数据结构课程设计



班级： 1618403

学号： 161840227

姓名： 韦 鑫

指导教师：孙 涵

目录

1.采用的数据结构 ………………………………………… 3

2.算法设计思想 …………………………………………… 3

3.关键代码 ………………………………………………… 4

4.测试数据和结果 ………………………………………… 11

5.算法的时间复杂度及其改进方法 ……………………… 13

6.结束语 …………………………………………………… 14

一、采用的数据结构

//平衡二叉树节点结构

typedef struct node

{

int data; //数值

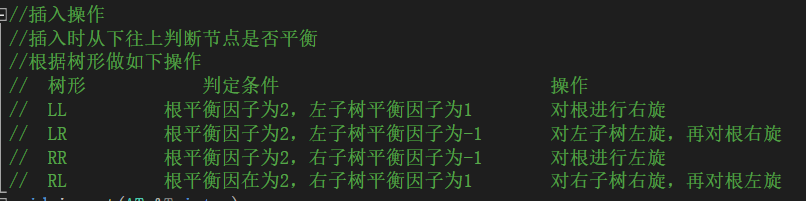
node \*left, \*right; //左右子树

int height; //树高

}Anode,\*AT;

二、算法设计思想

①AVL的插入与建立：



②删除操作(递归)：

1. 若删除的点为叶子节点，则直接删除
2. 若存在删除的点存在左子树，则用前驱(即左子树中最大值点)覆盖要删除的节点，递归删除前驱
3. 否则用后继对要删除的点进行覆盖，再递归删除后继
4. 执行以上操作后，需更新树高，若因删除右节点造成失衡，即左子树高度大于右子树，考虑左子树的平衡因子BF：若BF = 1或 BF = 0，即树形为LL，则右旋，若BF = -1则先对左子树左旋再右旋。若因删除左节点造成失衡，与上述类似。

三、关键代码

const int maxn = 5000;

//平衡二叉树节点结构

typedef struct node

{

int data; //数值

node \*left, \*right; //左右子树

int height; //树高

}Anode,\*AT;

//计算树高

int height(AT T)

{

if (T == NULL)return 0;

else return T->height;

}

//更新树高

void updateH(AT &T)

{

int h1 = height(T->left);

int h2 = height(T->right);

if (h1 > h2) T->height = h1 + 1;

else T->height = h2 + 1;

}

//计算树T的平衡因子(左子树高度 - 右子树高度)

int getBF(AT T)

{

int BF = height(T->left) - height(T->right);

return BF;

}

//查询操作

void search(AT T,int x)

{

if (T == NULL)

{

printf("平衡二叉树中不存在 %d \n",x);

return;

}

else if (T->data == x)

{

printf("平衡二叉树中存在 %d \n", T->data);

}

else if (T->data > x) //向左查询

{

search(T->left, x);

}

else search(T->right, x); //向右查询

}

//左旋

void L(AT &T)

{

Anode \*temp = T->right;

T->right = temp->left;

temp->left = T;

//更新高度

updateH(T);

updateH(temp);

//更新根节点

T = temp;

//更新根节点

T = temp;

}

//右旋

void R(AT &T)

{

Anode \*temp = T->left;

T->left = temp->right;

temp->right = T;

//更新高度

updateH(T);

updateH(temp);

}

//插入操作

//插入时从下往上判断节点是否平衡

//根据树形做如下操作

// 树形 判定条件 操作

// LL 根平衡因子为2，左子树平衡因子为1 对根进行右旋

// LR 根平衡因子为2，左子树平衡因子为-1 对左子树左旋，再对根右旋

// RR 根平衡因子为2，右子树平衡因子为-1 对根进行左旋

// RL 根平衡因在为2，右子树平衡因子为1 对右子树右旋，再对根左旋

void insert(AT &T,int x)

{

if (T == NULL)

{

T = (AT)malloc(sizeof(Anode));

T->data = x;

T->left = T->right = NULL;

T->height = 1;

return;

}

else if (T->data > x)//向左边插入

{

insert(T->left, x);

//更新树高

updateH(T);

//若失衡，即平衡因子 >= 2 (实际上只会又 BF == 2 的情况)

if (getBF(T) >= 2)

{

//根据树形选择操作

if (getBF(T->left) == 1) //LL，右旋

{

R(T);

}

else if (getBF(T->left) == -1) //LR，先左旋再右旋

{

L(T->left);

R(T);

}

}

}

else if (T->data < x)//向右边插入

{

insert(T->right, x);

//更新树高

updateH(T);

//若失衡，即平衡因子 >= 2 (实际上只会又 BF == -2 的情况)

if (getBF(T) <= -2)

{

//根据树形选择操作

if (getBF(T->right) == -1) //RR，左旋

{

L(T);

}

else if (getBF(T->right) == 1) //RL，先右旋再左旋

{

R(T->right);

L(T);

}

}

}

}

//平衡二叉树的创建

void create(AT &T, int data[], int n)

{

T = NULL;

for (int i = 0; i < n; i++)

{

insert(T, data[i]);

}

}

//寻找以T为根的最大值节点(前驱)

Anode \*findMax(AT T)

{

while (T->right) T = T->right;

return T;

}

//寻找以T为根的最小值节点(后继)

Anode \*findMin(AT T)

{

while (T->left) T = T->left;

return T;

}

//删除操作

void del(AT &T, int x)

{

if (!T)return;

//找到了要删除的节点T

if (T->data == x)

{

//若为叶子节点，则直接删除，检查是否平衡

if (!T->left && !T->right)

{

T = NULL;

free(T);

return;

}

//若存在左子树，则用前驱覆盖要删除的节点，再删除前驱

if (T->left && !T->right)

{

Anode \*pre = findMax(T->left);

T->data = pre->data;

del(T->left, pre->data);

}

//否则用后继覆盖，再删除后继

else

{

Anode \*next = findMin(T->right);

T->data = next->data;

del(T->right, next->data);

}

}

else if (T->data > x) del(T->left, x);

else del(T->right, x);

//更新树高

updateH(T);

//printf("%d-H:%d , %d\n", T->data, T->height, getBF(T));

//若失衡，即平衡因子 >= 2 (实际上只会又 BF == 2 的情况)(删除右节点)

if (getBF(T) >= 2)

{

//根据树形选择操作

if (getBF(T->left) == 1 || getBF(T->left) == 0) //LL，右旋

{

R(T);

}

else if (getBF(T->left) == -1) //LR，先左旋再右旋

{

L(T->left);

R(T);

}

}

//或者平衡因子 <= -2 (实际上只会又 BF == -2 的情况)(删除左节点)

else if (getBF(T) <= -2)

{

//根据树形选择操作

if (getBF(T->right) == -1 || getBF(T->right) == 0) //RR，左旋

{

L(T);

}

else if (getBF(T->right) == 1) //RL，先右旋再左旋

{

R(T->right);

L(T);

}

}

}

//以凹入表输出平衡二叉树

void print(AT T, int i)

{

if (T)

{

print(T->right, i + 1);

for (int j = 0; j < i \* 4; j++) printf(" ");

printf("%d\n", T->data,T->height);

print(T->left, i + 1);

}

}

//中序输出

void in(AT T)

{

if (T)

{

in(T->left);

printf("%d ", T->data);

in(T->right);

}

}

int main()

{

int n,data[maxn];

printf("创建平衡二叉树\n");

printf("请输入元素个数：");

scanf("%d", &n);

printf("请输入元素值：");

for (int i = 0; i < n; i++)

{

scanf("%d", &data[i]);

}

AT T;

create(T, data, n);

printf("凹入表\n");

print(T, 0);

printf("\n");

while (1)

{

system("pause");

system("cls");

printf

(

"1.显示\n"

"2.查找\n"

"3.插入\n"

"4.删除\n"

"5.退出\n"

"请输入选项："

);

int c,x,flag = 0;

scanf("%d", &c);

switch (c)

{

case 1:

printf("凹入表\n");

print(T, 0);

printf("\n");

break;

case 2:

printf("请输入要查找的值：");

scanf("%d", &x);

search(T, x);

break;

case 3:

printf("请输入要插入的值：");

scanf("%d", &x);

insert(T, x);

printf("凹入表\n");

print(T, 0);

printf("\n");

break;

case 4:

printf("请输入要删除的值：");

scanf("%d", &x);

del(T,x);

printf("凹入表\n");

print(T, 0);

printf("\n");

break;

case 5:

flag = 1;

break;

}

if (flag == 1)break;

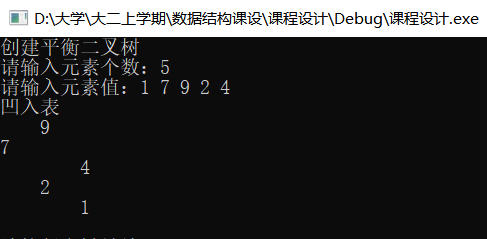
}

return 0;

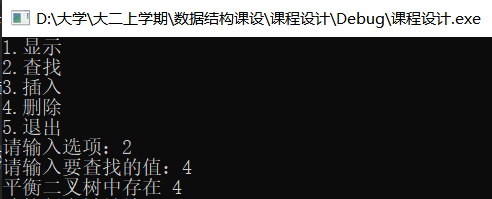
}

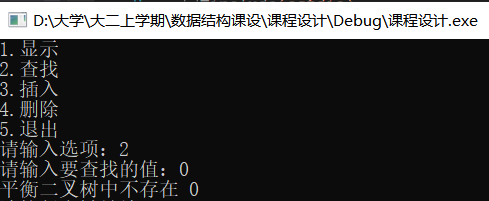
四、测试数据和结果

建AVL树并显示：



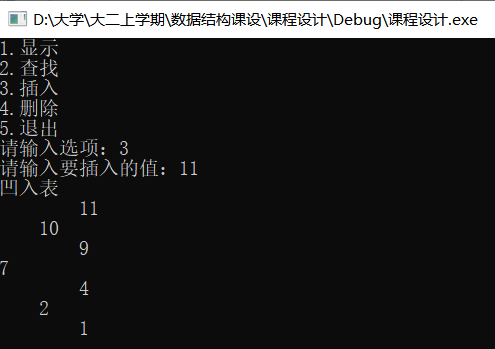
查找操作：





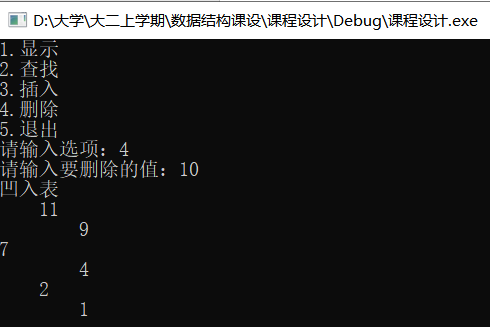
插入操作：

插入10、11



删除操作：

删除10



五、算法的时间复杂度即改进方法

插入：O(logn)

查找：O(logn)

删除：O(logn)

六、结束语

代码共334行

课本上仅仅介绍了AVL的查找与插入算法，而我认为本题的难点在于删除操作。删除操作我采用与BST（二叉查找树）类似的覆盖删除的方式，即找前驱或后继对要删除的点进行覆盖。需要注意的是删除后需要从树底层向上回溯更新树高、计算平衡因子判断离删除点最近的失衡点，对其进行左、右旋操作，直至树根。

本题让我收获最大的应该是对树的递归操作，通过找到递归边界和递归通项，再对递归算法进行书写往往会事半功倍。