数据结构课程设计

项目说明文档

二叉排序树

|  |  |
| --- | --- |
| 作者姓名： | 杨烜赫 |
| 学 号： | 2252709 |
| 指导教师： | 张 颖 |
| 学院专业： | 软件学院 软件工程 |



同济大学

Tongji University

1. **题目简介**

**1.实验背景**

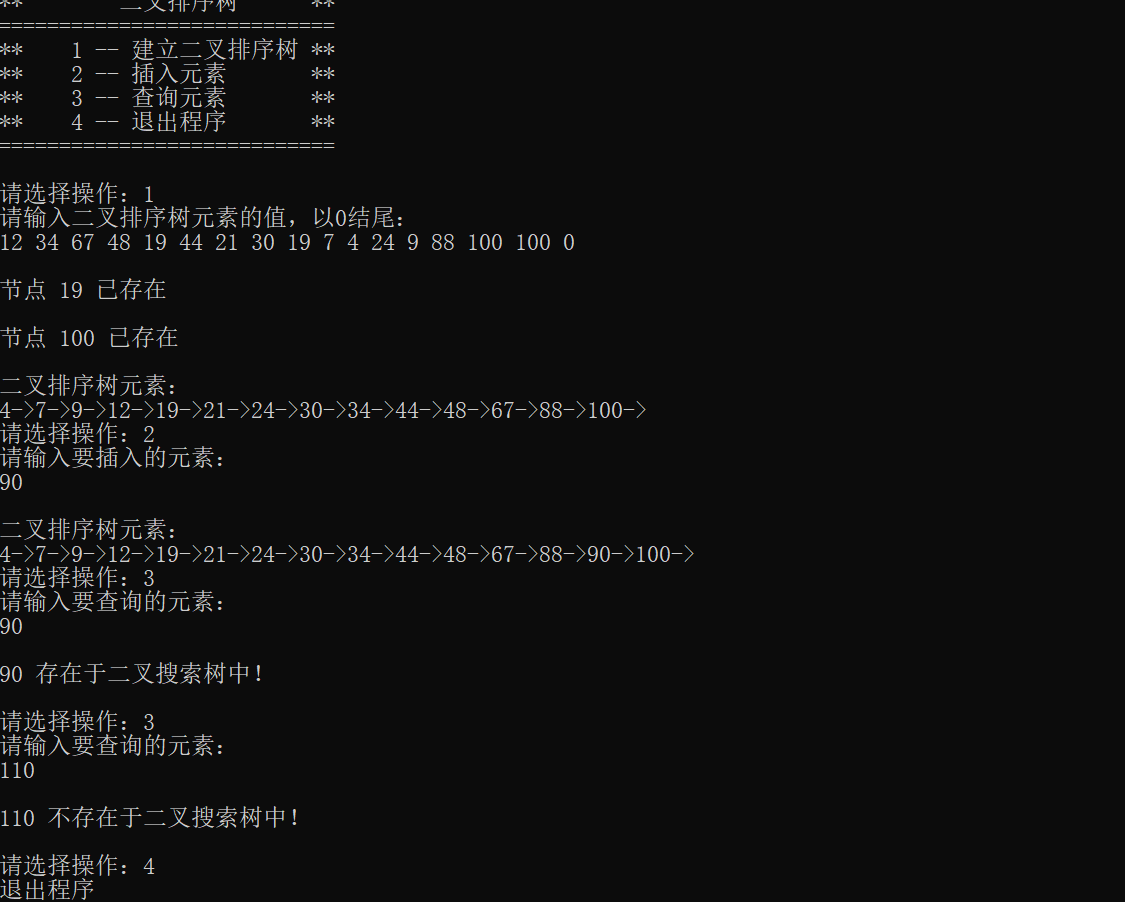
二叉排序树就是指将原来已有的数据根据大小构成一棵二叉树，二叉树中的所有结点数据满足一定的大小关系，所有的左子树中的结点均比根结点小，所有的右子树的结点均比根结点大。

二叉排序树查找是指按照二叉排序树中结点的关系进行查找，查找关键自首先同根结点进行比较，如果相等则查找成功；如果比根节点小，则在左子树中查找；如果比根结点大，则在右子树中进行查找。这种查找方法可以快速缩小查找范围，大大减少查找关键的比较次数，从而提高查找的效率。

**2.实验功能**

依次输入关键字并建立二叉排序树，实现二叉排序数的插入和查找功能。

**3.项目示例**

****

1. **项目设计**

**1.数据结构设计**

**a. BST\_Node（二叉搜索树节点）**

数据成员:

value: 存储节点的值（double 类型）。

left\_child: 指向左子节点的指针（BST\_Node\* 类型）。

right\_child: 指向右子节点的指针（BST\_Node\* 类型）。

**b. Binary\_Search\_Tree（二叉搜索树）**

数据成员:

root: 指向树的根节点的指针（BST\_Node\* 类型）。

**2.类结构设计**

**a. BST\_Node 类**

构造函数:

BST\_Node(double val = 0.0): 初始化节点值和子节点指针。

b. Binary\_Search\_Tree 类

构造函数:

Binary\_Search\_Tree(): 初始化根节点为NULL。

析构函数:

~Binary\_Search\_Tree(): 调用Clear\_Tree以清空整棵树。

**3.成员与操作设计**

**a. Binary\_Search\_Tree 类成员函数**

Insert\_Value(double key): 插入新值到树中。

Find\_Value(BST\_Node\* bst, double key): 在树中查找给定值。

Display\_InOrder(BST\_Node\* node): 中序遍历树并打印节点值。

Get\_Root(): 获取根节点。

Clear\_Tree(BST\_Node\* node): 递归删除树的所有节点。

Create\_Root\_Node(double key): 创建根节点。

Insert\_Node(BST\_Node\* bst, double key): 辅助函数，用于在适当位置插入新节点。

**b. 重载的操作符**

operator<<: 重载<<运算符以方便打印BST\_Node对象。

c. 辅助函数

GetUserChoice(): 获取并返回用户的输入选择。

## 4.算法设计

本程序的算法设计主要围绕二叉搜索树（BST）的基本操作，包括树的创建、节点的插入、值的查找和树的遍历。首先，算法通过Insert\_Value方法实现节点的插入，该方法首先检查是否存在根节点，不存在则创建根节点，存在则调用Insert\_Node递归地将新节点插入到合适的位置，保持二叉搜索树的特性（左子节点值小于父节点，右子节点值大于父节点）。其次，Find\_Value方法用于查找特定值的节点，递归地在树中搜索直到找到匹配的节点或遍历完树。然后，Display\_InOrder方法通过中序遍历打印出树中所有节点的值，展现树的结构。最后，Clear\_Tree方法递归地删除整棵树的所有节点，释放分配的内存。整体上，这些算法设计体现了二叉搜索树操作的经典方法，有效地实现了节点的插入、搜索和遍历功能，同时保证了数据结构的完整性和内存的安全管理。

1. **项目实现**

**3.1. 插入新节点**

1. **bool** Binary\_Search\_Tree::Insert\_Value(**double** key) {
2. **if** (root == NULL) {
3. Create\_Root\_Node(key);
4. **return** **true**;
5. }
6. **if** (Find\_Value(root, key) == NULL) {
7. Insert\_Node(root, key);
8. **return** **true**;
9. }
10. **else** {
11. cout << "\n节点 " << key << " 已存在" << endl;
12. **return** **false**;
13. }
14. }

这段代码是二叉搜索树（BST）中的Insert\_Value函数，它的主要功能是将一个新的键值（key）插入到二叉搜索树中。该函数首先检查树的根节点（root）是否为空；如果是，它创建一个新的根节点，并将传入的键值作为该节点的值。如果根节点已存在，则函数调用Find\_Value来检查要插入的键值是否已经存在于树中。如果该键值不存在，则通过调用Insert\_Node函数将新键值递归地插入到正确的位置，以保持二叉搜索树的有序性质。如果发现键值已存在于树中，函数将输出一条消息指出该节点已存在，并返回false。这样，Insert\_Value函数确保了二叉搜索树中不会有重复的元素，并且维护了其结构的完整性。

**3.2. 查找值**

1. BST\_Node\* Binary\_Search\_Tree::Find\_Value(BST\_Node\* bst, **double** key) {
2. **if** (bst == NULL) **return** NULL;
3. **if** (bst->value == key) **return** bst;
4. **if** (bst->value < key) **return** Find\_Value(bst->right\_child, key);
5. **return** Find\_Value(bst->left\_child, key);
6. }

这段代码定义了Find\_Value函数，其功能是在二叉搜索树中递归地查找一个给定的键值（key）。该函数首先检查当前节点是否为NULL，如果是，则返回NULL表示键值未找到。若当前节点的值与搜索的键值相等，则返回该节点。如果键值大于当前节点的值，函数递归地在右子树中搜索；反之，在左子树中搜索。这个过程利用了二叉搜索树的特性，即左子树的所有值小于节点值，右子树的所有值大于节点值，从而高效地定位所需的键值。

**3.3. 中序遍历与清空树功能实现**

1. **void** Binary\_Search\_Tree::Display\_InOrder(BST\_Node\* node) {
2. **if** (node == NULL) **return**;
3. Display\_InOrder(node->left\_child);
4. cout << \*node;
5. Display\_InOrder(node->right\_child);
6. }
7. **void** Binary\_Search\_Tree::Clear\_Tree(BST\_Node\* node) {
8. **if** (node == NULL) **return**;
9. Clear\_Tree(node->left\_child);
10. Clear\_Tree(node->right\_child);
11. **delete** node;
12. }

这两个函数是二叉搜索树（BST）类的成员方法，用于树的遍历和清理。Display\_InOrder方法实现了中序遍历，它递归地首先访问节点的左子树，然后输出节点本身的值，最后访问右子树，从而按升序打印出树中所有的节点值。另一方面，Clear\_Tree方法用于递归地清空整个树，它首先访问并删除左子树的所有节点，然后是右子树，最后删除根节点本身，从而安全地释放分配给树中所有节点的内存。这两个方法共同为二叉搜索树的管理提供了遍历和清理功能，是树数据结构操作的基础部分。

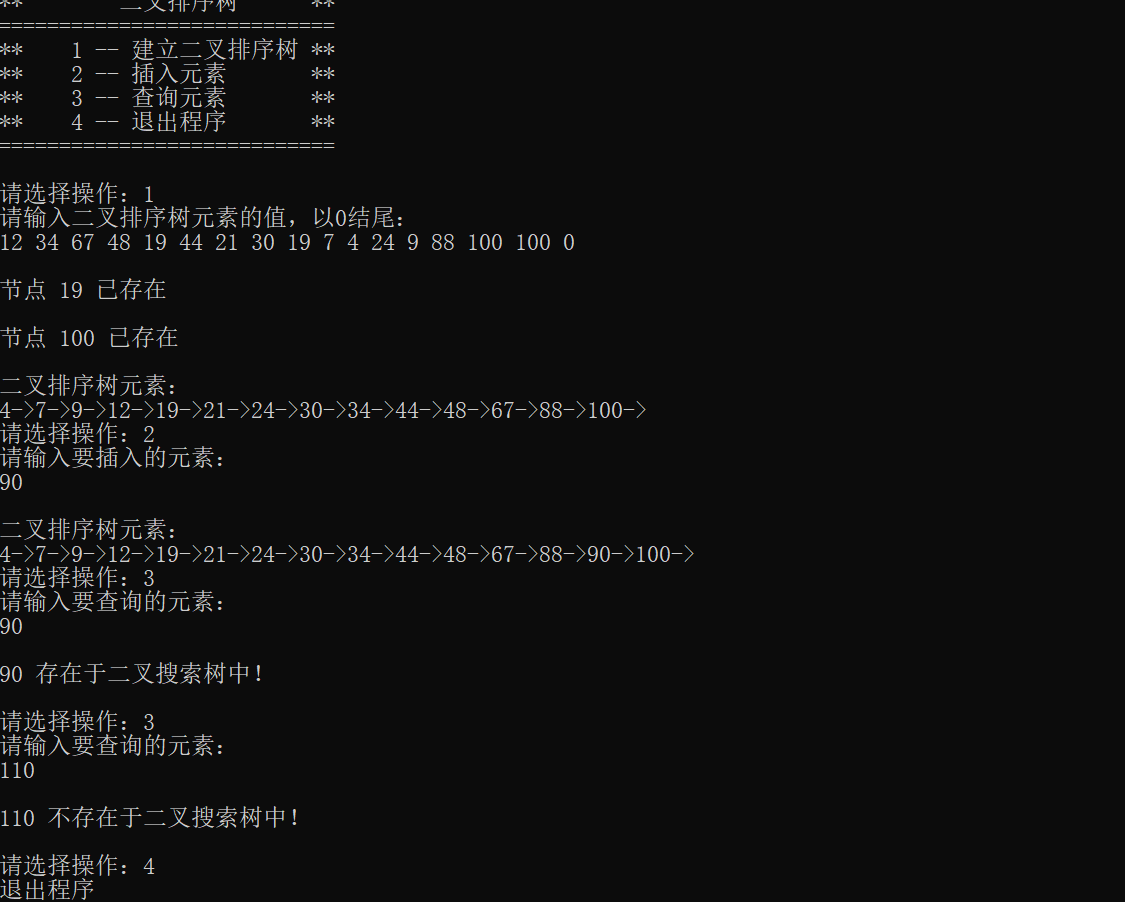
**3.4. 创建根节点和插入节点功能实现**

1. **void** Binary\_Search\_Tree::Create\_Root\_Node(**double** key) {
2. root = **new** BST\_Node(key);
3. **if** (root == NULL) {
4. cerr << "内存分配错误" << endl;
5. exit(1);
6. }
7. }
8. **void** Binary\_Search\_Tree::Insert\_Node(BST\_Node\* bst, **double** key) {
9. **if** (key < bst->value) {
10. **if** (bst->left\_child != NULL) Insert\_Node(bst->left\_child, key);
11. **else** bst->left\_child = **new** BST\_Node(key);
12. }
13. **else** **if** (key > bst->value) {
14. **if** (bst->right\_child != NULL) Insert\_Node(bst->right\_child, key);
15. **else** bst->right\_child = **new** BST\_Node(key);
16. }
17. }

这两个函数是二叉搜索树（BST）的关键部分，用于节点的创建和插入。Create\_Root\_Node函数负责创建树的根节点。它接收一个键值（key），并为根节点分配内存，将该键值赋给根节点。如果内存分配失败，则会输出错误消息并终止程序。另一方面，Insert\_Node函数用于将新节点插入到树中的适当位置。它递归地比较给定的键值与当前节点的值，根据二叉搜索树的特性（左子节点的值小于父节点，右子节点的值大于父节点），将新节点插入到左子树或右子树中的适当位置。如果相应的子节点已存在，函数将递归地继续在子树中寻找插入点；如果子节点不存在，将在那个位置创建新的节点。通过这两个函数，二叉搜索树能够维护其有序结构，同时允许动态地添加新的元素。

1. 窗体顶端

**健全性测试：**

****

**内存分配错误处理：**程序在创建根节点时检查内存分配是否成功。如果内存分配失败（例如，由于资源不足），程序会输出错误消息并安全地终止，防止了可能的未定义行为。

**防止重复插入：**在尝试插入新节点之前，程序先检查该值是否已经存在于树中。这样做避免了重复插入同一值，保持了树的正确性和一致性。

**输入验证：**用户界面中有输入验证机制，确保用户输入的值是有效的。无效输入（如错误的字符或不在预期范围内的值）会被识别并提示用户重新输入，从而防止了输入错误导致的程序异常。

**安全的内存释放：**在程序清空二叉树时，采用了递归方法来安全地释放所有节点的内存。这个过程中对空指针的检查避免了潜在的空指针解引用，确保了程序在删除节点时不会访问无效内存。

**健壮的递归操作：**树的遍历和清空操作使用了递归方法，这些方法被设计为能够优雅地处理边界情况，如空树或叶子节点，从而增加了程序的健壮性。

1. **项目创新点**

本程序的创新点主要体现在对二叉搜索树（BST）的高效管理和用户交互设计上。首先，它采用了递归方法来实现树的插入、搜索和遍历操作，这不仅提高了代码的简洁性和可读性，而且有效地利用了二叉搜索树的性质来优化性能。其次，程序在节点插入过程中加入了重复值检测，避免了树中数据的冗余，保证了数据的唯一性和树的结构完整性。此外，用户界面的设计考虑了用户体验，提供了清晰的指令和反馈，使得非专业用户也能轻松地进行树的操作。在内存管理方面，程序通过安全地释放内存和有效地处理潜在的错误情况（如内存分配失败），展现了对资源管理的重视。这些特点共同构成了本程序的创新之处，使其在基本功能的基础上增添了实用性和用户友好性。

**五、实验小结**

通过本次实验，我们成功实现了一个高效且用户友好的二叉搜索树管理程序。实验过程中，我们深入探讨了二叉搜索树的特性，并将这些特性应用于程序的设计与实现中。递归方法的应用在简化代码结构的同时，也提高了操作的效率。实验还强调了健全性和错误处理的重要性，我们通过合理的错误检测和处理确保了程序的稳定运行。用户界面的直观设计提高了程序的可访问性和易用性。总的来说，本实验不仅加深了对数据结构特别是二叉搜索树的理解，而且也增强了对软件设计原则和良好编程实践的认识，为今后开发更复杂的应用程序打下了坚实的基础。

窗体顶端