

南开大学

计算机学院

数据结构实验四

**实验四**实验报告

冯思程2112213

年级：2021级

专业：计算机科学与技术

指导教师：王玮

目录

**一、实验题目**

[（一）作业要求](#bookmark51" \o "Current Document)

[（二）问题描述](#bookmark60" \o "Current Document)

**[二、实验方案](#bookmark79" \o "Current Document)**

[（一）实验工具](#bookmark84" \o "Current Document)

[（二） 实验结果分析](#bookmark106" \o "Current Document)

[三](#bookmark147" \o "Current Document)**[、算法设计及分析](#bookmark147" \o "Current Document)**

[i代码展示与分析](#bookmark147" \o "Current Document)

[四](#bookmark313" \o "Current Document)**[、复杂度分析](#bookmark313" \o "Current Document)**

1. **[时间复杂度分析](#bookmark313" \o "Current Document)**
2. **[空间复杂度分析](#bookmark313" \o "Current Document)**

[五](#bookmark338" \o "Current Document)**[、实验总结](#bookmark338" \o "Current Document)**

1. **实验题目**

（一） 作业要求

1.给定序列，使用逐点插入法构建二叉搜索树；

2.使用非递归的方法按照升序序列输出上述二叉搜索树的关键字；

3.判断上述二叉搜索树是否为AVL搜索树，若不是，则将其转化为AVL搜索树。

（备注：输出树时将结点的左右孩子结点在括号内输出，若为空则输出#）

（二） 问题描述

主要算法逻辑部分：

1. 首先定义了二叉搜索树的结点类，由序号，关键字和左右孩子指针构成。然后是二叉搜索树类，创建的方法是采用逐点插入法，二叉搜索树的定义是左子树关键字小于跟结点关键字，右子树的关键字大于跟结点关键字。所以在插入是与跟结点比较，如果插入的关键字更大往右子树遍历，否则向左子树遍历，如果相等则本次插入失败继续插入下一个关键字。
2. 然后是find函数，是一个查找序号位置，然后进行输出的函数，先将左子树结点依次加入栈中，如果查找到序号，则进行输出，输出的形式是按题目中的要求，依次访问该结点的左孩子和右孩子，如果不为空则记录关键字，如果为空则记录成#。结合栈“后进先出”的特点，借助队列实现值的push与pop，沿着根的左孩子依次入栈，直到左孩子为空；此时栈顶元素出栈并访问：若其右孩子为空，栈顶出栈并访问，若右孩子不空，将其右孩子反复执行左孩子依次入栈的操作，这实现了用非递归的方法输出关键字。
3. 然后是solution类，inorder函数是中序遍历来将关键字存进vector容器，levelorder是对树进行层次遍历。为了得到二叉树的高度，用递归算法，递归比较左右子树高度，最后二叉树的高度就是高的+1。
4. 然后是对是否是平衡二叉树的判断，平衡二叉树是指任意结点的左右子树高度差的绝对值要小于等于1。用递归分别得出各个结点左右子树的高度，计算两者的差值，如果大于1或者小于-1，则不是平衡二叉树，反之则是平衡二叉树。
5. 然后是将不是平衡二叉树的二叉搜索树转化为平衡二叉树，由于二叉搜索树是有序的，故可以将其映射到一个一维数组中，得到所有数值的有序排列。然后选取数组的中位值，将其作为结点，构建一个新的二叉树，并利用递归的思想，将数组不断地进行二分，使所有数值都均匀排布在树中，二叉树构建完毕。
6. 然后是主函数部分，先输入结点个数，然后输入结点，首先先利用for循环输出逐点插入法的结点输出。然后是进行平衡二叉树的判断，分别生序输出关键字，如果是平衡二叉树，则输出true，然后输出结点；如果不是则输出False，然后调用转换函数，然后再输出结点。

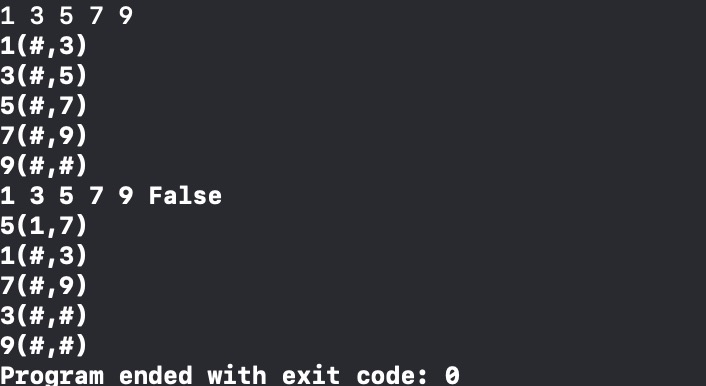
**二、实验方案**

（一）实验工具

1.由于本人的电脑是mac下的macos系统，在本次实验用采用在xcode编译器下使用c++语言完成本次实验。

（二）实验结果分析步骤

1.在编写完程序后，输入多组数据进行输出测试，发现可以成功实现所需的功能。如下图：



1. **算法设计与分析**
2. 全部代码讲解：

#include<iostream>

#include<stack>

#include<vector>

#include<queue>

using namespace std;

template<class T>

struct BinarySearchTreeNode//构造二叉搜索树节点，同时写一个带参的构造函数

{

int num=0;

T Key;

BinarySearchTreeNode<T>\* lchild;

BinarySearchTreeNode<T>\* rchild;

BinarySearchTreeNode(const T& key){

Key=key;

lchild=nullptr;

rchild=nullptr;

}

};

template<class T>

class BinarySearchTree

{

public:

typedef BinarySearchTreeNode<T> Node;//重命名为Node方便之后的使用

Node\* root=nullptr;

int count=0;

BinarySearchTree() = default;//强制生成默认的构造函数

bool insert(const T& key)//插入函数，如果插入成功返回true否则false

{

if (root == nullptr)//如果是空树，就new一个新值进去

{

root = new Node(key);

count++;

root->num = count;

return true;

}

Node\* temp = root;//用temp遍历，用parent存储上一个结点的位置

Node\* parent = temp;

while (temp)//遍历二叉搜索树，找到关键字key值应该在的位置

{

parent = temp;//更新parent的位置

if (temp->Key < key)//key值更大往右孩子继续遍历

temp=temp->rchild;

else if (temp->Key > key)//key值更小往左孩子继续遍历

temp= temp->lchild;

else//如果遇到一样的则不插入，返回false，插入失败

return false;

}//循环结束，说明插入结束

temp = new Node(key);//给temp动态创建一个新结点，链接到二叉搜索树上

count++;

temp->num = count;//结点序号加一

if (parent->Key < key)//key值更大插入到右子树，否则插左子树

parent->rchild = temp;

else

parent->lchild =temp;

return true;//插入成功，返回true

}

void find(int k)//

{

if (root == nullptr)//如果是空树，直接return结束查找

return ;

stack<Node\*> s;//建立栈

Node\* temp1 = root;//初始化temp1为跟结点，用来遍历

while (temp1 != nullptr || s.empty() == false)

{

while (temp1 !=nullptr)//将左子树节点依次加入栈中

{

s.push(temp1);

temp1 = temp1->lchild;

}

temp1 = s.top();//让temp1重新等于栈顶元素，即最后被放进去的结点

if (temp1->num == k)//对序号进行搜索

{

cout << temp1->Key;

cout << "(";

if (temp1->lchild == nullptr)//左孩子是空则输出#，右孩子是空则输出#

{

cout << "#";

}

else

{

cout << temp1->lchild->Key;

}

cout << ",";

if (temp1->rchild == nullptr)

{

cout << "#";

}

else

{

cout <<temp1->rchild->Key;

}

cout<< ")" << endl;

}

s.pop();//访问完出栈

temp1 =temp1->rchild;//当访问完一个左节点的时候，访问右节点

}

}

};

class Solution {

public:

int count1 = 0;

vector<int> nums;//nums是一个容器变量，容器名称为vector，容器内存的数据为int型，不指定长度

void inorder(BinarySearchTreeNode<int>\* root) {//用递归算法进行中序遍历

if (!root) return;//如果是空则返回

inorder(root->lchild);

nums.push\_back(root->Key);//添加跟结点的关键字进容器

inorder(root->rchild);

}

void levelorder(BinarySearchTreeNode<int>\* root)//递归算法进行层序遍历

{

queue<BinarySearchTreeNode<int>\*> Q;

Q.push(root);//跟结点入队

while (!Q.empty())

{

BinarySearchTreeNode<int>\* tmp = Q.front();//tmp保存队首元素

Q.pop();//本身元素出队

count1++;

tmp->num = count1;

if (tmp->lchild != nullptr)//先左后右，有则进队

{

Q.push(tmp->lchild);

}

if (tmp->rchild != nullptr)

{

Q.push(tmp->rchild);

}

}

}

int BinaryTreeHigh(BinarySearchTreeNode<int>\* root)//得到二叉树的高度，用递归算法，递归比较左右子树高度，最后二叉树的高度就是高的+1

{

if (root == NULL)

{

return 0;

}

int ret1 = BinaryTreeHigh(root->lchild);

int ret2 = BinaryTreeHigh(root->rchild);

return ret1 > ret2 ? ret1 + 1 : ret2 + 1;

}

int IsBlancedTree\_R(BinarySearchTreeNode<int>\* root)//也用递归算法判断树是否为平衡二叉树，如果是就返回1，否则返回0

{

//平衡二叉树是指以当前结点为根的左右子树高度差的绝对值不得超过1

if (root == NULL) //空树是平衡二叉树

return 1;

int right = BinaryTreeHigh(root->lchild);

int left = BinaryTreeHigh(root->rchild);

int gap = right - left;

if (gap > 1 || gap < -1)

return 0;

//利用递归判断左右子树

return IsBlancedTree\_R(root->lchild) && IsBlancedTree\_R(root->rchild);

}

BinarySearchTreeNode<int>\* tobalance(int l, int r) {

if (l > r) return nullptr;

int mid = (l + r) / 2;

BinarySearchTreeNode<int>\* root =new BinarySearchTreeNode<int>(nums[mid]);//为了使树保持平衡，从中间位置开始安排左右孩子

root->lchild = tobalance(l, mid - 1);

root->rchild = tobalance(mid + 1, r);//用递归算法再对两边分别平衡建立

return root;//返回跟结点

}

BinarySearchTreeNode<int>\* balanceBST(BinarySearchTreeNode<int>\* root) {

inorder(root);//先将树中序遍历存到容器内

return tobalance(0, nums.size() - 1);//然后调用tobalance函数进行平衡建立

}

};

int main()

{

BinarySearchTree<int> fsctree;//创建树

int length,b;

cin >> length;//输入结点个数

for (int i = 0; i < length; i++)//for循环逐点插入法创建二叉搜索树

{

cin >> b;

fsctree.insert(b);

}

Solution workout;//创建solution类来分解解答实验问题

for(int i=1 ;i<=length;i++)//遍历输出按逐点插入法的二叉搜索树

{

fsctree.find(i);

}

if (workout.IsBlancedTree\_R(fsctree.root) == 0)//如果不是avl树则转化成avl树，然后再

{

BinarySearchTree<int> avltree;//新建一个树来存转化完的avl树

avltree.root =workout.balanceBST(fsctree.root);

for(int i=0;i<workout.nums.size();i++){//输出关键字中序遍历对应关键字的升序排列

cout<<workout.nums[i]<<" ";

}

Solution workout1;

cout << "False" << endl;

workout1.levelorder(avltree.root);

for (int i = 1; i <=length; i++)

{

avltree.find(i);

}

}

else//如果是avl树则遍历输出结点

{

cout<<"true"<<endl;

for(int i=0;i<workout.nums.size();i++){//

cout<<workout.nums[i]<<" ";

}

Solution workout1;

workout1.levelorder(fsctree.root);

for (int i = 1; i <=length; i++)//循环输出结点

{

fsctree.find(i);

}

}

return 0;

}

1. **复杂度分析**
2. 时间复杂度分析：、

insert函数，需要逐点插入结点，与输入的节点个数有关，为，Find函数，进行了内外两层循环，复杂度为，平衡二叉树的查找算法，复杂度为，将二叉搜索树转化为平衡二叉树，类似于折半查找，复杂度为。计算树的高度是递归算法，时间复杂度也是，判断是否是平衡二叉树的算法也是利用的递归算法，时间复杂度也是，主函数中的for循环中调用insert函数，时间复杂度是，主函数中，在for循环中调用find函数，时间复杂度是O（n立方）。综上整个程序的时间复杂度是O（n立方）。

1. 空间复杂度分析：

分别观察每个函数，可以发现，利用空间最多的是函数开辟栈来存储结点，空间复杂度是O（n），其他函数开辟空间都是常数级，空间复杂度是O（1）。

综上空间复杂度是O（n）。

**五、实验总结**

对平衡二叉树和二叉搜索树有了更深的理解，而且学会了在逻辑上看似简单的算法的代码实现，同时在不断的debug过程当中，我逐渐提高了能力。