Day 4

树和二叉树存储结构

树型结构是以分支关系定义的层次结构。树形结构在客观世界中广泛存在，如人类的家庭族谱及各种社会组织机构。又如计算机文件管理和信息组织也用到树形结构。

树(tree)是由$n(n\geq 0)$个结点组成的有限集合T。n=0的树称为空树；对$n>0$的树，有:

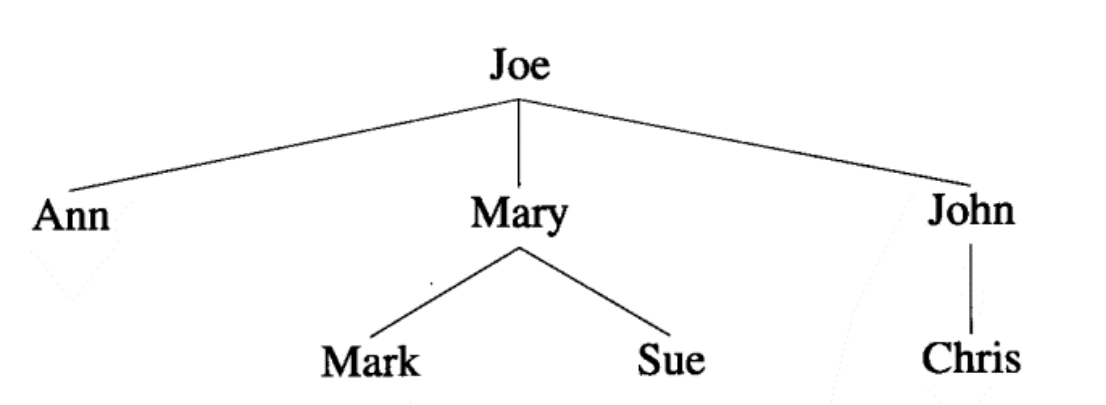
(1)仅有一个特殊的结点称为根(root)结点，根结点没有前驱结点；

(2)当$n>1$时，除根结点外其余的结点分为$m(m>0)$个互不相交的有限集合T_1,T_2，…，T_m，其中每个集合T_i本身又是一棵树，称之为根的子树（ subtree）。

注：树的定义具有递归性，即“树中还有树”。仅有一个根结点的树是最小树，

树t是一个非空的有限元素的集合，其中一个元素为根，余下的元素组成t的子树。

在画一棵树时，每个元素都代表一个节点。树根在上面，其子树画在下面。



如图所示，其中：

Ann，Mary,John是Joe的孩子(children)

而Joe是他们的父母(parent)

有相同父母的孩子是兄弟(sibling)。Ann，Mary,John都是兄弟。

此外，还有其他术语：孙子(grandchild),祖父(grandparent),祖先(ancestor),后代(descendent)等。

树中没有孩子的元素称为叶子(leaf)。图中Ann,Mark,Sue和Chris是树的叶子。

此处输入图片的描述树的另一个常用术语是级(level)。指定树根的级是1，其孩子的级是2，依次类推。上图中Joe的级是1，而Ann，Mary,John的级是2，然后Mark,Sue,Chris的级是3。

元素的度是指其孩子的个数。叶节点的度是0。树的度是其元素度的最大值。所以图中的度是3。

二叉树

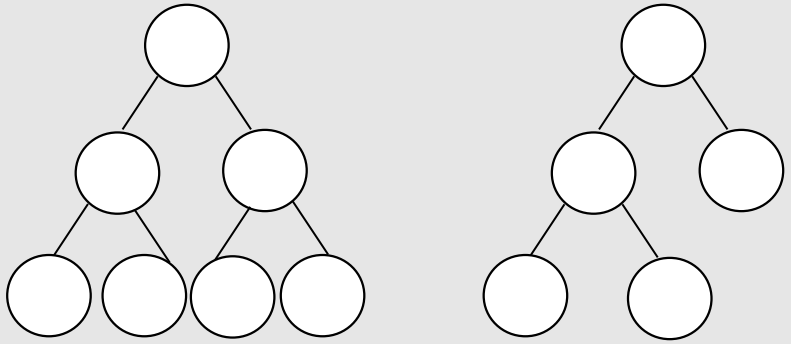
二叉树(binary tree)T是有限个元素的集合（可以为空）。当二叉树非空时，其中有一个称为根的元素，余下的元素（如果有的话）被组成2个二叉树，分别称为T的左子树和右子树。

二叉树种类：

完美二叉树（国内的满二叉树）：在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上，这样的二叉树称为完美二叉树。(如左图)

完全二叉树：如果一棵深度为k，有n个结点的二叉树中各结点能够与深度为k的顺序编号的满二叉树从1到n标号的结点相对应的二叉树称为完全二叉树。(如右图)

完满二叉树（国际的满二叉树）：在一棵二叉树中，所有非叶子结点的度都是2。 (如右图)



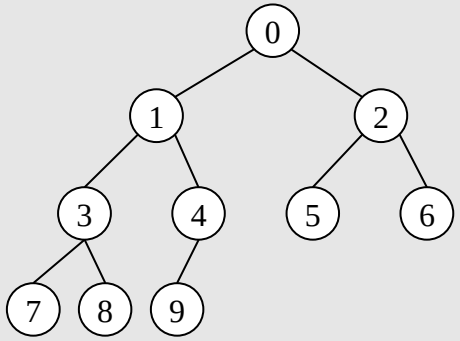
二叉树的链式存储

```
struct node
{
    int value;
    node *leftchild, *rightchild;
    int id;      // 结点编号
    node *parent;        // 指向父亲结点
} arr[N];
int top=-1;
node * head = NULL;

#define NEW(p)  p=&arr[++top]; p->leftchild=NULL; p->rightchild=NULL; p->value=0
```

二叉树的链式存储

```
struct node
{
    int value;
    int leftchild, rightchild;
    int id;      // 结点编号
    int parent;        // 指向父亲结点
} arr[N];
int top = -1;
int head= -1;

#define NEW(p)  p=&arr[++top]; p->leftchild=-
1; p->rightchild=-1; p->value=0
```

完全二叉树
一维数组存储法



如果一个二叉树的结点严格按照从上到下、从左到右的顺序填充，就可以用一个一维数组保存。

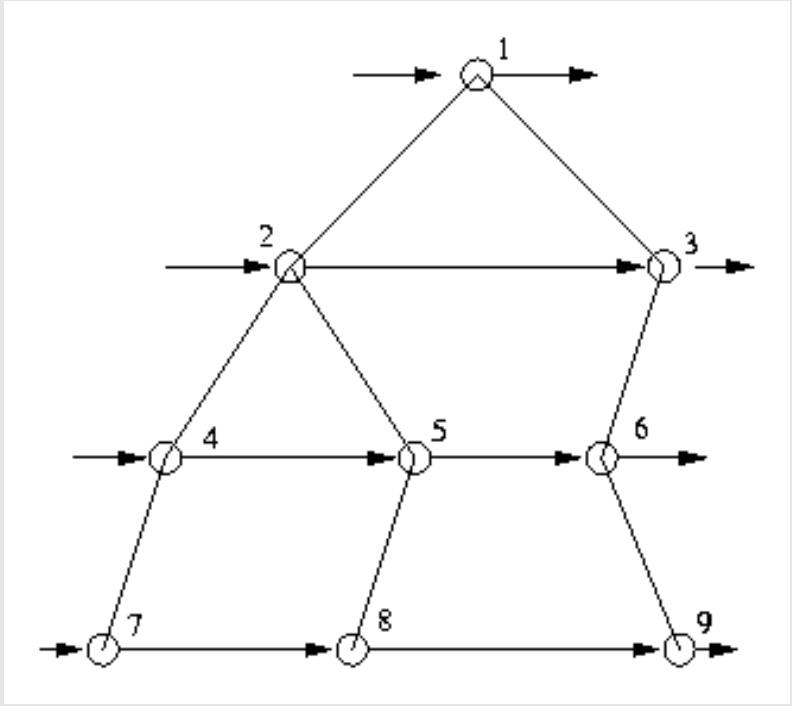
下面假设这个树有n个结点，待操作的结点是r（$0 \le r < n$）。

操作	宏定义	r 的取值范围
r 的父亲	`#define parent(r) (((r)-1)/2)`	$r \neq 0$
r 的左儿子	`#define leftchild(r) ((r)*2+1)`	$2r+1 < n$
r 的右儿子	`#define rightchild(r) ((r)*2+2)`	$2r+2 < n$
r 的左兄弟	`#define leftsibling(r) ((r)-1)`	r 为偶数 且 $0 < r \leqslant n-1$
r 的右兄弟	`#define rightsibling(r) ((r)+1)`	r 为奇数且 $r+1 < n$
判断 r 是否为叶子	`#define isleaf(r) ((r)>=n/2)`	$r < n$

二叉树的层次遍历

广度优先遍历（或宽度优先遍历，BFS）

首先访问根结点，然后逐个访问第一层的结点，接下来逐个访问第二层的结点



```
node *q[N];
void BFS(node *p)
{
    if (p==NULL) return;
    int front=1,rear=2;
    q[1]=p;
    while (front<rear)
    {
        node *t = q[front++];
        // 处理结点t
        cout<<t->value<<' ';
        if (t->leftchild!=NULL) q[rear++]=t->leftchild;
        if (t->rightchild!=NULL) q[rear++]=t->rightchild;
    }
}
```

对于完全二叉树，可以直接遍历：

```
for (int i=0; i<n; i++) cout<<a[i]<<' ';
```

深度优先遍历（DFS）

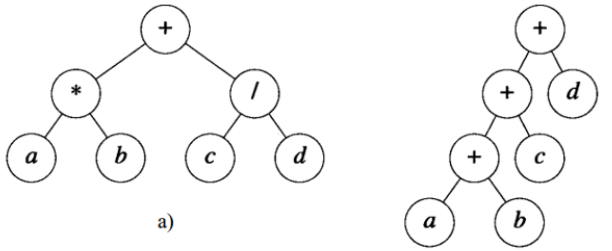
对于二叉树T，可以递归定义它的先序遍历、中序遍历和后序遍历，如下所示：

PreOrder(T)=T的根结点+PreOrder(T的左子树)+PreOrder(T的右子树)

InOrder(T)=InOrder(T的左子树)+T的根结点+InOrder(T的右子树)

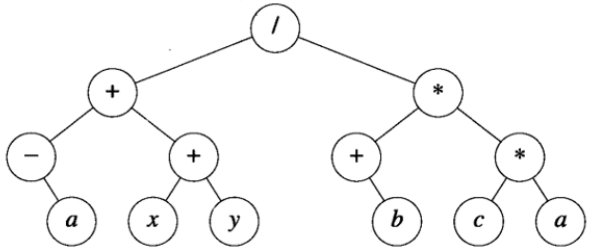
PostOrder(T)=PostOrder(T的左子树)+PostOrder(T的右子树)+T的根结点

图中给出了表示数学表达式的二叉树，总共有3个数学表达式。每个操作符可以有一个或两个操作数，左操作数是操作符的左子树，而右操作数则是右子树。树中的叶节点是常量或者变量。



a)

b)



c)

数学表达式树

a) $(a * b) + (c / d)$ b) $((a + b) + c) + d)$ c) $((-a) + (x + y)) / ((+b) * (c * a))$

对这些二叉树分别进行前序、中序、后序遍历，就可以得到该表达式的前缀、中缀、后缀表达式。例如,(a*b)+(c/d)的前缀表达式是+*ab/cd

	a)	b)	c)
前序	*+*ab/cd*	*+++abcd*	*/+-a+xy*+b*ca*
中序	*a*b+c/d*	*a+b+c+d*	*−a+x+y/+b*c*a*
后序	*ab*cd/+*	*ab+c+d+*	*a−xy++b+ca**/*

二叉树按前序，中序，后序遍历的结果

前序遍历

```
void preorder(node *p)
{
    if (p==NULL) return;
    // 处理结点p
    cout<<p->value<<' ';
    preorder(p->leftchild);
    preorder(p->rightchild);
}
```

中序遍历

```
void inorder(node *p)
{
    if (p==NULL) return;
    inorder(p->leftchild);
    // 处理结点p
    cout<<p->value<<' ';
    inorder(p->rightchild);
}
```

后序遍历

假如二叉树是通过动态内存分配建立起来的，在释放内存空间时应该使用后序遍历。

```
void postorder(node *p)
{
    if (p==NULL) return;
    postorder(p->leftchild);
    postorder(p->rightchild);
    // 处理结点p
    cout<<p->value<<' ';
}
```

二叉树的重建

二叉树的遍历方式有三种：前序遍历、中序遍历和后序遍历。现在给出其中两种遍历的结果，请输出第三种遍历的结果。

原理

前序遍历的第一个元素是根，后序遍历的最后一个元素也是根。所以处理时需要到中序遍历中找根，然后递归求出树。

注意！输出之前须保证字符串的最后一个字符是'\0'。

中序+后序→前序

```
void preorder(int n, char *pre, char *in, char *post)
{
    if (n<=0) return;
    int p=strchr(in, post[n-1])-in;
    pre[0]=post[n-1];
    preorder(p, pre+1, in, post);
    preorder(n-p-1, pre+p+1, in+p+1, post+p);
}
```

前序 + 中序→后序

```
void postorder(int n, char *pre, char *in, char *post)
{
    if (n<=0) return;
    int p=strchr(in, pre[0])-in;
    postorder(p, pre+1, in, post);
    postorder(n-p-1, pre+p+1, in+p+1, post+p);
    post[n-1]=pre[0];
}
```

中序 + 后序→前序

“中 + 前”和“中 + 后”都能产生唯一解，但是“前 + 后”有多组解。

树的直径

定义：最远的两个点的路径

一种方便的求法：任取一个点A

$O(n)$（DFS)求出距离A最远的某个点B

$O(n)$求出距离B最远的某个点C

则B与C之间的路径是一条直径

直径可能不唯一，但直径的长度唯一

树的中心

直径的中点

可能在节点上，也可能在边上

求出直径就能求中心了

注意边权不为1的情况

中心是唯一的

树的重心

对于树上的每一个点，在其所有子树中最大的子树节点数中，值最小的所在的点就是这棵树的重心。

（这里以及下文中的“子树”都是指无根树的子树，即包括“向上”的那棵子树，并且不包括整棵树自身。）

```cpp
void getCentroid(int u, int fa) {
  siz[u] = 1;
  wt[u] = 0;
  for (int i = head[u]; ~i; i = nxt[i]) {
    int v = to[i];
    if (v != fa) {
      getCentroid(v, u);
      siz[u] += siz[v];
      wt[u] = max(wt[u], siz[v]);
    }
  }
  wt[u] = max(wt[u], n - siz[u]);
  if (rt == 0 || wt[u] < wt[rt]) rt = u;  // rt 为重心编号
}
```

在 DFS 中计算每个子树的大小，记录“向下”的子树的最大大小，利用总点数 - 当前子树（这里的子树指有根树的子树）的大小得到“向上”的子树的大小，然后就可以依据定义找到重心了。

性质

以树的重心为根时，所有子树的大小都不超过整棵树大小的一半。

树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。

把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上。

在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离。

并查集

并查集最擅长做的事情——将两个元素合并到同一集合、判断两个元素是否在同一集合中。

并查集用到了树的父结点表示法。在并查集中，每个元素都保存自己的父亲结点的编号，如果自己就是根结点，那么父亲结点就是自己。这样就可以用树形结构把在同一集合的点连接到一起了。

```
struct node
{
    int parent; // 表示父亲结点。当编号i==parent时为根结点。
    int count;  // 当且仅当为根结点时有意义：表示自己及子树元素的个数
    int value;   // 结点的值
} set[N];

    int Find(int x){ // 查找算法的递归版本（建议不用这个）
        return (set[x].parent==x) ? x : (set[x].parent = Find(set[x].parent));
    }



    int Find(int x){     // 查找算法的非递归版本
        int y=x;
        while (set[y].parent != y)    // 寻找父亲结点
            y = set[y].parent;
        while (x!=y){      // 路径压缩，即把途中经过的结点的父亲全部改成y。
            int temp = set[x].parent;
            set[x].parent = y;
            x = temp;
        }
        return y;
    }
```

Find(x)：寻找*x*所在树的根结点。Find的时候，顺便进行了路径压缩。
上面的Find有两个版本，一个是递归的，另一个是非递归的。

```
void Union(int x, int y) {  // 小写的union是关键字。
    x=Find(x); y=Find(y);       // 寻找各自的根结点
    if (x==y) return;   // 如果不在同一个集合，合并
    if (set[x].count > set[y].count) {
// 启发式合并，使树的高度尽量小一些
        set[y].parent = x;
        set[x].count += set[y].count;
    }
    else {
        set[x].parent = y;
        set[y].count += set[x].count;
    }
}
```

Union(x,y)：把*x*和*y*进行启发式合并，即让节点数比较多的那棵树作为“树根”，以降低层次。

```
void Init(int cnt) {    // 初始化并查集，cnt是元素个数
    for (int i=1; i<=cnt; i++) {
        set[i].parent=i;
        set[i].count=1;
        set[i].value=0;
    }
}
```

```
void compress(int cnt) { // 合并结束，再进行一次路径压缩
    for (int i=1; i<=cnt; i++) Find(i);
}
```

使用之前调用Init()

判断*x*和*y*是否在同一集合：if (Find(x)==Find(y)) ……

在所有的合并操作结束后，应该执行compress()。

并查集的效率很高，执行*m*次查找的时间约为O(5*m*)。