定义

树是$n(n \geq 0)$个结点的有限集T

- 若$n = 0$，称为空树
- 若$n > 0$，则它满足如下两个条件
  - 有且仅有一个特定的称为根的结点
  - 其余结点可分为$m(m \geq 0)$个互不相交的有限集$T_1, T_2, T_3, \cdots, T_m$，其中每个集合本身又是一棵树

基本术语：

- 结点（node）：包括一个数据元素及若干指向其子树的分支
- 结点的度（degree）：结点拥有的子树个数
- 叶子（leaf）：度为0的结点（或称为终端结点）
- 分支结点（非终端结点）：度不为零的结点
- 树的度：树内各结点度的最大值
- 孩子（child）：结点的子树的根称为该结点的孩子
- 双亲（parents）：（相对孩子）结点的上层结点
- 兄弟（sibling）：同一双亲的孩子之间互称兄弟
- 结点的祖先：从根到该结点所经分支上的所有结点
- 子孙：某结点为根的子树中的任意结点
- 结点的层次（Level）：从根结点算起，根为第一层，它的孩子为第二层
- 深度（depth）：树中结点的最大层次数
- 森林（forest）：$m > (m \geq 0)$颗互不相交的树的集合

ADT（略）

二叉树

是$n(n \geq 0)$个结点的有限集，它或为空树$(n = 0)$，或由一个根结点和两棵分别称为左子树和右子树的互不相交的二叉树构成

特点：

- 每个结点至多有两棵子树（即不存在度大于2的结点）
- 二叉树的子树有左右之分，且其次序不能任意颠倒基本形态

ADT（略）

FxMarkBrown's Blog

性质（[redacted]）：

- 性质1：在二叉树的第i层上至多有2^{i-1}个结点（$i \geq 1$）（数学归纳法）
- 性质2：深度为k的二叉树至多有2^k-1个结点（$k \geq 1$）（等比数列求和）
- 性质3（要会证明）：对任何一棵二叉树T，如果其终端结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$
  设n_1为二叉树中度为1的结点数
  因为二叉树中所有结点的度均小于等于2
  所以其结点总数$n=n_0+n_1+n_2$
  又二叉树中，从下往上看，除根节点外，其余结点都只有一个分支进入
  设B为分支总数，则$n=B+1$（根结点没有分支进入，加上）
  又分支由度为1和度为2的结点射出，$B=n_1+2n_2$
  于是$n=B+1=n_1+2n_2+1=n_0+n_1+n_2$
  得$n_0=n_2+1$

两种特殊形式的二叉树

- 满二叉树
  - 定义：一棵深度为k且有2^k-1个结点的二叉树称为满二叉树
  - 特点：每一层上的结点数都是最大结点数
- 完全二叉树
  - 定义：深度为k，有n个结点的二叉树当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应时，称为完全二叉树
  - 特点：叶子结点只可能在层次最大的两层上出现：对任一结点，若其右分支下子孙的最大层次为l，则其左分支下子孙的最大层次必为l或$l+1$
  - 性质4：具有n个结点的完全二叉树的深度为$floor(log_2 n)+1$
    设其深度为k和完全二叉树的定义有$2^{k-1}-1<n\leq 2^k-1$
    或$2^{k-1}\leq n<2^k$
    于是$k-1\leq \log_2 n<k$
    因为k是整数，所以$k=floor(\log_2 n)+1$
  - 性质5：如果对一颗有n个结点的完全二叉树的结点按层序编号，则对任一结点$i(1\leq i\leq n)$，有：
    1. 如果$i=1$，则结点是二叉树的根，无双亲；如果$i>1$，则其双亲是$floor(\frac{i}{2})$
    2. 如果$2i>n$，则结点无左孩子，如果$2i\leq n$，则其左孩子是$2i$
    3. 如果$2i+1>n$，则结点无右孩子，如果$2i+1\leq n$，则其右孩子是$2i+1$

二叉树的存储结构

顺序存储结构：按从上到下，从左到右编号

FxMarkBrown's Blog

```cpp
#define MAX_TREE_SIZE 100

typedef TElemType SqBiTree [MAX_TREE_SIZE];

// 使用
SqBiTree bt;
```

特点：

- 结点间关系蕴含在其存储位置中
- 浪费空间，适于存满二叉树和完全二叉树，普通二叉树有空的空间

链式存储结构指针域：

- parent：双亲结点（可选）
- data：数据
- lchild：左孩子
- rchild：右孩子

```cpp
typedef struct BiTNode {
   TElemType data;
   struct BiTNode* lchild, rchild;
} BiTNode, *BiTree;
```

特点：

- 指针直接表示关系，操作简单
- 增加指针域，浪费空间，特别是存在多个空指针域

遍历二叉树

遍历二叉树：按某条搜索路径巡访树的每个结点，且使每个顶点仅被访问一次
遍历目的：得到树中的所有结点的一个线性排列
遍历用途：是树结构插入、删除、修改、查找和排序运算的前提，是二叉树一切运算的基础和核心

由二叉树的递归定义知：二叉树由三个基本单元组成

- 根结点（D）
- 左子树（L）
- 右子树（R）

可得到6种遍历方案：DLR、LDR、LRD、DRL、RDL、RLD

FxMarkBrown's Blog

- 先序遍历（DLR，根结点在前）：先访问根结点，然后分别先序遍历左子树、右子树
- 中序遍历（LDR，根结点在中）：先遍历左子树，然后访问根结点，最后中序遍历右子树
- 后序遍历（LRD，根结点在后）：先序遍历左、右子树，然后访问根结点

```c
Status PreOrderTraverse(BiTree T) {
    if(T) {
        Visit(T->data);
        PreOrderTraverse(T->lchild);
        PreOrderTraverse(T->rchild);
    }
}

Status InOrderTraverse(BiTree T) {
    if(T) {
        InOrderTraverse(T->lchild);
        Visit(T->data);
        InOrderTraverse(T->rchild);
    }
}

Status PostOrderTraverse(BiTree T) {
    if(T){
        PostOrderTraverse(T->lchild);
        PostOrderTraverse(T->rchild);
        Visit(T->data);
    }
}
```

中序遍历的非递归算法（用栈模拟）

FxMarkBrown's Blog

```C++
Status InOrderTraverse(BiTree T, Status(* Visit)(TElemType e)) {
    InitStack(S);
    Push(S, T);
    while(!StackEmpty(S)) {
        while(GetTop(S, p) && p) {
            Push(S, p->lchild);
            p = p->lchild;
        }z
        // 上方循环完毕后会把终端结点的lchild(nullptr)也入栈，所以先弹出
        Pop(S, p);
        if(!StackEmpty(S)) {
            Pop(S, p);
            if(!Visit(p->data)) return ERROR;
            Push(S, p->rchild);
        }
    }
}

Status InOrderTraverse(BiTree T, Status (*Visit)(TElemType e)) {
    InitStack(S); p = T;
    While(p || !StackEmpty(S)) {
        if(p) {
            Push(S, p);
            p = p->lchild;
        } else {
            Pop(S, p);
            if(!Visit(p->data)) return ERROR;
            p = p->rchild;
        }

    return OK;
}
```

按先序次序输入二叉树中结点的值（一个字符），空字符表示空树，构造二叉链表表示的二叉树

FxMarkBrown's Blog

```C++
Status CreateBiTree(BiTree &T) {
    scanf(&ch);
    if(ch == '') T = nullptr;
    else{
        if(!(T = (BiTNode*)malloc(sizeof(BiTNode))))
            exit(OVERFLOW);
        T->data = ch;
        CreateBiTree(T->lchild);
        CreateBiTree(T->rchild);
    }
    return OK;
}
```

层次遍历：

```C++
Status Level-OrderTraverse(BiTree T) {
    InitQueue(Q);
    if(!T) {
        EnQueue(Q,T);
        while(!Empty(Q)) {
            DeQueue(Q, p);
            Visit(p->data);
            if(p->lchild) EnQueue(Q, p->lchild);
            if(p->rchild) EnQueue(Q, p->rchild);
        }
    }
```

树与森林（一般情况）

树的双亲表示法

线性表存放树的结点，每个结点含两个域

- 数据域：结点本身信息
- 双亲域：指示结点的双亲结点在数组中的位置

从上到下把所有结点存一遍

特点：找双亲容易，找孩子难

FxMarkBrown's Blog

```C++
#define MAX_TREE_SIZE 100

typedef struct PTNode {
   TElemType data;
   int parent;
} PTNode;

typdef struct {
   PTNode nodes[MAX_TREE_SIZE];
   int n, r; // 总结点数和根的位置
} PTree;
```

求结点$t[i]$的长子

```C++
int FirstChild(PTree t, int i) {
   for(j = i+1; j < t.n; j++)
      if(t.nodes[j].parent == i) return j;
   return -1;
}
```

孩子表示法

桶，数组两个域

- 数据域：结点本身的信息
- 指针域：链表，指向孩子在数组中的位置

```C++
typedef struct CTNode {
   int child;
   struct CTNode* next;
} *ChildPtr;

typedef struct {
   TElemType data;
   ChildPtr firstchild;
} CTBox;

typedef struct {
   CTBox nodes[MAX_TREE_SIZE];
   int n, r; // 总结点数和根的位置
}
```

FxMarkBrown's Blog

孩子兄弟表示法

二叉链表表示，链表中每个结点的两个指针域分别指向其向下的第一个孩子结点和其向右的第一个兄弟结点

```C++
typedef struct CSNode {
    ElemType data;
    struct CSNode* firstchild, nextsibling;
} CSNode, *CSTree;
```

在找一个结点的第i个孩子时，只需要找到此结点的第一个孩子，之后再走$i-1$步即可

森林与二叉树的转换

树和二叉树的转换

- 加线：在兄弟之间加一连线
- 删线：对每个结点，除了其长孩子外，去除其与其余孩子之间的关系
- 旋转：以树的根结点为轴心，将整树顺时针旋转

树转换成的二叉树其右子树一定为空

从树与二叉树的转换可知：任何一棵树和树对应的二叉树，其右子树必为空，若把森林中第二棵树的根结点看成是第一棵树根结点的兄弟，则可以导出森林和二叉树的对应关系：
如果$F=\{T_1,T_2,\cdots,T_m\}$是森林，则可按如下规则转换成一棵二叉树$B=\{root,L_B,R_B\}$

1. 若F非空，即$m=0$，则B为空树
2. 若F非空，即$m\neq 0$，则B的根root即为森林中第一棵树的根root(T_1)；B的左子树L_B是从T_1中根结点的子树森林$F_1=\{T_{11},T_{12},\cdots,T_{1m_1}\}$转换而成的二叉树；其右子树$R_B$是从森林$F'=\{T_2,T_3,\cdots,T_m\}$转换而成的二叉树

方法：

- 将各棵树分别转换成二叉树
- 将每棵树的根结点用线相连
- 以第一棵树根结点为二叉树的根，再以根结点为轴心，顺时针旋转，构成二叉树型结构（在这次的旋转过程中原来森林中各孩子结点位置不变）

树和森林的遍历

先根（序）遍历：
若T非空，则：

FxMarkBrown's Blog

1. 访问根结点R
2. 依次先序遍历根的各子树
   后根（序）遍历

若T非空，则：

1. 依次后续遍历根的各子树
2. 访问根结点R

若将该树转为二叉树，则会发现树的先根序列等于转的二叉树的先序序列，树的后根序列等于转的二叉树的中序序列

按照森林和树的相互递归定义，可推出森林的两种遍历方法

1. 先序遍历森林
   若森林非空，则
   1. 访问森林中第一棵树的根结点
   2. 先序遍历第一棵树中根结点的子树森林
   3. 先序遍历除去第一棵树之后剩余的树构成的森林
2. 中序遍历森林
   若森林非空，则
   1. 中序遍历森林中第一棵树的根结点的子树森林
   2. 访问第一棵树的根结点
   3. 中序遍历除去第一棵树之后剩余的树构成的森林

Huffman树

一些概念

- 带权二叉树：给定一组权$w_1, w_2, \cdots, w_n$，且$w_1 \leq w_2 \leq w_n$，设有一个二叉树，共有n片叶子，分别带权$w_1 w_2 \cdots w_n$，该二叉树称为带权二叉树
- 路径：从树中一个结点到另一个结点之间的分支构成这两个结点间的路径
- 路径长度：路径上的分支数
- 树的路径长度：从树根到每一个结点的路径长度之和
- 权：将树中结点赋给一个有着某种含义的数值，这个数值称为结点的权
- 结点的带权路径长度：该结点到根的路径长度与结点上权的乘积
- 树的带权路径长度：树中所有叶子结点的带全路径长度之和，记作$WPL = \sum_{k=1}^{n} w_k l_k$

设有n个权值$w_1 w_2, \cdots w_n$，构造一棵有n个叶子结点的二叉树，每个叶子的权值为w_i，则WPL最小的那棵二叉树叫做最优二叉树或Huffman树

FxMarkBrown's Blog

构造（掌握）

1. 根据给定的n个权值$\{w_1, w_2, \cdots, w_n\}$构成n棵二叉树的集合$F = \{T_1, T_2, \cdots, T_n\}$，其中每棵二叉树$T_i$中只有一个带权为$w_i$的结点，其左右子树均为空（构造森林全是根）
2. 在F中选取两颗根结点权值最小的树作左右子树，构造一棵新的二叉树，且置新的二叉树结点的权值为其左右子树根结点权值之和（选用两小造新树）
3. 在F中删除这两棵树，同时将新得到的二叉树加入森林中（删除两小添新人）
4. 重新2、3上述两步，直到F只含一棵树为止，这棵树即为Huffman树

Huffan编码

在远程通讯中，要将待传字符串转换成二进制的字符串，若将编码设计为长度不等的二进制编码，即让待传字符串中出现次数较多的字符采用尽可能短的编码，则转换的二进制字符串长度便可减少

另外考虑到诸如0000有多个表示这种情况，所以还必须使任一字符的编码都不是另一个字符编码的前缀

前缀码：给定一个序列的集合，若没有一个序列是另一个序列的前缀，该序列集合称为前缀码
定理：任意一棵二叉树的树叶可对应一个前缀码，任何一个前缀码都对应一棵二叉树

假设每个字符在电文中出现的次数为w_i，其编码长度为l_i，电文中只有n种字符，则电文总长为$\sum_{i=1}^{n} w_i l_i$，对应到二叉树上，若w_i为叶子结点的权，l_i为根到叶子的路径长度。设计电文总长最短的二进制编码，即为以n种字符出现的频率作为权值，设计一棵Huffman树的问题。由此得到的二进制前缀码称为Huffman编码

```C++
typedef struct {
    unsigned int weight;
    unsigned int parent, lchild, rchild;
} HTNode, *HuffmanTree;
typedef char **HuffmanCode;
```

一棵n个叶子结点的Huffman树共有$2n-1$个结点

例如：某通信系统中可能出现8种字符，其概率分别为0.05、0.29、0.07、0.08、0.14、0.23、0.03、0.11，设计Huffman编码
设权：$w = \{5, 29, 7, 8, 14, 23, 3, 11\}$，$n = 8$，则结点数为$2n - 1 = 15$

FxMarkBrown's Blog

C++

```
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int* w, int n) {
    /**
        构造Huffman树
        @param HT: Huffman树
        @param HC: Huffman字符表
        @param w:  权值表
        @param n:  权值个数
    **/
    if(n <= 1) return;
    m = 2*n - 1;
    HT = (HuffmanTree)malloc(
        sizeof(HTNode) * (m+1) // 索引0不用
    ); // 开长度为m+1的Huffman表
    int i;
    for(p = HT, i = 1;i <= n;++i,++p,++w)
        *p = {*w,0,0,0}; // 初始化表中每个结点的值
    for(;i<=m; ++i,++p)
        *p = {0,0,0,0}; // 剩下的先全填0
    for(i=n+1; i <= m; ++i) {
        select(HT, i-1, s1, s2); // 从表中1到i-1中选择权值最小的两个结点，返回索引
        HT[s1].parent = i; HT[s2].parent = i;
        HT[i].lchild = s1; HT[i].rchild = s2;
        HT[i].weight = HT[s1].weight + HT[s2].weight;
    }

    // 之后从叶子结点回溯获得Huffman编码
    HC = (HuffmanCode)malloc(
        sizeof(char*) * (n+1)

    cd = (char*)malloc(n * sizeof(char));
    cd[n-1] = '\0';
    for(i = 1; i<= n; ++i) {
        start = n-1;
        for(c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) {
            if(HT[f].lchild == c) cd[--start] = '0';
            else cd[--start] = '1';
        }

        HC[i] = (char*)malloc(
            (n - start) * sizeof(char) // 实际上是(n-1) - start + 1, 索引相减长度+1等于数
组的长度
        );
        strcpy(HC[i], &cd[start]);
    }
```

FxMarkBrown's Blog

```
    free(cd);
```

FxMarkBrown's Blog