C/C++从入门到精通-高级程序员之路

【 数据结构 】

树及其企业级应用

第 1 节 树的故事导入

故事情节

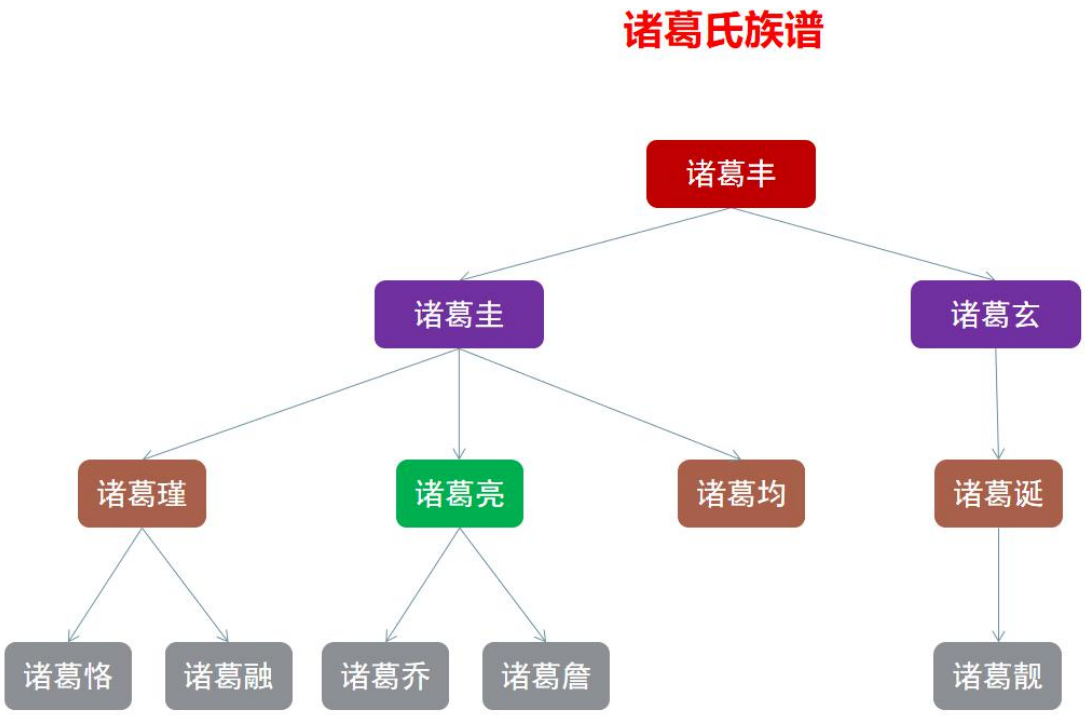
家谱：又称族谱、宗谱等。是一种以表谱形式，记载一个家族的世系繁衍及重要人物事迹的书。家谱是一种特殊的文献，就其内容而言，是中华文明史中具有平民特色的文献，记载的是同宗共祖血缘集团世系人物和事迹等方面情况的历史图籍。

<Martin 偶像> 诸葛亮

诸葛氏族谱

远祖	第一代	第二代	第三代	第四代	第五代
诸葛丰（汉司隶校尉）	诸葛圭	诸葛瑾（圭长子）	诸葛恪（瑾长子）	诸葛绰（恪长子）	
				诸葛竦（恪中子）	
				诸葛建（恪少子）	
			诸葛融（恪弟）		
		诸葛亮（圭次子）	诸葛乔	诸葛攀	诸葛显
			诸葛瞻	诸葛尚（瞻长子）	
				诸葛京（瞻次子）	
		诸葛均（圭三子）			
	诸葛玄（圭弟）				
		诸葛诞（亮族弟）	诸葛靓（诞小子）	诸葛恢	

更清晰的描述：

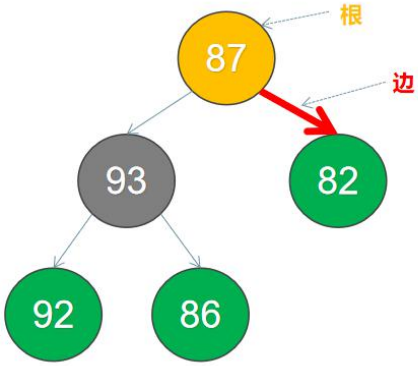


《族谱树状图》

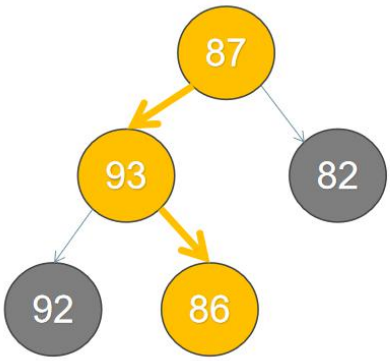
树状图是一种数据结构，它是由 n（n>=1）个有限结点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

每个结点有零个或多个子结点；没有父结点的结点称为根结点；每一个非根结点有且只有一个父结点；除了根结点外，每个子结点可以分为多个不相交的子树；

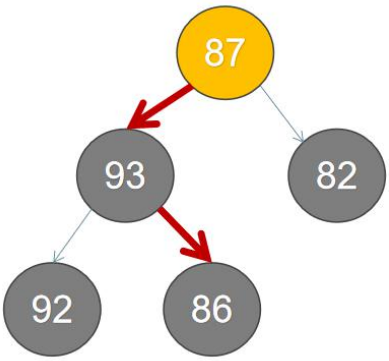
专业术语	中　文	描　述
Root	根节点	一棵树的顶点
Child	孩子节点	一个结点含有的子树的根结点称为该结点的子结点
Leaf	叶子节点	没有孩子的节点
Degree	度	一个节点包含的子树的数量
Edge	边	一个节点与另外一个节点的连接
Depth	深度	根节点到这个节点经过的边的数量
Height	节点高度	从当前节点到叶子节点形成路径中边的数量
Level	层级	节点到根节点最长路径的边的总和
Path	路径	一个节点和另一个节点之间经过的边和 Node 的序列



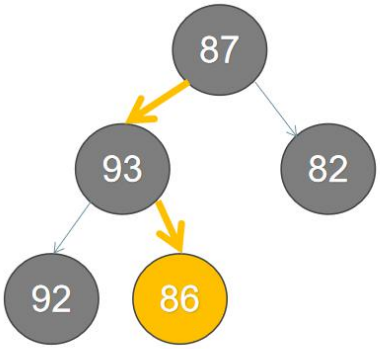
87 为根节点
93、82 为87的子节点
92、86、82为叶子节点



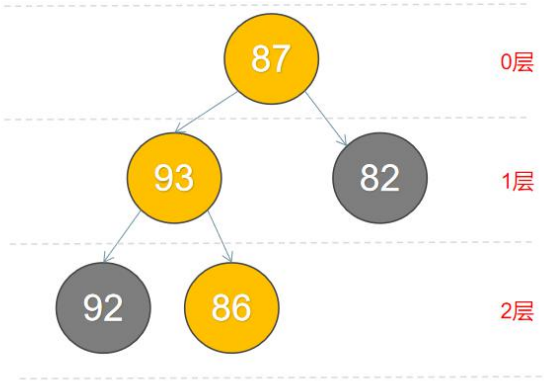
路径



节点高度： 该节点到叶子节点最长路径的边数之和， 节点87的高度是 2



节点深度： 该节点到根节点最长路径的边数之和， 节点86的深度是 2 ，节点87 的深度为 0



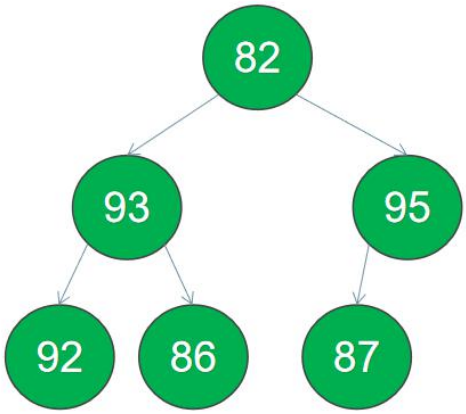
节点层级： 等于该节点到根节点最长路径的边数之和， 节点87的层是 0，93 的层为1， 86的层是 2

第 2 节 二叉树的原理精讲

我们以前介绍的线性表一样，一个没有限制的线性表应用范围可能有限，但是我们对线性表进行一些限制就可以衍生出非常有用的数据结构如栈、队列、优先队列等。

树也是一样，一个没有限制的树由于太灵活，控制起来比较复杂。如果对普通的树加上一些人为的限制，比如节点只允许有两个子节点，这就是我们接下来要介绍的二叉树。

二叉树是一个每个结点最多只能有两个分支的树，左边的分支称之为左子树，右边的分支称之为右子树。



（二叉树）

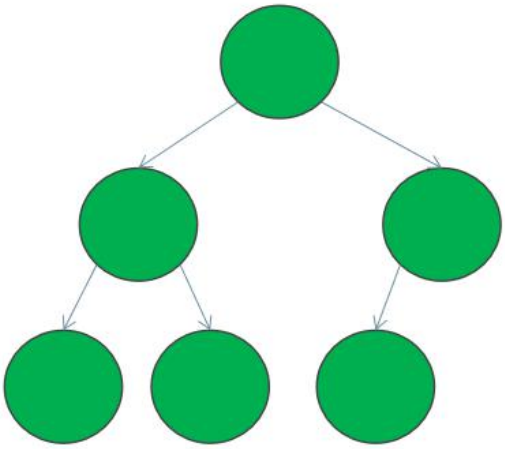
(1) 在非空二叉树中，第 i-1 层的结点总数不超过 2^{i-1}，i>=1；

(2) 深度为 h-1 的二叉树最多有2^h-1个结点(h>=1)，最少有 h 个结点；

(3) 对于任意一棵二叉树，如果其叶结点数为 N0，而度数为 2 的结点总数为 N2，则 N0=N2+1；

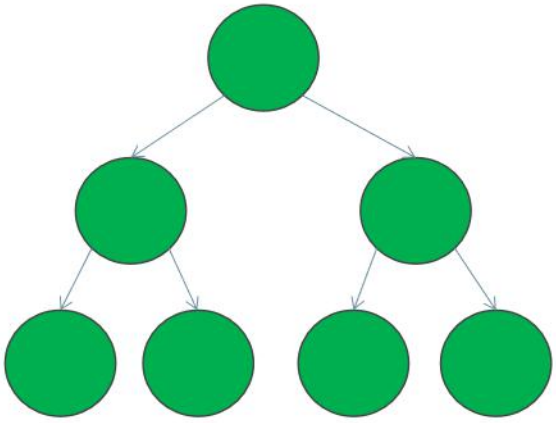
常见二叉树分类：

（1）完全二叉树 — 若设二叉树的高度为 h，除第 h 层外，其它各层（1～h-1）的结点数都达到最大个数，第 h 层有叶子节点，并且叶子结点都是从左到右依次排布，这就是完全二叉树（**堆**就是完全二叉树）。



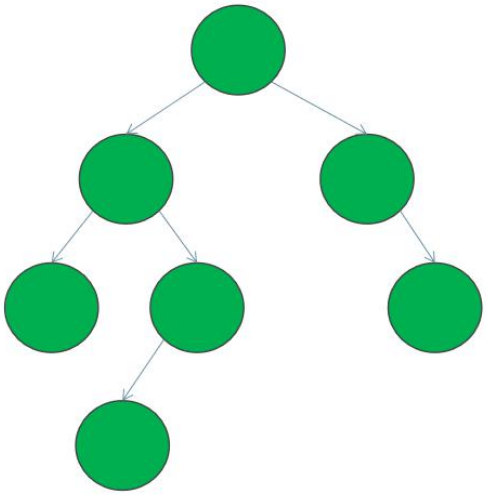
（完全二叉树）

（2）**满二叉树 —** 除了叶结点外每一个结点都有左右子节点且叶子结点都处在最底层的二叉树。



（满二叉树）

（3）平衡二叉树 — 又被称为 AVL 树，它是一颗空树或左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。



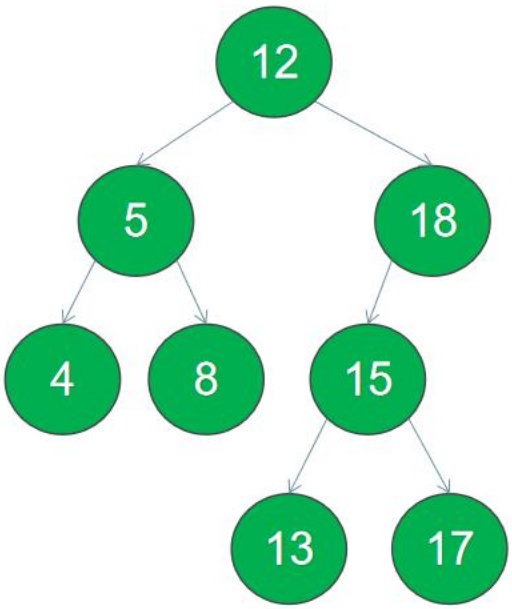
（平衡二叉树）

（4）二叉搜索树 — 又称二叉查找树、二叉排序树（Binary Sort Tree）。它是一颗空树或是满足下列性质的二叉树：

1）若左子树不空，则左子树上所有节点的值均小于或等于它的根节点的值；

2）若右子树不空，则右子树上所有节点的值均大于或等于它的根节点的值；

3）左、右子树也分别为二叉排序树。



（二叉排序树）

（5）红黑树 —— 是每个节点都带有颜色属性（颜色为红色或黑色）的自平衡二叉查找树，满足下列性质：

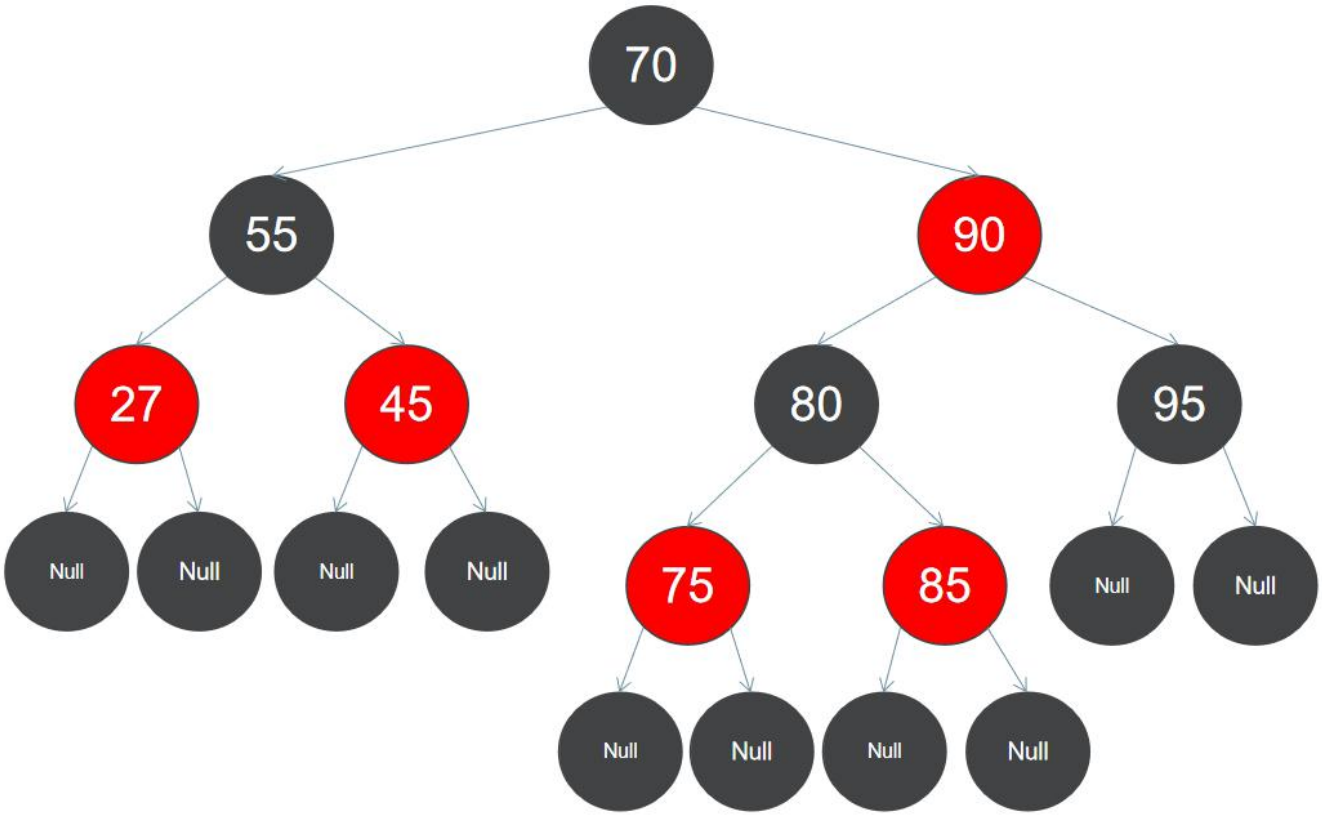
1）节点是红色或黑色；

2）根节点是黑色；

3）所有叶子节点都是黑色；

4）每个红色节点必须有两个黑色的子节点。(从每个叶子到根的所有路径上不能有两个连续的红色节点。)

5）从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。



（红黑树）

第 3 节　二叉搜索树的算法实现

1. 当我们要在一组数中要找到你女朋友（或未来女朋友）的年龄，比如 26？你该怎么找？

61	25	7	11	15	99	19	21	5	26

答案： 从左至右 或 从右至左遍历一次，找到这个数字

2. 当我们把数据进行排序（按照从小到大的顺序排列）后，再查找相应的这条记录？还是用上面的方法吗？

5	7	11	15	19	21	25	26	61	99

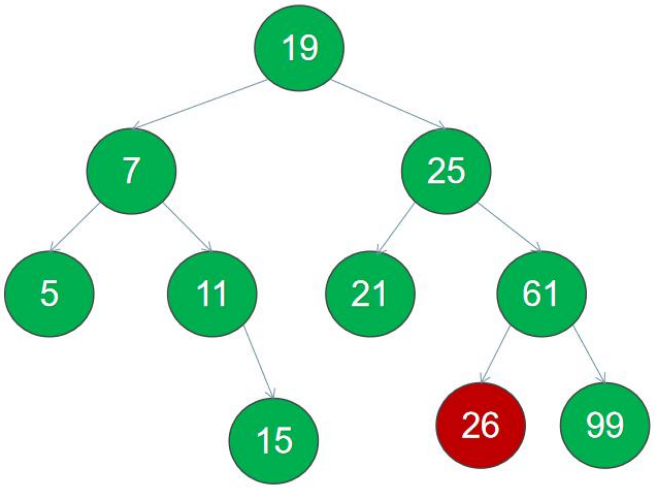
答案： 最快的方式，是采用折半法（俗称二分查找）

思考： 当我们有新的数据加进来，或者删除其中的一条记录，为了保障查找的效率，我们仍然要保障数组有序，但是，会碰到我们讲顺序表时的问题，涉及到大量数据的移动！在插入和删除操作上，就需要耗费大量的时间（需进行元素的移位），能否有一种既可以使得插入和删除效率不错，又可高效查找的数据结构和算法呢？

抛砖： 首先解决一个问题，插入时不移动元素，我们可以想到链表，但是要保证其有序的话，首先得遍历链表寻找合适的位置，那么又如何高效的查找合适的位置呢，能否可以像二分一样，通过一次比较排除一部分元素？

引玉： 那么我们可以用二叉树的形式，以数据集第一个元素为根节点，之后将比根节点小的元素放在左子树中，将比根节点大的元素放在右子树中，在左右子树中同样采取此规则。那么在查找 x 时，若 x 比根节点小可以排除右子树所有元素，去左子树中查找（类似二分查找），这样查找的效率非常好，而且插入的时间复杂度为 O（h），h 为树的高度，较 O（n）来说效率提高不少。故二叉搜索树用作一些查找和插入使用频率比较高的场景。

5	7	11	15	19	21	25	26	61	99



（二叉排序树）

二叉树一般采用链式存储方式：每个结点包含两个指针域，指向两个孩子结点，还包含一个数据域，存储结点信息。

（二叉树节点）

节点结构体的定义:

```
#define MAX_NODE 1024


#define isLess(a, b) (a<b)
#define isEqual(a, b) (a==b)

typedef  int ElemType;

typedef struct _Bnode{
    ElemType data;                  //元素类型
    struct _Bnode *lchild, *rchild;//指向左右孩子节点

}Bnode, *Btree;
```

二叉搜索树插入节点

将要插入的结点 e，与节点 root 节点进行比较，若小于则去到左子树进行比较，若大于则去到右子树进行比较，重复以上操作直到找到一个空位置用于放置该新节点。

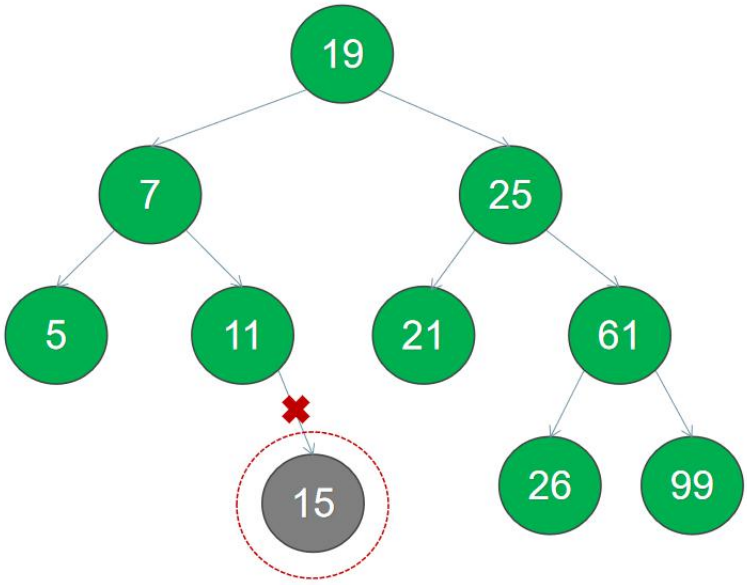
```
bool InsertBtree(Btree **root, Bnode *node){
    Bnode *tmp = NULL;
    Bnode *parent = NULL;

    if(!node){
        return false;
    }else {//清空新节点的左右子树
        node->lchild = NULL;
        node->rchild = NULL;
    }

    if(*root){//存在根节点
        tmp= *root;
    }else{   //不存在根节点
```

```
        *root = node;
        return true;
    }

    while(tmp != NULL){
        parent = tmp;//保存父节点
        printf("父节点：  %d\n", parent->data);
        if(isLess(node->data,tmp->data)){
            tmp = tmp->lchild;
        }else{
            tmp = tmp->rchild;
        }
    }

    //若该树为空树,则直接将 node 放置在根节点上
    if(isLess(node->data, parent->data)){//找到空位置后，进行插入
        parent->lchild = node;
    }else{
        parent->rchild = node;
    }

    return true;
}
```

二叉搜索树删除节点

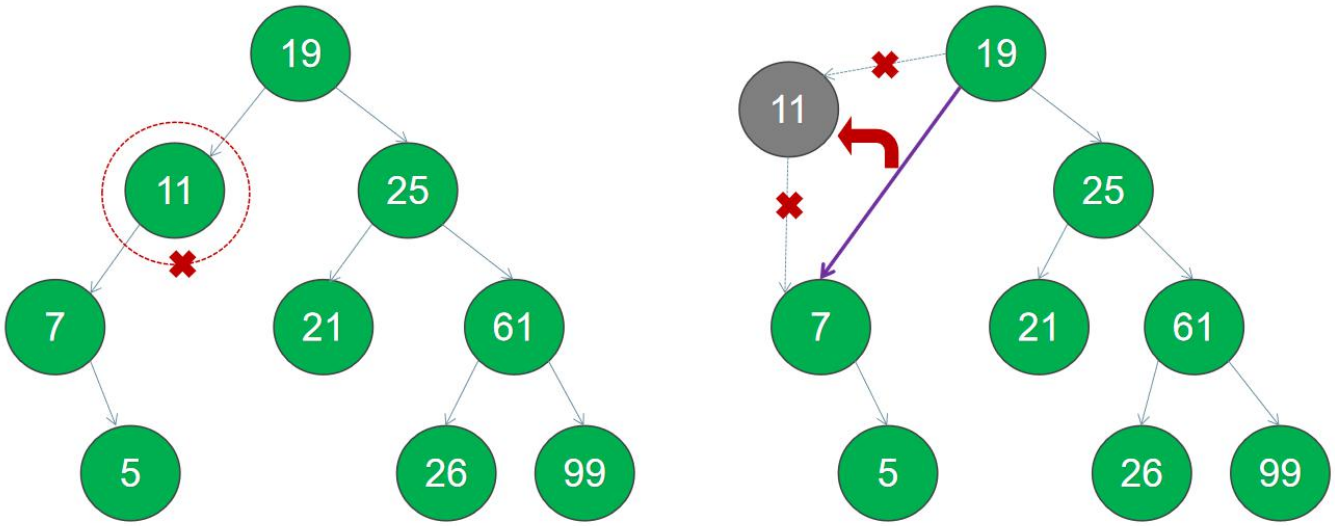
将要删除的节点的值，与节点 root 节点进行比较，若小于则去到左子树进行比较，若大于则去到右子树进行比较，重复以上操作直到找到一个节点的值等于删除的值，则将此节点删除。删除时有 4 中情况须分别处理：

✓ **删除节点不存在左右子节点，即为叶子节点，直接删除**



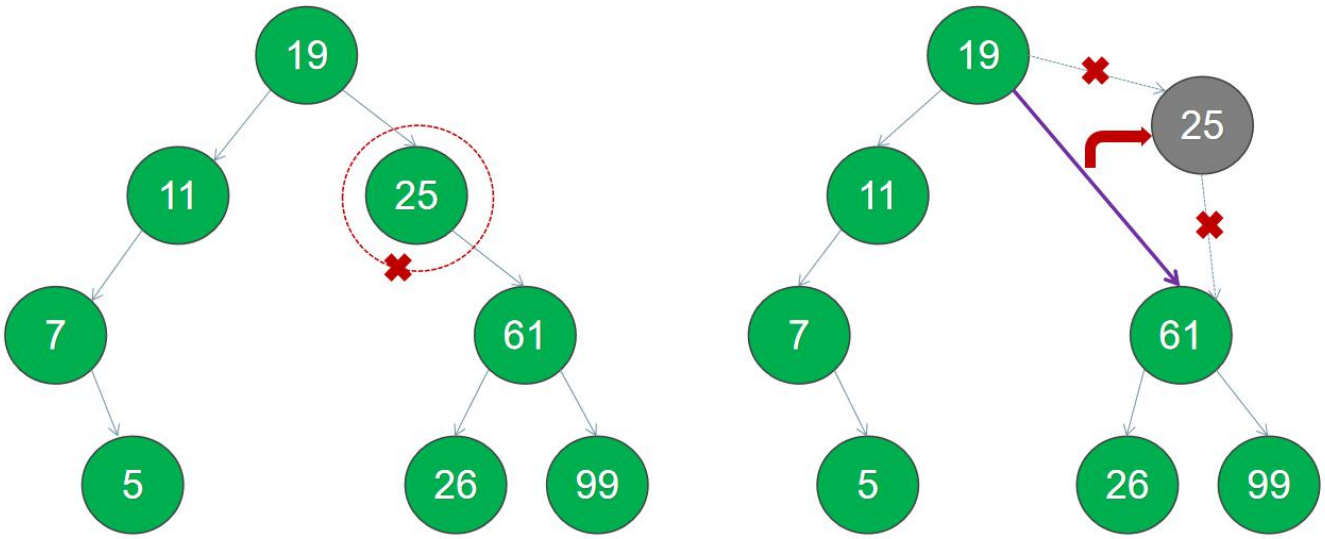
情形1 - 叶子节点直接删除

✓ **删除节点存在左子节点，不存在右子节点，直接把左子节点替代删除节点**



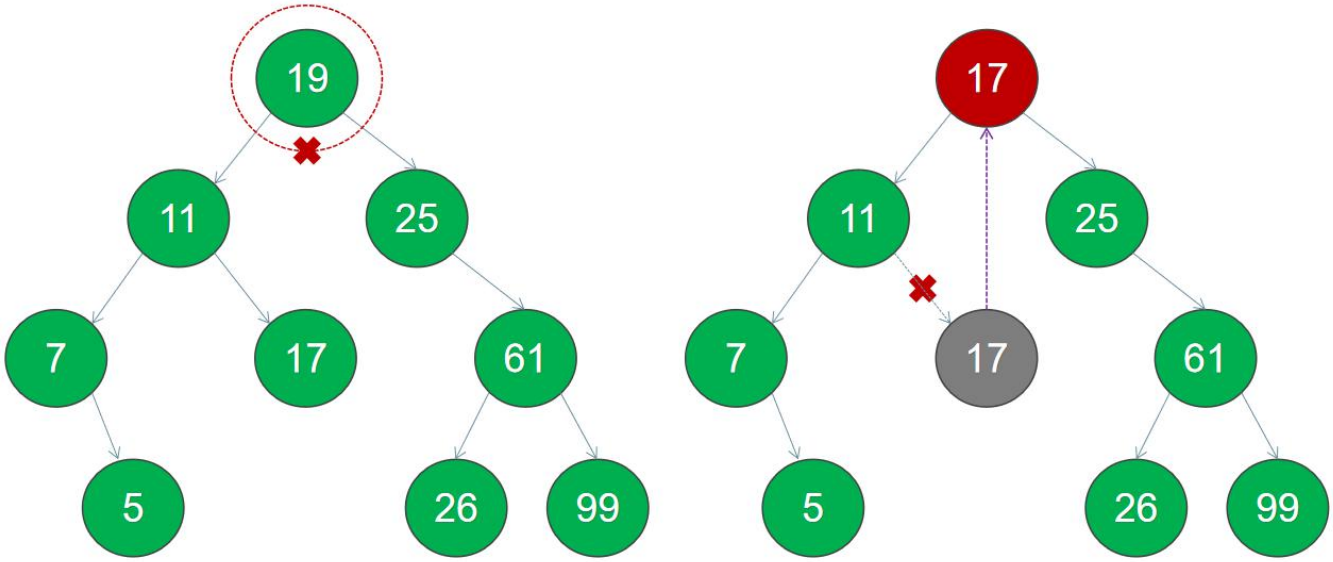
情形2 - 删除节点存在左子节点，不存在右子节点，直接把左子节点替代删除节点

✓ **删除节点存在右子节点，不存在左子节点，直接把右子节点替代删除节点**



情形3 - 删除节点存在右子节点，不存在左子节点，直接把右子节点替代删除节点

✓ **删除节点存在左右子节点，则取左子树上的最大节点或右子树上的最小节点替换删除节点。**



情形4 - 删除节点存在左右子节点，则取左子树上的最大节点或右子树上的最小节点替换删除节点。

```
/*************************
* 采用二叉搜索树上最大的结点
*************************/
int findMax(Btree* root)
{
    assert(root!=NULL);

    //方式一 采用递归
    /*if(root->rchild==NULL){
        return root->data;
    }
    return findMax(root->rchild);
    */

    //方式二 采用循环
    while(root->rchild){
        root = root->rchild;
    }

    return root->data;
}

/*************************
* 采用递归方式查找结点
*************************/
```

```
Btree* DeleteNode(Btree* root, int key) {
    if(root==NULL)return NULL;//没有找到删除节点

    if(root->data > key)
    {
        root->lchild = DeleteNode(root->lchild, key);
        return root;
    }
    if(root->data < key)
    {
        root->rchild = DeleteNode(root->rchild, key);
        return root;
    }

    //删除节点不存在左右子节点，即为叶子节点，直接删除
    if(root->lchild==NULL && root->rchild==NULL)return NULL;

    //删除节点只存在右子节点，直接用右子节点取代删除节点
    if(root->lchild==NULL && root->rchild!=NULL)return root->rchild;

    //删除节点只存在左子节点，直接用左子节点取代删除节点
    if(root->lchild!=NULL && root->rchild==NULL)return root->lchild;

    ////删除节点存在左右子节点，直接用左子节点最大值取代删除节点
    int val = findMax(root->lchild);
    root->data=val;
    root->lchild = DeleteNode(root->lchild,val);
    return root;
}
```

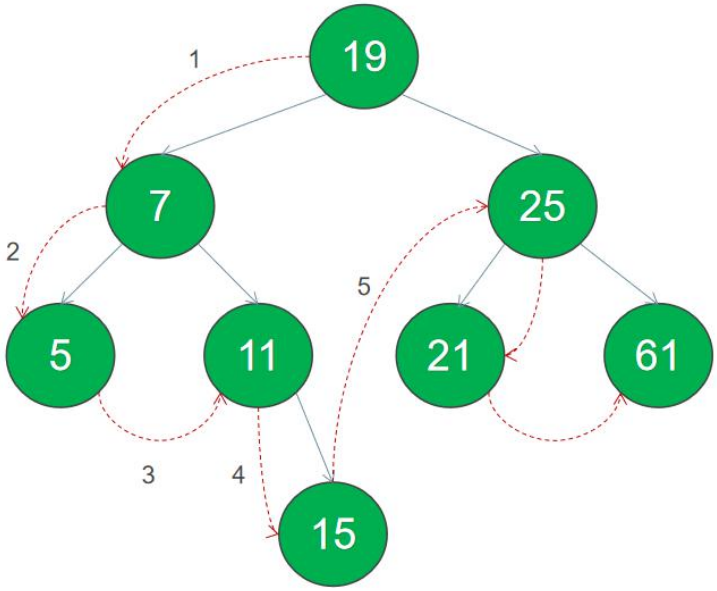
二叉搜索树搜索

```
/************************
* 采用递归方式查找结点
************************/
Bnode* queryByRec(Btree *root, ElemType e){
    if (root == NULL || isEqual(root->data, e)){
        return root;
    } else if(isLess(e, root->data)) {
        return queryByRec(root->lchild, e);
    } else {
        return queryByRec(root->rchild, e);
    }
}

/**
* 使用非递归方式查找结点
*/
Bnode* queryByLoop(Bnode *root, int e){
    while(root != NULL && !isEqual(root->data, e)){
        if(isLess(e, root->data)){
            root = root->lchild;
        }else{
            root = root->rchild;
        }
    }
    return root;
}
```

二叉树的遍历

二叉树的遍历是指从根结点出发，按照某种次序依次访问所有结点，使得每个结点被当且访问一次。共分为四种方式：

前序遍历 - 先访问根节点，然后前序遍历左子树，再前序遍历右子树



（ 前序遍历 ）

上图前序遍历结果：19 7 5 11 15 25 21 61

- 前序遍历 - 递归实现

```
/*************************
* 采用递归方式实现前序遍历
*************************/
void PreOrderRec(Btree *root)
 {
     if (root == NULL)
     {
         return;
     }
     printf("- %d -", root->data);
     preOrderRec(root->lchild);
     preOrderRec(root->rchild);
 }
```

- 前序遍历 - 非递归方式实现

具体过程：

首先申请一个新的栈，记为 stack；

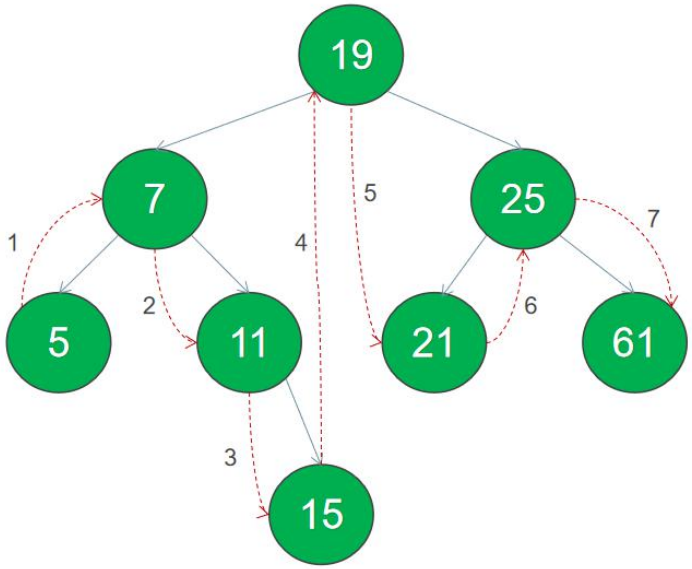
将头结点 head 压入 stack 中；

每次从 stack 中弹出栈顶节点，记为 cur，然后打印 cur 值，如果 cur 右孩子不为空，则将右孩子压入栈中；如果 cur 的左孩子不为空，将其压入 stack 中；

重复步骤 3，直到 stack 为空.

```
/************************
* 借助栈实现前序遍历
************************/
void PreOrder(Btree *root)
{
    Bnode cur ;
    if (root == NULL)
    {
        return;
    }
    SqStack  stack;
    InitStack(stack);
    PushStack(stack, *root);        //头节点先入栈
    while (!(IsEmpty(stack)))       //栈为空，所有节点均已处理
    {
        PopStack(stack, cur);       //要遍历的节点
        printf("- %d -", cur.data);

        if (cur.rchild != NULL)
        {
           PushStack(stack, *(cur.rchild));  //右子节点先入栈，后处理
        }
        if (cur.lchild != NULL)
        {
            PushStack(stack, *(cur.lchild)); //左子节点后入栈，接下来先处理
        }
    }
    DestroyStack(stack);
}
```

中序遍历 - 先访问根节点的左子树，然后访问根节点，最后遍历右子树

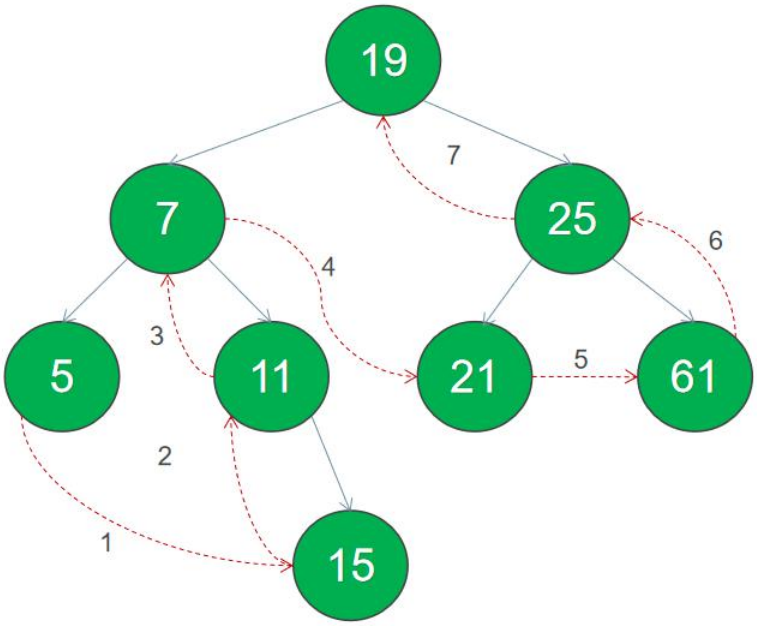


（ 中序遍历 ）

中序遍历结果：

```
5  7  11  15  19  21  25  61
```

后序遍历 - 从左到右，先叶子后节点的方式遍历访问左右子树，最后访问根节点

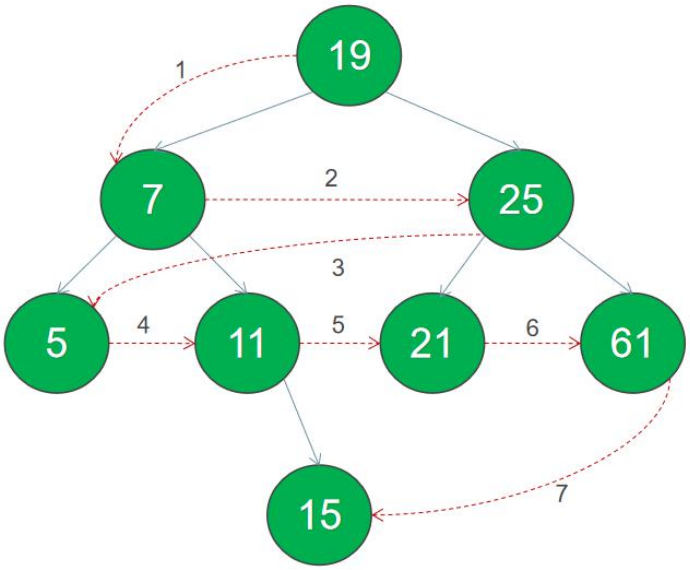


（ 后序遍历 ）

后序遍历结果：

```
5 15 11 7 21 61 25 19
```

层序遍历 - 从根节点从上往下逐层遍历，在同一层，按从左到右的顺序对节点逐个访问



（ 后序遍历 ）

上图层序遍历结果：

```
19 7 25 5 11 21 61 15
```

完整源码实现：

stack.h

```
#pragma once
#include<stdio.h>
#include<stdlib.h>
#include "tree.h"

#define MaxSize  128  //预先分配空间，这个数值根据实际需要预估确定

typedef struct _SqStack{
    Bnode *base;   //栈底指针
    Bnode *top;    //栈顶指针
}SqStack;


bool InitStack(SqStack &S) //构造一个空栈 S
{
    S.base = new Bnode[MaxSize];//为顺序栈分配一个最大容量为 Maxsize 的空间
    if (!S.base) //空间分配失败
        return false;
    S.top=S.base; //top 初始为 base，空栈
    return true;
}


bool PushStack(SqStack &S, Bnode e) // 插入元素 e 为新的栈顶元素

{
    if (S.top-S.base == MaxSize) //栈满
        return false;

    *(S.top++) = e; //元素 e 压入栈顶，然后栈顶指针加 1，等价于*S.top=e; S.top++;

    return true;
}


bool PopStack(SqStack &S, Bnode &e) //删除 S 的栈顶元素，暂存在变量 e 中

{
    if (S.base == S.top){ //栈空
        return false;
    }
```

```
    e = *(--S.top); //栈顶指针减 1，将栈顶元素赋给 e

    return true;
}


Bnode* GetTop(SqStack &S) //返回 S 的栈顶元素，栈顶指针不变
{
    if (S.top != S.base){ //栈非空
        return S.top - 1; //返回栈顶元素的值，栈顶指针不变
    }else{
       return NULL;
    }
}

int GetSize(SqStack &S){//返回栈中元素个数
    return (S.top-S.base);
}

bool IsEmpty(SqStack &S){//判断栈是否为空
    if (S.top == S.base){
        return true;
    }else{
        return false;
    }
}

void DestroyStack(SqStack &S){//销毁栈
    if(S.base){
        free(S.base);

       S.base = NULL;
       S.top = NULL;
    }
}
```

tree.h

```
#ifndef __TREE_H__
#define __TREE_H__

#define MAX_NODE 1024



#define isLess(a, b) (a<b)
#define isEqual(a, b) (a==b)

typedef  int ElemType;

typedef struct _Bnode{
    ElemType data;                  //元素类型
    struct _Bnode *lchild, *rchild;//指向左右孩子节点

}Bnode, Btree;

#endif
```

tree.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"
#include "stack.h"

bool InsertBtree(Btree **root, Bnode *node){
    Bnode *tmp = NULL;
    Bnode *parent = NULL;

    if(!node){
        return false;
    }else {//清空新节点的左右子树
        node->lchild = NULL;
        node->rchild = NULL;
    }

    if(*root){//存在根节点
        tmp= *root;
    }else{   //不存在根节点
        *root = node;
        return true;
    }
```

```
    while(tmp != NULL){
        parent = tmp;//保存父节点
        //printf("父节点：  %d\n", parent->data);
        if(isLess(node->data,tmp->data)){
            tmp = tmp->lchild;
        }else{
            tmp = tmp->rchild;
        }
    }

    //若该树为空树,则直接将 node 放置在根节点上
    if(isLess(node->data, parent->data)){//找到空位置后，进行插入
        parent->lchild = node;
    }else{
        parent->rchild = node;
    }

    return true;
}

/*************************
* 采用二叉搜索树上最大的结点
*************************/
int findMax(Btree* root)
{

    if(root->rchild==NULL){
        return root->data;
    }
    return findMax(root->rchild);
}

/*************************
* 采用递归方式查找结点
*************************/
Btree* DeleteNode(Btree* root, int key, Bnode* &deletedNode) {
    if(root==NULL)return NULL;

    if(root->data > key)
    {
        root->lchild = DeleteNode(root->lchild, key, deletedNode);
        return root;
    }
    if(root->data < key)
    {
        root->rchild = DeleteNode(root->rchild, key, deletedNode);
```

```
        return root;
    }

    deletedNode = root;

    //删除节点不存在左右子节点，即为叶子节点，直接删除
    if(root->lchild==NULL && root->rchild==NULL)return NULL;

    //删除节点存在右子节点，直接用右子节点取代删除节点
    if(root->lchild==NULL && root->rchild!=NULL)return root->rchild;

    //删除节点存在左子节点，直接用左子节点取代删除节点
    if(root->lchild!=NULL && root->rchild==NULL)return root->lchild;

    ////删除节点存在左右子节点，直接用左子节点最大值取代删除节点
    int val = findMax(root->lchild);
    root->data=val;
    root->lchild = DeleteNode(root->lchild,val, deletedNode);
    return root;
}

/************************
* 采用递归方式查找结点
************************/
Bnode* QueryByRec(Btree *root, ElemType e){
    if (isEqual(root->data, e) || NULL == root){
        return root;
    } else if(isLess(e, root->data)) {
        return QueryByRec(root->lchild, e);
    } else {
        return QueryByRec(root->rchild, e);
    }
}

/**
* 使用非递归方式查找结点
*/
Bnode* QueryByLoop(Bnode *root, int e){
    while(NULL != root && !isEqual(root->data, e)){
        if(isLess(e, root->data)){
            root = root->lchild;
        }else{
            root = root->rchild;
        }
    }
    return root;
}
```

```
/************************
* 采用递归方式实现前序遍历
************************/
void PreOrderRec(Btree *root)
{
    if (root == NULL)
    {
        return;
    }

    printf("- %d -", root->data);
    PreOrderRec(root->lchild);
    PreOrderRec(root->rchild);
}

/************************
* 借助栈实现前序遍历
************************/
void PreOrder(Btree *root)
{
    Bnode cur ;
    if (root == NULL)
    {
        return;
    }
    SqStack  stack;
    InitStack(stack);
    PushStack(stack, *root);        //头节点先入栈
    while (!(IsEmpty(stack)))       //栈为空，所有节点均已处理
    {
        PopStack(stack, cur);       //要遍历的节点
        printf("- %d -", cur.data);

        if (cur.rchild != NULL)
        {
            PushStack(stack, *(cur.rchild));  //右子节点先入栈，后处理
        }
        if (cur.lchild != NULL)
        {
            PushStack(stack, *(cur.lchild)); //左子节点后入栈，接下来先处
理
        }
    }
    DestroyStack(stack);
}
```

```
int main(void){
    int test[]={19, 7, 25, 5, 11, 15, 21, 61};
    Bnode * root=NULL, *node =NULL;

    node = new Bnode;
    node->data = test[0];
    InsertBtree(&root, node);//插入根节点

    for(int i=1;i<sizeof(test)/sizeof(test[0]);i++){
        node = new Bnode;
        node->data = test[i];
        if(InsertBtree(&root, node)){
            printf("节点 %d 插入成功\n", node->data);
        }else {

        }
    }

    printf("前序遍历结果： \n");
    PreOrderRec(root);
    printf("\n");
    system("pause");
    //二叉搜索树删除
    printf("删除节点 15\n");
    Bnode *deletedNode = NULL;
    root = DeleteNode(root, 15, deletedNode);
    printf("二叉搜索树删除节点 15, %s\n", deletedNode?"删除成功":"删除不
成功，节点不存在");
    if(deletedNode) delete deletedNode;


    printf("删除后，再次前序遍历结果： \n");
    PreOrderRec(root);
    printf("\n");

    //二叉搜索树查找节点
    Bnode * node1  = QueryByLoop(root, 20);
    printf("搜索二叉搜索树，节点 20 %s\n", node1?"存在":"不存在");


    Bnode * node2  = QueryByLoop(root, 21);
    printf("搜索二叉搜索树，节点 21 %s\n", node2?"存在":"不存在");

    system("pause");
    return 0;
}
```

QQ 交流群：875300321

第 4 节 树的企业级应用案例

4.1 哈夫曼编码

哈夫曼（Huffman）编码算法是基于二叉树构建编码压缩结构的，它是数据压缩中经典的一种算法。算法根据文本字符出现的频率，重新对字符进行编码。

首先请大家阅读下面两段中外小学作文：

中国 - 今天天气晴朗，我和小明出去玩！小明贪玩，不小心摔了一跤，小明被摔得哇哇哭了，小明的爸爸闻声赶来，又把小明痛扁了一阵。小明的小屁屁都被揍扁了，因为小明把妈妈刚买给他的裤子弄破了！

外国 - 今天天气晴朗，我和乔伊·亚历山大·比基·卡利斯勒·达夫·埃利奥特·福克斯·伊维鲁莫·马尔尼·梅尔斯·帕特森·汤普森·华莱士·普雷斯顿出去玩！乔伊·亚历山大·比基·卡利斯勒·达夫·埃利奥特·福克斯·伊维鲁莫·马尔尼·梅尔斯·帕特森·汤普森·华莱士·普雷斯顿贪玩，不小心摔了一跤，乔伊·亚历山大·比基·卡利斯勒·达夫·埃利奥特·福克斯·伊维鲁莫·马尔尼·梅尔斯·帕特森·汤普森·华莱士·普雷斯顿被摔得哇哇哭了，乔伊·亚历山大·比基·卡利斯勒·达夫·埃利奥特·福克斯·伊维鲁莫·马尔尼·梅尔斯·帕特森·汤普森·华莱士·普雷斯顿的爸爸闻声赶来，又把乔伊·亚历山大·比基·卡利斯勒·达夫·埃利奥特·福克斯·伊维鲁莫·马尔尼·梅尔斯·帕特森·汤普森·华莱士·普雷斯顿痛扁了一阵。乔伊·亚历山大·比基·卡利斯勒·达夫·埃利奥特·福克斯·伊维鲁莫·马尔尼·梅尔斯·帕特森·汤普森·华莱士·普雷斯顿的小屁屁都被揍扁了，因为乔伊·亚历山大·比基·卡利斯勒·达夫·埃利奥特·福克斯·伊维鲁莫·马尔尼·梅尔斯·帕特森·汤普森·华莱士·普雷斯顿把妈妈刚买给他的裤子弄破了！

同一段内容，当小明换成了外国小朋友的名字，篇幅就增加了几倍，有没有办法把内容缩减呢？当然有！在文章的开头，先声明一个缩写：

名字	缩写
乔伊·亚历山大·比基·卡利斯勒·达夫·埃利奥特·福克斯·伊维鲁莫·马尔尼·梅尔斯·帕特森·汤普森·华莱士·普雷斯顿	乔顿

那么，上面这段文字就可以缩成很小的一段：

今天天气晴朗，我和乔顿出去玩！乔顿贪玩，不小心摔了一跤，乔顿被摔得哇哇哭了，乔顿的爸爸闻声赶来，又把小明痛扁了一阵。乔顿的小屁屁都被揍扁了，因为乔顿把妈妈刚买给他的裤子弄破了！

哈夫曼编码就是这样一个原理！按照文本字符出现的频率，出现次数越多的字符用越简短的编码替代，因为为了缩短编码的长度，我们自然希望频率越高的词，编码越短，这样最终才能最大化压缩存储文本数据的空间。

哈夫曼编码举例： 假设要对“we will we will r u”进行压缩。

压缩前,使用 ASCII 码保存:

119	101	32	119	105	108	108	32	119	101	32	119	105	108	108	32	114	32	117

共需： 19 个字节 - 152 个位保存

下面我们先来统计这句话中每个字符出现的频率。如下表，按频率高低已排序：

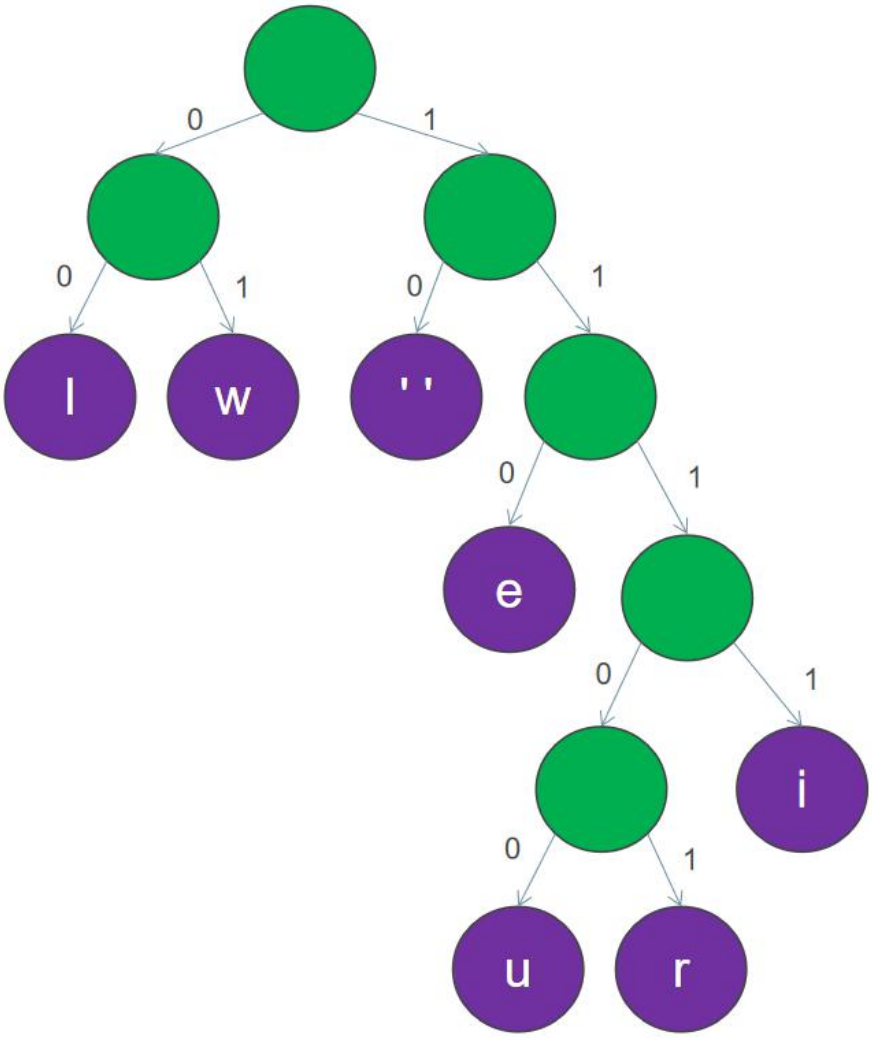
字符	空格	w	l	e	i	r	u
频率	5	4	4	2	2	1	1

接下来，我们按照字符出现的频率，制定如下的编码表：

字符	l	w	空格	e	i	u	r
编码	00	01	10	110	1111	11100	11101

这样，“we will we will r u”就可以按如下的位来保存：

01 110 10 01 1111 00 00 10 01 110 10 01 1111 00 00 10 11101 10 11100



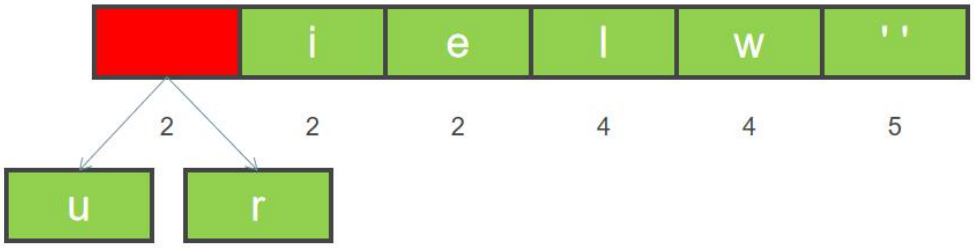
（哈夫曼编码）

哈夫曼二叉树构建

1. 按出现频率高低将其放入一个数组中，从左到右依次为频率逐渐增加

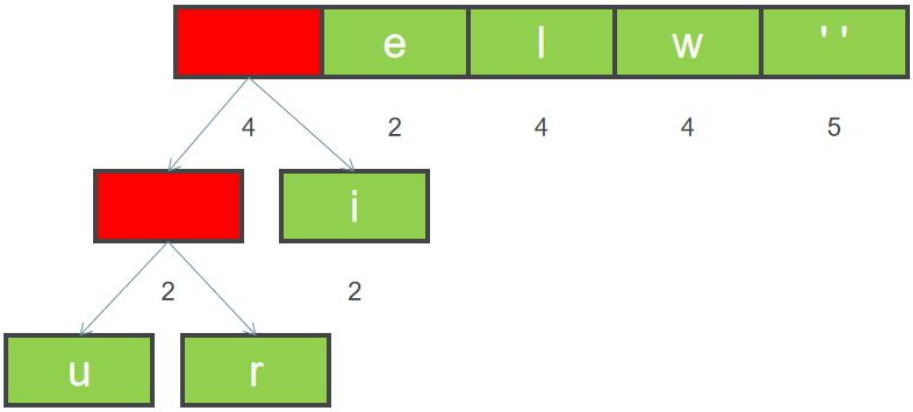
字符	u	r	i	e	l	w	''
频率	1	1	2	2	4	4	5

2. 从左到右进行合并，依次构建二叉树。第一步取前两个字符 u 和 r 来构造初始二叉树，第一个字符作为左节点，第二个元素作为右节点，然后两个元素相加作为新的空元素，并且两者权重相加作为新元素的权重。

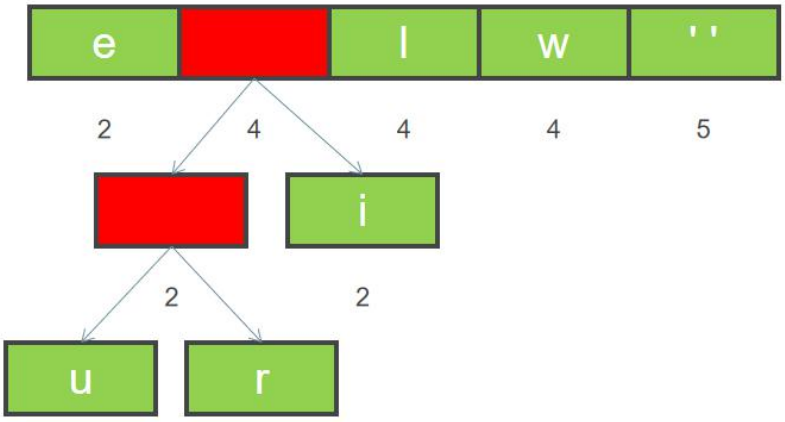


3. 新节点加入后，依据权重重新排序，按照权重从小到大排列，上图已有序。

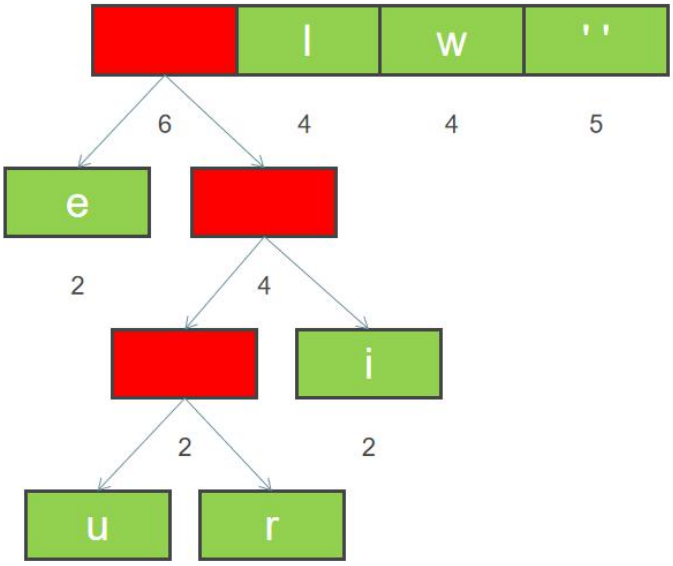
4. 红色区域的新增元素可以继续和 i 合并,如下图所示：



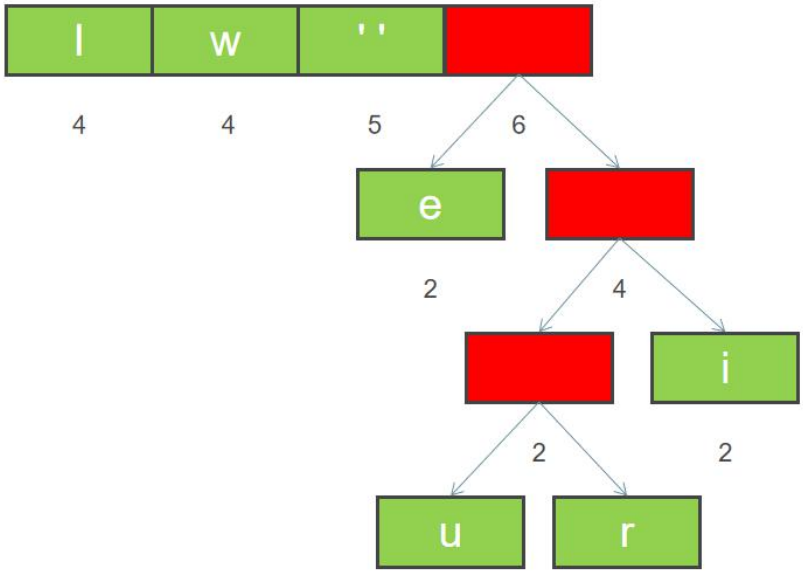
5. 合并节点后，按照权重从小到大排列，如下图所示。



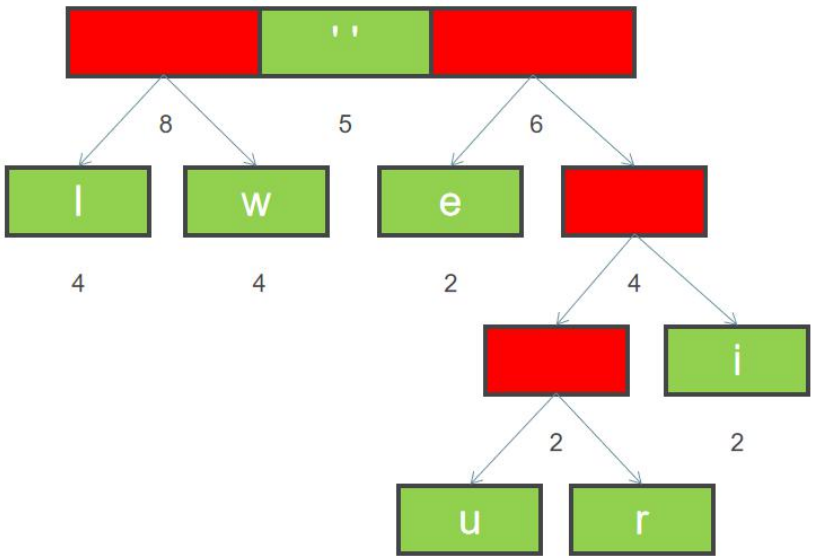
6. 排序后，继续合并最左边两个节点，构建二叉树，并且重新计算新节点的权重



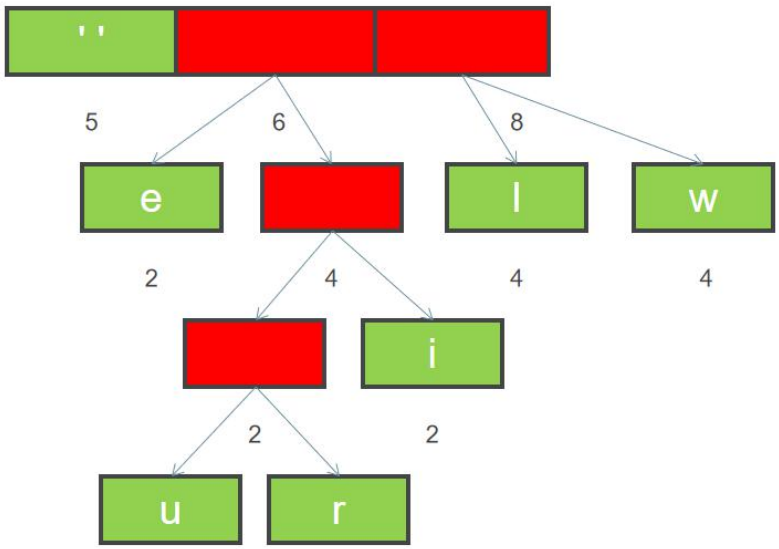
7. 重新排序



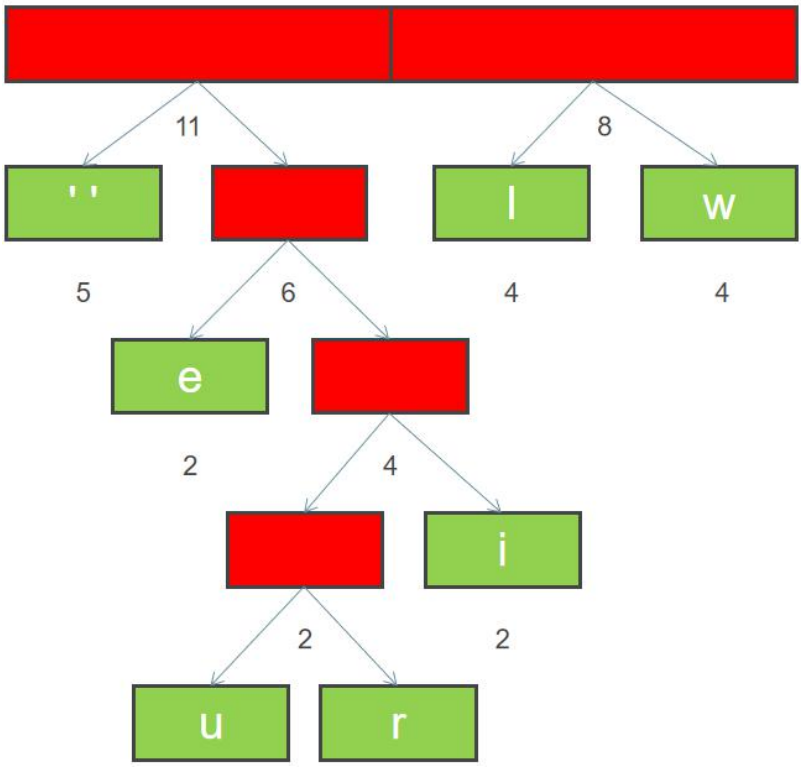
8. 重复上面步骤 6 和 7,直到所有字符都变成二叉树的叶子节点



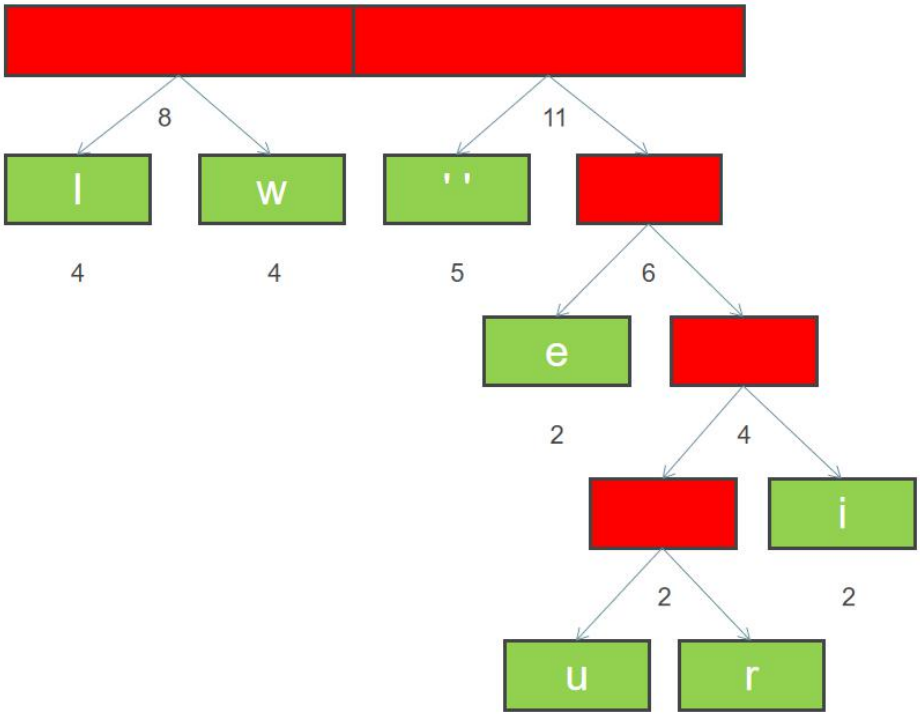
(左边两节点合并)



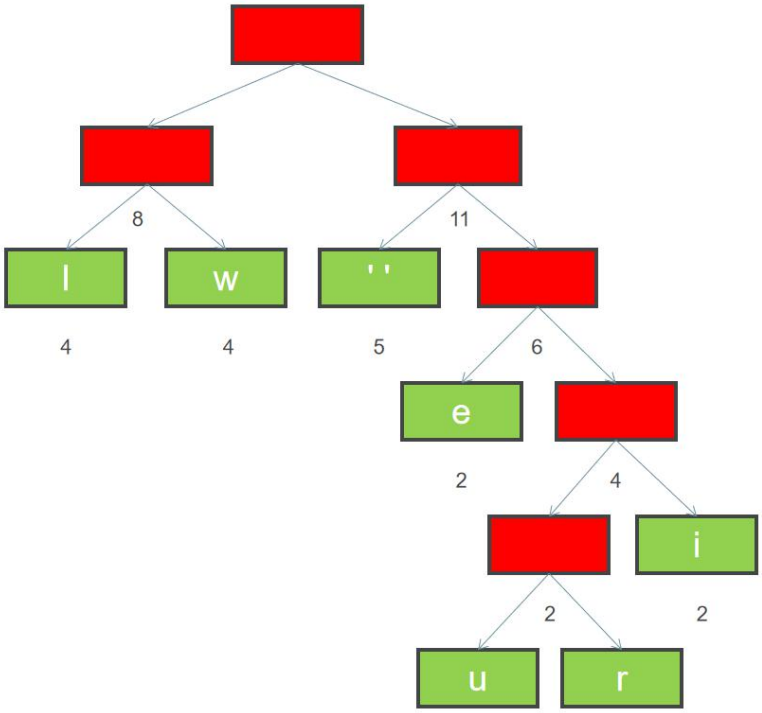
(重新排序)



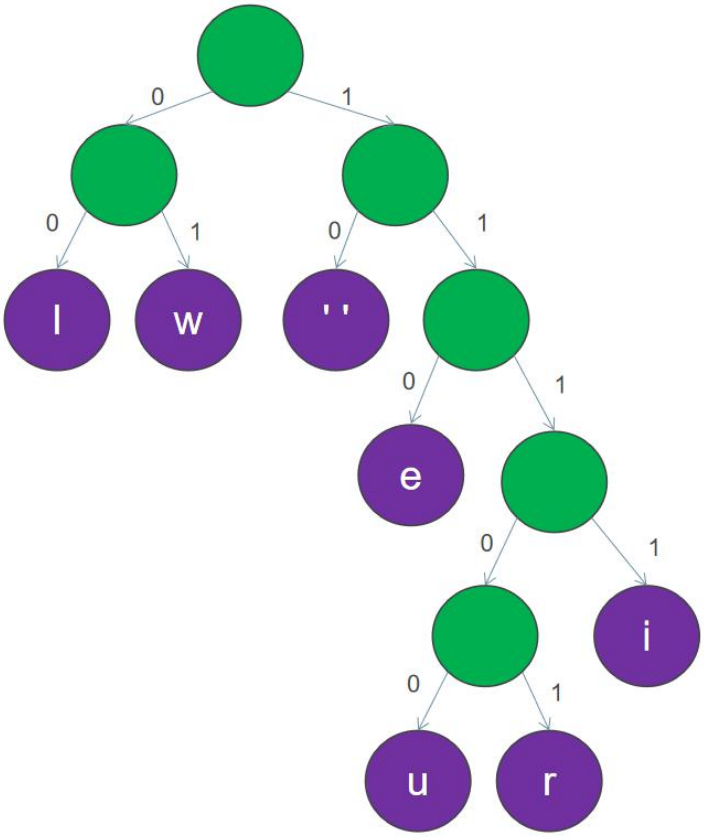
（左边两节点合并）



（重新排序）



（最后一次合并，生成哈夫曼编码树）



（哈夫曼编码）

编码实现

Huff.h

```
#pragma once

#include <stdio.h>
#include <assert.h>
#include <Windows.h>
#include <iostream>
#include <iomanip>

using namespace std;

#define MaxSize 1024        //队列的最大容量

typedef struct _Bnode
{
    char value;
    int weight;
    struct _Bnode *parent;
    struct _Bnode *lchild;
    struct _Bnode *rchild;

} Btree, Bnode;         /* 结点结构体 */


typedef Bnode *DataType;    //任务队列中元素类型

typedef struct _QNode {  //结点结构
    int priority; //每个节点的优先级,数值越大，优先级越高，优先级相同，
取第一个节点
    DataType data;
    struct _QNode *next;
}QNode;

typedef QNode * QueuePtr;

typedef struct Queue
{
    int      length;  //队列的长度
    QueuePtr front;   //队头指针
    QueuePtr rear;    //队尾指针
}LinkQueue;
```

```
//队列初始化，将队列初始化为空队列
void InitQueue(LinkQueue *LQ)
{
    if(!LQ) return ;

    LQ->length = 0;
    LQ->front = LQ->rear = NULL;  //把对头和队尾指针同时置 0
 }

//判断队列为空
int IsEmpty(LinkQueue *LQ)
{
    if(!LQ) return 0;

    if (LQ->front == NULL)
    {
        return 1;
    }
    return 0;
}


//判断队列是否为满
int IsFull(LinkQueue *LQ)
{
    if(!LQ) return 0;

    if (LQ->length == MaxSize)
    {
        return 1;
    }
    return 0;
}

//入队,将元素 data 插入到队列 LQ 中
int EnterQueue( LinkQueue *LQ,DataType data, int priority){
    if(!LQ) return 0;

    if(IsFull(LQ)){
        cout<<"无法插入元素 "<<data<<", 队列已满!"<<endl;
        return 0;
    }

    QNode *qNode = new QNode;
    qNode->data = data;
    qNode->priority = priority;
    qNode->next = NULL;
```

```
    if(IsEmpty(LQ)){//空队列
        LQ->front = LQ->rear = qNode;
    }else {
        qNode->next = LQ->front;
        LQ->front = qNode;
        //LQ->rear->next =qNode;//在队尾插入节点 qNode
        //LQ->rear = qNode;     //队尾指向新插入的节点
    }
    LQ->length++;

    return 1;
}


//出队，遍历队列，找到队列中优先级最高的元素 data 出队
int PopQueue(LinkQueue *LQ, DataType *data){
    QNode **prev = NULL, *prev_node=NULL;//保存当前已选举的最高优先级节
点上一个节点的指针地址。
    QNode *last = NULL, *tmp = NULL;

    if(!LQ || IsEmpty(LQ)){
        cout<<"队列为空！"<<endl;
         return 0;
    }

    if(!data) return 0;
    //prev  指向队头 front 指针的地址
    prev = &(LQ->front);
    //printf("第一个节点的优先级: %d\n", (*prev)->priority);
    last = LQ->front;
    tmp = last->next;
    while(tmp){
        if(tmp->priority <(*prev)->priority){
            //printf("抓到个更小优先级的节点[priority: %d]\n",
tmp->priority);
            prev = &(last->next);
            prev_node= last;

        }
        last=tmp;
        tmp=tmp->next;
    }

    *data = (*prev)->data;
    tmp = *prev;
    *prev = (*prev)->next;
```

```
    delete tmp;

    LQ->length--;

    //接下来存在 2 种情况需要分别对待
    //1.删除的是首节点,而且队列长度为零
    if(LQ->length==0){
        LQ->rear=NULL;
    }

    //2.删除的是尾部节点
    if(prev_node&&prev_node->next==NULL){
        LQ->rear=prev_node;
    }

    return 1;
}

//打印队列中的各元素
void PrintQueue(LinkQueue *LQ)
{   QueuePtr tmp;

    if(!LQ) return ;

    if(LQ->front==NULL){
        cout<<"队列为空！";
        return ;
    }

    tmp = LQ->front;
    while(tmp)
    {
        cout<<setw(4)<<tmp->data->value<<"["<<tmp->priority<<"]";
         tmp = tmp->next;
    }
    cout<<endl;
}

//获取队首元素，不出队
int GetHead(LinkQueue *LQ,DataType *data)
{
    if (!LQ || IsEmpty(LQ))
    {
         cout<<"队列为空!"<<endl;
        return 0;
    }
```

```
    if(!data) return 0;

    *data = LQ->front->data;
    return 1;
}

//清空队列
void ClearQueue(LinkQueue *LQ)
{
    if(!LQ) return ;

    while(LQ->front){
        QueuePtr tmp = LQ->front->next;
        delete LQ->front;
        LQ->front = tmp;
    }

    LQ->front = LQ->rear = NULL;
    LQ->length = 0;
}

//获取队列中元素的个数
int getLength(LinkQueue* LQ){
    if(!LQ) return 0;

    return LQ->length;
}
```

哈夫曼编码实现.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include "Huff.h"

using namespace std;

void PreOrderRec(Btree *root);

/* 构造哈夫曼编码树 */
void HuffmanTree (Btree * &huff,  int n)
{
    LinkQueue *LQ = new LinkQueue;
    int i = 0;
```

```
    //初始化队列
    InitQueue(LQ);


    /* 初始化存放哈夫曼树数组 HuffNode[] 中的结点 */
    for (i=0; i<n; i++)
    {
        Bnode * node = new Bnode;

        cout<<"请输入第"<<i+1<<"个字符和出现频率："<<endl;
        cin>>node->value;  //输入字符
         cin>>node->weight ;//输入权值
         node->parent =NULL;
         node->lchild =NULL;
         node->rchild =NULL;

        EnterQueue(LQ, node, node->weight);
    }

    PrintQueue(LQ);
    //system("pause");
    //exit(0);

    do{
        Bnode *node1 = NULL;
        Bnode *node2 = NULL;

        if(!IsEmpty(LQ)){
            PopQueue(LQ, &node1);
            printf("第一个出队列的数：%c, 优先级：%d\n", node1->value,
node1->weight);
        }else {
            break;
        }

        if(!IsEmpty(LQ)){
            Bnode *node3 = new Bnode;

            PopQueue(LQ, &node2);
            printf("第二个出队列的数：%c, 优先级：%d\n", node2->value,
node2->weight);
            node3->lchild = node1;
            node1->parent = node3;
            node3->rchild = node2;
            node2->parent = node3;

            node3->value = ' ';
```

```
            node3->weight=node1->weight+node2->weight;
            printf("合并进队列的数：%c, 优先级：%d\n", node3->value,
node3->weight);
            EnterQueue(LQ,node3, node3->weight);
        }else {
            huff = node1;
            break;
        }

    }while(1);
}


/************************
* 采用递归方式实现前序遍历
************************/
void PreOrderRec(Btree *root)
{
    if (root == NULL)
    {
        return;
    }

    printf("- %c -", root->value);
    PreOrderRec(root->lchild);
    PreOrderRec(root->rchild);
}

int main(void){
    Btree * tree = NULL;

    HuffmanTree(tree,  7);
    PreOrderRec(tree);
    system("pause");
    return 0;
}
```

4.2 红黑树

定义 — 是每个节点都带有颜色属性（颜色为红色或黑色）的自平衡二叉查找树，满足下列性质：

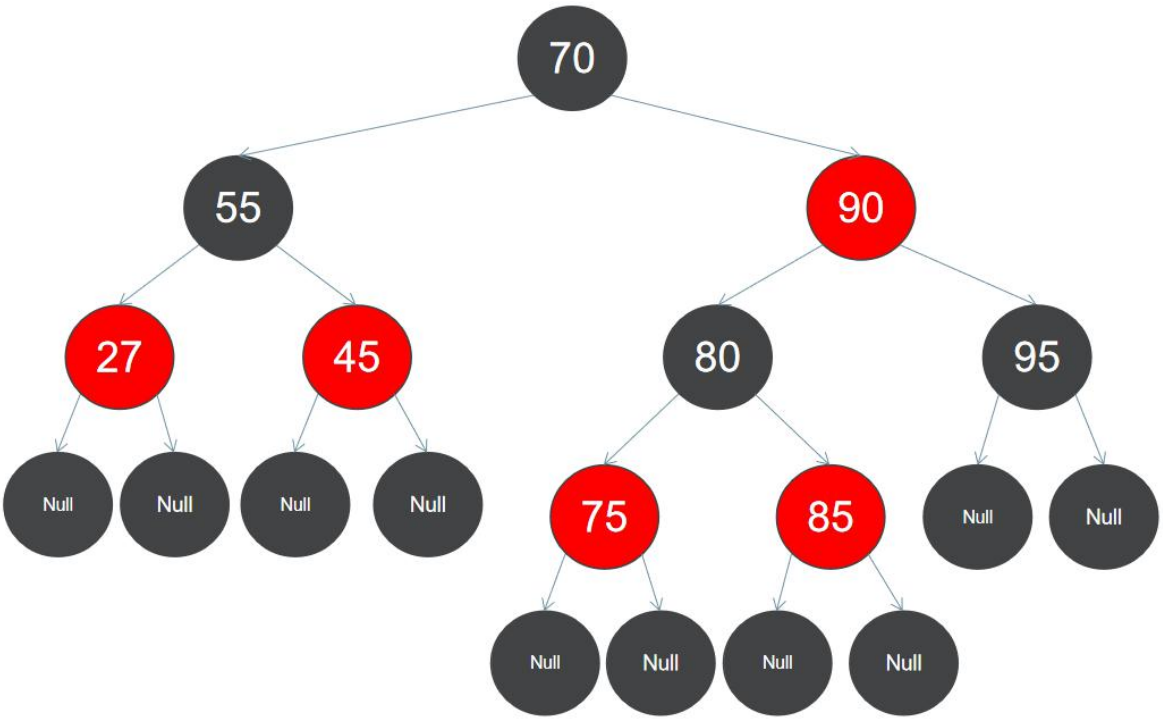
1）节点是红色或黑色；

2）根节点是黑色；

3）所有叶子节点都是黑色节点(NULL)；

4）每个红色节点必须有两个黑色的子节点。(从每个叶子到根的所有路径上不能有两个连续的红色节点。)

5）从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

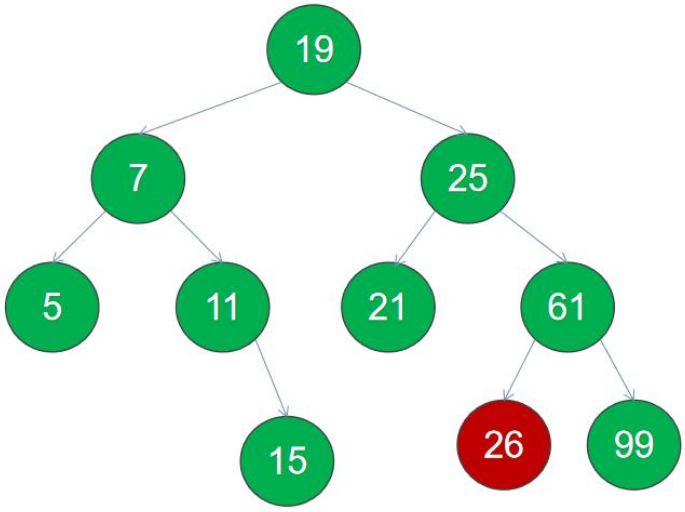


（红黑树）

为什么有了二叉搜索树，还需要红黑树？

下面的二叉树可以达到很好的搜索效果

5	7	11	15	19	21	25	26	61	99

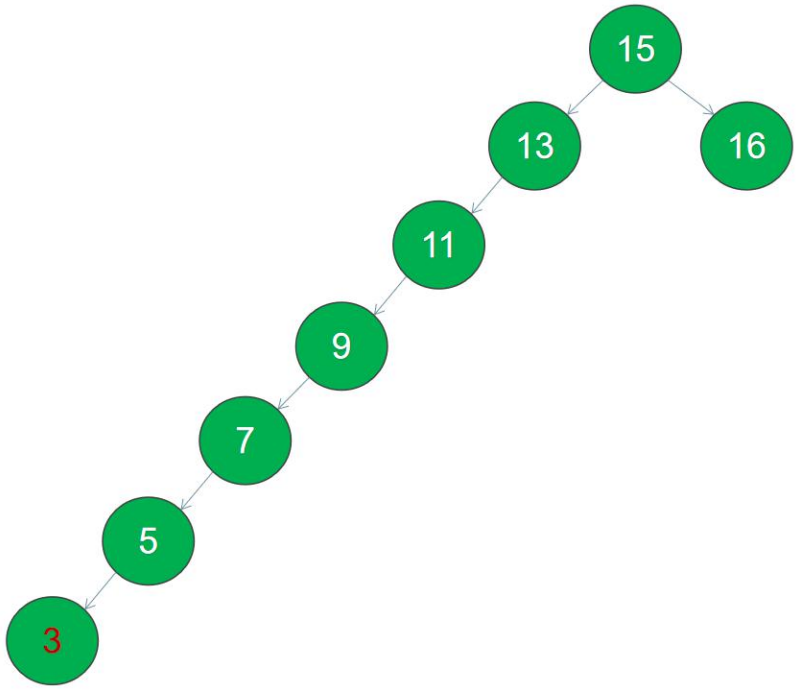


（二叉排序树）

我们再看看下面这棵树，我们将下面的数据按从左至右的顺序构造一棵二叉搜索树

15	13	16	11	9	7	5	3

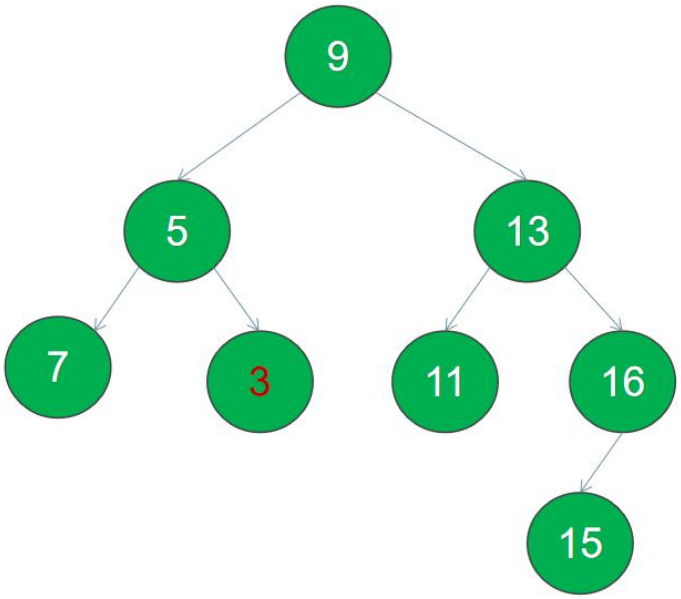
按照之前我们二叉搜索树构建构建的方式，我们将得到下面这样一棵树：



如果我们查找值为 3 的节点，9 个节点需要比较的次数是 7 次

同样的数据，如果我们按照以下顺序构造一棵二叉排序树：

9	5	13	7	3	11	16	15

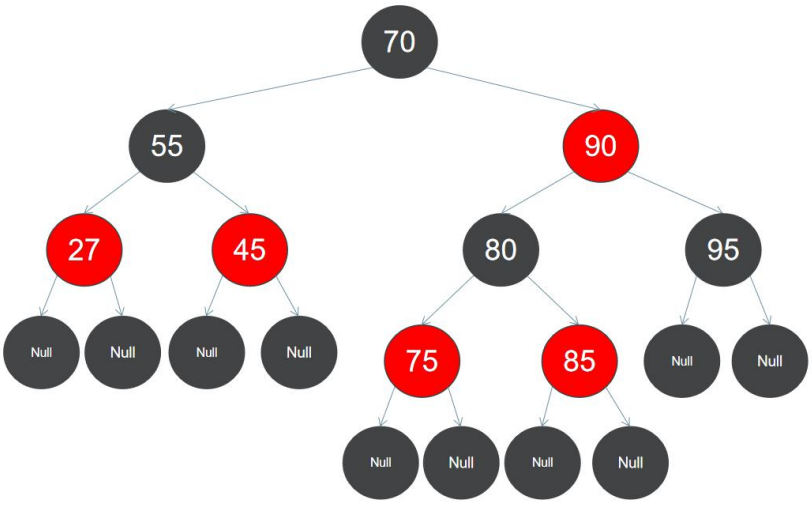


则查找值为 3 的节点，9 个节点需要比较的次数是 3 次.

为什么两者达到了如此之大的差距，原因是第一棵树左右不够平衡，导致出现比较极端的情况。

解决方案： 红黑树

红黑树是一种自平衡二叉查找树，从上面红黑树的图可以看到，根结点右子树显然比左子树高，但左子树和右子树的黑结点的层数是相等的，也即任意一个结点到到每个叶子结点的路径都包含数量相同的黑结点。所以我们叫红黑树这种平衡为黑色完美平衡。



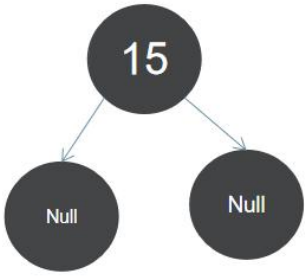
（红黑树）

红黑树构建

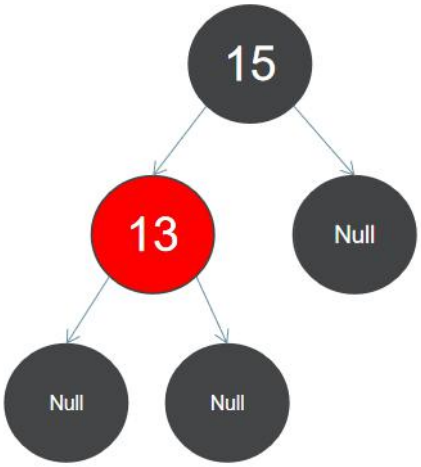
以上例数组为例构建红黑树：

15	13	16	11	9	7	5	3

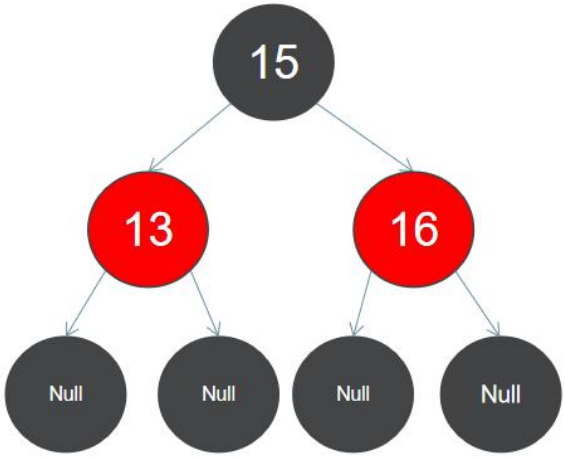
第一步：使用第一个元素 15 创建根节点，根节点一定是黑色，如图:



第二步：将 13 加入到红黑树，按照二叉搜索树的规则,13 应插入到 15 的左子节点上面，此时插入红色节点不会破坏红黑树的平衡，直接插入即可,如图:

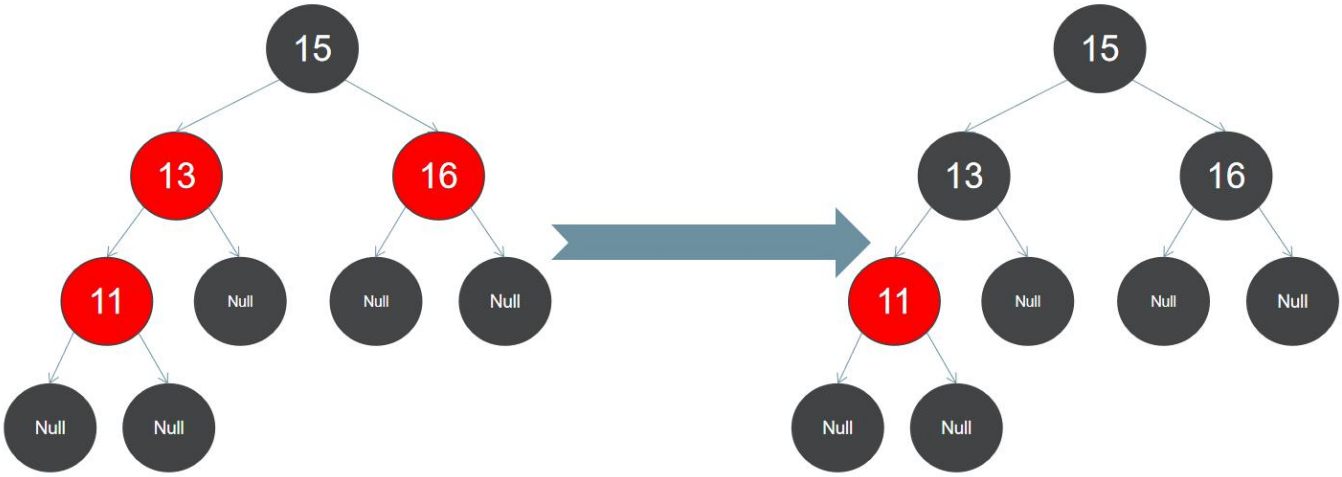


第三步：将 16 加入到红黑树，16 应插入到 15 的右子节点上面，此时插入红色节点不会破坏红黑树的平衡，直接插入即可，如图所示:

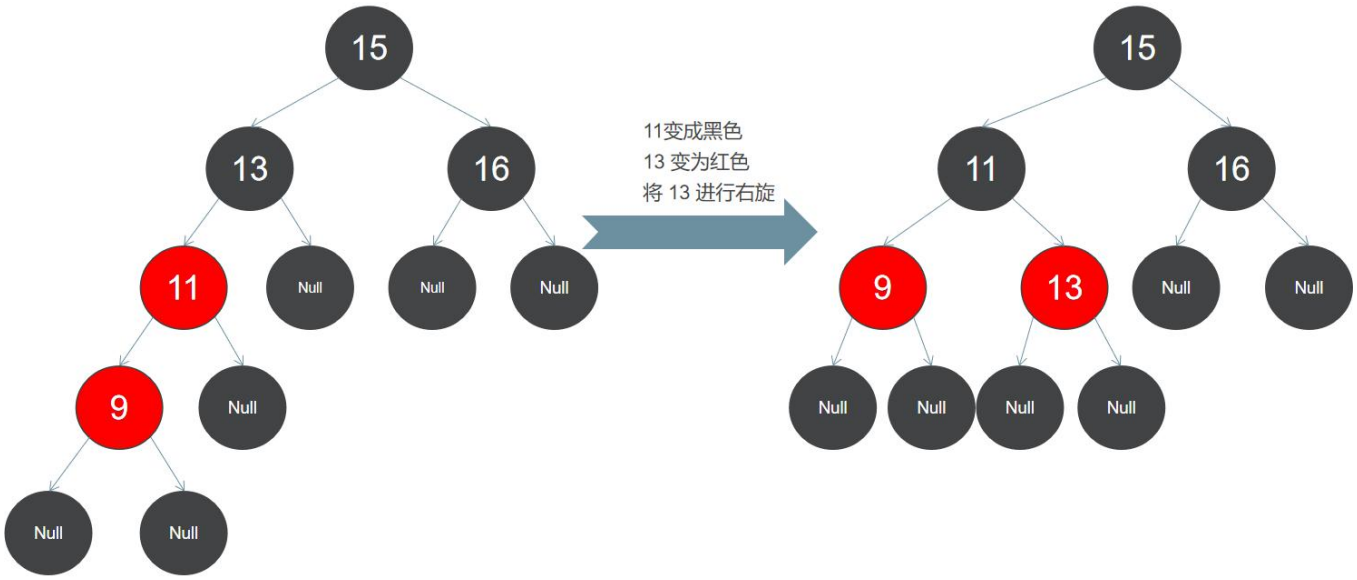


第四步：将 11 加入到红黑树，11 应插入到 13 的左子节点上面，此时继续插入红色节点会破坏红黑树的平衡规则，红色节点下必须是一对黑色子节点，而插入黑色节点也违背了规则 5（从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点）,所以必须进行如下特殊处理:

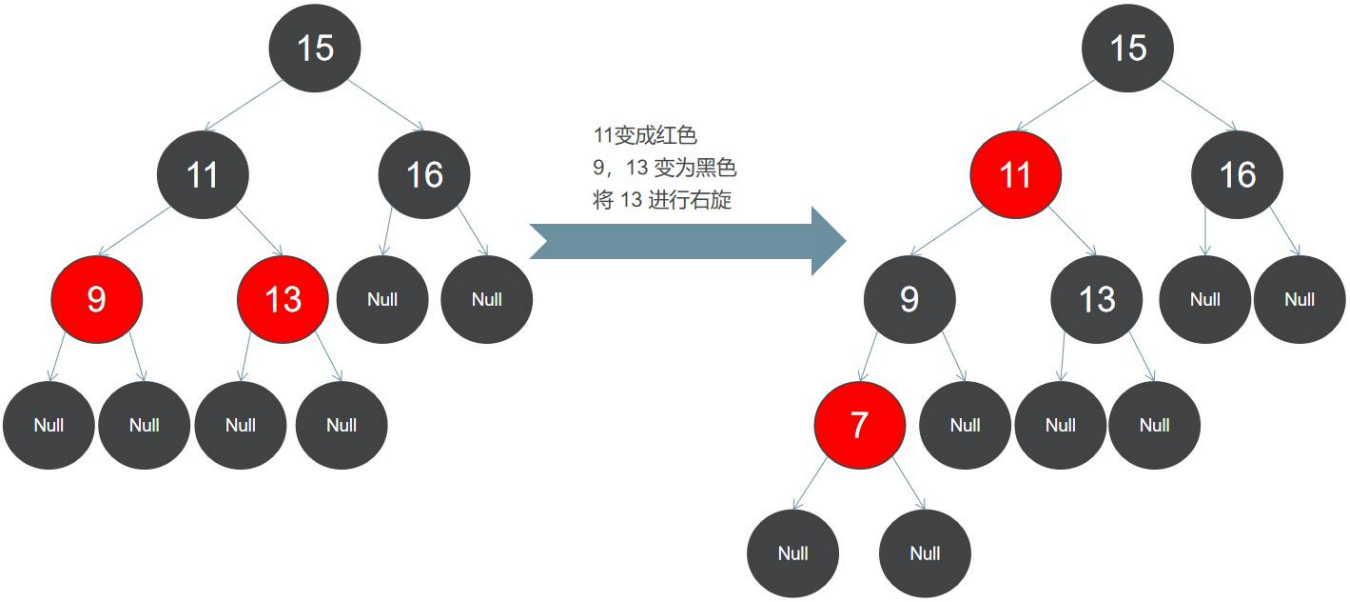
1. 将 11 的父节点 13 和叔父节点都设置为黑色



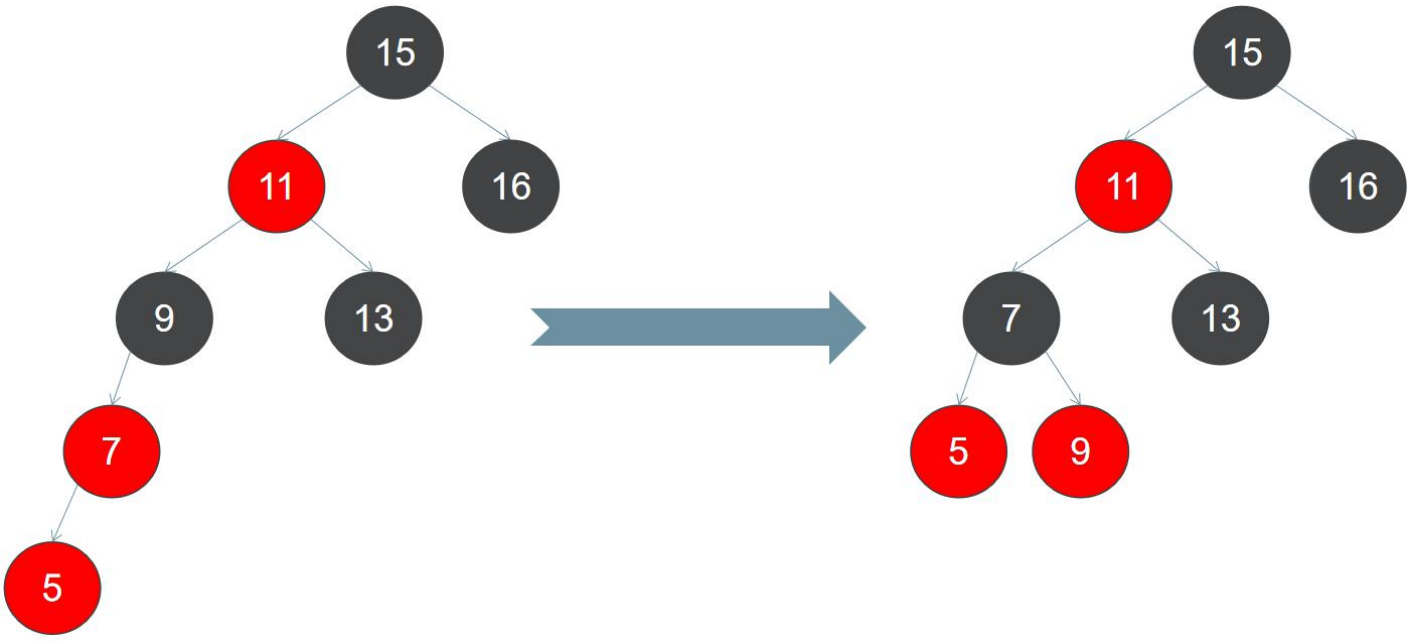
第五步 将 9 加入到红黑树，9 应插入到 11 的左子节点上面，此时无论插入红色节点或黑色节点都会破坏规则，必须对节点做乾坤大挪移处理：



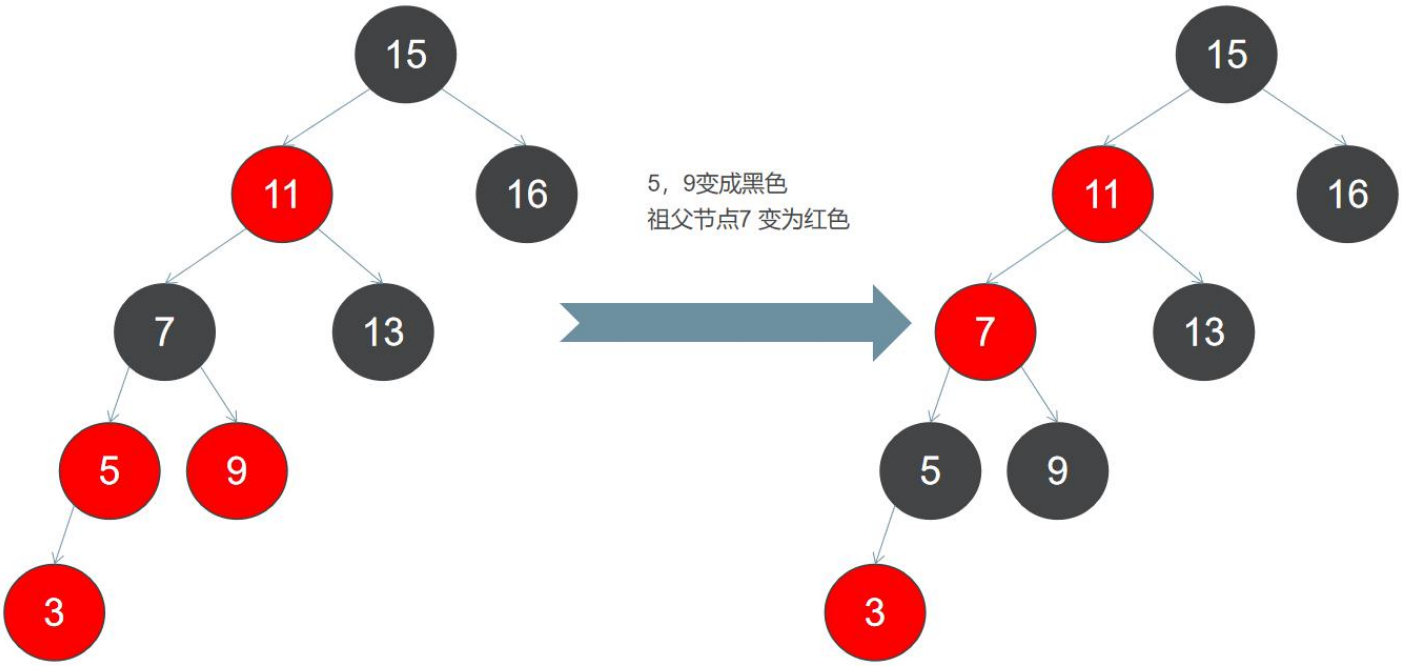
第五步 将 7 加入到红黑树，7 应插入到 9 的左子节点上面，此时无论插入红色节点或黑色节点都会破坏规则，必须对节点做变色处理：



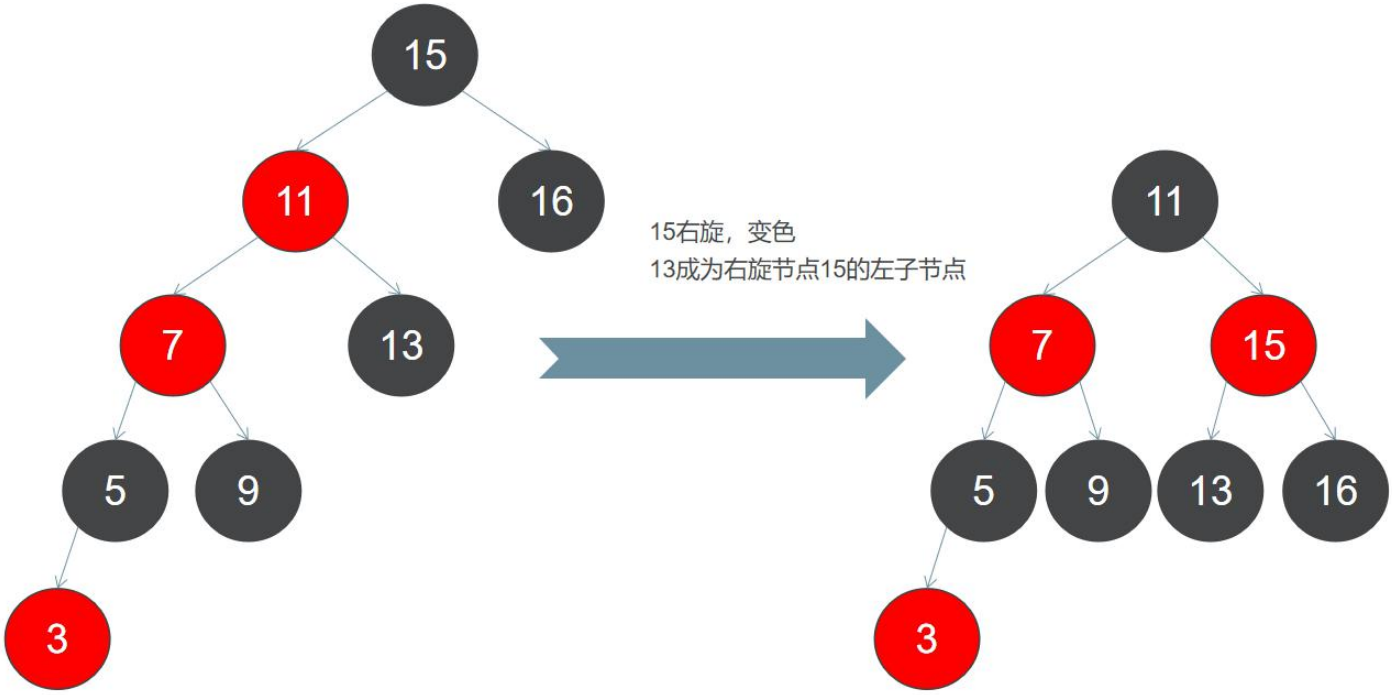
第六步 将 5 加入到红黑树，5 应插入到 7 的左子节点上面，此时无论插入红色节点或黑色节点都会破坏规则，必须对节点做旋转处理：



第七步 将 3 加入到红黑树，3 应插入到 5 的左子节点上面，此时无论插入红色节点或黑色节点都会破坏规则，我们先对父节点和叔叔节点做变色处理：



但这样破坏了红色节点 11 必须有两个黑色子节点的规则，因此还得进一步向上调整，此时，要保障红黑树的平衡，我们需要对根节点进行右旋！



红黑树查找

红黑树同二叉搜索树查找

QQ 交流群：875300321