

数据结构：数组、链表、栈、队列、散列表 (hash)、树、图、堆

## 线性结构

- 数组与矩阵
- **线性表（链表、栈和队列）**

## 非线性结构

- 广义表
- **树和二叉树**
- 图

## 算法

- **排序和查找（算法和时间空间复杂度计算）**
- **算法基础及常见算法**

PS：后面还会讲到其它算法和算法应用

研究数据结构的意义是：**提高性能**，不同数据结构适合不同场景的数据处理，性能上有着很大的区别。

## 数组

---

一维数组和二维数组

二维数组的物理存储结果也是和一维数组类似的

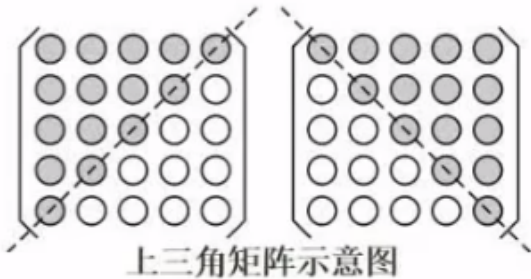
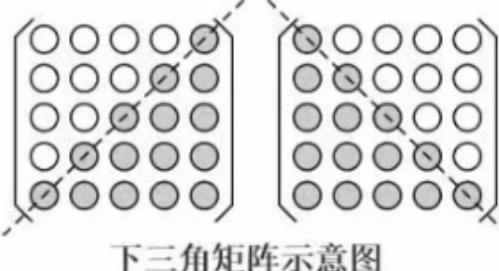
### 计算数组存储地址

## 稀疏矩阵

---

二维数组实现矩阵

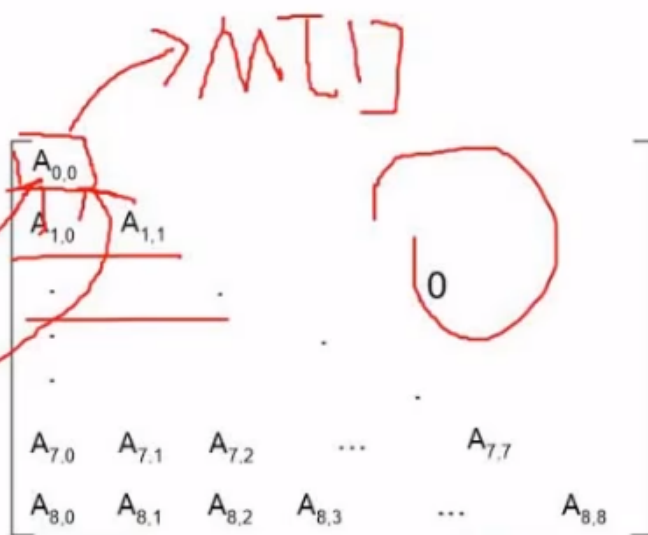
大量元素都是0时，二维数组只存储成：上三角矩阵和下三角矩阵

稀疏矩阵	示意图	要点
上三角矩阵	 <p>上三角矩阵示意图</p>	在矩阵中下标分别为 <i>i</i> 和 <i>j</i> 的元素，对应的一维数组的下标计算公式为： $(2n-i+1) \times i/2 + j$
下三角矩阵	 <p>下三角矩阵示意图</p>	在矩阵中下标分别为 <i>i</i> 和 <i>j</i> 的元素，对应的一维数组的下标计算公式为： $(i+1) \times i/2 + j$

## 稀疏矩阵

8x9

设有如下所示的下三角矩阵  $A[0..8, 0..8]$ ，将该三角矩阵的非零元素（即行下标不小于列下标的所有元素）按行优先压缩存储在数组  $M[1..m]$  中，则元素  $A[i, j]$  ( $0 \leq i \leq 8, j \leq i$ ) 存储在数组  $M$  的 ( ) 中。



- A.  $M[\frac{i(i+1)}{2} + j + 1]$   $M[1]$  B.  $M[\frac{i(i+1)}{2} + j]$   $M[0]$
- C.  $M[\frac{i(i-1)}{2} + j]$   $M[0]$  D.  $M[\frac{i(i-1)}{2} + j + 1]$   $M[1]$

## 链表

单向链表、双向链表、单向循环链表、双向循环链表

## 对比数组和链表

## 栈与队列

栈：先进后出，可通过链表数据结构来实现，控制结点的进出，即出入栈控制。

队列：先进先出，跟栈类似，主要也是控制结点的进出即可。

## 广义表

广义表是n个表元素组成的有限序列，是线性表和树的一种推广。

由于广义表是**对线性表和树的推广**，并且具有共享和递归特性的广义表可以和**有向图**建立对应，因此广义表的大部分运算与这些数据结构上的运算类似。

它被广泛的应用于人工智能等领域的**表处理语言LISP**语言中。在LISP语言中，广义表是一种最基本的数据结构，就连LISP 语言的程序也表示为一系列的广义表。（百度百科）

(1) 广义表常用表示

①  $E = ()$

E是一个空表，其长度为0。

②  $L = (a, b)$

L是长度为2的广义表，它的两个元素都是原子，因此它是一个线性表

③  $A = (x, L) = (x, (a, b))$

A是长度为2的广义表，第一个元素是原子x，第二个元素是子表L。

④  $B = (A, y) = ((x, (a, b)), y)$

B是长度为2的广义表，第一个元素是子表A，第二个元素是原子y。

⑤  $C = (A, B) = ((x, (a, b)), ((x, (a, b)), y))$

C的长度为2，两个元素都是子表。

⑥  $D = (a, D) = (a, (a, (a, (...))))$

D的长度为2，第一个元素是原子，第二个元素是D自身，展开后它是一个无限的广义表。

## (2) 广义表的深度

一个表的"深度"是指表展开后所含括号的层数。

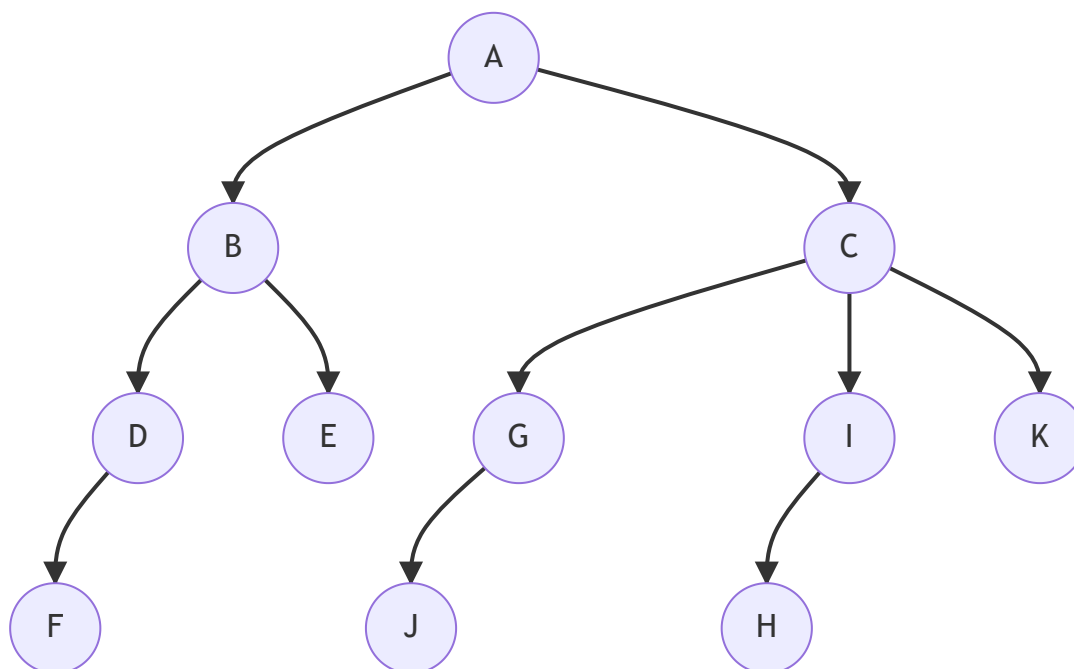
## 广义表例题

XXXXXXXXXX

## 树和二叉树

参考：[数据结构：树\(Tree\)【详解】\\_数据结构 树\\_UniqueUnit的博客-CSDN博客](#)

## 树的基本术语



- 结点的度：比如C结点的度为3、B结点的度为2、I结点的度为1；
- 树的度：结点的度最大值为树的度，即该树的度为3；
- 根结点：A；
- 叶子结点：度为0的结点，比如：F、E、J、H、K；
- 分支结点：度 $>0$ 的结点，比如：A、B、D、E、C、G、I；
- 内部结点：度 $>0$ 并非根节点，比如：B、D、E、C、G、I；
- 父结点：相对F来说，D是父结点，F是子结点；

- 子结点：相对F来说，D是父结点，F是子结点；
- 兄弟结点：相对B来说，D和E是兄弟节点；
- 堂兄弟结点：B和C为兄弟，D/E和G、I、K则为堂兄弟结点；
- 结点的层次：第1层--第4层；
- 结点的深度：自顶向下累加，4层；
- 结点的高度：自底向上累加，4层；
- 树的高度（深度）：最大层数，4层；
- 有序树和无序树：左右节点不可互换为有序，反之无序；

### 树的最基本的性质

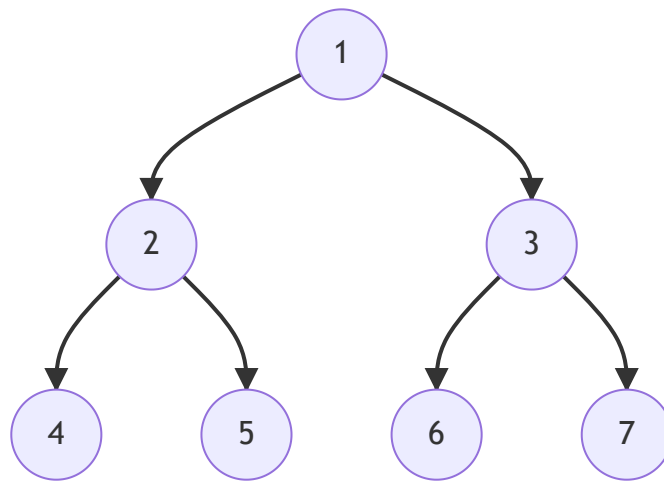
- 树中的结点数等于所有结点的度数加1；
- 度为m的树中第i层上至多有  $m^{(i-1)}$  个结点 ( $i \geq 1$ )
- 高度为h的m叉树至多有  $(m^h - 1) / (m - 1)$  个结点
- 具有n个结点的m叉树的最小高度为  $\lceil \log_m(n(m-1)+1) \rceil$

## 二叉树

树的度为2的树被称为二叉树。二叉树分为很多种类型，比如，满二叉树、完全二叉树、查询二叉树、最优二叉树、线索二叉树、平衡二叉树等等。

### 满二叉树

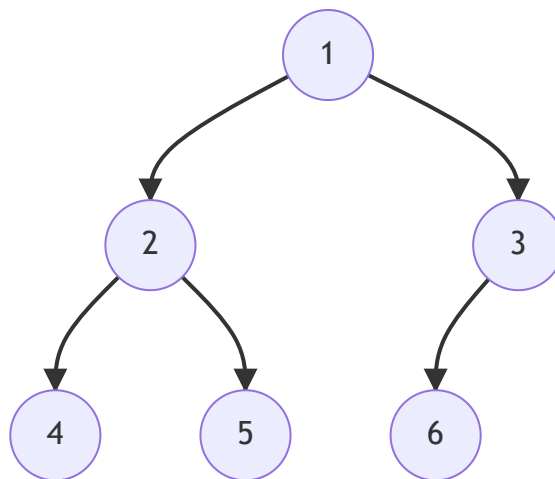
分支结点的度都为2的树称为满二叉树。满二叉树总的结点数为  $2^n - 1$ ，即二进制对应位数的最大值，如3位二进制的最大值为  $111 = 2^3 - 1 = 7$ ；第n层的节点数为  $2^{(n-1)}$ ，如3位二进制的第3位的值为  $100 = 2^{(3-1)} = 4$ 。



实际上，不可能每次都刚好填满最后一层，于是就有了完全二叉树。

## 完全二叉树

只存在1个度都为1的分支结点，并且该分支结点的子结点为左结点二叉树称为完全二叉树。即从上到下，从左到右，一个个新增结点到树上。完全二叉树总的结点数最多为 $2^n - 1$ ，即二进制对应位数的最大值，如3位二进制的最大值为 $111 = 2^3 - 1 = 7$ ；第n层的节点数最多为 $2^{n-1}$ ，如3位二进制的第3位的值为 $100 = 2^{3-1} = 4$ 。



## 总结

### 二叉树的重要特性

1. 深度为n的二叉树最多有 $2^n - 1$ 个结点，即n位二进制的最大值，如11111；
2. 二叉树的第i层最多有 $2^{i-1}$ 个结点，即i位二进制的值，如1000；
3. 对任意二叉树，如果其叶子结点数为 $n_0$ ，分支结点（度为2的结点）数为 $n_2$ ，则 $n_0 = n_2 + 1$ ；
4. 对完全二叉树按从上到下、从左到右的顺序依次编号1,2,...,i,..., n,则有以下关系:

- 如果 $i > 1$ ，则结点 $i$ 的双亲的编号是 $\lceil i/2 \rceil$ ，即当 $i$ 为偶数时，它是双亲的左孩子；当 $i$ 为奇数时，它是双亲的右孩子；当 $i=1$ 时，则结点 $i$ 为根节点；
- 如果 $2i \leq n$ ，则结点 $i$ 左子结点为 $2i$ ；否则为叶子结点（完全二叉树无左子结点的结点是叶子结点）；
- 如果 $2i + 1 \leq n$ ，则结点 $i$ 右子结点为 $2i + 1$ ，否则无右子结点；
- 结点 $i$ 所在层次(深度)为 $\log_2 i + 1$ ，延伸可得完全二叉树高度为 $\log_2 n + 1$ ，前提是 $n > 0$ ，不然没意义；

## 完全二叉树的意义

按照从上到下、从左到右的顺序依次编号 $1, 2, \dots, i, \dots, n$ ，可以根据结点 $i$ 很方便的计算出其它结点所在位置。

## 二叉树的遍历

- 层次遍历：从根节点开始从上到下从左到右一个个遍历；
- 前序遍历：先访问 **根结点**，再访问左子树结点，后访问右子树结点（根左右）；
- 中序遍历：先访问左子树结点，再访问 **根结点**，后访问右子树结点（左根右）；
- 后序遍历：先访问左子树结点，再访问右子树结点，后访问 **根结点**（左右根）；

遍历的这个“前中后序”指的是**根节点**时候访问的时序，而左右子树结点的遍历继续遵循前中后序原则进行遍历（相同规则嵌套，是一种递归遍历算法）。

### 例子：

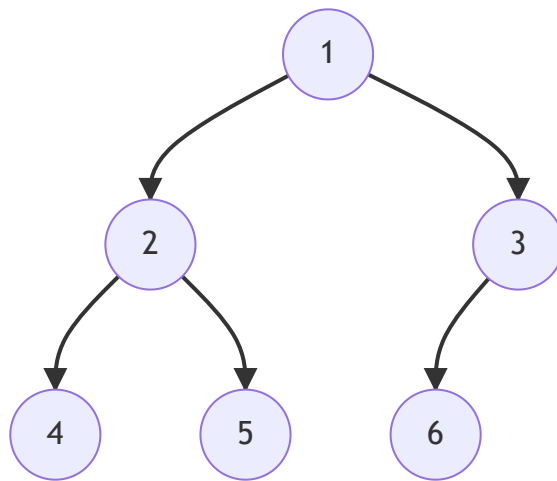
如图，

前序遍历结果：1、2、4、5、3、6；

中序遍历结果：4、2、5、1、6、3；

后序遍历结果：4、5、2、6、3、1；

层次序列结果：1、2、3、4、5、6；



## 反向构造二叉树

根据前序/后序/层次序列与中序，两种序列来推导出二叉树。

**推导逻辑：**前序、后序和层次序列都可以确定根节点，然后将序列拆分成左右两部分，再结合中序序列根据这个原理一直拆分下去，从而得到最终二叉树。

例题：

## 普通树转二叉树

转换规则：

- 孩子结点==》左子树结点
- 兄弟结点==》孩子节点的右子树结点

也可以采用连线法

## 查询二叉树（排序二叉树）

规定：结点的值左小于根，根小于右。

根据其规定可得知，根在中间，即采用中序遍历会得到一个递增的有序序列。

**插入结点：**

**删除结点：**

**主要优点：**提高查询效率；



**主要缺点：**当树为空时，以第一个插入值为根节点，但该值并不是数组中的中间值，所以会出现左右子树排序不平衡。例如，数组{9,5,2,7,3,10}按顺序插入，9为根结点，大于9的只有一个10，其它4个节点都小于根节点9，出现左边重右边轻的情况。

也就是说，我们希望排序二叉树是比较平衡的，即其深度与 完全二叉树 相同，那么查找的时间复杂也就为 $O(\log_n)$ ，近似于折半查找（二分查找）。不平衡的最坏情况就是左斜树/右斜树，查找时间复杂度为 $O(n)$ ，这等同于顺序查找。因此，我们希望把它构建成一棵平衡的排序二叉树。

## 平衡二叉树

**定义：**平衡二叉树(Self-Balancing Binary Search Tree 或 Height-Balanced Binary Search Tree)是一种二叉排序树，其中任意结点的左右子树深度相差不超过1。

**特点：**

- 任意结点的左右子树深度相差不超过1；
- 每个结点的平衡度只能为-1、0或1。

**平衡度：**当前结点的 左子树深度-右子树深度=平衡度

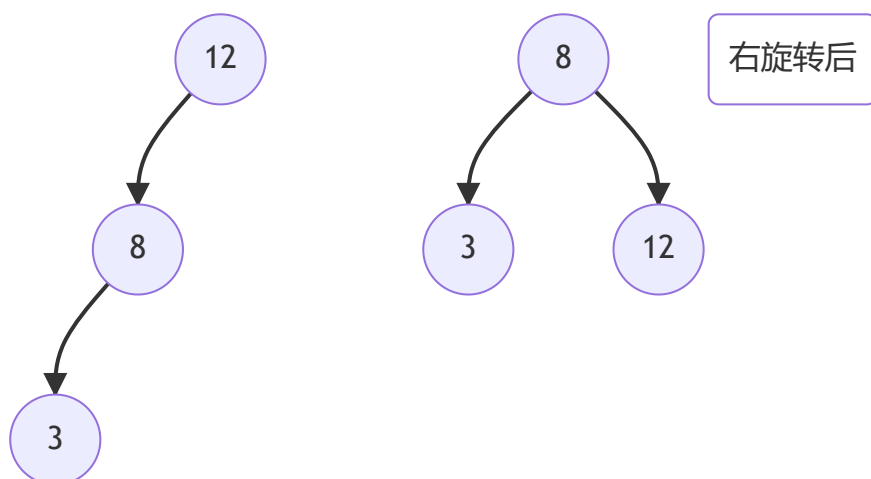
**查找：**在查找过程中，与给定值进行比较的关键字个数不超过树的深度，因此平衡二叉树的平均查找长度为 $O(\log_2 n)$ ；

**插入(动态调平衡问题)**

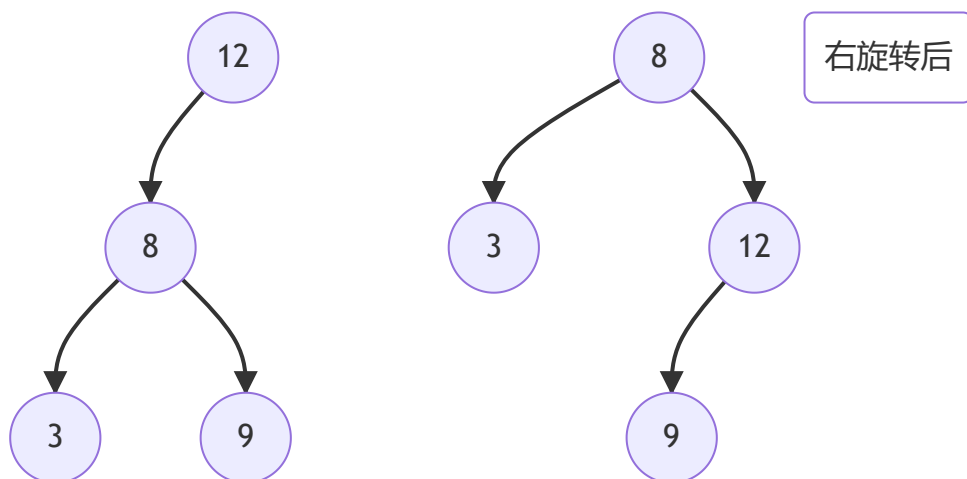
### LL平衡旋转（右单旋转）

树的根结点的左子树，再插入左结点出现不平衡，即左左（LL）不平衡，此时需要做右旋转。

情况一：如图，右旋转操作比较简单，原左子树的根结点作为树的根结点，原树的根结点作为新根结点的右结点，其它结点不变。



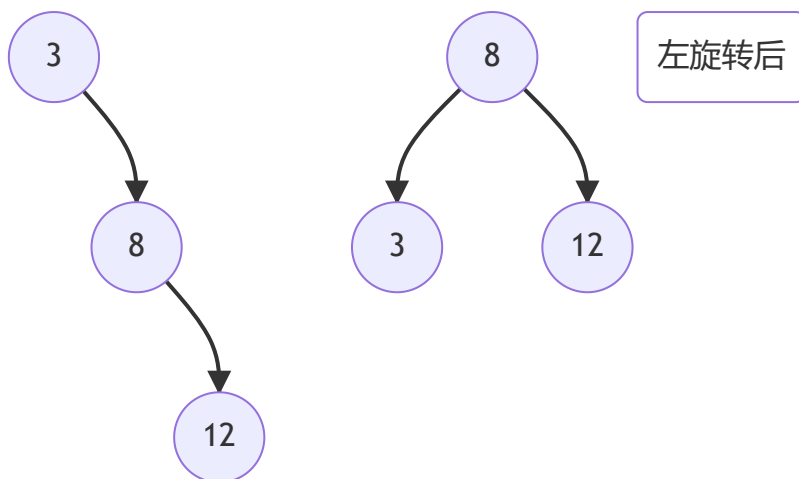
情况二：如图，这种情况一般在多次旋转时才会有出现，右旋转操作起来多一个步骤，原左子树的根结点作为树的根结点，原树的根结点作为新根结点的右结点，**原左子树的右结点作为原根结点的左结点**，其它结点不变。情况一可以看作是情况二的特例。



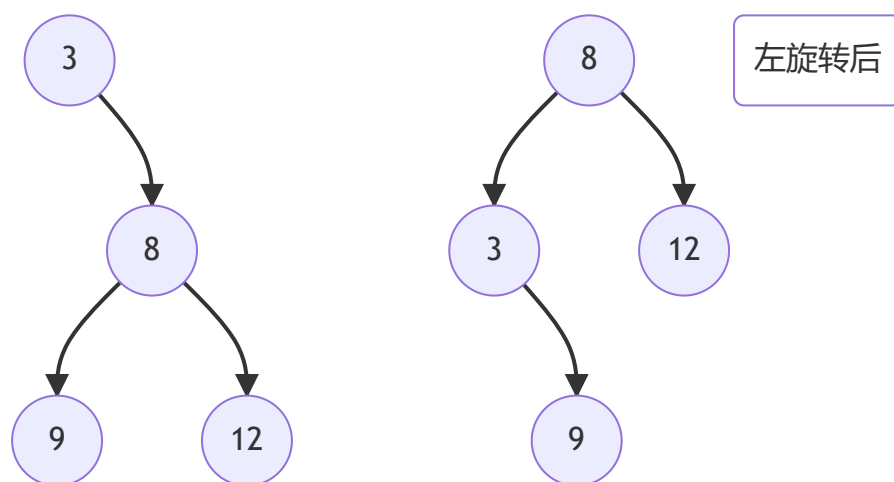
### RR平衡旋转（左单旋转）

树的根结点的右子树，再插入右结点出现不平衡，即右右（RR）不平衡，此时需要做左旋转。

情况一：如图，左旋转操作比较简单，原右子树的根结点作为树的根结点，原树的根结点作为新根结点的左结点，其它结点不变。



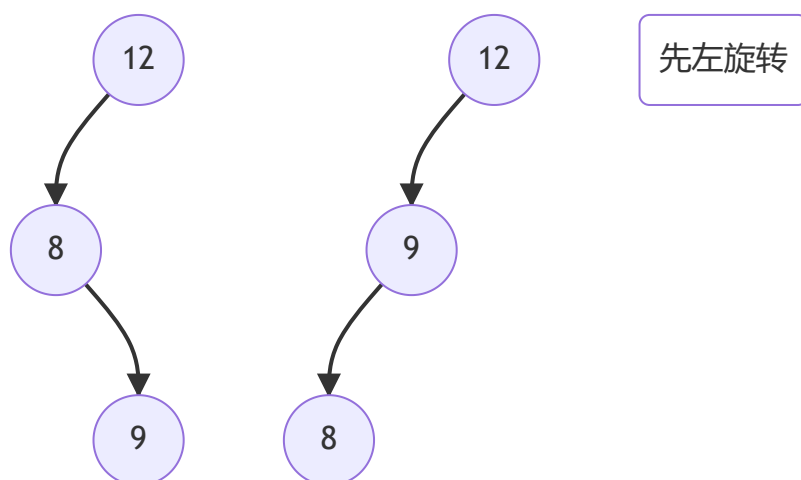
情况二：如图，这种情况一般在多次旋转时才会有出现，左旋转操作起来多一个步骤，原右子树的根结点作为树的根结点，原树的根结点作为新根结点的左结点，**原右子树的左结点作为原根结点的右结点**，其它结点不变。



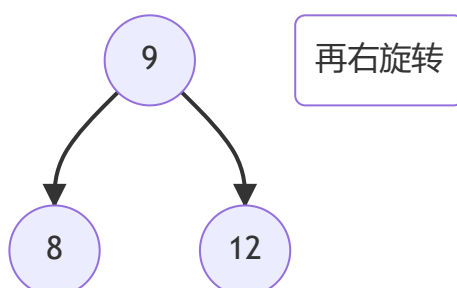
### LR平衡旋转（先左后右双旋转）

树的根结点的左子树，再插入右结点出现不平衡，即左右（LR）不平衡，此时需要先做RR平衡旋转，再做LL平衡旋转。

情况一：如图，左子树先做RR平衡旋转（左旋转），变成LL平衡旋转，



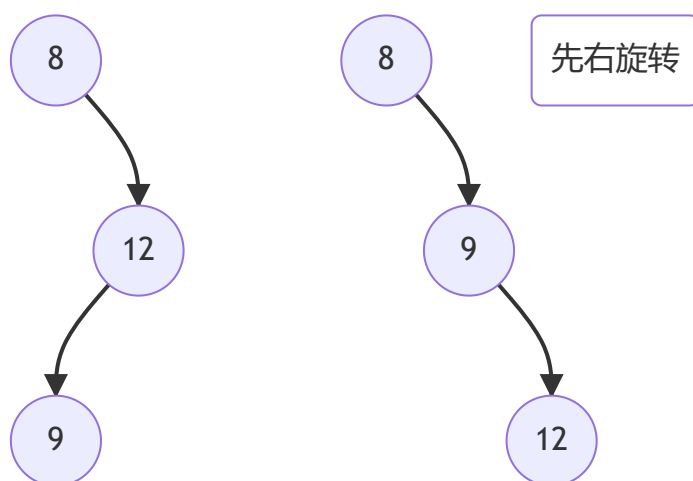
再做LL平衡旋转（右旋转），得到最终结果。



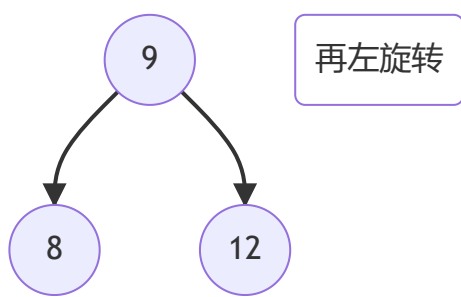
### RL平衡旋转（先右后左双旋转）

树的根结点的右子树，再插入左结点出现不平衡，即右左（RL）不平衡，此时需要先做LL平衡旋转，再做RR平衡旋转。

情况一：如图，右子树先做LL平衡旋转（右旋转），变成RR平衡旋转，



再做RR平衡旋转（左旋转），得到最终结果。



二叉排序树还有另外的平衡算法，如**红黑树**(Red Black Tree)等，与平衡二叉树(AVL树)相比各有优势。

### 最优二叉树（哈夫曼树）

它是一种**工具二叉树**，哈夫曼编码是一种被广泛应用而且非常有效的**无损数据压缩编码**。

#### 基本概念

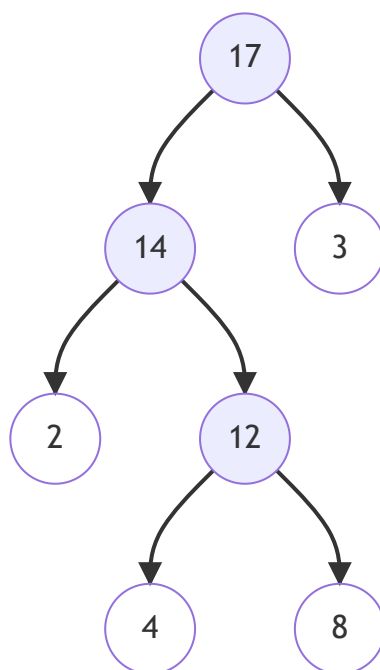
- 权：结点的数值；
- 结点的带权路径长度：从 **根结点** 到 **任意结点** 的路径长度(经过的边数)与该结点上权值的乘积；
- 树的带权路径长度（树的代价）：树中所有 **叶子结点** 的带权路径长度之和。

例题：如图计算结点4的带权路径长度和树的带权路径长度

结点4的带权路径长度： $4 \times 3 = 12$ ；

树的带权路径长度： $4 \times 3 + 8 \times 3 + 2 \times 2 + 3 \times 1 = 12 + 24 + 4 + 3 = 43$  =》（权重x层数）的累加

注意：叶子结点才是真正的结点，权值结点只是左右子结点的和，是中间结果。



## 哈夫曼树的构造

根据原始结点值构造出最小权值的根节点树？？？

步骤：

1. 先把有权值的叶子结点按照从大到小（从小到大也可以）的顺序排列成一个有序序列。
2. 取最后两个最小权值的结点作为一个新节点的两个子结点，注意相对较小的是左孩子。
3. 用第2步构造的新结点替掉它的两个子节点，插入有序序列中，保持从大到小排列。
4. 重复步骤2到步骤3，直到根节点出现。

例题：

XXXX

## 哈夫曼编码

XXXX

## 线索二叉树

二叉树中很多结点的指针指向是空（NULL）的，利用结点中空闲的空间指向特定的结点，从而提高遍历速度。

规则：在空闲的指针中，左指向前驱结点，右指向后继结点

例如：

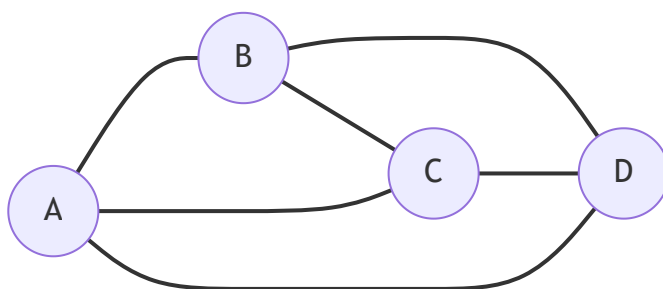
如图，前序线索二叉树，它的前序遍历为：ABDEHCFG I，D为叶子结点左右指针都是空的，所以D的左可以指向B结点，右可以指向后继的E结点。中序和后序线索二叉树原理一样。

## 图

### 图的概念

#### 无向图

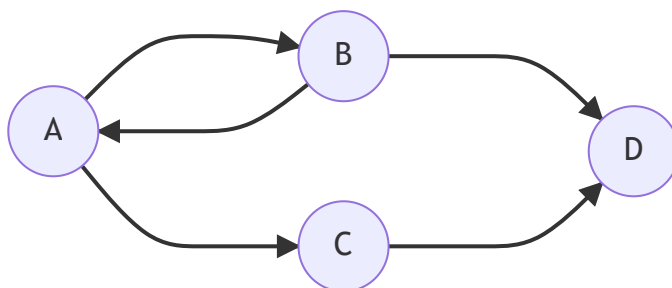
无向图表示为： $E_1 = \{(A, B), (A, C), (A, D), (B, C), (B, D), (C, D)\}$



#### 有向图

有向图表示为： $E_2 = \{ \langle A, B \rangle, \langle B, A \rangle, \langle A, C \rangle, \langle C, D \rangle, \langle B, D \rangle \}$

注意：有向图的A->B 和B->A是不同的两个。



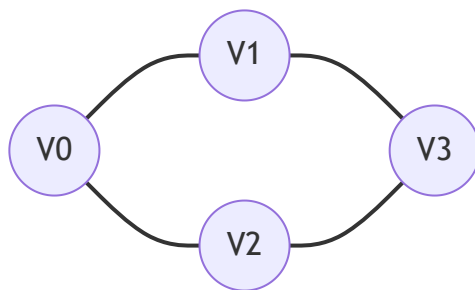
### 图的存储结构

图的结构比较复杂，任意两个顶点之间都可能存在联系，不可能用简单的顺序存储结构来表示。

#### 邻接矩阵

图的邻接矩阵(Adjacency Matrix) 存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组(称为邻接矩阵)存储图中的边或弧的信息。

例如无向图



**邻接矩阵**（图中边的数组）

如图可知是一个斜对称矩阵，可以用右上三角或左下三角矩阵表示。

$$\begin{array}{c} v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

**邻接表**

数组+链表

## 图的遍历

### 广度优先遍历

广度优先遍历(Breadth First Search)，又称为广度优先搜索，简称BFS。

类似于层次遍历

### 深度优先遍历

深度优先遍历(Depth First Search)，也有称为深度优先搜索，简称为DFS。

深度优先搜索类似于树的先序遍历，

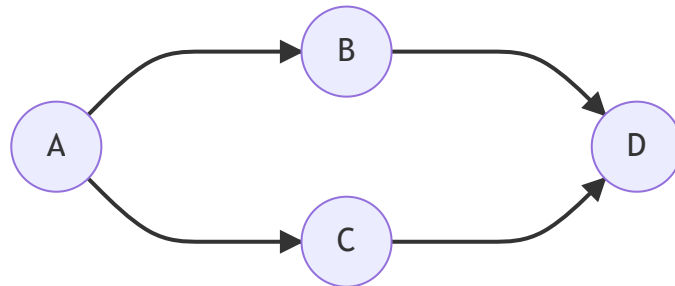
## 拓扑排序

所谓拓扑排序，其实就是对一个有向图构造拓扑序列的过程。每个AOV网都有一个或多个拓扑排序序列。

推算步骤

- ①从AOV网中选择一个没有前驱的顶点并输出，即入度为0的结点；
- ②从网中删除该顶点和所有以它为起点的有向边；
- ③重复①和②直到当前的AOV网为空或当前网中不存在无前驱的顶点为止。如果输出顶点数少了，哪怕是少了一个，也说明这个网存在环(回路)，不是AOV网。

例如：如图，拓扑排序为：ABCD 或ACBD。



## 最小生成树

### 普里姆 (Prim) 算法

从一个顶点出发，在保证不形成回路的前提下，每找到并添加一条最短的边，就把当前形成的连通分量当做一个整体或者一个点看待，然后重复“找最短的边并添加”的操作。

### 克鲁斯卡尔 (Kruskal) 算法

Kruskal 算法是一种按权值的递增次序选择合适的边来构造最小生成树的方法。

## 最短路径

求图从开始结点到结束结点的最短权重路径。

### 迪杰斯特拉( Dijkstra )算法

xxx

### 弗洛伊德( Floyd )算法

xxxx





微信搜一搜

Q Java全栈布道师