员



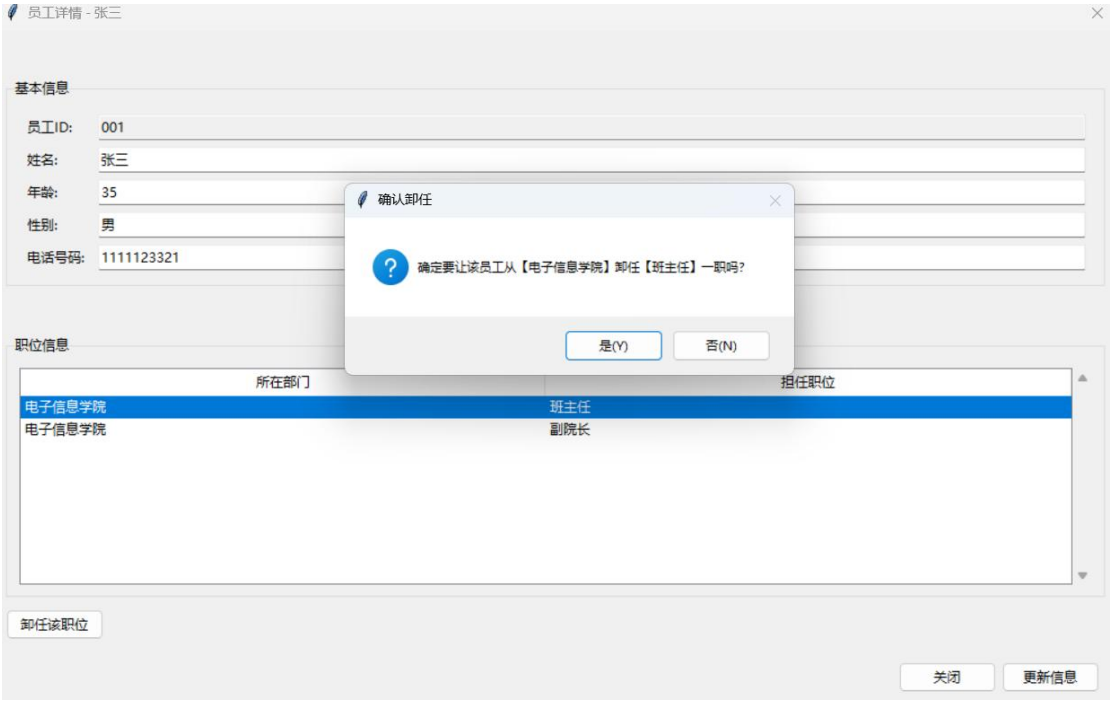
查找人员

默认显示所有人员

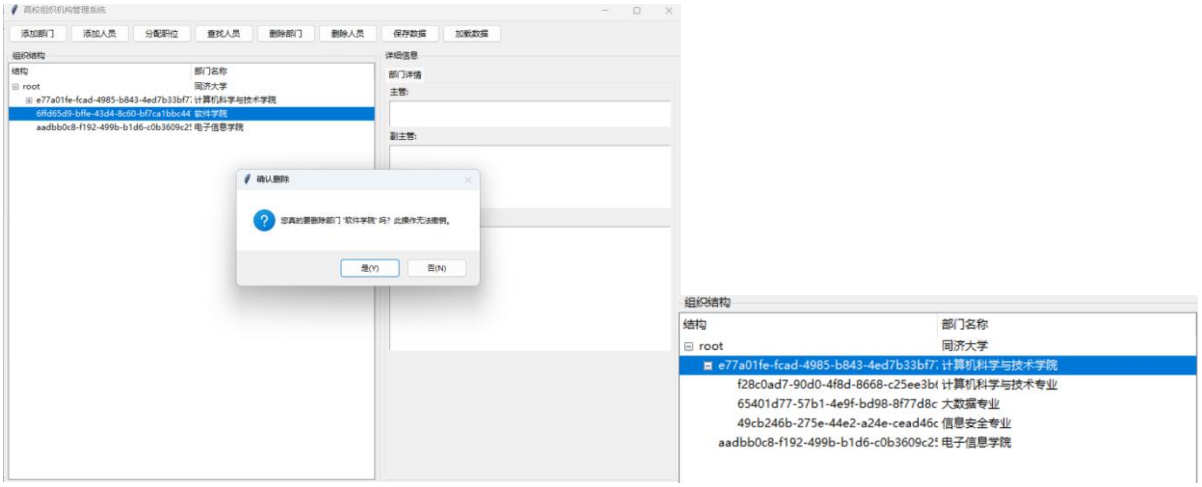
双击显示具体信息



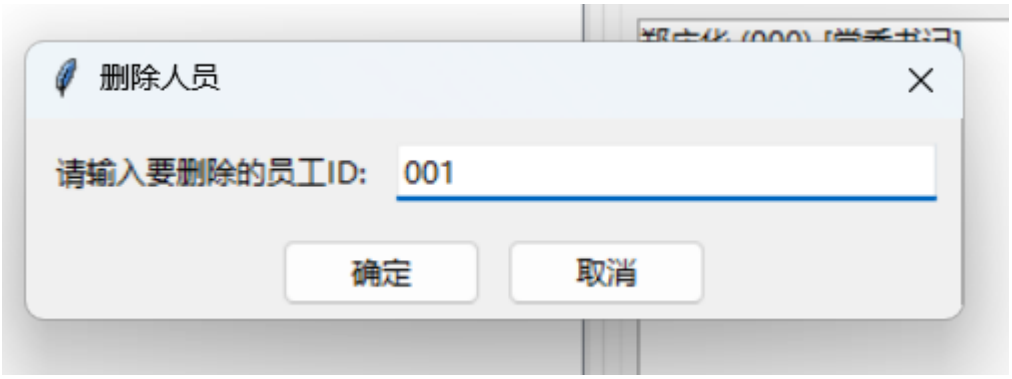
此处可以更新信息（id 只读），卸任职务

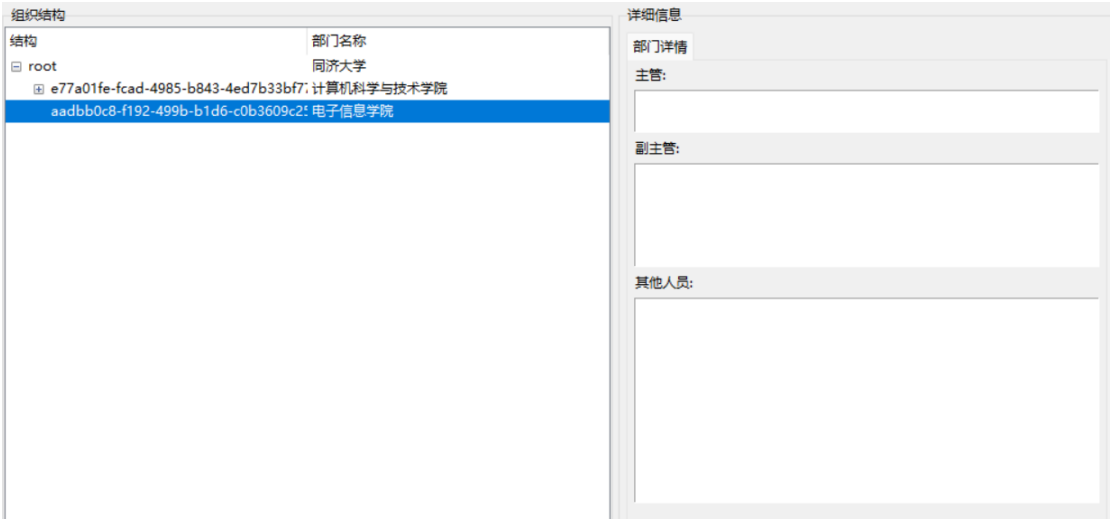


选中删除部门



根据人员 id 删除人员




组织结构
结构
部门名称
root
同济大学
e77a01fe-fcad-4985-b843-4ed7b33bf7
计算机科学与技术学院
aadbb0c8-f192-499b-b1d6-c0b3609c2
电子信息学院
详细信息
部门详情
主管:
副主管:
其他人员:

人员成功删除

第三部分 实践总结

3.1 所做的工作

3.1.1. 学习 tkinter 库的使用

通过本次项目，我们系统性地学习并实践了 Tkinter 库的核心功能，从搭建窗口到复杂的交互设计，具体体现在：

1.界面搭建与组件运用:

熟练使用 tk.Tk() 创建主窗口，并利用 ttk.Frame 和 ttk.Labelframe 对界面进行模块化分区。

掌握了多种核心组件的用法，例如用于操作的 ttk.Button、用于输入的 ttk.Entry、用于信息展示的 ttk.Label 和 tk.Listbox，以及用于下拉选择的 ttk.Combobox。

2.高级组件的深入使用:

重点实践了两个高级组件：

ttk.Treeview: 不仅用它来清晰地展示层级化的组织结构树，还用它来以表格形式显示所有员工列表，并掌握了如何使用其 iid 属性安全地存储和检索数据 ID，从根本上解决了数据显示与内部逻辑分离的问题。

ttk.Notebook: 用于在详情区域创建标签页，实现了在同一空间内切换显示“部门详情”和“人员信息”的功能，优化了信息组织的结构。

3.对话框设计与交互:

掌握了两种对话框的构建方法。一是通过继承 tkinter.simpledialog.Dialog 快速构建标准对话框（如删除确认、添加职位等），并学会了重写 body, buttonbox, apply 等方法；二是通过继承 tk.Toplevel 构建了功能更强大、布局更灵活的可编辑模态对话框（如员工详情页），并深入理解了通过 grab_set() 和 wait_window() 实现模态效果和数据回传的机制。同时，熟练运用 tkinter.messagebox 进行信息提示、警告和操作确认。

4.事件处理:

掌握了通过 command 参数绑定按钮点击事件，并通过 bind() 方法为组件绑定了更复杂的事件，如 "<<TreeviewSelect>>"（列表项选择事件）和 "<Double-1>"（鼠标双击事件），实现了界面元素之间的动态交互。

3.1.2. 深入实践 python 面向对象机制

整个项目是围绕面向对象编程（OOP）思想构建的，具体实践如下：

1.封装 (Encapsulation):

数据模型封装: 创建了 Person 和 Department 类来封装各自的属性和行为。例如，Department 类不仅存储了部门名称、ID 等数据，还封装了添加/删除子部门、管理内部人员等操作。OrgModel 类更是将整个应用的数据（部门树、人员花名册）和所有业务逻辑（增删改查、文件读写）封装起来，对控制器提供统一的接口。

视图封装: MainApplication 类和各个 Dialog 对话框类分别封装了主窗口和弹出窗口的所有 UI 组件及其基本配置，使 UI 层具有很强的独立性。

2.继承 (Inheritance):

在创建各类对话框时，通过继承 tk.Toplevel 或 simpledialog.Dialog，我们复用了父类大量关于窗口管理、事件循环等基础功能，只需专注于编写我们自己需要的特定 UI 和逻辑，极大地提高了开发效率。我们在讨论中也明确了 self.ok() 和 self.cancel() 等方法就是直接从父类继承而来的。

3.1.3. 复习拓扑排序和关键路径算法

在项目初期进行的有向图分析工具开发中，我们对数据结构与算法进行了复习和实践：

拓扑排序: 实现了基于卡恩算法（Kahn's algorithm）的拓扑排序，通过队列和节点的入度（in-degree）来确定一个有向无环图的线性序列。在项目中，我们还实现了该过程的可视化，能够动态展示排序过程中每个节点入度的变化情况。

关键路径: 实现了关键路径算法（Critical Path Method），用于计算项目管理网络中的核心活 动序列。实践中计算了事件的最早发生时间（Ve）和最晚发生时间（Vl），以及活动的最 早开始时间（E）和最晚开始时间（L），最终通过计算时间余量（Slack）来确定关键路径，并在界面上进行了高亮显示。

3.1.4. 掌握可视化界面布局设计

本次项目对 GUI 布局设计的掌握是全方位的，从简单的排列到复杂的响应式布局：

掌握核心布局管理器: 深入理解并实践了 pack(), grid() 两种主要布局管理器的使用场景和区别。例如，使用 pack() 进行顶层框架的线性布局（如顶部按钮栏），使用 grid() 进行二维对齐的表单布局（如部门详情和各类对话框）。通过解决实际的报错，深刻理解了“同一容器内不能混用 pack 和 grid”的核心原则。

模块化与响应式布局: 学会了使用 ttk.PanedWindow 创建可拖拽、可自由调整宽度的左右分栏布局，实现了组织树和详情页的灵活空间分配，提升了用户体验。通过将不同功能的 UI 元素封装在各自的 Frame 中，实现了布局的模块化，使得整体结构清晰，易于维护和修改。

3.2 总计与体会

本次项目最大的收获是深入实践了 MVC（模型-视图-控制器）设计模式。在初期，我可能只会将所有代码写在一起，但通过将负责数据处理的模型(Model)、负责界面展示的视图(View)和负责逻辑调度的控制器(Controller)**清晰地分离开来，我深刻体会到这种架构带来的巨大优势：代码结构清晰、逻辑互不干扰、极大地降低了维护和扩展的难度。

开发过程并非一帆风顺，但解决问题的过程让我收获巨大。从最初混用.pack()和.grid()导致的布局冲突，到因未正确传递 controller 引用而引发的 AttributeError，再到 Treeview 组件自动转换数据类型导致的前导零丢失问题，每一次调试都让我对 Tkinter 的底层机制和编程的严谨性有了更深的理解。