## 什么是AVL

平衡二叉树定义(AVL)：它或者是一颗空树，或者具有以下性质的二叉树：它的左子树和右子树的深度之差(平衡因子)的绝对值不超过1，且它的左子树和右子树都是一颗平衡二叉树。

平衡因子(bf)：结点的左子树的深度减去右子树的深度，那么显然-1<=bf<=1,这里我们定义:

#define EH 0,#define LH 1,#define RH -1.依次为等高，左高，右高。

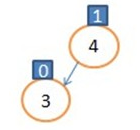
## AVL增添元素

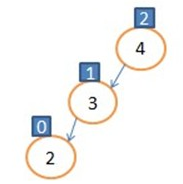
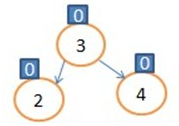
这里搜索和BST搜索一样，我就不浪费时间介绍了，我们先实现增加元素的，实现然后删除元素的。可是每次的插入和删除都要确保二叉树的平衡，那么怎么保持平衡呢？我们就引入平衡因子。注意这里再看下平衡因子的定义

我先演示下给一组数据，怎么组成一棵AVl Tree。。

int a[]={4,3,2,7,9,11,10};

1， 插入4，如图： ，平衡因子为0.

2， 插入3，如图： ，4的平衡因子因为4的左子树增长了，1-0=1，

3， 插入2，如图，显然4的平衡因子大于1了，为了保持平衡那我们就这样做：让4节点的左孩子指向3的右子树（此时为NULL），让3的右孩子指向4，让树根指向3，如图 ，这种操作我们规定为右旋操作，此图是以4为根进行旋转。 代码如下

void R\_Rotate(BiTree&T)

{

BiTree p; //

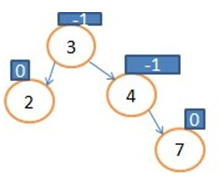
p=T->lchild; //假如此时T指向4，则p指向3；

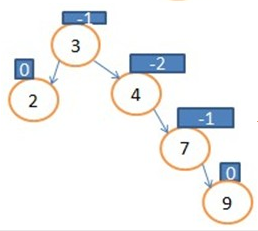
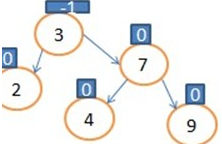
T->lchild=p->rchild; //把3的右子树挂接到4的左子树上（此例子3右子树为空）

p->rchild=T; //让3的右孩子指向4.

T=p; //根指向节点3

}

4，插入7，如图：

5，插入9，如图：显然节点4不平衡了。那我们就把4的右孩子7的左子树（此时为NULL），让7的左孩子指向4，让3的右孩子指向7，如图： ，我们规定此操作为左旋操作，此图是以4为根进行旋转，代码如下：

void L\_Rotate(BiTree&T)

{

BiTree p;

p=T->rchild; //假如此时T指向4，则p指向7.

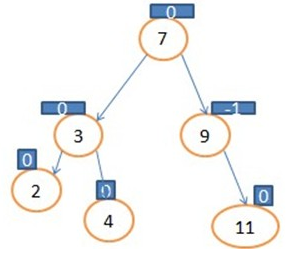
T->rchild=p->lchild; //让7的左子树挂接到4的右子树上

p->lchild=T; //让7的左孩子指向4

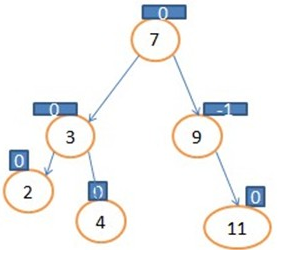
T=p; //树根指向7

}

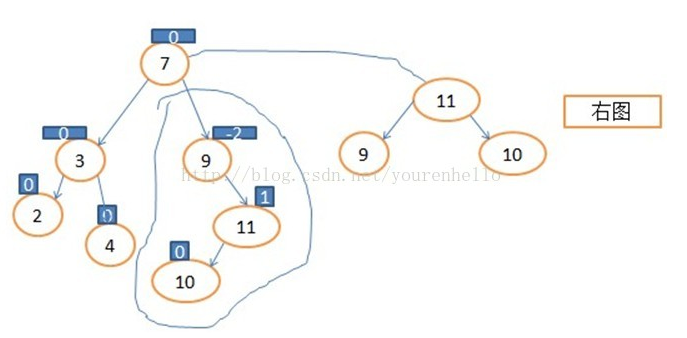
6.我们插入11，如图：



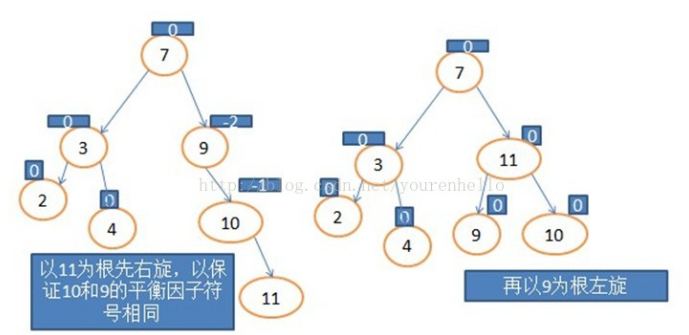
显然3节点，不平衡了，大家都应该知道以3为根进行左旋。让3的右孩子指向7的左子树（此时为4）。7的左孩子指向3，根指向7，如下图所示：



1. 我们插入10，如图：



显然节点9不平衡，且是右边高，那我们左旋吧，左旋后的效果是上图右图所示。显然这是不对的，10比11小，但在11的右孩子上。（根本原因是9和11的平衡因子符号不同）那我们在怎么办呢,看下图吧：



成功离我们不远了，我们很容易的把这组数据拼出了AVl 树，是不是很有成就感呀。好啦，我们总结下插入元素的有哪些规律吧

1，如上所述的第3步，当插入元素后导致左边高，右边低，并且为4和3的平衡因子符号相同，则右旋。

2, 如上所诉的第5步，当插入节点9后，导致以4为根的树右边高，左边低，4和7的平衡因子符号相同，则左旋

3，如上所述的第7步，当插入节点10后，导致以9为根的树右边高，左边低，由于9和11的平衡因子符号不同（也就是根和他的右孩子的平衡因子符号不同）不能进行左旋，正确操作：需要先右旋在左旋，要让根和根的右孩子平衡因子符号相同。

4，第4种旋转和3相反，当左边高于右边的话，且根和他的左孩子，平衡因子符号不同，需要先左旋再右旋

恩，就是这么简单。在实现插入函数之前，我们先封装2个函数。

RightBalance():当右高时需要右平衡时调用；

LeftBalance()功能:当左高时需要左平衡时调用；

void LeftBalance(BiTree \*T)

{

BiTree L,Lr;

L = (\*T)->lchild;

switch(L->bf)

{

case LH:

(\*T)->bf=L->bf = EH;

R\_Rotate(T);

break;

case RH:

Lr = L->rchild;

switch(Lr->bf)

{

case LH:

(\*T)->bf = RH;

L->bf = EH;

break;

case EH:

(\*T)->bf= L->bf = EH;

break;

case RH:

(\*T)->bf = EH;

L->bf = LH;

break;

}

Lr->bf = EH;

L\_Rotate(&(\*T)->lchild);

R\_Rotate(T);

case EH:

L->bf = RH;

(\*T)->bf = LH;

R\_Rotate(T);

break;

}

}

void RightBalance(BiTree \*T)

{

BiTree R,Rl;

R = (\*T)->rchild;

switch(R->bf)

{

case RH:

(\*T)->bf = R->bf = EH;

L\_Rotate(T);

break;

case EH:

R->bf = LH;

(\*T)->bf = RH;

L\_Rotate(T);

break;

case LH:

Rl = R->lchild;

switch(Rl->bf)

{

case EH:

(\*T)->bf = R->bf = EH;

break;

case RH:

R->bf = EH;

(\*T)->bf = LH;

break;

case LH:

R->bf = RH;

(\*T)->bf = EH;

break;

}

Rl->bf = EH;

R\_Rotate(&(\*T)->rchild);

L\_Rotate(T);

break;

}

}

## 代码实现

/\*\*

\* AVL.c 平衡二叉排序树

\*

\* @author Darbuly 970073804@qq.com

\* @copyright 2018-2019 Darbuly

\*/

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define LH 1

#define EH 0

#define RH -1

#define TRUE 1

#define FALSE 0

typedef struct BiTNode

{

int data;

int bf;

struct BiTNode \*lchild,\*rchild;

}BiTNode,\*BiTree;

void CreateBiTree(BiTree \*T)

{

(\*T) = (BiTNode \*)malloc(sizeof(BiTNode));

(\*T)->lchild = (\*T)->rchild = NULL;

(\*T)->bf = 0;

}

void R\_Rotate(BiTree \*T)

{

BiTree p;

p = (\*T)->lchild;

(\*T)->lchild = p->rchild;

p->rchild = (\*T);

(\*T) = p;

}

void L\_Rotate(BiTree \*T)

{

BiTree p;

p = (\*T)->rchild;

(\*T)->rchild = p->lchild;

p->lchild = (\*T);

(\*T) = p;

}

void LeftBalance(BiTree \*T)

{

BiTree L,Lr;

L = (\*T)->lchild;

switch(L->bf)

{

case LH:

(\*T)->bf=L->bf = EH;

R\_Rotate(T);

break;

case RH:

Lr = L->rchild;

switch(Lr->bf)

{

case LH:

(\*T)->bf = RH;

L->bf = EH;

break;

case EH:

(\*T)->bf= L->bf = EH;

break;

case RH:

(\*T)->bf = EH;

L->bf = LH;

break;

}

Lr->bf = EH;

L\_Rotate(&(\*T)->lchild);

R\_Rotate(T);

case EH:

L->bf = RH;

(\*T)->bf = LH;

R\_Rotate(T);

break;

}

}

void RightBalance(BiTree \*T)

{

BiTree R,Rl;

R = (\*T)->rchild;

switch(R->bf)

{

case RH:

(\*T)->bf = R->bf = EH;

L\_Rotate(T);

break;

case EH:

R->bf = LH;

(\*T)->bf = RH;

L\_Rotate(T);

break;

case LH:

Rl = R->lchild;

switch(Rl->bf)

{

case EH:

(\*T)->bf = R->bf = EH;

break;

case RH:

R->bf = EH;

(\*T)->bf = LH;

break;

case LH:

R->bf = RH;

(\*T)->bf = EH;

break;

}

Rl->bf = EH;

R\_Rotate(&(\*T)->rchild);

L\_Rotate(T);

break;

}

}

int InsertAVL(BiTree \*T,int e,int \*taller)

{

if(!\*T)

{

\*T = (BiTree)malloc(sizeof(BiTNode));

(\*T)->data = e;

(\*T)->lchild=(\*T)->rchild=NULL;

(\*T)->bf=EH;

\*taller = TRUE;

}

else

{

if(e==(\*T)->data)

{

\*taller = FALSE;

return FALSE;

}

if(e<(\*T)->data)

{

if(!InsertAVL(&(\*T)->lchild,e,taller))

{

return FALSE;

}

if(\*taller)

{

switch((\*T)->bf)

{

case LH:

LeftBalance(T);

\*taller = FALSE;

break;

case EH:

(\*T)->bf = LH;

\*taller = TRUE;

break;

case RH:

(\*T)->bf = EH;

\*taller = FALSE;

break;

}

}

}

else

{

if(!InsertAVL(&(\*T)->rchild,e,taller))

{

return FALSE;

}

if(\*taller)

{

switch((\*T)->bf)

{

case LH:

(\*T)->bf = EH;

\*taller = FALSE;

break;

case EH:

(\*T)->bf = RH;

\*taller = TRUE;

break;

case RH:

RightBalance(T);

\*taller = FALSE;

break;

}

}

}

}

}

void visit(int c,int level)

{

printf("\n %d located in %d layer\n",c,level);

}

void MidOrderTraverse(BiTree T,int level)

{

if(T)

{

MidOrderTraverse(T->lchild,level+1);

visit(T->data,level+1);

MidOrderTraverse(T->rchild,level+1);

}

}

int main()

{

BiTree T;

int e,taller,level=0;

printf("AVL.c Test\n");

//CreateBiTree(&T);

T = NULL;

printf("Please enter number with creating the AVL,ending by 65535:\n");

scanf("%d",&e);

while(65535!=e)

{

InsertAVL(&T,e,&taller);

scanf("%d",&e);

}

printf("\nOK!THe MidOrderTraverse can be followed :\n");

MidOrderTraverse(T,level);

return 0;

}

