树和二叉树

树是一种非线性结构，特点为：分支关系，一对多，层次结构。

二叉树是度为2的树。

基本概念

1. 结点node：一个数据元素和若干指向子树的分支。
2. 结点的度、树的度degree：结点度是结点所拥有的子树数，树的度是树中结点度的最大值。
3. 叶子节点left、非叶子节点（非终端节点、分支节点）：叶子结点是度为零的结点，非叶子节点反之。
4. 孩子节点（子节点）、双亲结点、兄弟节点：孩子节点是某一结点的子树根，双亲结点为该结点，兄弟结点是来自同一个根的子树根。
5. 层次、堂兄弟结点：从根开始，根为第一层，其孩子为第二层；堂兄弟节点是双亲结点在同一层的所有结点。
6. 层次路径、祖先ancestor、子孙descent：层次节点是从根到某一结点的层次路径，祖先是该路径上所有除自己的结点，子孙结点是以某一点为根的子树中任意节点。
7. 深度depth：树中结点的最大层次值，又为数的高度。
8. 有序树、无序树：有序树是每一个节点的子树有一定次序，否则即为无序树。
9. 森林forest：若干棵互不相交的树集合（若删除一棵树的根节点，子树就构成森林）。

性质

1. 节点个数等于所有节点的度之和+1。
2. 定义m度树的叶子节点数量为n0，节点总数为N，度为1的节点数量为n1...度为m的节点个数为nm:
   - $n_0 = n_2 + 2\times n_3 + \dots + (m-1)\times n_m + 1$
   - $N = n_0 + n_1 + n_2 + \dots + n_m$
     - $= \sum D + 1$
     - $= 0\times n_0 + 1\times n_1 + 2\times n_2 + \dots + m\times n_m + 1$
3. 在非空m度树，第i层至多由$m^{(i-1)}$个结点（$i>=1$）。
4. 高度为h的m度树，最多有$(m^h-1)/(m-1)$个结点：
   - 最多情况下结合性质3，将构成首项为1，公比为m的等比数列，当要求解前h项时：$S_h = a_1(1-m^n) / (1-m) = (m^n-1) / (m-1)$
5. 具有n个结点的m度树，最小高度 = $\lceil \log_m(n\times(m-1)+1) \rceil$：
   1. 当高度一定，每层节点个数越多，其对应的总结点数就越多；反之越少。
   2. 当总节点个数一定，每层节点个数越多，其高度越小；反之越高。

存储结构

双亲表示法（顺序存储）

用顺序存储来保存树的结点，同时每个节点附加一个指示器（整数域）来表示双亲结点的位置（下标值）。

利用了双亲唯一的性质，可以快速找到任意父节点，但子节点需要遍历整个数组。

```c
#define MAX_SIZE 100

typedef struct PTNode{
    int data;
    int parent;     //标记双亲结点
}PTNode;

typedef struct{
    PTNode nodes[MAX_SIZE];
    int root;   //根节点位置
    int num;    //节点数
}PTree;
```

孩子链表示法（链式结构）

每个节点有多个指针域指向对应子树的根节点。有定长节点结构，不定长结构，孩子兄弟表示法。

1. 定长节点结构

结构简单统一，指针域浪费明显。在一颗n个结点的树，度为k的树中必有n(k-1)+1个空指针域。

2. 不定长结构

树中每个结点的指针域数量不同，以此作为该节点的度。没有多余的指针域，但操作不便。

3. 孩子兄弟表示法（最常用）

用二叉链表作为存储结构（故也叫**二叉树表示法**），用两个指针域分别指向第一个子节点和下一个兄弟节点。

```c
typedef struct Csnode{
    ElemType data;
    struct Csnode *firstchild, *nextsibing;
}CSNode;
```

存储

顺序存储

自上而下、自左而右的完全二叉树，完全按照其编号来存储。

对于非完全二叉树，将用空指针填充，使其“成为完全二叉树”，容易造成空间浪费问题，性能较差。

```c
#define MAX_SIZE 100

int SQBTree[MAX_SIZE];
```

链式存储

每个节点包含三个域：数据域，左子节点指针域，右子节点指针域。

```
typedef struct btNode{
    int data;
    struct btNode *LChild, *RChild;
}BTNode;
```

对于三叉链表

额外增加一个指向父节点的指针域

```
typedef struct btNode_3{
    ElemType data;
    struct btNode_3 *LChild, *RChild, *parent;
}BTNode_3;
```

遍历

对二叉树的各个节点进行一次访问，按照先左后右的原则。

所有遍历里，叶子节点的顺序一定不变。

以此分为三种遍历情况：DLR、LDR、LRD，外加一个逐层遍历的层次遍历。

- DLR：先序遍历（根左右）

  ```
  //递归先序遍历
  void PreOrderTraversal(BTNode *root){
      if(root != NULL){
          printf(root->data);
          PreOrderTraversal(root->LChild);
          PreOrderTraversal(root->RChild);
      }
  }
  ```

- LDR：中序遍历（左根右）

  ```
  //递归中序遍历
  void inOrderTraversal(BTNode *root){
      if(root != NULL){
          inOrderTraversal(root->LChild);
          printf(root->data);
          inOrderTraversal(root->RChild);
  ```

```
    }
}
```

- LRD：后序遍历（左右根）

```
//递归后序遍历
void postOrderTraversal(BTNode *root){
    if(root != NULL){
        postOrderTraversal(root->LChild);
        postOrderTraversal(root->RChild);
        printf(root->data);
    }
}
```

- 层次遍历（逐层遍历）
  从根出发，逐层遍历

```
#define MAX_NODE 50
void levelOrder(BTNode *T){
    BTNode *Queue[MAX_NODE], *p = T;
    int front = 0, rear = 0;
    if(p != NULL){
        Queue[rear++] = p;
        while(front < rear){
            p = Queue[++front];
            printf(p->data);
            if(p->LChild != NULL) Queue[rear++] = p->LChild;
            if(p->RChild != NULL) Queue[rear++] = p->RChild;
        }
    }
}
```

二叉树的构造

中序确定左右，先序or后续确定根。如下遍历结果的组合可以唯一确定一棵二叉树：

- 中序遍历 + 层次遍历
- 中序遍历 + 后序遍历
- 中序遍历 + 先序遍历

线索二叉树

一颗二叉树有n个结点，有n-1条边（即指针的连线），有2n个指针域，有n+1个空闲指针域——那么可以利用它们来存放**遍历后**的直接前驱和直接后继的信息：

- 若结点没有左孩子，则LChild指向直接前驱
- 若结点没有右孩子，则RChild指向直接后继

但是对于后序遍历（后序二叉线索树）找直接后继节点依然很困难

树与树之间的转换

遵循先先后中：树的先序对应转化后二叉树的先序；树的后续对应转化后二叉树的中序。

将树转化为二叉树

对于非二叉树，可以将其转换为一颗唯一二叉树。具体方法如下：

1. 逐层遍历，从左往右在兄弟节点之间虚线连接；
2. 随后除了最左的第一个子节点，去除父节点与其它子节点的连线；随后顺时针旋转45°，原有实线左旋；
3. 最后所有虚线改为实线并右斜。

如此转换后，根节点没有右子树；左子树中沿右链往下的右子节点均为原来树中的兄弟结点。

将二叉树转回树

右子节点与父节点连上虚线，去除右子节点的连线，随后即可还原树。

森林转为二叉树【典型考题】

1. 将森林里的每棵树均转为二叉树。
2. 随后从最后一棵二叉树开始，每棵二叉树作为前一棵二叉树的根节点的右子树——
3. ——这样一来，第一棵树的根节点就成为转换后的二叉树根节点。

哈夫曼树Huffman（最优二叉树）

源于哈夫曼编码：一种不等长编码，更为常用的编码更短，不常用的可以用较长的编码——这样使得整段字节的编码量通常不会太长。

- 节点路径：一个节点到另一个结点之间的分支构成该二者间的路径
- 路径长度：节点路径上的分支数目
- 权（值）：各种开销、代价、频度、etc
- 结点的带权路径长度：某节点到根节点间的 路径长度 × 权
- 树的路径长度：树根到每一个结点的路径长度之和
- 树的带权路径长度（WPL）：所有叶子节点的**带权路径长度**之和
- Huffman树：在具有n个叶子节点的二叉树中，WPL值最小的数（让权重较大的结点放置于路径长度较小的位置）

Huffman树构造【高考频】

1. 根据n个权值W构成n棵二叉树集合F（每棵二叉树只有权值为Wi的结点，无左右子树）；
2. 在F中取两棵权值最小的树，作为左右子树，来构成新二叉树，其根节点的权值为左右子树权值之和；
3. 删除这两棵树，并将组合成的新树加入F；
4. 重复2和3，直到剩下一棵树。
5. P.S. 为了规范，权值较小的作为左子树。权值一样则让低的在左，高的在右。

如此操作后，每个字符都是叶子节点，字符不可能出现在路径上——所以每个字符的Huffman编码不可能是另一个字符编码的前缀。

哈夫曼树结论

- 只有0度和2度的结点
- WPL值最小
- 由于左右子树可以交换（规范上权值较小的在左边），哈夫曼树不唯一，但是WPL唯一
- 虽然本质上不属于二叉树，但考试中认为其是二叉树
- 上层结点权值不小于下层节点
- 哈夫曼编码只讨论叶子的编码

_

- n个叶子结点的哈夫曼树共有2n-1个结点
- 判断Huffman编码的步骤：
  1. 先找前缀，若存在编码可以作为其它编码的前缀，则判其不是Huffman
  2. 再按照左0右1的规律还原哈夫曼树，若出现度为1（n1）的结点，则不是Huffman树
- 平均编码长度 = WPL / ∑Wi

并查集【2022新加】

通常用树来表示。

并查集存储在一组不相交集合的动态集合S={S1, S2,...,Sk}，每个集合包含一个或多个元素，并选出某个元素作为代表，不关注其具体包含了何种元素——而关注于可以快速找到指定元素所在的集合，以及合并两个元素所在集合。

并查集的操作

1. makeSet(s); // 建立一个新的并查集，其中包含s个单元素集合
2. unionSet(x, y); // 把元素x和元素y所在集合合并，x和y所在集合不能相交（相交则不进行合并操作）。分为按高度合并（按秩合并）、按节点数量合并（数量较少的树的根其父节点指向节点数较多的树根）
3. find(x); // 找到元素x所在集合的代表，时间复杂度为树的高度。此操作亦可用于判断两个元素是否再同一个集合中，只需要比较集合代表即可