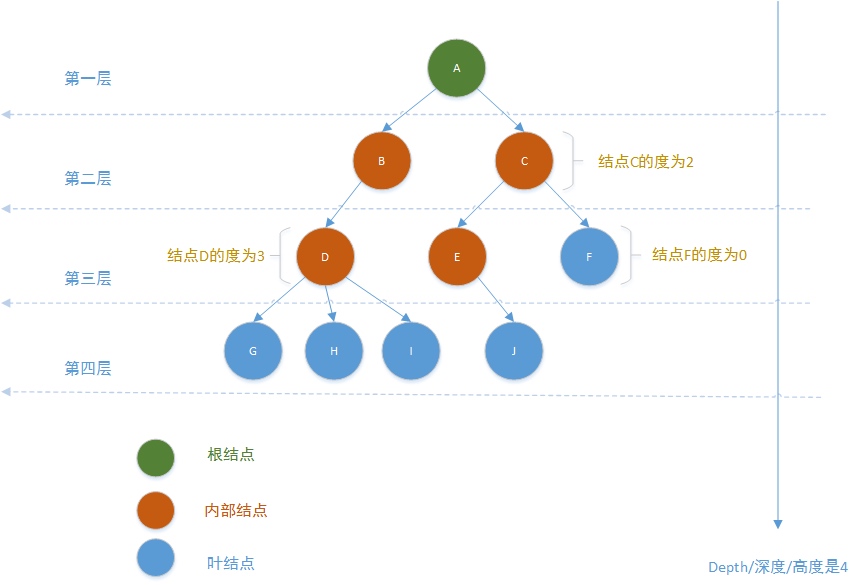
**二、树**

**树的基本概念和术语**

[](https://cloud.8hfq.com/img/15324886413081.jpg)  
树是一种非线性的数据结构  
他是若干结点（A,B,C…等都是结点）的集合，是由唯一的根A和若干颗互不相交的子树组成的。其中每一棵子树又是一棵树，也是由唯一的根结点和若干颗互不相交的子树组成的。  
由此可知，树的定义是递归的。

树的结点包含一个数据元素以及若干指向其子树的分支。  
结点拥有的子树数目称为结点的度。  
度为0的结点称为叶结点或终端结点；度不为0的结点称为非终端结点或分支结点。  
除根结点之外，分支结点也称为内部结点。  
树的度是树内各结点的度的最大值。  
结点的子树的根称为该结点的孩子。相应地，该结点称为孩子的双亲。  
同一个双亲的孩子之间互称为兄弟（Sibling）。  
结点的祖先(Ancestor)是从根到该结点所经分支上的所有结点。  
结点的层次：结点的层次从根开始定义起，根为第一层，根的孩子为第二层。若某结点在第L层，则其子树的根就在第L+1层。其双亲在同一层的结点互为堂兄弟。  
树中结点的最大层次称为树的深度或高度(Depth)。  
如果将树中结点的各子树看成从左至右是有次序的，不能互换的，则称该树为有序树，否则称为无序树。  
森林是m(m>=0)棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。

**二叉树的先序,中序,后序,层次序遍历;**

**先序遍历**

操作过程如下  
如果二叉树为空树，则什么都不做，否则：  
1）访问根节点  
2）先序遍历左子树  
3）先序遍历右子树  
对应的算法描述如下

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | void preorder(BTNode \*p){  if(p!=NULL){  Visit(p);  preorder(p->lchild);  preorder(p->rchild);    } } |

**中序遍历**

操作如下  
如果二叉树为空树，则什么都不做，否则：  
1）中序遍历左子树。  
2）访问根节点。  
3）中序遍历右子树。  
对应的算法描述如下

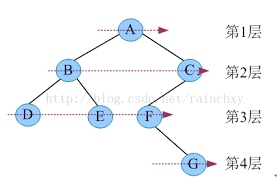
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | void inorder(BTNode \*p){  if(p!=NULL){  preorder(p->lchild);  Visit(p);  preorder(p->rchild);  } } |

**后序遍历**

操作如下  
如果二叉树为空树，则什么都不做，否则：  
1）后序遍历左子树。  
2）后序遍历右子树。  
3）访问根节点。  
对应的算法描述如下

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | void postorder(BTNode \*p){  if(p!=NULL){  preorder(p->lchild);  preorder(p->rchild);  Visit(p);  } } |

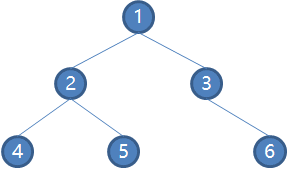
**层次遍历**

[](https://cloud.8hfq.com/img/15325064536209.jpg)  
如图所示为二叉树的层次遍历，即按照箭头所示方向，按照1，2，3，4的层次顺序对二叉树中的各个结点访问。  
要进行层次遍历需要建立一个循环队列，先将二叉树头结点入队列，然后出队列，访问该结点，如果他有左子树，则将左子树的根节点入队，如果他有右子树，则将右子树的根节点入队，然后出队列，对出对结点访问，如此反复，直到队列为空为止。  
得到的算法如下

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | void level(BTNode \*p) {  int front, rear;  BTNode \*que[maxSize];  front = rear = 0;  BTNode \*q;  if (p != NULL) {  rear = (rear + 1) % maxSize;  que[rear] = p;  while (front != rear) {  front = (front + 1)  maxSize;  q = que[front];  Visit(q);  if (q->lchild != NULL) {  rear = (rear + 1) % maxSize;  que[rear] = q->lchild;  }  if (q->rchild != NULL) {  rear = (rear + 1) % maxSize;  que[rear] = q->rchild;  }   }  } } |

**二叉树遍历算法的改进**

**二叉树深度优先遍历算法的非递归实现**

[](https://cloud.8hfq.com/img/20181116145316.png)

1. 先序遍历非递归算法

出栈时判断是否有孩子，右孩子先入栈，左孩子后入栈，因为对左孩子的访问要先于右孩子

* 结点1入栈
* 1出栈，输出结点1，并将1的左右孩子2，4入栈，右孩子先入栈，左孩子后入栈。因为对左孩子的访问要先于右孩子，后入栈的会先出栈访问
* 2出栈，并将2的左右孩子3和5入栈
* 3出栈，3无叶子节点
* 5出栈
* 4出栈，此时栈空，进入终态
* 遍历顺序为1，2，3，4，5

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | public List<Integer> preorderTraversal(TreeNode root){  List<Integer> resultList = new ArrayList<>();  Stack<TreeNode> treeNodeStack = new Stack<>();  if(root ==null)  return resultList;  treeNodeStack.push(root);  while (!treeNodeStack.isEmpty()){  TreeNode tempNode = treeNodeStack.pop();  if (tempNode!=null){  resultList.add(tempNode.val);  treeNodeStack.push(tempNode.right);  treeNodeStack.push(tempNode.left);  }  }  return resultList;  } |

1. 中序遍历非递归算法

入栈即考虑左孩子是否存在，存在则入，出栈考虑其右孩子是否存在，存在则入。

* 1入栈 1的左孩子2存在
* 2入栈 2的左孩子3存在
* 3入栈 3的左孩子不存在
* 3出栈 3的右孩子不存在
* 2出栈，2的右孩子5存在，故5入栈 5的左孩子不存在
* 5出栈，5的右孩子不存在
* 1出栈，1的右孩子4存在，4 入栈
* 4出栈，此时栈空  
  综上步骤得知：

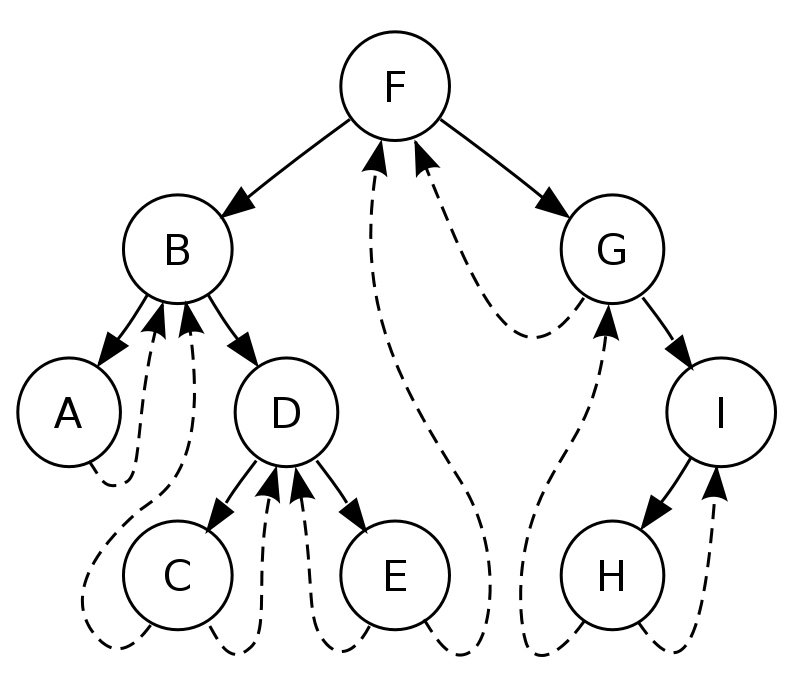
1. 开始根节点入栈
2. 循环进行如下操作：如果栈顶的左孩子存在，则左孩子入栈，如果栈顶的左孩子不存在，则出栈并且输出栈顶结点，然后检查其右孩子是否存在，如果存在，则右孩子入栈。
3. 当栈空时，算法结束

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | public List<Integer> inorderTraversal(TreeNode root){  List<Integer> resultList = new ArrayList<>();  Stack<TreeNode> treeNodeStack = new Stack<>();  TreeNode cur = root;  while(cur!=null || !treeNodeStack.empty()){  while (cur!=null){  treeNodeStack.add(cur);  cur= cur.left;  }  cur = treeNodeStack.pop();  resultList.add(cur.val);  cur = cur.right;  }  return resultList; } |

1. 后序遍历非递归算法

非递归先序遍历算法中的对左右子树的遍历顺序交换就可以得到逆后序遍历序列，然后将逆后序遍历序列逆序就得到了后序遍历。因此我们需要两个栈

**线索二叉树**

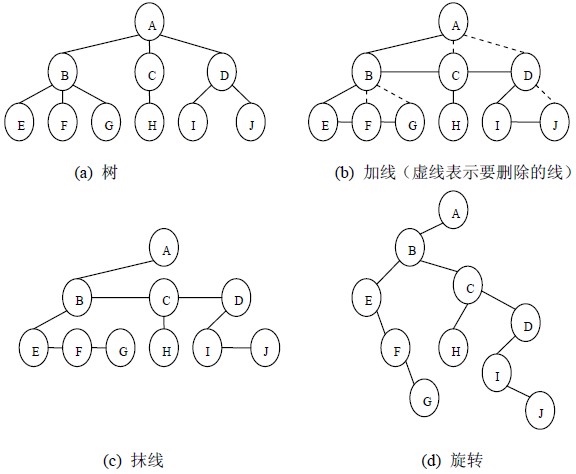
中序线索二叉树  
[](https://cloud.8hfq.com/img/15329385461570.jpg)  
线索二叉树(引线二叉树) 的定义如下:一个二叉树通过如下的方法“穿起来”：所有原本为空的右(孩子)指针改为指向该节点在中序序列中的后继，所有原本为空的左(孩子)指针改为指向该节点的中序序列的前驱。  
线索二叉树能线性地遍历二叉树，从而比递归的 中序遍历更快。使用线索二叉树也能够方便的找到一个节点的父节点，这比显式地使用父亲节点指针或者栈效率更高。这在栈空间有限，或者无法使用存储父节点的栈时很有作用（对于通过深度优先搜索来查找父节点而言)。 考虑这样的例子：一个节点k有一个右孩子r，那么r的左指针可能是指向一个孩子节点，或是一个指回k的线索。如果r有左孩子，这个左孩子同样也应该有一个左孩子或是指回k的线索。对于所有的左孩子同理。因此沿着这些从r发出的左指针，我们最终会找到一个指回k的线索。这种特性是对称的：当q是p的左孩子时，我们可以沿着q的右孩子找到一个指回p的线索。  
传统的二叉树一般都是以链式存储的结构来表示。这样，二叉树中的每个节点都可以用链表中的一个链节点来存储，每个链节点就包含了若干个指针。但是，这种传统的链式存储结构只能表现出二叉树中节点之间的父子关系，而且不能利用空余的指针来直接得到某个节点的在特定的遍历顺序（先序，中序，后序）中的直接前驱和直接后继。通过分析传统的二叉树链式存储结构表示的二叉树中，存在大量的空闲指针。若能利用这些空指针域来存放指向该节点的直接前驱或是直接后继的指针，则可以进行某些更方便的运算。这些被重新利用起来的空指针就被称为线索，加上了这些线索的二叉树就是线索二叉树。

**二叉树及其性质**

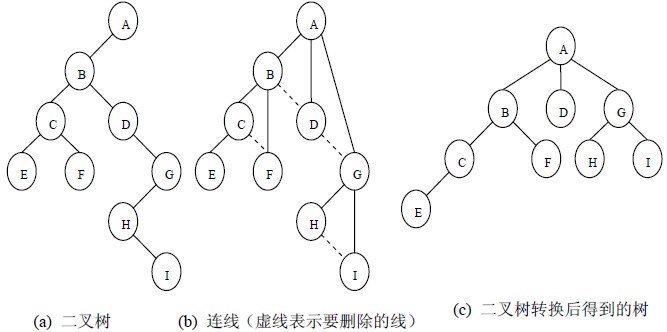
二叉树的定义  
1）每个结点最多只有2颗子树，即二叉树中节点的度只能为0，1，2  
2）子树有左右顺序之分，不能颠倒。  
性质1  
非空二叉树上叶子结点数等于双分支结点数加1  
性质2  
二叉树的第i层上最多有2i−12i−1个结点  
性质3  
高度或深度为k的二叉树最多有2k−12k−1个结点。换句话说，满二叉树中前k层的结点个数为2k−12k−1  
性质4  
有n个结点的完全二叉树，对各结点从上到下，从左到右依次编号（编号范围1~n）则结点之间有如下的关系  
若i为某结点a的编号则：  
如果i≠1，则a双亲结点的编号为 ⌊i/2⌋.  
如果2i≤n，则a左孩子的编号为2i；如果2i>n，则a无左孩子。  
如果2i+1≤n，则a右孩子的编号为2i+1；如果2i+1>n，则a无右孩子。  
性质5  
函数Catalan():给定n个结点，能够构成h(n)中不同的二叉树 h(n)=Cn=C(2n,n)/(n+1)h(n)=Cn=C(2n,n)/(n+1)  
注：C(n,r)=n!/[r!(n−r)!]C(n,r)=n!/[r!(n−r)!]  
性质6  
具有n(n>0)个结点的完全二叉树的高度为log2nlog2n

**普通树与二叉树的转换**

**树转换成二叉树**

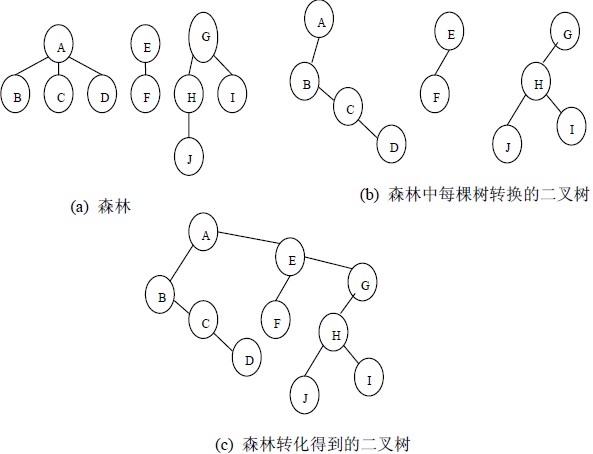
树转换成二叉树的过程如下  
1）将同一结点的各孩子结点用线串起来  
2）将每个结点的分支从左到右除了第一个外，其余的都剪掉，整理即可得到  
[](https://cloud.8hfq.com/img/15325127503357.jpg)

**二叉树转换成树**

二叉树转换为树是树转换为二叉树的逆过程，其步骤是：  
1）若某结点的左孩子结点存在，将左孩子结点的右孩子结点、右孩子结点的右孩子结点……都作为该结点的孩子结点，将该结点与这些右孩子结点用线连接起来；  
2）删除原二叉树中所有结点与其右孩子结点的连线；  
3）整理（1）和（2）两步得到的树，使之结构层次分明。  
[](https://cloud.8hfq.com/img/15325128959055.jpg)

**森林转换成二叉树**

森林是由若干棵树组成，可以将森林中的每棵树的根结点看作是兄弟，由于每棵树都可以转换为二叉树，所以森林也可以转换为二叉树。

将森林转换为二叉树的步骤是：  
1）先把每棵树转换为二叉树；  
2）第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子结点，用线连接起来。当所有的二叉树连接起来后得到的二叉树就是由森林转换得到的二叉树。  
[](https://cloud.8hfq.com/img/15325130681344.jpg)

**二叉树转换成森林**

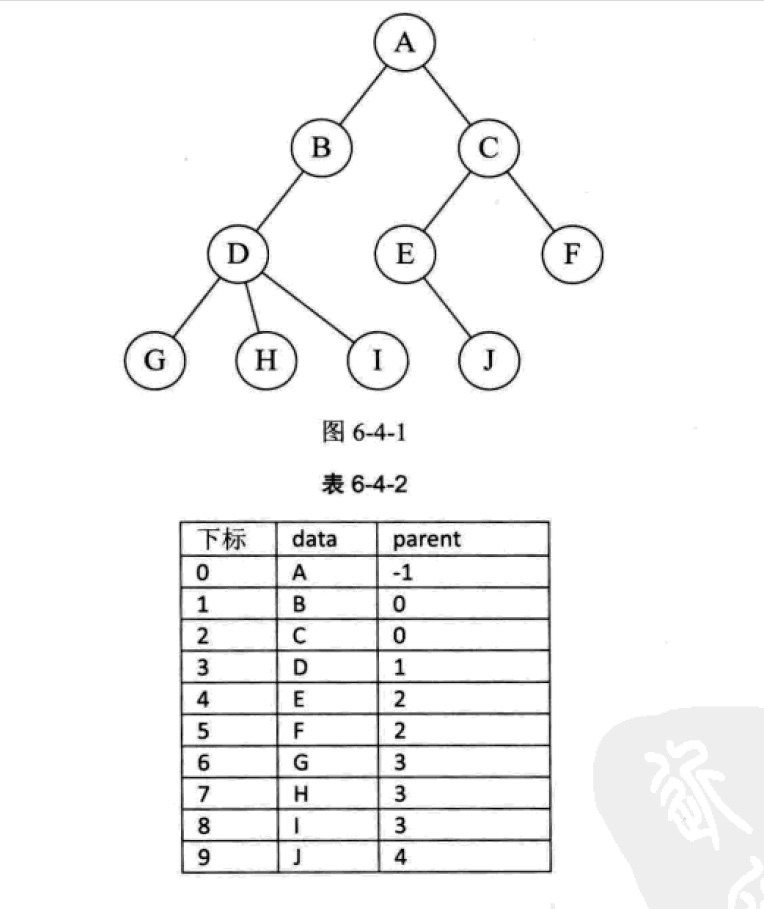
二叉树转换为森林比较简单，其步骤如下：  
1）先把每个结点与右孩子结点的连线删除，得到分离的二叉树；  
2）把分离后的每棵二叉树转换为树；  
3）整理第（2）步得到的树，使之规范，这样得到森林。

**森林和树的遍历**

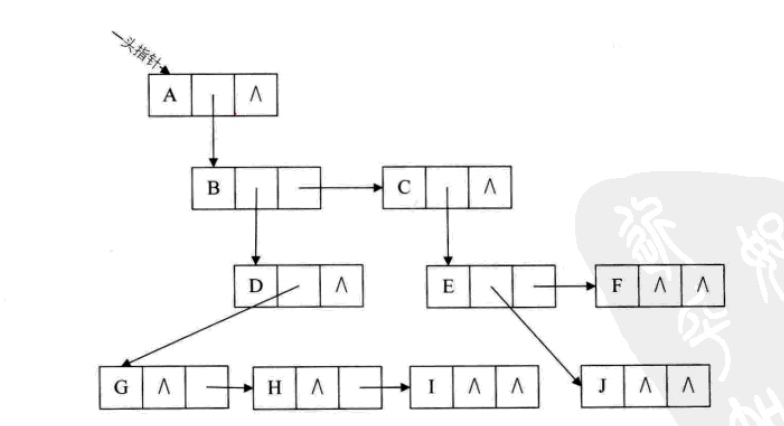
根据树与二叉树的转换关系以及二叉树的遍历定义可以推知，  
树的先序遍历与其转换的相应的二叉树的先序遍历的结果序列相同；树的后序遍历与其转换的二叉树的中序遍历的结果序列相同；树的层序遍历与其转换的二叉树的后序遍历的结果序列相同。  
由森林与二叉树的转换关系以及森林与二叉树的遍历定义可知，  
森林的先序遍历和中序遍历与所转换得到的二叉树的先序遍历和中序遍历的结果序列相同。

**树的存储结构,标准形式**

**顺序存储结构**

树的顺序存储结构最简单直观的是双亲存储结构，用一维数组即可实现。将所有结点存到一个数组中。每个结点都有一个数据域data和一个数值parent指示其双亲在数组中存放的位置。根结点由于没有父结点，parent用-1表示。  
int tree[maxSize]  
[](https://cloud.8hfq.com/img/15325138214744.jpg)

**链式存储结构**

1）孩子存储结构  
孩子存储结构实质上就是图的邻接表存储结构  
树就是一种特殊的图，把图中的多对多关系删减成一对多关系即可的得到树  
2）孩子兄弟存储结构  
树转换成二叉树的过程  
[](https://cloud.8hfq.com/img/15325141528648.jpg)

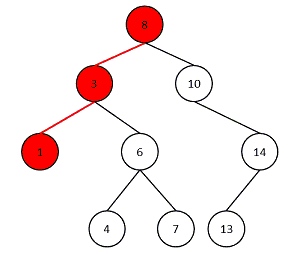
**完全树(complete tree)的数组形式存储**

**树的应用**

二叉排序树和平衡二叉树与查找关系密切，因为放到查找一章讲解

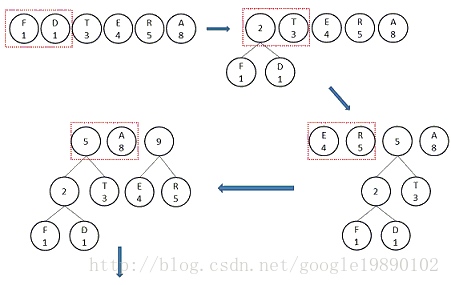
**Huffman树的定义与应用**

**Huffman树相关的概念**

1）路径：是指在一棵树中，从一个节点到另一个节点之间的分支构成的通路，如从节点8到节点1的路径如下图所示：[](https://cloud.8hfq.com/img/15325717690810.jpg)  
2）路径的长度：是指路径上的分支数目，在上图中，路径长度为2。  
3）树的路径长度：是指从根到每个结点的路径长度之和。  
4）带权路径长度：结点具有权值，从该结点到根之间的路径长度乘以结点的权值，就是该结点的带权路径长度。  
5）树的带权路径长度：是指树中所有叶子节点的带权路径之和。  
6 节点的权：指的是为树中的每一个节点赋予的一个非负的值，如上图中每一个节点中的值

**有了如上的概念，对于Huffman树，其定义为：**  
给定n权值作为n个叶子节点，构造一棵二叉树，若这棵二叉树的带权路径长度达到最小，则称这样的二叉树为最优二叉树，也称为Huffman树。

**Huffman树的构建**

给定n个权值，用这n个权值来构建赫夫曼树的算法描述如下：  
1）将这n个权值分别看成只有根节点的n棵二叉树，将这些二叉树的集合记为F。  
2）从F中选出两颗根结点的权值最小的树，作为左右子树，构建一棵新的二叉树，新的二叉树的根结点的权值为左右子结点的权值之和。  
3）从F中删去a,b，加入新构建的树C  
4）重复2，3步，直到F中只剩下一棵树为止，这棵树就是赫夫曼树。  
[](https://cloud.8hfq.com/img/15325743576122.jpg)

对于树中节点的结构为：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | struct huffman\_node{  char c;  int weight;  char huffman\_code[LEN];  huffman\_node \* left;  huffman\_node \* right; }; |

对于Huffman树的构建过程为：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 | int huffman\_tree\_create(huffman\_node \*&root, map<char, int> &word){  char line[MAX\_LINE];  vector<huffman\_node \*> huffman\_tree\_node;   map<char, int>::iterator it\_t;  for (it\_t = word.begin(); it\_t != word.end(); it\_t++){  // 为每一个节点申请空间  huffman\_node \*node = (huffman\_node \*)malloc(sizeof(huffman\_node));  node->c = it\_t->first;  node->weight = it\_t->second;  node->left = NULL;  node->right = NULL;  huffman\_tree\_node.push\_back(node);  }    // 开始从叶节点开始构建Huffman树  while (huffman\_tree\_node.size() > 0){  // 按照weight升序排序  sort(huffman\_tree\_node.begin(), huffman\_tree\_node.end(), sort\_by\_weight);  // 取出前两个节点  if (huffman\_tree\_node.size() == 1){// 只有一个根结点  root = huffman\_tree\_node[0];  huffman\_tree\_node.erase(huffman\_tree\_node.begin());  }else{  // 取出前两个  huffman\_node \*node\_1 = huffman\_tree\_node[0];  huffman\_node \*node\_2 = huffman\_tree\_node[1];  // 删除  huffman\_tree\_node.erase(huffman\_tree\_node.begin());  huffman\_tree\_node.erase(huffman\_tree\_node.begin());  // 生成新的节点  huffman\_node \*node = (huffman\_node \*)malloc(sizeof(huffman\_node));  node->weight = node\_1->weight + node\_2->weight;  (node\_1->weight < node\_2->weight)?(node->left=node\_1,node->right=node\_2):(node->left=node\_2,node->right=node\_1);  huffman\_tree\_node.push\_back(node);  }  }   return 0; } |

**Huffman树的特点**

1）权值越大的结点，距离根结点越近。  
2）树中没有度为1的结点，这类树又叫做正则（严格）二叉树。  
3）树的带权路径长度最短

**Huffman编码**

常见的.zip压缩文件和.jpeg图片文件的底层技术都用到了赫夫曼编码。  
例如字符串S=AAABBACCCDEEA

| **A** | **B** | **C** | **D** | **E** |
| --- | --- | --- | --- | --- |
| 5次 | 2次 | 3次 | 1次 | 2次 |

以出现的次数为权值，构建一个赫夫曼树，对每个结点的左右分支进行编号，左0右1，从根到每个结点的路径的数字序列即为每个字符的编码，对A~E的赫夫曼编码规则

| **A** | **B** | **C** | **D** | **E** |
| --- | --- | --- | --- | --- |
| 0 | 110 | 10 | 1110 | 1111 |

则H(S)=00011011001010101110111111110  
上述有赫夫曼树导出每个字符的编码，进而得到整个字符串的编码的过程称为赫夫曼编码。  
在前缀码中，任一字符的编码串都不是另一字符编码串的前缀，赫夫曼编码产生的是最短前缀码。

**赫夫曼n叉树**

赫夫曼二叉树是赫夫曼n叉树的一种特例，当对于结点数目大于等于2的待处理序列，都可以构造赫夫曼二叉树，但却不一定能构建赫夫曼n叉树，但无法构建时。需要补上权值为0的结点让整个序列凑成可以构造赫夫曼n叉树的序列