数据结构课程设计

项目说明文档

二叉排序树

|  |  |
| --- | --- |
| 作者姓名： | 高逸轩 |
| 学 号： | 2053385 |
| 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项目分析

## 1.1 项目需求分析

依次输入关键字并建立二叉排序树，实现二叉排序树的插入和查找功能。

* 功能完善

插入和查找功能均应该无误完成

* 执行效率高

针对数据量比较大的情况，本系统也应该具有在较短时间内求解出正确答案的能力。

* 代码可读性强

本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，同时进行了合理封装。

* 健壮性

当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。

## 1.2 项目要求

### 1.2.1 功能要求

建立二叉排序树，实现二叉排序树的插入和查找功能。

### 1.2.2 输入格式

不同操作编码、需要操作的二叉排序树的信息。

### 1.2.3 输出格式

执行相应的操作，并展示二叉排序树的信息。

### 1.2.4 项目示例

# 2 项目设计

## 2.1 数据结构设计

二叉排序树（Binary sort tree，BST），又称为二叉查找树，或者是一棵空树；或者是具有下列性质的二叉树：

(1)若它的左子树不为空，则左子树上所有节点的值均小于它的根节点的值；

(2)若它的右子树不为空，则右子树上所有节点的值均大于它的根节点的值；

(3)它的左、右子树也分别为二叉排序树。

其可以实现较高效率，较低时间复杂度的查找、插入、搜索等操作。若二叉排序树近似于平衡，其各操作的时间复杂度为，非常优秀；但当其完全不平衡时，深度可能为n，退化成为单链表，此时其各操作的时间复杂度为。**为解决这一问题，在最后的拓展部分讨论并实现了平衡树的操作。**

## 2.2 类设计

二叉排序树的结构为二叉树，基本类的构建和耦合关系于之前的二叉树相同。以下为部分代码：

### 2.2.1 节点

// 节点

typedef struct BinarySearchTreeNode

{

    // 构造函数

    BinarySearchTreeNode(double v = 0.0) :data(v) { leftChild = NULL, rightChild = NULL; };

    double data;         // 数据

    BinarySearchTreeNode\* leftChild;  // 左孩子

    BinarySearchTreeNode\* rightChild; // 右孩子

} BSTNode;

// 重载输出运算符

ostream& operator<<(ostream& os, const BSTNode& x)

{

    // 输出节点信息

    os << x.data << "->";

    return os;

}

重载输出运算符是为了后续可以更方便的输出节点信息。

### 2.2.2 二叉排序树

// 二叉排序树

typedef class BinarySearchTree

{

public:

    // 构造函数

    BinarySearchTree() { root = NULL; }

    // 析构函数释放空间

    ~BinarySearchTree() { MakeEmpty(root); }

    // 插入操作

    bool Insert(double key)

    // 中序遍历

    void inOrder(BSTNode\* current)

    // 寻找根节点地址

    BSTNode\* Root() { return root; }

    // 重置操作

    void MakeEmpty(BSTNode\* current)

private:

    BSTNode\* root; // 根节点

    // 初始化根节点

    void BuildRoot(double data)

    // 插入

void Insert(BSTNode\* bst, double key)

    // 前序遍历

    void preOrder(BSTNode\* current)

    // 后序遍历

    void postOrder(BSTNode\* current)

}BST;

## 2.3 项目流程图

# 3 核心代码介绍

## 3.1 插入操作

// 插入操作

    bool Insert(double key)

    {

        // 若为空树，先建立根节点

        if (root == NULL)

        {

            // 建立根节点

            BuildRoot(key);

            // 插入成功

            return true;

        }

        // 先在二叉排序树中查找要插入的值是否已经存在

        // 如果查找失败，则插入；此时p指向遍历的最后一个节点

        if (Search(root, key) == NULL)

        {

            Insert(root, key);

            return true;

        }

        else

        {

        cout << endl << "节点" << key << "已经存在于二叉排序树中" << endl;

            return false;

        }

    }

    // 插入

    void Insert(BSTNode\* bst, double key)

    {

        // 若插入值小于当前节点的值，应搜索其左子树

        if (key < bst->data)

        {

            // 若当前节点已经有左子树，则向下搜索

            if (bst->leftChild != NULL) Insert(bst->leftChild, key);

            // 若无左子树，则当前的值作为其左儿子

            else

            {

                BSTNode\* newNode = new BSTNode(key);

                // 若失败则给出提示，保证健壮性

                if (newNode == NULL)

                {

                    cerr << "内存分配错误！" << endl;

                    system("pause");

                    exit(1);

                }

                // 建立父子关系

                bst->leftChild = newNode;

            }

        }

        // 若插入值大于当前节点的值，应搜索其右子树

        if (key > bst->data)

        {

            // 若当前节点已经有左子树，则向下搜索

            if (bst->rightChild != NULL) Insert(bst->rightChild, key);

            // 若无右子树，则当前的值作为其右儿子

            else

            {

                BSTNode\* newNode = new BSTNode(key);

                // 若失败则给出提示，保证健壮性

                if (newNode == NULL)

                {

                    cerr << "内存分配错误！" << endl;

                    system("pause");

                    exit(1);

                }

                // 建立父子关系

                bst->rightChild = newNode;

            }

        }

    }

进行了Insert()插入函数的重载，实现了某个数据插入的功能。同时，处理了插入树中已经存在的数据、空间分配等异常情况，实现了健壮性。

## 3.2 查询操作

Search()函数除了实现基本的查询操作，也可以用于Insert()中，辅助判断某个数字是否已经存在于二叉排序树中，避免重复插入，实现了部分健壮性。

// 搜索二叉树中是否存在值为key的节点

    BSTNode\* Search(BSTNode\* bst, double key)  //查找成功时，p指向值为key的节点。如果查找失败，则p指向遍历的最后一个节点

    {

        // 当前节点为空，回溯

        if (bst == NULL)

            return NULL;

        // 查找成功，返回当前节点位置

        if (bst->data == key)

            return bst;

        // 若当前节点的值小于查找的值，搜索其右子树

        else if (bst->data < key)   return Search(bst->rightChild, key);

        // 若当前节点的值大于查找的值，搜索其左子树

        return Search(bst->leftChild, key);

    }

## 3.3 建立根节点

    // 初始化根节点

    void BuildRoot(double data)

    {

        // 为根节点分配空间

        root = new BSTNode(data);

        // 若失败则给出提示，保证健壮性

        if (root == NULL)

        {

            cerr << "内存分配错误！" << endl;

            system("pause");

            exit(1);

        }

    }

## 3.4 重置树

    // 重置操作

    void MakeEmpty(BSTNode\* current)

    {

        if (current == NULL) return;

        MakeEmpty(current->leftChild);

        MakeEmpty(current->rightChild);

        delete current;

    }

## 3.5 中序遍历

// 中序遍历

    void inOrder(BSTNode\* current)

    {

        if (current == NULL) return;

        inOrder(current->leftChild);

        cout << \*current;

        inOrder(current->rightChild);

    }

在输出二叉排序树信息时，需要进行中序遍历，同时进行信息的输出。

# 4 项目测试

## 4.1 健壮性测试

在本次程序中，需要对输入的数据进行许多输入错误处理，在这里部分截取代码如下：

  cout << "请输入二叉排序树元素的值，以0结尾：" << endl;

            double key;

            while (1)

            {

                while (1)

                {

                    cin >> key;

                    if (cin.fail()) // 若输入变量类型不同

                    {

                        cin.clear();

                      cin.ignore(1024, '\n');              // 清除缓存区

             cout << "输入错误，请重新输入：" << endl; // 给出错误输入的提示

                        continue;

                    }

                    break;

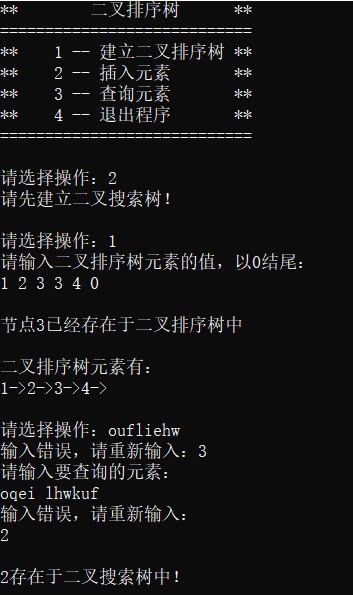
                }

                // 遇到0则结束插入

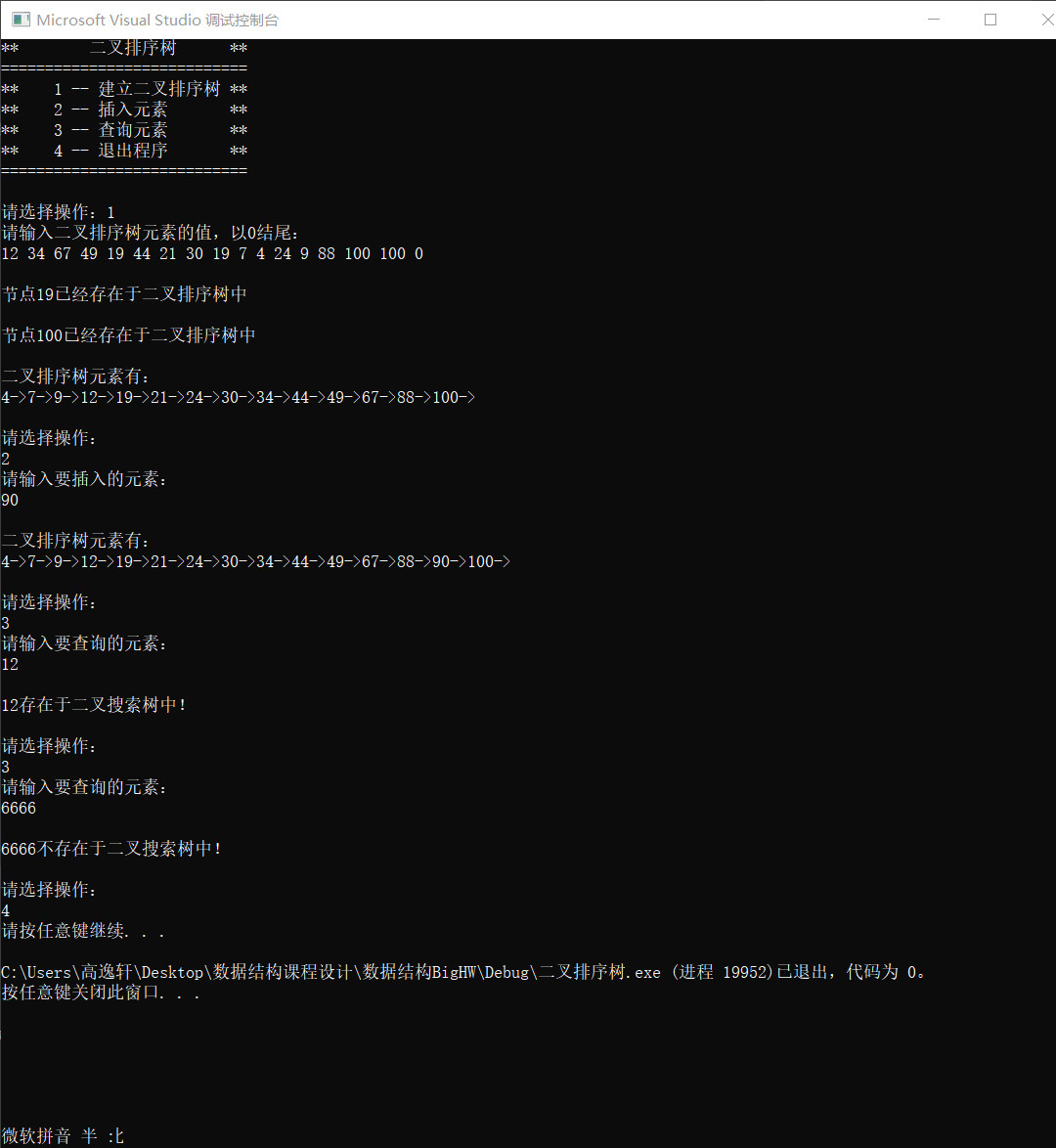
                if (key == 0.0)  break;

                else bst.Insert(key);

            }

 除此之外，还实现了对于数据重复插入、未建立树便进行操作等多种错误情况的处理。由于篇幅限制，请详见代码中的注释。以下为测试情况：

## 4.2 功能测试

为方便老师测试，提供了文件9\_test.txt，内含一组数据，测试了本程序的全部功能。以下为一组测试数据：

# 5 心得体会与扩展

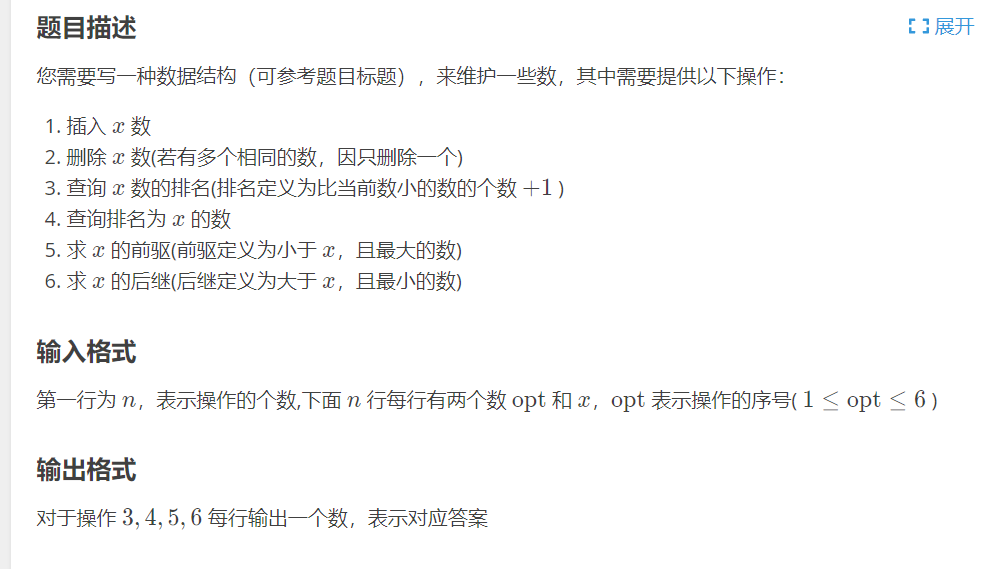
本次作业实现了二叉排序树的建立、插入、查询操作。其拥有比较优秀的时间复杂度。但在此文开始提到：在极端情况（树完全不平衡）时，二叉排序树由于退化为近似单链表的形态，会使得其效率大大下降。由此，我们引入了**平衡树**的概念，来保证各类操作的时间复杂度尽可能稳定在。

下面介绍一种经典的二叉搜索平衡树写法：Treap。其在键值满足二叉搜索树的前提下，增加了priority满足堆序的条件，即为了实现二叉搜索树的平衡化，才增加了堆序的性质。

可以证明，如果priority是随机的，那么treap的期望深度是的，也就是大部分操作可在的时间内完成。

同样，可以证明，**如果一个二叉排序树节点插入的顺序是随机的，这样我们得到的二叉排序树大多数情况下是平衡的**，即使存在一些极端情况，但是这种情况发生的概率很小，所以我们可以这样建立一颗二叉排序树，而不必要像AVL那样旋转，可以证明随机顺序建立的二叉排序树在期望高度是，但是某些时候我们并不能得知所有的待插入节点，打乱以后再插入。所以我们需要一种规则来实现这种想法，也就是说节点是顺序输入的，但我们可以通过一些规则实现“随机化”，所以提出了Treap。（即Tree+Heap（堆））

下附平衡树例题代码：（由于是高中信息学竞赛时编写，命名规范、面向对象等方面完成的不佳，可读性不高，请老师海涵）



#include<iostream>

#include<cstdio>

#include<queue>

#include<cstring>

#include<algorithm>

#include<climits>

typedef long long LL;

using namespace std;

int RD(){// 快读模板

    int out = 0,flag = 1;char c = getchar();

    while(c < '0' || c >'9'){if(c == '-')flag = -1;c = getchar();}

    while(c >= '0' && c <= '9'){out = out \* 10 + c - '0';c = getchar();}

    return flag \* out;

    }

const int maxn = 1000019,INF = 1e9;

//平衡树，利用BST性质查询和修改，利用随机和堆优先级来保持平衡，把树的深度控制在log N，保证了操作效率

//基本平衡树有以下几个比较重要的函数：新建，插入，删除，旋转

//节点的基本属性有val(值)，dat(随机出来的优先级)

//通过增加属性，结合BST的性质可以达到一些效果，如size(子树大小，查询排名)，cnt(每个节点包含的副本数)等

int na;

int ch[maxn][2];//[i][0]代表i左儿子，[i][1]代表i右儿子

int val[maxn],dat[maxn];

int size[maxn],cnt[maxn];

int tot,root;

int New(int v){//新增节点，

    val[++tot] = v;//节点赋值

    dat[tot] = rand();//随机优先级

    size[tot] = 1;//目前是新建叶子节点，所以子树大小为1

    cnt[tot] = 1;//新建节点同理副本数为1

    return tot;

    }

void pushup(int id){//和线段树的pushup更新一样

    size[id] = size[ch[id][0]] + size[ch[id][1]] + cnt[id];//本节点子树大小 = 左儿子子树大小 + 右儿子子树大小 + 本节点副本数

    }

void build(){

    root = New(-INF),ch[root][1] = New(INF);//先加入正无穷和负无穷，便于之后操作

    pushup(root);//因为INF > -INF,所以是右子树，

    }

void Rotate(int &id,int d){//id是引用传递，d(irection)为旋转方向，0为左旋，1为右旋

    int temp = ch[id][d ^ 1];//旋转理解：找个动图看一看就好(或参见其他OIer的blog)

    ch[id][d ^ 1] = ch[temp][d];//这里讲一个记忆技巧，这些数据都是被记录后马上修改

    ch[temp][d] = id;//所以像“Z”一样

    id = temp;//比如这个id，在上一行才被记录过，ch[temp][d]、ch[id][d ^ 1]也是一样的

    pushup(ch[id][d]),pushup(id);//旋转以后size会改变，看图就会发现只更新自己和转上来的点，pushup一下,注意先子节点再父节点

    }//旋转实质是({在满足BST的性质的基础上比较优先级}通过交换本节点和其某个叶子节点)把链叉开成二叉形状(从而控制深度)，可以看图理解一下

void insert(int &id,int v){//id依然是引用，在新建节点时可以体现

    if(!id){

        id = New(v);//若节点为空，则新建一个节点

        return ;

        }

    if(v == val[id])cnt[id]++;//若节点已存在，则副本数++;

    else{//要满足BST性质，小于插到左边，大于插到右边

        int d = v < val[id] ? 0 : 1;//这个d是方向的意思，按照BST的性质，小于本节点则向左，大于向右

        insert(ch[id][d],v);//递归实现

        if(dat[id] < dat[ch[id][d]])Rotate(id,d ^ 1);//(参考一下图)与左节点交换右旋，与右节点交换左旋

        }

    pushup(id);//现在更新一下本节点的信息

    }

void Remove(int &id,int v){//最难de部分了

    if(!id)return ;//到这了发现查不到这个节点，该点不存在，直接返回

    if(v == val[id]){//检索到了这个值

        if(cnt[id] > 1){cnt[id]--,pushup(id);return ;}//若副本不止一个，减去一个就好

        if(ch[id][0] || ch[id][1]){//发现只有一个值，且有儿子节点,我们只能把值旋转到底部删除

            if(!ch[id][1] || dat[ch[id][0]] > dat[ch[id][1]]){//当前点被移走之后，会有一个新的点补上来(左儿子或右儿子)，按照优先级，优先级大的补上来

                Rotate(id,1),Remove(ch[id][1],v);//我们会发现，右旋是与左儿子交换，当前点变成右节点；左旋则是与右儿子交换，当前点变为左节点

                }

            else Rotate(id,0),Remove(ch[id][0],v);

            pushup(id);

            }

        else id = 0;//发现本节点是叶子节点，直接删除

        return ;//这个return对应的是检索到值de所有情况

        }

    v < val[id] ? Remove(ch[id][0],v) : Remove(ch[id][1],v);//继续BST性质

    pushup(id);

    }

int get\_rank(int id,int v){

    if(!id)return 0;//若查询值不存在，返回；因为最后要减一排除哨兵节点，想要结果为-1这里就返回0

    if(v == val[id])return size[ch[id][0]] + 1;//查询到该值，由BST性质可知：该点左边值都比该点的值(查询值)小，故rank为左儿子大小 + 1

    else if(v < val[id])return get\_rank(ch[id][0],v);//发现需查询的点在该点左边，往左边递归查询

    else return size[ch[id][0]] + cnt[id] + get\_rank(ch[id][1],v);//若查询值大于该点值。说明询问点在当前点的右侧，且此点的值都小于查询值，所以要加上cnt[id]

    }

int get\_val(int id,int rank){

    if(!id)return INF;//一直向右找找不到，说明是正无穷

    if(rank <= size[ch[id][0]])return get\_val(ch[id][0],rank);//左边排名已经大于rank了，说明rank对应的值在左儿子那里

        else if(rank <= size[ch[id][0]] + cnt[id])return val[id];//上一步排除了在左区间的情况，若是rank在左与中(目前节点)中，则直接返回目前节点(中区间)的值

    else return get\_val(ch[id][1],rank - size[ch[id][0]] - cnt[id]);//剩下只能在右区间找了，rank减去左区间大小和中区间，继续递归

    }

int get\_pre(int v){

    int id = root,pre;//递归不好返回，以循环求解

    while(id){//查到节点不存在为止

        if(val[id] < v)pre = val[id],id = ch[id][1];//满足当前节点比目标小，往当前节点的右侧寻找最优值

        else id = ch[id][0];//无论是比目标节点大还是等于目标节点，都不满足前驱条件，应往更小处靠近

        }

    return pre;

    }

int get\_next(int v){

    int id = root,next;

    while(id){

        if(val[id] > v)next = val[id],id = ch[id][0];//同理，满足条件向左寻找更小解(也就是最优解)

        else id = ch[id][1];//与上方同理

        }

    return next;

    }

int main(){

    build();//不要忘记初始化[运行build()会连同root一并初始化，所以很重要]

    na = RD();

    for(int i = 1;i <= na;i++){

        int cmd = RD(),x = RD();

        if(cmd == 1)insert(root,x);//函数都写好了，注意：需要递归的函数都从根开始，不需要递归的函数直接查询

        else if(cmd == 2)Remove(root,x);

        else if(cmd == 3)printf("%d\n",get\_rank(root,x) - 1);//注意：因为初始化时插入了INF和-INF,所以查询排名时要减1(-INF不是第一小，是“第零小”)

        else if(cmd == 4)printf("%d\n",get\_val(root,x + 1));//同理，用排名查询值得时候要查x + 1名，因为第一名(其实不是)是-INF

        else if(cmd == 5)printf("%d\n",get\_pre(x));

        else if(cmd == 6)printf("%d\n",get\_next(x));

        }

    return 0;

    }

由于篇幅原因，报告内还有很多内容与解释没有展示，请老师和助教老师再移步源程序，在其中的注释写了每一步过程的详解。