

本人将树相关知识总结为初、中、高三篇，本文属于树结构的中篇，主要阐述几种经典的树形结构，是继承**树的基础**知识之后，进行相关拓展那么本文将以三种典型的树形结构进行总结，从查找的角度来进行分析各个树型结构的区别与优势。

基础篇在：[数据结构——树基础](#)（主要总结树、二叉树、线索二叉树、森林等基础相关知识）

二叉排序树

树表相关概念

查找表：查找表是由同一类型的数据元素（或记录）构成的集合。由于“集合”中的数据元素之间存在着松散的关系，因此查找表是一种应用灵便的结构。

关键字：用来标识一个数据元素(或记录)的某个数据项的值

主关键字：可唯一地标识一个记录的关键字是主关键字

次关键字：用以识别若干记录的关键字是次关键字。

查找表的分类：静态查找表仅用作于**查询**（检索）操作的查找表。如线性表等；**动态查找表**可以用作于**插入**和**删除**等操作的表如树表等，常用于数据库等地方。

平均查找长度：即ASL(Average Search Length)，关键字的平均比较次数。

$$ASL = \sum_{i=1}^n p_i c_i \quad (\text{关键字比较次数的期望值})$$

n:记录的个数

p_i :查找第*i*个记录的概率（通常认为 $p_i=1/n$ ）

c_i :找到第*i*个记录所需的比较次数

CSDN @程序lee

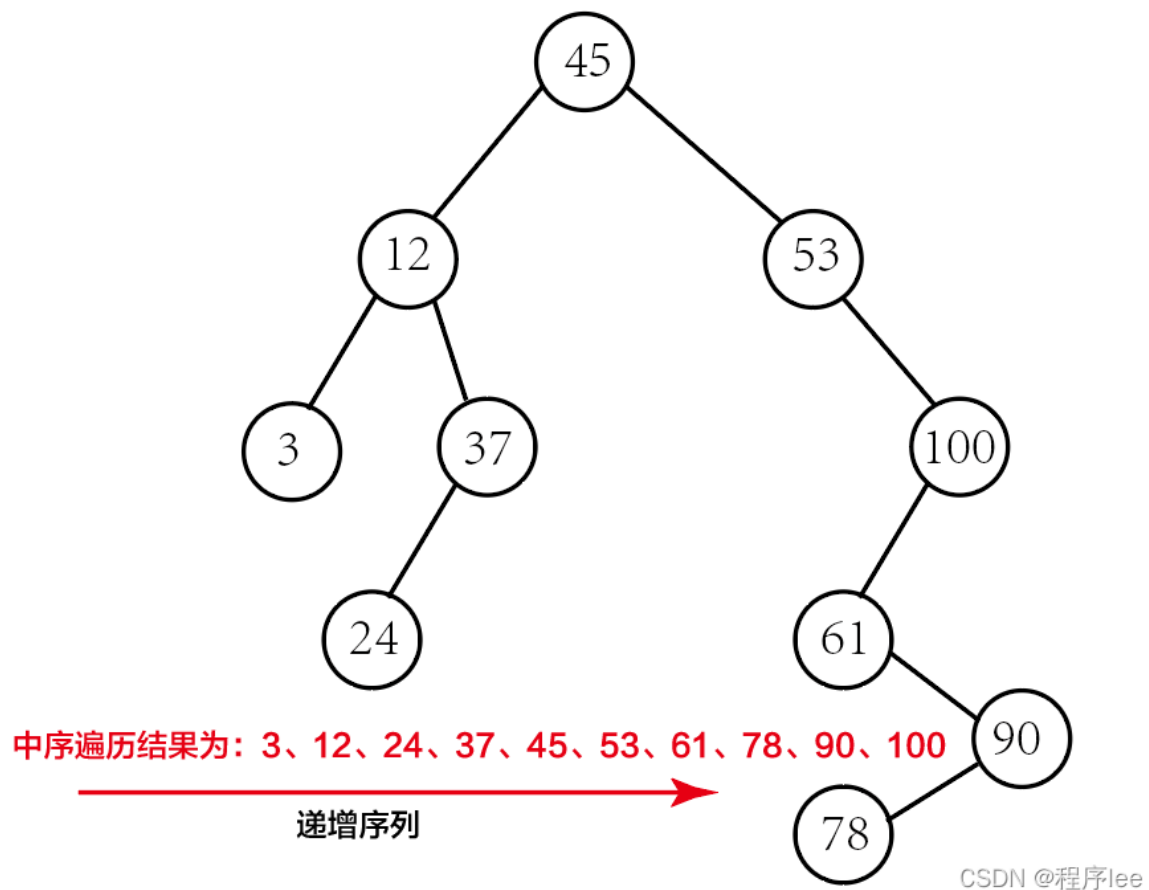
二叉排序树定义

（二叉搜索树、二叉查找树、BST）

若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；

若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；

任意节点的左、右子树也分别为二叉查找树；



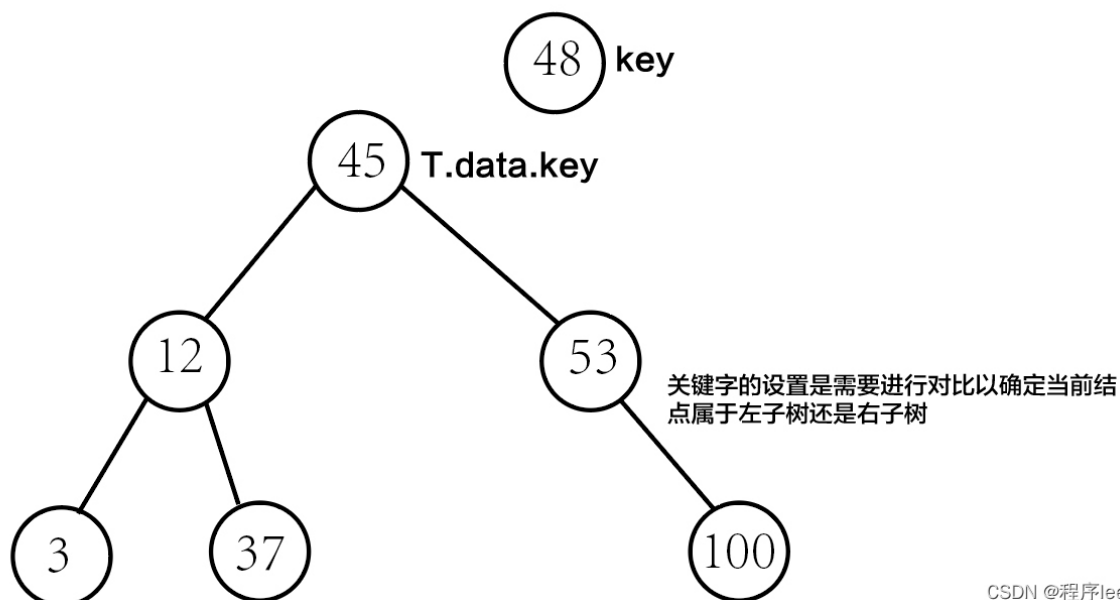
二叉排序树的性质:

中序遍历非空的二叉排序树所得到的数据元素序列是一个按关键字排列的**递增有序序列**。

查找操作

二叉树排序树的存储结构

```
typedef struct {
    keyType key;      //关键字项，需要与给定的关键字进行对比
    infoType otherinfo; //其他数据项
}ElemType;
typedef struct BSTNode {
    ElemType data;      //数据域(可以包含关键字和其他数据)
    struct BSTNode* Lchild, * Rchild; //左右孩子指针
}BSTNode, *BSTree;
```



CSDN @程序lee

查找递归代码

这里注意 **T->data.key** 为原有的树根结点，而 **key** 为需要查询的关键字

```

//递归查找算法
BSTree SearchBST(BSTree T,int key) { //ksy为新的关键字
    if (!T || T->data.key == key) return T; //递归结束条件T为空或给定关键字与原有关键字相等
    else if (T->data.key < key) return SearchBST(T->Rchild,key); //右子树查找
    else return SearchBST(T->Lchild, key); // 左子树查找
}

```

这里的 判断条件 **!T || T->data.key == key** 中T若为空return返回的是空指针，也可以将这句改成两行代码 即 T为空则返回false，相等则返回T。

查找非递归代码

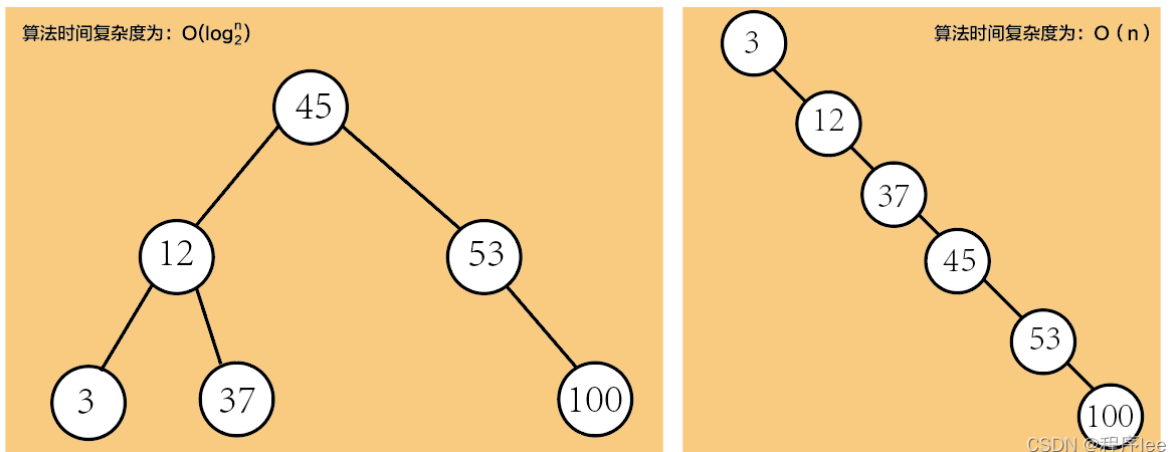
```

BSTree SearchBST(BSTree T, int key){
    while (!T || T->data.key == key) {
        if (T->data.key>key) {
            T=T->Lchild; //小于在左子树上查找。指针向后移动
        }
        else
        {
            T=T->Rchild; //大于在右子树上查找。指针向后移动
        }
    }
    return T;
}

```

二叉排序树的查找分析

由下图可知，给定一序列如：3,12,37,45,53,100 则可以产生不同的二叉排序树，得出的算法时间复杂度也不同，而平均查找长度也是左面偏小。因此可以看出越趋于平衡的二叉排序树查找效率越高。



插入操作

二叉排序树的插入的只能是**叶子结点**，因此插入后需要将 左右子树置空，并且将树与新节点连接有返回值时传址，传值，引用写法不同，可以参考不同的版本来写。我这里选择传值并用 T 来接收返回值，因此可以递归。当然会c++的可以选择引用的方式。

```

BSTree InsertBST(BSTree T, ElemType e) {
    BSTree s;
    if (!T) {           //找到要插入的地方
        s = (BSTree)malloc(sizeof(BSTNode)); //开辟空间将s指向这里
        s->data = e;
        s->Lchild = s->Rchild = NULL; //新插入的是叶子结点，因此需要置空
        T = s;                      //连接结点与树
    }
    else if(T->data.key>e.key)
    {
        T->Lchild=InsertBST(T->Lchild,e);
    }
    else if(T->data.key<e.key) {
        T->Rchild=InsertBST(T->Rchild,e);
    }
    return T;              //相等 直接会返回T
}
    
```

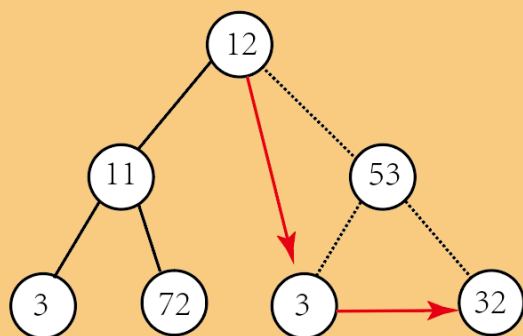
删除操作

不论是二叉树还是二叉搜索树都需要考虑三种情况即：①叶子结点 无左右孩子 ② 只有一个孩子的情况 ③ 有两个孩子的情况

对于前两种情况二叉树与二叉搜索树处理方式是一样的，主要区别在于有左右孩子的情况，二叉搜索树势必要考虑到关键字的顺序。如图所示，需要删除关键字为 53 二叉树中可以直接将53的左孩子作为新的根结点，则32作为新结点的右孩子。

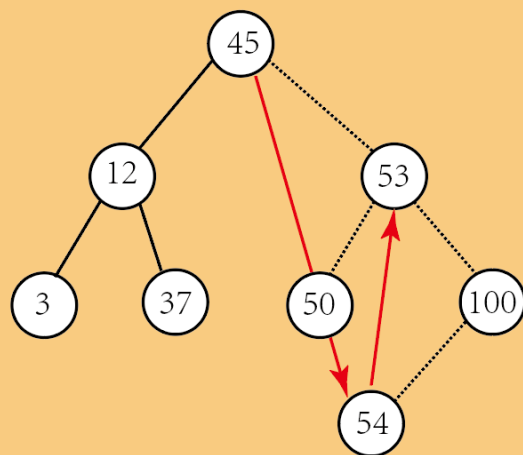
但在二叉排序中的处理方式是将要删除的53 的后继节点 54作为新的节点，这样可以保证 顺序不会发生错误。原53的左右孩子变成54的左右孩子。

二叉树



普通二叉树可以默认选择左面或右面的结点作为新的根结点

二叉排序树



二叉排序树需要考虑到关键字的顺序问题，然后在进行替换操作

CSDN @程序lee

代码实现

① 构建遍历函数

```

BSTree DeleteBST(BSTree &T, ElemType e) { //使用引用传参
    if (!T) return 0; //遍历全部未找到与e相等的关键字
    else if (T->data.key == e.key)
        { Delete(T,e); } //相等则执行删除函数，注意这里T传的是引用类型
    else if (T->data.key > e.key) { DeleteBST(T->Lchild, e); } //执行左子树
    else { DeleteBST(T->Rchild, e); } //执行右子树
}

```

这里删除会出现三种情况，0个孩子，1个孩子，2个孩子。

① 构建删除函数

```

//删除只有一个节点的状况
Delete(BSTree &T, ElemType e) { //叶子结点 //即0孩子
    if (T->Lchild==NULL&& T->Rchild == NULL) {
        BSTree p = T;
        T = NULL; //T为空同时因为是引用，因此也改变了双亲的指针为空
        free(p); //释放空间，防止成为野指针
    }
    else if (T->Lchild == NULL) { //只有右孩子 //即1孩子
        BSTree p = T;
        T = T->Rchild; //指向右孩子
        free(p);
    }
    else if (T->Rchild == NULL) { //只有左孩子 //即1孩子
        BSTree p = T;
        T = T->Lchild; //指向左孩子
        free(p);
    }
}
}

```

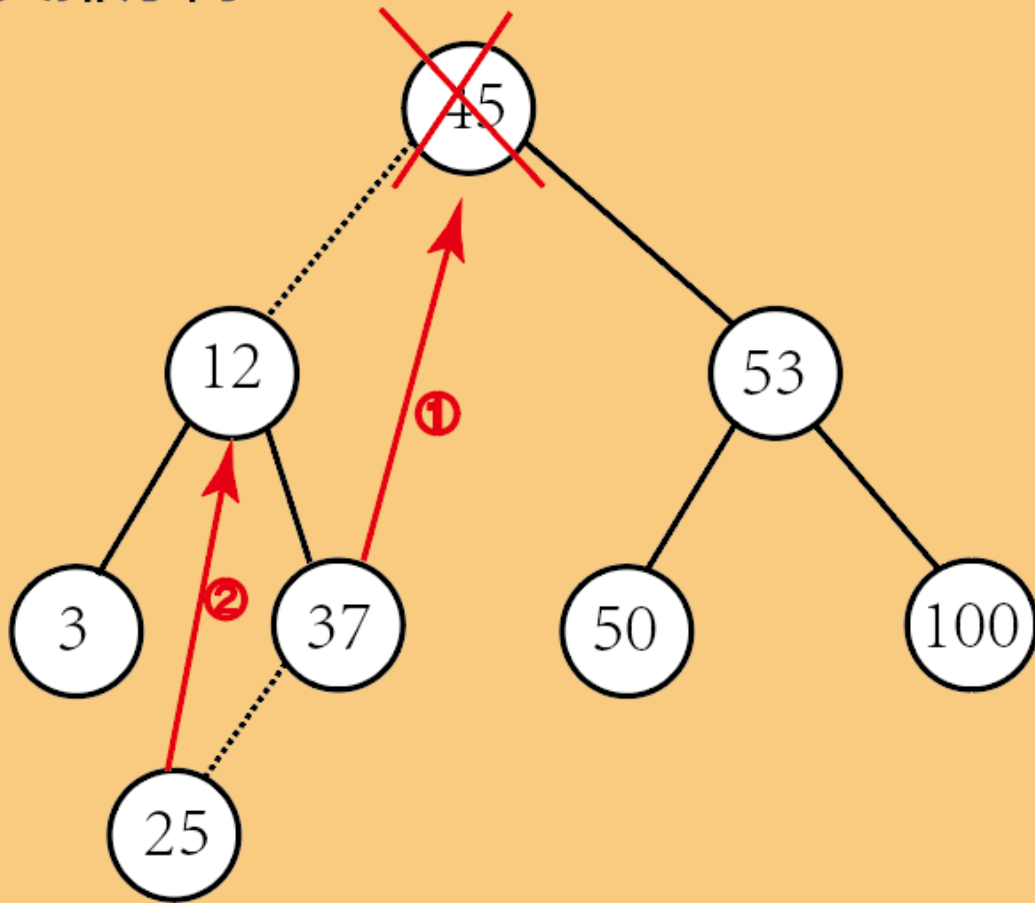
① 构建删除函数——删除二个节点算法

第一步：需要在左子树中找到最大的数（位于根结点左子树的右子树的右子树的右子树.....）

第二步：将他替换到根结点的位置

第三步：将他的左孩子25变成原来37的位置，也就是12的右孩子

二叉排序树

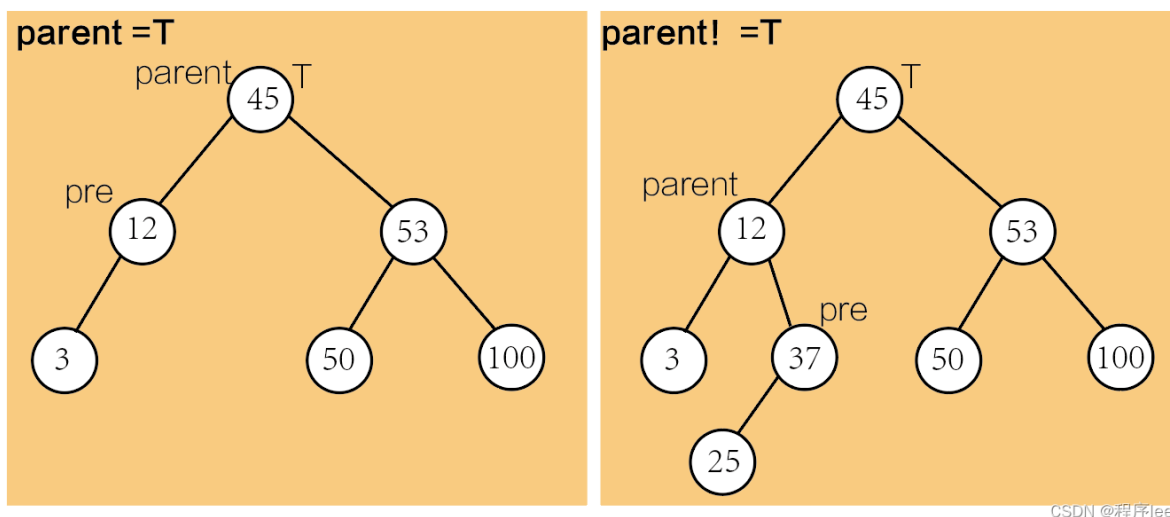


选择原根结点左子树中最大的数，或右子树中最小的作为新的根结点

CSDN @程序lee

```
else {    //俩孩子
    BSTree parent = T;
    BSTree pre = T->Lchild;
    while (pre->Rchild) {    //不断指向右子树的右子树
        parent = pre;        //记录双亲结点位置
        pre = pre->Rchild;    //指向最大值位置
    }
    T->data = pre->data;
    if (parent != T) {    // 最大值就是根结点的左孩子的情况
        parent->Rchild = pre->Lchild;
    }
    else
    {
        parent->Lchild = pre->Lchild;
        free(pre);
    }
}
```

```
}
```



平衡二叉树 AVL

那么通过以上可知，左右子树的深度差值越小，实际上查找效率越高。因此便引出了平衡二叉树的概念。为避免树的高度增长过快，降低二叉排序树的性能，规定在插入和删除二叉树结点时，要保证任意结点的左、右子树高度差的**绝对值不超过1**，将这样的二叉树称为**平衡二叉树**

平衡二叉树一定是二叉排序树。二叉排序树不一定是平衡二叉树 如上右图是AVL 左图是BST

相关概念

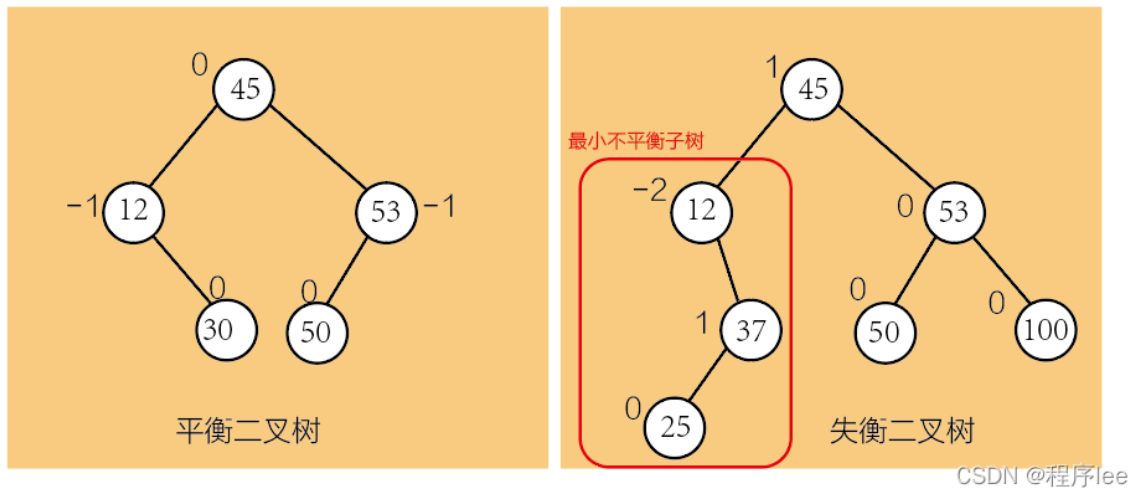
一棵**平衡二叉树**或者是空树，或者是具有下列性质

- ①二叉排序树:左子树与右子树的高度之差的绝对值小于等于1;
- ②左子树和右子树也是平衡二叉排序树。

为了方便起见，给每个结点附加一个数字，给出该结点左子树与右子树的高度差。这个数字称为结点的**平衡因子BF**。平衡因子只能是 0,1,-1。

平衡因子 = 结点左子树的高度 - 结点右子树的高度

对于一棵有n个结点的AVL树，其高度保持在 $O(\log_2 n)$ 数量级，ASL也保持在 $O(\log_2 n)$ 量级



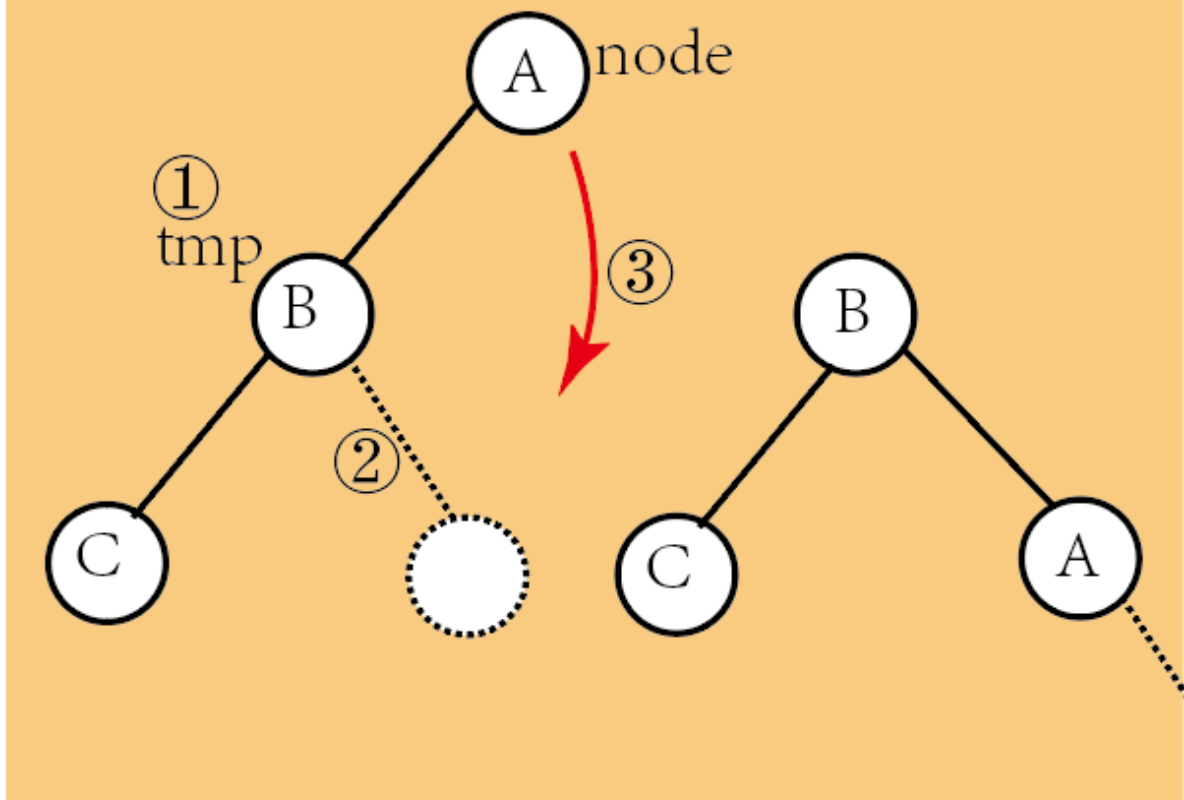
平衡二叉树调整方法

由上图可知，当平衡二叉树在插入时可能会产生失衡的状况，总共有以下四种状况：

LL平衡旋转（左单旋转）

由于插入过程中在左孩子的左子树上插入了节点导致的失衡

LL平衡旋转（左单旋转）



```
void Left_Left(AVLTree node, AVLTree*root) {  
    AVLTree tmp = node->lchild;  
    node->lchild = tmp->rchild;  
    tmp->rchild = node;  
}
```

CSDN @程序lee

LR平衡旋转(先左后右双旋转)

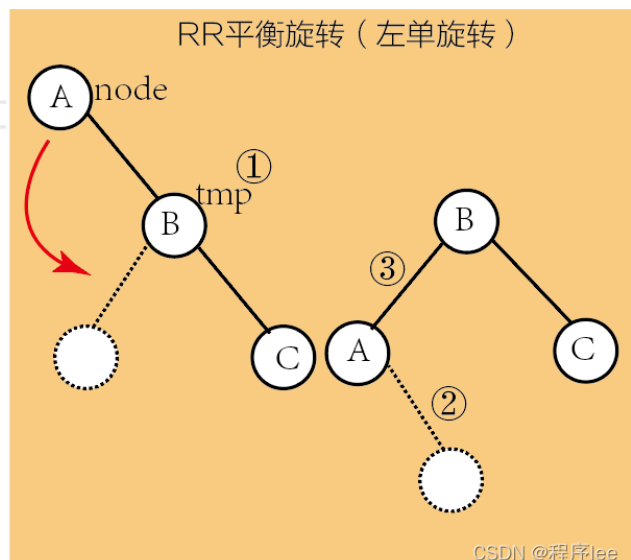
由于插入过程中在左孩子的右子树上插入了节点导致的失衡

RR平衡旋转（左单旋转）

由于插入过程中在右孩子的右子树上插入了节点导致的失衡

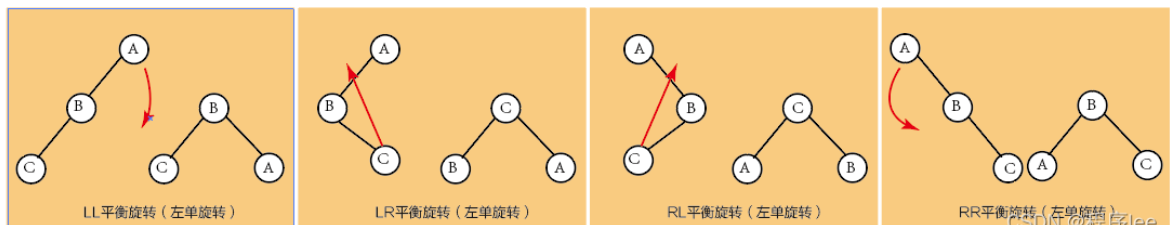
```
void Right_Right(AVLTree node, AVLTree* root){
    AVLTree tmp = node->rchild; ①
    node->rchild = tmp->lchild; ②
    tmp->lchild = node; ③
}
```

RR代码实现与图形解构



RL平衡旋转（先右后左双旋转）

由于插入过程中在右孩子的左子树上插入了节点导致的失衡



平衡二叉树创建完整代码

步骤：按照二叉搜索树进行创建，在子树创建时需要进行**判断**和**旋转**。重点在于LL,LR,RR,RL旋转的逻辑。关于删除等操作就不做展示了，基本上和二叉搜索树是一样的原理，但每次添加或删除后是需要进行旋转的。

```
typedef struct AVLNode {
    int data;
    int height; //数据 高度 左右孩子
    struct AVLNode* lchild;
    struct AVLNode* rchild;
}AVLNode,*AVLTree;

int getHeight(AVLTree node) {
    return node ? node->height : 0; //有值返回值 没有值返回零
}

int max(int a, int b) {
    return a > b ? a : b;
}

void Left_Left(AVLTree node, AVLTree*root){ //LL旋转
    AVLTree tmp = node->lchild;
    node->lchild = tmp->rchild;
    tmp->rchild = node;
    node->height = max(getHeight(node->lchild), getHeight(node->rchild)) + 1; //高度改变
    tmp->height = max(getHeight(tmp->lchild), getHeight(tmp->rchild)) + 1; //高度改变
    *root = tmp;
}
```

```

void Right_Right(AVLTree node, AVLTree* root){
    AVLTree tmp = node->rchild;
    node->rchild = tmp->lchild;
    tmp->lchild = node;
    node->height = max(getHeight(node->lchild), getHeight(node->rchild))+1; //求
    最大值
    tmp->height = max(getHeight(tmp->lchild), getHeight(tmp->rchild)) + 1;
    *root = tmp;
}

void Inserte(AVLTree* T,int data){
    if (*T==NULL) {
        *T = (AVLTree)malloc(sizeof(AVLNode));
        (*T)->data = data;
        (*T)->height = 0;
        (*T)->lchild =NULL;
        (*T)->rchild = NULL;
    }
    else if(data<(*T)->data){
        Inserte(&(*T)->lchild,data);
        //判断平衡因子是否合理
        //求出左右子树的高度
        int Lheight = getHeight((*T)->lchild); //获取左子树高度
        int Rheight = getHeight((*T)->rchild); //获取右子树高度
        //判断高度差 LL LR
        if (Rheight- Lheight==2) {
            if (data< (*T)->lchild->data) {
                //LL调整
                Left_Left(*T,T);
            }else{
                //LR调整
                Right_Right((*T)->lchild,&(*T)->lchild);
                Left_Left(*T, T);
            }
        }
    }
    else if (data > (*T)->data) {
        Inserte(&(*T)->rchild, data);
        int Lheight = getHeight((*T)->lchild); //获取左子树高度
        int Rheight = getHeight((*T)->rchild); //获取右子树高度
        //判断高度差 RR RL
        if (Rheight - Lheight == 2) {
            if (data > (*T)->rchild->data) {
                //RR调整
                Right_Right(*T, T);
            }
            else {
                //RL调整
                Left_Left((*T)->rchild, &(*T)->rchild);
                Right_Right(*T, T);
            }
        }
    }
    (*T)->height = max(getHeight((*T)->lchild), getHeight((*T)->rchild)) + 1;
}

void preOrder(AVLTree T) { //前序遍历
    if (T) {
        printf("%d ", T->data);
    }
}

```

```

        preOrder(T->lchild);
        preOrder(T->rchild);
    }
}

int main (){
    AVLTree T=NULL;
    int data[6] = {1,2,3,4,6,7};
    for (int i = 0;i < 6;i++) {
        Inserte(&T,data[i]);
    }
    preOrder(T);//先序遍历
    return 0;
}

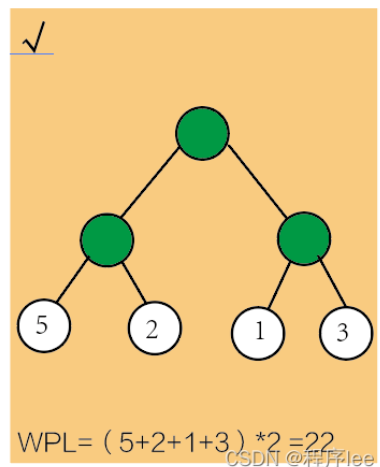
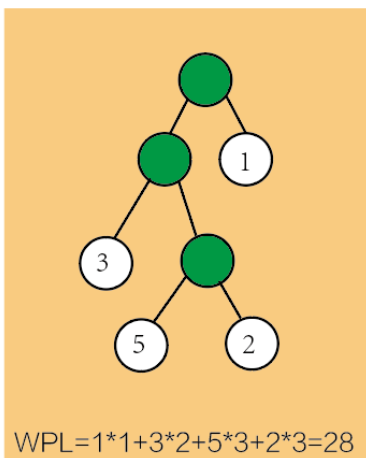
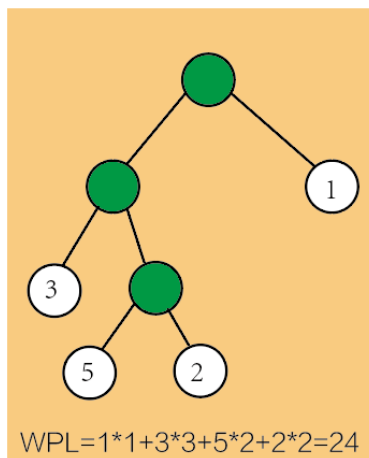
```

哈夫曼树及哈夫曼编码

相关概念

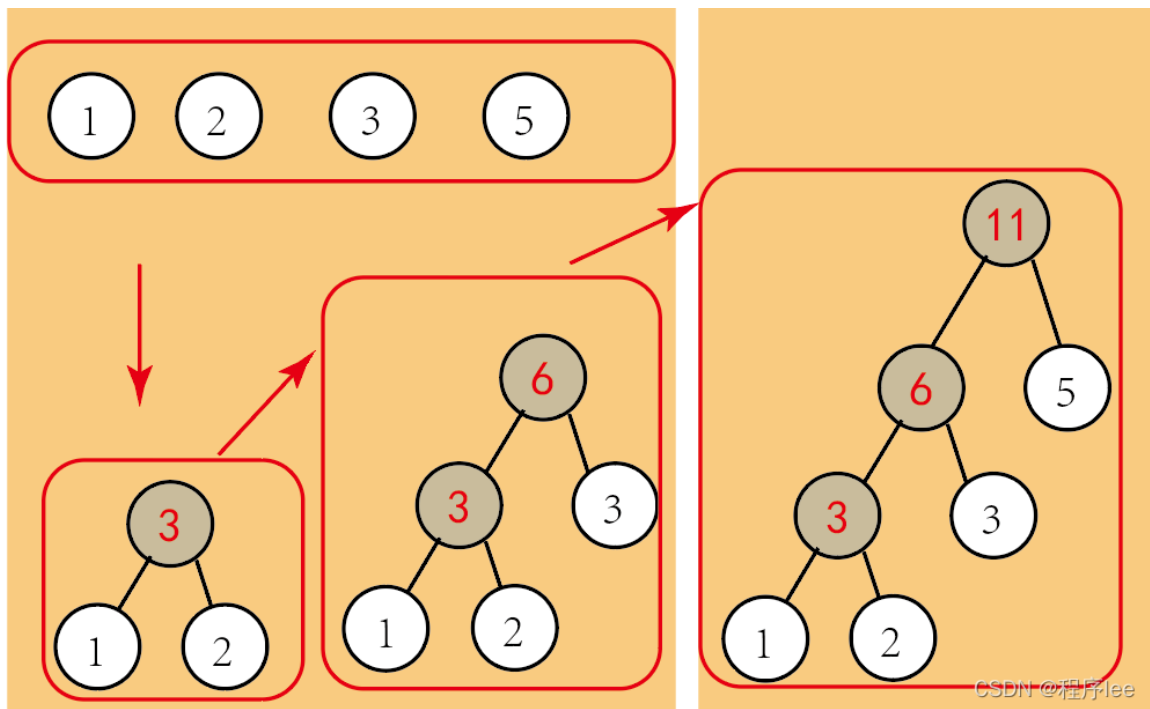
在许多应用中，树中结点常常被赋予一个表示某种意义的数值，称为该结点的**权**。从树的根到任意结点的**路径长度**（经过的边数）与该结点上权值的乘积，称为该结点的带权路径长度。树中所有叶结点的带权路径长度之和称为该树的**带权路径长度 WPL**

在含有n个带权叶结点的二叉树中，其中带权路径长度（WPL）最小的二叉树称为**哈夫曼树**，也称**最优二叉树**



哈夫曼树构造

- ①首先将以下的1,2,3,5看做是一个森林 ② 选择最小的两个构成第一个树 其跟为二者之和
- ③再重复第二步即可



图片来自 bilibili 懒猫老师

哈夫曼树存储结构

weight	parent	lchild	rchild
--------	--------	--------	--------

```
typedef struct{
    int weight;
    int parent, lchild, rchild;
}HNode,*HuffmanTree;
```

weight: 权值域, 保存该结点的权值;
lchild: 指针域, 结点的左孩子结点在数组中的下标;
rchild: 指针域, 结点的右孩子结点在数组中的下标;
parent: 指针域, 该结点的双亲结点在数组中的下标。

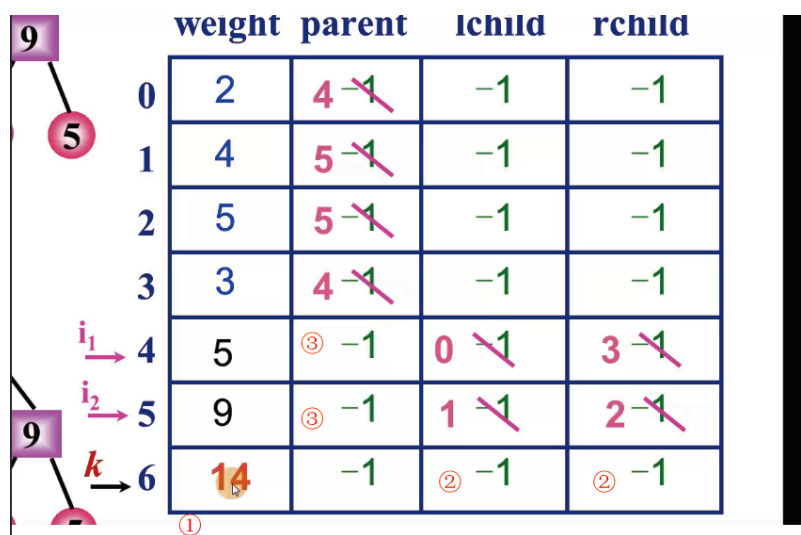
CSDN @程序lee

```
typedef struct{
    int weight;
    int parent, lchild, rchild;
}HNode,*HuffmanTree;
```

- 1、每个初始结点最终都成为叶结点，且**权值越小**的结点到根结点的路径长度**越大**。
- 2、构造过程中共新建了 $n-1$ 个结点（双分支结点），因此哈夫曼树的结点总数为 $2n-1$
- 3、每次构造都选择2棵树作为新结点的孩子，因此哈夫曼树中不存在度为1的结点。

哈夫曼树代码

可以采用先序遍历查看结果，重点在于select比较函数和哈夫曼表的操作顺序。在看代码之前一定要将表的顺序理清。



	weight	parent	lchild	rchild
0	2	4 -1	-1	-1
1	4	5 -1	-1	-1
2	5	5 -1	-1	-1
3	3	4 -1	-1	-1
4	5	③ -1	0 -1	3 -1
5	9	③ -1	1 -1	2 -1
6	14	-1	② -1	② -1

```

T->data[i].weight = T->data[Min].weight + T->data[secondMin].weight;
T->data[i].lchild = Min;
T->data[i].rchild = secondMin;
T->data[Min].parent = i;
T->data[secondMin].parent = i;

```

CSDN @程序lee

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
typedef struct TreeNode {
    int weight; //权值
    int parent; // 双亲结点位置
    int rchild, lchild;
}TreeNode;
typedef struct HFTree {
    TreeNode* data;
    int length; //当前长度
}HFTree;
HFTree* InitTree(int* weight, int length) {
    HFTree* T = (HFTree*)malloc(sizeof(HFTree));
    T->data = (TreeNode*)malloc(sizeof(TreeNode) * (2 * length - 1));
    T->length = length;
    for (int i = 0; i < length; i++) {
        T->data[i].weight = weight[i];
        T->data[i].parent = 0;
        T->data[i].lchild = -1;
        T->data[i].rchild = -1;
    }
    return T;
}

int* SelectMin(HFTree*T) {
    int min = 10000;
    int secondMin = 10000;
    int IndexMin;
    int SecondIndex;
    for (int i = 0; i < T->length; i++) {
        if (T->data[i].parent == 0) { //parent=0 才进行比较
            if (T->data[i].weight < min) { //选出权值最小的

```

```

        min = T->data[i].weight;
        IndexMin = i;
    }
}
}
for (int i = 0; i < T->length; i++) {
    if (T->data[i].parent == 0 && i != IndexMin) { //选出权值第二小的
        if (T->data[i].weight < secondMin) {
            secondMin = T->data[i].weight;
            SecondIndex = i;
        }
    }
}
int* res = (int*)malloc(sizeof(int) * 2);
res[0] = IndexMin; //返回最小值
res[1] = SecondIndex; //返回最第二小值
return res;
}
void CreatHFTree(HFTree* T) {
    int Min;
    int secondMin;
    int lenthMax = T->length * 2 - 1; //哈夫曼树最大值
    for (int i = T->length; i < lenthMax; i++) {
        int* res = SelectMin(T);
        Min = res[0];
        secondMin = res[1];
        T->data[i].weight = T->data[Min].weight + T->data[secondMin].weight; //父
节点权值
        T->data[i].lchild = Min; T->data[i].rchild = secondMin; //父节点的 左右孩
子值
        T->data[Min].parent = i; T->data[secondMin].parent = i; //孩子结点parent
值
        T->length++;
    }
}
int main() {
    int weight[4] = {1,2,3,4};
    HFTree* T = InitTree(weight,4);
    int* res = SelectMin(T);
    return 0;
}

```