**计 算 机 与 信 息 科 学 学 院**



**数据结构课程设计报告**

`

**项目名称： 二叉排序树的应用**

**专业年级： 2023计算机科学与技术**

**学生姓名： 谭祎（2023051603132）**

**穆江梅（2023051603125）**

**黄子幸（2023051603159）**

**蒋巧瑞（2023051603152）**

**蒋远籴（2023051603128）**

**指导教师： 李晓虹**

目录

[1.问题分析 4](#_Toc32402)

[1.1题目分析 4](#_Toc5546)

[1.2系统功能需求 5](#_Toc14661)

[1.2.1基本操作 5](#_Toc16506)

[1.2.2用户界面 5](#_Toc3770)

[1.2.3文件操作 5](#_Toc2702)

[1.3 补充条件说明 6](#_Toc27848)

[2.项目设计 7](#_Toc26794)

[2.1设计思想 7](#_Toc25154)

[2.1.1 程序结构 7](#_Toc28399)

[2.1.2 主要的数据结构 7](#_Toc22779)

[2.1.3算法的主要思想 7](#_Toc13648)

[2.2设计表示 8](#_Toc12507)

[2.2.1 函数及其功能描述 8](#_Toc11467)

[2.2.2 二叉树数据结构图 9](#_Toc19638)

[2.3详细设计 10](#_Toc21359)

[2.3.1 数据加载和文件操作 10](#_Toc26310)

[2.3.2二叉排序树的基本操作 10](#_Toc25963)

[2.3.2.1插入操作 10](#_Toc29167)

[2.3.2.2删除操作 11](#_Toc23483)

[2.3.2.3查找操作 11](#_Toc13884)

[2.3.3平衡二叉排序树的设计 11](#_Toc19768)

[2.3.4 平均查找长度（ASL）的计算和优化· 12](#_Toc2617)

[2.4项目创新点及思考 12](#_Toc11529)

[2.4.1平衡二叉排序树的构建 13](#_Toc22542)

[2.4.2 文件I/O操作 13](#_Toc31184)

[2.4.3 平均查找长度（ASL）的计算 13](#_Toc21643)

[3.使用说明 14](#_Toc19205)

[3.1功能简介 14](#_Toc16265)

[3.2 主要功能函数说明 14](#_Toc22105)

[3.2使用步骤 15](#_Toc67)

[4.调试及测试 15](#_Toc5411)

[4.1测试目的 15](#_Toc17820)

[4.2测试环境 15](#_Toc12822)

[4.3设计测试用例 15](#_Toc20842)

[4.3.1.基本测试用例 16](#_Toc27096)

[4.3.2.边界测试用例 18](#_Toc3901)

[4.3.3.性能测试用例 19](#_Toc7479)

[4.3.4.文件操作测试用例 20](#_Toc23098)

[4.4 开发中暴露的问题 22](#_Toc2569)

[4.4.1初始程序与文件操作 22](#_Toc13064)

[4.4.2.树深度与测试 22](#_Toc2038)

[4.4.3.ASL计算的复杂性与验证 22](#_Toc28125)

[5.存在的问题及解决思路 22](#_Toc20676)

[5.1测试问题分析 22](#_Toc25129)

[5.1.1 初始程序与文件操作 22](#_Toc22495)

[5.1.2 树深度与测试 23](#_Toc12408)

[5.1.3 ASL计算的复杂性与验证 23](#_Toc4910)

[5.2 解决方案 24](#_Toc6269)

[6.附录 26](#_Toc26479)

[6.1源程序 26](#_Toc26728)

[6.2 测试数据 33](#_Toc15548)

[6.3 运行结果 33](#_Toc6132)

0.小组分工

|  |  |
| --- | --- |
| 姓名 | 分工 |
| 谭祎 | 基础算法：二叉排序树的插入、删除  算法优化：对文件数据创建平衡二叉树，计算ASL;  文件：第二章2.3、2.4 |
| 穆江梅 | 基础算法：二叉排序树的查找、输出中序遍历  算法优化：编写输出ASL过程的辅助函数  文件：第二章2.1、2.2 |
| 黄子幸 | 基础算法：二叉排序树的结构定义和初步整合  算法优化：对文件数据创建平衡二叉树  文件：第一章及附录 |
| 蒋巧瑞 | 基础算法：二叉排序树的删除  算法优化：编写计算二叉排序树的高度的算法  文件：第五章 |
| 蒋远籴 | 基础算法：二叉排序树的创建、输出大于x的结点  算法优化：编写计算二叉排序树的高度的算法、计算ASL;  文件：第三章、第四章 |

1.问题分析

**1.1题目分析**

（1）创建一个包括15个data值的二叉排序树

首先，需要定义一个二叉排序树的结点结构，包括结点数据、左子树和右子树。其次，用15个data值创建平衡二叉排序树。

（2）对二叉排序树进行增加、删除、查找等操作

对数据进行的运算，主要借鉴书上的内容。对于插入（增加）而言，直接根据二叉排序树的性质来操作即可。对于删除而言，需要考虑有无删除结点、删除结点有无左右孩子。对于查找而言，根据二叉树的性质，运用递归查找指定结点。

（3）输出中序遍历结果

主要借鉴书上的内容。中序遍历可以通过递归得到从小到大的有序序列。

（4）从小到大输出排序树中所有数据大于等于给定x值得结点数据

首先访问左子树，如果当前结点值大于等于x，则输出当前结点值，并且继续访问右子树；如果当前结点值小于x，则查找右子树；

（5）菜单操作

设计一个简单的可操作菜单，让用户可以选择插入结点、删除结点 、查找结点、输出中序遍历结果、输出大于等于x值得结点数据、保存到文件、从文件加载、计算二叉排序树的深度、计算平均查找长度（ASL)、退出程序的操作。

（6）文件I/O

从文件中读取数据，加载到二叉排序树中；将对二叉排序树进行的的操作结果保存到文件中。

**1.2系统功能需求**

### **1.2.1基本操作**

插入节点、删除节点、查找节点、中序遍历、输出大于等于x的节点数据。

### **1.2.2用户界面**

（1）绘制菜单的用户界面，列出所有可用的操作选项。

（2）提供清晰的提示和指导，以便用户可以轻松地进行操作。

### **1.2.3文件操作**

（1）从文件中读取数据并加载到二叉排序树中。

（2）将二叉排序树的数据保存到文件。

（3）程序应能够处理可能的错误，如插入重复的值、删除不存在的值等。

（4）程序应经过充分的测试，确保所有功能都能按预期工作，并且没有明显的错误或漏洞。通过满足这些功能需求，我们可以创建一个可用性强且友好的的二叉排序树应用程序。

**1.3 补充条件说明**

（1）通过数据创建平衡的二叉排序树

平衡二叉树的创建作为普通二叉树创建的优化，满足了我们对数据高效计算的要求，并且平衡二叉树从中间作为根节点的思想与输出所有大于等于给定x值的结点数据的函数算法不谋而合。两个函数之间联系也让项目代码更加具有合理性和优化性。

（2）计算二叉排序树的深度

作为本次项目扩展的任务之一，计算二叉排序树的深度可以帮助我们更深入的了解其思想和算法，也可以帮助用户了解更多的关于二叉排序树的定义。

（3）计算平均查找长度（ASL)

作为本次项目扩展的任务之一，计算平均查找长度也是课程学习知识点之一，此项目不仅可以计算二叉树的平均查找长度，还增强了打印其详细过程的功能。不仅可以帮助我们更好的温习专业知识，还能够帮助用户理解平均查找长度是什么以及平均查找长度是如何计算的。

2.项目设计

## **2.1设计思想**

### **2.1.1 程序结构**

整个程序分为多个函数，包括创建二叉排序树、插入节点、删除节点、查找节点、中序遍历、输出大于等于给定值的节点等功能函数，主函数用于调用这些函数并实现菜单操作和文件 I/O。

### **2.1.2 主要的数据结构**

使用二叉树结构来存储数据。每个节点包含数据值、左子节点指针和右子节点指针。

### **2.1.3算法的主要思想**

创建二叉排序树时，通过比较新插入节点的值与当前节点的值，将其插入到合适的位置，保证左子树的值小于根节点，右子树的值大于根节点。

插入操作时，从根节点开始，按照二叉排序树的规则递归地找到合适的插入位置。

删除操作分为三种情况：删除没有子节点的节点、删除只有一个子节点的节点、删除有两个子节点的节点。对于有两个子节点的情况，通过找到右子树中的最小节点替换要删除的节点。

查找操作类似插入操作，从根节点开始递归比较。

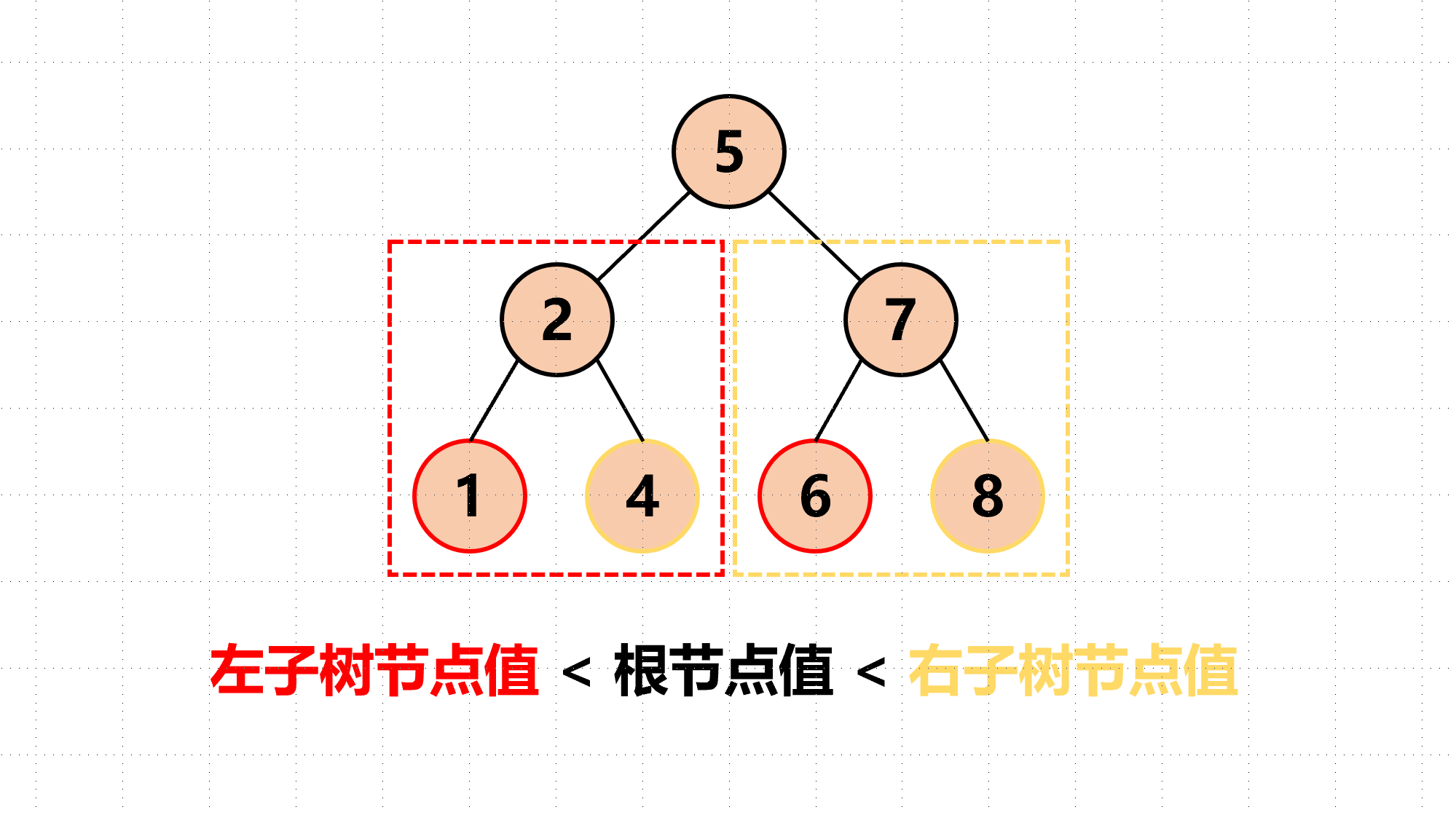
中序遍历按照左子树、根节点、右子树的顺序访问节点，从而得到有序的输出。

## **2.2设计表示**

### **2.2.1 函数及其功能描述**

|  |  |
| --- | --- |
| 函数及文件 | 功能描述 |
| InsertBST | 用于向二叉排序树中插入节点 |
| InsertBST | 对二叉排序树进行中序遍历并输出节点的关键字 |
| SearchBST | 在二叉排序树中查找指定关键字的节点 |
| DeleteBST | 从二叉排序树中删除指定关键字的节点 |
| PrintGreaterOrEqual | 输出二叉排序树中所有大于等于给定值 x 的节点数据 |
| SaveToFile | 将二叉排序树的节点数据保存到指定的文件中 |
| LoadFromFile | 从指定文件中加载数据来构建二叉排序树（代码中未实现此函数） |
| CalculateDepth | 计算二叉排序树的深度 |
| CalculateASL | 计算二叉排序树的平均查找长度（ASL） |
| CalculateASLAndPrint | 辅助计算 ASL 并打印详细过程 |
| ShowMenu | 显示操作二叉排序树的菜单选项 |
|  |  |

### 2.2.2 二叉树数据结构图

****

## **2.3详细设计**

### 2.3.1 数据加载和文件操作

目标：实现从文件中读取数据并构建二叉排序树。

实现：数据加载功能通过文件流实现，首先打开文件并检查是否成功打开。然后逐行读取文件中的数据，将其存储在数组中。在读取完数据后，关闭文件流并判断数组中是否有数据。如果数组为空，则提示用户手动插入数据的选项，允许用户通过标准输入逐个插入关键字，直到输入结束标志为止。

### 2.3.2二叉排序树的基本操作

#### 2.3.2.1插入操作

目标:将新节点按照大小关系插入到正确的位置，保持二叉排序树的有序性。

实现：递归地比较节点的关键字，根据大小关系插入到左子树或右子树，保证树的有序性。插入操作从根节点开始，递归地比较要插入的关键字与当前节点的关键字大小关系。若小于当前节点，则向左子树递归插入；若大于或等于当前节点，则向右子树递归插入。当遇到空节点时，创建新节点并将其插入。

#### 2.3.2.2删除操作

目标:从二叉排序树中删除指定关键字的节点。删除操作保证了树的有序性和平衡性。

实现:删除操作首先通过查找找到要删除的节点。根据节点的子节点情况，分为以下三种情况处理：

如果待删除节点没有为叶子，则直接删除。

如果待删除节点有左子树或者右子树，则用其子树替换待删除节点。

如果待删除节点有左右子树都有，则找到其左子树中的最大节点（或右子树中的最小节点），将其值替换到待删除节点，然后删除最大节点（或最小节点）。

#### 2.3.2.3查找操作

目标:在二叉排序树中查找指定关键字的节点。

实现:递归比较节点的关键字，根据大小关系向左或向右子树查找，直到找到目标节点或遍历完整棵树。

### 2.3.3平衡二叉排序树的设计

目标:保证二叉排序树的平衡，提高插入、删除和查找操作的效率。

实现:通过折半法选择中间节点作为根节点，递归地构建左右子树，确保左右子树的节点数目相近。这种设计不仅适用于已排序数据，还适用于任意数据情况下的树构建。

### 2.3.4 平均查找长度（ASL）的计算和优化·

目标:评估二叉排序树的查找效率。

实现:ASL是评估二叉排序树查找效率的重要指标。在实现中，我们通过递归计算每个节点的深度，并记录访问过的节点。最终根据总深度和节点数计算出ASL，从而评估树的整体性能。通过分析ASL可以发现树的平衡性和节点访问分布情况，从而优化树的结构和算法，提高整体的查找效率。

通过以上设计思路，我们可以清晰地了解到二叉排序树的构建过程及其在数据管理和操作中的应用。这种数据结构和算法的设计不仅仅提供了基本的数据管理功能，还通过优化和平衡性考虑，提升了操作效率和用户体验，适用于多种实际应用场景。

## **2.4项目创新点及思考**

我们的代码在实现二叉排序树（BST）的基本功能上，包括插入、删除、查找等操作，并且在此基础上进行了优化和扩展，提高了代码的可维护性和性能。

以下是我们代码相对于原始要求的创新点：

### 2.4.1平衡二叉排序树的构建

我们的代码通过折半法构建平衡的二叉排序树，而不仅限于手动插入固定数量的数据。这种方法确保了树的高度尽可能低，提高了插入、删除和查找操作的效率。平衡二叉排序树的设计不仅适用于固定数量的数据，还适用于任意数量的数据集合，保证了树的动态性和性能。

### 2.4.2 文件I/O操作

我们的代码实现了从文件中加载数据到二叉排序树的功能，并且可以将二叉排序树的数据保存到文件中。这种实现不仅仅是简单地读取和写入数据，还包括了错误处理、用户交互和数据加载的逻辑。通过文件I/O操作，用户可以方便地管理和维护大量的数据，同时保证了数据的持久化和安全性。

### 2.4.3 平均查找长度（ASL）的计算

我们实现了计算二叉排序树的平均查找长度（ASL），并且在计算过程中详细展示了节点的访问顺序和深度信息。这种功能不仅帮助用户评估树的查找效率，还可以优化树的结构和算法，提高整体的性能和响应速度。

3.使用说明

## **3.1功能简介**

本程序提供了对二叉搜索树（Binary Search Tree, BST）的基本操作，包括创建、插入、搜索、删除、遍历、保存到文件、从文件加载以及计算树的深度和平均路径长度等功能。

## **3.2 主要功能函数说明**

CreateBalancedBST：从给定的整型数组中创建一个平衡的二叉搜索树。

LoadAndCreateBST：从指定的文件中加载数据，并创建二叉搜索树；如果文件为空，则询问用户是否手动插入数据。

InsertBST：向二叉搜索树中插入新的键值。

SearchBST：在二叉搜索树中搜索指定的键值。

DeleteBST：从二叉搜索树中删除指定的键值。

InOrderTraverse：使用中序遍历打印二叉搜索树的键值。

SaveToFile：将二叉搜索树以特定的格式保存到文件中。

LoadFromFile：从文件中读取数据并重新构建二叉搜索树。

CalculateDepth：计算二叉搜索树的深度。

CalculateASL：计算二叉搜索树的平均路径长度。

## **3.2使用步骤**

(1)打开源文件中提供的visual studio工程文件（您需要保证您的电脑上已经安装visual studio，推荐使用2022版）。

(2)运行程序，将显示功能菜单。

(3)根据菜单选择相应操作。（选择从文件加载数据需确,保文件存在且包含有效的整型数据）

4.调试及测试

## **4.1测试目的**

首先确保InsertBST、SearchBST、DeleteBST、InOrderTraverse等基本操作函数能正确执行。其次验证CalculateASL和CalculateDepth函数是否准确计算了树的平均路径长度和深度。同时测试LoadFromFile和SaveToFile函数能否正确地从文件加载和保存BST。

## **4.2测试环境**

操作系统：windows11。

集成开发环境：Visual Studio 2022。

编译器：Visual Studio 2022自带的编译器。

项目类型：命令行应用程序。

## **4.3设计测试用例**

### **4.3.1.基本测试用例**

插入一个空树，然后逐个插入元素，并验证它们是否按BST规则排序。依次插入20，15，12，13，16，50，45，32，44，65后的中序遍历结果如图4-1，可初步验证插入元素是按照BST规则排序。

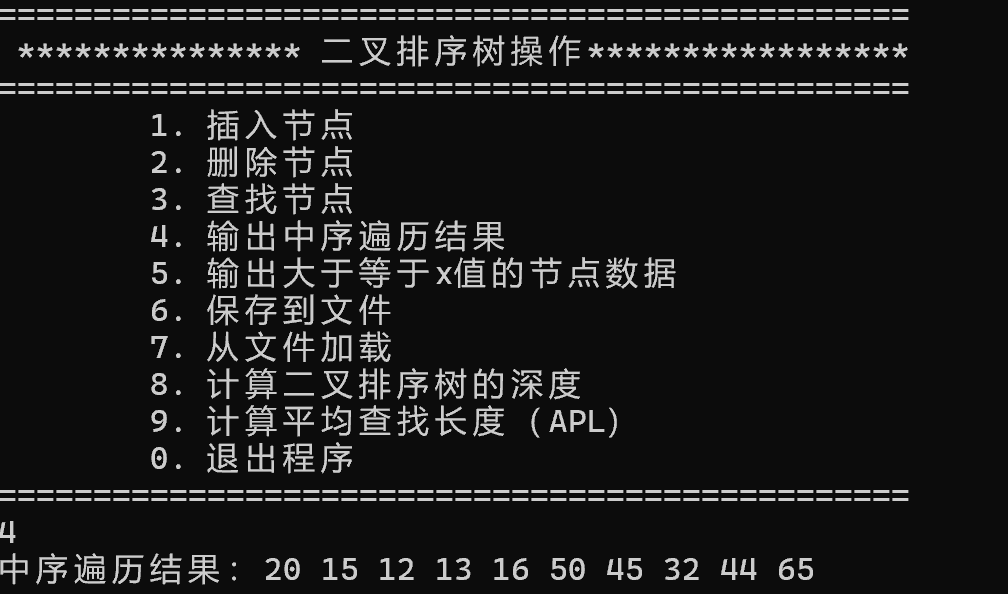


图 4-1 初步验证插入元素是按照BST规则排序

搜索上一步已插入的元素，验证查找函数是否正确返回节点或NULL。查找16和88的结果如图4-2，可以看到查找函数可以正确地返回结点和NULL。

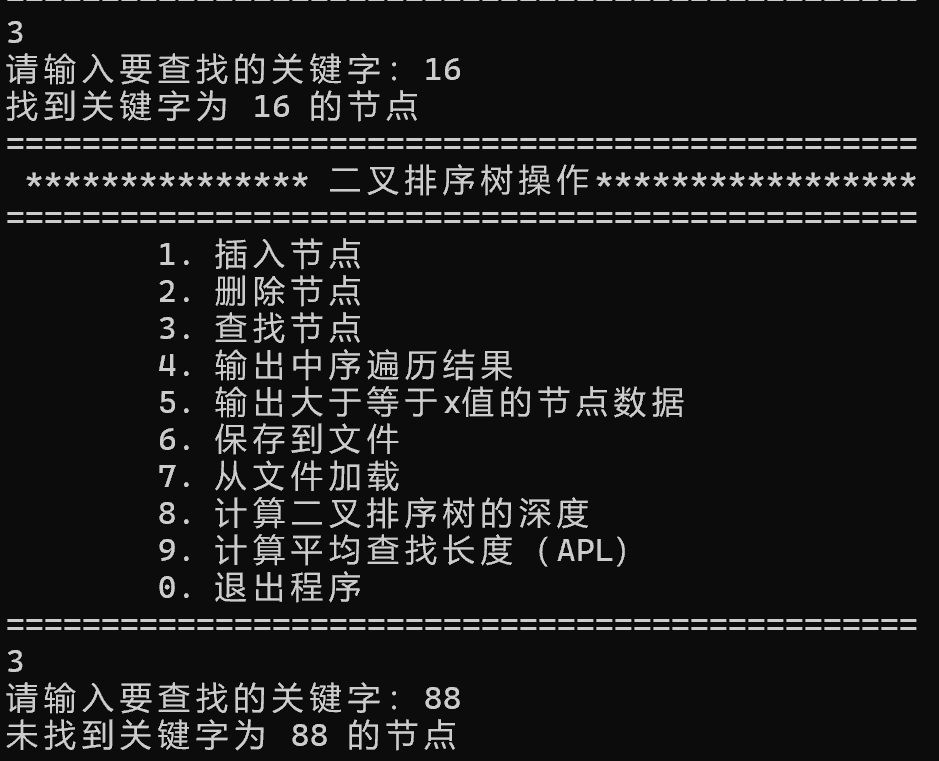
****

图 4-2验证查找函数是否正确返回节点或NULL

删除树中的一个或多个元素，验证树是否仍然保持BST性质。在已有元素的基础上删除树中的15和44，结果如图4-3，可以看到15和44都被成功删除，删除函数验证正确。

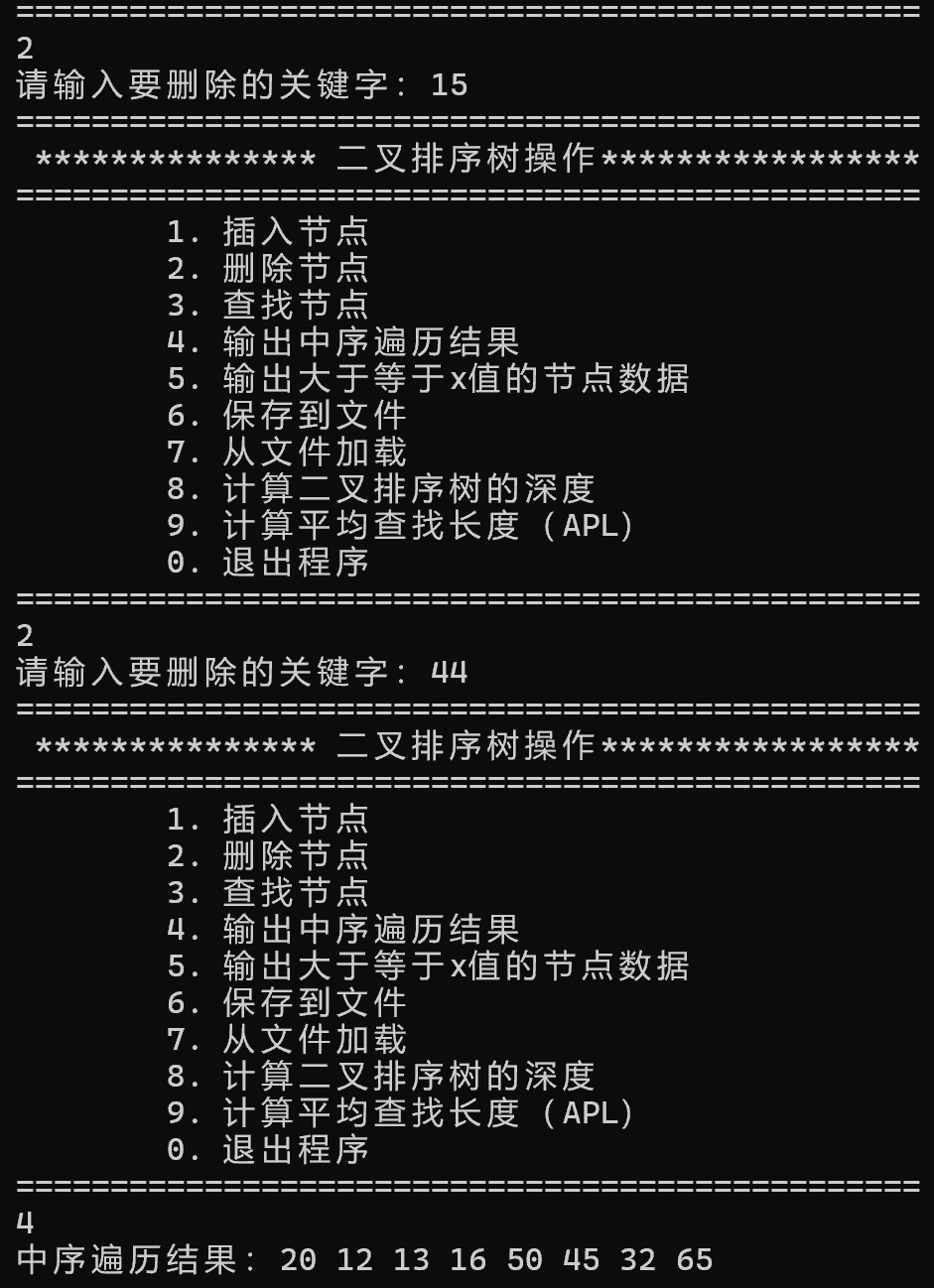
****

图 4-3删除树中的一个或多个元素，验证树是否仍然保持BST性质

### **4.3.2.边界测试用例**

插入一个只包含一个节点的树（如图4-4）。

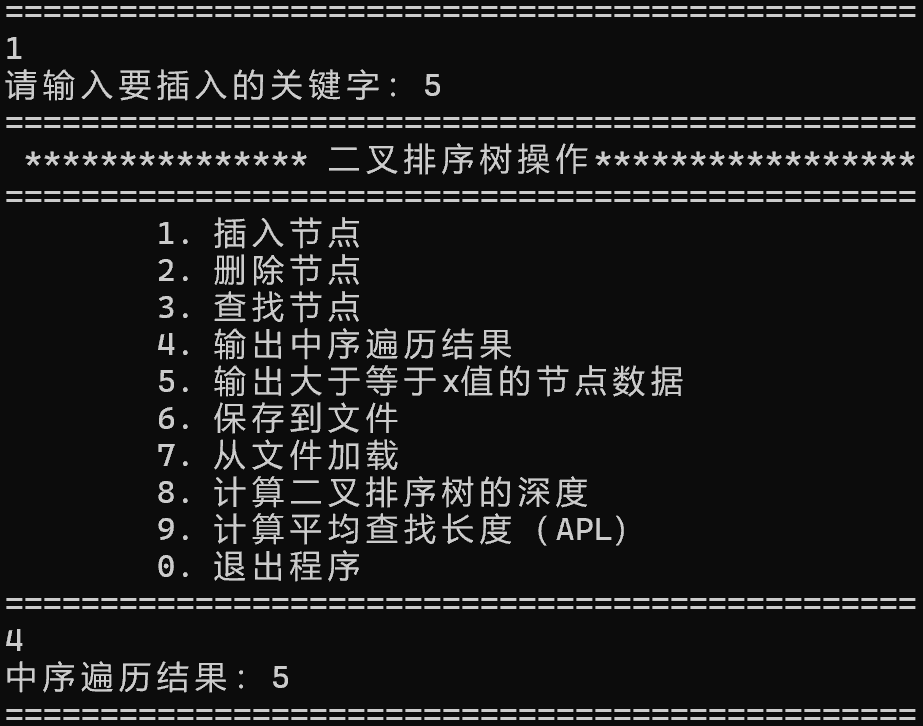
****

图 4-4插入一个只包含一个节点的树

### **4.3.3.性能测试用例**

对于大树，测试CalculateASL和CalculateDepth函数的性能。插入1,2,34,5。可知其深度为5，ASL为3。如图4-5，结果正确，则CalculateASL和CalculateDepth函数正确无误。

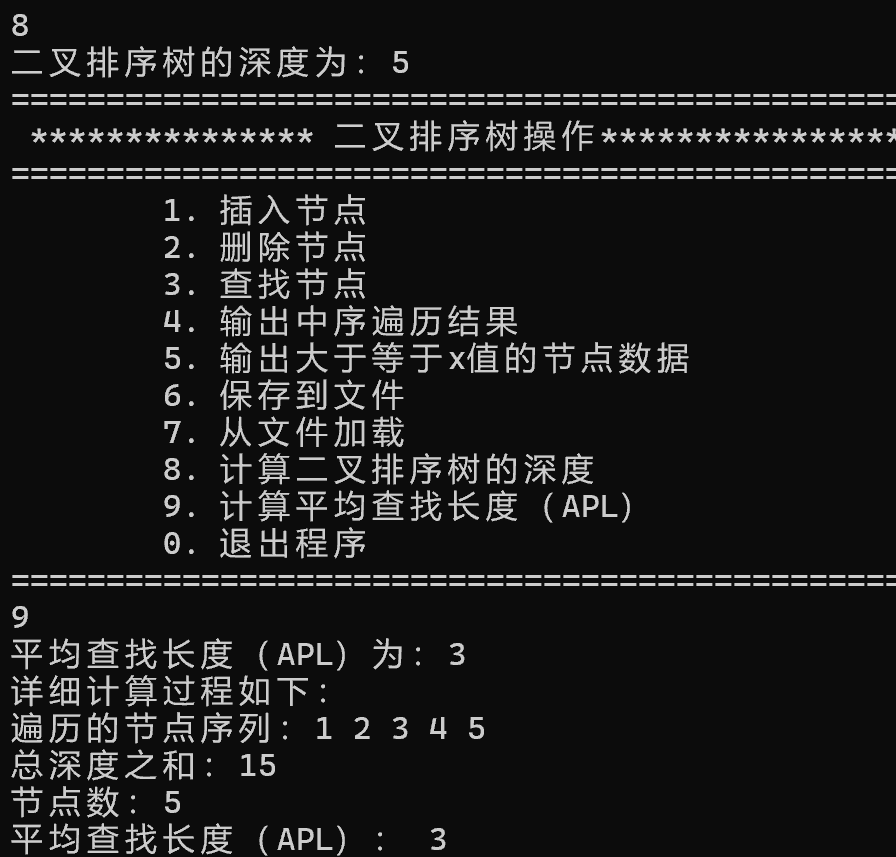
****

图 4-5测试CalculateASL和CalculateDepth函数的性能

### **4.3.4.文件操作测试用例**

创建一个包含BST数据的txt文件如（图4-6），并使用LoadFromFile加载它（图4-7）。

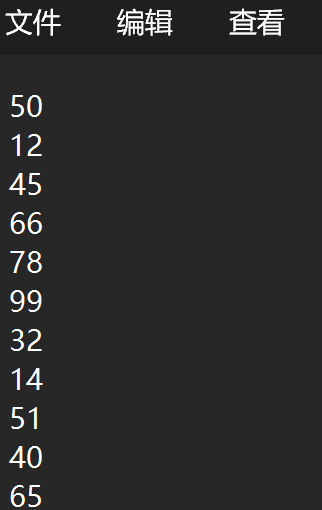
****

图 4-6创建一个包含BST数据的txt文件

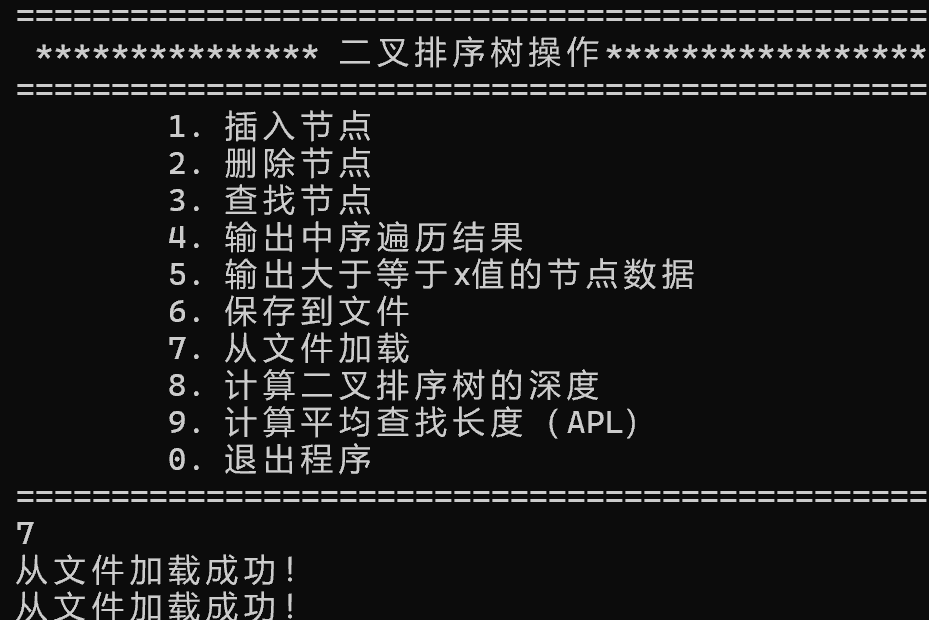
****

图 4-7使用LoadFromFile加载

验证加载后的树是否正确，如图4-8可以看到数据加载后的树的中序遍历结果正确，可推测加载后的树正确。

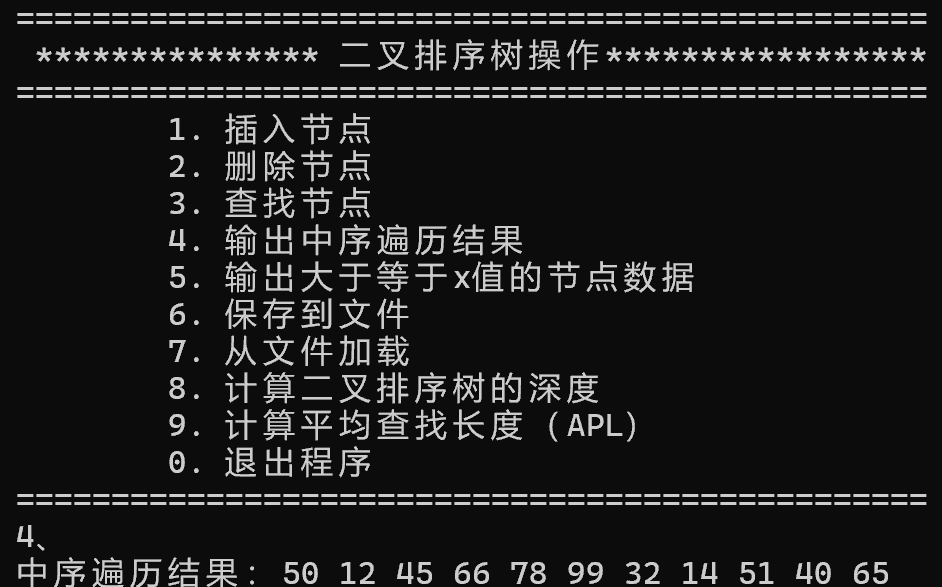
****

图 4-8加载后的树的中序遍历结果

对加载的树执行一些操作（如插入、删除、查找），然后使用SaveToFile保存。如图4-9，我们对树执行了删除操作，删除99。

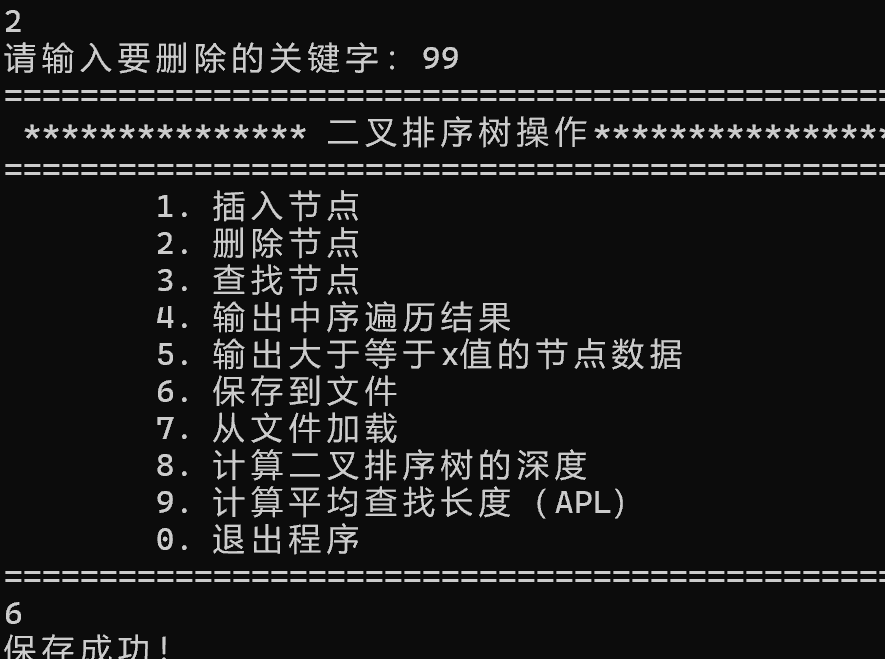
****

图 4-9对加载的树执行删除操作

验证保存的文件内容是否正确，如图4-10，数据99已被删除。

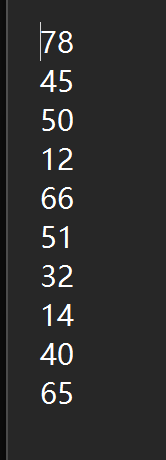
****

图 4-10保存的文件内容

## **4.4 开发中暴露的问题**

### **4.4.1初始程序与文件操作**

最初，我们的程序完全遵循了书籍上的算法，将第一个输入视为根节点，并在每次运行时创建空文件。然而，在实际应用中，我们发现如果文件已经存在且包含数据，则无需重复创建空文件。

### **4.4.2.树深度与测试**

直接从文件中读入数据构建树时，可能会导致树的深度较大，这对于测试ASL计算的正确性带来了不便。

### **4.4.3.ASL计算的复杂性与验证**

ASL的计算相对复杂，且由于树结构的多样性，验证其计算结果是否正确变得较为困难。

5.存在的问题及解决思路

## **5.1测试问题分析**

### **5.1.1 初始程序与文件操作**

在实际应用中，如果文件已经存在且包含数据，则无需重复创建空文件，为了解决这个问题，我们可以在加载文件中的数据之前先检查文件是否为空，如果文件为空，则提示用户手动插入数据；如果文件不为空，则直接从文件中读取数据并创建二叉排序树。

### **5.1.2 树深度与测试**

从文件中直接依次读入数据创建树可能导致树的深度较大，会影响ASL计算的效率和测试的便利性。为了解决树深度较大的问题，可以在构建树时使用平衡二叉排序树来保证树的深度不会过大。这样可以提高树的操作效率，同时也便于测试ASL计算的正确性。

### **5.1.3 ASL计算的复杂性与验证**

ASL的计算相对复杂，需要遍历整个树并累加每个节点的深度，验证其计算结果是否正确变得相对较困难。为了简化ASL的计算复杂性，可以运用递归算法，进行二叉排序树的深度计算，总深度除以节点数即得出平均查找长度。 同时，可以通过对数据不同的二叉排序树进行测试，验证ASL计算结果的正确性。此外，可以使用一些已知数据集和算法进行对比，以进一步验证程序的正确性。

## **5.2 解决方案**

**5.2.1 判断文件中是否有数据**

创建了LoadAndCreateBST函数，首先，打开文件流，准备从文件中读取数据。如果文件成功打开，那么它会创建一个数组data来存储从文件中读取的数据，同时设置一个索引变量index为0。然后，函数进入一个循环，不断从文件中读取数据，直到文件末尾，每读取一个数据，就将索引加1，再检查是否读取到了数据。

如果读取到了数据，那么它会调用CreateBalancedBST函数，将这些数据转换为一个平衡的二叉排序树，再将结果赋值给T，并输出一条消息，表示数据加载成功。

如果没有读取到数据，会提示用户是否要手动插入数据。如果用户选择是，那么函数会提示用户输入要插入的关键字，并将这些关键字插入到二叉排序树中。最后，输出一条消息，表示数据插入成功。

**5.2.2 创建平衡二叉排序树**

创建了CreateBalancedBST函数， 首先，函数检查起始下标是否大于结束下标，如果是，那么返回空指针。再计算中间下标，并创建一个新的节结点，将数据数组中中间位置的元素赋值给节点的关键字，然后，函数递归调用自身，分别创建左子树和右子树。左子树的起始下标为start，结束下标为mid - 1；右子树的起始下标为mid + 1，结束下标为end；最后，函数返回创建好的节点。

这样，以中间结点为根结点建立二叉排序树，可以保证每个节点的左右子树的高度差不超过1，从而使得整棵树保持平衡状态，不仅大大减小了树的深度，也提高ASL的计算效率。

**5.2.3 打印ASL的计算过程**

创建了CalculateASLAndPrint函数，打印出计算二叉排序树ASL的详细过程。首先判断当前节点是否为空，如果不为空，则将当前结点的关键字记录到keysVisited数组中（记录访问过的结点），并将累计深度和结点计数分别加上当前结点的深度和1；然后递归地遍历左子树和右子树，同时更新累计深度和结点计数。

通过这个辅助函数，我们可以方便地验证ASL的准确性，并且可以查看每个结点的访问顺序和累计深度的变化情况。

6.附录

**6.1源程序**

1. #include <iostream>
2. #include <fstream>
3. #include <algorithm>
4. using namespace std;
5. #define ENDFLAG 99999 // 定义结束标志，用于手动插入数据时的输入结束
6. typedef int KeyType; // 定义关键字类型为整型
7. typedef struct BSTNode {
8. KeyType key; // 每个节点的关键字
9. struct BSTNode\* lchild, \* rchild; // 左右孩子指针
10. } BSTNode, \* BSTree;
11. // 插入节点到二叉排序树中
12. void InsertBST(BSTree& T, KeyType key);
13. // 中序遍历二叉排序树
14. void InOrderTraverse(BSTree T);
15. // 在二叉排序树中查找关键字为key的结点
16. BSTree SearchBST(BSTree T, KeyType key);
17. // 删除二叉排序树中的结点
18. void DeleteBST(BSTree& T, KeyType key);
19. // 输出所有大于等于给定x值的结点数据
20. void PrintGreaterOrEqual(BSTree T, KeyType x);
21. // 将二叉排序树保存到文件中
22. void SaveToFile(BSTree T, ofstream& fout);
23. // 从文件中加载二叉排序树
24. void LoadFromFile(BSTree& T, ifstream& fin);
25. // 计算二叉排序树的深度
26. int CalculateDepth(BSTree T);
27. // 计算二叉排序树的平均查找长度（ASL）
28. double CalculateASL(BSTree T);
29. // 辅助函数：计算ASL并打印详细过程
30. void CalculateASLAndPrint(BSTree T, int depth, int& totalDepth, int& nodeCount, KeyType\* keysVisited);
31. // 显示菜单
32. void ShowMenu();
33. // 通过数据创建平衡的二叉排序树
34. BSTree CreateBalancedBST(KeyType\* data, int start, int end) {
35. if (start > end) {
36. return NULL;
37. }
38. int mid = (start + end) / 2;
39. BSTree node = new BSTNode();
40. node->key = data[mid];
41. node->lchild = CreateBalancedBST(data, start, mid - 1);
42. node->rchild = CreateBalancedBST(data, mid + 1, end);
43. return node;
44. }
45. // 从文件中加载并创建二叉排序树
46. void LoadAndCreateBST(BSTree& T, const string& filename) {
47. ifstream fin(filename); // 打开文件流，准备从文件中读取数据
48. if (fin.is\_open()) { // 如果文件成功打开
49. KeyType data[100]; // 假设最多存储100个数据
50. int index = 0; // 数据索引
51. // 从文件中读取数据直到文件末尾
52. while (fin >> data[index]) {
53. index++;
54. }
55. fin.close(); // 关闭文件流
56. if (index > 0) { // 如果文件中有数据
57. // 对读取的数据进行排序
58. sort(data, data + index);
59. T = CreateBalancedBST(data, 0, index - 1); // 创建平衡的二叉排序树
60. cout << "从文件加载成功！" << endl; // 输出加载成功消息
61. }
62. else { // 如果文件中没有数据
63. cout << "文件中没有数据。" << endl; // 输出文件无数据消息
64. cout << "是否手动插入数据？(y/n): "; // 提示用户手动插入数据的选择
65. char choice; // 用户选择变量
66. cin >> choice; // 输入选择
67. if (choice == 'y' || choice == 'Y') { // 如果用户选择是
68. cout << "请输入要插入的关键字（以 " << ENDFLAG << " 结束输入）：" << endl; // 提示用户输入关键字
69. KeyType key; // 插入关键字变量
70. while (true) { // 循环读取用户输入直到结束标志
71. cin >> key;
72. if (key == ENDFLAG) break;
73. InsertBST(T, key); // 将输入的关键字插入二叉排序树中
74. }
75. cout << "插入成功！" << endl; // 输出插入成功消息
76. }
77. }
78. }
79. else { // 如果文件打开失败
80. cout << "无法打开文件 " << filename << endl; // 输出无法打开文件消息
81. }
82. }
83. // 中序遍历二叉排序树
84. void InOrderTraverse(BSTree T) {
85. if (T != NULL) { // 如果树非空
86. InOrderTraverse(T->lchild); // 遍历左子树
87. cout << T->key << " "; // 输出当前节点的关键字
88. InOrderTraverse(T->rchild); // 遍历右子树
89. }
90. }
91. // 查找结点
92. BSTree SearchBST(BSTree T, KeyType key) {
93. if (!T || key == T->key) return T; // 如果树为空或者找到目标节点，返回该节点
94. else if (key < T->key) return SearchBST(T->lchild, key); // 如果目标值小于当前节点值，在左子树中查找
95. else return SearchBST(T->rchild, key); // 如果目标值大于当前节点值，在右子树中查找
96. }
97. // 插入结点
98. void InsertBST(BSTree& T, KeyType key) {
99. if (!T) { // 如果树为空
100. BSTNode\* S = new BSTNode; // 创建新节点
101. S->key = key; // 设置节点关键字
102. S->lchild = S->rchild = NULL; // 设置左右子树为空
103. T = S; // 将新节点作为根节点
104. }
105. else if (key < T->key) { // 如果插入值小于当前节点值，插入左子树
106. InsertBST(T->lchild, key);
107. }
108. else { // 如果插入值大于等于当前节点值，插入右子树
109. InsertBST(T->rchild, key);
110. }
111. }
112. // 删除结点
113. void DeleteBST(BSTree& T, KeyType key) {
114. BSTNode\* p = T; // p指向待删除节点
115. BSTNode\* f = NULL; // f指向p的父节点
116. while (p) { // 找到待删除节点p和其父节点f
117. if (p->key == key)
118. break;
119. f = p;
120. if (p->key > key)
121. p = p->lchild;
122. else
123. p = p->rchild;
124. }
125. if (!p) return; // 如果未找到待删除节点，直接返回
126. BSTNode\* q = p; // q指向实际删除的节点
127. if (p->lchild && p->rchild) { // 如果p有左右孩子
128. BSTNode\* s = p->lchild; // s指向p的左孩子
129. while (s->rchild) { // 找到s的右子树的最右节点
130. q = s;
131. s = s->rchild;
132. }
133. p->key = s->key; // 将s的值赋给p
134. if (q != p)
135. q->rchild = s->lchild;
136. else
137. q->lchild = s->lchild;
138. delete s; // 删除s节点
139. return;
140. }
141. else if (!p->rchild) { // 如果p没有右孩子
142. p = p->lchild; // 将p的左孩子赋给p
143. }
144. else if (!p->lchild) { // 如果p没有左孩子
145. p = p->rchild; // 将p的右孩子赋给p
146. }
147. if (!f) // 如果p是根节点
148. T = p; // 直接将p作为新的根节点
149. else if (q == f->lchild) // 如果p是f的左孩子
150. f->lchild = p; // 将p挂在f的左边
151. else
152. f->rchild = p; // 将p挂在f的右边
153. delete q; // 删除q节点
154. }
155. // 输出所有大于等于给定x值的结点数据
156. void PrintGreaterOrEqual(BSTree T, KeyType x) {
157. if (T != NULL) { // 如果树非空
158. PrintGreaterOrEqual(T->lchild, x); // 遍历左子树
159. if (T->key >= x) { // 如果当前节点的值大于等于x
160. cout << T->key << " "; // 输出当前节点的值
161. }
162. PrintGreaterOrEqual(T->rchild, x); // 遍历右子树
163. }
164. }
165. // 将二叉排序树保存到文件中
166. void SaveToFile(BSTree T, ofstream& fout) {
167. if (T != NULL) { // 如果树非空
168. fout << T->key << endl; // 将当前节点的值写入文件
169. SaveToFile(T->lchild, fout); // 保存左子树
170. SaveToFile(T->rchild, fout); // 保存右子树
171. }
172. }
173. // 从文件中加载二叉排序树
174. void LoadFromFile(BSTree& T, ifstream& fin) {
175. T = NULL; // 初始化树为空
176. KeyType key; // 用于存储从文件中读取的值
177. while (fin >> key) { // 从文件中读取值
178. InsertBST(T, key); // 插入树中
179. }
180. }
181. // 计算二叉排序树的深度
182. int CalculateDepth(BSTree T) {
183. if (T == NULL) return 0; // 如果树为空，深度为0
184. int lDepth = CalculateDepth(T->lchild); // 计算左子树深度
185. int rDepth = CalculateDepth(T->rchild); // 计算右子树深度
186. return max(lDepth, rDepth) + 1; // 树的深度为左右子树深度的较大值加1
187. }
188. // 辅助函数：计算ASL并打印详细过程
189. void CalculateASLAndPrint(BSTree T, int depth, int& totalDepth, int& nodeCount, KeyType\* keysVisited) {
190. if (T != NULL) { // 如果树非空
191. keysVisited[nodeCount] = T->key; // 记录访问的关键字
192. totalDepth += depth; // 累加深度
193. nodeCount++; // 结点数加1
194. CalculateASLAndPrint(T->lchild, depth + 1, totalDepth, nodeCount, keysVisited); // 递归处理左子树
195. CalculateASLAndPrint(T->rchild, depth + 1, totalDepth, nodeCount, keysVisited); // 递归处理右子树
196. }
197. }
198. // 计算二叉排序树的平均查找长度（ASL）
199. double CalculateASL(BSTree T) {
200. int totalDepth = 0; // 总深度
201. int nodeCount = 0; // 结点数
202. KeyType keysVisited[100]; // 记录访问的关键字
203. CalculateASLAndPrint(T, 1, totalDepth, nodeCount, keysVisited); // 计算总深度和结点数
204. double ASL = (double)totalDepth / nodeCount; // 计算平均查找长度
205. cout << "节点访问顺序：";
206. for (int i = 0; i < nodeCount; i++) {
207. cout << keysVisited[i] << " "; // 输出访问的关键字
208. }
209. cout << endl;
210. cout << "总查找长度：" << totalDepth << ", 节点总数：" << nodeCount << endl; // 输出总深度和结点数
211. return ASL; // 返回平均查找长度
212. }
213. // 显示菜单
214. void ShowMenu() {
215. cout << "================================================" << endl;
216. cout << "| \*\*\*\*\*\* 二叉排序树操作 \*\*\*\*\*\* |" << endl;
217. cout << "================================================" << endl;
218. cout << "| |" << endl;
219. cout << "| 1. 插入节点 |" << endl;
220. cout << "| 2. 删除节点 |" << endl;
221. cout << "| 3. 查找节点 |" << endl;
222. cout << "| 4. 输出中序遍历结果 |" << endl;
223. cout << "| 5. 输出大于等于x值的节点数据 |" << endl;
224. cout << "| 6. 保存到文件 |" << endl;
225. cout << "| 7. 从文件加载 |" << endl;
226. cout << "| 8. 计算二叉排序树的深度 |" << endl;
227. cout << "| 9. 计算平均查找长度（ASL） |" << endl;
228. cout << "| 0. 退出程序 |" << endl;
229. cout << "| |" << endl;
230. cout << "================================================" << endl;
231. }
232. // 主函数
233. int main() {
234. BSTree T = NULL; // 初始化树为空
235. int choice; // 用户选择
236. KeyType key, x; // 关键字和给定值
237. do {
238. ShowMenu(); // 显示菜单
239. cout << "请选择操作："; // 提示用户选择
240. cin >> choice; // 输入选择
241. switch (choice) {
242. case 1:
243. cout << "请输入要插入的关键字：";
244. cin >> key;
245. InsertBST(T, key);
246. break;
247. case 2:
248. cout << "请输入要删除的关键字：";
249. cin >> key;
250. DeleteBST(T, key);
251. break;
252. case 3:
253. cout << "请输入要查找的关键字：";
254. cin >> key;
255. if (SearchBST(T, key))
256. cout << "找到了关键字为 " << key << " 的结点。" << endl;
257. else
258. cout << "没有找到关键字为 " << key << " 的结点。" << endl;
259. break;
260. case 4:
261. InOrderTraverse(T);
262. cout << endl;
263. break;
264. case 5:
265. cout << "请输入给定值 x：";
266. cin >> x;
267. PrintGreaterOrEqual(T, x);
268. cout << endl;
269. break;
270. case 6: {
271. ofstream fout("bstree.txt");
272. if (fout.is\_open()) {
273. SaveToFile(T, fout);
274. fout.close();
275. cout << "保存成功！" << endl;
276. }
277. else {
278. cout << "无法打开文件保存数据。" << endl;
279. }
280. break;
281. }
282. case 7:
283. LoadAndCreateBST(T, "bstree.txt");
284. break;
285. case 8:
286. cout << "二叉排序树的深度为：" << CalculateDepth(T) << endl;
287. break;
288. case 9:
289. cout << "平均查找长度（ASL）为：" << CalculateASL(T) << endl;
290. break;
291. case 0:
292. cout << "退出程序。" << endl;
293. break;
294. default:
295. cout << "无效的选择，请重新输入。" << endl;
296. }
297. } while (choice != 0);
298. return 0;
299. }

**6.2 测试数据**

### **6.2.1文件数据**

创建文件，依次输入（注意换行）1,3,9,6,7,10,11,20,16,15,17,23,21,33,34。

### **6.2.2.基础操作数据**

（1）插入数据：8。

（2）删除数据：4。

（3）查找数据：3。

**6.3 运行结果**

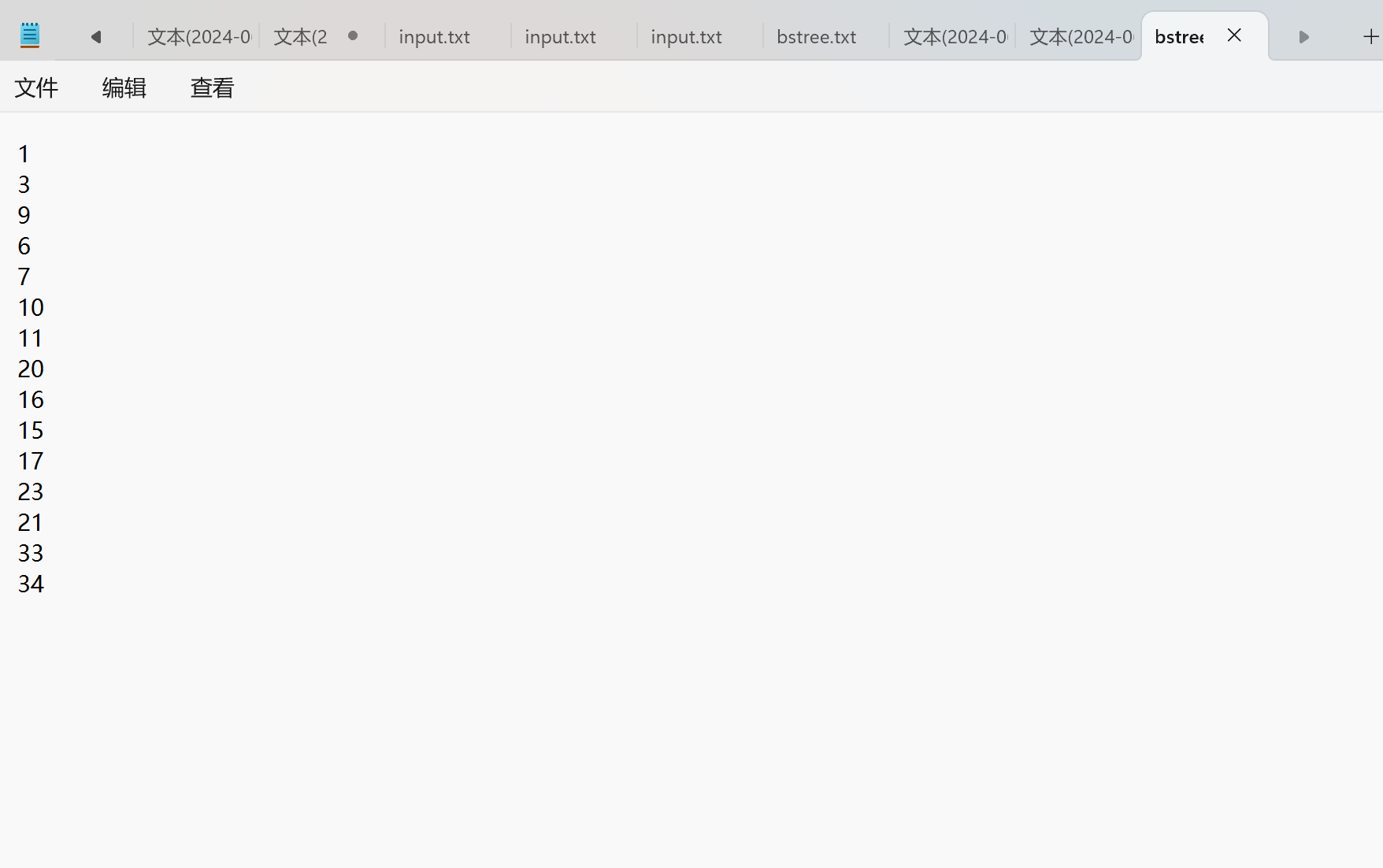


图6-1 bstree文件内容创建图

（1）操作7+操作4 可打印出中序遍历结果（所有操作都以操作7为基础）

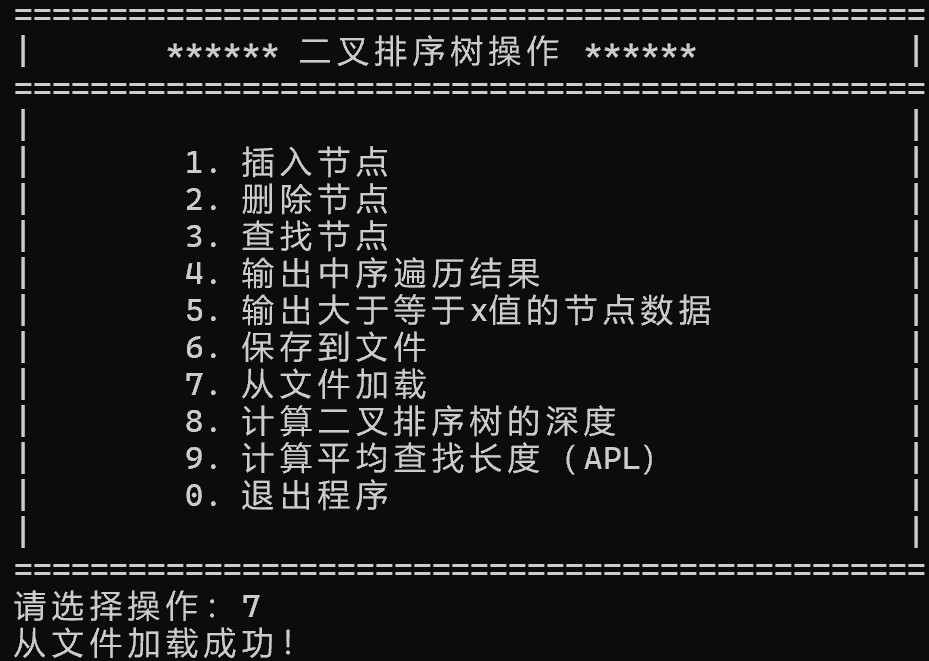


图6-2 操作7的运行结果



图6-3 操作4的运行结果

（2）选择操作5



图6-4 操作5的运行结果

（3）选择操作8（项目创新点）

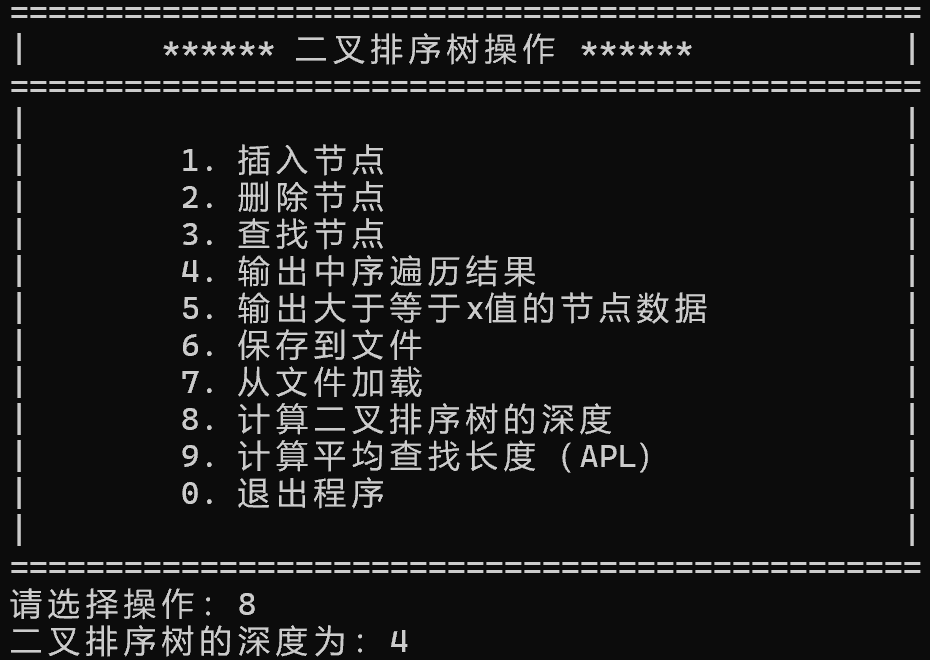


图6-5 操作8的运行结果

（4）选择操作9（项目创新点）

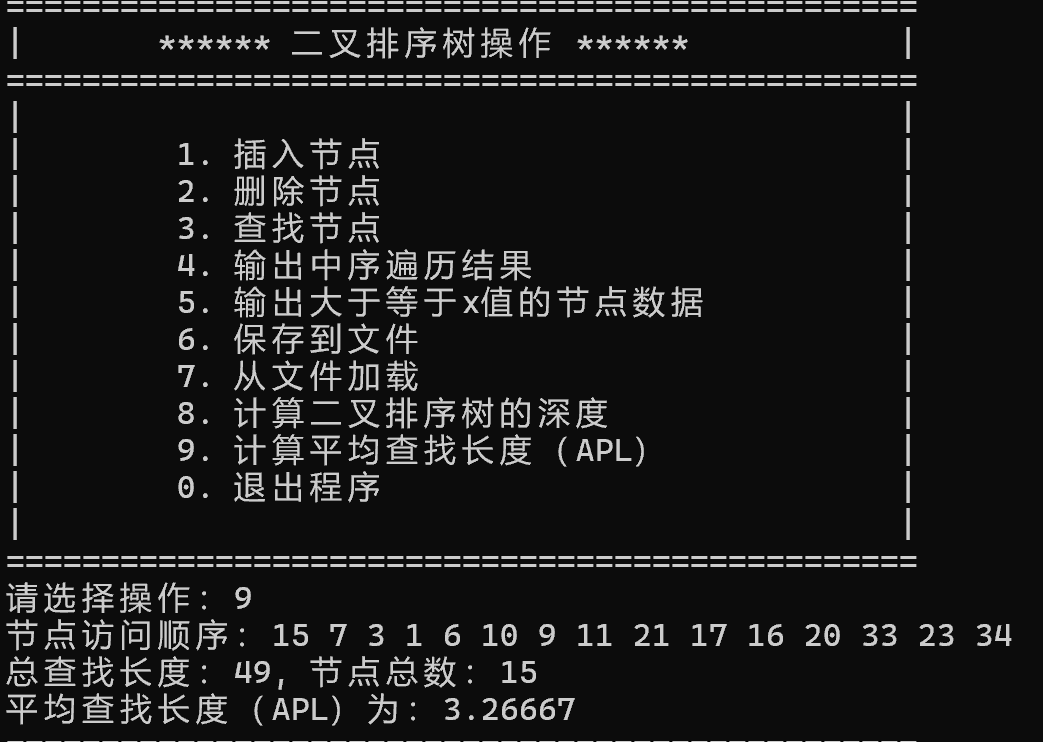


图6-6 操作9的运行结果

（5）选择操作1（1+4即可显示出插入节点之后的二叉树）

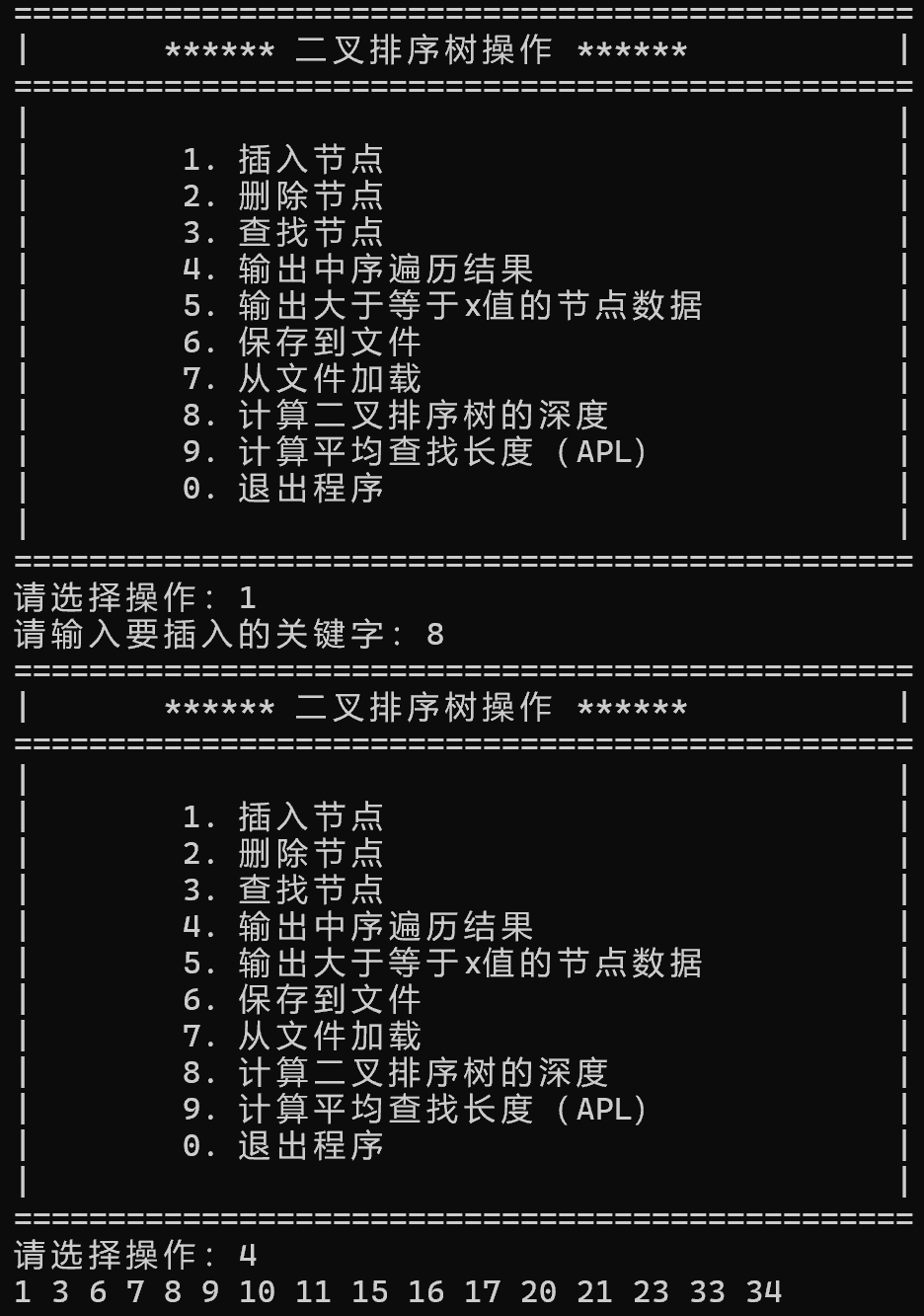


图6-7 操作1+4的运行结果

（6）选择操作2（2+4即可显示出删除节点之后的二叉树）

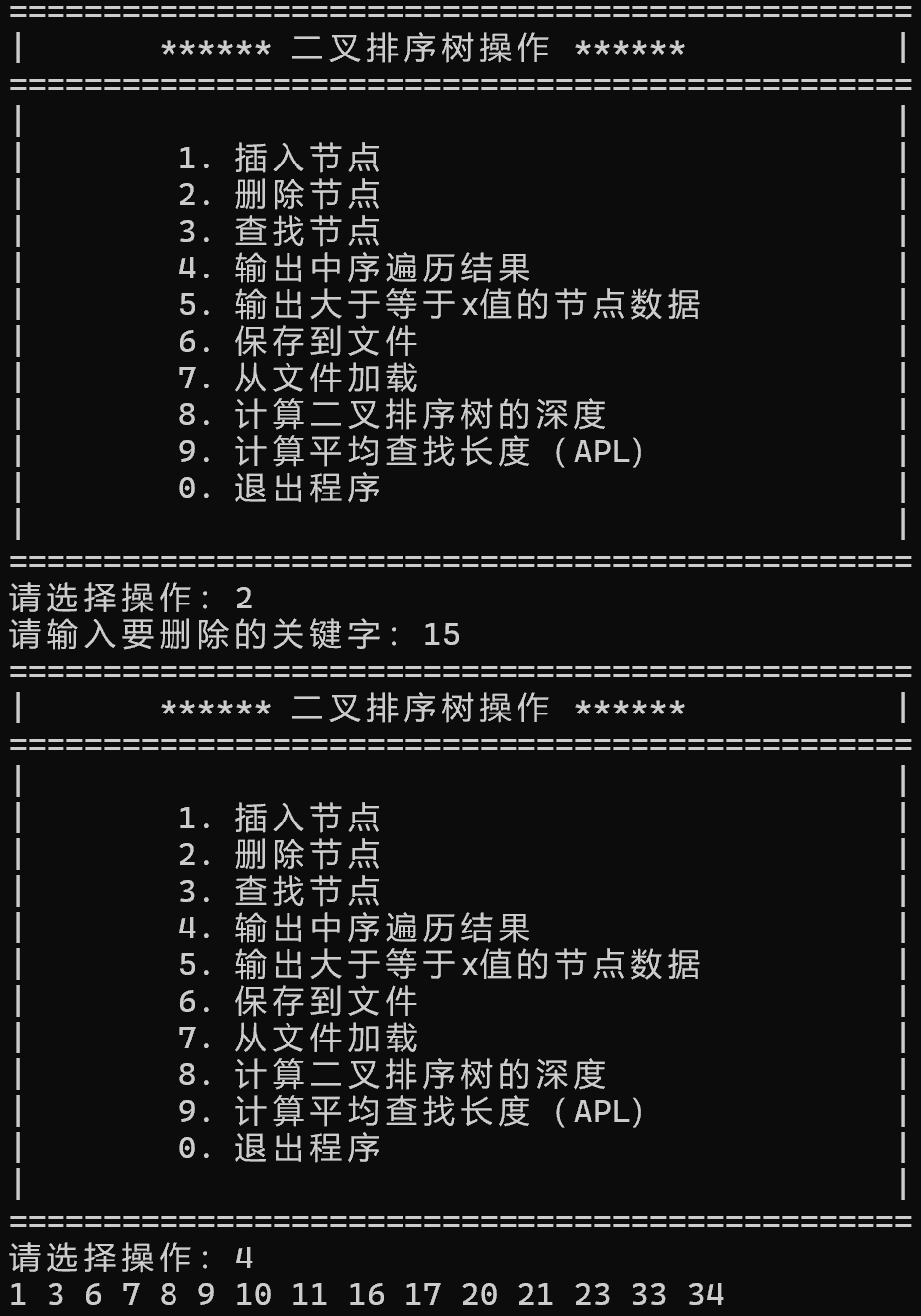


图6-8 操作2+4的运行结果

（7）选择操作3

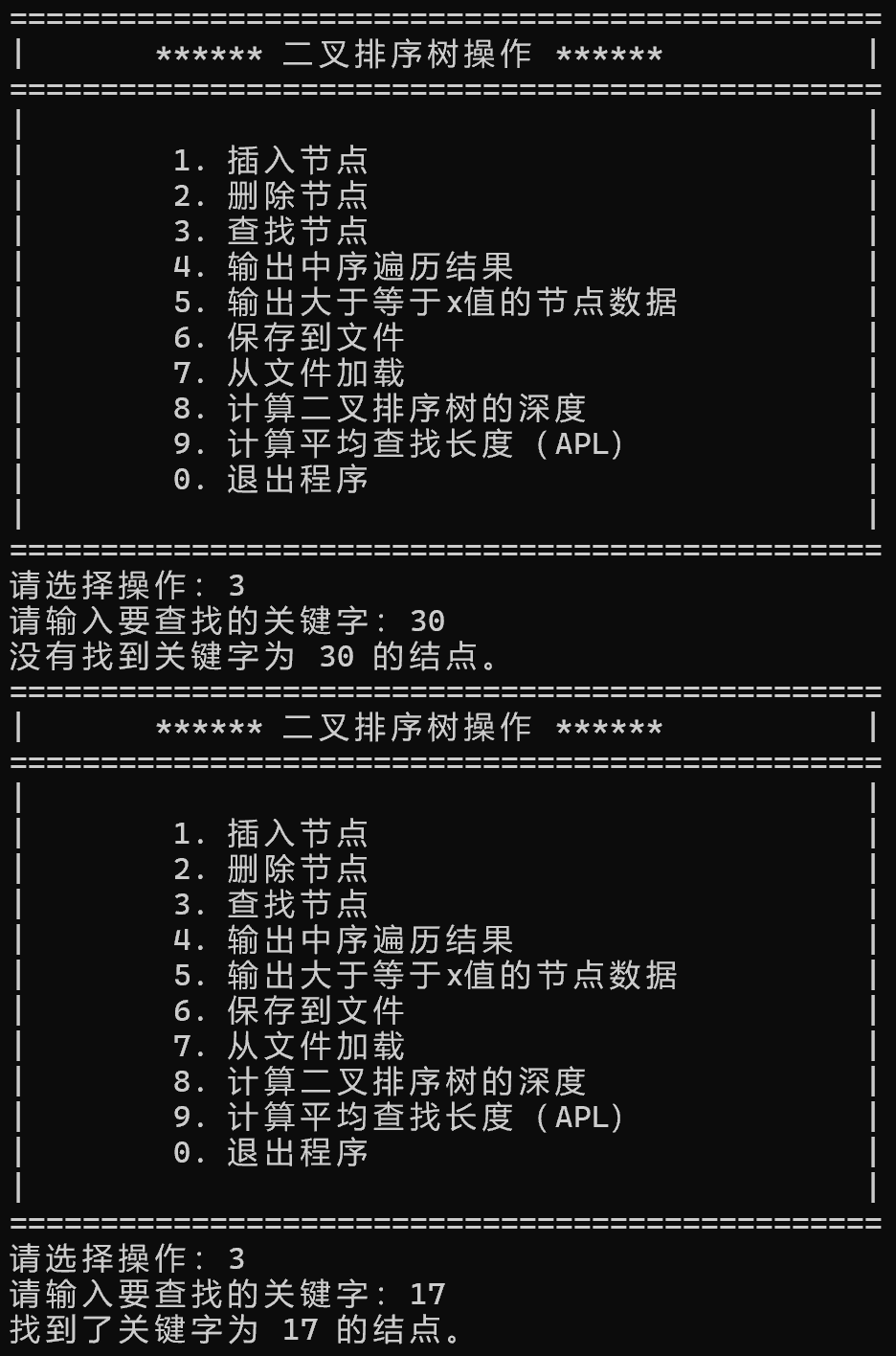


图6-9 操作3的运行结果(有关键字的结果和无关键字的结果）

（8）选择操作6

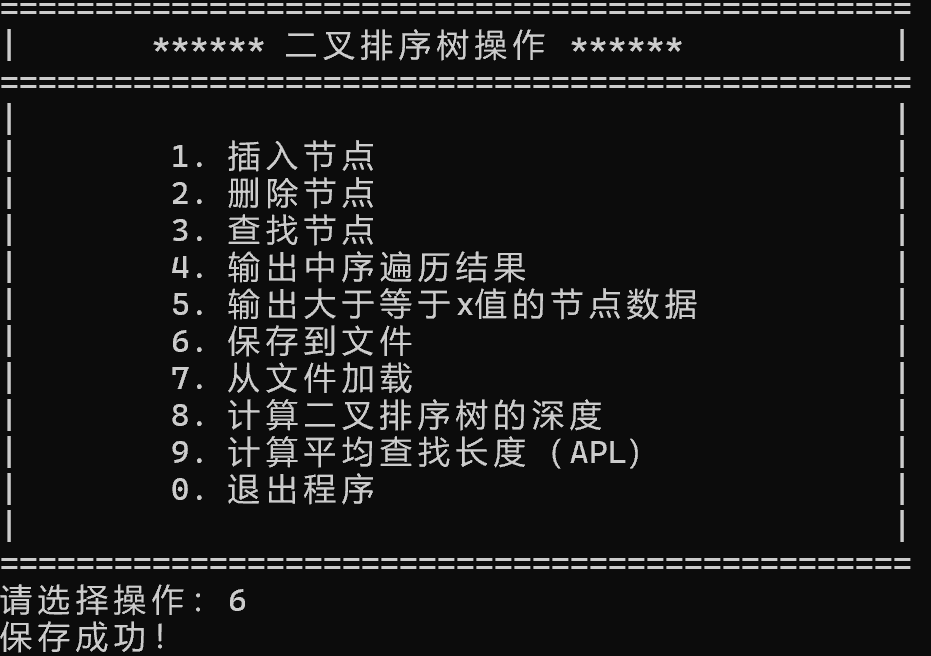


图6-10 操作6的运行结果