Huffman树

> 目前常用的图像，视频，音频等多媒体信息，一般用二进制流管道的方式传输，由于数据量大，必须进行重新编码，达到数据压缩的效果，压缩方式也分为无损压缩和有损压缩，而Huffman编码就是数据压缩技术中的无损压缩方式。

前言

Huffman树是一种变长的编码方案，（通信二进制编码），使用频率高的数据编码长，使用频率低的数据编码较短。从而是总数据量最小。

对比

在了解Huufman的具体编码方式之前，我们先来看看其他的集中编码方式

```
1 AAAABBBCDDBBAAA
```

ASCII编码：

ASC编码是一种等长编码，一个字符就是一个字节（1个btye，8个bit），就上面的字符串编码玩就是

> 15*8=120bit。

等长编码：

> A B C D
>
> 00 01 10 11
>
> 2*15 = 30

变长编码：

不用看了，理论上成立，但是用屁股想都知道没法解码。

> A B C D
>
> 0 1 10 11

Huffman编码：

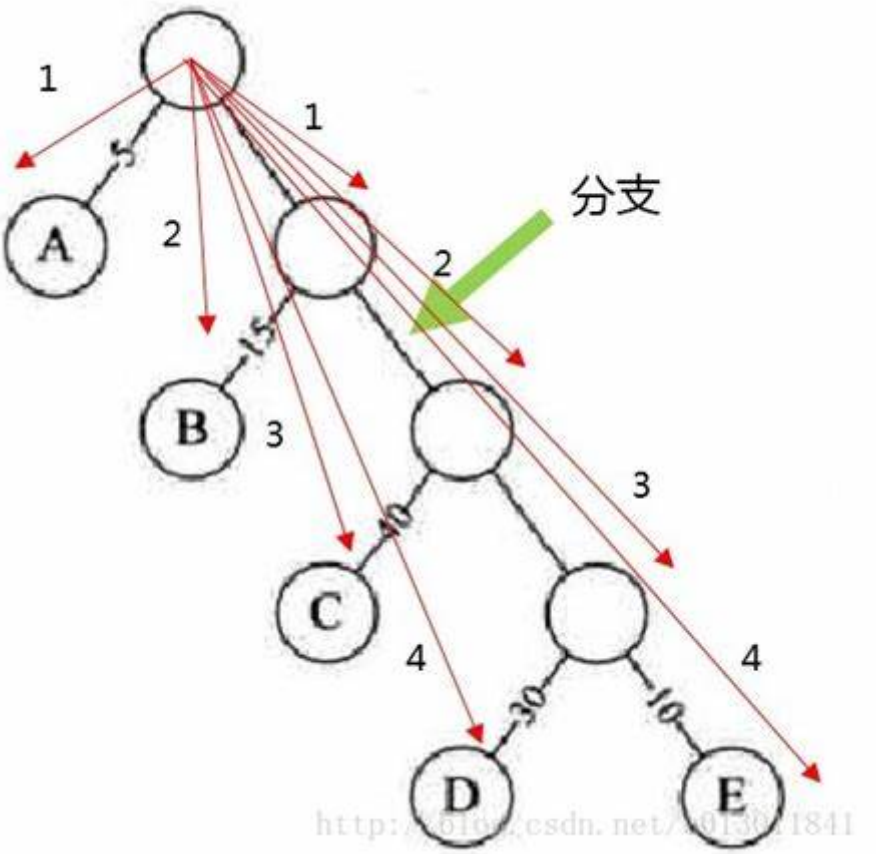
> A : 7 B : 5 C : 1 D : 2
>
> 0 11 100 101
>
> 7*1+5*2+.... = 26bit

Huffman编码原则

> 1.码长变短。
>
> 2.一个字符编码不能是其他字符编码的前缀。

Huffman的编码实现

1.二叉树的路径长度（path length）根节点到所有结点的路径之和



2.二叉树的外路径长度（ epl ）根节点到所有叶子结点的长度

> 完全二叉树的外路径长度最短，所以字符出现概率相等的情况下，完全二叉树最合适，定长编码的效率高于变长。

3.二叉树的带权路径长度

> 权（**weight**）：字符的使用概率， 用来统计叶子节点的值。

带权路径长度：（n个带权叶子结点。）

抽象类ADT

先说一下huffman树的作用范围，首先这个树的方法都是实例方法，所以没有实例化就别tm调用了，因为ACSII的编码范围就是256个字符，所以这个Huffman树的编码范围就是256个字符。

> 另外，传入的必须是编码字符的权重数组，比如有6个A，5个B，7个C，就要传入{6,5,7}。

huffman树节点类

0号地址不使用	index	weight	parent	lchild	rchild	
信息元所占用空间	1	5	9	0	0	最小的两个中序号小的放前面
	2	20	11	0	0	
	3	30	13	0	0	
	4	15	11	0	0	
	5	26	12	0	0	
	6	12	10	0	0	
	7	20	12	0	0	
	8	9	9	0	0	
分支节点(根和内点)	9	14	10	1	8	
	10	26	13	6	9	
	11	35	14	2	4	
	12	46	14	5	7	
	13	56	15	3	10	
	14	81	15	11	12	
	15	137	0	13	14	

> 这里我们用-1表示，没有储存元素。

```java
package DS2;

public class TriElement {
    int data;
    int parent,left,right;
    public TriElement(int data,int parent,int left,int right){
        this.data = data;
        this.parent = parent;
        this.left = left;
        this.right = right;
    }
    public TriElement(int data) {
        this(data,-1,-1,-1);
    }

    public String toString(){
        return "
("+this.data+","+this.parent+","+this.left+","+this.right+")";
    }

    public Boolean isLeaf(){
        return this.left == -1 && this.right == -1;
    }
}

```

huffman树类

构建哈夫曼编码树

> 实际上来说，这颗树是从下往上左右子树同时构建的，最后加出来的最大值作为根节点。

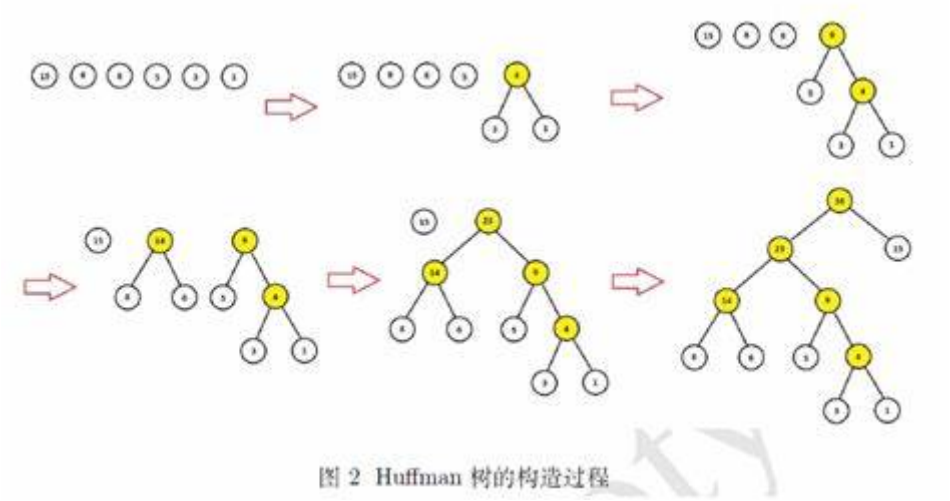


图 2 Huffman 树的构造过程

```
package DS2;

public class HuffmanTree {
    private String charset;
    //编译字符集范围，后文中做了限制，我们最多能编译256个字符。
    private TriElement[] hafftree;
    //表示字符叶子结点的只有前n（weight.length）个
    public HuffmanTree( int[] weight ){
        this.charset = "";
        for(int i = 0;i < weight.length; i++){
            this.charset += (char)('A'+i);
        }
        //初始化字符集。
        int n = weight.length;
        this.hafftree = new TriElement[2*n-1];
        //初始化树，这里我们用的是数组实现的二叉树，而haffman树是没有一度节点的所以，运用节点之间的计算　　公式算出储存空间为2n-1。
        for(int i = 0;i < n;i++){
            this.hafftree[i] = new TriElement(weight[i]);
            //将前n个权重作为子叶节点扔到数组的前n位。
        }

        for(int i = n;i < 2*n-1;i++){
            //这里开始，我们通过比较计算出，每一轮合并之前的最小值，和第二小值，然后进行合并。
            //2n-1在这里被我们作为循环次数
            int min1 = Integer.MAX_VALUE;
            int min2 = min1;
            int x1 = -1;
            int x2 = -1;

            for(int j = 0;j < i;j++){
                if(this.hafftree[j].parent == -1){
                    //当这个节点已经有父级节点了，就不用再比较了。
                    if(this.hafftree[j].data < min1){

                        min2 = min1;
                        x2 = x1;
                        min1 = this.hafftree[j].data;
                        x1 = j;
                    }else if(this.hafftree[j].data < min2){
                        min2 = this.hafftree[j].data;
```

```
                        x2 = j;
                    }
                }
            }
            this.hafftree[x1].parent = i;
            this.hafftree[x2].parent = i;
            //合并出来的节点作为两个最小值的父节点。
            this.hafftree[i] = new TriElement(min1+min2,-1,x1,x2);
            //在第i位创建这个合并出来的父点，并且此时的父节点没有父节点，所以如果值仍然是最小的话会再次              //参与比较和构建
        }
    }
}
```

获取树映射出的编码序列

> 这里我们的前n个数组元素正好是{a,b,c,n....}的权重便于我们获取他的编码。

```
   public String getCode(int i){
        int n = 8;
        char hufcode[] = new char[n];
        //按照ACS编码来说，二的八次幂能储存256个不同字符（等长编码。）
        int child = i;
        //要获取的第i个字符的编码一定是叶子结点，想要获得他的huffman编码就一定需要，从下往上遍历。
        int parent = this.hafftree[child].parent;
        //获取要获取字符的父级结点的id。
        for(i= n-1;parent!=-1;i--){
            // i=7，parent有值，i--
            hufcode[i] = (this.hafftree[parent].left == child)? '0' : '1';
            //从huffcode的最后一位开始添加二进制数据，判断child相对于parent的位置，左填0，右填1.
            child = parent;
            //节点上移。
            parent = hafftree[child].parent;
            //结点上移，同时更新判断条件。
        }
        return new String(hufcode,i+1,n-1-i);
        //因为我们是从后往前填的编码，所以要反向获取。
   }
```

打印huffman树的数组结构和编码

> nothing to say

```
    public String toString(){
        String str = "Huffman树节点数组：[";

        for(int i = 0;i < this.hafftree.length;i++){
            str += this.hafftree[i].toString() +',';
        }
        str += "]\n\r"+"Huffman编码：";
        for(int i = 0;i < this.hafftree.length;i++){
            str += this.charset.charAt(i)+';'+ this.getCode(i)+',';
        }
        return str;
    }
```

huffman树编码

> 就比如c-a = 2，而id刚好从零开始。

```
    public String encode(String text){
        String compressed = "";
        for(int i =0;i<text.length();i++){
            compressed += this.getCode(text.charAt(i)-'A');
        }
        return compressed;
    }
```

huffman解码

> 我的思维局限在了字符串匹配，但是其实这个编码反应的就是树从上向下遍历的路径，我们只需要遍历到结尾就可以拿到编码字符了。

```
    public String decode(String compressed){
        //解码的方式不是字符串匹配，而是根据预解码的顺序来从树顶向下索引，
        // 到叶子结点的时候，就记录字符，重置结点到根节点。
        String text = "";
        int node = this.hafftree.length-1;
        //获取根节点
        for(int i = 0;i<compressed.length();i++){
            if(compressed.charAt(i) == '0'){
                node = this.hafftree[node].left;
            }else if(compressed.charAt(i) == '1'){
                node = this.hafftree[node].right;
            }else{
                return "编码字符集错误！";
            }
            if(this.hafftree[node].isLeaf()){
                text += 'A' + node;
                node = this.hafftree.length-1;
            }
        }
        return text;
    }
```