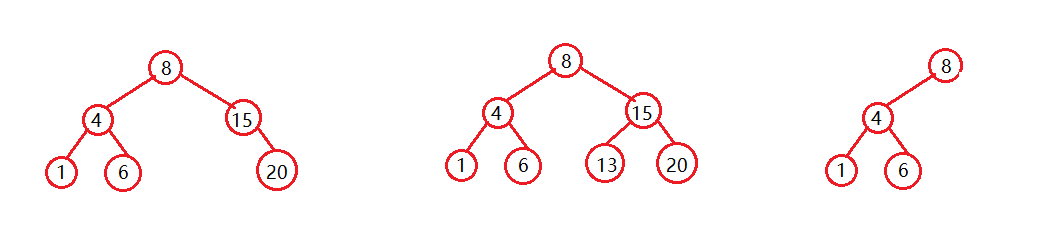
树属于一对多关系

如下的数据结构，我们称为二叉树



二叉树一个节点可以连接到左右2个节点

**二叉树**

**二叉树的遍历**

有时候，我们需要遍历二叉树的每一个元素，对这些元素进行操作，遍历的算法依据中间节点的访问顺序分为

前序遍历：访问中节点，访问左节点，访问右节点，简称VLR

中序遍历：访问左，访问中，访问右，简称LVR

后序遍历：访问左，访问右，访问中，简称LRV

如下是C++的一个前序遍历方法

class BinaryTreeNode

{

public:

    ...

    BinaryTreeNode \*left;

    BinaryTreeNode \*right;

    // 前序遍历

    void PerOrder(void Visit(BinaryTreeNode \*))

    {

        // 访问当前节点

        Visit(this);

        if (this->left)

        {

            // 访问左节点

            this->left->PerOrder(Visit);

        }

        if (this->right)

        {

            // 访问右节点

            this->right->PerOrder(Visit);

        }

    }

};

**Huffman（哈夫曼）树**

Huffman树一般应用在编码领域

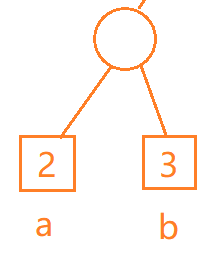
试想，一段字符串中a、b、c、d、e出现的次数为2 3 3 4 5，那么我们如可编码abcde字符才能使这段字符占用空间最小呢？

方法1：我们用 000，001，010，011，100代表a，b，c，d，e，那么我们需要51b的空间

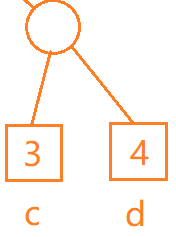
方法2：我们用 000，001，10，11，01 代表 a，b，c，d，e 那么我们需要 39b 的空间，上面的编码是由Huffman树确定的

如何构建Huffman树，以上面的例子为例，我们把字符出现的次数称为权

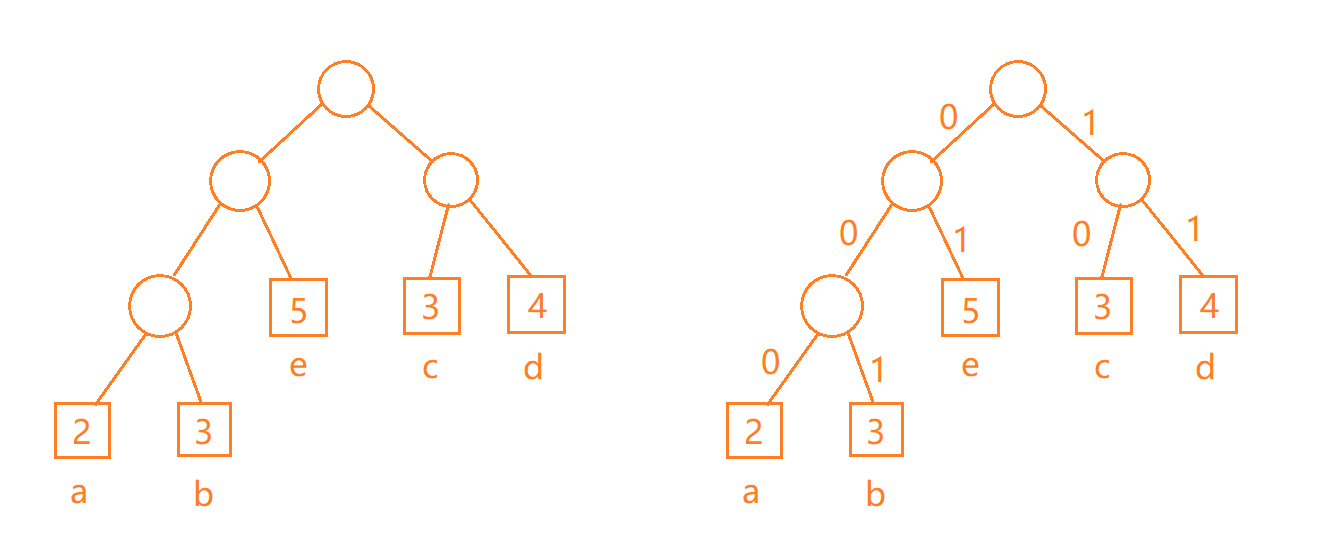
1. 取权最小的两个权构建二叉树，那么生成的这棵树的权为5



1. 然后再从这4个权中取最小的2个权构建二叉树，如下生成的树的权为7



1. 重复第2步，最终构建的树如下



我们对左右进行编码，最终确定了abcde的编码

**集合**

**集合的定义**

集合代表的是一组元素，如下

A = { 1, 3, 5 }，B = { 2, 7, 9, 10 }，C={ }

集合A有1, 3, 5元素

集合B有2, 7, 9, 10元素

集合C为空集

集合中不允许重复值，如 A = { 1, 2, 2, 3 } 不是集合

**子集**

如集合 A = { 1, 2, 4, 6, 8 }，集合B = { 2, 4, 6 }，我们成B为A的子集，记为

A ⊆ B

**集合的一般操作**

并集：将2个集合合并，去除重复部分

交集：取2个集合的相同部分

差集：A - B，为集合A中移除集合B所拥有的元素

插入：插入元素

删除：删除元素

**位向量实现集合**

如果一个集合中的元素只能是0到31，那么我们可以是由32位的int去代表这个集合，如10110000 00000000 00000000 00000000 代表的集合为 { 0, 3, 4 }，如果我们向集合添加31元素，则int变为10110000 00000000 00000000 00000001

**MFSet（并查集）**

如果元素ABCDEF存在关系（具体什么关系不要管），A和B有关系，B和C有关系，D和E F有关系，我随意输入2个元素，要如何知道这2个元素是否存在直接或间接关系呢？

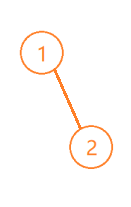
我们根据如下步骤对这些元素构建并查集

1. 我们是由123456代表ABCDEF这6个元素，并将这6个元素分别作为各自的集合，使用树代表集合，于是我们得到6棵树

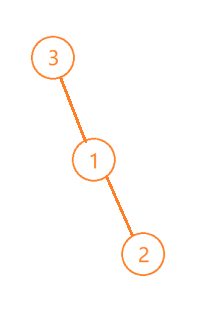


1. 我们根据元素之间的关系构建树

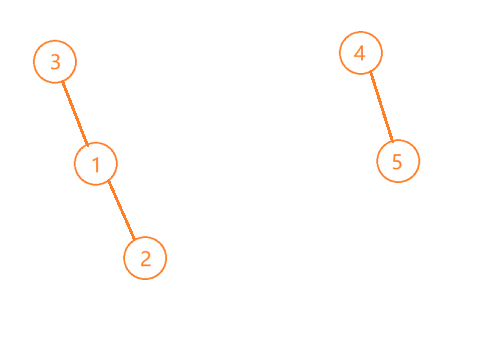
A和B有关系，所以我们构建



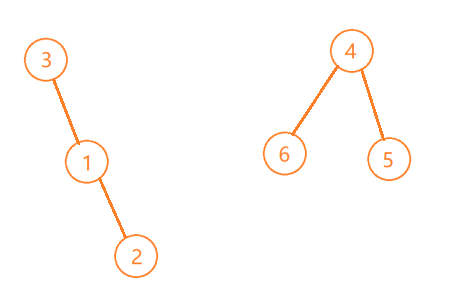
B和C有关系，所以我们找到2和3所在的树，2所在的树根为1，3所在的树根就是3，所以将1和3合并，我们得到如下树



D和E有关系，所以我们合并这2棵树



E和F有关系，所以我们合并这2棵树，最后我们得到得树如下



1. 查找2个元素之间是否存在关系

如查找A和E是否存在关系，我们只需要找到这2个元素得根，判断根是否相同即可

**并查集得实现**

上面介绍得并查集中，每个元素需要保存2个信息，一个是元素得编号，另一个是元素得父元素，如：元素2得父元素是1，元素1得父元素是3

我们只需要一个int数组就可以保存上面得并查集得元素信息了，如上面示例得并查集信息得数组为

var list = [

    0,

    3,

    1,

    3,

    4,

    4,

    4

]

我们将数组的下标对应到元素的编号（我们没有编号为0的元素，所以数组下标0弃用，当然，你应该从0开始对元素进行编号，这里为了方便理解就从1开始编号）

数组下标1为的位置值为3，这代表1元素的父元素是3

数组下标2为的位置值为1，这代表2元素的父元素是1

数组下标3为的位置值为3，由于3没有父元素，所以它的值就是她本身

并查集初始化

// 初始化，各个元素的为一颗树，其根就是元素本身，所以

function init(){

    list = [0, 1, 2, 3, 4, 5, 6]

}

合并树（合并集合）

// 合并 x 和 y 所在的树

function megre(x, y){

    var xRoot = x;

    while(list[xRoot] != xRoot){

        xRoot = list[xRoot];

    }

    var yRoot = y;

    while(list[yRoot] != yRoot){

        yRoot = list[yRoot];

}

    list[yRoot] = xRoot;

}

查找元素的根

function find(ele){

    var curele = ele;

    while(curele != list[curele]){

        curele = list[curele]

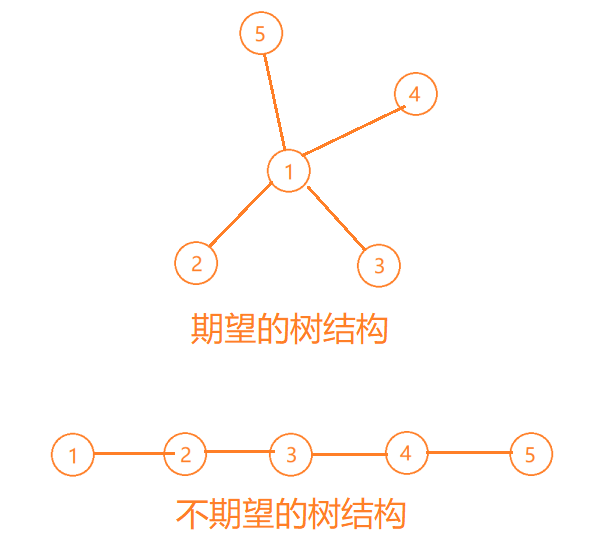
    }

    return curele;

}

改进的合并算法

如果我们不改进合并算法，那么可能会出现n个元素组成的树高度是n，这样的树很影响查找速度



如下改进算法，原理为合并时将树高度较小的树合并到高度较大的数，这样就需要我们保存树的高度信息，还记得我们的根元素的值是根元素的位置吗，现在我们将根元素保存的值为树的高度，为了和子元素的信息有区别，这个值为负数

// 初始化，各个元素的为一颗树，其根就是元素本身，所以

function init(){

    list = [0, -1, -1, -1, -1, -1, -1]

}

// 合并 x 和 y 所在的树

function megre(x, y){

    var xRoot = x;

    while(list[xRoot] > 0){

        xRoot = list[xRoot];

}

    var yRoot = y;

    while(list[yRoot] > yRoot){

        yRoot = list[yRoot];

}

    // 如果x树的高度大于y树，则将y树合并到x树

    if(-list[xRoot] > -list[yRoot]){

        // 将y树的父节点设为x

        list[yRoot] = xRoot;

    }

    // 如果y树的高度大于x树，则将x树合并到y树

    else if(-list[xRoot] < -list[yRoot]){

        list[xRoot] = yRoot;

    }

    // 如果2棵树高度相等

else{

        list[yRoot] = xRoot;

        // 更新树的高度

        list[xRoot] = list[xRoot] - 1;

}

    list[yRoot] = xRoot;

}

// 查找元素

function find(ele){

    var curele = ele;

    while(list[curele] > 0){

        curele = list[curele]

}

    return curele;

}

改进的查找算法

我们期望树的高度是2，但改进的合并算法无法满足这个期望，所以我们改进查找算法，我们将查找路径上的所有元素的父指向根

// 查找元素

function find(ele){

    var rootele = ele;

    while(list[rootele] > 0){

        rootele = list[rootele]

    }

    var curele = ele;

    // 如果当前元素不是根元素，则说明当前元素是树中的某个元素，所以我们需要将该元素的父修改为根

    while(curele != rootele){

        list[curele] = rootele;

        curele = list[curele];

}

    return rootele;

}

改进后虽然在第一次查找时耗时会增多，但以后的查找耗时会减少

注：上面的代码没有测试过，因为我懒得测了