实验八 查找实验

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 姓名 | 张渊 | 学号 | 2022210000 | 专业班级 | 计科22-0 |
| 指导教师 | 张先宜 | 实验时间 | 2023.6.11 | 实验地点 |  |

**7.1 实验目的**

(1) 掌握顺序表的查找方法，尤其是二分查找方法。

(2) 掌握二叉排序树的建立及查找。

查找是软件设计中的最常用的运算，查找所涉及到的表结构的不同决定了查找的方法及其性能。二分查找是顺序表的查找中的最重要的方法，应能充分理解其实现方法和有关性能，并能借助其判定树结构来加深理解。二叉排序树结构在实验时具有一定的难度，可结合二叉树的有关内容和方法来实现。

**7.2 实验要求**

1. 查找表的定义和算法实现放入库文件，如“search.h”中；
2. 查找的测试数据用文本文件方式给出，例如测试数据名为ser.dat；
3. 可多次连续测试。

**7.3 实验任务**

编写算法实现下列问题的求解。

(1) 对下列数据表，分别采用二分查找算法实现查找，给出查找过程依次所比较的元素（的下标），并以二分查找的判定树来解释。

第一组测试数据：

数据表为 (1,2,3,4,6,7,8,9,10,11,12,13,17,18,19,20,24,25,26,30,35,40,45,50,,100)

查找的元素分别为： 2，8，20, 30，50，5，15，33，110

第二组数据：

数据表为 (2,3,5,7,8,10,12,15,18,20,22,25,30,35,40,45,50,55,60, 80,100)

查找的元素分别为： 22，8，80，3，100，1，13，120

(2) 设计出在二叉排序树中插入结点的算法，在此基础上实现构建二叉排序树的算法。

测试数据：构建二叉排序树的输入序列如下：

第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175

第二组数据：

100，70，60，80，150，120，50，160，30，40，170，180，175，35

(3) 设计算法在二叉排序树中查找指定值的结点。

测试数据：在任务(1)中第一组测试数据所构造的二叉排序树中，分别查找下列元素： 150，70，160，190，10，55，175

(4) 设计算法在二叉排序树中删除特定值的结点。

测试数据：在任务(1)中第一组测试数据所构造的二叉排序树中，分别删除下列元素：30，150，100

(5) 设计算法，对给定的二叉排序树，求出在等概论情况下的平均查找长度。

测试数据：构建二叉排序树的输入序列如下：

第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175

第二组数据：

100，70，60，80，150，120，50，160，30，40，170，180，175，35

(6) 对给定的二叉树，假设其中各结点的值均不相同，设计算法以判断该二叉树是否是二叉排序树。

测试数据：构建二叉排序树的输入序列如下：

第一组数据：

100，150，120，50，70，60，80，170，180，160，110，30，40，35，175

第二组数据：

70，100，60，80，150，120，50，160，30，40，170，180，175，35

**7.4数据结构设计**

typedef struct avlNode

{

elementType data;

int bf;

struct avlNode\* lChild, \* rChild;

}bstNode, \* AvlTree;

**7.5算法设计**

1 【算法思想】有序数组中查找关键词所在的位置。首先确定整个查找区间的中间位置 mid = low+(high-low)/2。用待查关键字key值与中间位置的关键字值进行比较；若相等，则查找成功；若key小于a[k]，则在前（左）半个区域继续进行折半查找；若key大于a[k]，则在后（右）半个区域继续进行折半查找。对确定的缩小区域再按折半公式，重复上述步骤。另外，若不存在，则提示即可。

【算法描述】

int bin\_search(elementType a[], int n, elementType key)

{

int low = 0, high = n - 1, mid;

while (low <= high)

{

mid = (low + high) / 2;

cout << mid << " ";

if (key < a[mid])

{

high = mid - 1;

}

else if (key > a[mid])

{

low = mid + 1;

}

else

{

return mid;//查找成功，return mid;

}

}

return -1;//查找失败

}

2 【算法思想】首插入结点也要保持二叉排序树有序。首先若树为空，直接将其作为根节点。若不为空，且值大于当前比较的结点，那么向当前结点右孩子方向继续比较，寻找插入位置。若值小于当前比较的结点，那么向当前左孩子方向继续比较，寻找插入位置。

【算法描述】

void insert(bstNode\*& T, bstNode\* S)

{

if (T == NULL)

{

T = S;

}

else if ((T)->data < S->data)

{

insert(T->rChild, S);

}

else

{

insert(T->lChild, S);

}

}

void creatTree(bstNode\*& T)

{

cout << "请输入一个数,输入-9999停止创建" << endl;

int a;

cin >> a;

if (a != -9999)

{

T = new bstNode;

T->data = a;

T->lChild = NULL;

T->rChild = NULL;

}

cin >> a;

while (a != -9999)

{

bstNode\* S = new bstNode;

S->data = a;

S->lChild = NULL;

S->rChild = NULL;

insert(T, S);

cin >> a;

}

cout << "中序遍历为" << endl;

inTraverse(T);

}

3 【算法思想】先判断当前节点是否为要查找的节点，相等就返回当前节点，作为递归中值的条件。如果不相等，且左子树节点值大于key，则递归左子树查找。如果左递归先序查找，找到该节点，那就返回该节点。如果不相等，且右子树节点值小于key，则递归右子树查找。如果右递归先序查找，找到该节点，那就返回该节点。若未找到，return NULL。

【算法描述】

bstNode\* search(bstNode\* t, elementType key)

{

if (t == NULL)

return NULL;

if (t->data == key)

{

cout << "查找成功" << endl;

return t;

}

else if (t->data < key)

{

return search(t->rChild, key);

}

else

{

return search(t->lChild, key);

}

}

4 【算法思想】（1）要删除的结点为叶子结点时，直接删除，调整父结点的删除了的孩子为NULL；（2）要删除的结点只有一个孩子时候，删除当前结点，并且将要删除结点的孩子赋给要删除的结点的父结点即可。（3）要删除的结点有两个孩子时，这里采取顶替法，要寻找到要删除的结点的前驱结点，将值赋给要删除的结点，然后删除这个前驱结点，并且做适当的调整。删除的结点的前驱结点一定是没有右孩子的。找到之后将值赋给p，将其父结点的右孩子设置为自己的左孩子。其中还有一个特殊情况，当sf==p时，此时p的前驱结点就是自己的左孩子，将p的左孩子设置为s的左孩子即可。

【算法描述】

void deletekey(bstNode\*& t, elementType key)

{

bstNode\* p = search(t, key);

if (p == NULL)

{

cout << "删除的值不存在。" << endl;

return;

}

if (p->lChild && p->rChild)//要删除结点的左右孩子都不为空

{

bstNode\* sf = p; bstNode\* s = p->lChild;//找p的直接前驱(左子树的最大值)

while (s->rChild) {

sf = s;

s = s->rChild;

}

p->data = s->data;

if (sf == p) //s为p左子树的根结点。

sf->lChild = s->lChild;

else

sf->rChild = s->lChild;

delete s;

}

else if (p->lChild == NULL && p->rChild == NULL)//要删除结点的左右孩子都为空

{

delete p;

}

else if (p->lChild && p->rChild == NULL)//要删除结点的左孩子不为空右孩子为空

{

p->data = p->lChild->data;

bstNode\* temp = p->lChild;

p->lChild = p->lChild->lChild;

p->rChild = p->lChild->rChild;

delete temp;

}

else//要删除结点的左孩子为空右不孩子为空

{

p->data = p->rChild->data;

bstNode\* temp = p->rChild;

p->rChild = p->rChild->rChild;

p->lChild = p->rChild->lChild;

delete temp;

}

}

5 【算法思想】实现了计算二叉排序树高度的函数getHeight，以及计算所有叶子节点到根节点的路径长度之和的函数getLeafPathSum。最后，我们实现了计算二叉排序树平均查找长度的函数getAvgSearchLen，该函数先调用getLeafPathSum 计算所有叶子节点到根节点的路径长度之和，然后通过计算节点数得到平均查找长度。

【算法描述】

int getHeight(bstNode\* root) {

if (root == NULL)

return 0;

return max(getHeight(root->lChild), getHeight(root->rChild)) + 1;

}

int getLeafPathSum(bstNode\* root, int depth) {

if (root == NULL)

return 0;

if (root->lChild == NULL && root->rChild == NULL)

return depth;

return getLeafPathSum(root->lChild, depth + 1) + getLeafPathSum(root->rChild, depth + 1);

}

double getAvgSearchLen(bstNode\* root) {

int n = 0;

int pathSum = getLeafPathSum(root, 0);

bstNode\* cur = root;

while (cur) {

n++;

cur = cur->rChild;

}

return (double)pathSum / n;

}

6 【算法思想】对给定的二叉树进行中序遍历，假如二叉树是二叉排序树那么中序遍历的结果一定是递增的，假如不是那么就不是二叉排序树

【算法描述】

void method(bstNode\* T, vector<int> &v)

{

if (T)

{

method(T->lChild, v); //中序遍历左子树

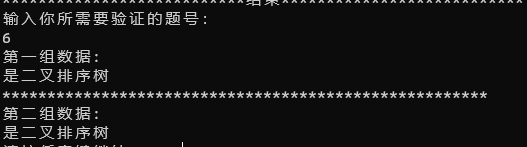
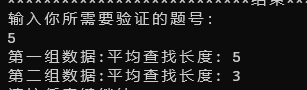
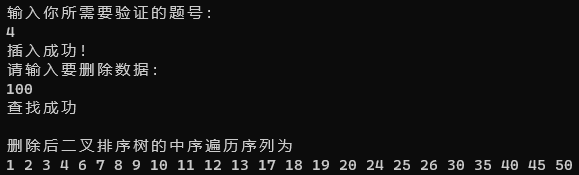
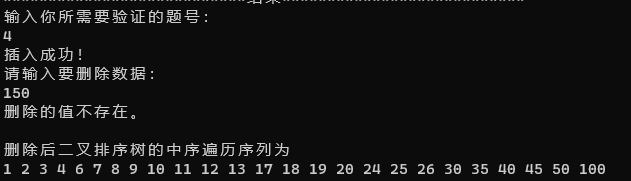
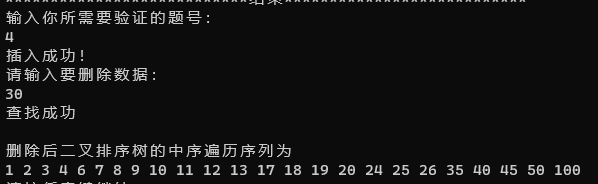
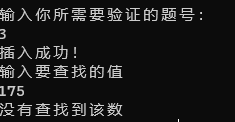
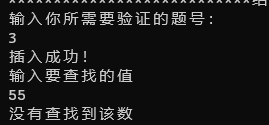
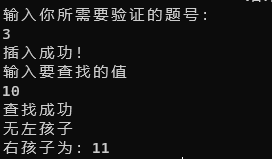
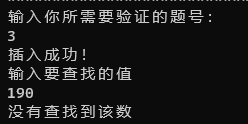
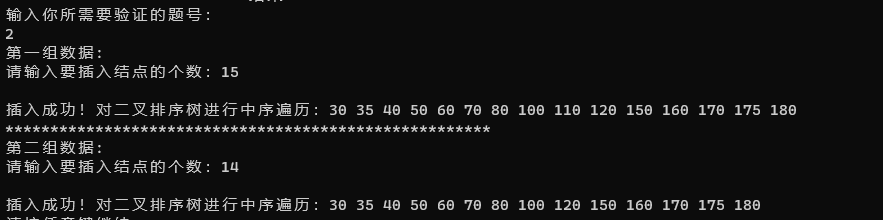
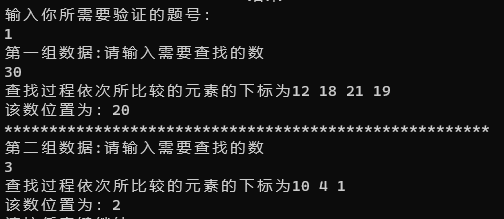
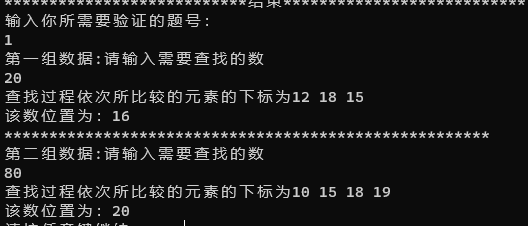
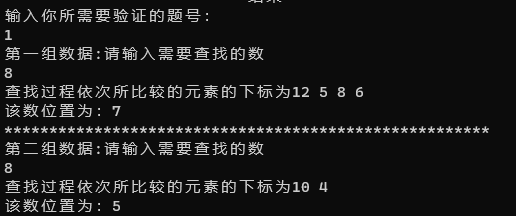
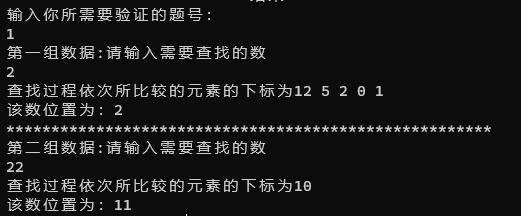
v.push\_back(T->data);

method(T->rChild, v); //中序遍历右子树

}

}

**7.6运行和测试**



**7.7总结、心得和建议**

本次查找实验，主要包括了二叉排序树的创建、插入、查找、删除、平均查找长度、判断是否为二叉排序树等。通过实验，我对数据结构中的查找算法有了更深刻的理解和认识。

二叉排序树是一种基于二叉树结构的查找算法，它的特点是每个节点的值都大于其左子树中的任何节点的值，小于其右子树中的任何节点的值。因此，创建二叉排序树的过程就是不断插入节点，并保持树的排序结构。在二叉排序树的查找和删除过程中，我发现二叉排序树的平均查找长度会因为树的结构产生变化而发生改变，导致相对较高的查询时间。为了解决这个问题，我对树进行了平衡化操作，以确保树的结构相对稳定，并减少查找时间。平均查找长度的计算。我了解到，二叉排序树的查找效率是受制于其左右子树的深度的；如果左右子树的深度差较大，则树的高度也会较大，查找效率会较低。因此，在实验中我编写了如何计算二叉排序树的平均查找长度，以便衡量该树的查找效率。最后是判断二叉树是否是二叉排序树。通过实验测试，我了解到判断一个二叉树是否是二叉排序树的方法：对于每个节点，它的值必须大于其左子树中任何节点的值，小于其右子树中任何节点的值。

通过这次实验，我对数据结构中的查找算法有了更深刻的理解和认识。在实验过程中，我遇到了一些问题，但通过自己的思考和参考他人的意见，最终找到了解决办法，希望今后能够继续探索和学习相关的知识。

**[7.8附录]**

（源代码清单。纸质报告不做要求。电子版报告，可直接附源文件，删除编译生成的所有文件）

****