南開大學

《数据结构》

实验报告

实验（ 五 ）

年　　级: 2023 级

专　　业: 计算机科学与技术

姓　　名: 袁田

学　　号: 2314022

一．实验内容

实现以下内容：

1. 利用逐点插入法实现一个基于链表的二叉搜索树，二叉搜索树中节点元素不重复。
2. 将二叉搜索树调整为 AVL 搜索树。
3. 输出 1 中二叉搜索树和 2 中 AVL 搜索树平均查找长度之差。

二．设计思想

T1.二叉搜索树的构建方法是根节点的左子树上的值均小于根节点的值，根节点右子树上的值均小于根节点上的值，对于子树来说也是这样。因此利用递归的思想，设计 Insert 函数，如插入的值比根节点小，则 Insert(root->left)，反之则 insert(root->right)，
递归终止的条件是找到合适的插入位置

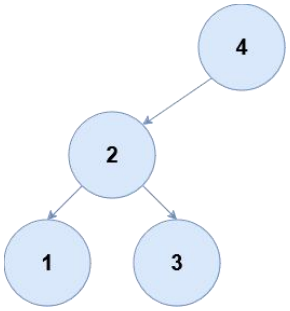
T2.相对于二叉搜索树而言，平衡二叉搜索树 AVL 主要是要在每次插入中进行一次判断，判断的依据是：插入节点所在链上的结点的 bf 值(左子树高度-右子树高度) 是否有为 2 或-2 的情况，根据祖先的 bf 值大小进行不同方向的旋转。

T3.在前序遍历时，每一次递归时都会向下一层，而平均查找长度 ASL 的值分为两部分：成功查找的长度均值和失败查找的长度均值，分别为每个节点的层数之和的平均值和每个存在空指针的结点的层数-1 之和的平均值。依次求得，进行做差。

三．程序效果

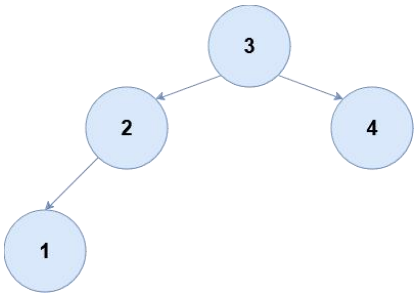
程序的运行效果，输入及输出的相关要求和具体执行结果如下所示：

Part1.逐点插入法实现一个基于链表的二叉搜索树，二叉搜索树中节点元素不重复。
输入：：4 2 1 3
输出：



一般通过其前序遍历和中序遍历值来确定其是否正确
Part2. 将二叉搜索树调整为 AVL 搜索树。

输入：一般与上一问中输入情况一致
输出：也是通过前序遍历和中序遍历来判断正确性的



由于转变得到的 AVL 树只要合法即可，有多种转化情况
(本人采用的是每次插入判断祖先 bf 值，进行旋转操作。)

Part3. 输出 1 中二叉搜索树和 2 中 AVL 搜索树平均查找长度之差。
输入：一般采用上两问中得到的树
输出：
平均查找长度之差为：((1+2+3*2)/4+(1+3*4)/5)-((1+2*2+3)/4+(2*3+3*2)/5)=0.45

四．核心代码

T1．part1.构建 node 结构体和 bst(二叉搜索树)类

```
template<class T>
struct bst_node{
    T data;
    bst_node<T>* left;
    bst_node<T>* right;
    bst_node<T>* father;
    bst_node()
    {
        left=NULL;
        right=NULL;
        father=NULL;
    }
    bst_node(T val)
    {
        data=val;
        left=NULL;
        right=NULL;
        father=NULL;
    }
};
```

```
template<class T>
class bst{
    public:
        int suc;
        int fai;
        bst_node<T>* root;
        int size;
        bst()
        {
            root=new bst_node<T>();
        }
        bst(T val)
        {
            suc = 0;
            fai = 0;
            root=new bst_node<T>(val);
        }
        ~bst()
        {
            delete root;
        }
```

Part2.根据二叉搜索树的性质写出 Insert 函数

Insert 函数这里主要用了递归的方法，简单来说就是确定了在哪棵子树就以该子树的根(如 root->left 为左子树的根)进行递归操作

```
void bst_Insert(T val,bst_node<T>* tmp)
{
    if(val<tmp->data)
    {
        if(tmp->left==NULL)
        {
            tmp->left=new bst_node<T>(val);
            tmp->left->father = tmp;
            return;
        }
        tmp=tmp->left;
        bst_Insert(val,tmp);
    }
    else if(val>tmp->data)
    {
        if(tmp->right==NULL)
        {
            tmp->right=new bst_node<T>(val);
            tmp->right->father = tmp;
            return;
        }
        tmp=tmp->right;
        bst_Insert(val,tmp);
    }
    else
    return;
}
```

用 FrontOrder 和 MiddleOrder 函数来检验树的正确性

Part3.构建前序遍历函数和中序遍历函数

前序遍历函数中，在递归的过程中还要获得成功查找的长度和以及失败查找的长度和

```
void bst_PreOrder(bst_node<T>* now,int layer)
{
    if(now==NULL)
    {
        fai += layer - 1;
        return;
    }
//  cout<<"test:"<<now->data<<endl;
    cout<<now->data<<" ";
    suc += layer;
    bst_PreOrder(now->left,layer+1);
    bst_PreOrder(now->right,layer+1);
}
void bst_MiddleOrder(bst_node<T>* now)
{
    if(now==NULL) return;
    bst_MiddleOrder(now->left);
    cout<<now->data<<" ";
    bst_MiddleOrder(now->right);
}
```

T2.part1.插入操作类似于上述二叉搜索树 bst 类中的 Insert 函数 ，但是插入之后要进行判断，进行树的更新(其实就是旋转)

```
void Update(node* cur)
{
    //插入新的结点cur后，父节点的更新状况
    node* tmp=cur;
    while(tmp->father!=NULL)
    {
        if(tmp->father->left==tmp)
        {
            tmp->father->bf+=1;
        }
        else
        {
            tmp->father->bf-=1;
        }
```

进行循环操作，对插入的结点 cur 的父节点们的 bf 值进行更新

```
        if( abs(tmp->father->bf)==2 )
        {
            if(tmp->bf==1&&tmp->father->bf==2) //LL
            {
                Rotate_LL(tmp->father);
            }
            else if(tmp->bf==1&&tmp->father->bf==-2)//Rl
            {
                Rotate_RL(tmp->father);
            }
            else if(tmp->bf==-1&&tmp->father->bf==2)//LR
            {
                Rotate_LR(tmp->father);
            }
            else if(tmp->bf==-1&&tmp->father->bf==-2)//RR
            {
                Rotate_RR(tmp->father);
            }
            break;
        }
```

如果得到某节点的父节点 bf 值为 2 或-2，进行不同的旋转操作

```
else if( abs(tmp->father->bf)==1 )
{
    tmp=tmp->father;
}
else
{
    break;
}
```

若为 1 或-1，则继续循环，让 tmp 变为其父节点；若为 0 则跳出循环进行下次插入

Part2.依次确定不同情况下的旋转方式

```
void Rotate_LL(node* tmp)
{
    //tmp为abs(bf)==2的结点；
    //进行的操作是：由于为LL型，则需要两个节点：分别为bf=2的tmp和bf=1的sonL
    //首先是tmp->father的孩子变为sonL，然后是sonL的原本的右孩子变成tmp的左孩子，现在的右孩子更新为tmp
    node* fa=tmp->father;
    node* sonL=tmp->left;
    node* sonLR=sonL->right;
    if(fa==NULL)
    {
        tmp->left= sonLR;
        sonL->right=tmp;
        sonL->father=NULL;
        if(sonLR!=NULL)sonLR->father=tmp;
        tmp->father=sonL;
        root=sonL;
    }
    else
    if(fa->left==tmp)
    {
        tmp->left= sonLR;
        sonL->right=tmp;
        fa->left=sonL;
        //变化后需要更新每一个变化节点的父节点指针
        sonL->father=fa;
        if(sonLR!=NULL)sonLR->father=tmp;
        tmp->father=sonL;
    }
    else
    {
        tmp->left= sonLR;
        sonL->right=tmp;
        fa->right=sonL;
        //变化后需要更新每一个变化节点的父节点指针
        sonL->father=fa;
        if(sonLR!=NULL)sonLR->father=tmp;
        tmp->father=sonL;
    }
    //更新bf
    sonL->bf=0;
    tmp->bf=0;
```

LL 型，整体右旋，RR 型与之类似

```
void Rotate_LR(node* tmp)
{
    int s=tmp->left->right->bf;
    //LR型的旋转是：左孩子左旋，再整体右旋
    Rotate_RR(tmp->left);
    Rotate_LL(tmp);
    if(s==1)
    {
        tmp->left->bf=0;
        tmp->bf=-1;
    }
    else if(s==-1)
    {
        tmp->left->bf=1;
        tmp->bf=0;
    }
    else if(s=0)
    {
        tmp->left->bf=0;
        tmp->bf=0;
    }
}
```

LR 型，RL 型与之类似

T3．依次计算获得值即可

```
//任务三
double bst_asl=1.0*t1.suc/len+1.0*t1.fai/(len+1);
double avl_asl=1.0*suce/len+1.0*fail/(len+1);
cout<<bst_asl-avl_asl<<endl;
```

五．总结

通过本次实验，我对于树的各项功能有了更加深入的理解：

二叉搜素树的核心目的是为了让搜索查找和删除操作均可在 O(logN)的情况下完成，而如果数据为持续增大或者持续减小时，由于这时的数据结构就更类似与一条链，插入和删除的操作为 O(N),引入 AVL 树就可以很好的平衡左右子树的数据，避免链式结构的形成