## 树

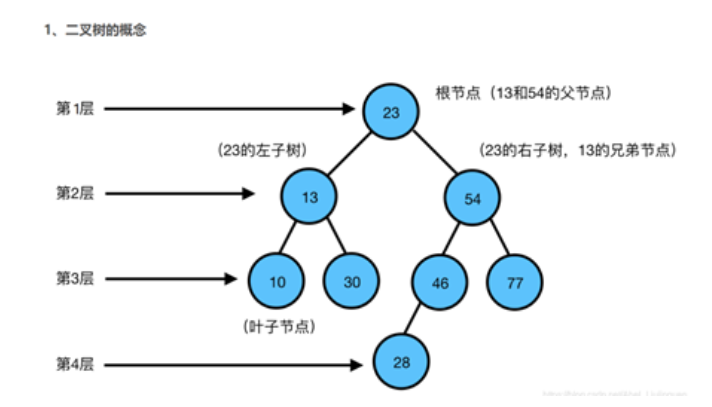
### 树的定义

一棵树是由n(n>0)个元素组成的有限集合，其中：

(1)每个元素称为结点(node)

(2)有一个特定的结点，称为根结点或树根(root)

(3)除根结点外，其余结点能分成m(m>=0)个互不相交的有限集合T0,T1,T2…… Tm-1。其中的每个子集又都是一棵树，这些集合称为这颗树的子树。

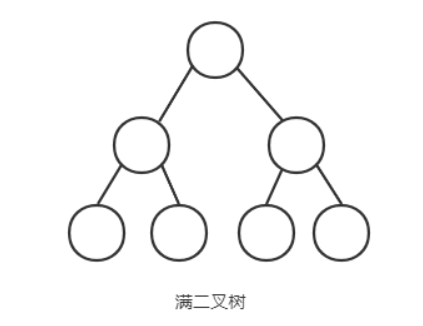
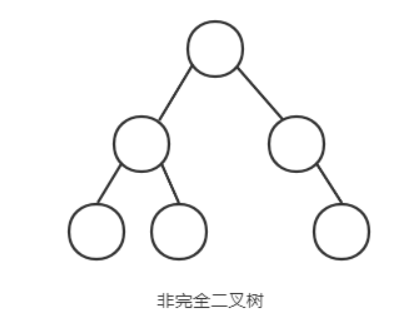
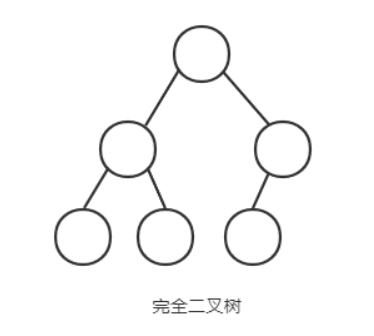


### **树的种类**

****二叉树****：每个节点最多含有两个子树的树称为二叉树；

****满二叉树****：叶节点除外的所有节点均含有两个子树的树被称为满二叉树

****完全二叉树****：二叉树除去最后一层为满二叉树，且最后一层的结点依次从左到右分布，这样的二叉树称为完全二叉树

### **二叉树的性质**

****性质1：****在非空二叉树的第i层做多有**2i-1**节点。

****性质2：****深度为k的二叉树最多有**2k -1**个节点。

****性质3：****对于任意一棵二叉树，如果度为0的节点个数为n0。度为2的节点数为n2。则**n0=n2+1**

****性质4:****具有n个节点的完全二叉树的深度**k=floor(log2n)+1**

****性质5:****对于含n个节点的完全二叉树中编号为i(1<=i<=n)的节点。

* 如果i=1,则i节点是这可完全二叉树的根，没有双亲。否则其双亲编号为i/2
* 如果2i>n,则i节点没有左孩子，否则其左孩子的编号为2i
* 如果2i+1>n,则i节点没有右孩子，否则其右孩子的编号为2i+1

补充性质1：**叶子结点数=度为2的结点+1**

叶子结点数=总结点数/2（慎用）

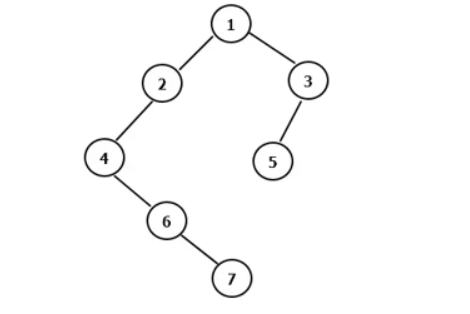
补充性质2：**结点总数=n1+n2+n0** ( n1 是度为1的点，n2是度为2的点，n0是度为0点)

补充性质3：对于一棵完全二叉树来说，n1（度为1的点）只有1或0

补充性质4：**结点总数=度数 \* 该度数对应的结点数 + 1**

### **二叉树的遍历**

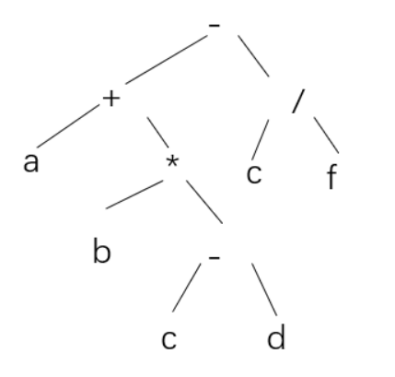
遍历表达法是有3种，先序遍历、中序遍历、后序遍历。



**先序遍历（1246735）**：对于当前节点，先输出该节点，然后输出他的左孩子，最后输出他的右孩子。

**中序遍历（4672153）**：对于当前结点，先输出它的左孩子，然后输出该结点，最后输出它的右孩子。以上图为例：

**后序遍历（7642531）**：对于当前结点，先输出它的左孩子，然后输出它的右孩子，最后输出该结点。



一棵二叉树的先序遍历的结果就是前缀表达式(****波兰式****)

一棵二叉树中序遍历的结果就是中缀表达式

一个二叉树后序遍历的结果就是后缀表达式(****逆波兰式****)

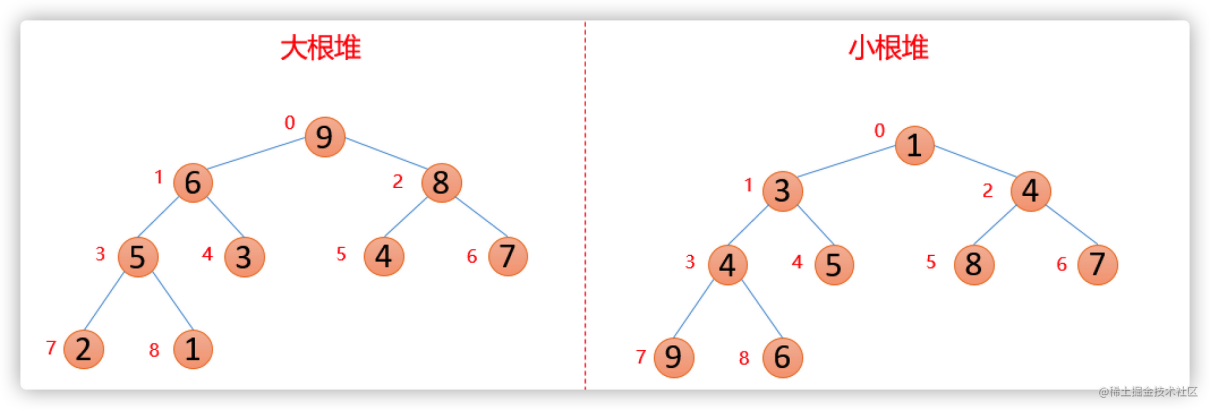
结论：

1. 已知**前序序列和中序序列**可以确定二叉树
2. 已知**中序序列和后序序列**也可以确定出二叉树
3. 已知前序序列和后序序列不能确定二叉树

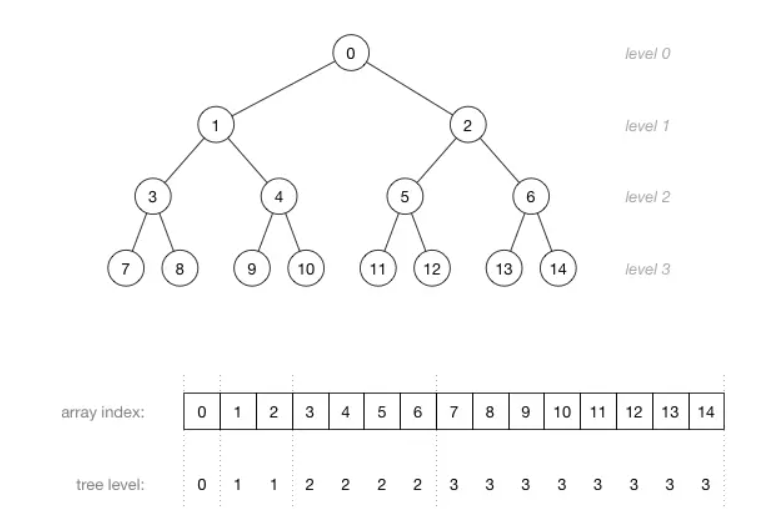
### **堆的概念**

堆就是用数组实现的二叉树。

堆分为两种：**大根堆和小根堆**，两者的差别在于节点的排序方式。在大根堆中，父节点的值比每一个子节点的值都要大。在小根堆中，父节点的值比每一个子节点的值都要小。这就是所谓的“堆属性”，并且这个属性对堆中的每一个节点都成立。



堆的存储：

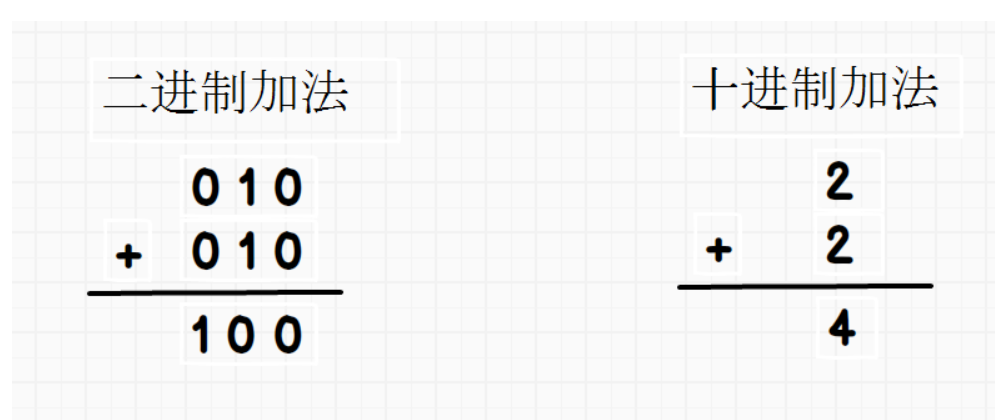


### 习题：信奥一本通434-435

## **二进制**

### **二进制由来**

二进制是计算技术中广泛采用的一种数制。二进制数据是用0和1两个数码来表示的数。它的基数为2，进位规则是“**逢二进一**”，借位规则是“**借一当二**”，由18世纪德国数理哲学大师莱布尼兹发现。



参照二进制的进位规律，十进制1~16的二进制，八进制，十六进制表示如下：

| **十进制** | **二进制** | **八进制** | **十六进制** |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |
| 16 | 10000 | 20 | 10 |

注意： 十六进制由于只有数字0-9，所以10-15用大写或小写的A-F代替，并且逢F进1

### **位和字节**

（1）什么是位？

来自英文 bit，音译为“比特”，表示二进制位。位是计算机内部数据储存的最小单位，11010100是一个 8 位二进制数。一个二进制位只可以表示 0 和 1 两种状态；两个二进制位可以表示 00、01、10、11 四种状态；三位二进制数可表示八种状态……，也就是其转换为10进制数的值，也就是其取值范围。

举个例子：

| **第几位** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 数据 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

一个8bit的空间，取值范围就是0~28-1=0-255

一个16bit的空间，取值范围就是0~216-1=0-65535

（2）什么是字节?

字节来自英文 Byte，音译为“拜特”，习惯上用大写的“B”表示。

字节是计算机中数据处理的基本单位。计算机中以字节为单位存储和解释信息，规定一个字节由八个二进制位构成，即 1 个字节等于 8 个比特（1Byte=8bit）。

### **二进制的运算**

**（1）加法**

当某位满2就向高位进1

0+0=0，0+1=1，1+0=1，1+1=10(向高位进位)；

例如 3+1=0011+0001=0100 5+2=0101+0010=0111

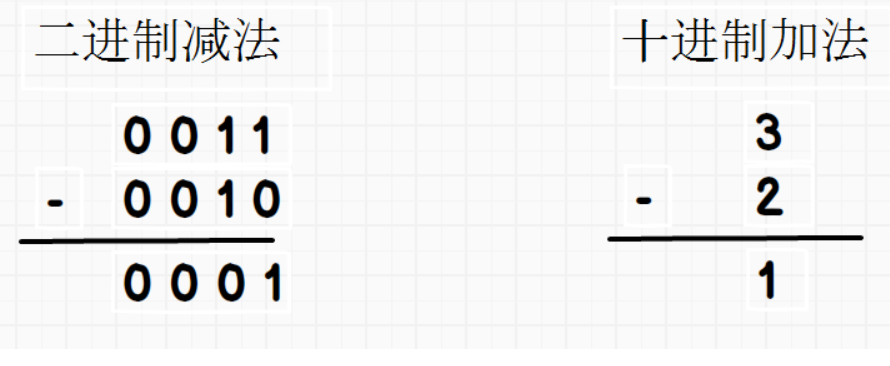


1. **减法**

当某位不足时向高位借1

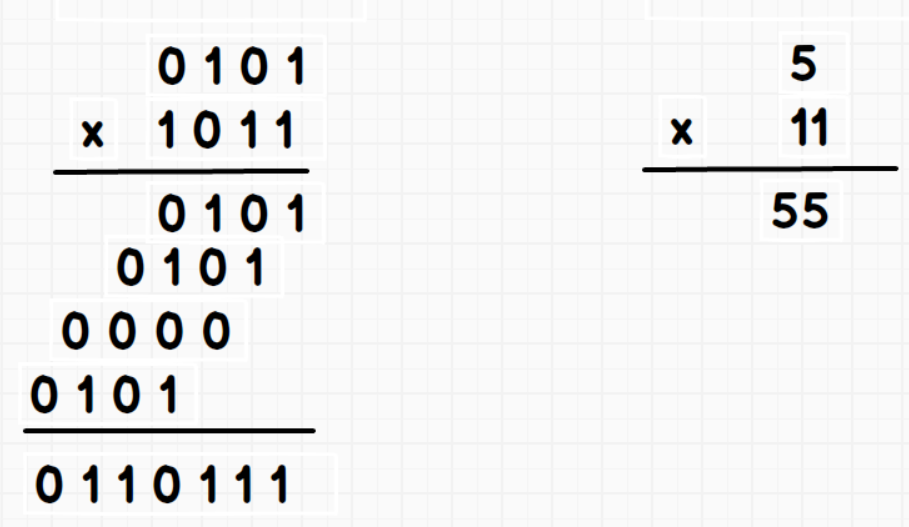
0-0=0， 0-1=1(向高位借位) 1-0=1， 1-1=0 (模二加运算或异或运算) ；

例如 3-1=0011-0001=0010 5-2=0101-0010=0011



1. 乘法

与10进制乘法类似，当某位满2就向高位进1



### **位运算**

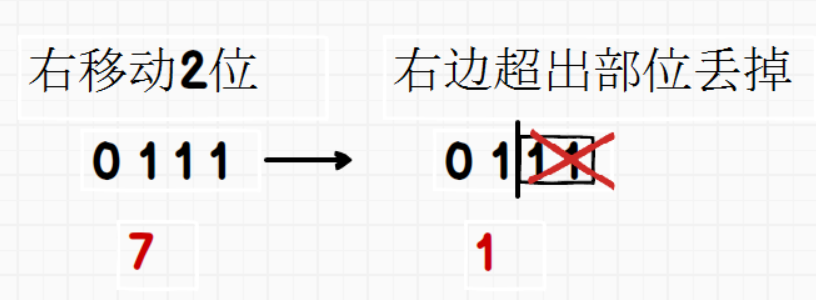
1. 左移<<

左移运算的符号为<<，通常是右边空位补0



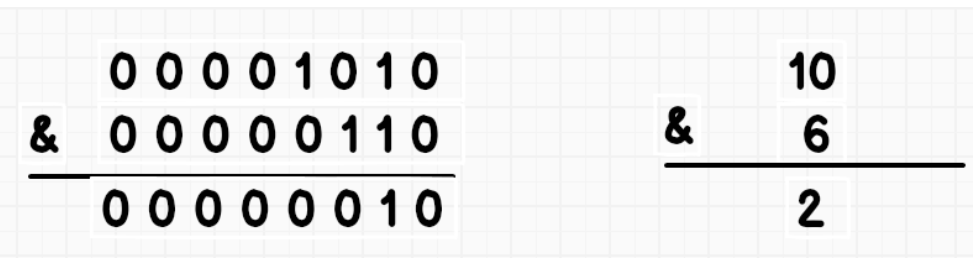
1. 右移>>

右移运算是将整个二进制数右移，超出的部位直接丢掉



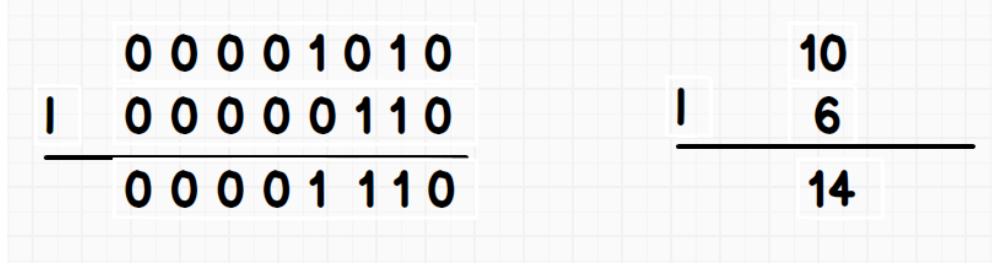
1. 与 &

参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0



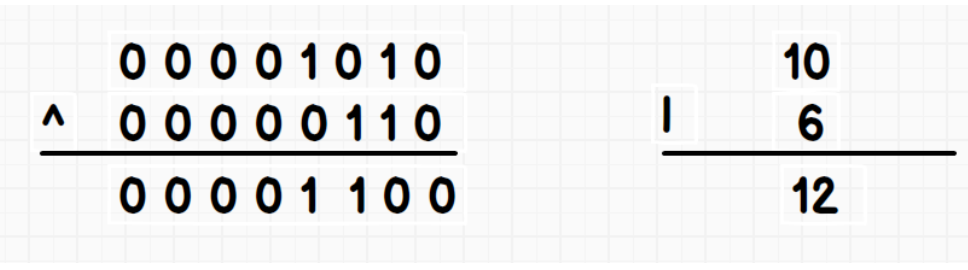
1. 或 |

参与运算的两个值,如果两个相应位只要有一个为1,则该位的结果为1,只有两个相应位都为0才为0。



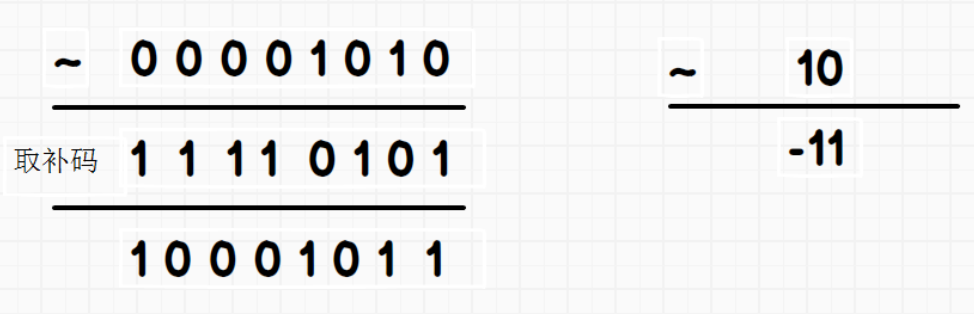
1. ^

当两对应的二进位相异时，结果为1



1. 取反 ~

对数据的每个二进制位取反,即把1变为0,把0变为1



注意：10取反后为11110101，这明显是一个负数，负数一般是以补码形式存储的，所以要还原为原码，需要再取反，为1 0 0 0 1 0 1 0，再加1得1 0 0 0 1 0 1 0，结果为-11。

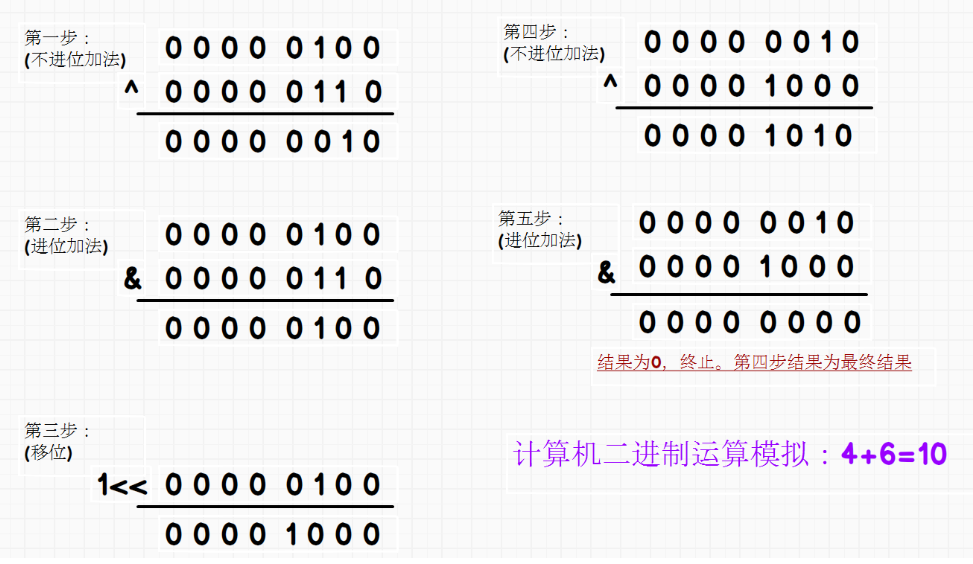
**为什么要设置位运算？**

计算机只会存储1或0，是不会自动进位的，进位只有移位，所以加法运算的过程如下：

计算机计算二进制加法是分四步：

1. 将两个加数转换为二进制数，计算两个加数不需要进位的和，得出的结果。
2. 将两个加数进行与运算。
3. 利用与运算得到结果进行左移运算,得出结果。
4. 将或异运算的结果和左移运算的结果作为两个新的加数，重复此操作。直到当与运算的结果为0，则异或运算的结果则为两个加数的和所对应的二进制数。

具体的过程如下：



### **原码、反码、补码**

（1）原码

原码就是符号位加上真值的绝对值, 即用第1位表示符号, 其余位表示值. 比如如果是8位二进制:

[+1]原 = 0000 0001

[-1]原 = 1000 0001

注意：符号位为0则表示正数，为1则表示负数

因为第一位是符号位, 所以8位二进制数的取值范围就是:

[1111 1111 , 0111 1111] 即 [-127 , 127]

1. 反码

反码的原则：

1. 正数的反码是其本身；
2. 负数的反码是在其原码的基础上, 符号位不变，其余各个位取反。

[+1] = [00000001]原 = [00000001]反

[-1] = [10000001]原 = [11111110]反

1. 补码：

补码的表示方法是:

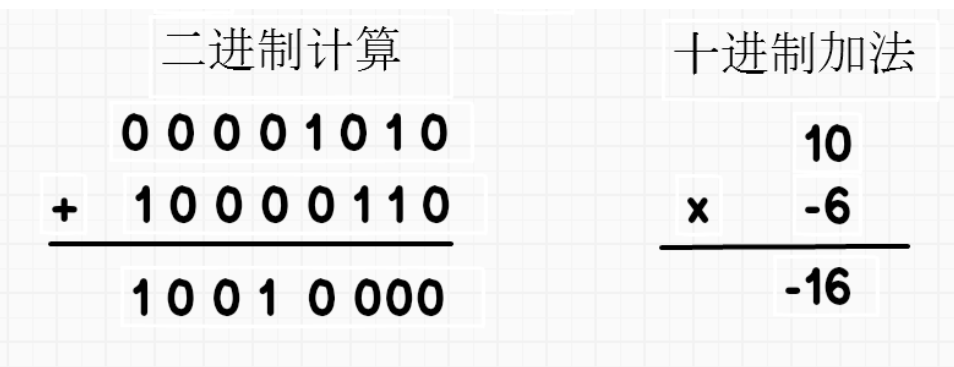
1. 正数的补码就是其本身
2. 负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1. (即在反码的基础上+1)

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

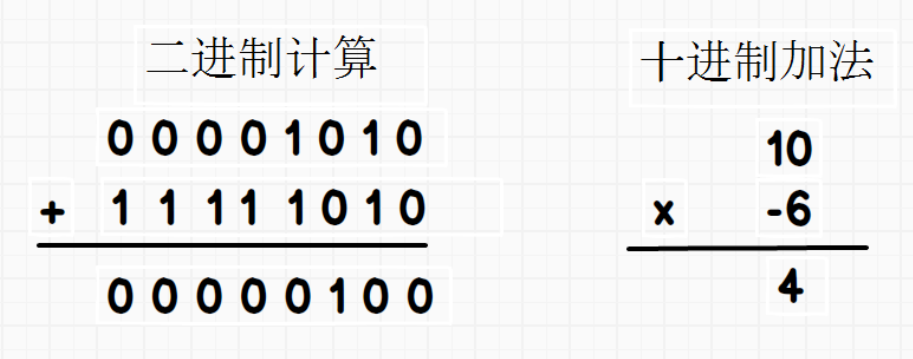
**为什么要设置反码与补码？**主要为了提高计计算机运行的效率，例如：

用原码进行计算一个正数与一个负数相加如下（固定8位）：



可以看到结果明显是错的，想象一下真实的十进制计算，计算机如果还要去判断符号位，那就增加了计算负担。

用补码计算一个正数与一个负数相加如下（固定8位）：



再把（0 0 0 0 1 0 0）转换为原码(正数的补码为本身)，结果为4。这样计算机每次运算就不用判断正负号了。注意：**负数参与的计算都要转换为补码形式进行运算**

## 3.抽样

### **3.1 有编号与无编号**

有编号：

3个不一样的苹果，先后放入三个盘子中，共有3\*2\*1=6种放法

分别为 123 132 213 231 312 321

无编号：

3个相同的苹果，先后放入三个盘子中，共有1种放法



解法：

将两红球看成不同的球，先放两红球，再放蓝球，再放白球，放法：10\*9\*8\*7

由于两红球没有编号，故要除去重复的个数，最终：10\*9\*8\*7/2=2520

### **3.2 绑定**

将连在一起的元素算成一个进行排序，最后再对连在一起的元素单独处理



解法：先将小明小华绑定在一起算一个人，放在圈中，只有一种放法，再放其他人有4\*3\*2，由于小明小华之的相互位置算不同，因此最终结果：4\*3\*2\*2

3.3 放入环与放与线

环可以旋转，因此放一个元素只有一种放法。

由于线有前后顺序，因此放一个元素有多种放法。

### **3.4 分而治之**

将复杂的问题分解为简单熟悉的问题



将复杂问题简单化，分为以下几种情况：

两个空盘：

7，1种

一个空盘：

25，34，16 3种

无空盘：

124，133，115，223，4种

### **3.5插空**

如要一些数不能排在一起，先其他数，再将这些数进行插空排列



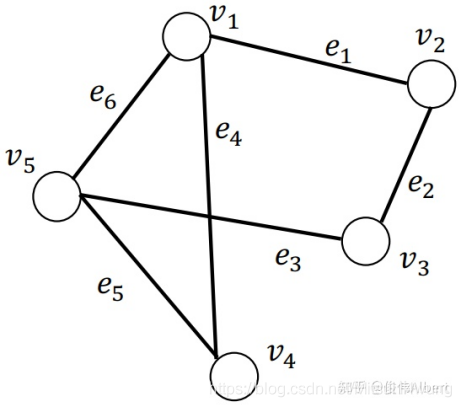
解法：先排2个节目 2\*1，再插空排3个节目2\*1\*3\*2=12

## 4.图论

### **4.1 什么是图？**

**图论**（Graph Theory）是数学的一个分支。它以图为研究对象。

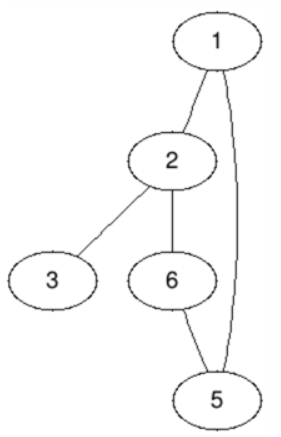
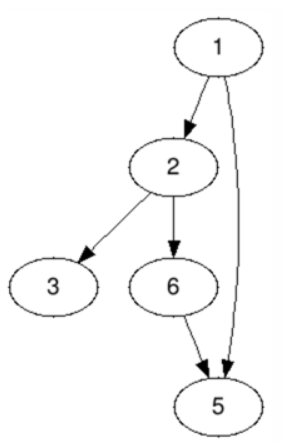
**图**可以被表示为 G={V, E}，其中 V={v1, ... , vN}，E= {e1, ... , eM}。（也就是说，V是节点的集合，即所谓的点集，E是连接两点的线即两点间关系的集合，即所谓的边集）



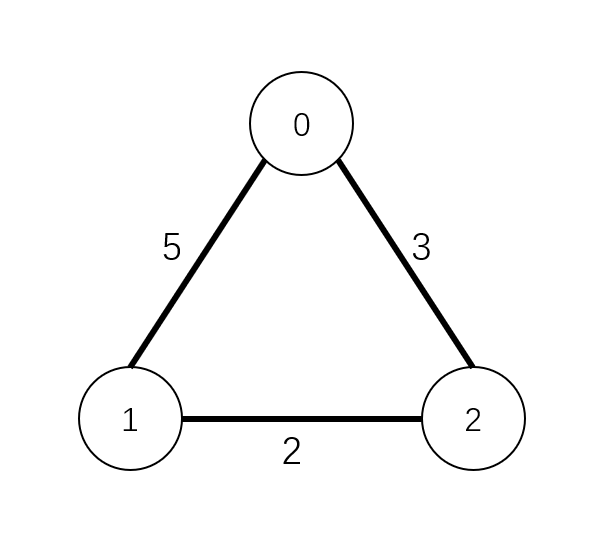
### **4.2 图的分类**

无向图与有向图

如果图中所有边都是无向边，则称为**无向图**，反之称为**有向图**。

连接顶点与顶点的边是否有数值与之对应，有的话就是**有权图**，否则就是**无权图**。



### **4.3 图的节点**

对于无向图

度/次数：与顶点关联的边的数目。

奇点：度为奇数的点。

偶点：度为偶数的点。

对于有向图

出度：从顶点引出的边的数目

入度：从顶点引入的边的数目。

### **4.4 图的连通性**

**连通**：如果图中结点U、V之间存在一条从U通过若干条边、点到达V的通路，则称U与V是连通的。

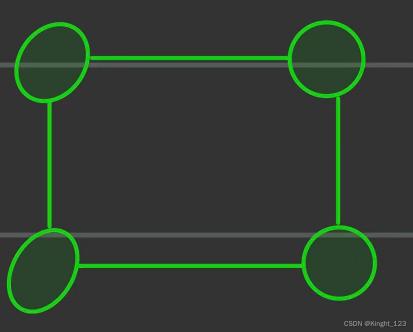
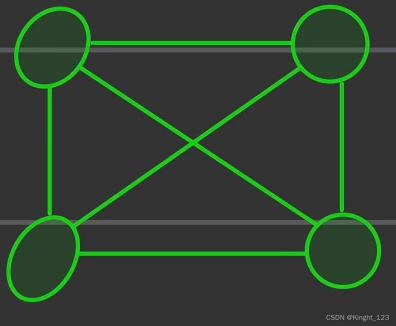
**回路**：起点和终点相同的路径，称为回路，或“环”

**完全图**：一个n阶的完全无向图含有n\*(n-1)/2条边；一个n阶的完全有向图含有n\*(n-1)条边

有向完全图与无向完全图的区别是，**有向完全图的两个结点可以连接两条边**。

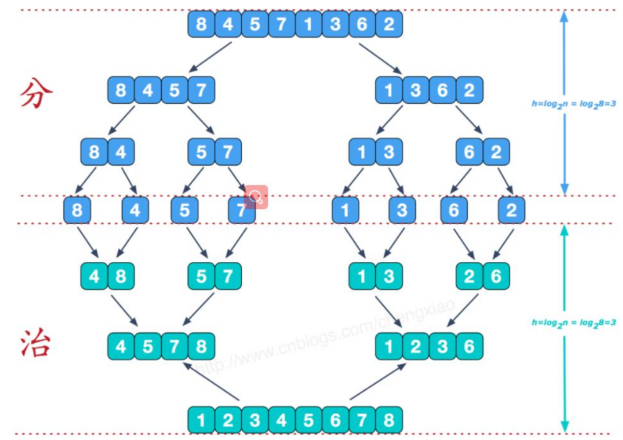
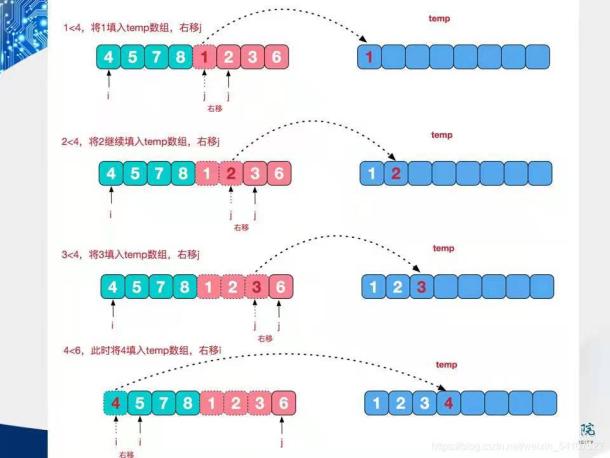
完全无向图：度数=边数\*2

完全有向图：度数（入度与出度）=边数\*2

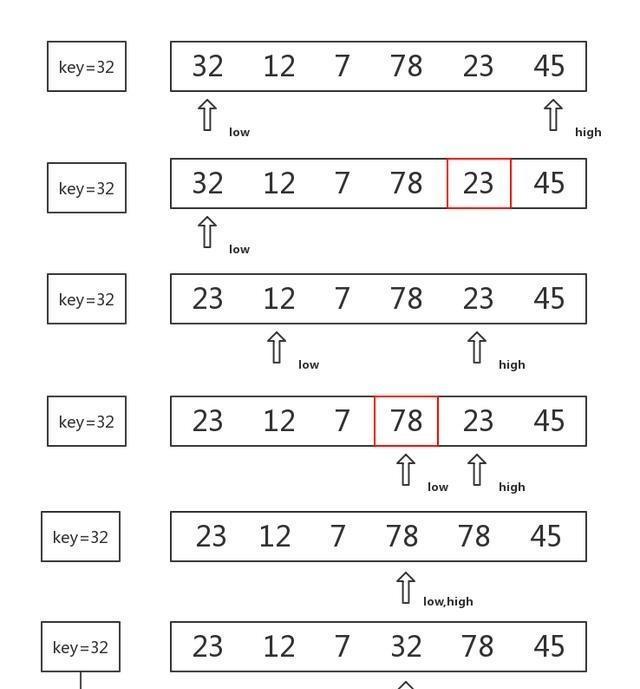
 

## 5.排序

### **归并排序**

### **快速排序**



区别：

