数据结构课程设计报告

学　　号：20211003337

班级序号：220211

姓　　名：曾康慧

指导教师：杨之江　曾文

成　　绩：

中国地质大学地理与信息工程学院

2024 年 12 月

目 录

一．总体介绍

本次课程设计需要包括基础题和综合题两部分。基础题旨在考查结构课程中重点章节的理解和实践能力，综合题旨在考查学生对课程内容的灵活运用能力。

二．基础题

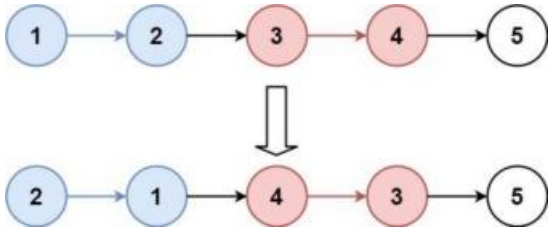
2.1 K 组翻转链表【LintCode 450】

2.1.1 问题描述

给你链表的头节点 head ，每 K 个节点一组进行翻转，请你返回修改后的链 表。其中， K 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 K 的整数倍，那么请将最后剩余的节点保持原有顺序。

要求: 不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

【样例输入】



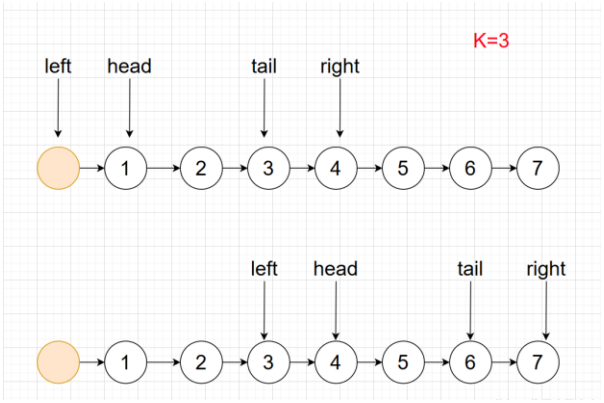
head = [1,2,3,4,5], K = 2

【样例输出】

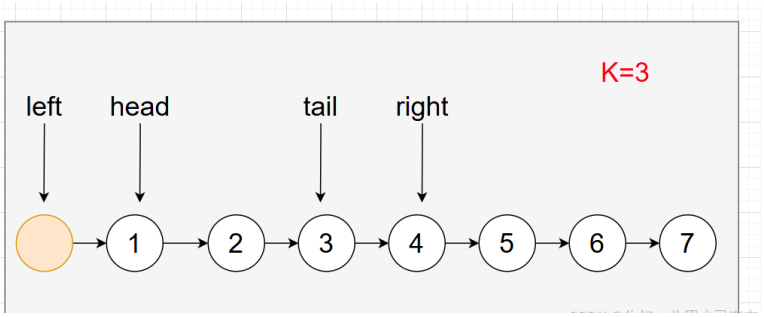
[2,1,4,3,5]

2.1.2 算法思想

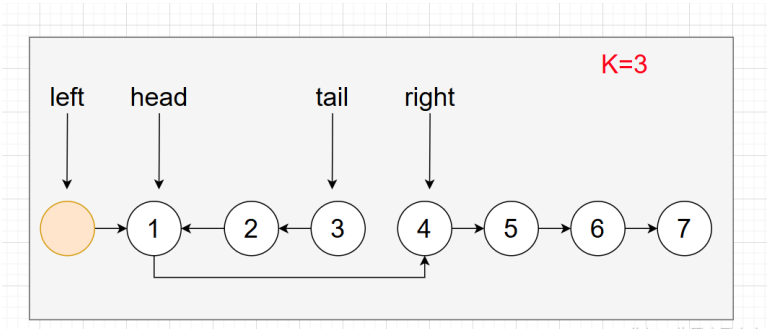
1. 将链表中的 k 个节点视为一组数据
2. 遍历链表，对划分的每组数据进行链表翻转
3. 如果某组数据的节点个数不足 k 个，则不进行任何操作
4. 问题在于，我们如何定义一组数据，或者说，我们如何定义区间。
5. 本题中，我们使用开区间进行定义，如下图所示，假如 k=3，则我们要进行链表翻转的区间范围应该是[head,tail]，left 指针指向 head 的前一个节点，right 指向 tail 的后一个节点，用于每次链表翻转后，重新将翻转后的链表更新链接起来。



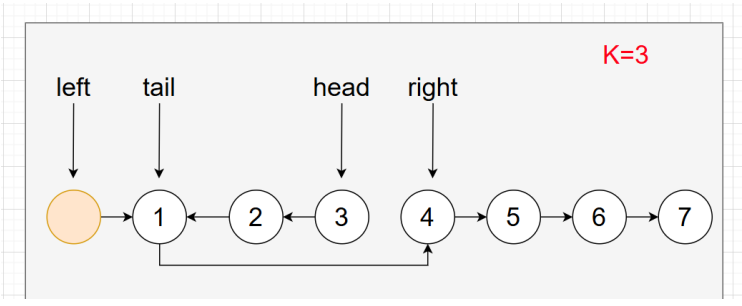
寻找符合条件的 k 个节点，作为一组数据



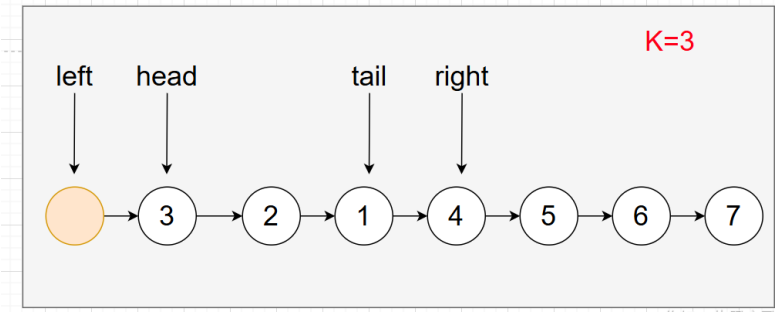
链表翻转



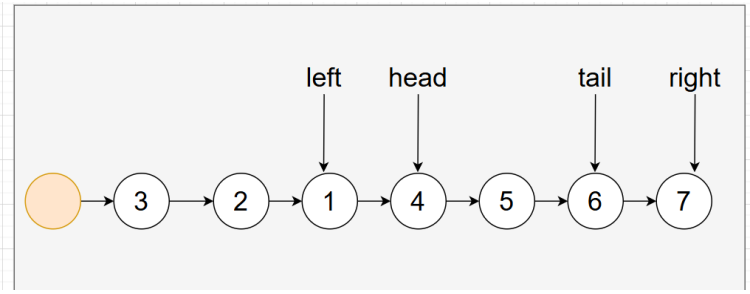
更新翻转后的链表的首尾节点位置

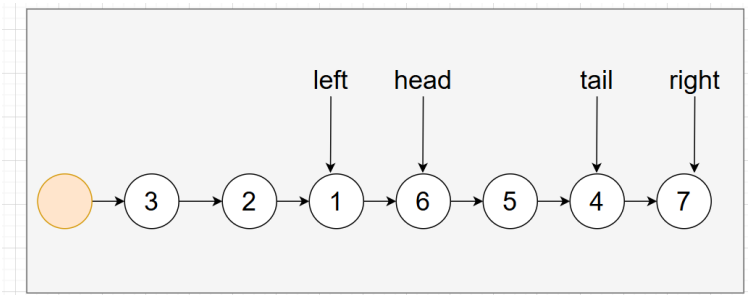


重新将翻转后的链表链接到原链表上



处理下一组链表





2.1.3 问题与解决方法

问题：不清楚链表元素的反转。

解决方法：为了实现链表元素的反转，我们只需要修改元素的指针指向，如此遍历完整个链表，即可完成链表的反转。

2.1.4 运行结果

2.1.5 算法分析

时间复杂度为 O(n∗K) 最好的情况为 O(n) 最差的情况未 O(n2)

空间复杂度为 O(1) 除了几个必须的节点指针外，我们并没有占用其他空间

2.2 最大矩形【LintCode 510】

2.2.1 问题描述

（<小四号宋体>，简要描述题干）

给定一个仅包含 0 和 1 、大小为 Rows * Cols 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

【样例输入】

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

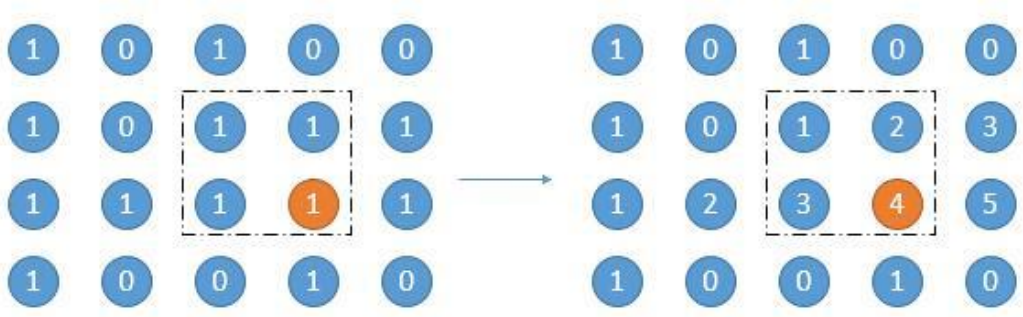
matrix =
[["1","0","1","0","0"],["1","0","1","1","1"],
["1","1","1","1","1"],["1","0","0","1","0"]]

【样例输出】

6

2.2.2 算法思想

本题只需要求出以每一行作为底最大的矩形是多少，每一行都有一个 height 数组，利用单调栈，每次更新 height 数组，height 数组代表的是这一列上面有多少个连续的 1，即矩形的高度，以每一行作为底(直方图最下面)时最大矩形面积，然后记录最大值即可。



- 初始化 dp 数组，用 dp 数组记录当前位置上方有多少个连续 1
- 对于每一行作为底，利用单调栈求高度，寻找最大的底乘高
    - 这个栈是从栈底到栈顶依次是从小到大的
    - 如果栈中的数比当前的数大(或者等于)就要处理栈顶的(记录左右

两边的比它小的第一个数)

- 然后如果遍历完之后，单独处理栈，此时所有元素右边都不存在比它小的
- height[j]表示目前的底上(第 1 行)，j 位置往上(包括 j 位置)有多少连续的 1

- 不断更新最大面积

2.2.3 问题与解决方法

直接寻找矩形是有点麻烦的。计算机程序不像人眼，可以直接获取到图形相关的信息，计算机不行，只能获得单个位置的信息。所以我们让程序直接判断矩形是不现实的，但我们可以通过特征点来锁定矩形，这个也是业内常用的套路。

锁定一个矩形的方法一般有两种，第一种是用矩形的中心点和长宽来确定。这一种在各种图像识别和目标检测算法当中经常用到，模型预测的结果就是图像中心点的坐标以及长宽的长度。

第二种方法可以通过矩形的对角线上的两个点来确定，这种方法只适用于和坐标轴平行的矩形。比如下图当中，无论我们知道了(x2, y2), (x3, y3)还是(x1, y1), (x4, y4)，我们都可以将这个矩形确定下来。

2.2.4 运行结果

2.2.5 算法分析

时间复杂度 O(m*n)：取决于 dp 的 O(n*m)和单调栈的 O(n)

空间复杂度 O(n*m)：取决于 dp 的大小

2.3 二叉搜索树中最接近的值 II【LintCode 901】

2.3.1 问题描述

给定一棵非空二叉搜索树以及一个 target 值（浮点数），要求设计一个算法， 找到 BST 中最接近给定 target 值的唯一的 k 值集合。

【测试用例】

输入：

{3,1,4,#,2}

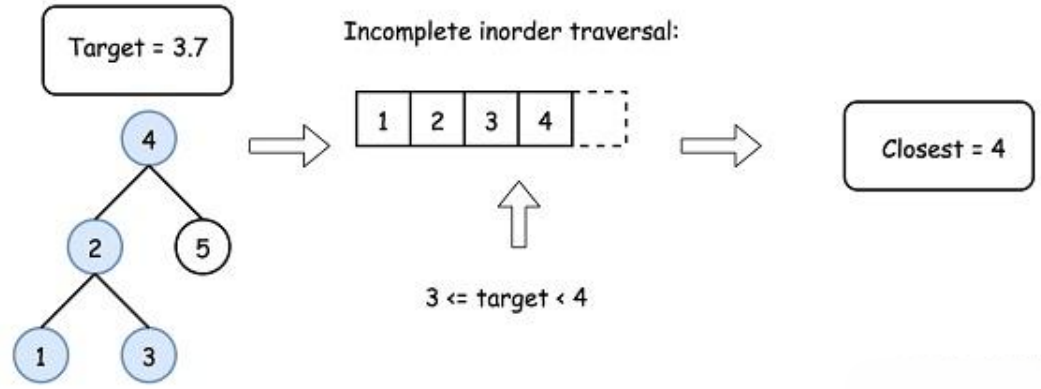
0.275000　　//Given target value is a floating point

2　　//You may assume k is always valid, that is: k ≤ total nodes 二叉树{3,1,4,#,2}，表示如下的树结构：

```
  3
/   \
1   4
  \
  2
```

输出：[1,2]

2.3.2 算法思想



使用迭代的方法：

如果最接近的元素的索引远小于树的高度，那么我们可以进行优化。

首先，我们可以一边遍历树一边搜索最接近的值。

第二，我们可以再找到了最

接近的值以后立即停止遍历。若目标值位于两个有序数组元素之间 nums[i]<=target<nums[i+1]，则说明我们找到了最接近的元素。

算法：

- 初始化一个空栈和设 pred 为一个很小的数字。
- 当 root 不为空：
- 为了要迭代构建一个中序序列，我们要尽可能的左移并将节点添加到栈中。
- 弹出栈顶元素：root = stack.pop()。
- 若目标值在 pred 和 root.val 之间，则最接近的元素在这两个元素之间。
- 设置 pred = root.val，且向右走一步 root = root.right。
- 如果我们在循环过程中无法找到最接近的值，这意味着最接近的值是顺序遍历中的最后一个值，返回它。

2.3.3 问题与解决方法

问题：不知道改用什么方法来确定 target 在树的左边还是右边。

解决方法：利用二分查找的思想，在中序遍历时直接比较

2.3.4 运行结果

（<小四号宋体>，给出 LintCode 代码运行通过的清晰截图和简要说明）

2.3.5 算法分析

时间复杂度：平均情况下 O(k)，最坏的情况下是 O(H+k)，其中 k 是最接近元素的索引。

空间复杂度：最坏的情况是树为非平衡树的情况，栈需 O(H) 的空间。

2.4 课程表 II【LintCode 616】

2.4.1 问题描述

你需要去上n 门课才能获得offer，这些课被标号为 0 到 n-1 。有一些课 程需要“前置课程”，比如如果你要上课程0，你需要先学课程1，我们用一个匹 配来表示他们：[0,1]。要求：给你课程的总数量和一些前置课程的需求，返回 你为了学完所有课程所安排的学习顺序。可能会有多个正确的顺序， 你只要返回 一种就可以了。如果不可能完成所有课程，返回一个空数组。

【测试用例】

输入：

n = 2, prerequisites = [[1,0]] 输出：

[0,1]

输入：

n = 4, prerequisites = [1,0],[2,0],[3,1],[3,2]] 输出：

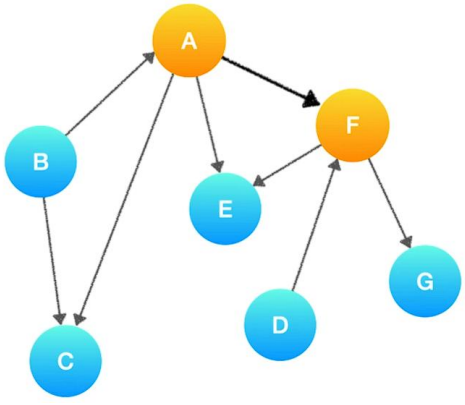
[0,1,2,3] or [0,2,1,3]

2.4.2 算法思想

本题是一道经典的「拓扑排序」问题。

我们将每一门课看成一个节点；

- 如果想要学习课程 A 之前必须完成课程 B，那么我们从 B 到 A 连接一条有向边。这样以来，在拓扑排序中，B 一定出现在 A 的前面。
- 求出该图的拓扑排序，就可以得到一种符合要求的课程学习顺序。



深度优先搜索思路:

我们可以将深度优先搜索的流程与拓扑排序的求解联系起来，用一个栈来存储所有已经搜索完成的节点。

对于一个节点 u，如果它的所有相邻节点都已经搜索完成，那么在搜索回溯到 u 的时候，u 本身也会变成一个已经搜索完成的节点。这里的「相邻节点」指的是从 u 出发通过一条有向边可以到达的所有节点。

假设我们当前搜索到了节点 u，如果它的所有相邻节点都已经搜索完成，那么这些节点都已经在栈中了，此时我们就可以把 u 入栈。可以发现，如果我们从栈顶往栈底的顺序看，由于 u 处于栈顶的位置，那么 u 出现在所有 u 的相邻节点的前面。因此对于 u 这个节点而言，它是满足拓扑排序的要求的。

这样以来，我们对图进行一遍深度优先搜索。当每个节点进行回溯的时候，我们把该节点放入栈中。最终从栈顶到栈底的序列就是一种拓扑排序。

算法:

对于图中的任意一个节点，它在搜索的过程中有三种状态，即：

「未搜索」：我们还没有搜索到这个节点；

「搜索中」：我们搜索过这个节点，但还没有回溯到该节点，即该节点还没有入栈，还有相邻的节点没有搜索完成）；

「已完成」：我们搜索过并且回溯过这个节点，即该节点已经入栈，并且所有该节点的相邻节点都出现在栈的更底部的位置，满足拓扑排序的要求。

通过上述的三种状态，我们就可以给出使用深度优先搜索得到拓扑排序的算法流程，在每一轮的搜索搜索开始时，我们任取一个「未搜索」的节点开始进行深度优先搜索。

我们将当前搜索的节点 u 标记为「搜索中」，遍历该节点的每一个相邻节点 v：

如果 v 为「未搜索」，那么我们开始搜索 v，待搜索完成回溯到 u；

如果 v 为「搜索中」，那么我们就找到了图中的一个环，因此是不存在拓扑排序的；

如果 v 为「已完成」，那么说明 v 已经在栈中了，而 u 还不在栈中，因此 u 无论何时入栈都不会影响到 (u,v) 之前的拓扑关系，以及不用进行任何操作。

当 u 的所有相邻节点都为「已完成」时，我们将 u 放入栈中，并将其标记为「已完成」。

在整个深度优先搜索的过程结束后，如果我们没有找到图中的环，那么栈中存储这所有的 n 个节点，从栈顶到栈底的顺序即为一种拓扑排序。

2.4.3 问题与解决方法

问题：使用广度优先搜索还是深度优先搜索

解决方法：深度优先搜索是一种「逆向思维」：最先被放入栈中的节点是在拓扑排序中最后面的节点。我们也可以使用正向思维，顺序地生成拓扑排序，这种方法也更加直观。

2.4.4 运行结果

2.4.5 算法分析

时间复杂度: O(n+m)，其中 n 为课程数，m 为先修课程的要求数。这其实就是对图进行深度优先搜索的时间复杂度。

空间复杂度: O(n+m)。题目中是以列表形式给出的先修课程关系，为了对图进行深度优先搜索，我们需要存储成邻接表的形式，空间复杂度为 O(n+m)。在深度优先搜索的过程中，我们需要最多 O(n) 的栈空间（递归）进行深度优先搜索，并且还需要若干个 O(n) 的空间存储节点状态、最终答案等。

三．综合题

3.1 英语词典软件（搜索树，15 分）

3.1.1 问题描述

英语词典是一种将传统的印刷词典转成数码方式、进行快速查询的学习工　具，成为 21 世纪学生学习生活的掌上利器。在很多实际应用中，动态索引结构在 文件创建或初始装入记录时生成，在系统运行过程中插入或删除记录时，为了保 持较好的检索性能，索引结构本身将随之发生改变。教材上已经介绍的动态查找 数据结构包括：二叉搜索树（BST）、平衡二叉树（AVL）、红黑树（RBT）、B-树。 本题要求选取一种已经学过的动态搜索树结构，设计并实现一个基于四级词汇的英语词典软件。

【基本要求】

一个完整的英语词典软件应具有以下功能：

（1）支持词库数据的导入、导出操作， 外部数据采用 TXT 格式。注意： 文本文件打开一次后，下次不用再装载。

（2）支持英文单词的添加、修改、删除等功能；

① 添加操作要防止重复输入；

② 修改或删除某个单词的信息，在操作前，需先进行查找定位。

（3）支持英文单词的各种查询操作，具体包括：

① 逐条翻看

能按照英文单词的顺序，显示所有的单词记录，支持分页滚动查看。

② 单词查找

输入英文单词，给出其汉语意思。要求支持两种方式：精确查找（按单词 查）和模糊查找（按前几个字母查）。

③ 单词记忆

最近浏览过的 10 个单词，允许回退/前进进行浏览。

（4）要求使用 BST 或者 AVL 实现动态索引结构。

（5）考虑输入的容错性。

【提高要求】

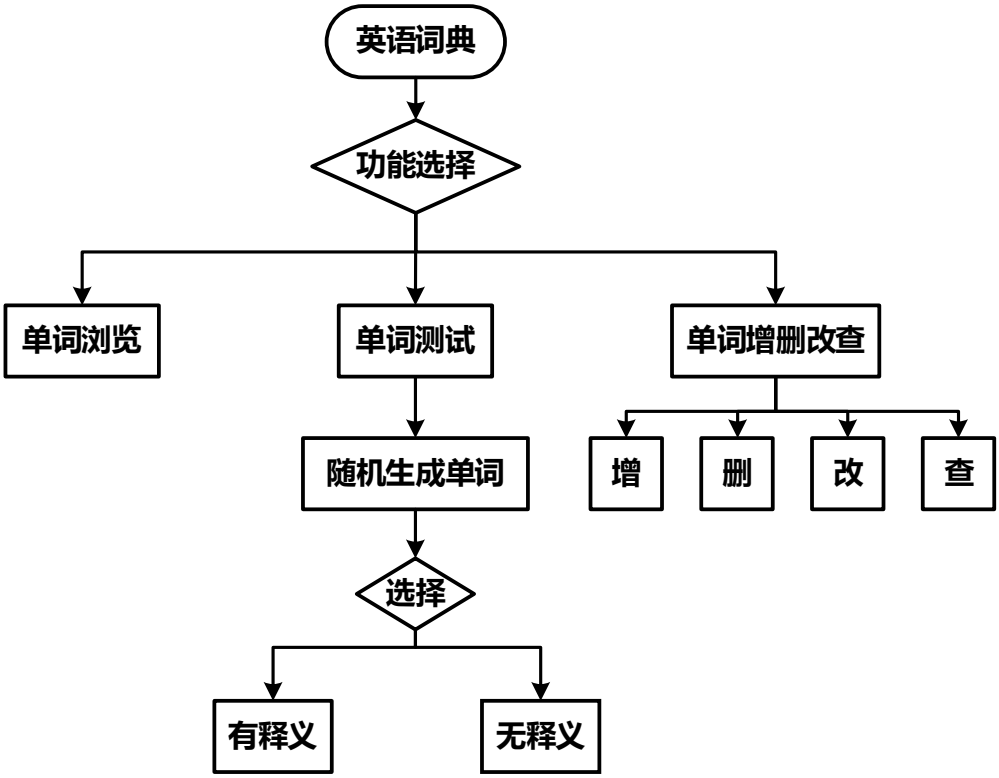
（1）提供可视化操作界面；

（2）支持“根据中文反查英文单词”的功能。提示：可考虑由汉字机内码构 造英文哈希表；

（3）背单词功能：支持“单词填空”的简单背单词软件功能。提示：可参考 的软件为“轻轻松松背单词”等。

（4）使用红黑树或者 B-树的数据结构，来实现动态索引结构。

3.1.2 总体设计



英语词典软件采用 QT 框架，设计了包括单词浏览，单词测试和单词增删改查在内的三大功能：

- **单词浏览**：顺序查看单词表，显示所有的单词，支持分页滚动查看，达到记单词的目的。
- **单词测试**：随机生成无中文释义的单词进行测试，若不认识，可点击按钮显示中文释义。左侧的窗格能够显示最近浏览过的 10 个单词。
- **单词增删改查**：基于字典树实现了查找单词、插入单词（插入时，先查找，找不到则插入，找到则提示用户）、删除单词（删除时，先查找，找到则删除，找不到则提示用户）。

3.1.3 算法思想

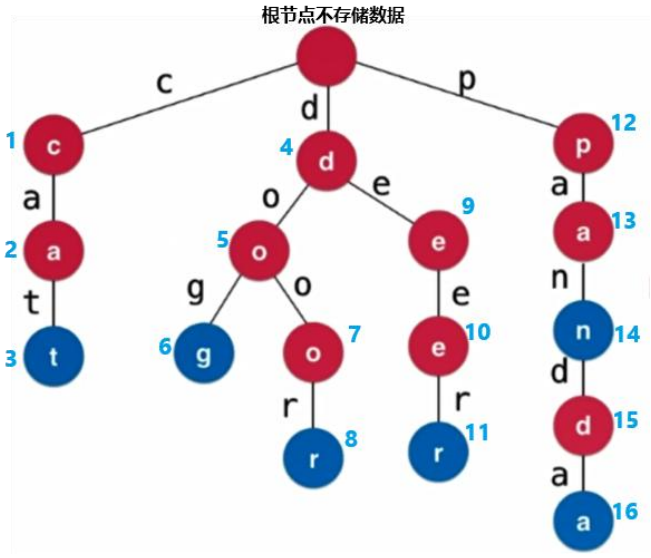
对字典内单词的增删改查操作是基于 Trie 树(字典树)结构实现的。

- **Trie 树(字典树)介绍**

字典树（Trie），又称前缀树或单词查找树，是一种树形数据结构。字典树的特点是每个节点代表一个字符，从根节点到叶子节点的路径表示一个完整的字符串（或单词）。字典树主要用于高效地存储和检索字符串集合。

字典树的核心思想是通过共享字符前缀来减少存储空间和提高查询效率。例如，如果多个字符串共享相同的前缀（如 “cat” 和 “car” 共享 “ca”），那么在字典树中，这些前缀只会存储一次。

假设插入的顺序为：“cat”，“dog”，“door”，“deer”，“pan”，“panda”。就会形成如下字典树。



可见当两个不同的单词拥有共同前缀时，字典树可以显著的减少存储空间，并能高效查找单词。

- **字典树的结构**
  - 根节点：字典树的根节点通常为空或代表一个特殊标记（如空字符）。
  - 节点：每个节点代表一个字符，并且包含指向子节点的指针（通常是一个数组或哈希表），用于存储下一个字符。

- 结束标记：每个单词的结尾需要一个特殊标记（如布尔值 isEndOfWord），用于表示从根节点到当前节点的路径是有效的字符串。

- **基于字典树的增删改查**
  - **插入（Insertion）**

1. 从根节点开始，遍历单词的每个字符。
2. 对于每个字符，它检查是否存在对应的子节点。
3. 如果不存在，则创建一个新的子节点。
4. 所有字符处理完毕后，标记最后一个节点为单词的结尾，并存储单词及其释义。

  - **查找（Search）**

1. 从根节点开始，遍历单词的每个字符。
2. 对于每个字符，它检查是否存在对应的子节点。
3. 如果不存在，则返回“未找到”。
4. 如果所有字符都存在，则返回最后一个节点存储的中文释义。

  - **删除（Delete）**

1. 从根节点开始，遍历单词的每个字符。
2. 对于每个字符，它检查是否存在对应的子节点。如果不存在，则返回“删除失败”。
3. 如果所有字符都存在，则标记最后一个节点为非单词结尾，并清空存储的单词及其释义。

- **函数介绍**
  - bool insertWord(const QString &ew, const QString &cm);　插入单词
  - bool deleteWord(const QString &ew);　删除单词
  - QString searchWord(const QString &ew); 查找英语单词并输出它的中文释义
  - void fileWrite(const QString &fp); 将数据写入文件
  - void fileRead(const QString &fp); 从文件读取数据
  - int getLength() 返回当前表单词数量

3.1.4 问题与解决方法

问题:字典树的删除操作相对复杂。当一个单词被删除时，可能会导致树的结构变得不必要地复杂。

解决方法:标记删除: 在节点上使用标记，标记某个单词是否被删除，而不是实际删除节点。后续查询时忽略这些标记。

3.1.5 算法分析

- 程序运行结果分析
  - **主界面（功能选择）**

图 1 主界面

  - **单词浏览：**顺序查看单词表，显示所有的单词，支持分页滚动查看，达到记单词的目的。

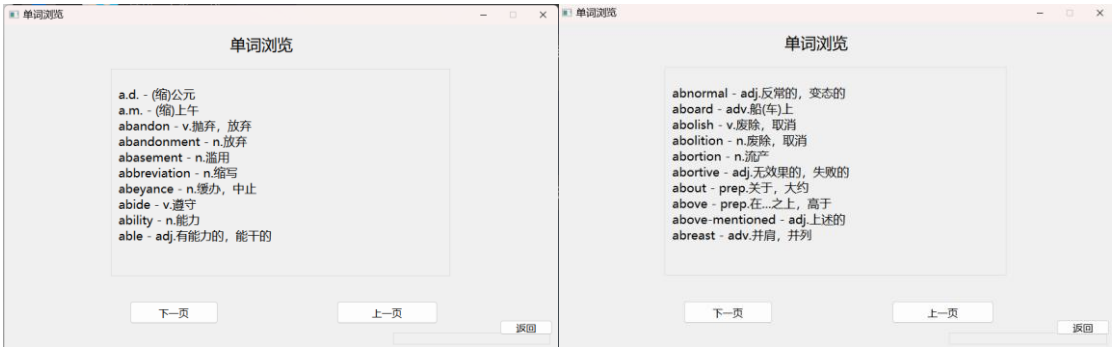


图 2 单词浏览

- **单词测试：**随机生成无中文释义的单词进行测试，若不认识，可点击按钮显示中文释义。左侧的窗格能够显示最近浏览过的 10 个单词。

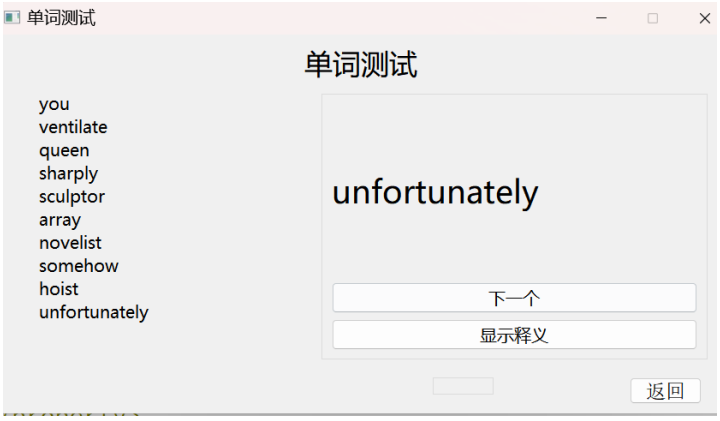


图 3 单词测试（无中文释义）

图 4 单词测试（显示中文释义）

- **单词增删改查：**查找单词、插入单词（插入时，先查找，找不到则插入，找到则提示用户）、删除单词（删除时，先查找，找到则删除，找不到则提示用户）。
  - *单词增删改查界面*

图 5 单词增删改查界面

◆ *单词查询*

已有单词可查到：

图 6 单词查询（已有单词）

没有的单词查不到：

图 7 单词查询（没有的单词）

可见单词 hhh 不在词库中

◆ *单词插入*

图 8 单词插入

成功插入单词 hhh，可以查到释义！

◆ *单词删除*

图 9 单词删除

成功删除单词 hhh，现在查不到释义了。

- **字典树时空复杂度分析：**
  - 时间复杂度：
    - 在查找和插入时，单词时的复杂度为 O(E)，E 为单词的长度。
    - 删除单词时如果该单词不是叶子节点，则删除的时间复杂度为 O(E)，E 为单词的长度；如果该单词是叶子节点，则删除单词时有递归回溯的过程，平均时间复杂度为 O(1.5 * E)，E 为单词的长度。总体的时间复杂度为 O(E)。
  - 空间复杂度：

    字典树的目的是减少字符串的比较，从而使用了多叉树在空间上以指数级的增长，树的高度为单词的长度平均在 15，节点数在 60 左右。

3.1.6 本题小结

英语词典软件采用 QT 框架，基于字典树结构设计了包括单词浏览，单词测试和单词增删改查在内的三大功能，但是字典树通常用于单词查找，但在处理部分匹配（如模糊判断某单词子串是否存在）时，效率会降低。

3.2 拼图软件（图搜索，15 分）

3.2.1 问题描述

打开任意一张图像，按照可以设定的切片数进行分割并打乱排序，通过 A* 算法（启发式搜索）恢复原来图像。

这个问题可以简单表述为：3*3 的格子装了 1-8 的 8 个数字，数字是随机分布 于各个格子中，是否可以利用空格的格子，移动装有数字的格子最终达到某种序列，如下图所示：

【实现原理】

状态空间搜索，就是将问题求解过程表现为从初始状态到目标状态寻找这个 路径的过程。由于求解问题的过程中分枝有很多， 主要是求解过程中求解条件的不确定性和不完备性所造成，使得求解的路径很多这就构成了一个图，我们说这个图就是“状态空间”。问题的求解实际上就是在这个图中找到一条路径可以从开始到结果，这个寻找的过程就是“状态空间搜索”。常用的状态空间搜索有深度优先和广度优先，它们的缺陷在于都是在一个给定的状态空间中穷举。因此，需要使用启发式搜索。

启发式搜索就是在状态空间中的搜索对每一个搜索的位置进行评估，得到最 好的位置，再从这个位置进行搜索直到目标，这样可以省略大量无谓的搜索路径 以提到效率。在启发式搜索中，对位置的估价是十分重要的。启发中的估价是用 估价函数表示的，如：f(n) = d(n) + h(n)，其中 f(n)是节点 n 的估价函数，d(n) 实在状态空间中从初始节点到 n 节点的实际代价，h(n)是从 n 到目标节点最佳路径 的估计代价。

在拼图游戏中，可以采用如下的评价函数，f(n) = d(n) + h(n)，其中 d(n) 为当前状态从初始状态开始移动的步数，h(n)计算当前状态与目标状态相比错位 的个数。搜索过程总是往 f(n)最小的分枝方向进行，以便快速达到最终状态。

【基本要求】

（1）基于 A*算法（启发式搜索）实现，通过最少的移动次数，恢复原图像；

（2）根据提供测试数据，如下图所示：X 代表空格，把 X 与四方向网格（上/ 下/左/右）的交换操作定义为 u、d、l、r，要求输出空格 X 的交换操作序列。如 果答案不唯一，输出一种求解过程即可。如果无解，输出“无解决方案”。

【提高要求】

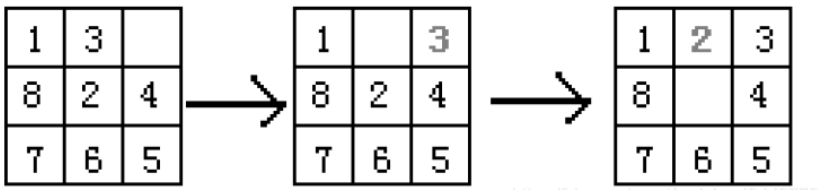
（1）提供可视化操作界面；

（2）支持 JPG、BMP 等文件格式，图片分割由系统自动完成，用户可以设置 切片数量。游戏开始时，图片位置随机生成；

3.2.2 总体设计

- **A*算法介绍**

A*算法是一种启发式图搜索算法，其特点在于对估价函数的定义上。对于一般的启发式图搜索，总是选择估价函数 f 值最小的节点作为扩展节点。因此，f 是根据需要找到一条最小代价路径的观点来估算节点的，所以，可考虑每个节点 n 的估价函数值为两个分量：从起始节点到节点 n 的实际代价 g(n)以及从节点 n 到达目标节点的估价代价 h(n),且 h（n）≤h*（n） , h*（n） 为 n 节点到目的结点的最优路径的代价。



- **启发式函数**

启发式函数为 f(n) = g(n) + h(n)

其中 F 为 Final Score，代表 A*算法衡量该节点最终值。G 为 Goal，计算了从起始节点到该节点的实际消耗。H 为 Heuristic Score，预估了当前节点到目标节点的消耗。

启发式函数听起来很有学问，其实可以很简单的理解为从源点到目标的所需要消耗的总代价 f(n)（和适应度函数比较相像），这个总代价可以分成两个部分从源点到中间节点（搜索的中间状态）已经消耗的实际代价 g（n），另一个部

分就是对从中间节点到目标的预测 h(n)。

通常来说，这里的代价一般是指各种距离，像欧式距离，曼哈顿距离等等，这个根据你所求解的实际问题决定。

另一个值得指出的就是预测，预测值直接影响了问题求解的效率以及能否求得合理的解。这里给出一个结论：对于任意预测值 h(n）均小于等于实际值的话，我们可以说最终解就是问题的最优解。

- **open 表与 close 表的维护**

open 表：先可以简单认为是一个未搜索节点的表

close 表：先可以简单认为是一个已完成搜索的节点的表（即已经将下一个状态放入 open 表内）

规则一：对于新添加的节点 S（open 表和 close 表中均没有这个状态），S 直接添加到 open 表中

规则二：对于已经添加的节点 S（open 表中并且 close 表中没有这个状态），若在 open 表中，与原来的状态 S0 的 f(n)比较，取最小的一个。

规则三：下一个搜索节点的选择问题，选取 open 表中 f(n)的值最小的状态作为下一个待搜索节点

规则四：每次需要将带搜索的节点下一个所有的状态按照规则一二更新 open 表,close 表，搜索完该节点后，移到 close 表中。

- **算法**

初始化：将起始节点添加到开放列表中。

循环执行以下步骤：

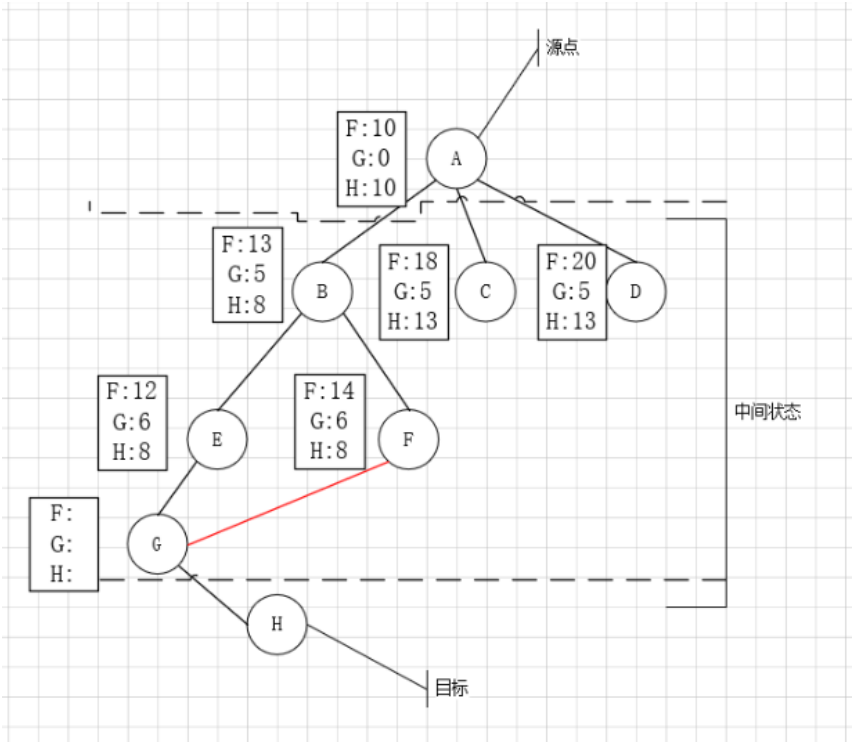
1 从开放列表中找出 F 评分最低的节点，将其设为当前节点。

2 检查当前节点是否是目标节点。如果是，则算法结束，路径被找到。

3 将当前节点移至关闭列表，并处理所有邻近节点：

4 如果邻近节点不在开放列表中，计算其 G、H 和 F 值，然后将其添加到开放列表。

5 如果邻近节点已在开放列表中，检查通过当前节点到达它的路径是否更好。如果更好，更新其 F、G 和 H 值。



3.2.3 算法思想

本题采用 A*算法求解 N 数码难题。

- **八数码问题分析**

1. 启发式函数的确定

   h(n):已经移动的步数

   g(n):此状态与目标状态九宫格中相异数字的个数

2. 状态保存

   A*算法有个很大的问题就是消耗内存资源，我们可以用 char 型数据保存，这里我另一种保存策略：用一个 long int 数值表示，0-8 九个状态可以四位二进制数来表示 0000B-1000B,所以九个状态就可以用 36 个二进制位来表示，然后这 36 位二进制数就可以用一个 long int 型数据来表示，这样增加编码和解码工作。

3. 算法优化

   在找最小值的时候，我们可以用二分查找，o(n)优化到 o(logn)，这就要求我们再插入时顺序插入，因为查询次数是要大于添加 open\close 表项的，所以这个方法是可以优化执行效率的

4. 无解情况

九宫格变成线性后，计算初始状态和目标状态的奇偶性是否一致，一致有解，否则无解。

3.2.4 问题与解决方法

问题：初始值的八数码初始值的选择对于实验结果的影响很大，在选取一些样例时，比如 1,3,0,2,8,4,7,6,5，实验结果达到 20000 次依然没有停止。

解决方法：鉴于时间原因，选取一些迭代次数较小就可以达到目标状态的样例进行验证，

3.2.5 算法分析

- 程序运行结果
  - **主界面**

图 10 解八数码问题的主界面

- ✓ 在此界面确定初始矩阵和目标矩阵，中间过程可以动态显示解八数码问题的每一步的过程矩阵。
- ✓ 过程展示模块可以全部显示解八数码问题的过程矩阵。

- **确定初始矩阵和目标矩阵：**

图 11　确定初始矩阵和目标矩阵

- **无解情况**

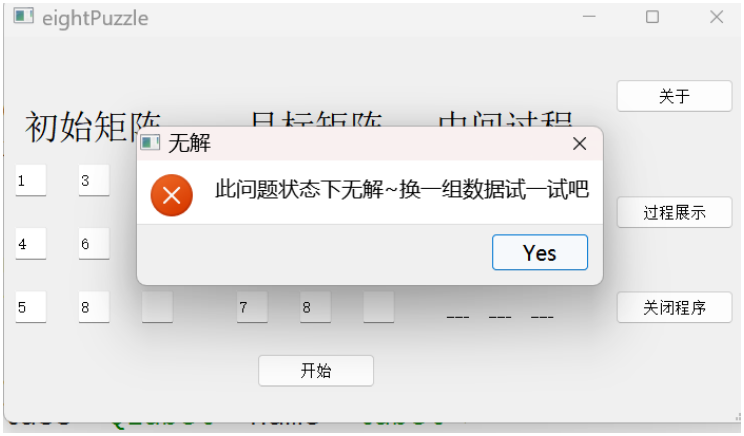


图 12 无解情况

此情况无解，换一组试试。

- **有解情况**

图 13 有解情况

➢ **过程展示**

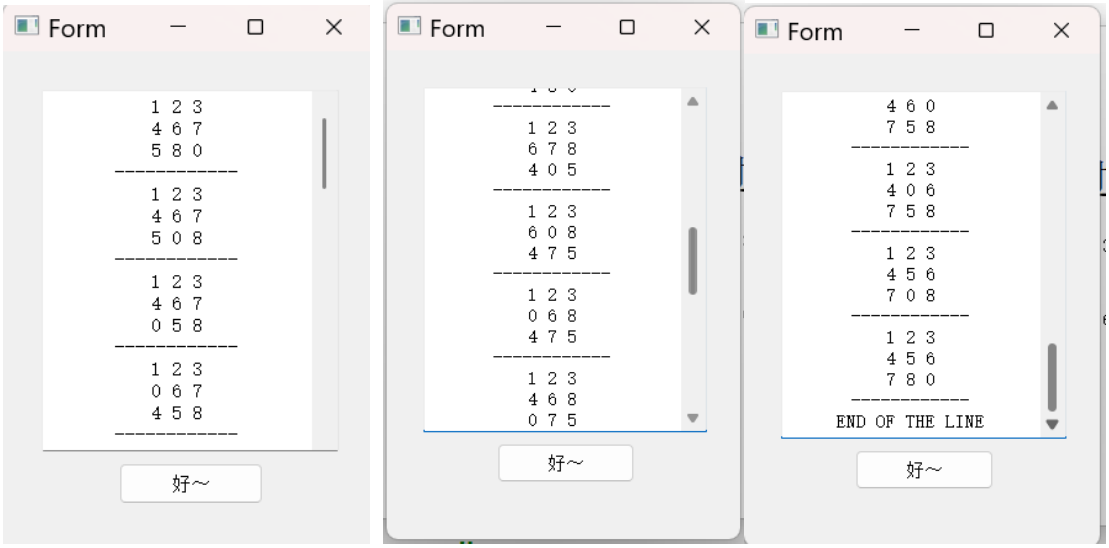


图 14 解八数码问题的全部过程矩阵

✓ 过程展示模块可以显示解八数码问题的全部过程矩阵。

- **时空复杂度分析**

时间复杂度：最坏情况下为 O(b^d)，这里 b 是分支因子，d 是目标状态的深度，实际复杂度会受到启发式函数的影响。

空间复杂度：最坏情况下也为 O(b^d)，因为需要存储生成的节点信息。

3.2.6 本题小结

通过本次实验，发现选用不同的启发函数，对于实验的结果有较大的影响。采用 A*算法远远优于普通的广度优先搜索，同时，明显的感觉到启发函数效率更高，更快的找到最优解。但是，在实验过程中，也遇到了一些问题，比如初始值的八数码初始值的选择对于实验结果的影响很大，在选取一些样例时，比如 1,3,0,2,8,4,7,6,5，实验结果达到 20000 次依然没有停止，无法比较两种启发函数的优越性，鉴于时间原因，选取一些迭代次数较小就可以达到目标状态的样例进行验证，发现第二种结果优于第一种启发函数的结果。总的来说，实践出真知，只有把书上的理论知识运用到实践，才是真正地掌握。

3.3 磁盘文件同步软件（哈希，10 分）

3.3.1 问题描述

每天我们都会持续生产电子资料，如文档更新、资料下载、拍照录像等。因此，需要对资料进行智能化的备份与保存。文件同步软件是一类针对电脑各种文件同步的软件，可以在指定的两个文件夹之间进行单向或双向的同步，同时也能满足用户实现硬盘之间、硬盘与移动存储设备之间的备份与同步。需要注意的是：备份≠同步。备份就是把资料一股脑儿拷贝或上传，有很大的局限性，资料包越来越大，每次拷贝上传很耗时间，因此也无法频繁备份。而同步， 只针对有变化 的文件随时同步更新，速度快，每天至少可以同步一次。本题要求设计并实现一 款磁盘文件同步软件。

【背景知识】

有序哈希树（Ordered Hash Tree, OHT）可以用来检查两个目录中文件的变 化，它的主要优势在于不需要记录时间信息，而且可以快速获取文件变化。磁盘 有序哈希树的构建，是文件为粒度，文件(目录)的相对路径和修改时间戳的哈希 值为校验依据，构建文件(目录)的有序哈希树。

树中每个节点可以用五元组<p,h, Parent, FistChild, NextSibling>表示。 其中，p 表示该节点对应的文件相对路径，h 表示该节点存储的 Hash 值，Parent, FistChild 和 NextSibling 分别代表该节点在 OHT 中的父节点、头子孙节点和后继 兄弟节点。

对于文件，OHT 中对应节点的哈希值即为文件数据自身的哈希值（如 MD5 值）；对于目录，则其哈希值为文件相对路径的哈希值；若文件（目录）不存在，则其哈希值为 NULL。

创建 OHT 树时，首先分别创建其直属子目录或子文件的对应节点；然后将其直属子节点的哈希值和文件名进行拼装，并将拼装结果的哈希值作为其对应节点的哈希值。由于有序哈希树中子节点按照文件名进行了排序， 因此只要目录的某 个子文件内容或目录结构发生变化，其对应的 OHT 节点的哈希值必然发生变化。 这也是同步的依据。

【基本要求】

（1）提供友好的界面操作：选择需要同步的文件或者目录，左右两个视图对 比待浏览同步的两个文件或者目录，点击“同步”按钮即可完成；

（2）以树型结构列出磁盘目录上的文件，优化目录树创建过久导致的用户等 待时间；

（3）能够实现电脑、移动硬盘之间的文件同步和文件备份; 可以同步任何数 据类型的文件（文档、图像、音频、视频等）；

（4）冲突解决。如果上次同步之后，某文件在两侧都分别进行了改动，那么 下次同步时这个文件将标记为“冲突”。用户可以通过选定同步的单一方向来解 决“冲突”的问题；选择方案：左侧优先、右侧优先、新文件优先；

（5）设置同步任务计划，定时同步。

【提高要求】

（1）当文件或者目录被改变时（增加、删除、重命名等），实现自动同步。

（2）可以同时执行多个同步任务。

【测试须知】

在电脑上，选择数据量大小>100M 的单文件、包含多文件的目录。

（1）复制功能：原始文件夹（含多个文件及目录）---目的文件夹（空）；

（2）同步功能：原始文件夹（增加、删除、重命名一个文件或者目录）-目的 文件夹（非空）；

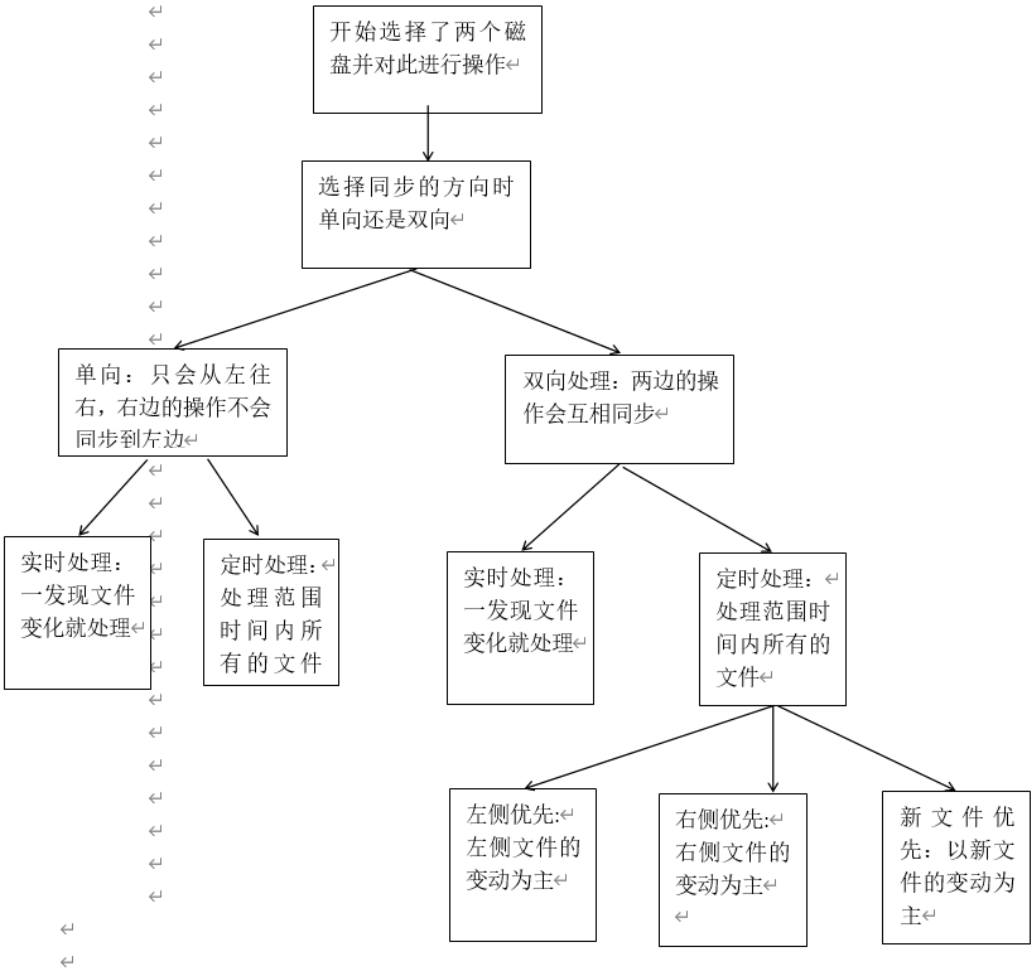
3.3.2 总体设计

- 设计思想：
  - 储存结构：哈希树
  - 主要算法思想：根据两边文件，构建两个相应的哈希树，然后通过哈希树比对等算法，得出两边文件的差异然后进行操作
- 设计表示：
  - 主要类 Synchronous，控制着同步的操作，主要数据成员有哈希树 oht、ohts，文件不同的集合 L。
  - Synchronous 中的函数 Compare_OHT 、Add_Sub_OHT 、Remove_Sub_OHT、Update_Sub_OHT 用来

- Synchronous 函数 Creat、Delete、Modify，对应的时文件的创造、删除、更新操作
- 主窗口类主要负责显示，然后根据用户选择来操作类 Synchronous 实现对应的操作

3.3.3 算法思想

- **流程图：**



哈希树（Ordered Hash Tree, OHT）用于文件同步的逻辑如下：

1) 哈希树的结构

哈希树的每个节点（HashNode）包含以下信息：

- path: 文件或目录的相对路径
- h: 文件或目录的哈希值
- Parent: 父节点

- FistChild: 第一个子节点
- NextSibling: 下一个兄弟节点

2) 哈希树的构建

哈希树通过递归构建：

- 对于目录，遍历其子目录和文件，递归构建子节点，并计算目录的哈希值（基于子节点的哈希值）。
- 对于文件，读取文件内容并计算其哈希值。

3) 文件同步的逻辑

文件同步通过比较两棵哈希树来实现。主要步骤如下：

a) 比较哈希树

Compare_OHT 函数比较两棵哈希树，返回 QList<FileNode>文件差异集合。比较逻辑如下：

- 如果一个节点存在于一棵树中但不存在于另一棵树中，则记录为添加或删除操作。
- 如果两个节点路径相同但哈希值不同，则记录为更新操作。
- 递归比较子节点。

b) 处理文件差异

solvefile 函数处理文件差异集合，执行具体的文件操作（创建、删除、更新）。根据差异集合中的每个文件节点（FileNode），执行相应的操作：

- Creat: 创建文件或目录。
- Delete: 删除文件或目录。
- Modify: 更新文件内容。

4) 文件监视器

文件监视器（QFileSystemWatcher）用于监视文件和目录的变化。当检测到变化时，重新构建哈希树并同步文件。

3.3.4 问题与解决方法

问题 1：不同文件可能会产生相同的哈希值，导致哈希冲突。

解决方法 1：

✓ 使用强哈希算法：选择强哈希算法（如 SHA-256）来减少哈希冲突的概率。

✓ 多级哈希：在哈希值相同时，进一步比较文件内容，确保文件的一致性。

问题 2：在一个文件夹中删除文件后，另一个文件夹中的同名文件可能仍然存在。

解决方法 2：

✓ 删除标记：在哈希树中添加删除标记，当检测到文件被删除时，在同步过程中删除对应的文件。

✓ 双向同步：在双向同步模式下，确保删除操作在两个文件夹中都执行。

3.3.5 算法分析

- 程序运行结果
  - **同步主界面**

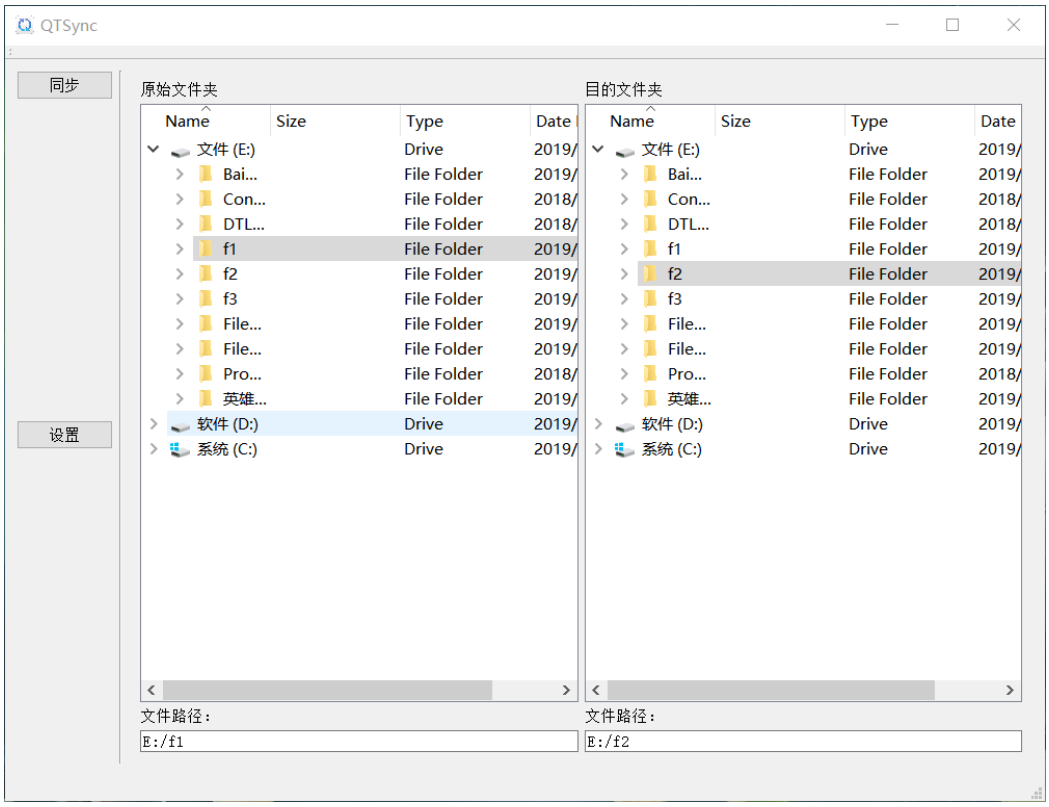


图 15 同步主界面

在同步主界面可以选择原始母文件夹和目标子文件夹，点击同步就可以进行同步，当文件或者目录被改变时（增加、删除、重命名等），实现自动同步。

➢ **设置页面**

图 16 设置页面

在设置页面可以设置：

✓ 左侧优先/右侧优先/新文件优先

✓ 单向同步/双向同步

✓ 自动更新/定时更新（可自定义定时时间）

➢ **同步效果**

✓ 同步前的原始母文件夹和目标子文件夹：

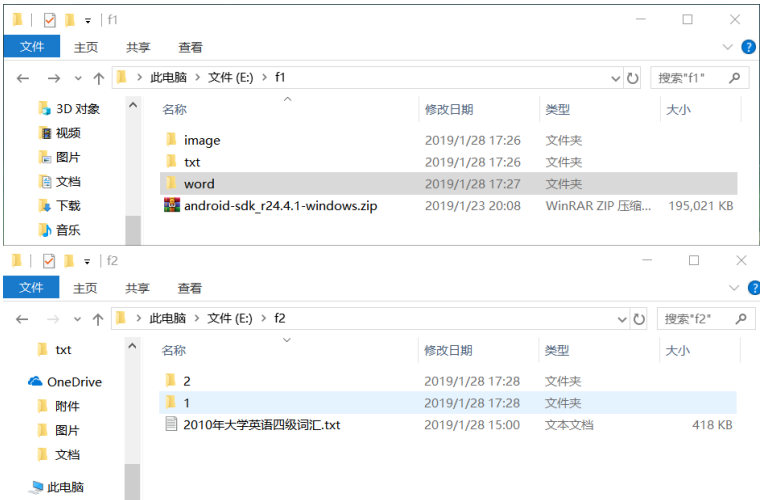


图 17 同步前的原始母文件夹和目标子文件夹

可见同步前的原始母文件夹和目标子文件夹不一致。

✓ 同步后的原始母文件夹和目标子文件夹：

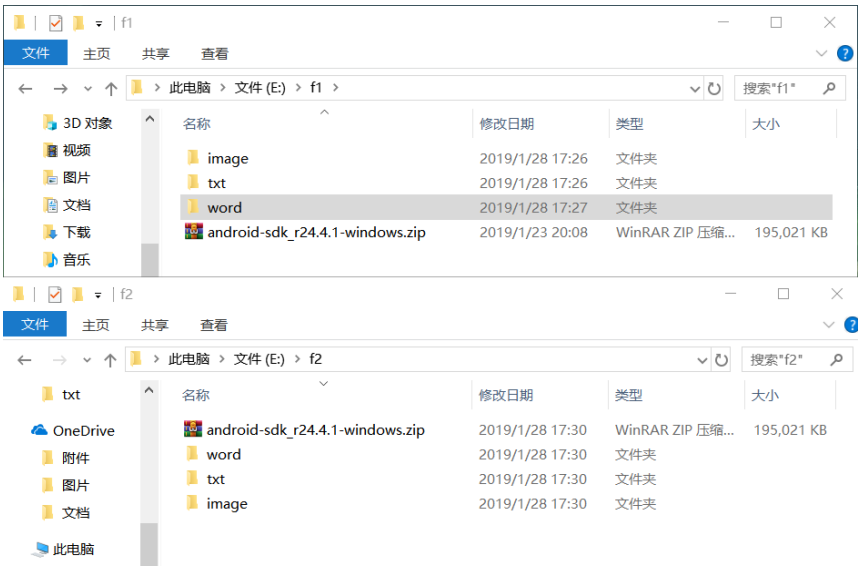


图 18 同步后的原始母文件夹和目标子文件夹

可见同步后的原始母文件夹和目标子文件夹内的文件已完全一致。

- **时空复杂度分析**
  - 时间复杂度
    - ✓ 构建哈希树：O(n * m)
    - ✓ 比较哈希树：O(n1 + n2)
    - ✓ 处理文件差异：O(d1 * m + d2 + d3 * m)
  - 空间复杂度
    - ✓ 构建哈希树：O(n)
    - ✓ 比较哈希树：O(d)
    - ✓ 处理文件差异：O(m)

  其中：

  - n 是文件系统中的文件和目录总数。
  - m 是文件的平均大小。
  - n1 和 n2 分别是两棵哈希树的节点数。
  - d 是文件差异集合中的文件节点数。
  - d1、d2 和 d3 分别是需要创建、删除和更新的文件节点数。

3.3.6 本题小结

本项目通过哈希树实现文件同步，主要包括构建哈希树、比较哈希树和处理文件差异等核心算法。哈希树的结构和递归构建方法使得文件同步过程高效且准确。项目中使用了文件监视器（QFileSystemWatcher）来实时监控文件变化，并根据变化进行同步操作。

在实际应用中，可能会遇到文件冲突、文件删除、文件夹结构变化、文件监视器性能、哈希冲突、文件锁定、网络延迟和文件权限等问题。通过优化文件监视器性能、改进冲突处理机制、增强文件同步的鲁棒性、改进用户界面、支持更多同步模式和优化哈希算法，可以进一步提升文件同步的效率和用户体验。

四．课程小结

基础题难度有点大，想要搞懂每一道题还是有点费力，但同时也帮我巩固了数据结构的知识。

综合题也同样具有挑战性，我选择三道综合题都使用 Qt 搭建，在学习字典树，A*算法和哈希树的同时对 Qt 地使用也更熟练了，极大的提高了我的编程水平。

五．课程建议

1. 基础题太难了，而且很多，光写第一题就花了很久，可以出的简单一点。
2. 综合题也太难太多了，我三个综合题都用了 Qt 实现，是因为我现在大四，去年大三的时候用 qt 写过项目，所以比较熟练，但是大二的同学学上手 Qt 可能会比较吃力。
3. 建议减少题目。