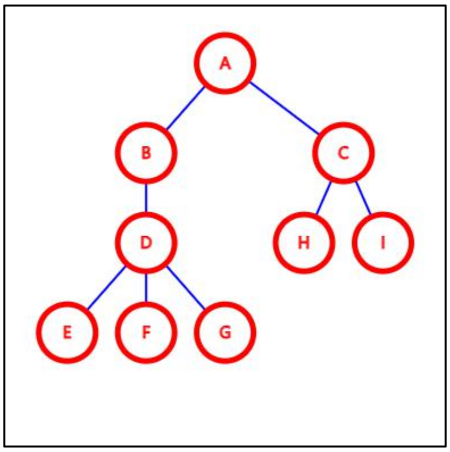
树的基础

1. 树的概念



树形结构是一种重要的**非线性结构**，讨论的是层次和分支关系。树是n个结点的有限集合，在任一棵非空树中：

（1）有且仅有一个称为根的结点；

（2）其余结点可分为m个互不相交的集合，而且这些集合中的每一集合本身又是一棵树，称为根的子树.

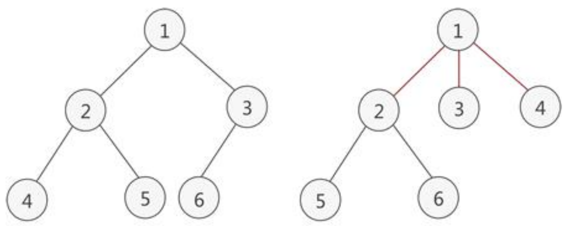
树是递归结构，在树的定义中又用到了树的概念.

2. 树的特点

（1）树中只有根结点没有前趋；
（2）除根外，其余结点都有且仅一个前趋；
（3）树的结点，可以有零个或多个后继；
（4）除根外的其他结点，都存在唯一一条从根到该结点的路径；
（5）树是一种分支结构 （除了一个称为根的结点外）每个元素都有且仅有一个直接前趋，有且仅有零个或多个直接后继.

3. 二叉树

◆ 二叉树的**定义**：二叉树是n（$n \geq 0$）个结点的有限集合，该集合为**空集（称为空二叉树）**，或者**由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树组成**.

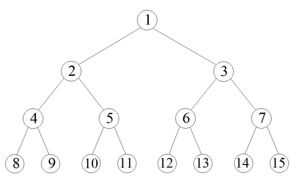


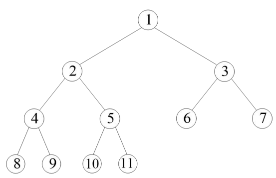
a) 二叉树　　　　b) 非二叉树

◆ 二叉树的**特点**：(1).每个结点至多有二棵子树；
(2).二叉树的子树有左、右之分，且其次序不能任意颠倒.

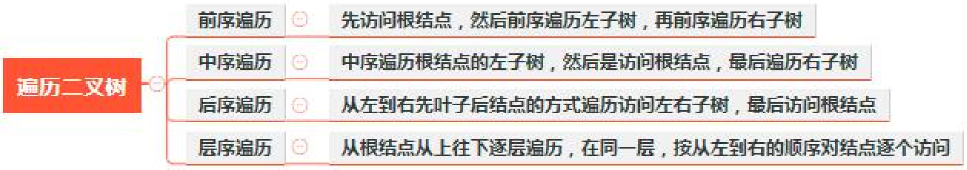
注：二叉树有 5 种形态：空二叉树、只含根结点的二叉树；左子树为空树的二叉树；右子树为空树的二叉树；左右子树均不为空树的二叉树.

如果每一层的结点数都是满的，称为**满二叉树**；如果满二叉树只在最后一层有缺失，并且缺失的编号都在最后，那么称为**完全二叉树**.（左边图为满二叉树，右边图为完全二叉树）

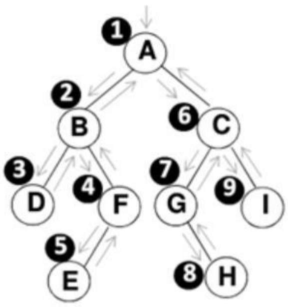




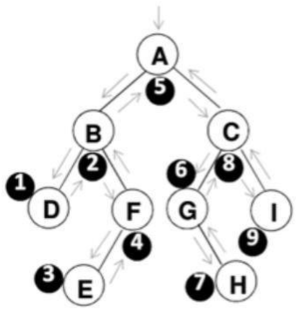
◆ **二叉树的遍历：**二叉树的遍历是指从二叉树的根结点出发，按照某种次序依次访问二叉树中的所有结点，使得每个结点被访问一次，且仅被访问一次.



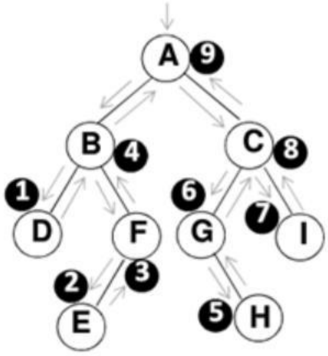
先序遍历（根左右）:A B D F E C G H I



中序遍历（左根右）：D B E F A G H C I



后序遍历（左右根）：D E F B H G I C A



层序遍历（按层）：A B C D F G I E H

```
void PreOrderTraverse(BiTree *T){/*二叉树的前序遍历递归算法*/
    if(T!=NULL){
    printf("%c", T->data);  /*显示结点数据，可以更改为其他对结点操作*/
    PreOrderTraverse(T->lchild);    /*再先序遍历左子树*/
    PreOrderTraverse(T->rchild);    /*最后先序遍历右子树*/
    }
}

void InOrderTraverse(BiTree *T){/*二叉树的中序遍历递归算法*/
    if(T!=NULL){
    InOrderTraverse(T->lchild); /*中序遍历左子树*/
    printf("%c", T->data);  /*显示结点数据，可以更改为其他对结点操作*/
    InOrderTraverse(T->rchild); /*最后中序遍历右子树*/
    }
}
```

```c
void PostOrderTraverse(BiTree *T){/*二叉树的后序遍历递归算法*/
    if(T!=NULL){
    PostOrderTraverse(T->lchild);   /*先后序遍历左子树*/
    PostOrderTraverse(T->rchild);   /*再后序遍历右子树*/
    printf("%c", T->data);  /*显示结点数据，可以更改为其他对结点操作*/
    }
}
```

拓展：已知二叉树的“中序遍历+先序遍历”或者“中序遍历+后序遍历”，都能确定一棵树.

◆ 二叉树的**建立**：

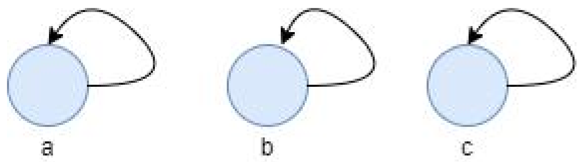
```c
typedef struct node{
    char data;
    struct node *lchild;
    struct node *rchild;
}BiTree;

BiTree *createTree(){
    char ch;
    BiTree *treenode;
    scanf("%c",&ch);
    if(ch=='#')//当前结点为空
        treenode=NULL;
    else{
        treenode=(BiTree*)malloc(sizeof(BiTree));
        treenode->data=ch;
        treenode->lchild=createTree();
        treenode->rchild=createTree();
    }
    return treenode;
}
```

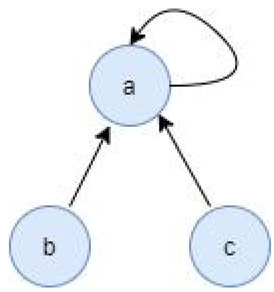
并查集

并查集是一种树形的数据结构，顾名思义，它用于处理一些不交集的**合并**及**查询**问题.它支持两种操作：

- 查找（find）：确定某个元素处于哪个子集；
- 合并（union/join）：将两个子集合并成一个集合.

基本原理：并查集的重要思想在于**用集合中的一个元素代表集合**。并查集由一群集合构成，最开始时所有元素各自单独构成一个集合。比如，有一批元素 $arr[]=\{a,b,c,d,e\}$，我们需要将这一批元素初始化成单个元素的集合，即 a 单独构成一个集合，b 单独构成一个集合。其中并查集中的单个集合结构如下所示：





当集合中只有一个元素时，这个元素的父节点为自己，也就是这个集合的代表节点。当一个集合有多个节点时，代表节点为集合中父节点指向其自己的节点，比如右图的节点 a。

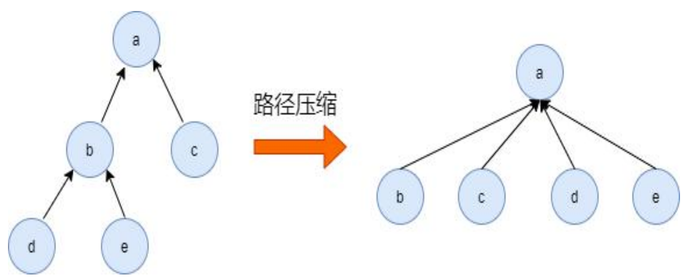
初始化：

```
int fa[MAXN];

void init(int n){
    for(int i=1;i<=n;i++)
        fa[i]=i;
}
```

查找：

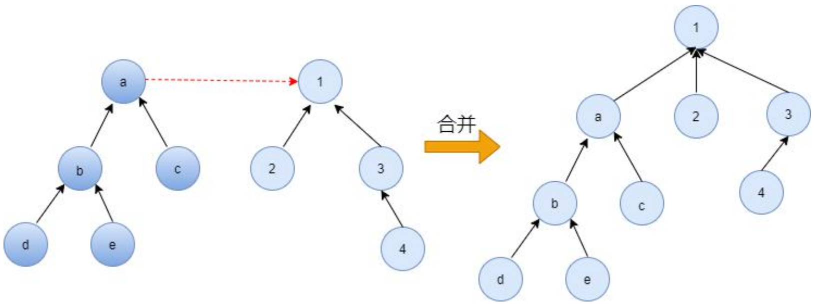
```
int find(int x){//寻找x的祖先
    if(fa[x]==x)     //如果x是祖先则返回
        return x;
    else return find(fa[x]); //如果不是则问x的父结点
}
```

但是由于一层一层的询问，会导致效率较低，此时应该让找到的点成为 x 的父结点(不必意原来父结点的关系)，故采用**路径压缩(把在路径上的每个节点都直接连接到根上，这就是路径压缩)**.



```
int find(int x){//寻找 x 的祖先
    if(fa[x]!=x) //x 不是自身的父亲，即 x 不是该集合的代表
        fa[x]=find(fa[x]); //查找 x 的祖先直到找到代表,于是顺手路径压缩
    return fa[x];
}
```

合并：




```
void join(int x,int y){//x 与 y 合并
    int fx=find(x),fy=find(y);//寻找 x,y 的祖先
    if(fx!=fy)//如果不等则任取一个作为另一个的前驱
        fa[fx]=fy;//fa[fy]=fx;
}
```