同济大学

TONGJI UNIVERSITY

数据结构课程设计

课题名称	数据结构课程设计
副 标 题	二叉树与社会关系网络
学　　院	国豪书院
专　　业	计算机科学与技术（精英班）
学生姓名	倪雨舒
学　　号	2353105
日　　期	2025 年 9 月 7 日

目 录

1 算法实现题目设计——线索二叉树

1.1 题目内容

按照选题规则，我选了算法实现题的第 5 题《二叉树》，题目描述如下：二叉树，完成：(1) 建立一棵二叉树，并对它进行先序、中序、后序遍历；

(2) 统计树中的叶子结点个数；

(3) 分别对它进行先序、中序、后序线索化；

(4) 实现先序、中序线索树的遍历；

显示该树和线索化后的树（此要求可视情况选择是否完成）。

1.2 软件功能

具体实现的所有功能如下：

1.2.1 建树与编辑树

在“建树”界面输入层高后，程序按层级递归生成满二叉树；为每个结点自动编号并记录父指针，便于后续删除与可视化布局。构建完成后立即重绘：清理旧图元与覆盖物（连线、箭头、高亮），按层级计算坐标并生成圆形节点与父子连线。点击任意结点时，若该结点在语义上为“真实叶子”（仅当左右两侧标记为 Child 的孩子均为空；线索不算孩子），弹出确认框后可删除：**唯一根结点：**若为唯一根，删除后整棵树置空。**一般叶子：**断开其与父结点的左/右链接并释放该叶子。

删除后统一重绘，确保动画序列、统计结果、线索箭头与最新树形一致；若之前做过线索化，回到展示页会先清理旧线索再进行新的线索化，避免悬挂指针。

1.2.2 统计节点个数

自根起递归计数，仅沿 Child 方向遍历，使“普通树”和“已线索化的树”得到一致的节点总数。统计值写入右侧文本输出区，画面保持不变，便于对照结构与数值。

1.2.3 先序/中序/后序遍历

每次遍历同时生成（1）用于动画的访问序列，（2）用于文本区的编号序列。遍历只沿 Child 方向下探，避免把线索当子树，保证线索化前后结果一致。使用定时器按序列逐点高亮，播放期间可清高亮或重播。

1.2.4 先序/中序/后序线索化

每次线索化开始前先全树清线索，再按所选次序对结点执行“前驱–后继”补线索。线索化完成后，遍历全树把 ltag=Threadltag=Thread 或 rtag=Threadrtag=Thread 的指针绘制为覆盖物箭头（左线索

表示前驱箭头、右线索表示后继箭头)；箭头端点采用圆周锚点计算，使连线从节点边缘射入/射出，避免穿过圆心。

1.2.5 先序/中序线索化遍历

在已完成对应线索化后点击“先序/中序线索化遍历”后进行线索化遍历；**中序线索遍历：**从整棵树的最左结点起步；若当前结点右侧为线索，直接跳到线索后继；否则进入右子树并一路取最左。**先序线索遍历：**优先沿左孩子前进；走不动时顺着右指针推进。与普通遍历一致，逐点高亮并在文本区输出访问序列；避免系统栈，直观展示线索的作用。

1.2.6 限制条件

在所有的功能展示按钮点击之后可以通过点击其他按钮来展示其他功能，这里按钮之间的关系并不冲突，体现了软件的适应性较好。此外树高限制在 2-6 层，输入其他数字将无法输入，建立树的时候，如果将所有节点全部删除的话也无所谓，程序也可以进行功能展示，只不过没有任何与树有关的信息产生。

1.3 设计思想

1.3.1 界面切换

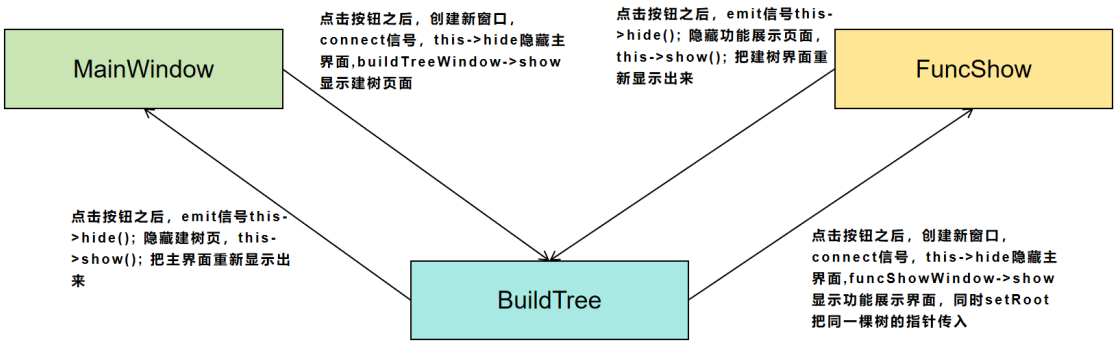

图 1.1 界面切换逻辑示意图

主窗口 (MainWindow) 只充当调度器。点击“创建二叉树”时，如果 BuildTree 子窗口尚未创建就 newBuildTree，并把它的自定义信号 returnToMain() 连接到自身槽 handleReturn()；随后 buildTreeWindow->show() 并对主窗口调用 hide()，实现前-后台切换。

建树页 (BuildTree) 内部在“返回”按钮槽中发射 returnToMain()，同时 hide() 自己；主窗口收到该信号后简单 show() 自己即可。

建树页点击“功能展示”按钮时，按需创建 FuncShow，把当前树根注入给它，然后 show() 展

示页并 hide() 建树页；展示页点击返回按钮发射 backToBuildTree()，建树页槽 handleBack() 再把自己 show() 出来。这样三层页面用信号/槽+show()/hide() 完成切换。

1.3.2 各个类之间的继承关系

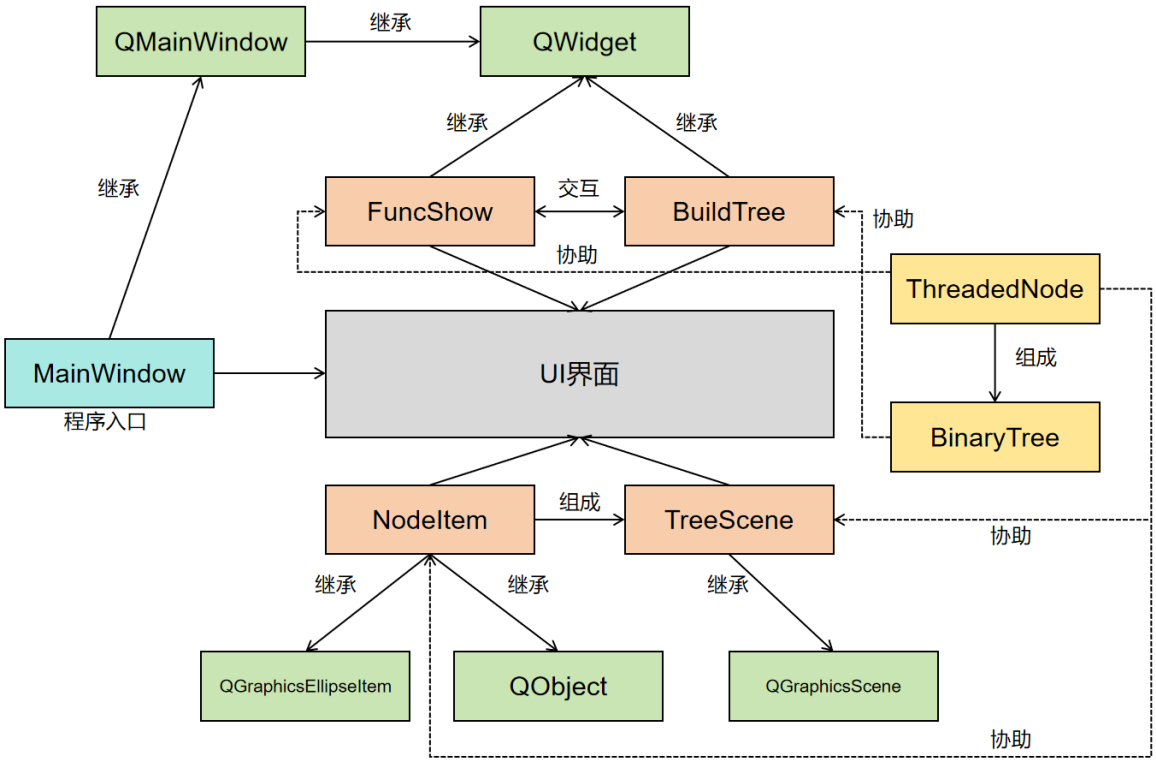

图 1.2 各个类之间的继承关系示意图

数据层（详细的介绍在逻辑结构和物理结构部分）：

ThreadedNode：二叉线索树结点，维护 left/right/parent 与 ltag/rtag（Child/Thread）语义，以及可视化辅助信息（例如布局坐标）；不继承任何 Qt 基类。

BinaryTree：树的容器与算法实现（建树、遍历、线索化、删除叶子等）；不继承任何 Qt 基类。

可视化层:

TreeScene:QGraphicsScene 负责布局（为每个结点计算坐标）、绘制节点与父子连线、绘制线索箭头覆盖物、节点高亮。

NodeItem:QObject,QGraphicsEllipseItem（多继承）每个节点对应一个可交互图元（圆形节点+文本标签+高亮）；之所以多继承，是因为既要有图元外观（QGraphicsEllipseItem），又要能发信号（QObject）。点击时发出 clicked(ThreadedNode*) 供场景转发。

窗口/页面层：

BuildTree:QWidget 建树与基础交互页：持有 BinaryTree（数据）与 TreeScene（渲染），连接场景的 nodeClicked 信号以处理删除叶子等逻辑；向 MainWindow 发出 returnToMain()。

FuncShow:QWidget 演示遍历与线索化动画：内部也持有一个 TreeScene；通过 setRoot() 与 BuildTree 共享同一棵树；提供 backToBuildTree() 用于返回。

MainWindow:QMainWindow 入口与页面调度：负责创建/显示 BuildTree，响应其回退信号以恢复主界面。

1.4 逻辑结构与物理结构

本程序的核心数据结构是二叉树。二叉树的逻辑结构: 二叉树是 n(n≥0) 个结点的有限集合，该集合或者为空集合，或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树的二叉树组成。

二叉树的物理结构: 在本程序中采用链式存储结构，利用二叉链表的方式将父节点和它的两个孩子结点关联在一起。以下对物理结构进行详细说明，物理结构包含两部分组成，分别为数据层和可视化层，最顶层为处理相关按钮的逻辑，在此不做介绍。

1.4.1 数据层

（1）ThreadedNode——线索二叉树的最小数据单元

ThreadedNode 是这棵线索二叉树里最小、最纯粹的“节点模型”（不继承任何 Qt 类）。它内部有三根基本指针 left/right/parent，并用一对标记 ltag/rtag 来区分每根指针当前的语义——到底指向“真实孩子”（Child=0）还是一条“线索”（Thread=1）。节点还保存 value，以及给可视化用的辅助信息：posHint（预布局坐标）、gfx（可选地绑定到图形项）和 selected（是否高亮）。

在结构判断上，hasLeftChild()/hasRightChild()/isLeaf() 都遵循同一原则：只认 Child 方向的非空指针，线索一律不算孩子。为了不破坏这套语义，所有修改都必须走统一入口：设孩子用 setLeftChild()/setRightChild()（同时把对应 tag 置为 Child 并维护好 parent）；设线索用 setLeftThread()/setRightThread()（把 tag 置为 Thread，但不改 parent）；清线索有两级——节点级的 clearThreads() 只在本节点把线索侧清空并还原为 Child，而整树级的 ClearAllThreads(root) 会“只沿孩子侧下钻、只在线索侧清空”，ResetParent 则专门沿孩子方向重刷父指针。为了配合中序相关的功能，节点还提供 firstInorder()：从当前子树出发，沿 Child 的左分支一直走到最左端，常用来作为中序遍历或线索遍历的起点。

```
/**
 * 线索二叉树的“节点”——仅封装节点本身的状态与常用操作。
 * 不负责节点的生命周期管理（释放由树统一处理）。
 */
class ThreadedNode
{
public:
    // 指针语义：Child 表示真实孩子；Thread 表示线索指向前驱/后继
    enum Tag : quint8 { Child = 0, Thread = 1 };
    // —— 数据域（按需替换为 QString / QVariant / 模板）——
    int value = 0;
    // —— 指针域 ——（parent 非必需，但用于可视化/删除判断很方便）
```

```
    ThreadedNode* left   = nullptr;
    ThreadedNode* right  = nullptr;
    ThreadedNode* parent = nullptr;
    // ——— 线索标记 ——— (与 left/right 配对)
    Tag ltag = Child;
    Tag rtag = Child;
    // ——— 可视化相关———
    QGraphicsItem* gfx = nullptr;   // 绑定到场景中的图元
    QPointF        posHint;         // 预布局坐标 (可选)
    bool           selected = false;
public:
    explicit ThreadedNode(int v = 0, ThreadedNode* p = nullptr);
    // 结构类工具
    bool hasLeftChild()  const;   // 左孩子存在且为 Child
    bool hasRightChild() const;   // 右孩子存在且为 Child
    bool isLeaf()        const;   // 无真实左右孩子 (线索不算孩子)
    bool deletableLeaf() const;   // 是否安全作为“叶子”删除
    // 线索存在性 (用于可视化与遍历判定)
    bool hasLeftThread()  const { return ltag == Thread && left  != nullptr; }
    bool hasRightThread() const { return rtag == Thread && right != nullptr; }
    // 修改关系 (统一维护 parent 与标记)
    void setLeftChild (ThreadedNode* child);
    void setRightChild(ThreadedNode* child);
    // 线索设置/清理 (不会动 parent)
    void setLeftThread (ThreadedNode* predecessor);
    void setRightThread(ThreadedNode* successor);
    void clearThreads();                    // 如果当前为线索则清空并还原为 Child
    // ——— 树级辅助 (为“重新线索化/删除后恢复”等流程准备) ———
    static void ClearAllThreads(ThreadedNode* root);          // 递归清整棵树线索
    static void ResetParent(ThreadedNode* root,
                            ThreadedNode* parent = nullptr);  // 递归刷新 parent
    // 可视化辅助
    void bindGraphics(QGraphicsItem* item) { gfx = item; }
    void setSelected(bool on) { selected = on; }
    // ——— 中序导航: 结合线索与孩子, 做局部邻接查找 ———
    ThreadedNode* firstInorder();       // 以当前结点为根找到最左端
    ThreadedNode* inorderSuccessor();   // 中序后继 (考虑线索)
    ThreadedNode* inorderPredecessor(); // 中序前驱 (考虑线索)
};
```

(2) BinaryTree——树与相关的算法

BinaryTree 是整棵树的核心：一手管理节点的创建/销毁，一手实现所有与树相关的算法。它主要负责几件事：建树、清理、统计叶子数、三种普通遍历（先/中/后）、三种线索化与相应的线索遍历（先序/中序），以及删除叶子等修改操作。

在生命周期上，clear()/destroy(p) 会先把所有“线索侧”的指针清掉，再做递归释放，避免把线索误当作孩子一路走下去。建树用 buildFullByHeight(h)：先清旧树，从第 1 层开始递归生成满二叉树，并给每个新节点顺序编号，方便在 UI 上展示；具体的连接通过 buildFullRec 完成，内部只用 setLeftChild()/setRightChild() 去建立“孩子语义”，这能自动维护好 parent 与标记位。

日常算法都遵循同一个原则：只沿 Child 方向下探。比如叶子计数只计算真实孩子，不把线索算进来；普通遍历（先/中/后）也只沿 Child 侧递归，并把结果组织成 QString 以便直接显示在文本区。

线索化采用统一的“pre 规则”：访问当前结点 p 时，如果 p->left 为空就把它的左指针指向“前驱”pre 并将 ltag 标为 Thread；如果 pre 存在且 pre->right 为空，就让 pre->right 指向 p 并将 pre->rtag 标为 Thread；最后更新 pre=p——三种（先/中/后）线索化都遵循这套流程，且都会先清线索再线索化。基于线索的遍历是非递归的：中序从 firstInorder() 起步，反复“走线索后继或进右子树再找最左”；先序则能走左孩子就走左，走不动就顺着右指针（右孩子或线索后继）前进。

修改操作里最典型的是删除叶子：只允许删除真实叶子。如果目标节点既是根又是唯一节点，就把根置空；否则断开它与父节点的左/右连接并释放该节点。整套实现把“孩子/线索”的语义牢牢区分开来，让建树、遍历、线索化和删除都各司其职而又互不干扰。

```
/**
 * 纯“数据结构层”的二叉树：
 * - buildFullByHeight(h): 依据层高建立满二叉树（h>=1）
 * - clear(): 释放整棵树
 * - 遍历：先/中/后序（递归），返回 QString（便于 UI 显示）
 * - leafCount(): 叶子结点数（不把线索当孩子）
 * - 线索化：先/中/后序；并提供中序与先序的线索遍历
 */
class BinaryTree
{
public:
    BinaryTree() = default;
    ~BinaryTree();
    ThreadedNode* root() const { return m_root; }
    // 依据层高建立满二叉树；会自动清空旧树
    ThreadedNode* buildFullByHeight(int height);
    // 清空整棵树
    void clear();
    // 统计叶子数（真实孩子，线索不算孩子）
    int leafCount() const;
    // —— 普通递归遍历（不依赖线索）——
    QString preorder()  const;
    QString inorder()   const;
    QString postorder() const;
    // —— 线索化：会把“空指针”按给定遍历序补为 前驱/后继 线索 ——
    void makePreorderThread();   // 先序线索化
    void makeInorderThread();    // 中序线索化
    void makePostorderThread();  // 后序线索化
    // —— 线索遍历（非递归）——
    QString inorderThreadedWalk()  const;  // 中序线索遍历
    QString preorderThreadedWalk() const;  // 先序线索遍历（简单可靠）
    // 删除叶子（真实孩子意义上的叶子）
    bool removeLeaf(ThreadedNode* n);
private:
    ThreadedNode* m_root = nullptr;
    // 释放整棵树
    void destroy(ThreadedNode* p);
    // 递归建树：当前层 curr，目标层 max（根为 1）
    ThreadedNode* buildFullRec(int curr, int max, ThreadedNode* parent, int& nextVal);
    // 遍历工具（普通）
    void preorderRec (ThreadedNode* p, QString& out)  const;
    void inorderRec  (ThreadedNode* p, QString& out)  const;
    void postorderRec(ThreadedNode* p, QString& out)  const;
```

```
    int  leafCountRec(ThreadedNode* p) const;
    // 线索化工具（所有版本都按“访问当前结点时”去补线索）
    void clearThreadsRec(ThreadedNode* p);                  // 线索化前清线索
    void preorderThreadRec (ThreadedNode* p, ThreadedNode*& pre);
    void inorderThreadRec  (ThreadedNode* p, ThreadedNode*& pre);
    void postorderThreadRec(ThreadedNode* p, ThreadedNode*& pre);
};
```

1.4.2 可视化层

（1）TreeScene——数据到图形的中继

renderTree(root) 先把所有“覆盖物”（箭头、高亮等）清掉，再调用 clear() 把场景里的图元和内部映射一并清空，防止残留的箭头或高亮悬挂；若根为空直接返回。随后按“先布局、后绘制”的顺序完成整棵树的重建，最后根据所有图元的包围盒设置 sceneRect，四周留一点边距，避免视图裁切。

紧凑布局进行两遍：第 1 遍 layoutAssignX 负责 X 坐标：从底向上布点——叶子按 leafCursor 从左到右依次落位，内部结点取左右子树 X 的中点；水平间距会随深度按 shrink 逐层收紧。结果写入 p->posHint.x()。第 2 遍 applyYByDepth 只干一件事：按深度给出 Y 坐标，并且只沿 Child 方向递归（线索不参与布局）。布局密度由 mdxBase/mshrink/mdy 三个参数调节；mitems 维护 ThreadedNode*→NodeItem*的映射；moverlays 统一管理箭头和高亮，这些成员共同决定了树形的宏观外观与后续清理行为。

ensureNodeItem 会为每个节点创建/复用 NodeItem，把它放到 posHint 指定的位置，设置显示编号，并把 NodeItem::clicked 直接连到场景的 nodeClicked 信号，方便页面层统一处理点击事件。结构连线在 drawDfs 中完成：只画 Child 方向的“父子边”，端点用圆周锚点（避免线穿过圆心），同时把边放在下层、节点放在上层，层级更清晰。

clearOverlays() 一键清除所有覆盖物；clearHighlightsOnly() 只恢复节点的高亮状态而保留箭头；highlightNode() 可对某个节点上色高亮。需要画箭头时用 addArrow(from,to,color,dashed)：根据两端的圆周锚点画出箭身和箭头三角，支持虚线样式，并把生成的图元登记进 moverlays，便于之后统一清理。无论是“线索箭头”还是“遍历辅助箭头”，都走这一套流程。

```
class TreeScene : public QGraphicsScene
{
    Q_OBJECT
public:
    explicit TreeScene(QObject* parent = nullptr);
    // 渲染整棵树（会清空 scene 与 item 映射；覆盖物也会清理）
    void renderTree(ThreadedNode* root);
    // 展示页：临时覆盖物/高亮控制
    void clearOverlays();                                   // 清临时覆盖物（箭头等）
    void clearHighlightsOnly();                             // 只还原高亮
    void highlightNode(ThreadedNode* n, bool on,
                       const QColor& fill = QColor("#FFE08A"));     // 高亮/恢复
    void addArrow(ThreadedNode* from, ThreadedNode* to,
```

```
                  const QColor& color = QColor("#0080FF"),
                  bool dashed = true);                                  // 画箭头
signals:
    void nodeClicked(ThreadedNode* n);
private:
    // ——— 布局（两遍）———
    void layoutCompact(ThreadedNode* root);
    double layoutAssignX(ThreadedNode* p, int depth, double& leafCursor);
    void applyYByDepth(ThreadedNode* p, int depth);
    // ——— 绘制 ———
    void drawDfs(ThreadedNode* p);
    NodeItem* ensureNodeItem(ThreadedNode* p);
    NodeItem* itemOf(ThreadedNode* p) const { return m_items.value(p, nullptr); }
private:
    // 布局参数
    double m_dxBase    = 80.0;   // 叶子在第 0 层（根的层为 0）基础水平间距
    double m_shrink    = 0.85;   // 每深入一层水平间距乘该因子（越小越“瘦”）
    double m_dy        = 110.0;  // 垂直层距
    QMap<ThreadedNode*, NodeItem*> m_items;
    QList<QGraphicsItem*> m_overlays;      // 箭头、高亮辅助形状
};
```

（2）NodeItem——可交互的节点圆半径 R=12，白底黑边，Z 值置 1 保证“点在边之上”；内部内置 QGraphicsSimpleTextItem 显示编号并居中。centerScene() 返回圆心场景坐标；anchorTowards(dst) 用向量归一+半径计算圆周锚点（依赖 std::hypot），供连线/箭头端点使用。

setHighlighted()/setStrokeColor()/setLabel()/setLabelColor() 提供高亮、描边与文字控制；mousePressEvent 发射 clicked(ThreadedNode*)。NodeItem::clicked→TreeScene::nodeClicked（ensureNodeItem 中连接）→ 页面层槽函数（如删除叶子/高亮）。

```
// 既要信号又要图形项：多继承 QObject + QGraphicsEllipseItem
class NodeItem : public QObject, public QGraphicsEllipseItem
{
    Q_OBJECT
public:
    explicit NodeItem(ThreadedNode* n, QGraphicsItem* parent = nullptr);
    ThreadedNode* node() const { return m_node; }
    // 半径：供外部需要时使用（例如布局/碰撞计算）
    static constexpr qreal radius() { return R; }
    // 圆心（场景坐标）
    QPointF centerScene() const;
    // 从本节点指向目标点 dst（场景坐标）时，返回“圆周上的锚点”。
    // 用它作为连线端点，线就不会穿过节点。
    QPointF anchorTowards(const QPointF& dst) const;
    // 高亮/还原（遍历动画、选中等）
    void setHighlighted(bool on, const QColor& fill = QColor("#FFD86E"));
    void setStrokeColor(const QColor& c, qreal width = 2.0);
    //设置节点内文字（例如编号）
    void setLabel(const QString& text);
    void setLabelColor(const QColor& c);
signals:
    void clicked(ThreadedNode* n);
protected:
    void mousePressEvent(QGraphicsSceneMouseEvent* ev) override;
```

```
private:
    ThreadedNode* m_node;
    QGraphicsSimpleTextItem* m_label = nullptr;
    static constexpr qreal R = 10;   // 节点圆半径（稍微比原来 10 大一点更清晰）
    QColor m_normalFill = Qt::white;
    QColor m_highlightFill = QColor("#FFD86E");
};
```

数据层（ThreadedNodeBinaryTree）：ThreadedNode 只负责“一个节点”的语义与原子操作：左右/父指针与 ltag/rtag（Child/Thread）、结点值以及用于可视化的 posHint，并提供统一的“设孩子/设线索/清线索/判叶子”等接口；BinaryTree 则作为容器与算法中心，持有根指针 mroot，负责建树、遍历、线索化、计数与删除，并在销毁前先清线索以防把线索当孩子。两者是组合关系：树通过 buildFullRec 动态创建节点，并且只经由节点的统一接口建立孩子关系；节点不反向依赖树。删除与清理由 BinaryTree 统一管理。遍历、线索化、计数等算法一律只沿 Child 方向下探，线索化前先清线索，再按“访问当前结点时补前驱/后继”的统一规则执行。

可视化层（TreeSceneNodeItem）：TreeScene（继承 QGraphicsScene）充当“数据 → 图形”的桥，负责两遍布局、绘制父子边、管理覆盖物（线索箭头/高亮），并对外发出 nodeClicked(ThreadedNode*) 信号；NodeItem（QObject+QGraphicsEllipseItem 多继承）表示单个可交互节点，封装高亮/描边/标签与几何辅助 anchorTowards()，点击时发出 clicked(ThreadedNode*)。两者通过“映射与转发”关联：TreeScene 维护 ThreadedNode*→NodeItem*映射，ensureNodeItem() 创建/复用图元，并把 NodeItem::clicked 直接连接为场景的 nodeClicked 用于事件上抛；绘制连线时使用圆周锚点避免穿过圆心；覆盖物集中记录、重绘前统一清除，杜绝残留。

数据到视图的输入边界是 TreeScene::renderTree(ThreadedNode*root)——页面层把数据层的根指针交给场景，场景完成“先布局、后绘制”并设置边界。交互从视图回流到数据经由事件链：NodeItem::clicked(n)→TreeScene::nodeClicked(n)→BuildTree::handleSceneClick(n)；页面层在槽函数里调用数据层（如 removeLeaf），随后再次调用 renderTree 刷新，实现“改数据 → 重渲染”的闭环。BuildTree 同时持有 BinaryTree 与 TreeScene，建树后把 tree.root() 给场景渲染；并与功能页 FuncShow 共享同一批节点指针，确保两页操作与展示的是同一棵树。

```
void BuildTree::on_showfuncButton_clicked() {
    if (!funcShowWindow) {
        funcShowWindow = new FuncShow;
        connect(funcShowWindow, &FuncShow::backToBuildTree, this, &BuildTree::handleBack);
    }
    funcShowWindow->setRoot(tree.root());   // 两页共用同一批节点指针
    funcShowWindow->show();
    this->hide();
}
void FuncShow::setRoot(ThreadedNode* root) {
    root_ = root;                   // 直接复用 BuildTree 的根指针
    threadOrder_ = ThreadOrder::None;
    scene_->renderTree(root_);      // 用同一棵树渲染
```

```
    if (ui->outputBox) ui->outputBox->clear();
}
```

1.5 核心算法实现

（1）建树与编辑树：建立二叉树（满二叉树，按层高）UI 点击“建立二叉树”后，从输入框读层高 h，调用数据层 tree.buildFullByHeight(h) 并立即重画场景：scene->renderTree(tree.root())。数据层先清空旧树，然后自顶向下递归创建结点并连接左右孩子；每个新结点会记录 parent，左右指针通过 setLeftChild/RightChild 统一把 tag 设为 Child：删除节点（只允许删除“真实叶子”）用户点击场景里的结点，若不是叶子直接提示；是叶子则弹确认框，确认后调用 tree.removeLeaf(n) 并整棵树重渲染绘图。ThreadedNode::isLeaf() 只把 tag 为 Child 的左右指针当作孩子，线索不计入孩子，因此不会把线索误当子树。数据层删除实现如下，首先进行保护性检查如果为空指针/非叶子直接返回 false；若目标是根且唯一结点：直接 delete 并将根节点置空；否则通过 parent 断开与父节点的左/右链接，再 delete 该叶子，返回 true

```
//最顶层，建立满二叉树的交互逻辑
void BuildTree::on_buildButton_clicked()
{
    const int h = ui->treeheightBox->value();  // 读 SpinBox 的层高
    tree.buildFullByHeight(h);                 // 数据结构层生成满二叉树
    scene->renderTree(tree.root());            // 画到 QGraphicsView
}
//数据层，处理建立二叉树的过程
ThreadedNode* BinaryTree::buildFullRec(int curr, int max, ThreadedNode* parent, int& nextVal)
{
    if (curr > max) return nullptr;
    auto* node = new ThreadedNode(nextVal++, parent);
    // 满二叉树：除最后一层外必有左右孩子
    node->setLeftChild ( buildFullRec(curr + 1, max, node, nextVal) );
    node->setRightChild( buildFullRec(curr + 1, max, node, nextVal) );
    // 线索标记默认是 Child，保持不动
    return node;
}
//最顶层，通过点击删除叶子节点的交互逻辑
void BuildTree::handleSceneClick(ThreadedNode* n)
{
    if (!n) return;

    if (!n->isLeaf()) {
        QMessageBox::information(this, "提示", "这个结点不是叶子，不能删除。");
        return;
    }

    auto ret = QMessageBox::question(this, "删除叶子",
                                     QString("确认删除叶子结点 %1 ？").arg(n->value),
                                     QMessageBox::Yes | QMessageBox::No,
                                     QMessageBox::No);
    if (ret == QMessageBox::Yes) {
        tree.removeLeaf(n);                // 修改数据
```

```
        scene->renderTree(tree.root());   // 重新渲染（简单稳妥）
        }
}
// 数据层，删除叶子
bool BinaryTree::removeLeaf(ThreadedNode* n)
{
    if (!n || !m_root) return false;
    if (!n->isLeaf())  return false;

    ThreadedNode* p = n->parent;
    if (!p) {  // 根且是唯一节点
        delete n; m_root = nullptr; return true;
    }
    if (p->left == n)  p->left = nullptr;
    if (p->right == n) p->right = nullptr;
    delete n;
    return true;
}
//可视化层，用于绘制树，在建树和编辑树的过程中都用到了
void TreeScene::renderTree(ThreadedNode* root)
{
    clearOverlays(); // 先处理覆盖物
    clear(); // 删除所有图元（节点、线等）
    m_items.clear();
    if (!root) return;
    layoutCompact(root);
    drawDfs(root);
    setSceneRect(itemsBoundingRect().adjusted(-40,-40,40,40));
}

```

(2) 先序、中序、后序遍历：为了便于可视化展示，首先使用对应的函数把对应的遍历顺序收集，然后利用收集到的顺序通过动画进行可视化展示，在 collectPreorder/collectInorder/collectPostorder 中分别按三种顺序规则访问结点，并且只在 ltag/rtag==Child 时才沿 left/right 下探，避免把线索当孩子；先序为“根 → 左 → 右”，中序为“左 → 根 → 右”，后序为“左 → 右 → 根”，访问到的结点依次写入序列（用于动画）与字符串（用于输出）

```
//最顶层，点击先序遍历按钮的交互逻辑
void FuncShow::on_preorder_search_clicked()
{
    abortAndReset(false);                  // ← 先停旧动画并清场景
    qDebug() << "前序遍历（普通）";
    QVector<ThreadedNode*> v; collectPreorder(root_, v);
    playTraversal(v, 450);
}
//最顶层，直接通过递归搜集遍历的顺序
void FuncShow::collectPreorder(ThreadedNode* p, QVector<ThreadedNode*>& out)
{
    if (!p) return;
    out.push_back(p);
    if (p->ltag == ThreadedNode::Child) collectPreorder(p->left, out);
    if (p->rtag == ThreadedNode::Child) collectPreorder(p->right, out);
}
```

```
//最顶层，播放遍历动画
void FuncShow::playTraversal(const QVector<ThreadedNode*>& order,
                             int intervalMs,
                             bool preserveOverlays)
{
    if (!root_) return;
    timer_->stop();
    if (preserveOverlays) {
        scene_->clearHighlightsOnly();          // 保留箭头，只还原高亮
    } else {
        scene_->renderTree(root_);              // 重画树（清覆盖物）
    }
    animOrder_ = order;
    animIdx_ = 0;
    if (ui->outputBox) ui->outputBox->clear();
    timer_->start(intervalMs);
}
```

(3) 先序、中序、后序线索化：线索化由“数据层+可视化层”协同完成：数据层用 ThreadedNode 维护 left/right 和成对标记 ltag/rtag（Child=孩子，Thread=线索），BinaryTree 在每次线索化前先清除旧线索，访问结点 p 时：若 p->left 为空则将其指向前驱 pre 并置左标记为线索；若 pre 存在且其右指针为空则把 pre->right 指向 p 并置右标记为线索；最后 pre=p。可视化层中，FuncShow 触发线索化与线索遍历（若未线索化会自动兜底先线索化），随后遍历整树把 ltag/rtag==Thread 的指针转成“覆盖物”箭头交给 TreeScene 绘制；TreeScene 负责紧凑布局与父子连线，NodeItem::anchorTowards 用圆周锚点绘制箭头避免穿过节点，动画由 QTimer 逐点高亮呈现，从而把逻辑层的“前驱/后继”直观地显示为可交互的线索箭头。

```
//数据层，线索二叉树的“节点”——仅封装节点本身的状态与常用操作。不负责节点的生命周期管理（释放由树统一处理）。
class ThreadedNode  //部分
{
public:
    // 指针语义：Child 表示真实孩子；Thread 表示线索指向前驱/后继
    enum Tag : quint8 { Child = 0, Thread = 1 };
    // —— 数据域（按需替换为 QString / QVariant / 模板）——
    int value = 0;
    // —— 指针域 ——（parent 非必需，但用于可视化/删除判断很方便）
    ThreadedNode* left   = nullptr;
    ThreadedNode* right  = nullptr;
    ThreadedNode* parent = nullptr;
    // —— 线索标记 ——（与 left/right 配对）
    Tag ltag = Child;
    Tag rtag = Child;
    // —— 可视化相关——
    QGraphicsItem* gfx = nullptr;   // 绑定到场景中的图元
    QPointF        posHint;         // 预布局坐标（可选）
    bool           selected = false;
};
#endif // THREADEDNODE_H
//数据层，与线索化相关的代码
// 线索化的统一原则：在“访问到当前结点 p”时：
```

```
//  1) 如果 p->left 为空，则让 left 指向“本遍历序的前驱 pre”，并置 ltag=Thread;
//  2) 如果 pre 不为空且 pre->right 为空，则让 pre->right 指向“本遍历序的后继 p”，并置 pre->rtag=Thread;
//  3) 最后 pre = p;
// 遍历序（先/中/后）不同，得到的前驱/后继关系不同，这正是“线索化”的本质。
void BinaryTree::clearThreadsRec(ThreadedNode* p)
{
    if (!p) return;
    p->clearThreads(); // 只清本节点的线索标记与指针
    if (p->ltag == ThreadedNode::Child) clearThreadsRec(p->left);
    if (p->rtag == ThreadedNode::Child) clearThreadsRec(p->right);
}
void BinaryTree::preorderThreadRec(ThreadedNode* p, ThreadedNode*& pre)
{
    if (!p) return;
    // —— 访问 p（先序）——
    if (!p->left)  p->setLeftThread(pre);
    if (pre && !pre->right) pre->setRightThread(p);
    pre = p;
    if (p->ltag == ThreadedNode::Child) preorderThreadRec(p->left, pre);
    if (p->rtag == ThreadedNode::Child) preorderThreadRec(p->right, pre);
}
//最顶层，点击先序线索化按钮的交互逻辑
void FuncShow::on_preclue_Button_clicked()
{
    abortAndReset(false);                 // ← 先停旧动画并清场景
    qDebug() << "先序线索化";
    makePreorderThread_();
    scene_->renderTree(root_);
    //绘制带线索二叉树
    drawThreadsFromTree_(root_, QColor("#F39C12"), QColor("#1F80FF")); // 橙=前驱，蓝=后继
}
void FuncShow::makePreorderThread_()
{
    if (!root_) return;
    ThreadedNode::ClearAllThreads(root_);
    ThreadedNode* pre = nullptr;
    preorderThreadRec_(root_, pre);
    threadOrder_ = ThreadOrder::Pre;
}
// —— 三种递归：在“访问当前结点 p”时补前驱/后继 ——
// 通用规则：
//   if (!p->left)  p->setLeftThread(pre);
//   if (pre && !pre->right) pre->setRightThread(p);
//   pre = p;
void FuncShow::preorderThreadRec_(ThreadedNode* p, ThreadedNode*& pre)
{
    if (!p) return;
    // 访问 p（先序）
    if (!p->left)  p->setLeftThread(pre);
    if (pre && !pre->right) pre->setRightThread(p);
    pre = p;
    if (p->ltag == ThreadedNode::Child) preorderThreadRec_(p->left,  pre);
    if (p->rtag == ThreadedNode::Child) preorderThreadRec_(p->right, pre);
}
```

（4）先序、中序线索化遍历线索遍历（非递归，用于演示先/中序）：中序从 firstInorder() 找到最

左端开始，反复用 inorderSuccessor() 沿“线索或右子树最左”前进；先序则优先走左孩子，走不了就沿右指针推进。

```
void FuncShow::on_preclue_search_clicked()
{
    abortAndReset(false);                  //  先停旧动画并清场景
    // 需要先处于“先序线索化”状态；若不是则自动线索化
    if (threadOrder_ != ThreadOrder::Pre) makePreorderThread_();
    scene_->renderTree(root_);
    drawThreadsFromTree_(root_, QColor("#F39C12"), QColor("#1F80FF"));
    QVector<ThreadedNode*> v; collectPreorderThreaded(root_, v);
    playTraversal(v, 450, /*preserveOverlays=*/true);
}
```

1.6 开发平台

- 内存：16GB
- 操作系统：Windows11
- CPU：Intel Core i7-13650HX CPU
- 开发语言：C++ (C++ 11 标准及以上)
- 开发框架：Qt 6.9.1
- 集成开发环境：Qt Creator 17.0.0 (Community)
- 运行环境：Debug 版本和 Release 版本使用 QCreator 集成开发环境可以正常编译运行，使用 Qt 6.9.1 的 windeployqt 工具打包后的可执行文件基本可在 windows 环境的机型下正常运行。

1.7 系统的运行结果分析说明

1.7.1 调试与开发过程

调试项目时，我主要用 Qt Creator 自带的 gdb：在关键函数调用阶段（如 buildFullRec、removeLeaf、layoutAssignX/applyYByDepth）打断点、单步跟进，并在右侧“变量/表达式”窗口观察 left/right/parent、ltag/rtag 等的变化来定位问题；同时配合 QDebug() 打点，快速确认线索建立与遍历序列是否一致。开发前，我先画了简要的类图/对象图，明确了页面切换与职责边界：MainWindow(QMainWindow) 作为入口，只做路由；BuildTree(QWidget) 负责建树与基本交互；FuncShow(QWidget) 负责遍历与线索化展示；数据层采用 BinaryTree 组合管理 ThreadedNode，可视化层用 TreeSceneNodeItem 映射呈现；两页通过 setRoot(tree.root()) 共享同一棵树，并以信号/槽 + show()/hide() 实现切页。实现上坚持模块化：数据层（建树/遍历/线索化/删除）与可视化层（布局/绘制/覆盖物）解耦，页面层只做 orchestrate；模块间用少量清晰接口传递数据与事件。遇到细节不熟的地方，则查 Qt Assistant 文档或者借助大模型、CSDN 等快速补齐。

1.7.2 软件的成果分析

以下以三层的二叉树为示例，先证明程序的正确性。在线索化中，蓝色箭头为后继，橙色箭头为前驱。

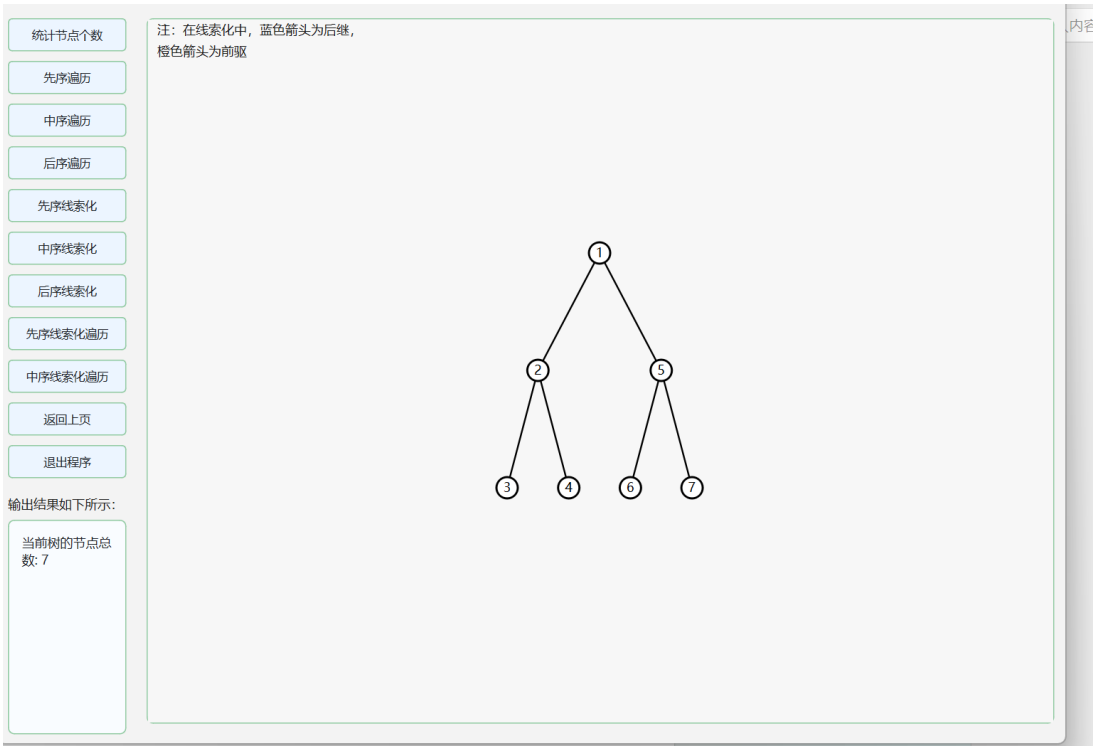

图 1.3 统计节点个数

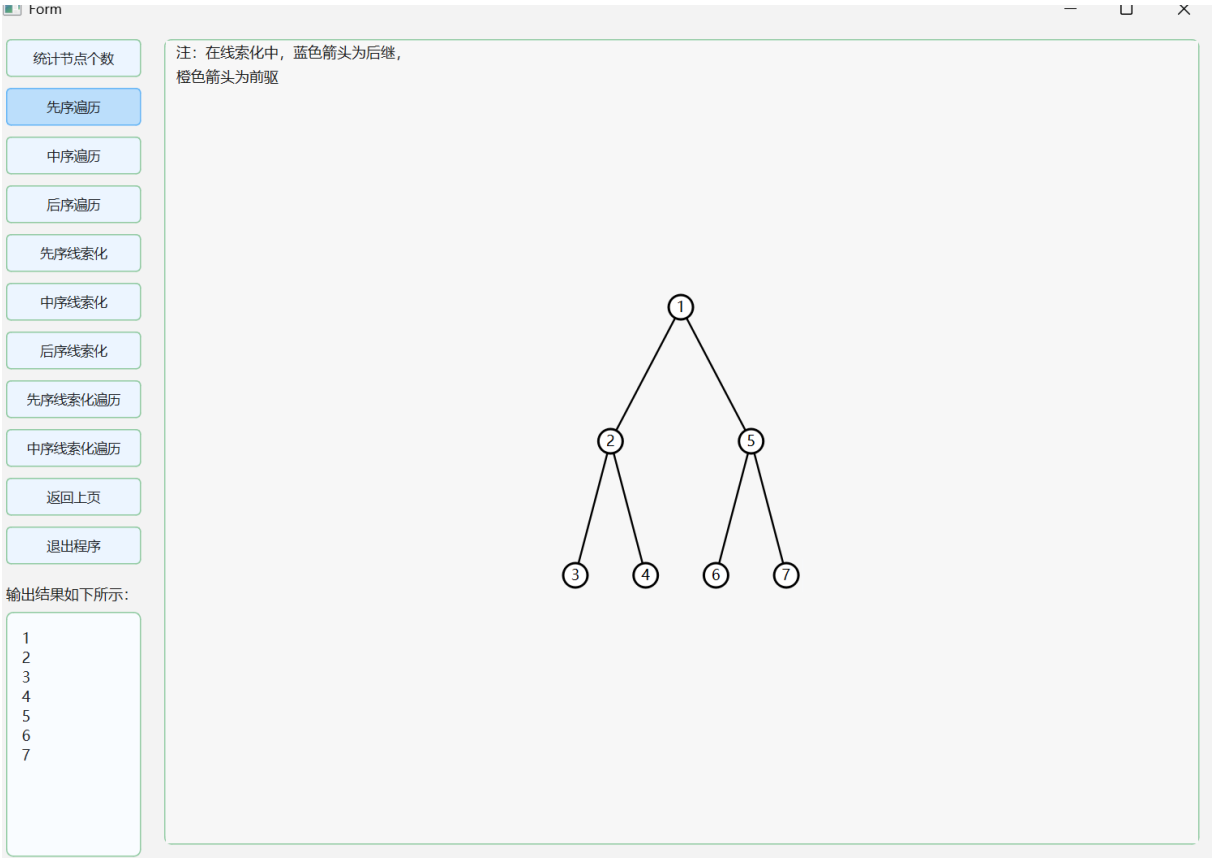

图 1.4 先序遍历结果

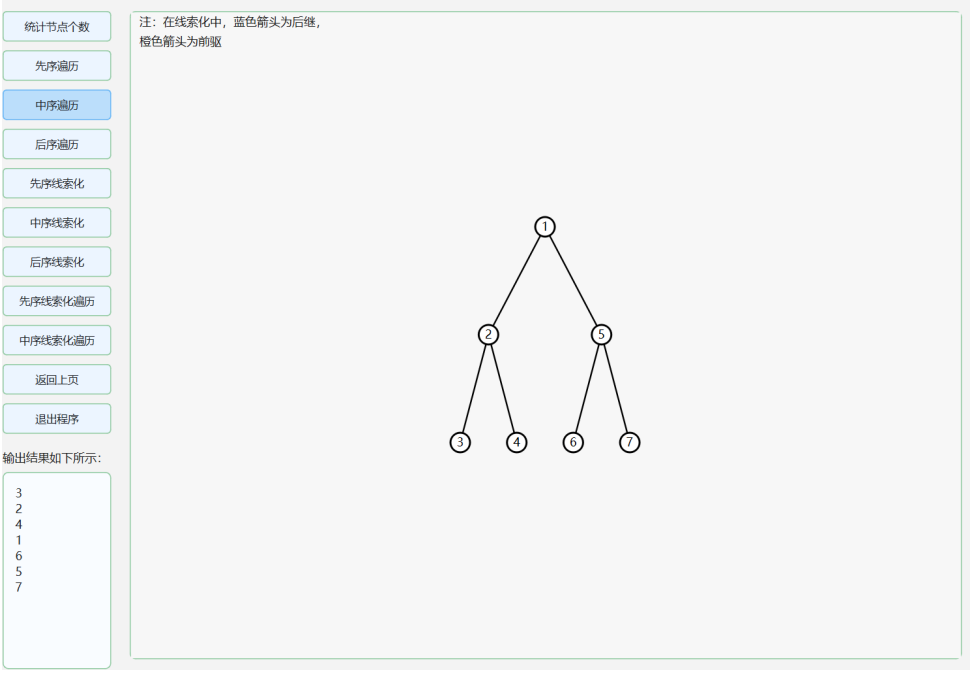

图 1.5 中序遍历结果

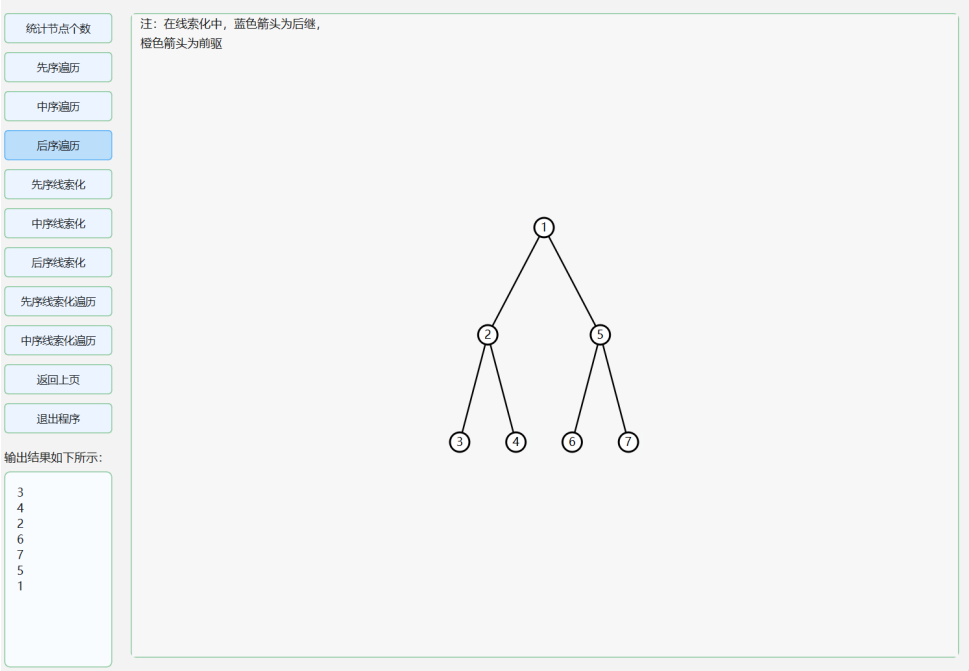

图 1.6 后序遍历结果

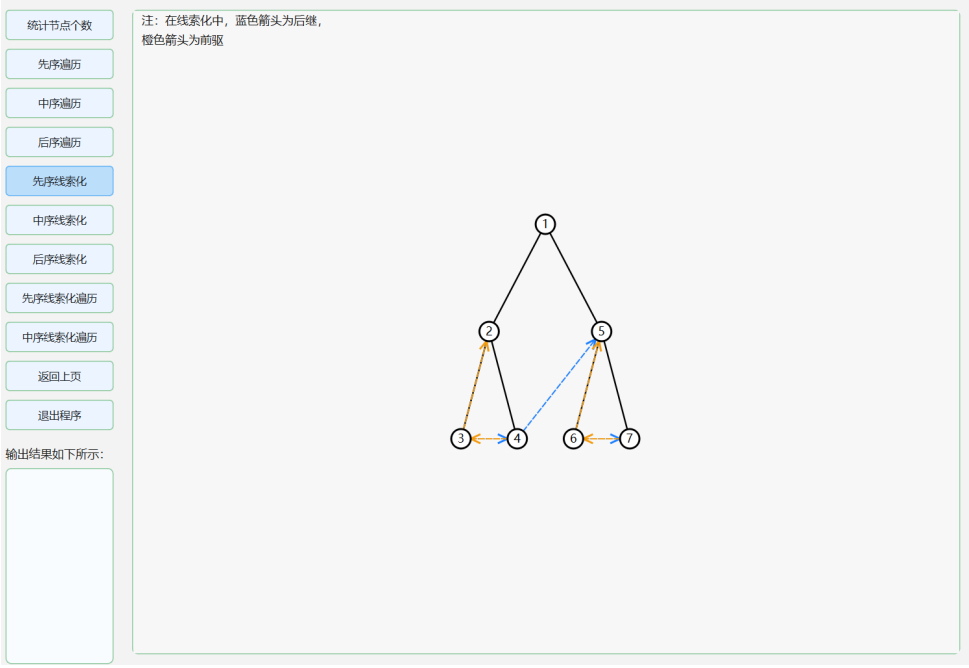

图 1.7 先序线索化

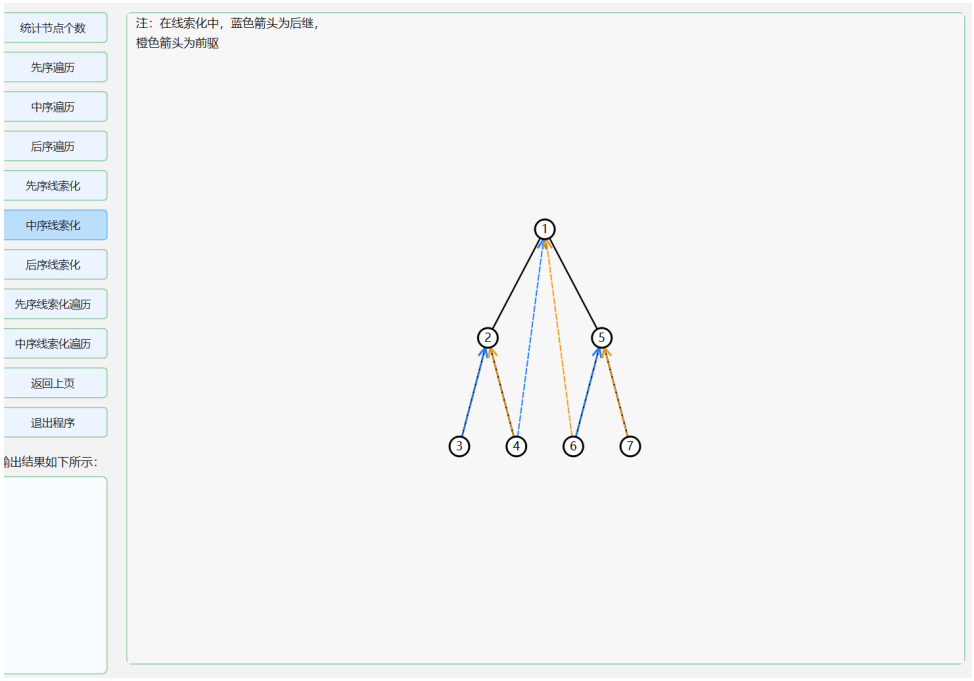

图 1.8 中序线索化

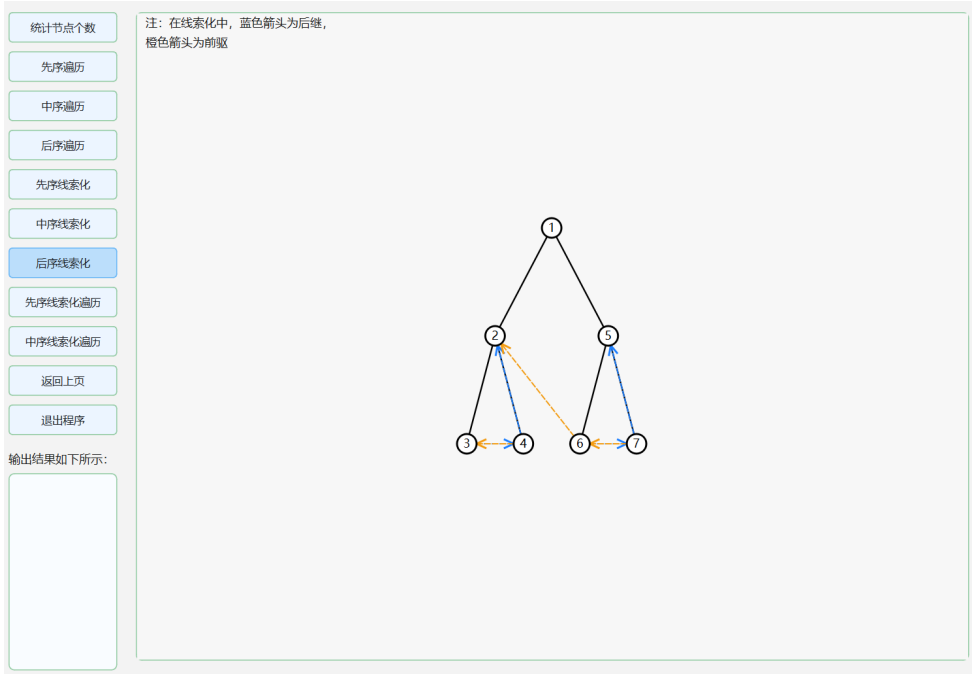

图 1.9 后序线索化

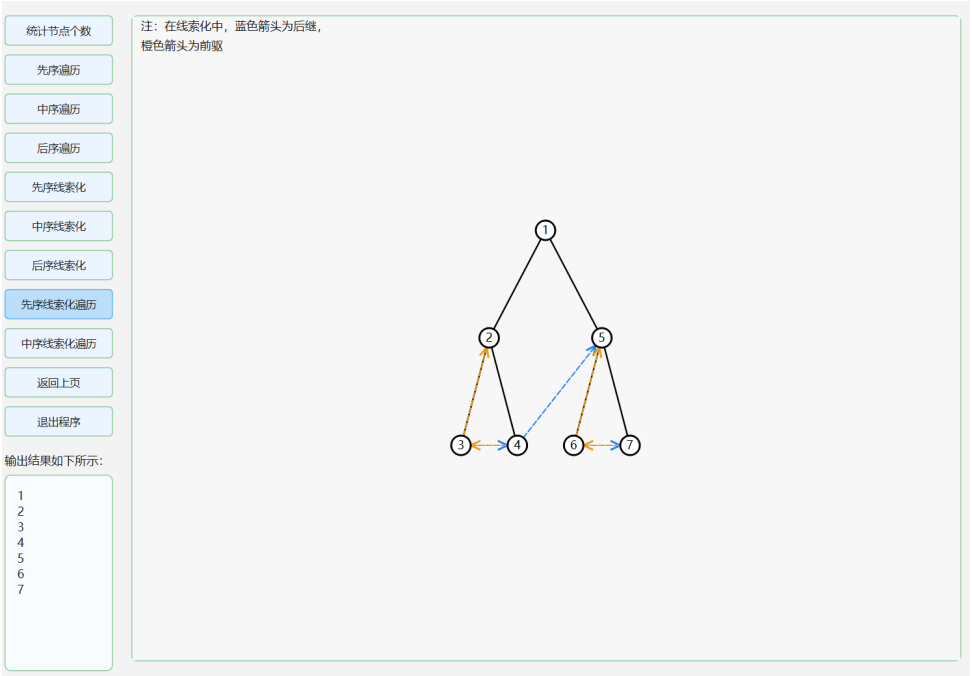

图 1.10 先序线索化遍结果

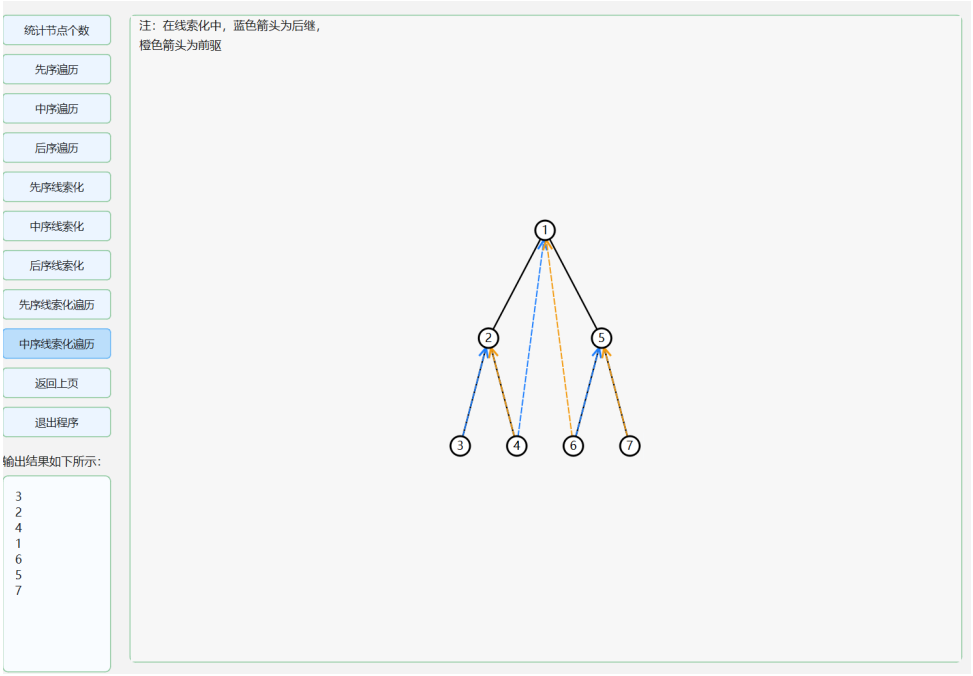

图 1.11 中序线索化遍历结果

1.7.3 可容性与鲁棒性

为了保证软件的鲁棒性与高容错性，在开发的过程中采取了以下的各类方法以保证高稳定性。

首先是任意时刻都允许“空树”的存在。进入功能展示页前传递根指针，即使没建树也把 nullptr 传给展示页。展示页始终以传入的根为准。统一渲染树的时候，入口先清空，再判空返回，因此不会对空指针做布局/绘制，避免崩溃；也能把上一轮遗留的图元/覆盖物彻底清掉。展示页设置根后立即重画，但渲染内部仍做判空保护，若为空会在 renderTree 早退并清空输出区。这样，无论是没建树就直接进入功能展示页，还是把整棵树删光再进入展示页，都只会得到“没有任何可视化效果”，而不会崩溃。

删除与点击操作的多重防护方面，为了实现对树结构的编辑，因此采用了对叶子节点进行编辑的方法。首先如果点击到空节点直接返回，代码部分：if (!n) return;（BuildTree 场景点击处理）并且只允许删除“真实叶子”（线索不算孩子）：UI 层先判断!n->isLeaf() 给出提示并返回。数据层中 removeLeaf 中先做 if (!n || !mroot) return false; if (!n->isLeaf()) return false;ThreadedNode::isLeaf() 仅在左右 tag==Child 且非空时才认作孩子，线索不计入，避免误删。删除根且为唯一节点的特殊分支需要安全置空根并释放内存。代码实现：if (!p) delete n; mroot = nullptr; return true; 最后删除后统一重画：避免 scene 内留有悬挂图元。

渲染/覆盖物/动画的可容性也十分重要，考虑到交互的时候用户对某些功能早有预料，不需要看完，因此会直接切换展示，因此需要考虑好不同功能之间的并发性处理。渲染前统一清理覆盖物与旧图元作为保险并且将覆盖物管理统一出入口（箭头与高亮一键清理）。动画播放前先终止并复位，FuncShow::abortAndReset 停止计时器、清动画状态，并按需绘制树 或仅清高亮所有遍历按钮进入前都会先调它。

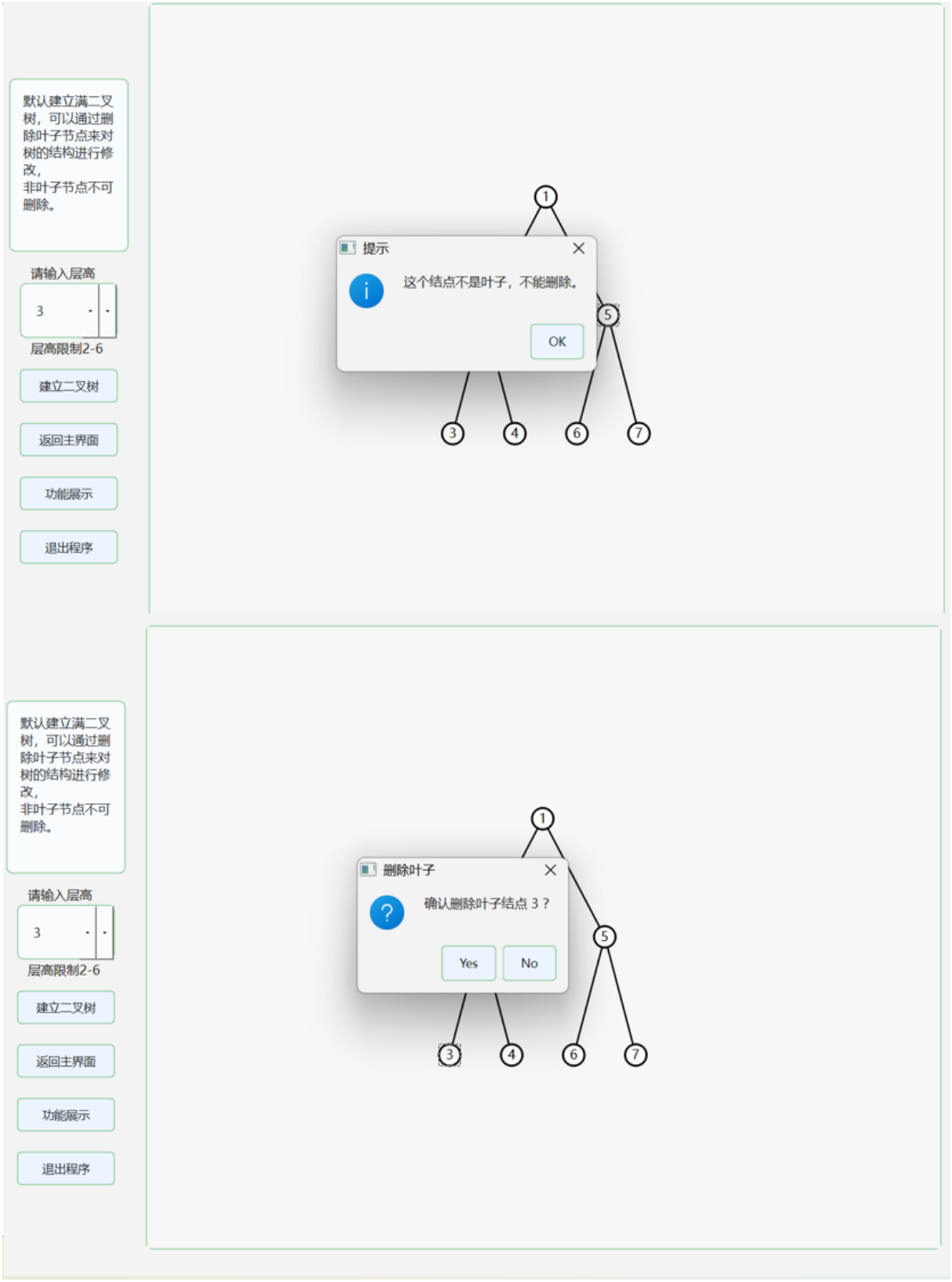

图 1.12 编辑树之前会进行提示，如果是叶子节点可以决定是否删除，如果为非叶节点会提示无法删除

1.8 操作说明

1、首先打开文件夹中的可执行文件文件夹，打开里面的二叉树文件夹，其中的 BinaryTree.exe 为可执行文件，打开即可进入程序。

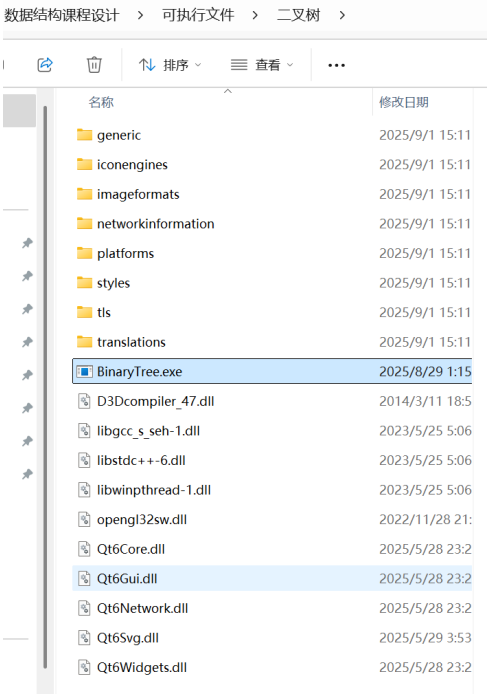

图 1.13 文件结构

2、点击进入程序后，显示如下界面，点击“创建二叉树”即可进入创建界面。

图 1.14 程序入口

3、进入创建界面之后用户可以通过界面左侧的按钮选择二叉树的层数 (2-6), 右侧画板中会默认建立对应层数的满二叉树。点击“建立二叉树”按钮即可建立对应的满二叉树。除此之外，用户可以通过点击右侧画板上的对应结点对相应结点执行删除操作。程序只允许对叶子结点执行删除操作，否则会弹出相应警告对话框。二叉树创建完成之后，点击左侧面板的”功能展示”按钮即可进入到功能演示界面。同时，用户也可以通过点击“返回主界面”按钮返回起始页。

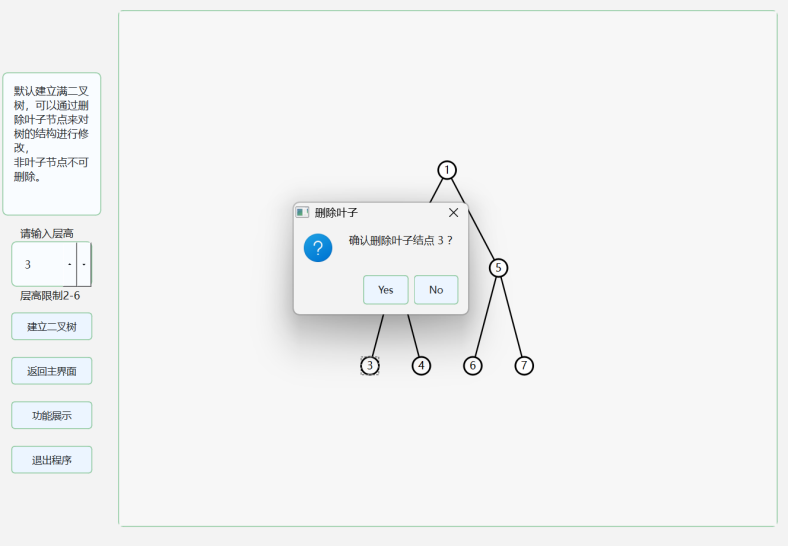

图 1.15 建树界面

4、进入到功能演示界面之后，用户可以通过点击左侧面板上不同功能的按钮实现不同功能的演示效果。用户也可以通过点击上方的“返回上页”返回创建二叉树的界面。此处实现了冲突的并行化，不同功能演示的时候可以随时切换，使得程序更加高效。

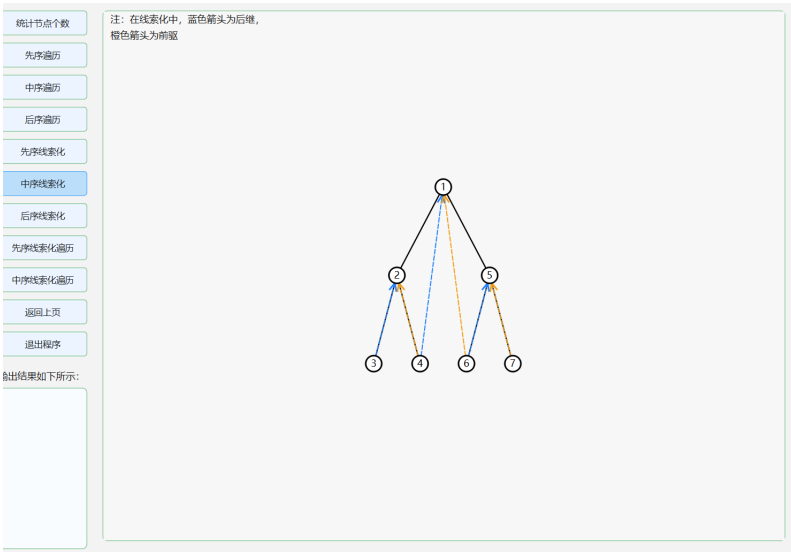

图 1.16 功能演示界面

2 综合应用题目设计——社会关系网络

2.1 题目内容

按照相应选题规则，我选了综合应用题的第 5 题《社会关系网络》，难度为 3 颗星，题目描述如下：在某社会关系网络系统中，一个人属性包括所在地区，就读的各级学校、工作单位等，每个人有一众好友，并可以根据个人兴趣及社会活动加入到某些群组。现需设计一算法，从该社会关系网络中某一人出发，寻找其可能认识的人。例如根据两个人共同好友的数量及所在群组的情况，来发现可能认识的人；通过就读学校发现可能认识的同学。

- 通过图形化界面，显示某一人的关系网络。
- 寻找某一人可能认识的人（不是其好友），并查看这些人与其关联度（共同好友数）。
- 根据可能认识的关联度对这些人进行排序。

2.2 软件功能

具体实现的所有功能如下：

2.2.1 添加新成员

在演示界面可以通过点击“添加新成员”按钮来添加新成员，点击之后弹出窗口，可以对个人信息进行编辑，并且个人信息中名字为必填项，如果未填则会弹出提示。构建过程中还可以添加群组信息以及与已有成员的朋友关系。添加之后新节点会随机出现在一个位置，此外节点可以被拖动改变位置。信息的修改是同步的，新增加的群组会随后出现在群组选项中。单击节点可以显示该节点的详细信息，可能认识的人通过改变颜色直接现实在网络中

2.2.2 编辑成员信息

双击成员节点可以对当前成员的关系以及信息进行编辑，也可以删除该成员，这里没对群组进行约束。只从成员层面对群组等信息进行修改。修改的信息也是同步的，可以保证信息的同步性。群组的总数量固定位 11 个，除了学历等信息外，还有额外的 5 个群组，供用户自行决定，在提交的程序中已经包含了若干信息，使用者可以自行继续修改

2.2.3 限制条件

添加新成员的过程中，名字是必选项，如果没输入姓名的话点击 OK 确认键的时候会出现提示。其余各信息包括成员关系等均未做限制，保证了程序的灵活性。

2.3 设计思想

本项目旨在构建一个具有可视化交互功能的社会关系网络系统，采用模块化设计理念，结合数据模型与图形界面展示，全面实现社会关系的建模、管理、分析与展示。项目基于 Qt 框架开发，综合运用了面向对象设计方法，扩展性强。

2.3.1 界面切换

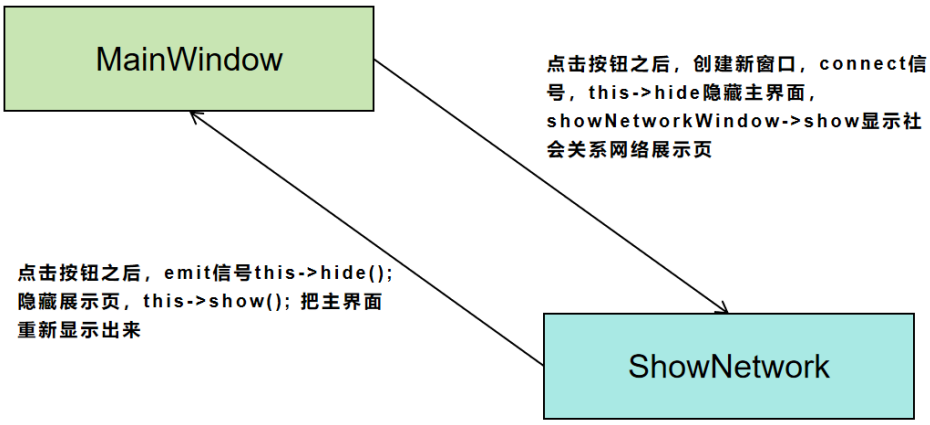

图 2.1 界面切换逻辑示意图

界面切换主要通过在主界面类 MainWindow 中控制不同窗口组件的显示与隐藏来实现。当程序启动时，首先显示 MainWindow 界面，用户点击“进入演示”按钮后，会通过信号与槽机制触发对应的槽函数，在该函数中实例化 ShowNetwork 界面并将其显示，同时隐藏或关闭主界面；在 ShowNetwork 界面内部，又通过点击“返回主界面”按钮发出自定义的 returnToMain 信号，该信号连接到主窗口中相应的槽函数，从而重新显示 MainWindow 并关闭当前视图；类似地，在需要添加或编辑成员时，ShowNetwork 会创建 AddMemberDialog 或 EditMemberDialog 的实例，并以对话框形式弹出，等待用户操作完成后再返回原界面。

2.3.2 各个类之间的继承关系

MainWindow 继承自 QMainWindow，作为程序的主界面窗口，负责导航与启动主功能模块；ShowNetwor 继承自 QWidget，是系统的核心界面类，用于展示整个社会关系网络视图；用于添加和编辑成员信息的 AddMemberDialog 和 EditMemberDialog 均继承自 QDialog，负责弹出表单获取用户输入；图中用于表示成员节点的 NodeItem 继承自 QGraphicsObject，支持图形绘制、事件响应和信号发送；连接节点的 EdgeItem 则继承自 QGraphicsLineItem，用于绘制社交关系边；系统数据核心 SocialGraph 类继承自 QObject，负责管理所有成员、组织、朋友关系以及数据的保存与推荐逻辑；而成员和组织本身分别由无继承结构体 Person 和 Group 表示，作为 SocialGraph 的组成数据单元。整个

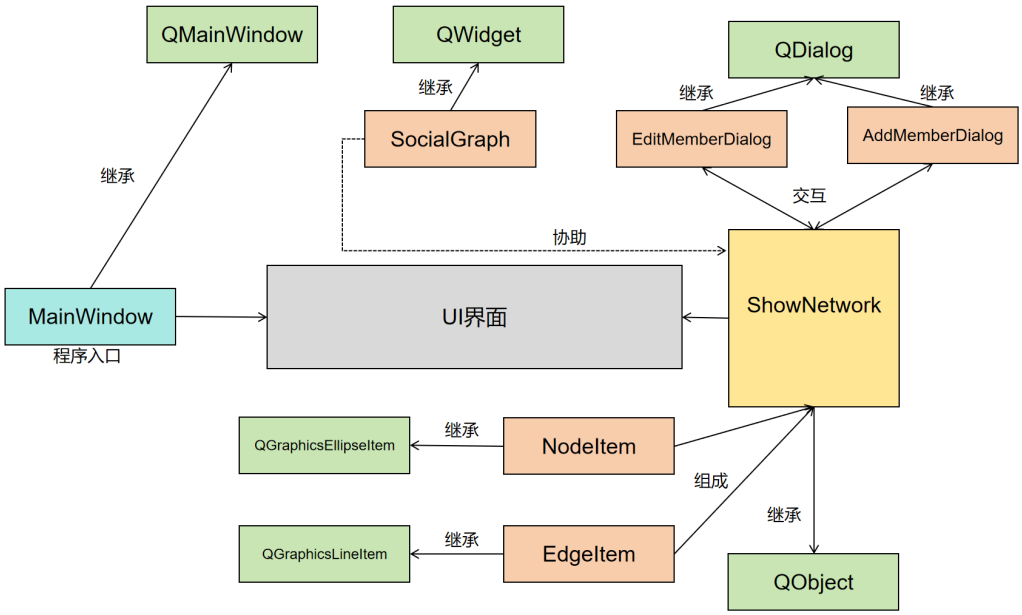

图 2.2 各个类之间的继承关系示意图

项目通过不同类之间的配合实现了可视化与数据逻辑的分离与协同。

2.4 逻辑结构与物理结构

本程序的核心部分在于用无向图建模和展示社会关系网络。

逻辑结构：社会关系网络被抽象为一个无向图 G(V,E)，其中 V 表示成员顶点的集合，每个顶点对应一位成员，E 表示边的集合，每条无向边对应两位成员之间的朋友关系。由于人与人之间的联系没有方向性，因此所有边均为无向边，从而整个社会关系网络构成了一个无向图。

物理结构：在本程序中，为了高效处理成员之间的关系并便于动态更新，采用邻接表的方式存储无向图。每个成员在邻接表中维护一个好友集合，既能快速查找某个成员的所有朋友，又能灵活地进行添加和删除操作。同时，程序还维护群组索引和节点坐标等辅助信息，群组索引便于按类别管理和查询成员，节点坐标则用于持久化图形界面布局。邻接表结合哈希表的存储方式属于链式存储结构，兼具空间利用率高和查找效率好的特点，能够很好地支持本项目中规模适中但联系紧密的社会网络数据。

2.4.1 数据层

（1）Person、GroupType、Group、Suggestion——轻量级基础数据单元

Person 是一个轻量值类型，包含唯一 id、name、六类固定归属字段（地区/小学/中学/高中/大学/工作单位）与最多 5 个“其余群组”自定义字段 custom[5]；并维护其已加入组织的 groups 集合；相等性由 id 判定。该类型不依赖 UI，适合作为持久化与算法输入。

GroupType 枚举覆盖 6 个固定类别+1 个“兴趣”+1 个“地区”+5 个“其余群组”类别 Custom1 Custom5，用于抽象分组维度（同一类别下成员在 UI 中通常保持单选）。

Group 记录组织 id、name、type 与可选 desc 备注，用于表达同一类别下的命名分组（如某所大学的具体名称）。

Suggestion 是“可能认识的人”结果条目：包含候选 person、共同好友数 commonFriends、共同组织数 commonGroups 与综合 score 用于排序展示。

```
// socialgraph.h
struct Person {
    PersonId id = 0;
    QString  name;
    // 固定 6 类
    QString  region, primarySchool, middleSchool, highSchool, university, company;
    // 新增：最多 5 个自定义类型的“选中名称”（为空表示没选）
    QString  custom[5];
    QSet<GroupId> groups;
    bool operator==(const Person& other) const noexcept { return id == other.id; }
};
enum class GroupType : quint8 {
    PrimarySchool,
    MiddleSchool,
    HighSchool,
    University,
    Company,
    Interest,
    Region,
    Custom1, Custom2, Custom3, Custom4, Custom5  // ← 新增最多 5 个自定义
};
struct Group
{
    GroupId   id = 0;
    QString   name;
    GroupType type = GroupType::Custom1;
    QString   desc;
};
struct Suggestion
{
    PersonId person;
    int      commonFriends = 0;
    int      commonGroups  = 0;
    double   score         = 0.0;
};
```

（2）SocialGraph——核心数据中心

SocialGraph 部分采取了图+组的双索引设计，就是在数据层里同时维护两类核心索引结构：图的邻接表和组织成员的倒排索引。

第一部分是图的邻接表，即 adj，它以 PersonId 为键，映射到该成员所有好友的集合，用来表示人与人之间的无向边关系。这保证了任意时刻可以快速查询某个人的所有朋友、共同好友数量，支持边的添加与删除，构成了整个社会关系网络的基础。

第二部分是组的倒排索引，即 groupIndex，它以 GroupId 为键，映射到该组织下的所有成员集合，用来表示成员与群组的从属关系。这一结构可以高效回答“某个群组里有哪些人”，同时配合 Person.groups（正向集合）保证了成员和组织间关系的双向一致性。

之所以这样设计，是因为社会网络的数据查询既涉及人与人的直接关系（例如找共同好友），又涉及人与群组的隶属关系（例如找同一学校或公司的同学和同事）。如果只存邻接表，就难以快速找群组成员；如果只存群组索引，就难以快速遍历好友关系。双索引结合能让系统同时兼顾两类操作的效率，并且支持“可能认识的人”的推荐算法：候选人既来自好友的好友（利用邻接表），也来自同一组织的其他成员（利用组倒排，虽然本次项目没有实现，本次项目中规定可能认识的人只是朋友的朋友）。**内部存储构成如下：**（1）成员表：QHash<PersonId,Person>persons。

（2）组织表：QHash<GroupId,Group>groups。

（3）好友邻接表（无向）：QHash<PersonId,QSet<PersonId»adj。

（4）组织 → 成员倒排：QHash<GroupId,QSet<PersonId»groupIndex。

（5）节点坐标缓存（便于可视化记忆布局）：QHash<PersonId,QPointF>positions。

（6）自增 ID：nextPersonId_、nextGroupId_。

（7）自定义类别标题：customTitles_（5 个，供 UI 显示/保存）。

```

class SocialGraph : public QObject
{
    Q_OBJECT
public:
    explicit SocialGraph(QObject* parent = nullptr);

    // --- 基本增删改查 ---
    PersonId addPerson(const Person& person);          // 分配 id 后返回
    bool     updatePerson(const Person& person);       // 按 id 覆盖基础信息（不改关系）
    bool     removePerson(PersonId id);                // 同时清理关系
    GroupId  addGroup(const Group& group);
    bool     updateGroup(const Group& group);
    bool     removeGroup(GroupId id);                  // 同时移除成员关系
    // 好友（无向边）
    bool addFriendship(PersonId a, PersonId b);
    bool removeFriendship(PersonId a, PersonId b);
    // 组织成员关系
    bool addMembership(PersonId p, GroupId g);
    bool removeMembership(PersonId p, GroupId g);
    const Person* getPerson(PersonId id) const {
        auto it = persons.constFind(id);
        return it == persons.cend() ? nullptr : &it.value();   // value() 返回 const T&
    }
    const Group* getGroup(GroupId id) const {
        auto it = groups.constFind(id);
        return it == groups.cend() ? nullptr : &it.value();
    }
    // --- 查询辅助 ---
    int  mutualFriends(PersonId a, PersonId b) const;   // 共同好友数
    int  sharedGroups (PersonId a, PersonId b) const;   // 共同组织数
```

```
32      struct Suggestion
33      {
34          PersonId person;
35          int      commonFriends = 0;
36          int      commonGroups  = 0;
37          double   score         = 0.0;   // 可按权重计算
38      };
39      // 可能认识的人 (非好友且非本人), 按 score 降序; limit < 0 表示不截断
40      QVector<Suggestion> potentialAcquaintances(PersonId source,
41                                                 int limit = -1,
42                                                 double wFriends = 1.0,
43                                                 double wGroups  = 1.0) const;
44
45      // 便于 UI: 取某人全部好友
46      QSet<PersonId> friendsOf(PersonId id) const {
47          return adj.contains(id) ? adj.value(id) : QSet<PersonId>{};
48      }
49      // 便于 UI: 取某组织的全部成员
50      QSet<PersonId> membersOf(GroupId gid) const {
51          return groupIndex.contains(gid) ? groupIndex.value(gid) : QSet<PersonId>{};
52      }
53      void clear();
54      bool saveToFile(const QString& path) const;
55      bool loadFromFile(const QString& path);
56      void rebuildGroupsFromAttributes();  // 仅用 5 类字段还原组织
57      QList<PersonId> allPersons() const { return persons.keys(); }
58      // 节点位置的存取
59      void     setPosition(PersonId id, const QPointF& p) { positions[id] = p; }
60      bool     hasPosition(PersonId id) const { return positions.contains(id); }
61      QPointF  positionOf(PersonId id) const { return positions.value(id, QPointF()); }
62      // 取全量好友边 (a<b 去重)
63      QVector<QPair<PersonId,PersonId>> allFriendEdges() const;
64      // 列出某类型群组的名字 (去重、排序后)
65      // socialgraph.h  (class SocialGraph public:)
66      QStringList groupNames(GroupType t) const;                  // 列出某类型已有群组名 (去重/排
        ↪ 序)
67      GroupId     findOrCreateGroupByName(const QString& name,    // 按名+类型查找, 找不到就创建
68                                          GroupType t);
69      void setMembershipOfType(PersonId p, GroupType t, const QString& name);
70      int         customTypeCount() const { return 5; }
71      QString     customTitle(int i) const { return customTitles_[i]; }
72      void        setCustomTitle(int i, const QString& title) {
73          if (i>=0 && i<5) customTitles_[i] = title.trimmed();
74      }
75      QStringList allCustomTitles() const { return customTitles_; }
76
77  private:
78      QHash<PersonId, QPointF> positions; // 新增: 节点坐标
79      PersonId nextPersonId_ = 1;
80      GroupId  nextGroupId_  = 1;
81      QHash<PersonId, Person> persons;                    // 人节点
82      QHash<GroupId,  Group>  groups;                     // 组织
83      QHash<PersonId, QSet<PersonId>> adj;                // 邻接表 (好友)
84      QHash<GroupId,  QSet<PersonId>> groupIndex;         // 组织 -> 成员 倒排
85      bool checkPerson(PersonId id) const { return persons.contains(id); }
86      bool checkGroup (GroupId  id) const { return groups.contains(id); }
87      GroupId ensureGroup(const QString& name, GroupType type);
88      void removeGroupIfEmpty(GroupId gid);
```

```
    QStringList customTitles_ = {"", "", "", "", ""};
};
```

成员/组织的基本增删改查（部分略去）：

（1）addPerson：拷贝入表、分配新 id、初始化空邻接。

（2）updatePerson：按 id 覆盖基础字段但保留其 groups（关系不丢）。

（3）removePerson：从所有好友的邻接中移除该点、从其加入的每个组织移除并必要时删空组、清理坐标缓存、最后删人本体。

（4）addGroup：分配 id、入组表并在 groupIndex 建空集合。

（5）removeGroup：先从所有成员的 groups 移除该 id，再从 groupIndex 与 groups 删除。

（6）addFriendship(a,b)/removeFriendship(a,b)：双向增删邻接，带边界/存在性检查。

（7）addMembership(p,g)/removeMembership(p,g)：维护 Person.groups 与 groupIndex，并在删除成员关系后自动删除空组。

```
PersonId SocialGraph::addPerson(const Person& p)
{
    Person copy = p;
    copy.id = nextPersonId_++;
    persons.insert(copy.id, copy);
    adj.insert(copy.id, {});           // 初始化空邻接
    return copy.id;
}
bool SocialGraph::updatePerson(const Person& p)
{
    if (!checkPerson(p.id)) return false;
    Person kept = persons.value(p.id);
    // 保留 groups（组织关系），只覆盖基础字段
    Person updated = p;
    updated.groups = kept.groups;
    persons[p.id] = updated;
    return true;
}
bool SocialGraph::removePerson(PersonId id)
{
    if (!checkPerson(id)) return false;
    // 1) 从所有朋友那里移除这条无向边
    if (adj.contains(id)) {
        for (PersonId f : adj.value(id))
            adj[f].remove(id);
        adj.remove(id);
    }
    // 2) 从所有组织移除，并清空空组
    if (persons.contains(id)) {
        const auto gs = persons[id].groups;   // 拷贝一份，避免遍历时修改
        for (GroupId g : gs) {
            if (groupIndex.contains(g)) {
                groupIndex[g].remove(id);
                removeGroupIfEmpty(g);        //  删空组
            }
        }
```

```
    }
    // 3) 清除坐标缓存
    positions.remove(id);
    // 4) 删人本体
    persons.remove(id);
    return true;
}
GroupId SocialGraph::addGroup(const Group& g)
{
    Group copy = g;
    copy.id = nextGroupId_++;
    groups.insert(copy.id, copy);
    groupIndex.insert(copy.id, {});
    return copy.id;
}
bool SocialGraph::removeGroup(GroupId id)
{
    if (!checkGroup(id)) return false;
    // 从所有成员里删除该组织
    for (PersonId p : groupIndex.value(id))
        persons[p].groups.remove(id);
    groupIndex.remove(id);
    groups.remove(id);
    return true;
}
bool SocialGraph::addFriendship(PersonId a, PersonId b)
{
    if (a == b || !checkPerson(a) || !checkPerson(b)) return false;
    adj[a].insert(b);
    adj[b].insert(a);
    return true;
}
bool SocialGraph::addMembership(PersonId p, GroupId g)
{
    if (!checkPerson(p) || !checkGroup(g)) return false;
    persons[p].groups.insert(g);
    groupIndex[g].insert(p);
    return true;
}
```

组名/组管理部分

（1）groupNames(t)：列出某类别下已存在的组名，去重并大小写不敏感排序。

（2）findOrCreateGroupByName(name,t) 与私有 ensureGroup(name,t)：按名+类型查找，不存在则新建并建倒排。

（3）removeGroupIfEmpty(gid)：当组在倒排中无人时，删除该组及索引。

```
QStringList SocialGraph::groupNames(GroupType t) const {
    QStringList names;
    for (auto it = groups.cbegin(); it != groups.cend(); ++it) {
        if (it.value().type == t) names << it.value().name;
    }
    names.removeDuplicates();
    names.sort(Qt::CaseInsensitive);
    return names;
```

```
}
GroupId SocialGraph::findOrCreateGroupByName(const QString& name, GroupType t)
{
    const QString n = name.trimmed();
    if (n.isEmpty()) return 0;

    for (auto it = groups.begin(); it != groups.end(); ++it) {
        if (it.value().type == t &&
            it.value().name.compare(n, Qt::CaseInsensitive) == 0) {
            return it.key();
        }
    }
    return ensureGroup(n, t);
}
void SocialGraph::removeGroupIfEmpty(GroupId gid)
{
    auto it = groupIndex.find(gid);
    if (it == groupIndex.end() || it.value().isEmpty()) {
        groupIndex.remove(gid);
        groups.remove(gid);
    }
}
```

2.4.2 可视化层

（1）ShowNetwork——主可视化窗口

ShowNetwork 承载整个网络视图与信息面板，负责把 SocialGraph 的数据接收到图形场景上，并处理新增/编辑成员、查看群组、保存/加载、配色与推荐等交互逻辑。

UI 初始化与数据加载：创建 QGraphicsScene、把场景挂到 graphicsView；设置右侧信息框属性；确定数据文件 social_network.json 的路径；启动时尝试 loadFromFile 并在空数据时初始化两个人与一条边；应用退出时自动保存。

```
ShowNetwork::ShowNetwork(QWidget *parent)
    : QWidget(parent), ui(new Ui::ShowNetwork)
{
    ui->setupUi(this);
    ui->infoBox->setReadOnly(true);
    ui->infoBox->setWordWrapMode(QTextOption::WrapAtWordBoundaryOrAnywhere);
    ui->infoBox->setPlaceholderText(u8"点击图中的节点查看详细信息…");
    scene_ = new QGraphicsScene(this);
    scene_->setSceneRect(-600, -400, 1200, 800);
    ui->graphicsView->setScene(scene_);
    ui->graphicsView->setRenderHint(QPainter::Antialiasing, true);
    connect(scene_, &QGraphicsScene::selectionChanged,
            this, &ShowNetwork::onSceneSelectionChanged);
    // 数据文件路径
    dataPath_ = QDir(QCoreApplication::applicationDirPath()).filePath("social_network.json");
    // 启动时加载
    graph_.loadFromFile(dataPath_);
    // 1) 读取文件
    bool loaded = graph_.loadFromFile(dataPath_);
```

```
    // 2) 若文件不存在或读到的人为空 —— 初始化两个人并保存
    if (!loaded || graph_.allPersons().isEmpty()) {
        Person p1; p1.name = "SYM"; p1.primarySchool = "省二"; p1.region = "吉林";
        Person p2; p2.name = "WJC"; p2.university    = "天津大学";  p2.company = "百度";
        PersonId id1 = graph_.addPerson(p1);
        PersonId id2 = graph_.addPerson(p2);
        graph_.addFriendship(id1, id2);
        graph_.rebuildGroupsFromAttributes();
        graph_.saveToFile(dataPath_);
    }
    // 3) 选择默认中心并绘图
    auto ids = graph_.allPersons();
    if (!ids.isEmpty()) {
        current_ = ids.first();   // 选第一个为中心
        showFullNetwork();
    } else {
        scene_->clear();          // 理论上不会到这里
    }
    // 退出时保存
    connect(qApp, &QCoreApplication::aboutToQuit,
            this, &ShowNetwork::saveToDisk);
}
```

全网绘制（showFullNetwork）：清场并重建节点与边。节点位置优先使用 SocialGraph 中持久化的 positions；若没有则随机放置并写回；每个 NodeItem 连接点击与位置变化信号：点击切换 current_ 并刷新配色，拖动后写回坐标并立即保存；边集由 allFriendEdges() 一次性生成。

```
void ShowNetwork::showFullNetwork()
{
    scene_->clear();
    nodeMap_.clear();
    edgeItems_.clear();

    const QRectF world(-550, -350, 1100, 700);
    auto* rng = QRandomGenerator::global();

    auto randomFreePos = [&](int maxTry = 80)->QPointF {
        for (int k=0; k<maxTry; ++k) {
            const qreal x = world.left() + rng->bounded(world.width());
            const qreal y = world.top()  + rng->bounded(world.height());
            const QPointF p(x, y);
            bool ok = true;
            for (NodeItem* n : nodeMap_.values()) {
                if (QLineF(n->pos(), p).length() < 90) { ok = false; break; }
            }
            if (ok) return p;
        }
        return QPointF(0,0);
    };

    // 1) 节点
    for (PersonId id : graph_.allPersons()) {
        const Person* per = graph_.getPerson(id);
        if (!per) continue;
        QPointF pos = graph_.hasPosition(id) ? graph_.positionOf(id)
```

```
                                            : randomFreePos();
        if (!graph_.hasPosition(id)) graph_.setPosition(id, pos);
        NodeItem::Role role = (id==current_) ? NodeItem::Role::Current
                                             : NodeItem::Role::Other;

        auto* n = new NodeItem(id, per->name, role);
        connect(n, &NodeItem::editRequested, this, &ShowNetwork::editMember);
        n->setPos(pos);
        scene_->addItem(n);
        connect(n, &NodeItem::clicked, this, [=](PersonId pid){
            current_ = pid;
            refreshColorsAndInfo();
        });
        connect(n, &QGraphicsObject::xChanged, this, [=]{
            graph_.setPosition(id, n->pos());
            graph_.saveToFile(dataPath_);
        });
        connect(n, &QGraphicsObject::yChanged, this, [=]{
            graph_.setPosition(id, n->pos());
            graph_.saveToFile(dataPath_);
        });
        nodeMap_.insert(id, n);
    }
    // 2) 边（全网、无向去重）
    const auto edges = graph_.allFriendEdges();
    for (const auto& e : edges) {
        const PersonId a = e.first;
        const PersonId b = e.second;
        auto* na = nodeMap_.value(a, nullptr);
        auto* nb = nodeMap_.value(b, nullptr);
        if (na && nb) {
            auto* edge = new EdgeItem(na, nb);
            scene_->addItem(edge);
            edgeItems_.push_back(edge);
        }
    }
    // 3) 初次进入也刷新一次颜色与说明
    refreshColorsAndInfo();
}
```

新增成员：弹出 AddMemberDialog 收集表单；addPerson 后按勾选结果批量 addFriendship；对六类固定字段与自定义 5 类依次按名找组或创建并 addMembership；保存并以新成员为中心刷新。

```
void ShowNetwork::on_add_new_member_Button_clicked()
{
    AddMemberDialog dlg(graph_, this);
    if (dlg.exec() != QDialog::Accepted) return;
    // 1) 取回用户输入
    Person p = dlg.person();
    // 2) 加入图（分配 id）
    PersonId pid = graph_.addPerson(p);
    // 3) 朋友关系
    for (PersonId fid : dlg.selectedFriends()) {
        graph_.addFriendship(pid, fid);
    }
    // 4) 按 6 类字段：查找或新建群组 + 建立成员关系
```

```
    auto join = [&](const QString& name, GroupType t) {
        const QString n = name.trimmed();
        if (n.isEmpty()) return;                          // 允许不选
        GroupId gid = graph_.findOrCreateGroupByName(n, t); // 在 SocialGraph 对外提供的新封装
        if (gid) graph_.addMembership(pid, gid);
    };
    join(p.primarySchool, GroupType::PrimarySchool);
    join(p.middleSchool,  GroupType::MiddleSchool);
    join(p.highSchool,    GroupType::HighSchool);
    join(p.university,    GroupType::University);
    join(p.company,       GroupType::Company);
    join(p.region,        GroupType::Region);
    // 6 类之后，追加 5 个“其余群组”
    const QStringList customs = dlg.customGroupTexts();   // 长度 5：自定义 1~5
    if (!customs.isEmpty()) {
        if (customs.size() > 0) join(customs.at(0).trimmed(), GroupType::Custom1);
        if (customs.size() > 1) join(customs.at(1).trimmed(), GroupType::Custom2);
        if (customs.size() > 2) join(customs.at(2).trimmed(), GroupType::Custom3);
        if (customs.size() > 3) join(customs.at(3).trimmed(), GroupType::Custom4);
        if (customs.size() > 4) join(customs.at(4).trimmed(), GroupType::Custom5);
    }
    // 5）保存到 JSON
    graph_.saveToFile(dataPath_);
    // 6）以新成员为中心刷新
    current_ = pid;
    showFullNetwork();
}
```

编辑/删除成员：NodeItem 双击触发 editRequested；在 editMember 中，若删除则清理并换中心；否则用 updatePerson 与 setMembershipOfType 同步六类与 5 个自定义类别的“同类唯一”关系，然后对比勾选前后差集批量增删好友，最后保存与重绘。

```
void ShowNetwork::editMember(PersonId id)
{
    const Person* po = graph_.getPerson(id);
    if (!po) return;
    // 进入对话框前，记下“编辑前”的好友集合
    const QSet<PersonId> friendsBefore = graph_.friendsOf(id);
    EditMemberDialog dlg(graph_, id, this);
    if (dlg.exec() != QDialog::Accepted) return;
    if (dlg.deleted()) {
        // 若删除的是当前中心，换一个中心（可选）
        if (current_ == id) {
            auto ids = graph_.allPersons();
            ids.removeAll(id);
            current_ = ids.isEmpty() ? 0 : ids.first();
        }
        graph_.removePerson(id);          // 会清理好友与群组倒排
        graph_.saveToFile(dataPath_);
        showFullNetwork();
        return;
    }
    Person np = dlg.updatedPerson();
    np.id = id; // 保持 id 不变
    graph_.updatePerson(np);
```

```
    graph_.setMembershipOfType(id, GroupType::PrimarySchool, np.primarySchool);
    graph_.setMembershipOfType(id, GroupType::MiddleSchool,  np.middleSchool);
    graph_.setMembershipOfType(id, GroupType::HighSchool,    np.highSchool);
    graph_.setMembershipOfType(id, GroupType::University,    np.university);
    graph_.setMembershipOfType(id, GroupType::Company,       np.company);
    graph_.setMembershipOfType(id, GroupType::Region,        np.region);
    graph_.setMembershipOfType(id, GroupType::PrimarySchool, np.primarySchool);
    graph_.setMembershipOfType(id, GroupType::MiddleSchool,  np.middleSchool);
    graph_.setMembershipOfType(id, GroupType::HighSchool,    np.highSchool);
    graph_.setMembershipOfType(id, GroupType::University,    np.university);
    graph_.setMembershipOfType(id, GroupType::Company,       np.company);
    graph_.setMembershipOfType(id, GroupType::Region,        np.region);
    // 追加：5 个“其余群组”（每类保持单选）
    {
        const QStringList customs = dlg.customGroupTexts();  // 长度 5
        graph_.setMembershipOfType(id, GroupType::Custom1, customs.value(0));
        graph_.setMembershipOfType(id, GroupType::Custom2, customs.value(1));
        graph_.setMembershipOfType(id, GroupType::Custom3, customs.value(2));
        graph_.setMembershipOfType(id, GroupType::Custom4, customs.value(3));
        graph_.setMembershipOfType(id, GroupType::Custom5, customs.value(4));
    }
    const QSet<PersonId> friendsAfter = dlg.selectedFriends();
    // 需要添加的好友：after - before
    for (PersonId f : (friendsAfter - friendsBefore)) {
        if (f != id) graph_.addFriendship(id, f);
    }
    // 需要删除的好友：before - after
    for (PersonId f : (friendsBefore - friendsAfter)) {
        if (f != id) graph_.removeFriendship(id, f);
    }
    graph_.saveToFile(dataPath_);
    showFullNetwork();
}
```

（2）NodeItem——可视化的成员节点

NodeItem 的职责是以圆形节点的形式展示一个成员，它会根据角色枚举 RoleCurrent,Friend,Suggest,Other 在 paint() 中分别映射为红色、黄色、绿色或蓝色，并在节点中央绘制白色加粗的名字，从而直观区分不同类别的成员。当节点位置发生变化时，itemChange(ItemPositionHasChanged) 会遍历已登记的所有 EdgeItem 并调用其 adjust() 方法，使相关连线实时跟随节点移动。交互上，左键单击节点会发出 clicked(PersonId) 信号，双击则发出 editRequested(PersonId) 信号并阻止事件继续传递，保证视图能够正确响应用户操作。此外，它还提供了若干公共接口：通过 addEdge 可以登记边指针，setRole 用于更新节点的角色并重绘外观，而 id() 与 role() 则是最基本的属性访问方法。

```
class EdgeItem;
class NodeItem : public QGraphicsObject
{
    Q_OBJECT
public:
    enum class Role { Current, Friend, Suggest, Other };   // 四态
    NodeItem(PersonId id, const QString& name, Role role,
             QGraphicsItem* parent = nullptr);
```

```
    QRectF boundingRect() const override;
    void   paint(QPainter* p, const QStyleOptionGraphicsItem*, QWidget*) override;
    void addEdge(EdgeItem* e) { if (e) edges_.push_back(e); }
    void setRole(Role r) { role_ = r; update(); }
    PersonId id()  const { return id_; }
    Role     role() const { return role_; }
signals:
    void clicked(PersonId id);
    void editRequested(PersonId id);            // 双击触发
protected:
    QVariant itemChange(GraphicsItemChange change, const QVariant& value) override;
    void mousePressEvent(QGraphicsSceneMouseEvent* ev) override;
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent* ev) override;
private:
    static constexpr qreal R = 36.0;  // 半径
    PersonId id_{0};
    QString  label_;
    Role     role_{Role::Other};
    QVector<EdgeItem*> edges_;
};

```

（3）EdgeItem——可视化的朋友关系边

EdgeItem 的职责是把两个人之间的好友关系绘制成一条美观的无向线段，并能在节点移动时自动调整位置。其构造过程中会保存两端的 NodeItem*指针，并将自己登记到这两个节点的边列表中，随后立即调用 adjust() 来初始化连线，线条采用棕色且线宽为 3。几何计算上，连线是按两节点圆心连线绘制的，但为了避免线段直接穿过节点圆心造成视觉突兀，程序在两端各缩进了约 18 像素的圆边距，使连线更自然地连接到节点边缘。

```
class NodeItem;
class EdgeItem : public QGraphicsLineItem
{
public:
    EdgeItem(NodeItem* a, NodeItem* b, QGraphicsItem* parent = nullptr);
    void adjust();
private:
    NodeItem* a_{nullptr};
    NodeItem* b_{nullptr};
};
```

2.5 核心算法实现

2.5.1 数据流动与持久化

本项目通过内存容器与文件持久化相结合实现了数据的存储与更新。运行时，所有成员、组织、好友关系与节点坐标都存放在 SocialGraph 的容器中，作为界面与表单的唯一数据源；**跨运行时，系统利用 saveToFile 将这些数据写入 JSON 文件，在程序启动时再通过 loadFromFile 读回**，并用 rebuildGroupsFromAttributes 全量重建组织结构，保证字段与索引一致。在实时更新方面，每

一次新增、编辑或删除成员操作都会先修改内存中的容器，再立即调用 saveToFile 持久化，并执行 showFullNetwork 或 refreshColorsAndInfo 重绘界面，使最新数据立刻体现在网络图与信息面板中；表单下拉选项也在每次弹出时直接读取仓库数据，因此新建的群组与关系无需重启即可显示，再加上节点拖动时的位置实时写回，保证了整个系统在运行过程中始终保持数据的一致性和即时可见性。

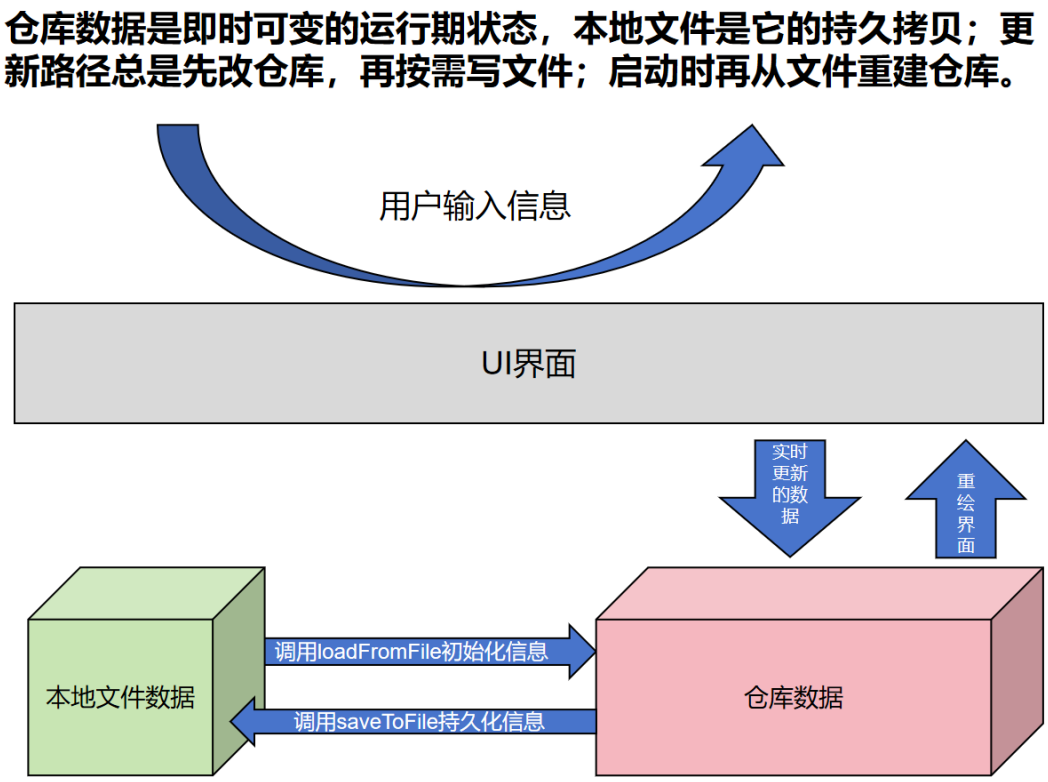

图 2.3 数据流动与持久化示意图

（1）数据的内存存储

在程序运行期间，数据的内存存储由核心仓库类 SocialGraph 负责。它内部维护了五大核心容器：成员表 persons、组织表 groups、好友邻接表 adj（用于存放无向边）、组织到成员的倒排索引 groupIndex 以及节点坐标 positions，同时还包括自增 ID 与五个自定义类别标题 customTitles_。这些容器就是界面和表单读写的真实数据源，所有展示与编辑操作都会直接访问并修改这里的数据。与之配合，Person 用于保存成员的 ID、姓名、六大固定字段与五个自定义字段，以及该成员所加入组织的集合；Group 用于保存组织的 ID、名称和类型。此外，节点的坐标通过 setPosition、hasPosition 和 positionOf 接口进行存取，支持在用户拖拽节点时实时写回位置。由此可见，UI 的所有显示与交互最终都依赖 SocialGraph，修改也会立即反映到这里，从而实现了单一真实数据源的设计。

（2）数据的持久化存储

为了在跨运行之间保持数据的一致性，系统还提供了持久化存储机制。saveToFile(path) 会将当前版本号、自定义标题、所有成员信息（包括六大固定字段、五个自定义字段和节点坐标）以及去重

后的无向好友边写入 JSON 文件；而 loadFromFile(path) 则会将这些数据重新读入内存，重置自增 ID，并基于每个成员的 6+5 文本字段调用 rebuildGroupsFromAttributes() 全量重建组织和倒排索引，以确保表单字段与组织结构始终保持一致。在程序启动时，ShowNetwork 会定位 social_network.json 并尝试加载；如果文件不存在或数据为空，则会初始化两个成员及一条好友关系，调用 rebuildGroupsFromAttributes() 建立初始组织，再立即保存文件。由此可见，文件仅仅作为落盘快照存在，而真正的运行时逻辑全部以内存数据为准，持久化的作用则是保证下次启动时能够完整还原之前的状态。

（3）编辑增删如何写回与维持一致性

图 2.4 添加新人员（YTQ）过程中，给 TA 的大学群组内容设置为南洋理工大学，随后无论是在编辑界面（孙誉铭）还是在添加新成员界面都可以找到南洋理工大学群组

在本项目中成员与关系的增删改都严格遵循**“先更新内存、再持久化、最后重绘”**的流程：新增成员时，通过 AddMemberDialog 获取信息并调用 addPerson 分配 ID，按勾选建立好友关系，再根据 6 个固定类别和 5 个自定义类别调用 findOrCreateGroupByName 与 addMembership 建立组织归属，最后保存并重绘；编辑成员时，则对比前后好友集合差集，增量调用 addFriendship 或 removeFriendship，并用 updatePerson 与 setMembershipOfType 保持表单字段与组织关系的一致，必要时自动清理空组；删除成员时则通过 removePerson 清除其在邻接表和组织索引中的全部引用。无论哪种情况，所有修改都会立刻写回 SocialGraph 的内存容器，UI 重绘与下次表单下拉都会直接读取最新数据，从而保证在同一次运行内即时可见。为进一步维持一致性，系统提供了两类机制：rebuildGroupsFromAttributes() 用于全量重建，在加载或需要纠偏时通过成员的 6+5 字段一次性对齐组织结构；setMembershipOfType() 则用于局部更新，在单个成员的同类单选场景下移除旧组、创建新组并同步字段，两者互为补充，确保了数据在全局与局部两方面始终保持统一。

新增成员 on_add_new_member_Button_clicked() 的逻辑为：① 从 AddMemberDialog 取回 Person；②addPerson 分配 id；③ 对勾选好友逐一 addFriendship；④ 对 6 类与 5 个“其余群组”按名 findOrCreateGroupByName+addMembership 建立从属关系；⑤saveToFile；⑥ 设为中心并 showFullNetwork() 重绘全图。

编辑/删除成员 editMember()：① 对比对话框前后的“好友集合”差集，增量 addFriendship/removeFriendship；② 通过 updatePerson 覆盖基础字段、保持其 groups；③ 用 setMembershipOfType 按“同类单选”规则同步 6 类与 5 个自定义类别（内部会移除旧组、找/建新组并加入，同时更新 Person 的显示字段与倒排，空组自动清理）；④ 若选择“删除”，则 removePerson 会从所有邻接与组织中清理引用；⑤ 统一 saveToFile 并 showFullNetwork()。

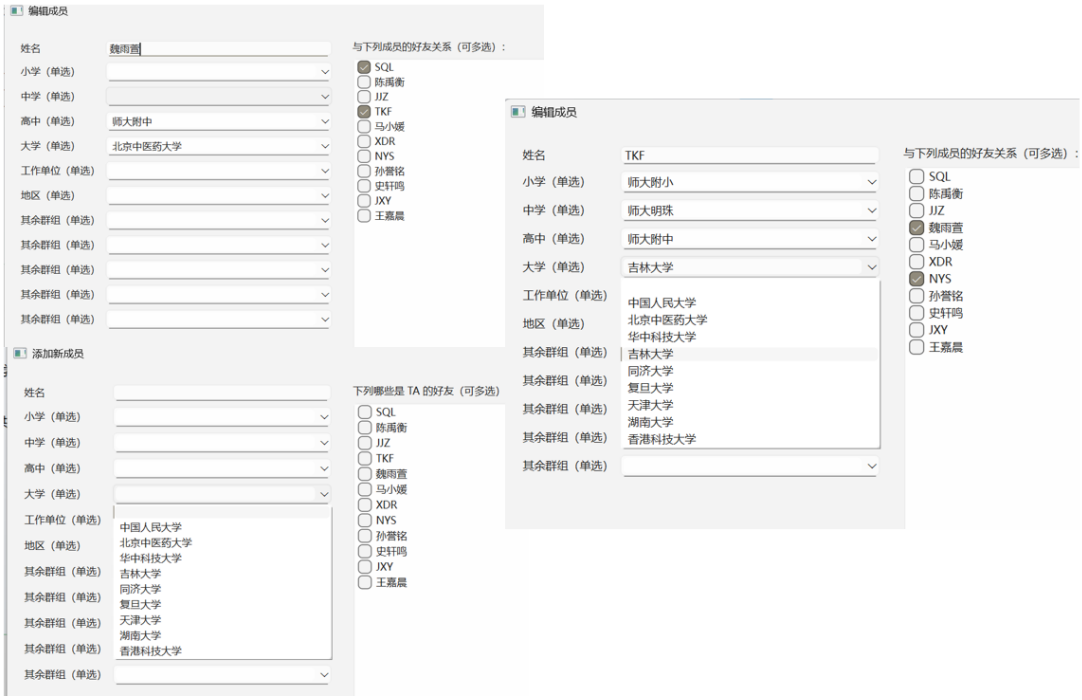

图 2.5 编辑人员过程中，给魏雨萱的大学群组内容设置为北京中医药大学，随后无论是在编辑界面（TKF）还是在添加新成员界面都可以找到北京中医药大学群组

（4）组/字段一致性的全量保障

rebuildGroupsFromAttributes() 是一种全量重建机制。程序在加载文件或需要刷新组织结构时，并不直接依赖存储的旧组织信息，而是先清空当前的 groups 与 groupIndex，然后逐个成员读取其 6 个固定类别字段和 5 个自定义类别字段（即这些纯文本属性），对每个非空值调用 ensureGroup 来创建或查找对应的组织，再用 addMembership 建立成员与组织的隶属关系。

setMembershipOfType() 则是一种单点更新机制，更贴合 UI 的交互逻辑。因为对于小学、中学、大学、公司等类别，一个成员在同类里通常只能属于一个群组，所以这个函数会在修改时，先检查并移除该成员在该类别原有的组织关系，如果用户选择了新的群组名，则通过 findOrCreateGroupByName 建立组织并调用 addMembership 加入。过程中同时会更新 Person 对应的显示字段（包括 custom[i]），

并在原有组织被清空后自动删除该空组。这样保证了同类单选的业务规则，同时维持了 groups 和 groupIndex 的一致性。

```
void SocialGraph::rebuildGroupsFromAttributes()
{
    // 清空旧组织（不动 persons/adj/positions）
    groups.clear();
    groupIndex.clear();
    nextGroupId_ = 1;
    for (auto it = persons.begin(); it != persons.end(); ++it) {
        it.value().groups.clear();
    }
    auto addIf = [&](PersonId pid, const QString& s, GroupType t){
        const QString n = s.trimmed();
        if (n.isEmpty()) return;
        GroupId gid = ensureGroup(n, t);
        addMembership(pid, gid);
    };
    for (auto it = persons.begin(); it != persons.end(); ++it) {
        const Person& p = it.value();
        const PersonId id = p.id;
        // 固定 6 类
        addIf(id, p.primarySchool, GroupType::PrimarySchool);
        addIf(id, p.middleSchool , GroupType::MiddleSchool );
        addIf(id, p.highSchool   , GroupType::HighSchool   );
        addIf(id, p.university   , GroupType::University   );
        addIf(id, p.company      , GroupType::Company      );
        addIf(id, p.region       , GroupType::Region       );
        // 自定义 5 类
        for (int i = 0; i < 5; ++i) {
            if (!p.custom[i].trimmed().isEmpty()) {
                addIf(id, p.custom[i],
                    static_cast<GroupType>(static_cast<int>(GroupType::Custom1) + i));
            }
        }
    }
}

void SocialGraph::setMembershipOfType(PersonId p, GroupType t, const QString& name)
{
    if (!checkPerson(p)) return;
    // 旧组（同类最多一个）
    GroupId oldG = 0;
    for (GroupId g : persons[p].groups) {
        if (groups.contains(g) && groups[g].type == t) { oldG = g; break; }
    }
    // 若新名称为空 -> 目标为不属于任何同类群组
    const QString trimmed = name.trimmed();
    GroupId newG = 0;
    if (!trimmed.isEmpty()) {
        newG = findOrCreateGroupByName(trimmed, t);
    }
    if (oldG == newG) {
        // 仍然要把“显示字段”刷一次（避免外部只改字符串而没换组名时不同步）
        switch (t) {
        case GroupType::PrimarySchool: persons[p].primarySchool = trimmed; break;
        case GroupType::MiddleSchool : persons[p].middleSchool  = trimmed; break;
```

```
        case GroupType::HighSchool    : persons[p].highSchool    = trimmed; break;
        case GroupType::University    : persons[p].university    = trimmed; break;
        case GroupType::Company       : persons[p].company       = trimmed; break;
        case GroupType::Region        : persons[p].region        = trimmed; break;
        default: {
            const int idx = customIndex(t);
            if (idx >= 0 && idx < 5) persons[p].custom[idx] = trimmed;
        } break;
        }
        return;
    }
    // 先移除旧组（若存在）
    if (oldG) {
        persons[p].groups.remove(oldG);
        if (groupIndex.contains(oldG)) {
            groupIndex[oldG].remove(p);
            removeGroupIfEmpty(oldG);  // 清空组
        }
    }
    // 再加入新组（若非空）
    if (newG) {
        persons[p].groups.insert(newG);
        groupIndex[newG].insert(p);
    }
    // 同步“显示字段”
    switch (t) {
    case GroupType::PrimarySchool: persons[p].primarySchool = trimmed; break;
    case GroupType::MiddleSchool : persons[p].middleSchool  = trimmed; break;
    case GroupType::HighSchool   : persons[p].highSchool    = trimmed; break;
    case GroupType::University   : persons[p].university    = trimmed; break;
    case GroupType::Company      : persons[p].company       = trimmed; break;
    case GroupType::Region       : persons[p].region        = trimmed; break;
    default: {
        const int idx = customIndex(t);
        if (idx >= 0 && idx < 5) persons[p].custom[idx] = trimmed;
    } break;
    }
}
```

（5）UI 层实时更新逻辑实现

在界面层中，系统通过多种机制确保数据的即时可见性。绘图逻辑完全依赖仓库数据，showFullNetwork() 会在每次调用时清空场景，重新遍历 graph_.allPersons() 与 allFriendEdges() 构建所有节点和边，并通过 refreshColorsAndInfo() 动态计算角色与说明，因此只要数据层发生修改，再次调用该方法即可立刻反映最新状态。所有新增、编辑或删除成员的操作在结束时都会调用 **graph_.saveToFile(dataPath_) 并执行 showFullNetwork()**，确保视图和信息面板第一时间同步更新。与此同时，AddMemberDialog 与 EditMemberDialog 的下拉框每次弹出时都会通过 fillCombo(...,graph_.groupNames(type)) 从仓库实时获取组名，并允许手动输入，因此新增的群组或归属关系会在同一次运行中立刻出现在下一个对话框中，而无需重新启动。此外，在 showFullNetwork() 中还将每个 NodeItem 的 xChanged 和 yChanged 信号绑定到 graph_.setPosition(id,n->pos()) 并即时保存，这使得用户拖拽节点位置时系统能够立即记忆布局，刷新或下次进入时依然保持。

在读取与展示层面，系统同样保证了数据的一致性。refreshColorsAndInfo() 会基于当前成员的 p->groups，通过 graph_.getGroup(gid) 聚类并按类型顺序输出群组信息，同时利用朋友的朋友关系与共同群组计数生成“可能认识的人”，这些内容完全依赖仓库的最新数据，因此在编辑后只需刷新即可看到更新结果。

群组总览功能 on_check_group_Button_clicked() 则通过 graph_.allPersons()、成员的 groups 以及 graph_.getGroup(gid) 与 membersOf(gid) 组合出群组清单，形成当前状态的快照，因此展示的群组及成员情况会随着数据层的变化而即时更新，从而实现界面与仓库之间的高度一致。

2.5.2 寻找可能认识的人

该算法以 SocialGraph 的双索引为基础（好友邻接表 adj 与组织 → 成员倒排 groupIndex），先构造候选集合为“好友的好友 同组成员”（剔除本人与已是好友），随后对每个候选计算共同好友数 cf，仅当 cf>0 时才将其视为“可能认识的人”，并在此基础上再计算共同群组数 cg 作为加分因子，形成综合分 score=wFriends*cf+wGroups*cg（其中“关联度”定义即为 cf）；最后按 score 降序、再按 cf 与 cg 降序稳定排序，必要时截断取前 K 个用于展示。这样既保证了“必须有共同好友”这一判定约束，又兼顾了同群组带来的属性相似度，以统一且高效的方式给出可能认识的人。

设源点为 s，其度为 $d = \deg(s)$；令 $S = \sum_{f \in N(s)} \deg(f)$ 表示“好友的好友”展开时访问的邻接规模，总计扫描量为 S；令 $g = |\text{groups}(s)|$ 为 s 所在群组的个数，且第 i 个群组规模为 $|M_i|$，则“同组成员”分支的扫描量为 $G = \sum_{i=1}^{g} |M_i|$；记并集去重后的候选数为 C。候选构建阶段时间为 $O(S+G)$（基于哈希集合插入/查重的均摊 $O(1)$）；对每个候选 c 的过滤与计数中，`mutualFriends`(s,c) 在代码实现里固定遍历 $N(s)$ 并哈希查询 $N(c)$，故为 $O(d)$，而 `sharedGroups`(s,c) 固定遍历 groups(s) 并查询 c 的群组集合，故为 $O(g)$，于是该阶段合计为 $O\big(C(d+g)\big)$；最终按 `score` → cf → cg 的排序代价为 $O(C \log C)$。综上，总时间复杂度为 $O\big(S+G+C(d+g)+C\log C\big)$，空间复杂度为 $O(C)$。

（1）数据层主算法：SocialGraph::potentialAcquaintances(...)

目标：给定 source，返回一组候选人及其特征（共同好友 cf、共同群组 cg、综合分 score）。

核心数据结构:

adj：无向图邻接表，adj[u] 是 u 的好友集合。

groupIndex：组织 → 成员倒排，按组直接取其所有成员。

persons[p].groups：某人的所有组织集合（与 groupIndex 对应）。

算法先用双索引构造候选集合：把当前人的“好友的好友”和“同组成员”（均排除本人及已是好友）取并集，获得既体现结构相近又体现属性相似的候选；随后对每个候选先计算共同好友数 cf，**若 cf=0 直接舍弃**（必须有共同好友才算候选），仅当 cf>0 时再计算共同群组数 cg，并按‘score=wFriends*cf+wGroups*cg‘合成评分（权重默认 1,1）；最后按分数降序输出需要的前若干名。

按 score 降序排列；若分数相同，按 commonFriends 降序；再相同按 commonGroups 降序。若传入 limit>=0，则截断前 limit 项返回。结果类型为 Suggestionperson,commonFriends=cf,common-

Groups=cg,score。算法源码如下：

```
QVector<SocialGraph::Suggestion>
SocialGraph::potentialAcquaintances(PersonId source, int limit,
                                    double wFriends, double wGroups) const
{
    QVector<Suggestion> out;
    if (!checkPerson(source)) return out;
    const auto& friends = adj.value(source);
    // 候选：好友的好友 + 同组织成员（集合并集）
    QSet<PersonId> candidates;
    for (PersonId f : friends) {
        for (PersonId fof : adj.value(f)) {
            if (fof != source && !friends.contains(fof))
                candidates.insert(fof);
        }
    }
    for (GroupId g : persons.value(source).groups) {
        for (PersonId m : groupIndex.value(g)) {
            if (m != source && !friends.contains(m))
                candidates.insert(m);
        }
    }
    out.reserve(candidates.size());
    for (PersonId c : candidates) {
        int cf = mutualFriends(source, c);
        //  没有共同好友则直接忽略（“同组不算数”）
        if (cf <= 0) continue;
        // 只有在 cf>0 时才计算/计入共同群组
        int cg = sharedGroups(source, c);
        double score = wFriends * cf + wGroups * cg;
        out.push_back(Suggestion{c, cf, cg, score});
    }
    // 排序：score 降序 → commonFriends 降序 → commonGroups 降序
    std::sort(out.begin(), out.end(),
              [](const Suggestion& a, const Suggestion& b){
                  if (a.score != b.score) return a.score > b.score;
                  if (a.commonFriends != b.commonFriends) return a.commonFriends >
                  ↪  b.commonFriends;
                  return a.commonGroups > b.commonGroups;
              });
    if (limit >= 0 && out.size() > limit)
        out.resize(limit);
    return out;
}
```

（2）界面层使用：ShowNetwork::refreshColorsAndInfo()

首先获取推荐推荐，recs = graph_.potentialAcquaintances(current_, -1, 1.0, 1.0) 得到的 recs 已经满足“cf>0 才出现”，并按新规则排好序。上色不变，当前节点 → 红，好友 → 黄；recs 中出现的人构成“可能认识的人集”（Suggest）→ 绿；其他 → 蓝。因为“可能认识”集合来自 recs，自然与 infoBox 保持一致：没有共同好友的人不会涂绿。

输出到 infoBox：逐条输出 姓名（关联度：cf，共同群组：cg）；其中“关联度”显示 commonFriends，与定义一致；共同群组仅在 cf>0 的情况下才会>0（否则该人不会进入列表）。

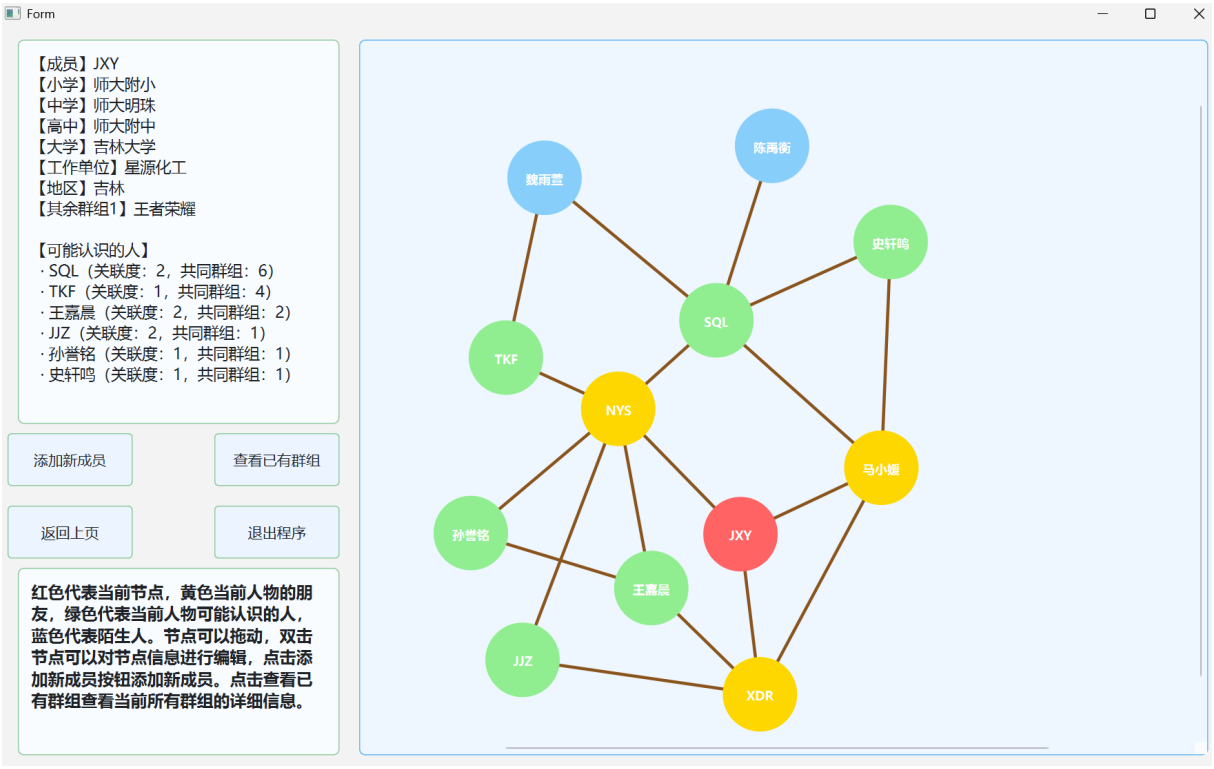

图 2.6 如图所示红色代表当前节点，黄色当前人物的朋友，绿色代表当前人物可能认识的人，蓝色代表陌生人。

2.6 开发平台

- 内存：16GB
- 操作系统：Windows11
- CPU：Intel Core i7-13650HX CPU
- 开发语言：C++ (C++ 11 标准及以上)
- 开发框架：Qt 6.9.1
- 集成开发环境：Qt Creator 17.0.0 (Community)
- 运行环境：Debug 版本和 Release 版本使用 QCreator 集成开发环境可以正常编译运行，使用 Qt 6.9.1 的 windeployqt 工具打包后的可执行文件基本可在 windows 环境的机型下正常运行。

2.7 系统的运行结果分析说明

2.7.1 调试与开发过程

调试阶段我主要用 QtCreator 自带的 gdb：在关键路径如 ShowNetwork::showFullNetwork()、ShowNetwork::refreshColorsAndInfo()、SocialGraph::potentialAcquaintances()、mutualFriends()、sharedGroups()、setMembershipOfType()、rebuildGroupsFromAttributes()、loadFromFile()/saveToFile() 以及图形联动的 NodeItem::itemChange()、EdgeItem::adjust() 处打断点、单步跟进；在“变量/表达式”窗口

重点观察 adj 与 groupIndex 的尺寸变化、候选集合与 cf/cg/score 的取值、persons[p].groups 的前后差异以及 positions 的写入时机（拖拽节点时 xChanged/yChanged→graph_.setPosition() 是否触发）。同时配合 qDebug() 在新增/编辑成员的入口（on_add_new_member_Button_clicked()、editMember()）打点，打印当前中心点、好友数、推荐条目数与 infoBox 输出内容，核对“只有 cf>0 才计入候选”这一新约束是否生效，并验证排序键为 score→cf→cg；对数据一致性问题，我还在 removeFriendship() 后检查邻接与倒排是否对称更新，以及在加载初始两人样例后

开发前我先画了简要类/对象图，明确分层与职责边界：MainWindow(QMainWindow) 只做路由与切页；ShowNetwork(QWidget) 负责整张网络的绘制、交互与信息面板；AddMemberDialog/EditMemberDialog(QDialog) 采集与修改数据；数据层由 SocialGraph(QObject) 统一管理 persons/groups/adj/groupIndex/positions 并提供 JSON 持久化与推荐算法接口。实现坚持模块化：数据层负责增删改查/双索引维护/推荐计算，可视化层只根据仓库数据重建场景与上色，页面层通过信号/槽 orchestrate；更新算法时，我们在数据层统一了规则（只有存在共同好友时才计算共同群组并计分），并让界面直接复用 potentialAcquaintances() 输出，避免“着色与列表来源不一致”的逻辑冲突。遇到 API 与事件细节不熟的地方，我查 QtAssistant（如 QGraphicsObject 的变更通知、QSet/QHash 语义）并结合社区经验快速补齐。

2.7.2 软件的成果分析

以下是根据我的身边的具体情况构建的社会关系网络图，注意本次演示的成员信息基本上正确但是为了演示的简洁性，部分成员的朋友关系被删去，请当事人不要在意。

进入程序后，即可直接观察到此时社会网络的结构，通过单击不同节点可以在左上角输出不同成员的具体信息。包括姓名以及其余群组信息，以及可能认识的人。在网络中也会实时通过不同节点的颜色反馈出当前节点的朋友，可能认识的人（我们在此规定只有朋友的朋友才有可能是可能认识的人）等信息。

通过点击添加新成员按钮即可添加新成员并且对新成员进行信息的编辑。双击节点即可对节点信息进行编辑，可以对所属群组以及朋友关系进行编辑。点击查看已有群组即可在左上角查看目前所有的群组的详细信息（受限于窗口大小这里只展示部分）。在网络展示部分可以看出，对可能认识的人进行倒序输出，综合打分结果，关联度与共同群组的比例为 1:1，因此在 JXY 视角下，可能认识的人的优先级里，TKF 要大于王嘉晨。

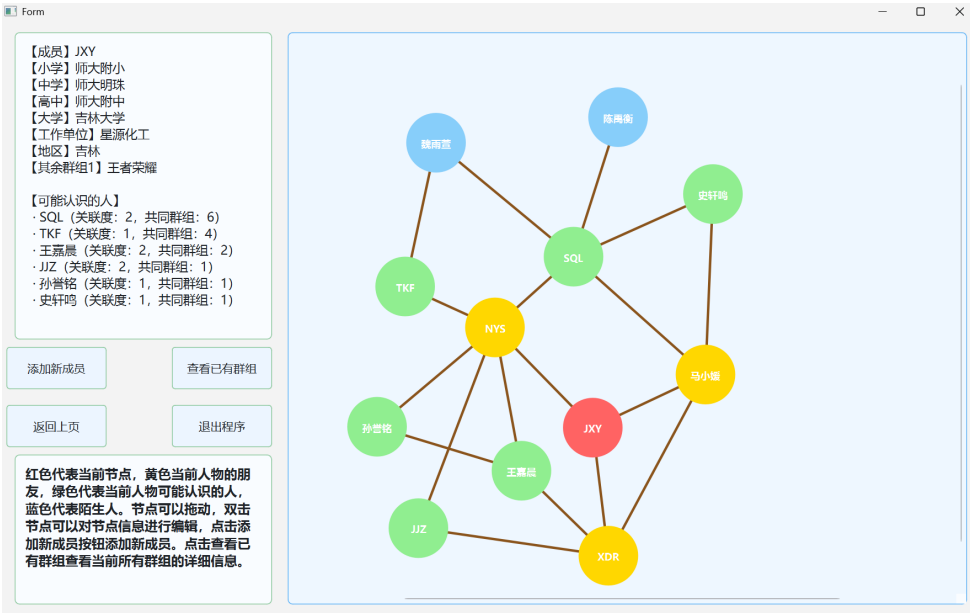

图 2.7 展示社会关系网络页面

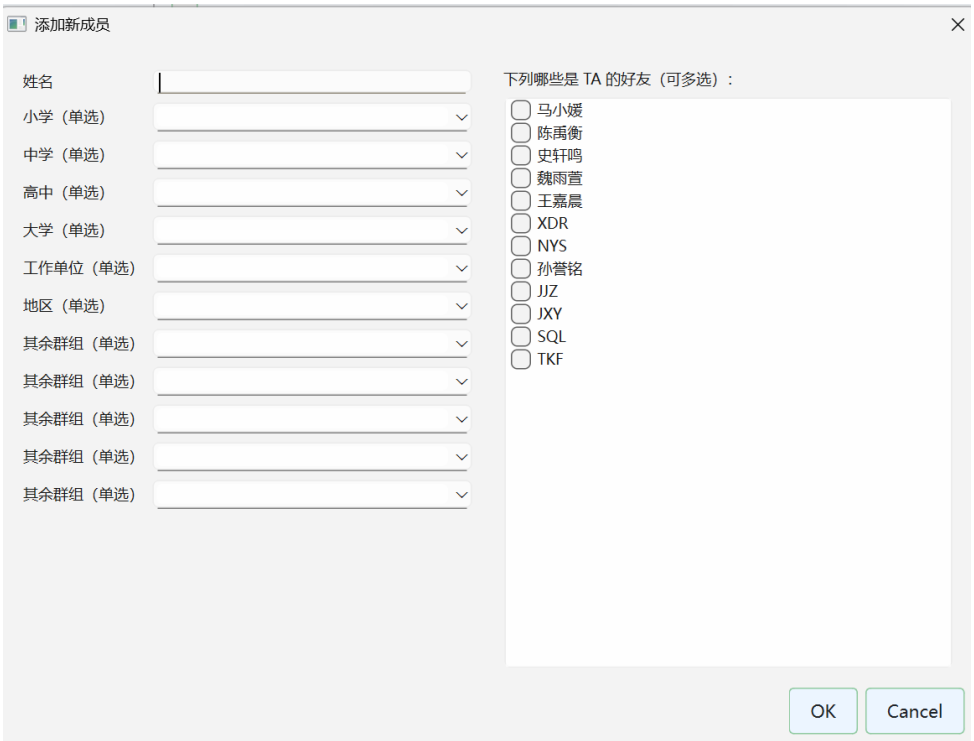

图 2.8 添加新成员窗口

图 2.9 编辑成员信息窗口

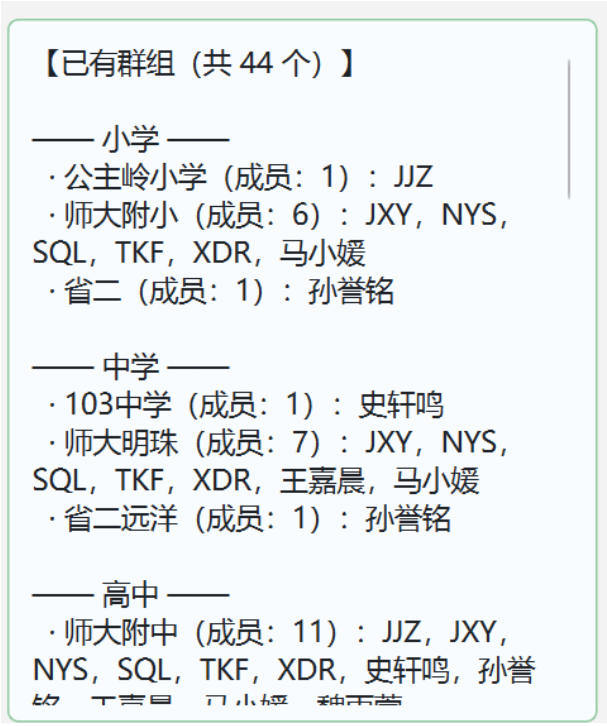

图 2.10 详细群组信息展示

2.8 操作说明

1、首先打开文件夹中的可执行文件文件夹，打开里面的二叉树文件夹，其中的 SocialNetwoks.exe 为可执行文件，打开即可进入程序。

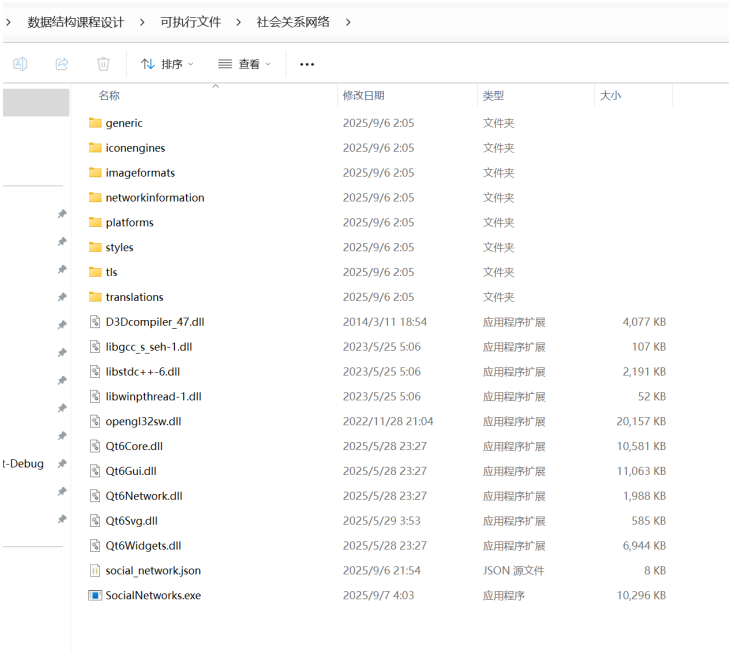

图 2.11 文件结构

2、点击进入程序后，显示如下界面，点击“进入演示”即可进入演示界面。

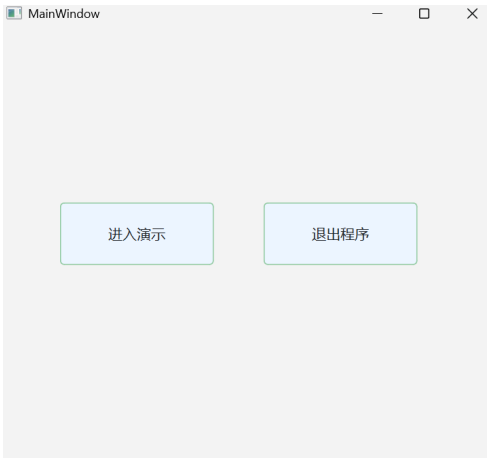

图 2.12 程序入口

3、在演示界面可以通过点击“添加新成员”按钮来添加新成员，点击之后弹出窗口，可以对个人信息进行编辑，并且个人信息中名字为必填项，如果未填则会弹出提示。构建过程中还可以添加群组信息以及与已有成员的朋友关系。添加之后新节点会随机出现在一个位置，此外节点可以被拖

动改变位置。信息的修改是同步的，新增加的群组会随后出现在群组选项中。单击节点可以显示该节点的详细信息，可能认识的人通过改变颜色直接现实在网络中

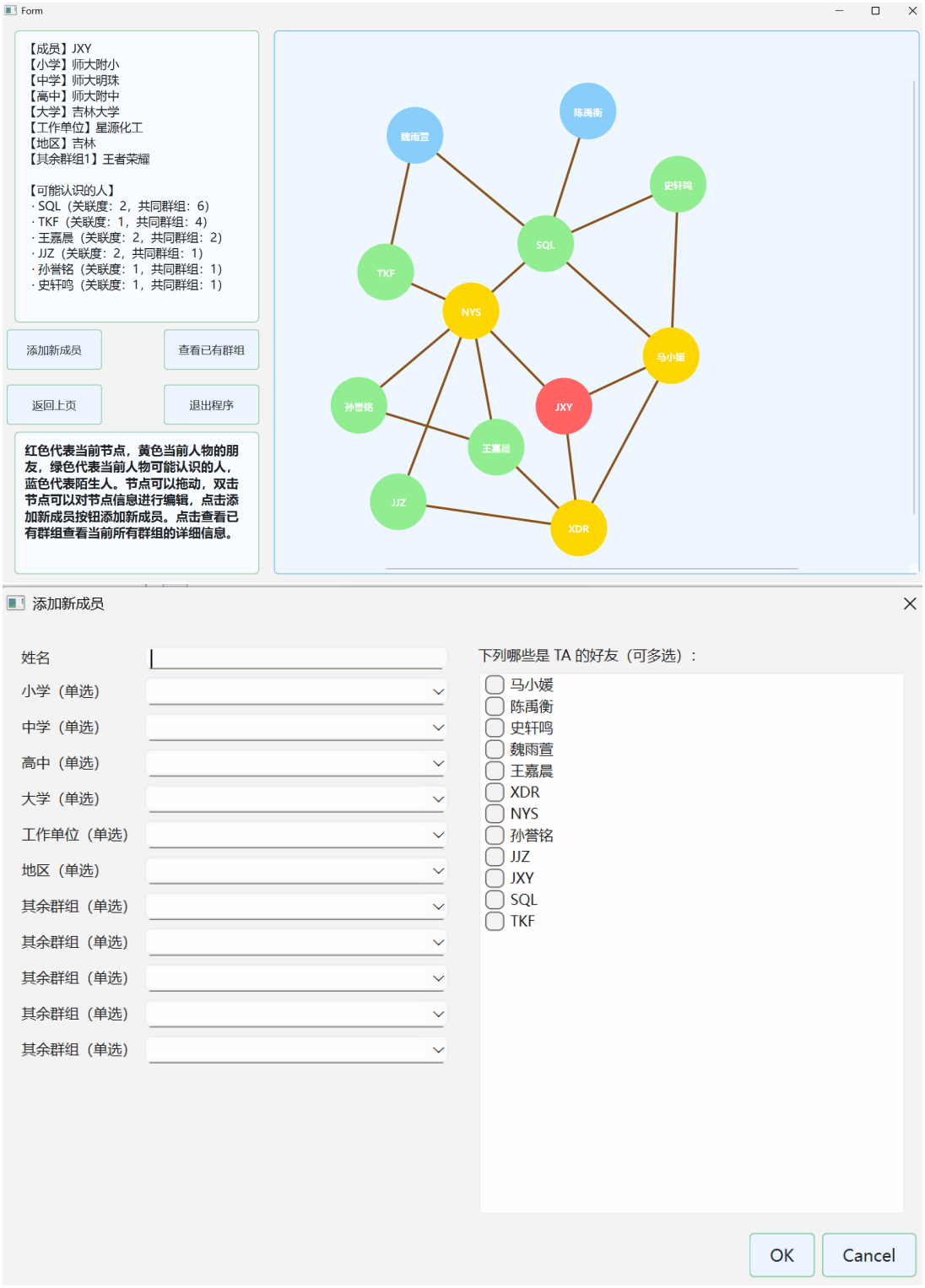

图 2.13 演示页面


【成员】孙誉铭
【小学】省二
【中学】省二远洋
【高中】师大附中
【大学】湖南大学
【工作单位】大疆
【地区】湖南
【其余群组1】音游
【其余群组2】工业设计专业
【其余群组3】有对象
【其余群组4】擅长棒球
【其余群组5】喜欢星球大战
【可能认识的人】
· XDR（关联度：1，共同群组：3）
· JXY（关联度：1，共同群组：1）
· SQL（关联度：1，共同群组：2）
添加新成员
查看已有群组
返回上页
退出程序
红色代表当前节点，黄色当前人物的朋
编辑成员
姓名
孙誉铭
小学（单选）
省二
中学（单选）
省二远洋
高中（单选）
师大附中
大学（单选）
湖南大学
工作单位（单选）
大疆
地区（单选）
湖南
其余群组（单选）
音游
其余群组（单选）
工业设计专业
其余群组（单选）
有对象
其余群组（单选）
擅长棒球
其余群组（单选）
喜欢星球大战
与下列成员的好友关系（可多选）：
马小媛
陈禹衡
史轩鸣
魏雨萱
王嘉晨
XDR
NYS
JJZ
JXY
SQL
TKF
OK
删除该成员
Cancel
【已有群组（共 44 个）】
—— 小学 ——
· 公主岭小学（成员：1）：JJZ
· 师大附小（成员：6）：JXY，NYS，SQL，TKF，XDR，马小媛
· 省二（成员：1）：孙誉铭
—— 中学 ——
· 103中学（成员：1）：史轩鸣
· 师大明珠（成员：7）：JXY，NYS，SQL，TKF，XDR，王嘉晨，马小媛
· 省二远洋（成员：1）：孙誉铭
—— 高中 ——
· 师大附中（成员：11）：JJZ，JXY，NYS，SQL，TKF，XDR，史轩鸣，孙誉

图 2.14 演示页面

3 实践总结

3.1 所做的工作

在线索二叉树部分，我先明确功能目标：建树、先/中/后序遍历、统计叶子、先/中/后序线索化与相应遍历、可视化展示及交互删除等。数据模型采用“节点+树”两层：ThreadedNode 仅承载节点的左右指针与父指针，并用 ltag/rtag 区分“孩子”与“线索”的语义边界；常用操作（判断是否为叶子、清理线索、寻找中序前驱/后继等）都围绕这套语义展开，避免把线索当孩子来走。BinaryTree 统一管理节点的创建与销毁，负责建满二叉树、清理、统计与三种遍历，并提供三类线索化及对应遍历；在生命周期管理上先清线索再递归释放，保证不沿“线索边”误走结构。

可视化层以场景类呈现节点、父子连线与线索箭头，节点图元支持点击高亮与删除，页面层划分为“建树/基础交互页”和“遍历与线索化演示页”，由主窗口统一调度并共享同一棵树的根，实现“边学边做”的交互式演示流程。

社会关系网络部分，通过一个仓库类集中维护成员、群组、无向好友邻接、组 → 成员倒排与节点位置等容器，并提供增删改查与 JSON 持久化能力；界面层基于图形场景重建整张网络、支持拖拽与连线联动；页面层负责路由与对话框采集。有关“可能认识的人”的判定，我将推荐规则定义：必须存在共同好友才计入候选，再按共同群组加分排序；视图直接消费该接口的结果，上色与信息面板与算法完全同源，保证一致性与可维护性。

3.2 总结与收获

万事开头难。在这个过程中，我对 Qt 的图形视图体系、事件与信号/槽机制、跨页面的数据共享以及资源打包发布有了更系统的掌握。更重要的是，我意识到：只要在设计之初把关键问题想透，并用少量而清晰的接口贯通各层，就能在开发与调试中兼顾正确性与稳定性。对知识与技术要保持热情与信心——唯有热爱，方有足够的动力把事情做好；也唯有勇敢迈出第一步，才能迎接后续的挑战与惊喜。总体来看，这次课程设计让我收获颇丰，编程能力与学习能力都有明显提升。由衷感谢老师的指导与同学们的支持，没有你们的付出，就没有我的进步与成长。

4 参考文献

[1] 陆文周.Qt5 开发及实例 (第 3 版)[M]. 北京: 电子工业出版社,2017.

[2] 王维波, 栗宝鹃, 侯春望.Qt 5.9 C++ 开发指南 [M]. 北京: 人民邮电出版社,2018.

[3] 霍亚飞.Qt Creator 快速入门 (第 3 版)[M]. 北京: 北京航空航天大学出版社,2017.

[4]Thomas H.Cormen,Charles E.Leiserson,Ronald L.Rivest,Clifford Stein. 算法导论 [M]. 北京: 机械工业出版社,2013.

[5] 严蔚敏, 吴伟民. 数据结构 (C 语言版)[M]. 北京: 清华大学出版社,2007.

[6] 查康. 精通 Git (第二版) [M] 北京: 人民邮电出版社,2017.

[7] 哲学的天空.QT 中使用 QSS 进行界面美化及很好的 QSS 样式表 [EB/OL]. (2020-5-3) [2025-9-07]. https://blog.csdn.net/u012278016/article/details/105906577.

[8] 狄奥尼索斯.QT · 页面跳转，怎么切换到另一个界面 [EB/OL].(2020-5-8)[2025-9-7].https://blog.csdn.net/qq_46305940/article/details/105993158 .

[9] 友善啊，朋友.Qt 图形视图图框架：QGraphicsView 详解 [EB/OL]. (2022-10-24) [2025-9-07]. https://blog.csdn.net/kenfan1647/article/details/117383803 .

[10] 在下小吴. 线索二叉树（图解+完整代码）[EB/OL].(2022-04-26) [2025-09-07]. https://blog.csdn.net/weixin_54186646/article/details/124435916

[11] 胡乱 huluan. 数据结构图（二）社交网络 [EB/OL].(2020-2-22) [2025-9-07]. https://blog.csdn.net/qq_44867435/article/details/104443790.

[12]iw1210.Qt 程序打包发布方法（使用官方提供的 windeployqt 工具）[EB/OL]. (2016-4-26) [2025-9-07]. https://blog.csdn.net/iw1210/article/details/51253458 .