

《数据结构与算法设计》

课程设计总结

学号： xxxxxxx

姓名： Ray

专业： 计算机科学与技术

2021 年 7 月

目 录

第一部分 算法实现设计说明.....	1
1.1 题目	1
1.2 软件功能	1
1.3 设计思想	1
1.3.1 功能设计思想	2
1.3.2 UI 设计思想	4
1.4 逻辑结构与物理结构	7
1.4.1 逻辑结构	7
1.4.2 物理结构	8
1.5 开发平台	8
1.6 系统的运行结果分析说明	9
1.6.1 调试开发过程	9
1.6.2 软件成果展示	10
1.6.3 运行结果案例	10
1.7 操作说明	13
1.7.1 键值操作	13
1.7.2 阶值操作	13
1.7.3 视图操作	13
第二部分 综合应用设计说明.....	14
2.1 题目	14
2.2 软件功能	14
2.3 设计思想	15
2.3.1 数据结构选择	15
2.3.2 功能设计思想	17
2.3.3 UI 设计思想	19
2.4 逻辑结构与物理结构	23
2.4.1 逻辑结构	23
2.4.2 物理结构	24

2.5 开发平台	24
2.6 系统的运行结果分析说明	24
2.6.1 调试开发过程	24
2.6.2 软件成果展示	26
2.6.3 运行结果案例	27
2.7 操作说明	30
2.7.1 线性表操作	30
2.7.2 树结构操作	32
2.7.3 树视图选项	32
2.7.4 其他操作	32
第三部分 实践总结.....	33
3.1. 所做的工作.....	33
3.1.1 底层数据结构	33
3.1.2 平台与 UI.....	33
3.1.3 调试与拓展	33
3.2. 总结与收获.....	34
3.2.1 课设总结	34
3.2.2 心得体会	35
第四部分 参考文献.....	36

第一部分 算法实现设计说明

1.1 题目

0. 试从空树出发构造一棵深度至少为 3 (不包括失误节点) 的 3 阶 B-树 (又称 2-3 树), 并可以随时进行查找、插入、删除等操作。

要求: 能够把构造和删除过程中的 B-树随时显示输出来, 能给出查找是否成功的信息。

1.2 软件功能

根据题目的要求, 本软件实现的基本功能以及对应的实现方法如下:

- ① 正确的 B-树结构: 灵活运用 C++、QT 基本的数据结构, 自行实现 B-树类;
- ② 正确的查找、插入、删除操作: 根据 B-树的特点和已有算法, 自行设计算法逻辑;
- ③ 树形结构可视化: 使用 QT 中的 QTreeView 类, 设计算法实现底层 B-树的展示;
- ④ 面向对象的操作方式: 使用 QT 中的 QPushButton 类, 方便用户的对 B-树操作;
- ⑤ 提供查找结果的信息: 使用 QT 中的 QMessageBox 类, 结合底层 B-树查找算法;
- ⑥ 随时显示 B-树: 添加更新信号, 当 B-树底层数据发生变化时, 触发窗口的更新;

此外, 为了使用户的操作更加流畅、B-树的结构特点更加突出, 程序在最终实现时, 还额外增加了以下功能:

- ① 选择演示题目: 在演示窗口之前增加“选择窗口”, 整合课设的两个演示程序;
- ② 显示 B-树的基本信息: 通过 QT 中的 QLabel 类, 实时更新 B-树的基本信息;
- ③ 自定义 B-树的阶数: 在底层数据结构中添加阶数信息, 以实现更多类型的 B-树;
- ④ 自定义 B-树的可视化方式: 为不同的可视化方式设计相应的可视化算法;
- ⑤ 提供用户操作指南: 设计“帮助”工具栏和对应弹窗, 方便用户查看操作说明;

最终设计的 B-树演示程序, 正确实现了构造、维护、展示一个深度为 n 、阶数为 m 的 B-树的功能 ($n \geq 0$, $m > 2$ 且初始值为 3), 同时也实现了用户交互的功能, 便于进行查找、插入、删除、清空、重置阶数等操作, 还可以自行设置 B-树展示的信息。

1.3 设计思想

软件在设计时采用瀑布模型, 将功能设计与 UI 设计分离, 后期再根据实际需求微调应用程序, 但程序内核基本不变。以下将从“功能设计”与“UI 设计”两个方面分别说明。

1.3.1 功能设计思想

本软件的功能设计，主要是 m 阶 B-树的数据结构和各个操作的设计。关于 B-树数据结构设计的内容，将在 1.4 中详细介绍，这里主要介绍其各个操作的实现思路和算法流程。

(1) B-树的插入操作：

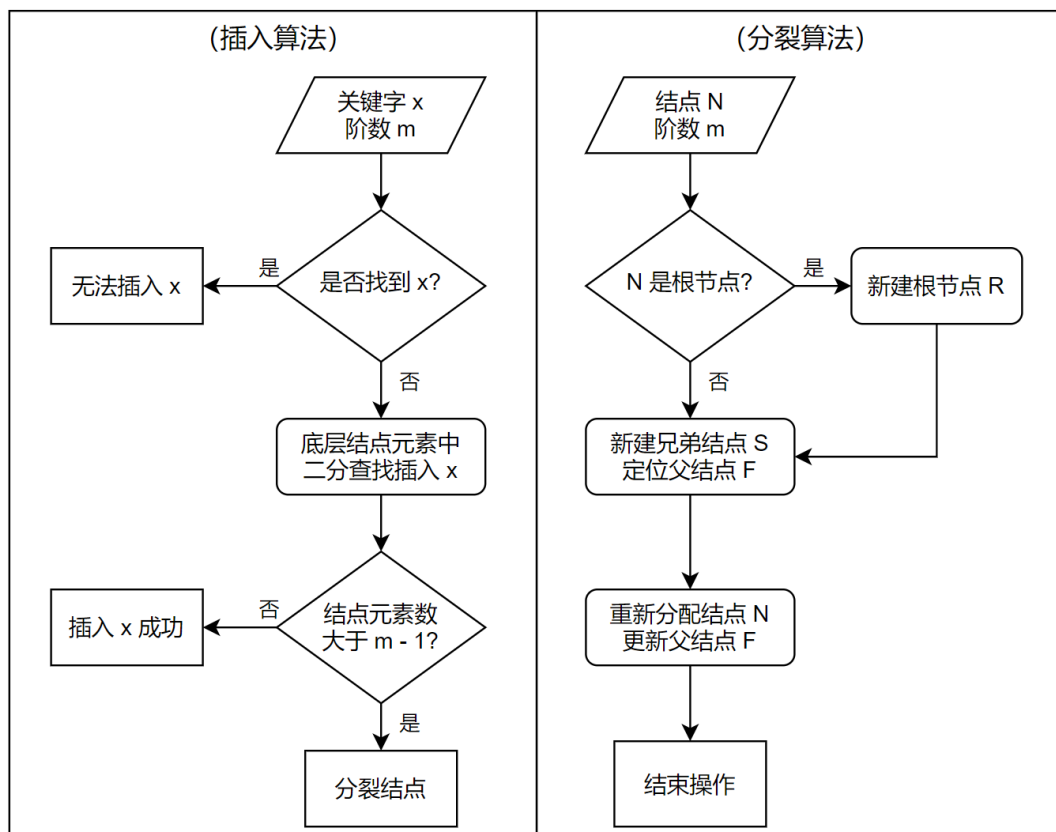
B-树的插入操作是构造、维护一棵 B-树的重要操作。B-树是从空树开始通过不断插入新的数据元素构建的，但它并不是每次都向树中插入新的节点。 m 阶 B-树在定义中规定：

非叶子节点中包含关键字个数范围 $[m/2-1, m-1]$ ，叶子节点关键字个数为 0

所以在插入新的数据元素时，首先向底层的某个非叶子节点中添加，有下面两种情况：

- ① 该节点中的关键字个数没有超过 $m-1$ ：直接插入成功；
- ② 该节点添加完关键字后(排好序)达到 m ：该节点分裂成两个节点，同时维护 B-树；

因此，设计的 B-树插入算法流程如下(右侧是对应的分裂节点算法流程)：

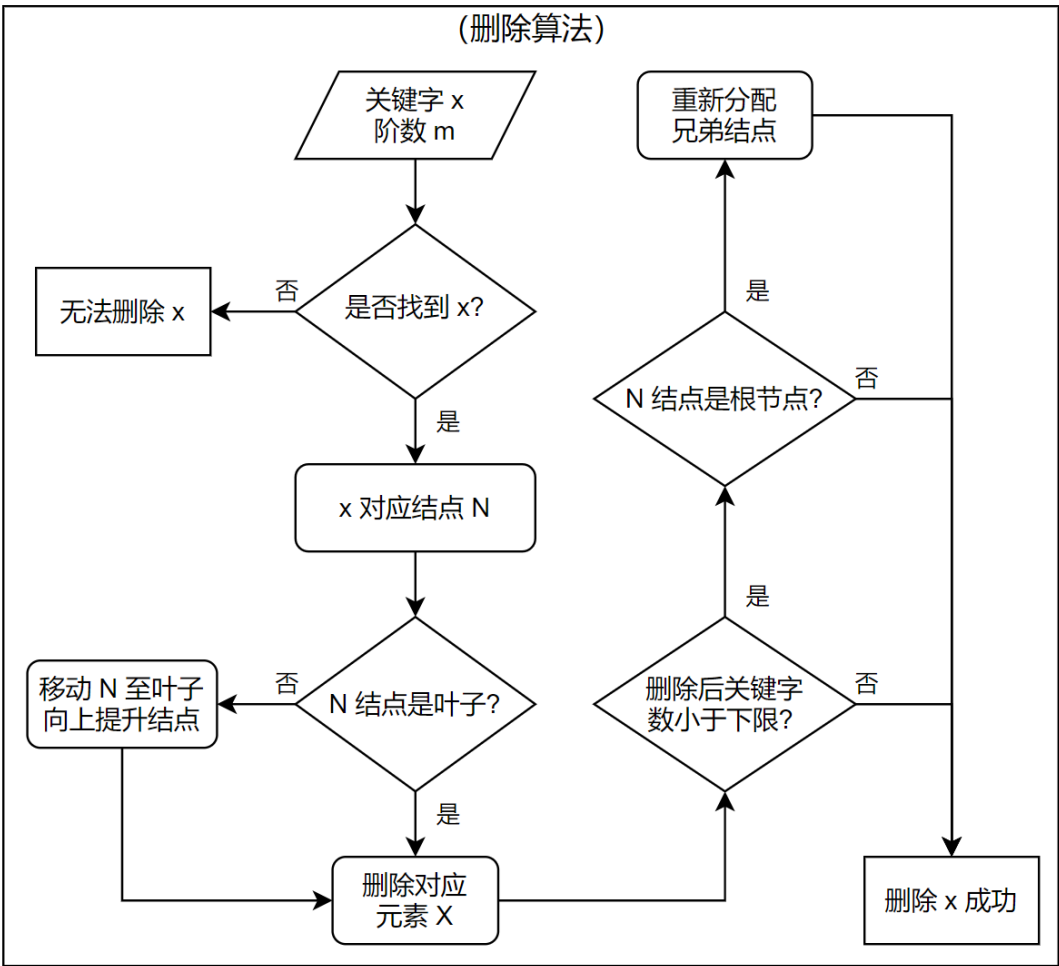


(2) B-树的删除操作：

B-树的删除操作需要考虑删除之后节点的关键字数量是否小于下限。如果删除之后节点中的关键字小于下限，则需要从子树中获取关键字，或从相邻兄弟节点获取关键字，或和相邻兄弟节点合并。如果从子树获取关键字之后，子树依然能保持 B 树的基本要求，则可以从子树中获取，否则看相邻兄弟节点元素是否富余，富余的话可以从兄弟节点获取元素，相邻

兄弟节点也不富余就需要和兄弟节点合并。和兄弟节点合并时，需要把当前节点和兄弟节点之间的那个父节点关键字下移，然后和该节点还有兄弟节点组成新的节点，基于 B-树节点关键字的数量要求，这样合并出来的新节点关键字数不会超上限。

因此，设计的 B-树删除算法流程如下：



(3) B-树的查找操作：

B-树在构造、修改的过程中，需要一直的维护和调整，这也给 B-树在搜索时提供了巨大的优势。在使用 B-树进行查找操作时，先从整棵树的根节点开始，根据节点中的键值和待查找数据的大小关系，确定下一步进入的子树的指针，然后重复该操作，顺着指针自上而下，直到找到存有关键字的节点。若查找到叶子节点后仍没有找到关键字，则查找失败。

查找的流程比较简单，不再绘制算法流程图。需要注意的是，由于每个节点内部的键值排序有序，因此在查找时，可以使用二分查找提高内部查找速率。同时，由于查找算法在整个 B-树的操作过程中应用广泛，因此特别设计搜索结果结构体(Result)，存储搜索的结果信息(查找是否成功、查找最终停留的节点、查找最终定位到的键值位置或失误范围)，这样方便在别的函数中直接调用搜索结果，避免二次查找这些数据。

1.3.2 UI 设计思想

本软件的 UI 设计，主要包括以下几个部分：

- ① 选择界面和帮助界面：选择 B-树、综合实验或帮助文档；
- ② 对 B-树操作的指令按钮：插入、删除、查找、重设阶数的按钮；
- ③ 树结构的显示窗口：以文件树的形式显示 B-树；
- ④ 对树结构显示的指令按钮：显示失误节点、展开/收起、清空的按钮；
- ⑤ B-树结构的信息基本信息：B-树的阶数、深度(不算失误节点)和键值数；

以下将按照本程序整体的操作流程，简要介绍各个模块 UI 的设计思想和算法流程。

(1) 选择界面和帮助界面：

设计选择界面时，经过反复的斟酌与考查，最终选择模仿“佳能打印机功能选择”程序的界面和操作方法(如右图)：

仅提供选项工具栏和必要信息，窗口以横排小窗呈现，不可放大且工具栏固定。单机工具栏选项后，该窗口自动隐藏，弹出所选功能子窗口，从而模拟窗口切换的效果(实际只是隐藏，并未关闭)。



帮助界面的设计主要模仿了 office 软件中常见的各类设置子窗口，使用的 Tab 窗口的方式，管理不同演示程序的帮助文档。帮助界面的具体样式请参照 1.6 或 1.7 中程序截图。

(2) 各类指令按钮和输入框：

各类指令按钮与对应的功能绑定，样式与 window 常见按钮和输入框一致。此外，程序还添加有选择框样式按钮，输入框也设置为只能接收整数的状态，确保输入正确。



为了区分各个按键，特别添加了分类框(QGroupBox 类)，将不同的操作按钮打包分离，从而增加界面的美观性，让程序结构更加鲜明。

(3) 基本信息文本框：

基本信息文本框也使用了 QGroupBox 进行分类，内容更新与绘制树同步，不再赘述。

(4) 树结构的显示:

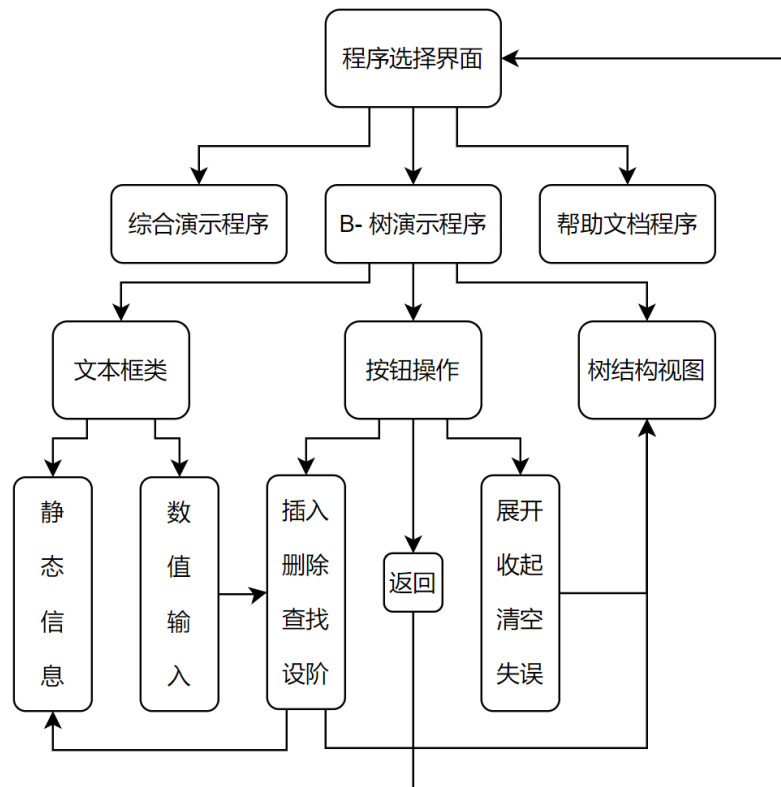
为了正确、美观的显示树结构,这里采用 Model/View 结构,使用 QT 的 QTreeView 视图类和 QStandardItem 模型类,通过设计的算法将 B-树底层数据结构转化为 QStandardItem 数据模型,最后通过 QTreeView 来显示。采用这样的设计结构,可以将界面组件和编辑的数据分离开来,又通过数据源的形式连接起来,符合原定软件设计时“功能设计”与“UI 设计”相分离的特点,较好的实现了树结构的可视化。

```
// 递归绘制树
void BTreeWindow::drawBTree(QStandardItem *fItem, BNode *node, int d, const QString l, const QString r){
    // 统计B树信息
    keyNum = keyNum + node->num;
    if(d > treeDepth)
        treeDepth = d;
    // 绘制子节点
    for (int i = 0; i <= node->num; i++){
        BNode *cNode = node->child.at(i);
        // 计算范围
        QString left=(i == 0 ? l:QString::number(node->key.at(i-1)));
        QString right=(i == node->num ? r:QString::number(node->key.at(i)));
        // 结点
        if (cNode){ // 生成结点
            QStandardItem *cItem = new QStandardItem(QString::number(d)+"层子结点");
            cItem->setIcon(iconBNode);
            fItem->appendRow(cItem);
            fItem->setChild(cItem->index().row(), 1, new QStandardItem(QString::number(d)+"层子结点"));
            fItem->setChild(cItem->index().row(), 2, new QStandardItem(QString::number(cNode->num)));
            drawBTree(cItem, cNode, d+1, left, right);
        }
        else if(failShow){ // 生成失败结点
            QStandardItem *cItem = new QStandardItem(QString::number(d)+"层子结点");
            cItem->setIcon(iconBFail);
            fItem->appendRow(cItem);
            fItem->setChild(cItem->index().row(), 1, 0);
            fItem->setChild(cItem->index().row(), 1, new QStandardItem(QString::number(d+1)+"层失败结点"));
        }
        // 键值
        if (i != node->num){
            QStandardItem *cItem = new QStandardItem(right);
            if(d < treeDepth){
                cItem->setIcon(iconBKey);
                fItem->appendRow(cItem);
                fItem->setChild(cItem->index().row(), 1, new QStandardItem(QString::number(d)+"层键值"));
            }
            else{
                cItem->setIcon(iconBLeaf);
                fItem->appendRow(cItem);
                fItem->setChild(cItem->index().row(), 1, new QStandardItem(QString::number(d)+"层叶子结点"));
            }
        }
    }
}
```

底层 B-树数据转化为 QTreeView 视图的算法如下(实际效果见 1.6 或 1.7 截图):

函数 drawBTree(QStandardItem *fItem, BNode *node, int d, QString l, QString r) 中, fItem 是存在于 QStandardItem 中的父项, 对应 B-树一个节点的父节点。node 是当前在 B-树中的节点, 函数要绘制的也是它的节点树(把当前 node 转化为 Item, 并插入到父项当中, 建立关联)。d 是当前递归的深度, l 和 r 是当前节点的上、下界, 这几个数据是用来输出节点信息的, 方便用户直观的了解 B-树各个节点的状态和位置。此外, 函数还会通过两个全局变量(keyNum->树中的键值总数、treeDepth->B-树的深度)记录树的信息, 从而在树完全绘制完成后, 更新基本信息文本框。

(5) 程序整体流程图和 UI 效果图:



1.4 逻辑结构与物理结构

1.4.1 逻辑结构

本次 B-树演示程序的使用到的逻辑结构主要有：树形结构(B-树及其节点)，此外还有部分辅助结构，如线性结构(节点中存储键值和子节点指针的顺序表)、其他结构体(存储搜索结果的 Result 结构体)。

(1) B-树结构：

B-树内部(非叶子)节点可以拥有可变数量的子节点，其每一个内部节点会包含一定数量的键值，而一个节点的分支(或子节点)数量会比存储在节点内部键值的数量大 1。当数据被插入或从一个节点中移除，它的子节点数量发生变化，为了维持在预先设定的数量范围内，内部节点可能会被合并或者分离。一棵 B-树通过约束所有叶子节点在相同深度来保持平衡，深度在元素添加至树的过程中缓慢增长。

<pre>typedef struct BNode{ int num; // 键值数 BNode* father; // 父结点 QList<int> key; // 键值表 QList<BNode*> child; // 孩子表 }BNode;</pre>	<pre>typedef struct BTree{ int degree; // 树阶 BNode *p; // 动态根节点 }BTree;</pre>
--	---

在本程序中，B-树节点和 B 树的主要结构设计如下：

BNode 是 B-树的节点，一个节点中，除了存储当前节点的键值数、父节点指针，还有两个线性结构的顺序表，这里使用 QT 的 QList 类实现动态分配内存的、可以随机读取的顺序表。key 中存储着当前节点内的键值，child 中存储着当前节点的子节点指针，指向下一个节点，或者是失误节点，是典型的一对多的树形结构。

BTree 是 B-树的整体结构，包含了一个 BNode 类型的根节点指针(哨兵节点)，以及该 B-树的阶数数据。树的阶数决定了节点的键值数、孩子指针数的上界和下界，因此在两者的构造函数中，需要加入传递阶数的信息，便于构造匹配的新节点。

为了方便操作，后期将 BNode 和 BTree 转化为对应的结构体，同时封装两个类型基本的算法操作，各个操作的内容可以参照文档 1.3.1 的功能设计。

(2) 顺序表及其他结构：

顺序表是典型是线性结构，可以实现随机读取。QList 类是 QT 中最常用的顺序容器类，以数组列表的形式实现，通过下标索引的方式对数据项进行访问，在本软件中，主要是在 B-树的节点中使用，用来作为键值和子节点指针的容器，从而实现高效的快速的读取。关于顺序表的操作这里不再赘述。

Result 结构体是用来记录搜索信息的结构体，其设计如下：

```
// B-树搜索结果结构体
typedef struct Result {
    bool f;           // 是否找到
    int place;        // 对应位置
    BNode* node;      // 对应结点
}Result;
```

其中包括了 bool 变量 f 用来记录是否查找到改数值,BNode 型指针*node 指向最后搜索到的节点(对应叶子节点或键值所在的节点),以及 int 型变量 place 记录的最终定位在节点中表的位置,方便实现随机读取。该结构体与 B-树息息相关,是不可或缺的结构之一。

1.4.2 物理结构

本软件中,B-树结构使用链式存储的方式,即一棵完整的 B-树,各个节点在物理内存上的映射并不像相连,每一个节点的指针域指向多个子节点,节点与节点之间由许多分散的内存通过各节点的指针域相关联,不可以随机读取与存储。

选择这样的物理结构符合 B-树的树形结构的特征,只要有剩余的内存,就可以随时添加新的子节点,不需要预先开辟大量空间导致内存浪费,也不会出现因为预先开辟的内存不够而需要重新开辟空间或发生溢出。而且父子节点之间的指针联系更能表征树的特点。

此外,QList 类(B-树节点中的键值表和指针表)使用顺序存储的方式,即一个 BNode 节点内部,键值和子节点指针的在物理内存上的映射相连,是完整的区域,节点之间为一对一的关系,因此可以实现随机读写。采取这样的物理结构,可以大大缩短查找的时间,降低节点内部搜索的时间复杂度,也便于 B-树频繁的插入、删除、维护操作。

1.5 开发平台

本次软件的开发平台以及运行环境如下：

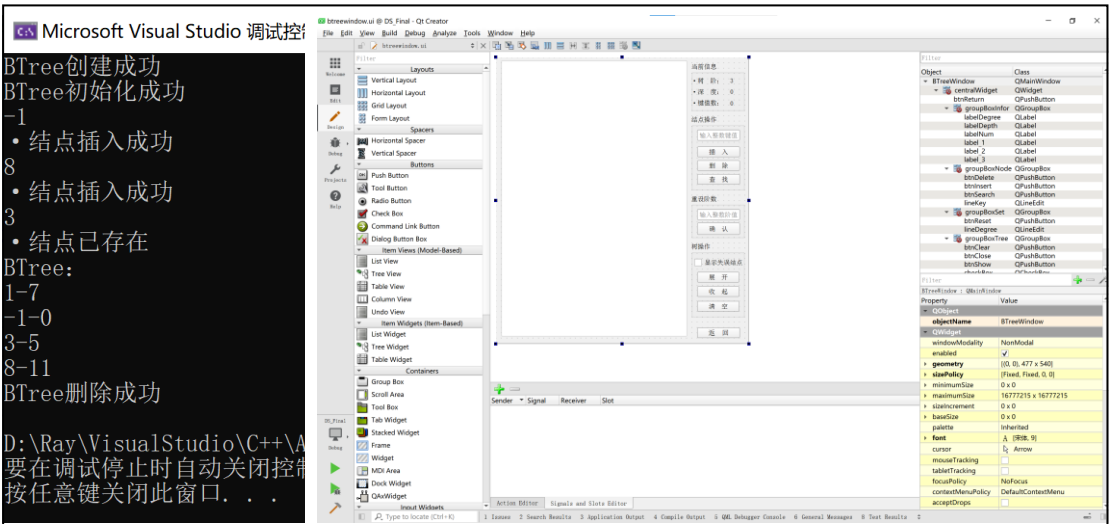
开发平台	Qt 平台(版本 6.1.2)
操作系统	Windows 10 专业版 21H1
编程语言	C++, xml (Designer 辅助生成)
开发工具	Visual Studio 2019, Qt Creator 4.15.2 (Community), Designer 6.1.2
打包工具	Qt Creator 4.15.2 (Community), Enigma Virtual Box
核心码库	Qt 6.1.2 (MSVC 2019, 64-bit), C++标准库
运行环境	Win x64

1.6 系统的运行结果分析说明

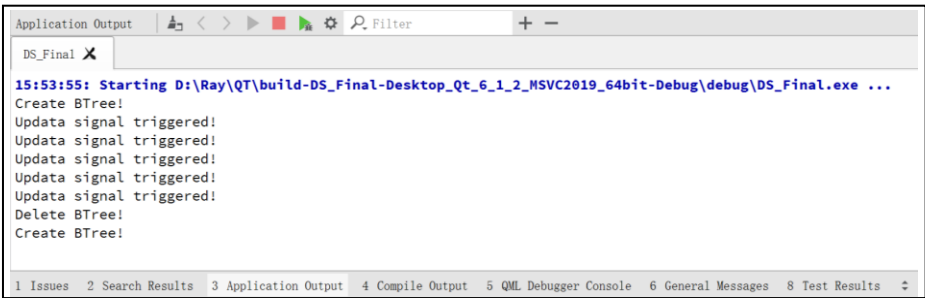
1.6.1 调试开发过程

在开发过程中，首先使用 Visual Studio 2019 设计 2-3 树的底层数据结构，通过控制台输出构造结果，同时配合设置断点、std::cout 输出、查看变量等方式进行调试，最终完成 B 树的控制台应用。

在设计完成底层逻辑之后，使用 Qt Creator 4.15.2 设计 GUI 程序。本人在 UI 设计过程中首先设计窗口界面，即 QMainWindow 类以及窗口对应的.ui 文件。由于 Qt Creator 同时配备着 Designer 6.1.2 作为设计工具，因此.ui 文件中的 xml 代码，主要通过 Designer 的可视化设计自行生成。UI 文件设计完成后，在窗口类中手动连接信号与槽函数，并完善可视化设计无法完成的部分，例如窗口的稳定、窗口选项的删除、TreeView 视图的菜单栏设计等。此过程可以通过运行程序直观调试。



窗口界面设计完成后，需要引入底层结构，实现数据向视图的转化。此过程中，为了统一文件的命名空间，选择将所有 std 容器(vector 类)转化为 Qt 容器(QVector、QList 类)，从而实现代码的美观统一。由于 GUI 程序需要更多的交互，因此在这个环节的实现了，需要向底层结构添加部分触发信号，也需要时刻关注底层数据的变化。在调试过程中，本人结合 qDebug() 函数与 QMessageBox 类输出程序运行情况，并通过设置断点、查看状态栏等方式寻找程序内部错误。



1.6.2 软件成果展示

最终实现的软件完整实现了题目要求，正确实现了插入、删除、查找的功能，允许构造的 B 树深度也远大于 3 层。在后期多组数据的检测过程中，本程序都体现出了较好的容错能力与运行速度，也不曾出现崩溃或异常退出的情况，实时构造的 B 树也一直符合定义要求。可以说，软件在功能上具有较好的正确性、稳定性、容错能力以及延展性。

此外，软件的 UI 设计采取简约风，指令按钮齐全且分类规范，视图窗口美观且提示信息友好。值得一提的是，窗口内使用的所有 UI 图标，均是通过 Adobe AI 软件自行设计，整体上风格统一、样式清新，具有良好的视觉效果。

1.6.3 运行结果案例

进入程序后，初始界面如图所示，共 3 中程序可供选择，此界面不可放大。



选择 B-树演示程序后，窗口跳转至如下左图。B-树演示的初始状态下树的阶数为 3，树中没有其他键值。在依次插入：

1 3 -7 9 10 2 5 7 -10 -12 0 -34 后，树的结构图如下右图所示：

The left screenshot shows the initial state of the B-tree interface. The tree is empty, with a root node labeled "0层树根" and a value of 0. The range is $(-\infty, +\infty)$. The current information shows a tree order of 3, depth of 0, and 0 key values. The interface includes buttons for "插入" (Insert), "删除" (Delete), "查找" (Search), "重设阶数" (Reset Order), "输入整数阶数" (Input Integer Order), "确认" (Confirm), "树操作" (Tree Operation), "显示失误结点" (Show Error Node), "展开" (Expand), "收起" (Collapse), "清空" (Clear), and "返回" (Return).

(初始状态/清空状态)

The right screenshot shows the B-tree structure after inserting the data set {1, 3, -7, 9, 10, 2, 5, 7, -10, -12, 0, -34}. The tree has a root node labeled "0层树根" with a value of 1. The range is $(-\infty, +\infty)$. The current information shows a tree order of 3, depth of 3, and 12 key values. The tree structure is as follows:

- 0层树根 (1): $(-\infty, +\infty)$
 - 1层子结点 (2): $(-\infty, 3)$
 - 2层子结点 (2): $(-\infty, -10)$
 - 3层叶子结点: -34
 - 3层叶子结点: -12
 - 2层键值: -10
 - 2层子结点 (2): $(-10, 1)$
 - 3层叶子结点: -7
 - 3层叶子结点: 0
 - 2层键值: 1
 - 2层子结点 (1): $(1, 3)$
 - 3层叶子结点: 2
 - 1层键值: 3
 - 1层子结点 (1): $(3, +\infty)$
 - 2层子结点 (2): $(3, 9)$
 - 3层叶子结点: 5
 - 3层叶子结点: 7
 - 2层键值: 9
 - 2层子结点 (1): $(9, +\infty)$
 - 3层叶子结点: 10

(输入数据组 1)

在该案例的基础上，选择“显示失误结点”后情况如左图，“收起”后如右图：

B-树

范围	类型	键值数
$(-\infty, +\infty)$	0层树根	1
$(-\infty, 3)$	1层子结点	2
$(-\infty, -10)$	2层子结点	2
$(-\infty, -34)$	4层失误结点	
-34	3层叶子结点	
$(-34, -12)$	4层失误结点	
-12	3层叶子结点	
$(-12, -10)$	4层失误结点	
-10	2层键值	
$(-10, 1)$	2层子结点	2
$(-10, -7)$	4层失误结点	
-7	3层叶子结点	
$(-7, 0)$	4层失误结点	
0	3层叶子结点	
$(0, 1)$	4层失误结点	
1	2层键值	
$(1, 3)$	2层子结点	1
$(1, 2)$	4层失误结点	
2	3层叶子结点	
$(2, 3)$	4层失误结点	
3	1层键值	
$(3, +\infty)$	1层子结点	1
$(3, 9)$	2层子结点	2
$(3, 5)$	4层失误结点	

当前信息
• 树阶: 3
• 深度: 3
• 键值数: 12

结点操作
输入整数键值
插入
删除
查找

重设阶数
输入整数阶数
确认

树操作
☒ 显示失误结点
展开
收起
清空
返回

B-树

范围	类型	键值数
$(-\infty, +\infty)$	0层树根	1

当前信息
• 树阶: 3
• 深度: 3
• 键值数: 12

结点操作
输入整数键值
插入
删除
查找

重设阶数
输入整数阶数
确认

树操作
☒ 显示失误结点
展开
收起
清空
返回

(显示失误结点)
(收起)

在该案例基础上，展开树并隐藏失误结点，删除 3、-10 后如左图，查找 3、1 如右图：

B-树

范围	类型	键值数
$(-\infty, +\infty)$	0层树根	1
$(-\infty, 5)$	1层子结点	2
$(-\infty, -7)$	2层子结点	2
-34	3层叶子结点	
-12	3层叶子结点	
-7	2层键值	
$(-7, 1)$	2层子结点	1
0	3层叶子结点	
1	2层键值	
$(1, 5)$	2层子结点	1
2	3层叶子结点	
5	1层键值	
$(5, +\infty)$	1层子结点	1
$(5, 9)$	2层子结点	1
7	3层叶子结点	
9	2层键值	
$(9, +\infty)$	2层子结点	1
10	3层叶子结点	

当前信息
• 树阶: 3
• 深度: 3
• 键值数: 10

结点操作
输入整数键值
插入
删除
查找

重设阶数
输入整数阶数
确认

树操作
☐ 显示失误结点
展开
收起
清空
返回

查找

查找失败，未找到该键值

OK

B-树

范围	类型	键值数
$(-\infty, +\infty)$	0层树根	1
$(-\infty, 5)$	1层子结点	2
$(-\infty, -7)$	2层子结点	2
-34	3层叶子结点	
-12	3层叶子结点	
-7	2层键值	
$(-7, 1)$	2层子结点	1
0	3层叶子结点	
1	2层键值	
$(1, 5)$	2层子结点	1
2	3层叶子结点	
5	1层键值	
$(5, +\infty)$	1层子结点	1
$(5, 9)$	2层子结点	1
7	3层叶子结点	
9	2层键值	
$(9, +\infty)$	2层子结点	1
10	3层叶子结点	

当前信息
• 树阶: 3
• 深度: 3
• 键值数: 10

结点操作
输入整数键值
插入
删除
查找

重设阶数
输入整数阶数
确认

树操作
☐ 显示失误结点
展开
收起
清空
返回

查找

查找成功，键值存在

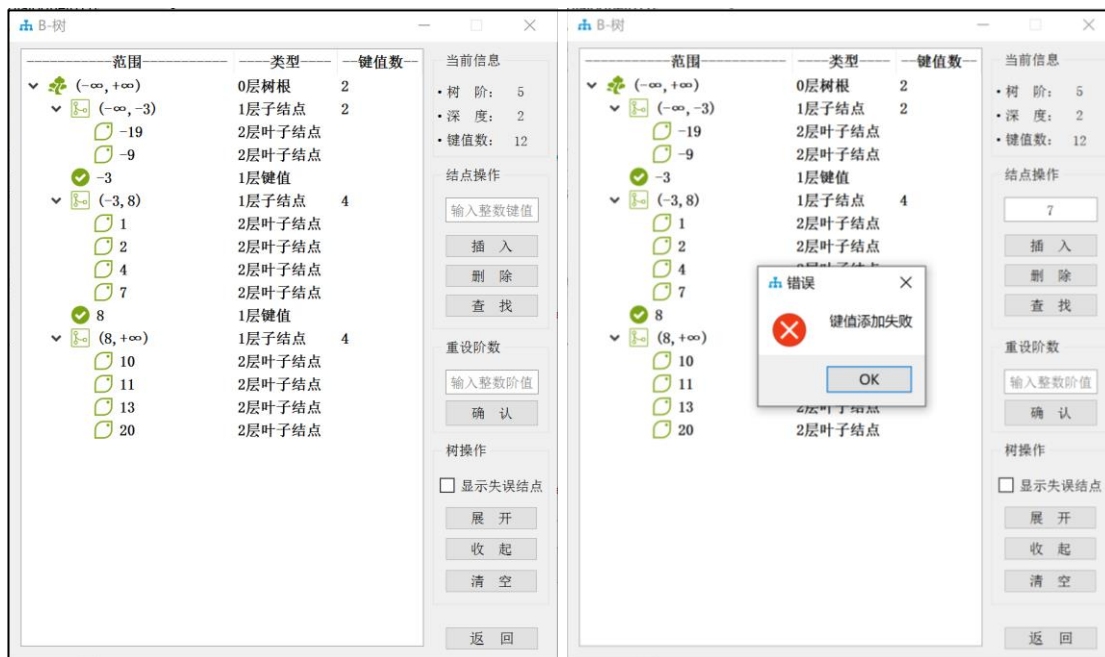
OK

(删除数据组 2)
(查找数据组)

之后的几组演示使用的数据组信息如下：

- ①数据组 A：树阶为 5，插入 10 8 -9 20 1 11 13 -19 7 -3 4 2，再插入 7；
- ①数据组 B：树阶为 3，插入 10 55 28 11 -9 -7 -3 4 2 66 99 90 -5 -1 -88 79；
- ①数据组 C：B 组基础上，删除 10 55 -9 4 -88 90；

输入以上数据后，B-树在构造和展示过程中均正确，可见程序的稳定性。



(数据组 A)

(数据组 A 插入失败)



(数据组 B)

(数据组 C)

演示结束后，点击返回将返回选择窗口；如直接关闭，将弹出关闭提示，说明结束进程。



1.7 操作说明

打开软件，在选择页面单击“B-树”工具栏，进入该演示程序。

1.7.1 键值操作

在“结点操作”对应栏目下，可对视图中 B-树进行键值的插入、删除、查找操作。用户首先需要在文本输入框①中输入一个整数数值，例如：1，-9 等，之后按照下方指南完成对应操作：

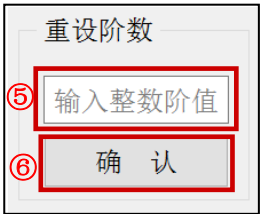
- A. 插入：单击按键②，或在输入数值后按回车键(Enter 键)；
- B. 删除：单击按键③；
- C. 查找：单击按键④；



1.7.2 阶值操作

在“重设阶数”对应栏目下，可对视图中 B-树进行修改阶值操作。用户首先需要在文本输入框⑤中输入一个大于 2 的整数数值，之后单击按键⑥(或按回车键/Enter 键)，确认修改键值。

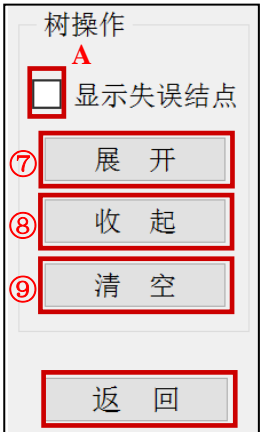
注意：修改键值后，原 B-树自动清空，需重新输入数据



1.7.3 视图操作

在“树操作”对应栏目下，可对视图中 B-树的展示方式进行展开、收起、清空的修改，也可以选择是否显示失误结点。用户可以按照下方指南完成对应操作：

- A. 显示失误结点：单击框格 A，勾选“显示失误结点”；
- B. 隐藏失误结点：单击框格 A，取消勾选“显示失误结点”；
- C. 展开树结构：单击按键⑦；
- D. 收起树结构至根节点：单击按键⑧；
- E. 清空视图中的树：单击按键⑨，清空后阶数不变；



最后，用户可以单击“返回”按钮，返回选择界面，返回后输入的信息不会丢失。

第二部分 综合应用设计说明

2.1 题目

0. 在关系数据库中，所有数据对象都以表的形式存储，如需在关系数据中存储树结构，需设置一指向父节点的属性来实现，该过程可以通过如下线性表节点的存储结构模拟：

```
struct Node {  
    int id; //该线性表中所有节点的 ID 都唯一  
    ElemType data; //节点的值，自定义数据类型  
    int pid; //表示该节点的父节点，0 表示根，对应线性表中其他节点的 id 值  
    Node *next; // 指向线性表下一节点  
}
```

- (1) 请设计一算法，根据线性表中 pid 指向，将该线性表中存储的节点转换为树形结构。
- (2) 在树结构中插入一节点，自动将其加入到线性表中。
- (3) 在树结构中删除一节点，自动更新线性表的结构。

2.2 软件功能

根据题目的要求，本软件实现的基本功能以及对应的实现方法如下：

- ① 正确的链表结构：灵活运用 C++ 知识，自行实现单链表类；
- ② 正确的树形结构：灵活运用 C++ 知识，选择合适的表示方式，自行实现树类；
- ③ 链表向树结构转化：自行设计算法，实现链表向树结构的单向转化；
- ④ 同步的插入、删除操作：将树类中的算法与链表关联，实现同步的更新；
- ⑤ 异步的插入、删除操作：链表类中的算法独立，链表更新时树结构不更新；
- ⑥ 数据可视化：使用 QT 中的 QTreeView、QTableView 类，设计算法实现数据展示；
- ⑦ 面向对象的操作方式：设计菜单工具栏，方便用户的对表和树操作；

此外，为了使用户的操作更加流畅、链表与树之间的关联更加突出，程序在最终实现时，还额外增加了以下功能：

- ① 选择演示题目：在演示窗口之前增加“选择窗口”，整合课设的两个演示程序；
- ② 同步、异步的修改操作：为两结构设计算法，允许直接修改两者的数据；
- ② 显示数据基本信息：通过 QT 中的 QLabel 类，实时更新链表与树的对应关系；
- ③ 自定义树的结构：为孩子兄弟表示法、孩子表示法设计相应的可视化算法；
- ④ 提供用户操作指南：设计“帮助”工具栏和对应弹窗，方便用户查看操作说明；
- ⑤ 提供操作结果的信息：使用 QT 中的 QMessageBox 类结合底层数据结构的算法；

最终设计的综合实验演示程序，正确实现了构造、维护、展示一个单链表形式的数据库以及其对应的树结构的功能，同时也实现了用户交互的功能，便于进行插入、删除、修改、清空、转化等操作，还可以自行设置树结构展开的形式。

2.3 设计思想

软件在设计时采用瀑布模型，将功能设计与 UI 设计分离。在分析功能需求后，根据需求选择合适的数据结构，并完成相应的控制台应用(完成功能设计)。后期根据实际需求，为控制台应用和底层的数据结构添加 GUI 界面，实现用户交互，但数据结构和程序内核基本不变。以下将从“数据结构选择”、“功能设计”与“UI 设计”三个方面分别说明。

2.3.1 数据结构选择

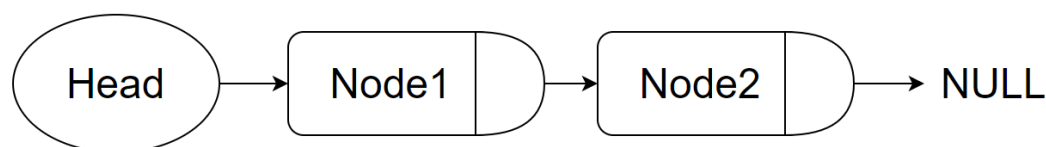
分析题目要求，软件需要设计两种数据结构：线性表和树形结构。

(1) 线性表：单链表(带头节点)

线性表的结构选择相对简单。根据题目中所给线性表结构可知，单个表的节点有指向下一个节点的指针，即线性表之间的节点依靠指针相连，所以选择单链表这一数据结构，模拟关系数据库中的数据。带头节点是为了简化算法，避免操作时对头节点的讨论。

单链表是采用链式存储结构的线性表，它用一组任意的存储单元存储线性表的数据元素(存储单元可以是连续的，也可以是不连续的)，所以对任意数据元素，除了存储其本身的信息外，还需存储其直接后继的存储位置。单链表的结构特点，决定了它在内存占用上具有更多的灵活性，同时也决定了节点之间的正确联系十分重要。也因为这一特点，链表在执行插入等与节点之间联系相关的操作时会更加方便一些，但在调用单个节点时会比较复杂，不易直接找到需要调用的特定元素。因此，链表会更适合处理一些不经常直接调用特定数据，经常增删数据，占用内存较大的数据，而关系数据库正符合这一要求。

该结构下的线性表大体可以用下图表示：(具体细节见 2.4)



(Head 中不存储数据)

(2) 树形结构：孩子兄弟表示法(独立根节点，链式存储，节点中外加指向父节点的指针)

相对线性表，树形结构的选择就需要综合考查许多因素。首先分析关系数据库中的树的表示：所有数据对象都以表的形式存储，存储时线性表节点会设置一指向父节点的属性。根据题目描述，关系数据库使用单链表形式的双亲表示法表示树，树的关联靠一个 int 型 pid 数据建立。再分析需要完成的主要操作：树中插入节点的操作需要频繁的定位、建立关系；树中删除节点操作需要频繁的定位、快速确定子树和双亲、断开联系；表向树转化也需要频繁的定位和建立关系；考虑可视化操作，我们还希望选择的结构可以快速确定树的所有孩子。

综合分析以上内容，可见选择的树结构需要具有以下特点：①灵活的内存占用 ②易于建立和断开节点联系 ③快速的节点定位(易于遍历) ④根据父节点快速定位子节点 ⑤根据子节点快速定位父节点 ⑥简洁明了的算法和较高的效率。现讨论以下三种基本表示法：

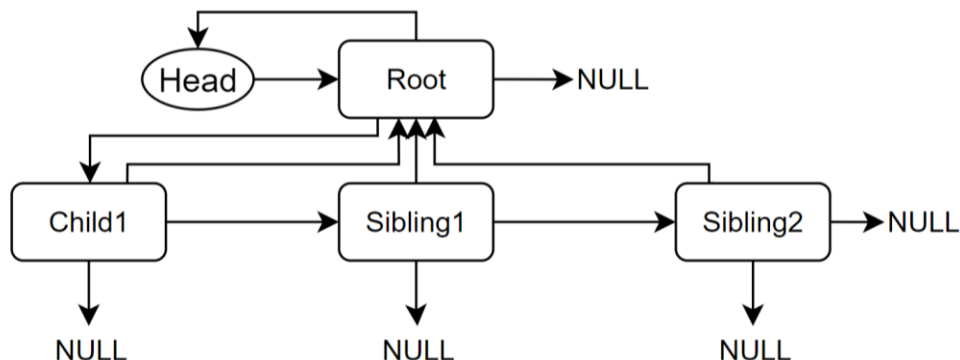
a. 双亲表示法：虽然容易定位父节点，但是在针对孩子的操作时效率很低，频繁的删除操作明显不适合用双亲表示法表示，因此也不必再讨论双亲表示法下的物理结构；

b. 孩子表示法：容易找到所有子节点，若在节点中添加父节点指针，也容易定位到父节点，建立与断开连接也相对容易。再讨论其物理结构，由于节点的孩子数层次不齐，如果采用单个节点的孩子指针开辟固定内存，很容易造成内存浪费或溢出；如果动态申请连续内存，建立树的过程又将消耗大量的时间；依靠指针建立连接，则可以有效避免这些问题。再考虑链表形式下的孩子表示法的算法，算法的简洁性和效率有待考查；

c. 孩子兄弟表示法：容易找到所有子节点，若在节点中添加父节点指针，也容易定位到父节点，建立与断开连接也相对容易。再讨论其物理结构，每个节点大小固定，只有左右两个孩子的指针。同时作为二叉树的结构，算法成熟简洁，操作简单，效率高；

最后，结合题目需求和个人喜好，选择使用孩子兄弟表示法完成树结构。节点之间链式存储，依靠指针建立连接，同时在单个节点中添加指向父节点的指针，便于父节点的提取。此外，还为树结构设立独立的根节点(不存储数据，左孩子指向数据的根)，以此来简化算法。

该结构下的树结构大体可以用下图表示：(具体细节见 2.4)



2.3.2 功能设计思想

本软件的功能设计，主要是单链表和树结构(孩子兄弟表示的二叉树)的操作的设计。由于单链表和二叉树的算法都很基础，这里只简要介绍各个操作的实现思路和算法流程。

(1) 单链表的操作：

带有头节点的单链表，插入、修改和删除操作无需讨论节点是否是头节点，算法较为简洁。以上操作最重要的输入判断，分析题目要求，可以推断出以下几条隐性数据要求：

① id 值唯一(id 不等于 0)，且 id 与 pid 不能相同；

② pid 为 0 的节点只有一个；

因此，在表中插入、删除、修改数据时，都需要时刻注意以上几点。此外，由于表中的单条数据相对独立，没有树结构的显性关系，所以每一条数据都应当理解为独立的个体，即可以单条插入，也可以单条删除，不必删除所有的子节点。

链表的插入操作先为待插入节点开辟内存，之后判断 id、pid 是否合法，不合法则输出错误提示，并返回 0；合法则直接将新节点插入到表头，返回值为 1，时间复杂度为 $O(1)$ 。链表的删除操作先判断删除 id 是否合法，不合法则输出错误提示，返回值为 0；合法则通过移动指针找到并删除元素，返回值为 1，时间复杂度为 $O(n)$ ；链表的修改操作先判断待修改的 id 以及修改后的数据是否合法，不合法则输出错误提示，返回值为 0；合法则通过移动指针找到并修改元素，返回值为 1，时间复杂度为 $O(n)$ 。此外，链表中还设计了“判断是否是树”的功能，辅助链表转化树结构，该功能的实现会在树的操作中讨论。

(2) 树结构的操作：

带有根节点的二叉树，插入、修改和删除操作无需讨论节点是否是头节点，算法较为简洁。操作中的输入判断基本同单链表，但结合树结构的特点，还要有以下额外的数据要求：

① 节点的父节点的 id 不能不存在；

因此，树的插入操作只能从根开始，逐个插入，新增节点需要是原有节点的子节点。这点在操作中会有一些不便，所以可以先在表中输入所有数据，再通过转化的操作转化为树。

树的插入操作先为待插入节点开辟内存，之后判断 id、pid 是否合法，不合法则输出错误提示，并返回 0；合法则将新节点插入到父节点下，返回值为 1，时间复杂度为 $O(n)$ 。树的删除操作先判断删除 id 是否合法，不合法则输出错误提示，返回值为 0；合法则通过移动指针找到并删除元素，返回值为 1，时间复杂度为 $O(n)$ ；树的修改操作先判断待修改的 id 以及修改后的数据是否合法，不合法则输出错误提示，返回值为 0；合法则通过移动指针找到并修改元素，返回值为 1，时间复杂度为 $O(n)$ 。

链表转化树结构的算法可以分为以下两个部分：①判断表中的数据能否成为树 ②将表中的数据转化为树结构。在判断的过程中，主要考核表中的数据是否有且仅有一个根节点，且从唯一的根节点出发能否遍历所有的子节点，该算法以递归的形式实现，体现在链表结构中的 isTree() 函数中。转化的过程也是递归实现，两个递归对应的代码如下：

```
// 将线性表转化为树
bool Tree::Trans(pTree& f, LinkList* list){ // f为父节点指针
    // 连接父节点和子节点
    Node* p = list->head->next;
    for (int i = 0; i < list->length; i++) {
        if (p->pid == f->id) {
            pTree temp = new TreeNode();
            if (!temp) {
                QMessageBox::critical(new QWidget, "错误", "内存损坏");
                return 0;
            }
            this->size = this->size + 1;
            temp->data = p->data;
            temp->id = p->id;
            temp->pid = p->pid;
            temp->fChild = temp->nSibling = NULL;
            temp->father = f;
            temp->nSibling = f->fChild;
            f->fChild = temp;
        }
        p = p->next;
    }
    // 递归子节点
    pTree child = f->fChild;
    while (child != NULL) {
        Trans(child, list);
        child = child->nSibling;
    }
    return 1;
}

// 递归计算子树个数
int LinkList::CntChild(const int pid) const{
    QVector<int> children;

    // 存储父节点为pid的子节点
    Node* p = this->head->next;
    for (int i = 0; i < this->length; i++) {
        if (p->pid == pid)
            children.push_back(p->id);
        p = p->next;
    }

    // 向下搜索计数
    int cnt = 0;
    for (int i = 0; i < children.size(); i++) {
        cnt = cnt + CntChild(children[i]);
    }
    return cnt + children.size();
}
```

2.3.3 UI 设计思想

本软件的 UI 设计，主要包括以下几个部分：

- ① 选择界面和帮助界面：选择综合实验、B-树或帮助文档；
- ② 软件操作的工具栏：插入、修改、删除、转化、帮助、返回的工具栏设计；
- ③ 操作对应的独立弹窗：插入、修改、删除、帮助均需要弹窗；
- ④ 表结构、树结构的显示窗口：以表格的形式显示链表，以文件树的形式显示树；
- ⑤ 对树结构显示的指令按钮：选择显示方式、展开/收起的按钮；
- ⑥ 数据的基本信息：树中节点个数、表中节点个数，以及表能否转化为树；

以下将按照本程序整体的操作流程，简要介绍各个模块 UI 的设计思想和算法流程。

(1) 选择界面和帮助界面：

设计选择界面时，经过反复的斟酌与考查，最终选择模仿“佳能打印机功能选择”程序的界面和操作方法(如右图)：

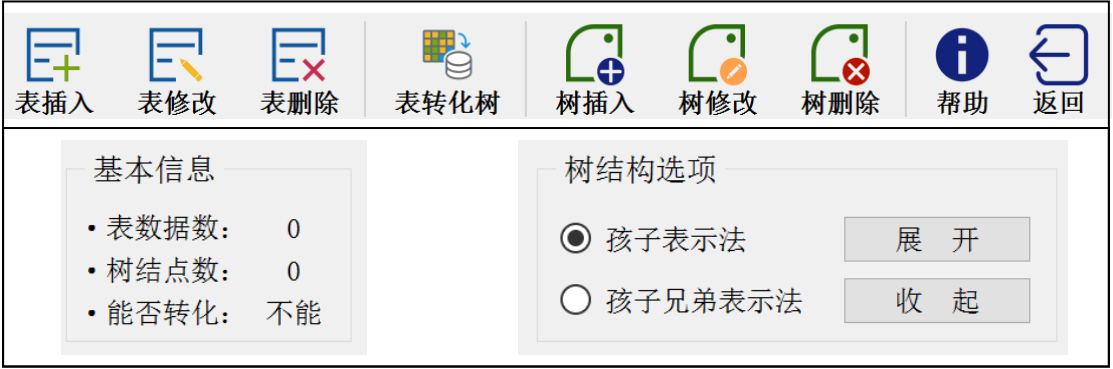
仅提供选项工具栏和必要信息，窗口以横排小窗呈现，不可放大且工具栏固定。单机工具栏选项后，该窗口自动隐藏，弹出所选功能子窗口，从而模拟窗口切换的效果(实际只是隐藏，并未关闭)。



帮助界面的设计主要模仿了 office 软件中常见的各类设置子窗口，使用的 Tab 窗口的方式，管理不同演示程序的帮助文档。帮助界面的具体样式请参照 1.6 或 1.7 中程序截图。

(2) 工具栏和其他指令按钮：

各工具栏、指令按钮与对应的功能绑定，样式与 office 常用工具栏、window 常见按钮一致。为了区分各个按键，特别添加了分类框(QGroupBox 类)，将不同的操作按钮打包分离，从而增加界面的美观性，让程序结构更加鲜明；此外，程序还添加有选择框样式按钮，将输入框打包在弹窗之中，简化主界面；基本信息文本框也使用了 QGroupBox 进行分类，内容更新与表和树同步。其中工具栏的大部分 UI 图标，均是使用 Adobe AI 自主设计，效果如下：



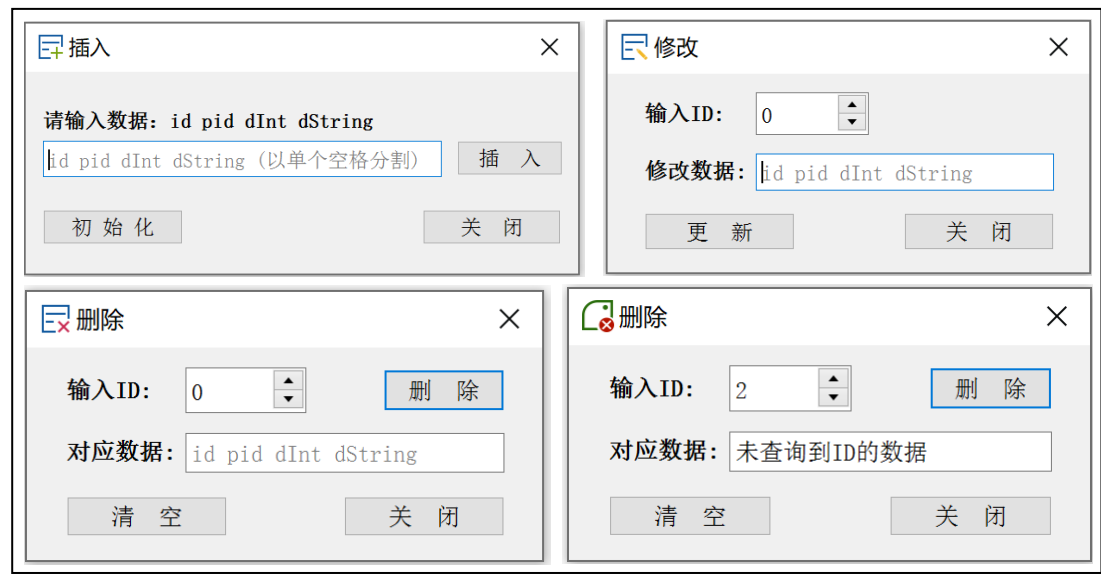
工具栏除普通的功能操作外，还有如下的内在逻辑：①帮助工具栏随时可以启用；②单独的执行表操作后，树操作工具栏禁用；③表成功转化为树后，所有工具栏启用。以上逻辑是为了保证树视图、表视图的正常运行，也符合题目要求中的内在逻辑。

(3) 工具栏弹窗

在 UI 设计中，每个工具栏的弹窗都有独立的类的设计，但它们都是 `QDialog` 的派生类。其中，帮助弹窗是以模态方式显示，即用 `QDialog::exec()` 函数显示，该状态下显示的窗口不允许再运行其他窗口，直到对话框退出。而插入、删除和修改弹窗则采用非模态函数显示，使用 `QDialog::show()` 函数。非模态显示的对话框在显示后继续运行主程序，也可以在主程序中操作，两者之间可以实现交互控制，变化的数据也可以实时显示在主窗口当中。

此外，为减少用户负担，还在“插入”窗口中，加入了“初始化”选项，为用户自动填充数据，方便用户使用；在“删除”窗口中，加入了“清空”选项，为用户自动删除所有的数据，方便用户插入新数据。在“修改”和“删除”窗口中，当前栏的信息会随着用户输入的 `id` 自动更新，方便用户查看自己要删除、修改的数据，避免出错。

各个弹窗的运行样式如下：



(4) 表和树结构的显示：

为了正确、美观的显示树结构，这里采用 Model/View 结构，树结构使用 QT 的 `QTreeView` 视图类和 `QStandardItem` 模型类，表结构使用 QT 的 `QTableView` 视图类和 `QStandardItem` 模型类，并通过设计的对应的算法将底层数据转化为数据模型，最后通过 `QTreeView`、`QTableView` 来显示。采用这样的设计结构，可以将界面组件和编辑的数据分离开来，又通过数据源的形式连接起来，较好的实现了表和树结构的可视化。

树的数据转化为两种视图的算法如下：

```
// 孩子表示法画树
void DSWindow::drawTreeChildList(QStandardItem *fItem, pTree f){
    // 从f的头孩子开始
    pTree child = f->fChild;
    while(child){
        QStandardItem *cItem;
        // 根据有无孩子判断是否是叶子
        if(child->fChild){
            cItem = new QStandardItem(QString::number(child->id).append("->结点"));
            cItem->setIcon(iconNode);
        }
        else{
            cItem = new QStandardItem(QString::number(child->id).append("->叶子"));
            cItem->setIcon(iconLeaf);
        }
        // 为父节点添加孩子结点
        fItem->appendRow(cItem);
        fItem->setChild(cItem->index().row(), 1, new QStandardItem(QString::number(child->data.dInt)));
        fItem->setChild(cItem->index().row(), 2, new QStandardItem(child->data.dStr));
        // 递归孩子结点添加
        drawTreeChildList(cItem, child);
        // 继续下一个孩子
        child = child->nSibling;
    }
}
```

(树的孩子表示)

```
// 孩子兄弟法画树
void DSWindow::drawTreeChildSibling(QStandardItem *fItem, pTree f, int& i){
    // 左孩子
    if(f->fChild){
        i = i + 1;
        QStandardItem *cItem;
        cItem = new QStandardItem(QString::number(f->fChild->id).append("->").append(QString::number(i)).append("代-孩子"));
        cItem->setIcon(iconChild);
        fItem->appendRow(cItem);
        fItem->setChild(cItem->index().row(), 1, new QStandardItem(QString::number(f->fChild->pid)));
        fItem->setChild(cItem->index().row(), 2, new QStandardItem(QString::number(f->fChild->data.dInt)));
        fItem->setChild(cItem->index().row(), 3, new QStandardItem(f->fChild->data.dStr));
        drawTreeChildSibling(cItem, f->fChild, i);
        i = i - 1;
    }
    // 右兄弟
    if(f->nSibling){
        QStandardItem *cItem;
        cItem = new QStandardItem(QString::number(f->nSibling->id).append("->").append(QString::number(i)).append("代-兄弟"));
        cItem->setIcon(iconSibling);
        fItem->appendRow(cItem);
        fItem->setChild(cItem->index().row(), 1, new QStandardItem(QString::number(f->nSibling->pid)));
        fItem->setChild(cItem->index().row(), 2, new QStandardItem(QString::number(f->nSibling->data.dInt)));
        fItem->setChild(cItem->index().row(), 3, new QStandardItem(f->nSibling->data.dStr));
        drawTreeChildSibling(cItem, f->nSibling, i);
    }
}
```

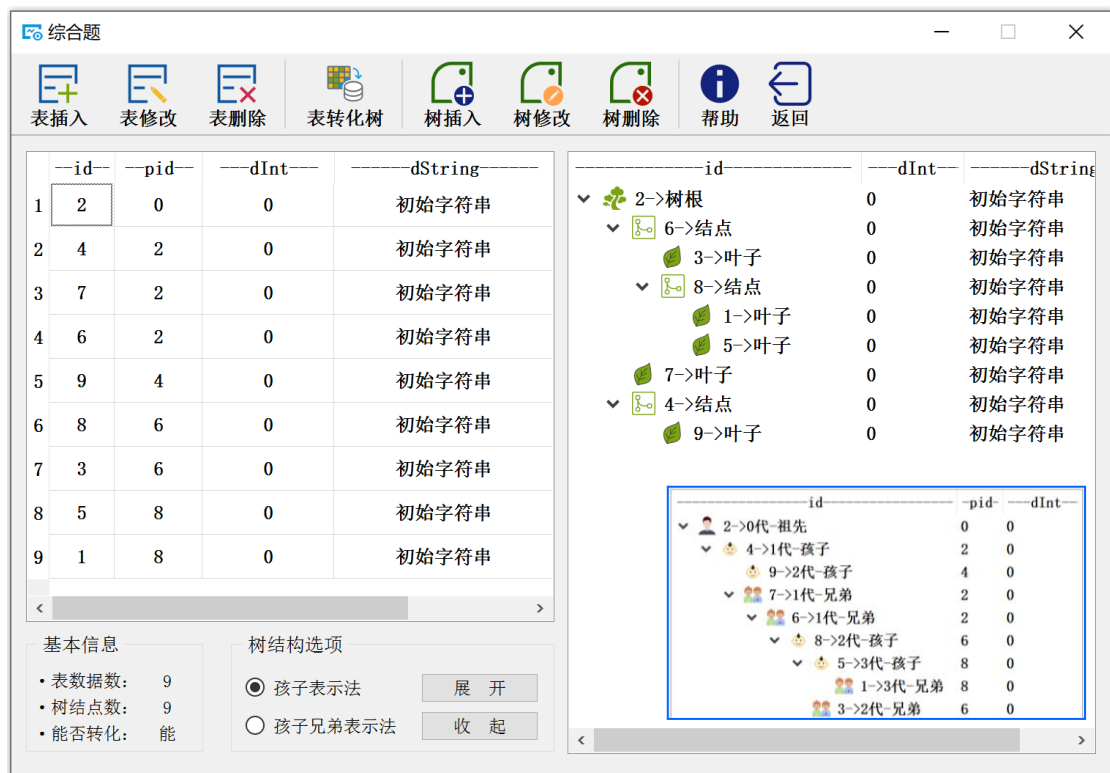
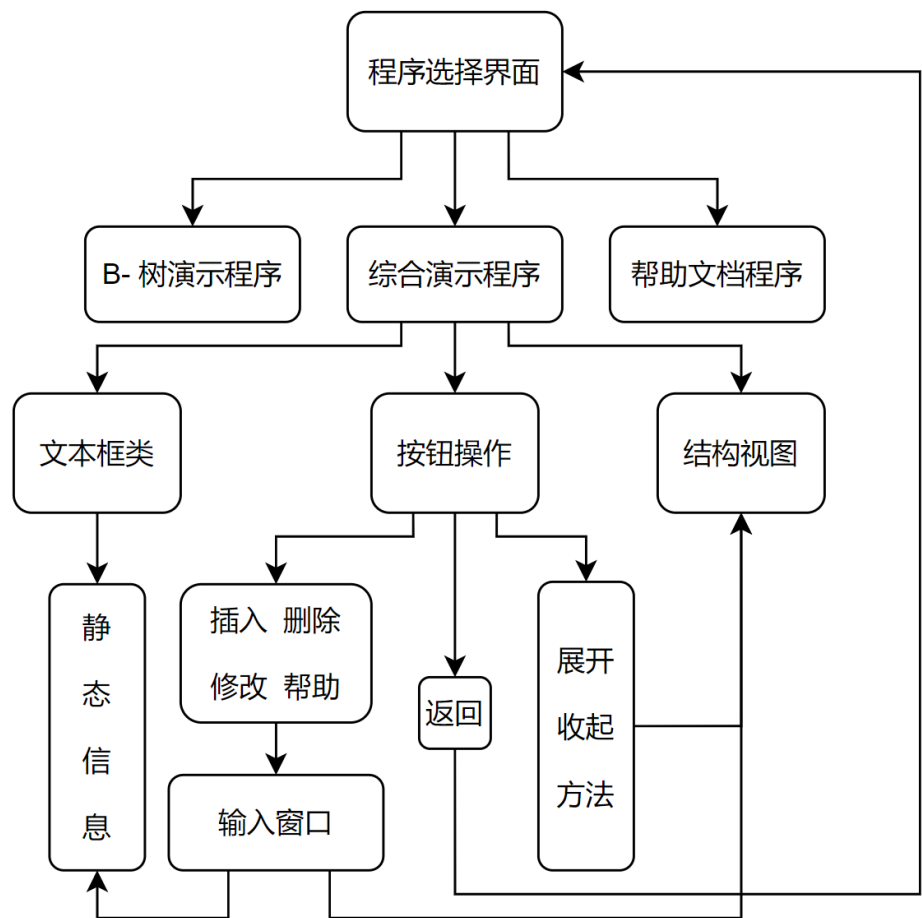
(树的孩子兄弟表示)

函数 drawTreeChildList(QStandardItem *fItem, pTree f)中，fItem 是已经建立好的父项，存在于 QStandardItem 中，对应树的一个节点的父节点。f 是当前在树中的节点，函数要绘制的也是它的节点树(把当前 f 转化为 Item，并插入到父项当中，建立关联)。

函数 drawTreeChildSibling(QStandardItem *fItem, pTree f, int& i)中，fItem 是已经建立好的父项，f 是当前在树中的节点，i 是当前节点所处的代际，便于用户在孩子兄弟表示法中，快速找到一个节点的兄弟姐妹。

以上两个递归函数，同时使用在 updateTreeView() 函数中，用来更新树的视图。该函数会在树的数据修改、视图方式变化、表转化树激活的时候触发，以实现树结构的实时更新和显示，具有较好的延展性和美观度。

(5) 程序整体流程图和 UI 效果图：



2.4 逻辑结构与物理结构

2.4.1 逻辑结构

本次综合实验演示程序的使用到的逻辑结构主要有：树形结构(孩子兄弟表示法)、线性结构(单链表)。此外还有部分辅助结构，如 QVector 的向量线性结构、用来存储数据的 Elem 结构体等。

(1) 单链表：

单链表是典型是线性结构，可以实现快速插入、删除，不可以随机读取。本次题目设计的 LinkList 类的结点由 Node 结构体构成，具体的设计如下：

<pre>// 线性表节点结构体 typedef struct Node { int id; // 该线性表中所有节点的 int pid; // 表示该节点的父节点， Elem data; // 节点的值，自定义数据 Node* next; // 指向线性表下一节点 }Node;</pre>	<pre>class LinkList : public QObject { Q_OBJECT public: Node* head; // 表头 int length; // 表长 bool iRoot; // 是否有根节点 };</pre>
--	--

单链表的特点和操作在 2.3 中已经详细讨论，这里不再赘述。为了方便操作，在 LinkList 类中有许多成员函数，这里没有展示。各个操作和成员函数的内容，可以参照文档 2.3 的功能设计，也可以在源代码对应的 linklist.h 和 linklist.cpp 中查看。

(2) 树形结构：

孩子兄弟表示法是树形结构的常见表示法之一，可以以二叉树的形式表示普遍的树。在本软件中，树结构的节点有 TreeNode 结构体构成，具体的设计如下：

<pre>// 树结构节点结构体 typedef struct TreeNode { int id; // 该线性表中所有节点的 ID int pid; // 表示该节点的父节点，值为 Elem data; // 节点的值，自定义数据类型 TreeNode* father; // 父节点指针 TreeNode* fChild; // 头孩子指针 TreeNode* nSibling; // 右兄弟指针 }TreeNode, * pTree;</pre>	<pre>class Tree : public QObject { Q_OBJECT public: pTree root; // 根节点 int size; // 节点数 };</pre>
---	--

Elem 结构体是存储数值的结构体，在软件的设计中，暂定是由 int dInt 和 QString dStr 两个成员组成。使用者可以自己在源代码中修改，根据实际的需求，加入更多的数据类型，之后对插入、删除、修改的窗口内容稍作修改即可。

孩子兄弟法表示的树结构的特点和操作在 2.3 中已经详细讨论，这里不再赘述。为了方便操作，在 Tree 类中有许多成员函数，这里没有展示。各个操作和成员函数的内容，可以参照文档 2.3 的功能设计，也可以在源代码对应的 tree.h 和 tree.cpp 中查看。

2.4.2 物理结构

本软件中，单链表结构、树形结构均使用链式存储的方式，即各个节点在物理内存上的映射并不像相连，节点与节点之间由许多分散的内存通过各节点的指针域相关联，不可以随机读取与存储。

选择这样的物理结构符合数据库数据量大、删插频繁的特点，也符合树形结构一对多的特征。只要有剩余的内存，就可以随时添加新的子节点，不需要预先开辟大量空间导致内存浪费，也不会出现因为预先开辟的内存不够而需要重新开辟空间或发生溢出，而父子节点之间的指针联系更能表征树的特点。

此外，QVector 类(函数中辅助操作时用到)使用顺序存储的方式，即在物理内存上的映射相连，是完整的区域，节点之间为一对一的关系，可以实现随机读写。函数中使用这样的物理结构存储中间数据，便于简化代码量。

2.5 开发平台

本次软件的开发平台以及运行环境如下：

开发平台	Qt 平台(版本 6.1.2)
操作系统	Windows 10 专业版 21H1
编程语言	C++, xml (Designer 辅助生成)
开发工具	Visual Studio 2019, Qt Creator 4.15.2 (Community), Designer 6.1.2
打包工具	Qt Creator 4.15.2 (Community), Enigma Virtual Box
核心码库	Qt 6.1.2 (MSVC 2019, 64-bit), C++标准库
运行环境	Win x64

2.6 系统的运行结果分析说明

2.6.1 调试开发过程

在开发过程中，首先使用 Visual Studio 2019 设计综合题目的底层数据结构，通过控制台输出线性表、孩子兄弟表示树的构造结果，同时配合设置断点、std::cout 输出、查看变量等方式进行调试，最终完成数据结构综合课设的控制台应用。

在设计完成底层逻辑之后，使用 Qt Creator 4.15.2 设计 GUI 程序。本人在 UI 设计过程中首先设计主窗口界面，即综合题目的大窗口 QMainWindow 类以及窗口对应的.ui 文件。

之后，在主窗口和工具栏都搭建好后，继续设计 4 个弹出的对话框(QDialog 类)，实现“插入”、“删除”、“修改”以及“帮助”的交互界面和交互逻辑。

D:\Ray\VisualStudio\C++\AG_Final_or\Debug\AG_Final_or.e

创建

Create LinkList!

Create Tree!

测试

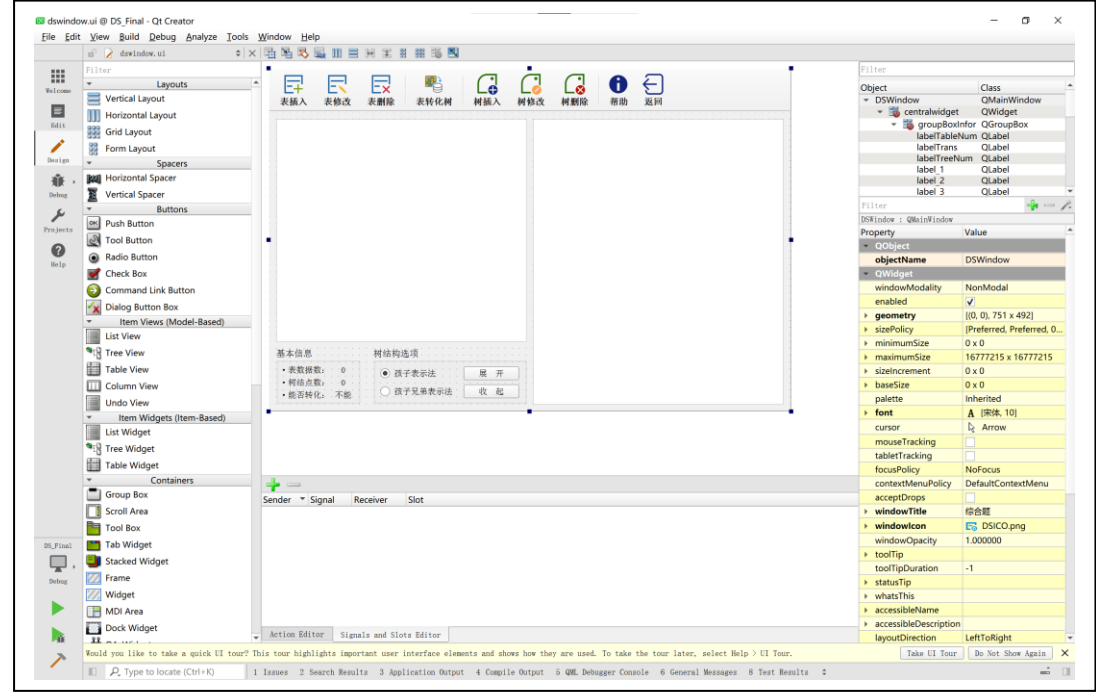
表结点插入-0 表结点更新-1 表结点删除-2
表数组删除-3 表树型删除-4 表转树结构-5
树结点插入-6 树结点更新-7 树结点删除-8
输出两结构-9 退出进程-10及以上

0 2 0 1 o
• 表结点插入: - 完成
0 4 2 1 o
• 表结点插入: - 完成
0 7 2 1 o
• 表结点插入: - 完成
0 6 2 1 o
• 表结点插入: - 完成
0 9 4 1 o
• 表结点插入: - 完成
0 8 6 1 o
• 表结点插入: - 完成
0 3 6 1 o
• 表结点插入: - 完成
0 5 8 1 o
• 表结点插入: - 完成
0 1 8 1 o
• 表结点插入: - 完成
5
• 表转树结构: - 完成

• 表转树结构: - 完成
8 8
• 树结点删除: 是头孩子
P.S. 删除后父节点6的孩子: 3 - 完成
6 8 6 1 o
• 树结点插入: - 完成
7 6 6 7 1 o
• 树结点更新: - 完成
9
• 输出两结构: • 表结构:
7 1 1
8-6-1-o
3-6-1-o
9-4-1-o
6-7-1-o
7-2-1-o
4-2-1-o
2-0-1-o

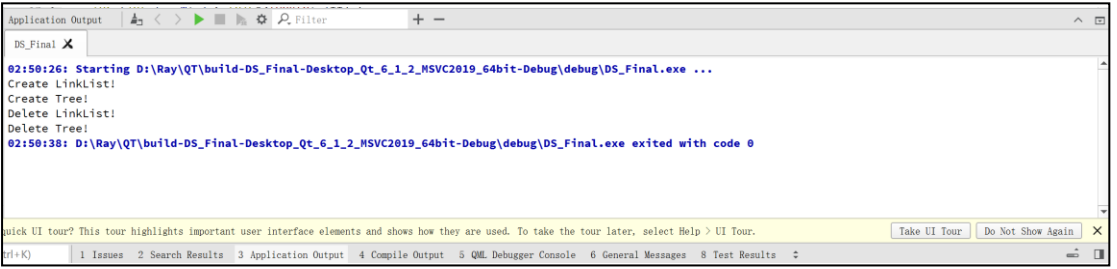
• 树结构:
7
0-0 2-0 4-2 9-4 |
7-2 6-7 8-6 |
3-6 |
|
|
|

由于 Qt Creator 同时配备着 Designer 6.1.2 作为设计工具，因此.ui 文件中的 xml 代码，主要通过 Designer 的可视化设计自行生成。UI 文件设计完成后，在单个窗口类中手动连接信号与槽函数，同时建立主窗口与对话框的连接，并完善可视化设计无法完成的部分，例如窗口的稳定、窗口选项的删除、TableView、TreeView 视图的菜单栏设计等。此过程可以通过运行程序直观调试。



窗口界面设计完成后，需要引入底层结构，实现数据向视图的转化。此过程中，为了统一文件的命名空间，选择将所有 std 容器 (vector 类) 转化为 Qt 容器 (QVector 类)，从而实现代码的美观统一。由于 GUI 程序需要更多的交互，因此在这个环节的实现了，需要向底层结构添加部分触发信号，也需要时刻关注底层数据的变化。在调试过程中，本人结合 qDebug() 函数与 QMessageBox 类输出程序运行情况，并通过设置断点、查看状态栏等方式寻找程序内部错误。

具体操作中，曾遇到 LinkList 类 isTree() 成员函数未使用 const 修饰，导致程序窗口在调用树相关函数时，过早调用链表析构函数消除了有效数据，使程序提示修改空指针，无法正常运行。面对这样的情况，自己反复通过 qDebug() 函数，寻找空指针发生的位置，以及递归函数的参数引用情况，最终确定了是过早调用析构函数的问题，修改后可以正常运行。这样的问题还遇到过很多，不再一一列举。



2.6.2 软件成果展示

最终实现的软件完整且超额实现了题目要求，正确实现了插入、删除、转化的功能，且额外添加了对数据修改的功能。在后期多组数据的检测过程中，本程序都体现出了较好的容错能力与运行速度，拥有多样的错误提示和人性化的工具栏管理，也不曾出现崩溃或异常退出的情况，实时显示的链表数据和树结构数据也一直符合要求。可以说，软件在功能上具有较好的正确性、稳定性、容错能力以及延展性。

同时，树视图选择以文件树的形式反馈，也符合 window 应用程序的特色。树形结构特有的两种表示方式下 (孩子表示法、孩子兄弟表示法)，树视图中每个节点的提示信息均不相同，可以让用户更加快速的分辨每个节点在树结构中的位置。

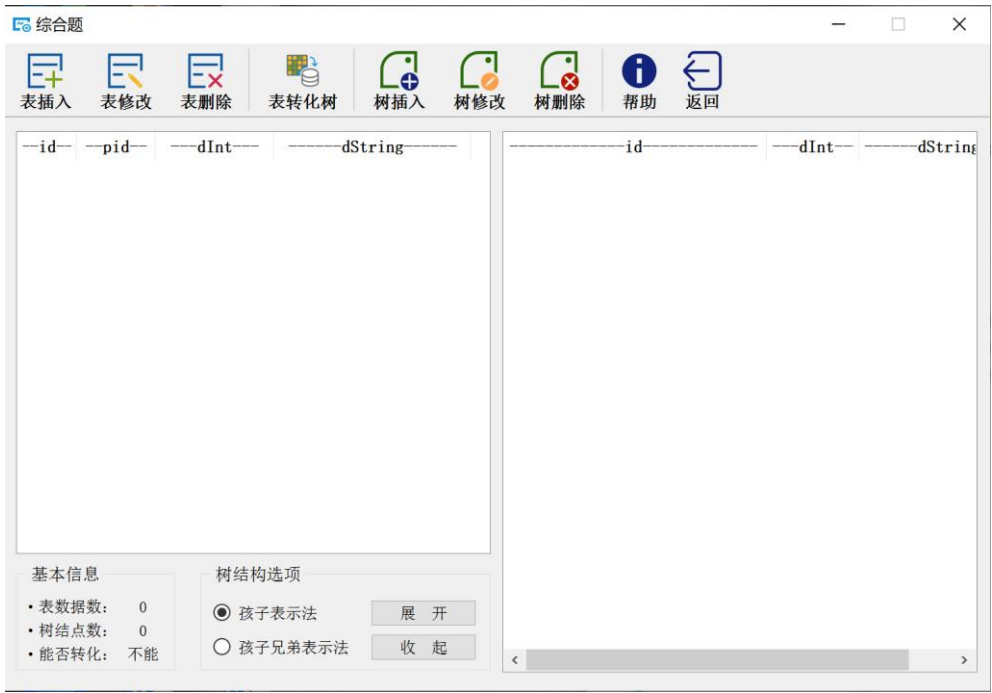
此外，本软件的 UI 设计采取简约风，指令工具栏齐全且分类规范，视图窗口美观且提示信息友好。值得一提的是，窗口内使用的所有 UI 图标，均是通过 Adobe AI 软件自行设计，整体上风格统一、样式清新，色彩选择也符合大众认知和艺术审美，具有良好的视觉效果。而对话框的设计也充分考虑了从简原则，简化了用户操作步骤，方便实现大规模数据操作。

2.6.3 运行结果案例

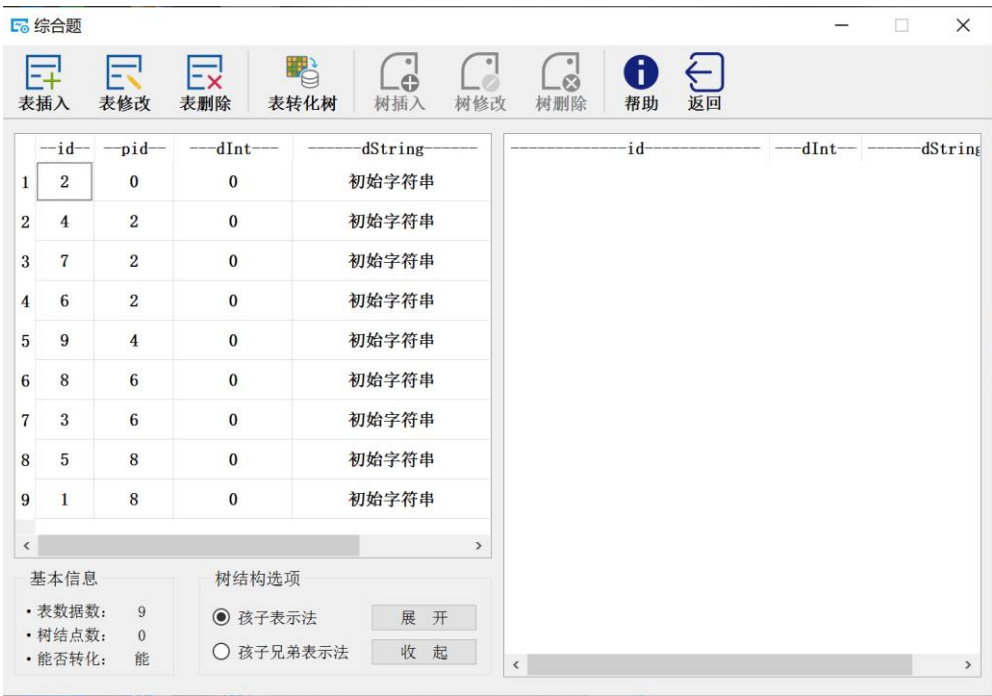
进入程序后，初始界面如图所示，共 3 中程序可供选择，此界面不可放大。



选择综合实践演示程序后，跳转至如下图窗口。初始状态下表和树中没有数据：



这里先在表中处理数据，表中插入、删除或修改数据后，树的操作将暂时不可用：



在选择表转化树后，如果转化成功，树的操作将恢复为可用状态，以下是两种展示：

综合题

表插入

表修改

表删除

表转化树

树插入

树修改

树删除

帮助

返回

	id	pid	dInt	dString
1	2	0	0	初始字符串
2	4	2	0	初始字符串
3	7	2	0	初始字符串
4	6	2	0	初始字符串
5	9	4	0	初始字符串
6	8	6	0	初始字符串
7	3	6	0	初始字符串
8	5	8	0	初始字符串
9	1	8	0	初始字符串

基本信息

表数据数: 9

树结点数: 9

能否转化: 能

树结构选项

☒ 孩子表示法

展开

☐ 孩子兄弟表示法

收起

	id	dInt	dString
2->树根	0	初始字符串	
6->结点	0	初始字符串	
3->叶子	0	初始字符串	
8->结点	0	初始字符串	
1->叶子	0	初始字符串	
5->叶子	0	初始字符串	
7->叶子	0	初始字符串	
4->结点	0	初始字符串	
9->叶子	0	初始字符串	

	id	pid	dInt
2->0代-祖先	0	0	
4->1代-孩子	2	0	
9->2代-孩子	4	0	
7->1代-兄弟	2	0	
6->1代-兄弟	2	0	
8->2代-孩子	6	0	
5->3代-孩子	8	0	
1->3代-兄弟	8	0	
3->2代-兄弟	6	0	

之后，处理树的数据，表的数据将会同步更改，如下在树中删除 id 为 8 的节点，由于已经删除，所以删除对话框中提示未查询到该 id，可见软降自动将 id 为 8 的节点及其对应的子树全部删除，正如 window 删除文件夹后文件也自动删除(如果不想删除子树，需要在表中执行删除操作)。还可见的是，当一个对话框弹出时(除帮助对话框)，工具栏中只有帮助可用，以免用户需要操作提示，而帮助对话框打开时，所有工具栏均不可用：

综合题

表插入

表修改

表删除

表转化树

树插入

树修改

树删除

帮助

返回

	id	pid	dInt	dString
1	2	0	0	初始字符串
2	4	2	0	初始字符串
3	7	2	0	初始字符串
4	6	2	0	初始字符串
5	9	4	0	初始字符串
6	3	6	0	初始字符串

基本信息

表数据数: 6

树结点数: 6

能否转化: 能

树结构选项

☒ 孩子表示法

展开

☐ 孩子兄弟表示法

收起

	id	dInt	dString
2->树根	0	初始字符串	
4->结点	0	初始字符串	
9->叶子	0	初始字符串	
7->叶子	0	初始字符串	
6->结点	0	初始字符串	
3->叶子	0	初始字符串	

删除

输入ID: 8

删除

对应数据: 未查询到ID的数据

清空

关闭

由于该软件对话框过多，以下是处理几组实际数据后的演示结果，用户可以自行测试：

综合题

表插入 表修改 表删除 表转化树 树插入 树修改 树删除 帮助 返回

	id	pid	dInt	dString
1	2	0	0	Life
2	4	6	0	初始字符串
3	7	2	0	初始字符串
4	6	2	0	初始字符串
5	3	6	0	初始字符串

基本信息

- 表数据数: 5
- 树结点数: 5
- 能否转化: 能

树结构选项

☒ 孩子表示法 展开

☐ 孩子兄弟表示法 收起

插入

请输入数据: id pid dInt dString

id pid dInt dString (以单个空格分割) 插入

初始化 关闭

综合题

表插入 表修改 表删除 表转化树 树插入 树修改 树删除 帮助 返回

	id	pid	dInt	dString
1	2	0	0	Life
2	4	6	0	初始字符串
3	7	2	0	初始字符串
4	6	2	0	初始字符串
5	3	6	0	初始字符串
6	9	4	9	My
7	8	7	10	lif

错误

父结点ID不存在

OK

综合题

表插入 表修改 表删除 表转化树 树插入 树修改 树删除 帮助 返回

	id	pid	dInt	dString
1	2	0	0	Life
2	4	6	0	初始字符串
3	7	2	0	初始字符串
4	6	2	0	初始字符串
5	3	6	0	初始字符串
6	9	4	9	My
7	8	7	10	lif
8	10	13	899	没有父结点的节点

基本信息

- 表数据数: 8
- 树结点数: 7
- 能否转化: 不能

树结构选项

☒ 孩子表示法 展开

☐ 孩子兄弟表示法 收起

插入

请输入数据: id pid dInt dString

id pid dInt dString (以单个空格分割) 插入

初始化 关闭

错误

表中数据不能转化为树

OK

2.7 操作说明

为了加快用户录入、修改、阅读数据的速度，本程序的所有对话框，在展示或输入节点信息时，只提供简单的一个文本栏作为输入框，信息格式为：id pid dInt dString。其中：

- ① id, pid, dInt 都是整型数据，dString 是任意字符串数据；
- ② 每个数据之间，依靠一个空格字符分开；
- ③ 整型数据如果其他字符，会默认输入为 0；
- ④ id 是节点的身份表征，值唯一且不为 0；
- ⑤ pid 是节点的父节点 id，不与本身的 id 相同，且只能有一个节点的 pid 为 0；
- ⑥ dInt 和 dString 是存储在节点中的数据，用户可以随意输入；

在以上要求中，要求①②③不会加以输入判断。这样设计主要是考虑到要方便用户输入大量的数据，如果为每个数据单独设计输入框，用户可能要耗费大量时间切换数据框，反而影响了用户的体验。

对于工具栏，所有工具栏点击即可弹出对话框，在之后的操作说明中不再赘述。

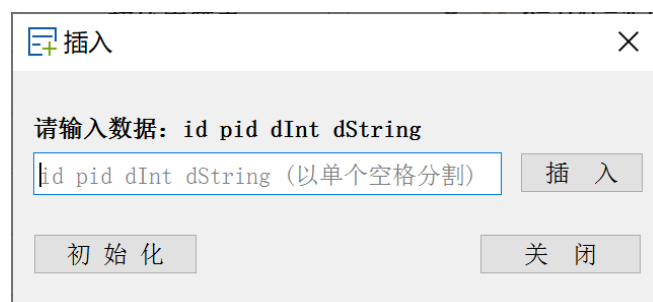
2.7.1 线性表操作

线性表的“插入”、“修改”、“删除”、“表转化树”图标如下，该工具栏长期可用：



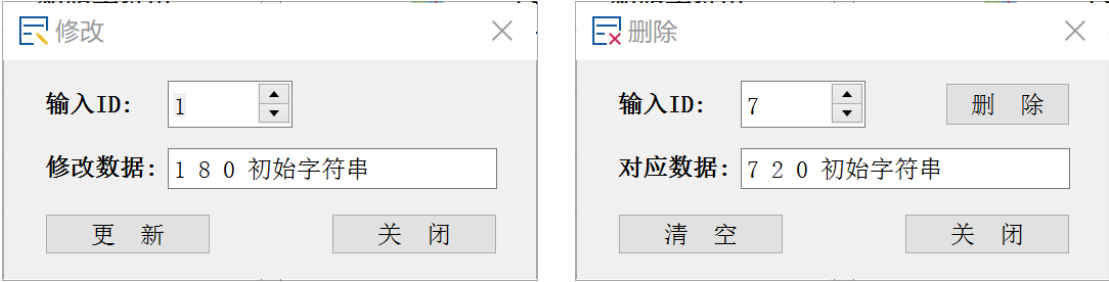
在“表插入”对话框中，用户需要在文本框中按照要求输入待插入节点的信息，之后可以单击按键“插入”或按回车键(Enter 键)实现数据插入，插入的信息会实时更新在主窗口的表中，而树结构不发生变化。同时，树结构的三个操作也会出于安全考虑被暂时禁用。

插入后，可以再次输入下一条数据，继续插入操作。单击左下角“初始化”按键，表中数据将被清空且填入原始数据，以减少用户的操作。该窗口可以通过点击右下角“关闭”或右上角“X”关闭。



在“表修改”对话框中，用户需要在滚动栏中输入待修改节点的 id。id 输入之后，下方“修改数据”的文本框会自动填充当前节点的信息，以便用户修改。在按照输入要求，修改文本框中信息之后，可以单击按钮“插入”或按回车键(Enter 键)实现数据修改。修改后，可以再次输入下一个 id，继续修改操作。

修改的信息会实时更新在主窗口的表中，而树结构不发生变化。同时，树结构的三个操作也会出于安全考虑被暂时禁用。该窗口可以通过点击右下角“关闭”或右上角“X”关闭。



在“表删除”对话框中，用户需要在滚动栏中输入待修改节点的 id。id 输入之后，下方“对应数据”的文本框会自动填充当前节点的信息，以便用户查看。之后可以单击按钮“删除”或按回车键(Enter 键)实现单个节点的删除，删除的信息会实时更新在主窗口的表中，而树结构不发生变化。同时，树结构的三个操作也会出于安全考虑被暂时禁用。

删除后，可以再次输入下一个 id，继续删除操作。单击左下角“清空”按钮，表和树中的数据将被同步清空。该窗口可以通过点击右下角“关闭”或右上角“X”关闭。



“表转化树”操作没有对话框。如果表中的数据可以转为树结构数据(只有一个根节点，且每个节点的父节点都存在)，点击后，树视图更新，数据与表同步，树的操作全部解锁。若表中数据不能转化，会给予提示，树的操作保持不可使用。

2.7.2 树结构操作

树结构的“插入”、“修改”、“删除”图标如下，该工具栏在表中数据被单独修改后禁用：



树的操作对话框与表操作的完全相同，但处理方式不同之处有：

- ① 处理树中节点的同时，表的信息被同步处理；
- ② “树插入”中，待插入节点的 pid 必须存在，即：只能为存在的节点添加子节点；
- ③ “树删除”中，删除节点时，连通该节点的子树一同删除；

2.7.3 树视图选项

在“树结构选项”对应栏目下，可对视图中树的展示方式进行展开、收起、表现形式的切换。用户可以按照下方指南完成对应操作：



- A. 以孩子表示法显示：单击选项 A，勾选“孩子表示法”；
- B. 以孩子兄弟表示法显示：单击框格 B，勾选“孩子兄弟表示法”；
- C. 展开树结构：单击按钮①；
- D. 收起树结构至根节点：单击按钮②；

2.7.4 其他操作

用户可以单击“返回”工具栏，返回选择界面，返回后输入的信息不会丢失。用户也可以单击“帮助”工具栏，打开帮助界面，阅读了解操作的含义和软件的使用方法。



第三部分 实践总结

3.1. 所做的工作

3.1.1 底层数据结构

本次课程实践当中完成的工作较多，最主要的工作之一是底层数据结构的设计。在算法题目当中，通过学习 B-树的知识，最终使用 C++ 完成了 B-树结构的设计；在综合应用题目当中，结合相关论文和大二上学期所学数据结构的知识，完成了单链表以及孩子兄弟表示树的设计。此外，根据题目的要求，数据结构相关的操作算法也是设计的重点之一。

在 B-树的设计过程中，由于课堂上对 B-树的拓展限于理论，而编写相关代码的经验较少，因此在设计相关算法时遇到了一定困难。为了使软件拥有坚实而牢固的底层元素，在参考许多资料、学习许多代码之后，经过多次的尝试与试错，基本上完成了 B-树的操作算法。

对于单链表与树形结构，虽然这两种数据结构在课程当中多有练习，但应对实际的情况还有许多的算法操作需要灵活改动。在针对实际问题的设计过程中，我也重新认识了许多数据结构的概念，学到了许多新的知识，对数据结构与算法的关系有了更深入的了解。

3.1.2 平台与 UI

除了底层的数据结构，在本次的课程实践当中，还自学了许多 QT 平台和 UI 设计的知识，最终使用 QT 完成了应用的可视化设计，同时使用部分商业软件完成了软件图标和 UI 的设计。

在平台的学习过程当中，本人花费大量的时间，参考了许多书籍与资料，前期通过写一些简单的应用程序磨练自己的技能，以在正式的实践当中不产生畏惧心理，最终较好的完成了本次课设的可视化交互设计。

在 UI 的设计方面，结合平时对一些商业软件的了解，本次软件中的 UI 图标和界面排版布局，都是在参考大量的现有优秀作品后，自行使用相关软件设计完成的。其中，界面的布局由 Qt Designer 辅助设计完成，而程序当中使用的图标，则使用 Adobe AI 软件手工绘制而成。可以说，本次的课程实践不仅磨练了自己平台使用与 GUI 程序设计的能力，也提高了自身的审美、打磨了设计技能。这些在所做的工作中都有所体现。

3.1.3 后期调试与拓展

后期的调试与功能的拓展也是本次课程设计的重要工作之一。在程序完成之后，需要大量的数据对程序的准确性、稳定性、容错性、实用性进行系统的测试，也需要考虑实际情况，

结合数据结构的特点，对设计的应用进行一些功能的拓展，以更好的体现题目的要求、体现底层数据结构的正确性以及选用结构的优势所在。因此，除底层数据结构、平台学习和 UI 设计之外，本次的课程实践中，本人在后期应用调试以及功能拓展也付出了巨大的努力。

总而言之，在本次的课程设计当中，自己通过 C++ 完成了底层数据结构和相关算法的设计，并结合 QT 平台和相关商业开发软件完成了应用程序的可视化以及交互设计，最后经由后期调试和功能拓展，使程序不断完善，最终实现了较好的运行效果。以上就是本次课设中完成的主要工作。

3.2. 总结与收获

3.2.1 课设总结

本次的特色课程设计历时一个多月。在这一个多月时间里，通过作业的督促和兴趣爱好的驱使，自己对 QT 开发平台进行了系统的了解，并在短时间内上手开发，顺利完成了课程设计的 GUI 程序。在这个过程当中，自己对 B-树的结构特点、树与二叉树的转化关系、数据库中线性表与树结构的关系等多个方面的底层数据结构的知识，有了更加深入的了解，也对他们的算法有了更加深刻的认识。通过实践，将之前学习的理论知识与实际应用相结合，也让我认识到理论与实际之间的差距和联系。

除了对数据结构有更深刻的认知外，本次课设还提升了我的学习能力和论文研读能力。通过自主学习 B-树的相关知识、阅读有关关系数据库的相关论文，自己从一开始对学术文章敬而远之，到课设结束后能够大胆的找论文、读论文、复现论文，可以说，课程设计为我的今后的学术道路做好了铺垫，打好的基础。

本次课设还提升了个人的文档阅读、资料查找、资料学习能力。网上关于 QT 的辅助资料十分丰富，但质量参差不齐，大部分因为版本过旧，已经不再兼容如今 Qt 6.1 版本的特点。为此，综合考虑未来趋势，自己没有选择捷径：按照众多的参考书或参考资料，使用 QT 5.0 进行开发。而是通过阅读 QT 官方的英文文档，结合先前的收集资料择优阅读，最终以文档为主、资料为辅，在短时间内快速上手的 QT 6.1.2。在此过程当中，我逐渐了解了文档阅读的重要性，也掌握了阅读文档的诀窍和要领。相信在之后的各种平台的学习当中，有了这次学习 QT 平台的经验，其他平台的学习也会更加顺利。

此外，本次的课设中，自己的审美能力和设计能力也有一定的提高。UI 的设计、界面的布局，以及图标的选择，这些在我们日常的教学当中提到的少之又少，但却是实际应用当

中必不可少的开发环节。本次的课设，督促着自己提高个人审美能力，也激发了我对于设计的兴趣，通过学习 Adobe AI 等设计工具，自己的能力得到了很大的提升。

可以说本次的课程设计让我受益匪浅，相信之后的学习生涯有本次课程设计积累的经验，会更加的顺利。

3.2.2 心得体会

本次的课程设计个人有许多感触。首先是惊叹于开发平台的多样性。在选择可视化平台时，Visual C++、Qt、py 的 tkinter 以及其他的开发平台、可视化库让人眼花缭乱，不知该何从下手。这也让我意识到，在计算机技术百花齐放的今天，我们虽然不可能全部精通，但也必须要熟练一两种平台开发手段，有一技之长才能在编程的路上越走越远。

第二，让我感触很深的还有如今技术更新迭代的速度之快。从 2017 年的 QT 5.9，到如今的 QT 6.2，短短几年时间内，各大编译器、核心库都进行了大量重要的更新，但许多的技术指导、开发教材却止步不前。在这样的情况下，我们必须要提高个人的资料搜索能力和快速的学习能力，不能坐等他人把知识传授给你，必须要主动出击，赶上时代的步伐。

第三，我也体会到了课程教学与实际应用的差距和联系所在。课堂上所学的算法与数据结构都是基础，在实际的应用当中还有许多的变式。而我们只有打好了基础，才能在实际开发当中拥有更强的灵活变通能力。此外，对以往知识的复习也很重要，我们需要经常的练习才能避免遗忘。算法、数据结构等这些未来我们技术开发的重要基石必须要牢牢掌握。此外，实践出真知。代码的编写、平台的使用不能只是纸上谈兵，在脑中感觉会写的，就认为自己实际真的会了。代码的设计和能力的提高必须要在电脑上把代码敲出来，只用运行正确了，才说明自己可能学会了。

感谢这次的课设，让我意识到了自己的不足，也意识到了技术更新之快和不断学习的重要性。今后我会打牢基础，同时也自主学习一些新的平台和核心的开发框架，让自己不至于被时代落伍淘汰，在竞争当中更具优势。当然，在本次的课程当中，个人也存在着许多的不足。对 QT 平台虽然学到了很多知识，但依然有许多未曾涉足的功能，等待着我去进一步的学习和探索。本次课设中遇到的问题与不足，将会激励着我在下一步的学习生活当中更加精益求精，不断前进。

第四部分 参考文献

- [1] 邓宏涛. 关系数据库中树型结构信息的处理方法研究[J]. 江汉大学学报(自然科学版), 2010, 38(02): 50-53.
- [2] 许孝元, 杨继赢. 关系数据库中树形数据结构的处理[J]. 计算机工程与应用, 1997(07): 9-12.
- [3] 洪熹. 树型数据结构中递归算法的实现[J]. 福建电脑, 2012, 28(06): 124-126+155.
- [4] 叶军伟. 树型数据结构存储方法的探讨[J]. 科技创业家, 2014(07): 155.
- [5] 毛法尧. B 树分析[J]. 计算机工程与应用, 1995, 31(2): 22-24.
- [6] 魏小亮, 蔡弘. B-树/B+树的批量插入算法[J]. 中央民族大学学报(自然科学版), 2001(01): 57-61.
- [7] Johnson T, Sasha D. The performance of current B-tree algorithms[J]. ACM Transactions on Database Systems (TODS), 1993, 18(1): 51-101.
- [8] 严蔚敏, 吴伟民. 数据结构: C 语言版[M]. 清华大学出版社有限公司, 1997.
- [9] 徐孝凯, 王凤禄. 数据结构简明教程[M]. 清华大学出版社有限公司, 2005.
- [10] Blanchette J, Summerfield M. C++ GUI programming with Qt 4[M]. Prentice Hall Professional, 2006.
- [11] Krajewski M. Hands-On High Performance Programming with Qt 5: Build cross-platform applications using concurrency, parallel programming, and memory management[M]. Packt Publishing Ltd, 2019.
- [12] 陆文周. Qt5 开发及实例[M]. 2014.
- [13] 王维波, 栗宝鹃, 侯春望. Qt 5.9 C++开发指南[M]. 人民邮电出版社, 2018. 5.