# 树

## 基础概念

### 高度，深度和层数

*若讨论的是整棵树，则树高度和深度在数值上相等*
*高度是从下往上计数，深度是从上往下计数*
**有两种常见定义**

1. 按边数计(只有一个结点，值为0)
   - 高度：结点的高度=结点到叶子结点的最大路径
   - 深度：结点的深度=根节点到结点的最大路径
2. 按结点数计(只有一个结点，值为1)[**教材采用这种定义，这时深度的值=层数**]
   - 高度：结点高度=结点到叶子结点最长路径上的结点数
   - 深度：结点深度=根节点到结点最长路径上的节点数

### 二叉树的性质

- **性质一**：对于一棵二叉树，第 `i` 层的最大结点数量为 $2^{i-1}$ 个（每个结点都有左右孩子，第i层节点数=2*第i-1层结点数，一路回推到第1层，实际上就是1为首项的等比数列求第i项的值）
- **性质二**：对于一棵深度为 `k` 的二叉树，可以具有的最大结点数量为：

$$n = 2^0 + 2^1 + 2^2 + ... + 2^{k-1}$$

  实际上就是每层最大结点数相加，化简得

$$S_n = \frac{a_1 \times (1-q^n)}{1-q} = \frac{1 \times (1-2^k)}{1-2} = -(1-2^k) = 2^k - 1$$

  所以一棵深度为 `k` 的二叉树最大结点数量为 $n = 2^k - 1$，顺便得出，结点的边数为 $E = n - 1 = 2^k - 2$。
- **性质三**：对于任何一棵二叉树，如果其叶子结点个数为 $n_0$ ，度为2的结点个数为 $n_2$ ，那么两者满足以下公式：

$$n_0 = n_2 + 1$$

*证明如下：*

**思路：联立两个求总结点数的公式求出目标关系**

**1. 总结点数=三个不同类型的结点数量的和**

**2. 边数=总结点数-1**

**那么问题就转成求边数，度为1的结点有一条边，度为2的结点有两条边，度为0的结点没有，加在一起就是整棵二叉树的边数之和**

假设一棵二叉树中度为0、1、2的结点数量分别为$n_0$、$n_1$、$n_2$，由于一棵二叉树中只有这三种类型的结点，那么可以直接得到结点总数：

$$n = n_0 + n_1 + n_2$$

再从二叉树的边数上考虑，因为每个结点有且仅有一条边与其父结点相连，那么边数之和就可以表示为：

$$E = n_1 + 2n_2$$

结合在**性质二**中推导的结果，可以得到另一种计算结点总数的方式：

$$E = n - 1 = n_1 + 2n_2$$

$$n = n_1 + 2n_2 + 1$$

再我们第一个公式：

$$n = n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$$

化简得对于任何一棵二叉树，如果其叶子结点个数为 $n_0$ ，度为2的结点个数为 $n_2$ ，那么两者满足以下公式：

$$n_0 = n_2 + 1$$

- **性质四**：一棵具有 n 个结点的完全二叉树深度为 $k = \lceil log_2(n+1) \rceil$

  假设层数为 k ，深度为k-1层的树的最多结点数<n<=深度为k层的树的最多结点数（**性质二**）

  $$2^{k-1} - 1 < n <= 2^k - 1$$

  对不等式两边同时取对数，得到：

  $$k - 1 < log_2(n+1) <= k$$

$log_2(n+1)$在k-1到k之间且不等于k-1，而且为整数，因此有
$k=\lceil log_2(n+1)\rceil$

- **性质五**：一颗有 n 个结点的完全二叉树，对于任意一个编号为 i 的结点，结点的顺序为从上往下，从左往右：
  - 如果 2i > n ，则结点 i 没有左孩子
  - 如果 2i + 1 > n ，则结点 i 没有右孩子
  - 其左孩子为 2i ，右孩子为 2i + 1 （前提是有孩子）
  - 如果 i = 1 ，那么此结点为二叉树的根结点，如果 i > 1 ，那么其父结点就是 $\lfloor i/2 \rfloor$（换一种表示：(n-1)/2）【这里的除法采用计算机int类型除法的截断尾数】

# 判断二叉树形状

## 判断是否为完全二叉树

> 完全二叉树：根节点到倒数第二层为止都是满二叉树，最后一层叶子节点必须连在一起（倒数第二层的根节点要么有两个孩子，要么只有左孩子）

*思路：根据上面描述，完全二叉树每一个结点的 Height(left)==Height(right) 或 Height(right)+1*

```
bool isCompleteTree(TreeNode<T>* root) {
  //层序遍历，出队后判断出队结点是否满足完全二叉树要求
                Queue<TreeNode<T>*> s;
                s.EnQueue(root);
                while (!s.isEmpty()) {
                        s.DeQueue(root);
                        int leftHeight = getHeight(root->leftChild);
                        int rightHeight = getHeight(root->rightChild);
                        if (leftHeight > rightHeight + 1 || leftHeight < rightHeight)return fals
                        else {
                                if (root->leftChild != nullptr)s.EnQueue(root->leftChild);
                                if (root->rightChild != nullptr)s.EnQueue(root->rightChild);
                        }
                }
                return true;
        }
```

## 判断是否为BST

- **错误代码**
  BST是根节点值>左子树所有结点值且<右子树所有结点值
  以下代码只考虑了当前结点和左右孩子的大小关系

```
bool isBST(TreeNode<T>* root) {
        if (root == nullptr)return true;
        if (root->leftChild != nullptr && root->leftChild->data > root->data)return false;
        else if (root->rightChild != nullptr && root->rightChild->data < root->data)return false
        if (isBST(root->leftChild) && isBST(root->rightChild))return true;
        else return false;
}
```

- 正确思路1
  中序遍历，将结点放入数组中，判断是否升序
- 正确思路2
  对错误代码的思路进行纠正：每次判断都为该节点设定一个上限max和下限min，越界则不符合BST要求

```
#include<iostream>
using namespace std;
struct TreeNode {
        TreeNode* leftChild;
        TreeNode* rightChild;
        int data;
};
bool isBST(TreeNode* root, int Max, int Min) {
        if (root == nullptr)return true;
        if (root->data > Max || root->data < Min)return false;
        return isBST(root->leftChild, root->data, -10086) && (isBST(root->rightChild, 10086, roo
}
TreeNode* CreateBinaryTree(TreeNode*& root) {//前序遍历创建二叉树
                int a;
                cin >> a;
                if (a == 0)return nullptr;
                else {
                        root = new TreeNode;
                        root->data = a;
                        root->leftChild = CreateBinaryTree(root->leftChild);
                        root->rightChild = CreateBinaryTree(root->rightChild);
                }
                return root;
        };
```

# 哈夫曼树及哈夫曼编码

### 定义

叶子节点带权路径之和最小的二叉树。思路就是权值大的结点里根节点越近，反之越远

计算权值：（每个叶子节点权值*到根节点距离）的总和

**应用在最优判断树**

**应用在编码里**

通过设计变长编码，可以让频率高的字符编码短，频率低的字符编码长，这样的编码平均编码长度短，效率高。同时也要考虑歧义问题，每个编码都不能是其他编码的前缀，也就是说要设计出前缀编码。哈夫曼编码就是平均编码长度最短的前缀编码（设计左1右0，左0右1都可以）

**构建哈夫曼树的算法**

每次将权值最小的两个结点拿出并合并（新节点的权值是它们之和），然后扔回去继续上述操作，直到剩下一个结点

**主要函数的功能注释**

- `createHuffmanTree()` :根据传进来的元素值数组和对应的权重数组，及元素个数建立哈夫曼树，并用 `initPath()` 编码
- `merge()` :在 `createHuffmanTree()` 里调用，将两个权值最小的结点合并
- `initPath()` :对建好的哈夫曼树每个结点进行编码，并将叶子结点的值和权合并在 `Ecode` 结构体里，保存在 `save` 数组中
- `writeTreeFile()` :将save数组中的哈夫曼编码写入文件HfmTree.txt中保存
- `Encoding()` : 按照哈夫曼编码将文件ToBeTran.txt中的内容转成01串，存入Code.txt文件里
- `Decoding()` : 按照哈夫曼编码将Code.txt文件中的01串进行译码，结果存入Text.txt文件中

**代码如下**

以下代码需要注意的地方还是很多的，特别是 `createHuffmanTree()` 和 `initPath()` 。还有那些指针，内存分配，什么时候传引用，什么时候传参，一些细微的地方可能在意料之外改了root指针的值等等一堆乱七八糟的东西（md调了好几天，一点写错就一堆bug，淦！！！！！）

```cpp
#include<iostream>
#include<string>
#include<fstream>
#include"Hp.h"//引入最小堆
using namespace std;
template <class T>
struct HuffmanNode {
	double weight;
	T data;
	string path;
	HuffmanNode* leftChild;
	HuffmanNode* rightChild;
	HuffmanNode* parent;
	HuffmanNode(){};
	HuffmanNode(int w) :weight(w) {};
	HuffmanNode(int w, T d) :weight(w), data(d), leftChild(nullptr), rightChild(nullptr), pa
	bool operator <= (HuffmanNode& R) {
		if (weight <= R.weight)return true;
		else return false;
	}
	bool operator < (HuffmanNode& R) {
		if (weight < R.weight)return true;
		else return false;
	}
	bool operator >=(HuffmanNode& R) {
		if (weight >= R.weight)return true;
		else return false;
	}
	bool operator >(HuffmanNode& R) {
		if (weight > R.weight)return true;
		else return false;
	}
	friend ostream& operator <<(ostream& cout, HuffmanNode<T>& r) {
		cout << r.weight << " ";
		return cout;
	}
};
template <class T>
struct Ecode {//保存元素值及其编码的结点
	T val;
	string enCode;
	Ecode(){};
	Ecode(T v, string e) :val(v), enCode(e) {};
};
template <class T>
class HuffmanTree {
public:
	HuffmanTree() {
		root = nullptr;
		save = nullptr;
		countLeaf = 0;
```

```
        }
        ~HuffmanTree() { delete[]save; }
public:
        bool isEmpty() {
                if (root == nullptr)return true;
                else return false;
        }
        HuffmanNode<T>* getRoot() { return root; }
        void PrePrint() {
                PrePrint(root);
        }
        void createHuffmanTree( T initData[], double initWeight[], int count) {
                HuffmanNode<T> temp;
                MinHeap<HuffmanNode<T>> hp;
                //将权重及其对应的元素值存放在哈夫曼结点里，丢到最小堆中
                for (int i = 0; i < count; i++) {
                        temp.weight = initWeight[i];
                        temp.data = initData[i];
                        temp.leftChild = temp.rightChild = nullptr;
                        hp.EnMinHeap(temp);
                }
                //每次拿出权重最小的两个结点进行合并，得到的parent结点再扔回最小堆里
                while (hp.getCurrentSize() > 1) {//最小堆只剩下一个结点时，操作结束
                        HuffmanNode<T>* firstMin = new HuffmanNode<T>;
                        HuffmanNode<T>* secondMin = new HuffmanNode<T>;
                        hp.DeMinHeap(*firstMin);
                        hp.DeMinHeap(*secondMin);
                        HuffmanNode<T>* parent = new HuffmanNode<T>;
                        merge(firstMin, secondMin, parent);
                        hp.EnMinHeap(*parent);
                }
                this->root = &hp.getMin();//哈夫曼树建成，赋给root
                countLeaf = count;
                initPath();//给结点编码
        }
        void initPath() {
                /*这里有点奇怪：不要这个public，把private的这个函数直接写在
                                                createHuffmnaTree()函数里，就会出现bug，打印结点
                                                打不出来还报错*/
                string code;
                save = new Ecode<T>[countLeaf];
                initPath(this->root, code);
        }
        void PrePrintPath() {
                PrePrintPath(root);
        }
        bool writeTreeFile() {//将建好的哈夫曼树写入文件HfmTree中保存
                ofstream file("HfmTree.txt", ios::out);
                if (file) {
                        writeTreeIntoFile(file,this->root);
                        file.close();
```

```
                return true;
        }
        else cout << "HfmTree.txt打开失败" << endl;
        return false;
}
bool Encoding() {//对文件ToBeTran的内容编码，结果存入Code文件
        string saveText;//保存从ToBeTran文件中读出的内容
        //读取ToBeTran内容,用saveText保存
        ifstream ifile("ToBeTran.txt", ios::in);
        if (ifile) {
                string buf;
                while (getline(ifile, buf)) {
                        saveText += buf;
                }
                ifile.close();
        }
        else {
                cout << "ToBeTran.txt文件打开失败" << endl;
                return false;
        }
        //将读到的内容编码并存入Code.txt文件中
        //如果内存中没有现成哈夫曼树,就先从HfmTree.txt文件里读取，并存到save数组里
        if (save == nullptr) {
                //去MSVN了解fopen_s(),fscanf_s()这两个函数的用法
                errno_t err;
                FILE* ifp;
                err = fopen_s(&ifp, "HfmTree.txt", "r");//r：以只读的方式打开
                if (err==0) {
                        /*char val;
                        char enCode[50];*/
                        //char saveVal[50];
                        Ecode<char> temp[5000];
                        int i = 0;
                        //while (fscanf_s(ifp, "%c %s", &val, enCode, 1, _countof(enCode
                        while (fscanf_s(ifp, "%c %s", temp[i].val,temp[i].enCode) != -1)
                                //用temp数组暂存哈夫曼数据，并用i记录有多少个
                                //string v = val;
                                /*string e = enCode;
                                Ecode<char> t(val, e);
                                temp[i++] = t;*/
                                i++;
                        }//end while
                        //将获得的哈夫曼数据存入save数组
                        countLeaf = i;
                        save = new Ecode<T>[countLeaf];
                        for (int j = 0; j < countLeaf;j++)save[j] = temp[j];
                }
                else {
                        cout << "HfmTree.txt文件打开失败" << endl;
                        return false;
                }
```

```
        }
        ofstream ofile("Code.txt", ios::out);
        if (ofile) {
                for (int i = 0; i < saveText.length(); i++) {
                        for (int j = 0; j < countLeaf; j++) {
                                //匹配到了就将对应编码写入Code文件
                                if (save[j].val == saveText[i]) {
                                        ofile << save[j].enCode<<" ";
                                        break;
                                }
                                else if(j==countLeaf) {
                                        cout << "出现未知的字符，编码失败" << endl;
                                        return false;
                                }
                        }
                }
                ofile.close();
        }
        else {
                cout << "Code.txt", ios::out;
                return false;
        }
        return true;
}
bool Decoding() {//将Code文件中的编码进行译码，结果存入Text文件中
        //去MSVN了解fopen_s(),fscanf_s()这两个函数的用法
        ofstream ofile("Text.txt", ios::out);
        errno_t err;
        FILE* ifp;
        err = fopen_s(&ifp,"Code.txt", "r");//r：以只读的方式打开
        if (err==0&&ofile) {
                char code[50];//这里不严谨，还是要按最长编码来定数组长度，不然可能溢出缓冲区
                //还有一点可以优化：文件中如果有换行，现在这个代码翻译出来换不了行
                while (fscanf_s(ifp, "%s", code,_countof(code)) != -1) {//每次从文件按里读
                        for (int j = 0; j < countLeaf; j++) {
                                //匹配到了就将对应字符写入写入Text文件
                                if (save[j].enCode == code){
                                        ofile << save[j].val;
                                        break;
                                }
                                else if (j == countLeaf) {
                                        cout << "出现未知的编码，译码失败" << endl;
                                        return false;
                                }
                        }//end for
                }//end while
                ofile.close();
        }
        else {
                cout << "Code.txt或Text.txt打开失败" << endl;
                return false;
```

```
                }
                return true;
        }
private:
        HuffmanNode<T>* root;
        int countLeaf;//记录叶子节点的数量
        Ecode<T>* save;//创建哈夫曼树后，将元素的值和对应编码记录在这个数组里
        void merge(HuffmanNode<T> *&firstMin, HuffmanNode<T> *&secondMin, HuffmanNode<T>* & pare
        {
                //合并两个权值最小的结点
                double totalWeight = firstMin->weight + secondMin->weight;
                parent->weight = totalWeight;
                parent->data = '*';
                firstMin->parent = secondMin->parent = parent;
                parent->leftChild = firstMin;
                parent->rightChild = secondMin;
        }
        void PrePrint(HuffmanNode<T>* root) {
                if (root == nullptr)return;
                if(root->leftChild==nullptr&& root->rightChild==nullptr)
                cout << root->data << " ";
                PrePrint(root->leftChild);
                PrePrint(root->rightChild);
        }
        void initPath(HuffmanNode<T>* &root, string code) {
                if (root == nullptr)return;
                //给每个结点编码，并将对应叶子结点的元素值和编码保存在数组中去
                static int i = 0;//应该要是静态变量，而不是函数签名里传一个形参进来
                root->path = code;
                if (root->rightChild == nullptr && root->leftChild == nullptr) {
                        Ecode<T> temp(root->data, root->path);
                        save[i++] = temp;
                }
                initPath(root->leftChild, code + "1");
                initPath(root->rightChild, code + "0");
        }
        void PrePrintPath(HuffmanNode<T>* root) {
                if (root == nullptr)return;
                if (root->leftChild == nullptr && root->rightChild == nullptr) {
                        cout << root->data <<"的编码是："<<root->path<<endl;
                }
                PrePrintPath(root->leftChild);
                PrePrintPath(root->rightChild);
        }
        void writeTreeIntoFile(ofstream& f,HuffmanNode<T>*root) {
                if (root == nullptr)return;
                if (root->leftChild == nullptr && root->rightChild == nullptr)
                        f << root->data << " "<<root->path<<endl;
                writeTreeIntoFile(f,root->leftChild);
                writeTreeIntoFile(f,root->rightChild);
```

```
        }
};
```