------------------------------数据结构之线性结构-------------------------------

--------------------------------------数组-------------------------------------

**数组（Array）** 是一种很常见的数据结构。它由相同类型的元素（element）组成，并且是使用一块连续的内存来存储。

数组的特点是：**提供随机访问** 并且容量有限。

数组的优点:存取速度快

数组的缺点:

* 事先必须知道数组的长度
* 插入删除元素很慢
* 空间通常是有限制的
* 需要大块连续的内存块
* 插入删除元素的效率很低

假如数组的长度为 n。

访问：O（1）*//访问特定位置的元素*

插入：O（n ）*//最坏的情况发生在插入发生在数组的首部并需要移动所有元素时*

删除：O（n）*//最坏的情况发生在删除数组的开头发生并需要移动第一元素后面所有的元素时*

===========================================================

---------------------------------------链表------------------------------------

链表（LinkedList） 虽然是一种线性表，但是并不会按线性的顺序存储数据，使用的不是连续的内存空间来存储数据。

链表的插入和删除操作的复杂度为 O(1) ，只需要知道目标位置元素的上一个元素即可。但是，在查找一个节点或者访问特定位置的节点的时候复杂度为 O(n) 。

常见链表分类：

1. 单链表
2. 双向链表
3. 循环链表
4. 双向循环链表

假如链表中有n个元素。

访问：O（n）*//访问特定位置的元素*

插入删除：O（1）*//必须要要知道插入元素的位置*

应用场景

* 如果需要支持随机访问的话，链表没办法做到。
* 如果需要存储的数据元素的个数不确定，并且需要经常添加和删除数据的话，使用链表比较合适。
* 如果需要存储的数据元素的个数确定，并且不需要经常添加和删除数据的话，使用数组比较合适。

数组 vs 链表

* 数组支持随机访问，而链表不支持。
* 数组使用的是连续内存空间对 CPU 的缓存机制友好，链表则相反。
* 数组的大小固定，而链表则天然支持动态扩容。如果声明的数组过小，需要另外申请一个更大的内存空间存放数组元素，然后将原数组拷贝进去，这个操作是比较耗时的！

===========================================================

---------------------------------------栈--------------------------------------

**栈** (stack)只允许在有序的线性数据集合的一端（称为栈顶 top）进行加入数据（push）和移除数据（pop）。因而按照 **后进先出（LIFO, Last In First Out）** 的原理运作。**在栈中，push 和 pop 的操作都发生在栈顶。**

栈常用一维数组或链表来实现，用**数组**实现的栈叫作 **顺序栈** ，用**链表**实现的栈叫作 **链式栈** 。

假设堆栈中有n个元素。

访问：O（n）*//最坏情况*

插入删除：O（1）*//顶端插入和删除元素*

栈既可以通过数组实现，也可以通过链表来实现。不管基于数组还是链表，入栈、出栈的时间复杂度都为 O(1)。

=========================================================

队列

**队列** 是 **先进先出( FIFO，First In, First Out)** 的线性表。在具体应用中通常用链表或者数组来实现，用数组实现的队列叫作 **顺序队列** ，用链表实现的队列叫作 **链式队列** 。**队列只允许在后端（rear）进行插入操作也就是 入队 enqueue，在前端（front）进行删除操作也就是出队 dequeue**

队列的操作方式和堆栈类似，唯一的区别在于队列只允许新数据在后端进行添加。

假设队列中有n个元素。

访问：O（n）*//最坏情况*

插入删除：O（1）*//后端插入前端删除元素*

1. 单队列

单队列就是常见的队列, 每次添加元素时，都是添加到队尾。单队列又分为 **顺序队列（数组实现）** 和 **链式队列（链表实现）**。

**顺序队列存在“假溢出”的问题也就是明明有位置却不能添加的情况。**

1. 循环队列

循环队列可以解决顺序队列的假溢出和越界问题。解决办法就是：从头开始，这样也就会形成头尾相接的循环，这也就是循环队列名字的由来。

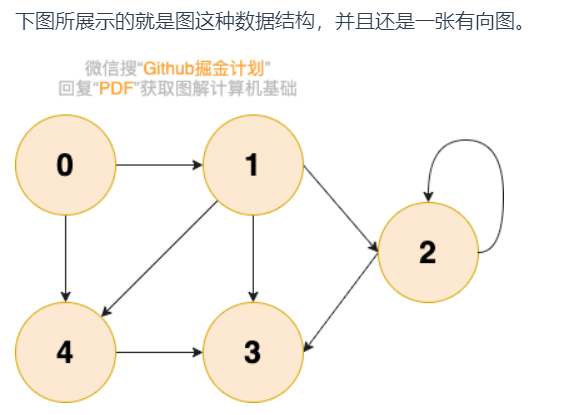
场景：当我们需要按照一定顺序来处理数据的时候可以考虑使用队列，如：堵塞队列，线程池中的请求/任务队列，Linux内核进程队列（按优先级排队），播放器的播放列表、消息队列……..

=========================================================

-----------------------------------------------图-------------------------------------------------

但是，图形结构的元素之间的关系是任意的。

**何为图呢？** 简单来说，图就是由顶点的有穷非空集合和顶点之间的边组成的集合。通常表示为：**G(V,E)**，其中，G表示一个图，V表示顶点的集合，E表示边的集合。



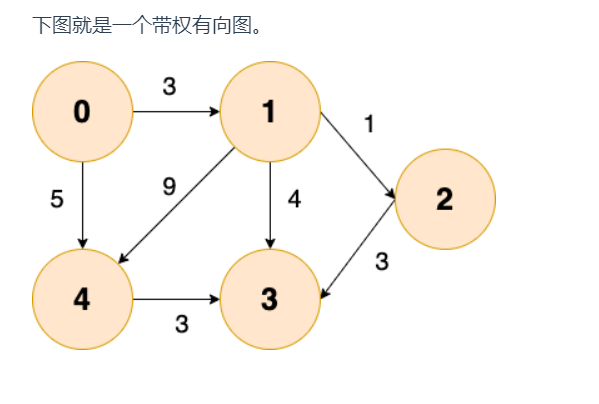
顶点：图中的数据元素，我们称之为顶点，图至少有一个顶点（非空有穷集合）对应到好友关系图，每一个用户就代表一个顶点。

边：顶点之间的关系用边表示。对应到好友关系图，两个用户是好友的话，那两者之间就存在一条边。

度：度表示一个顶点包含多少条边，在有向图中，还分为出度和入度，出度表示从该顶点出去的边的条数，入度表示进入该顶点的边的条数。对应到好友关系图，度就代表了某个人的好友数量。

无权图和带权图

对于一个关系，如果我们既关心关系的有无，也关心关系的强度，比如描述地图上两个城市的关系，需要用到距离，那么就用带权图来表示，带权图中的每一条边一个数值表示权值，代表关系的强度。

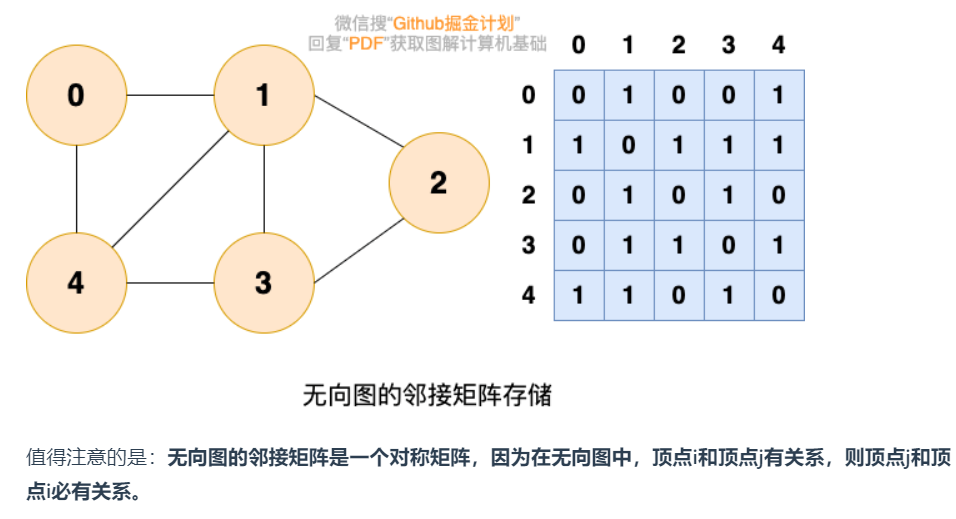


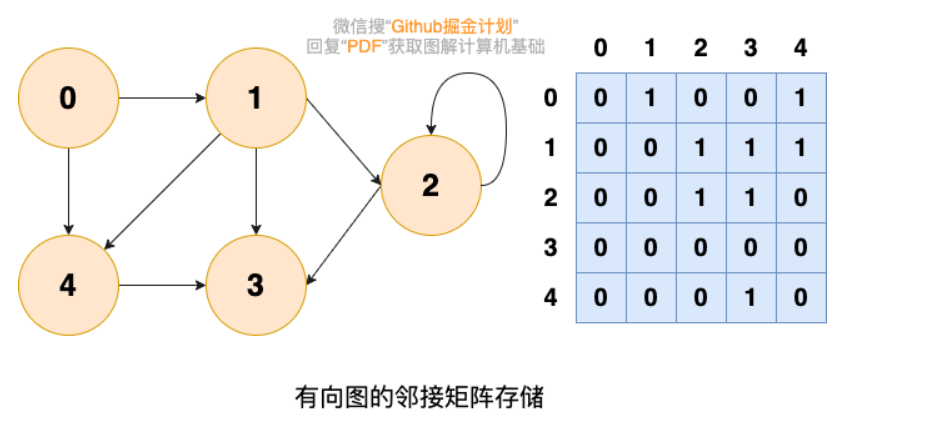
图的存储

1.邻接矩阵存储

如果第i个顶点和第j个顶点之间有关系，且关系权值为n，则 A[i][j]=n 。

在无向图中，我们只关心关系的有无，所以当顶点i和顶点j有关系时，A[i][j]=1，当顶点i和顶点j没有关系时，A[i][j]=0。如下图所示：

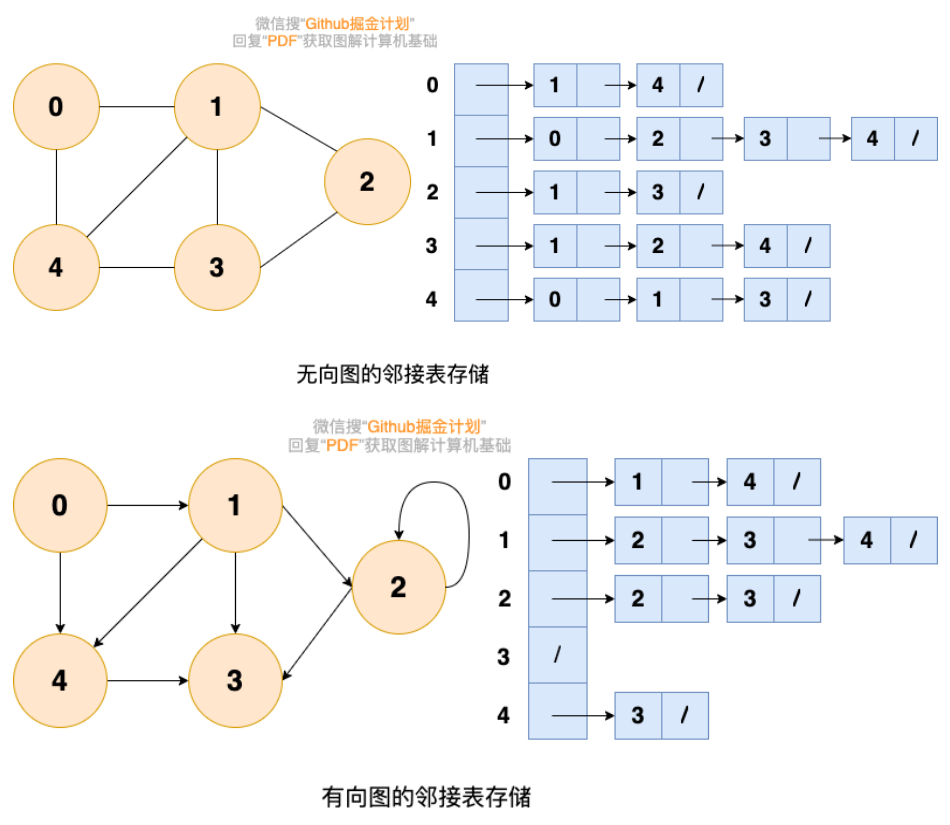




邻接矩阵存储的方式优点是简单直接（直接使用一个二维数组即可），并且，在获取两个定点之间的关系的时候也非常高效（直接获取指定位置的数组元素的值即可）。但是，这种存储方式的缺点也比较明显，那就是比较浪费空间

2. 邻接表存储------针对上面邻接矩阵比较浪费内存空间的问题

邻接链表使用一个链表来存储某个顶点的所有后继相邻顶点。对于图中每个顶点Vi，把所有邻接于Vi的顶点Vj链成一个单链表，这个单链表称为顶点Vi的 **邻接表**。

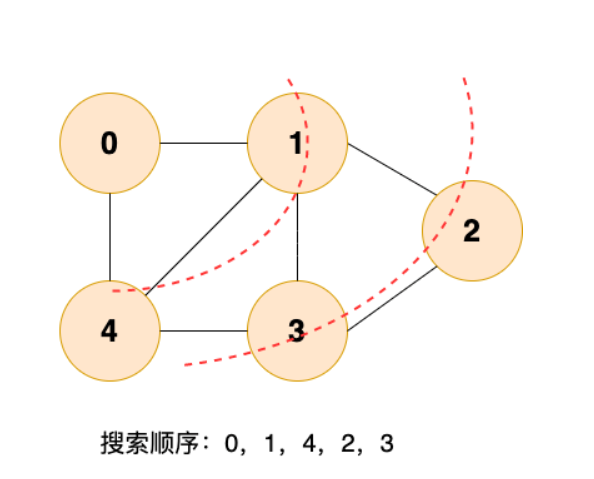


大家可以数一数邻接表中所存储的元素的个数以及图中边的条数，你会发现：

* 在无向图中，邻接表元素个数等于边的条数的两倍，如左图所示的无向图中，边的条数为7，邻接表存储的元素个数为14。
* 在有向图中，邻接表元素个数等于边的条数，如右图所示的有向图中，边的条数为8，邻接表存储的元素个数为8。

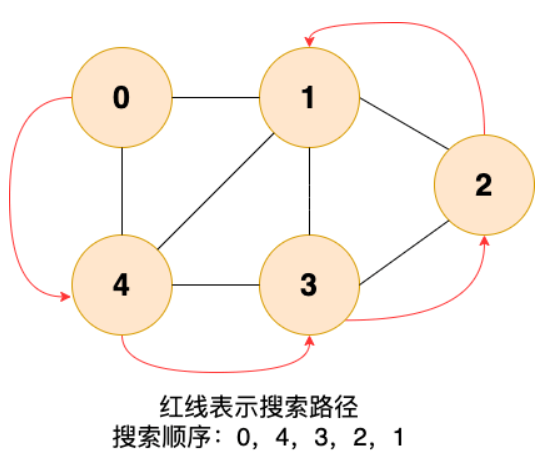
图的搜索

1. **广度优先搜索**---“一层层向外扩展”，用队列实现



1. **深度优先搜索**---“一条路走到黑”，用栈实现

深度优先搜索就是“一条路走到黑”，从源顶点开始，一直走到没有后继节点，才回溯到上一顶点，然后继续“一条路走到黑”，如下图所示：



===========================================================

堆

堆中的每一个节点值都大于等于（或小于等于）子树中所有节点的值。或者说，任意一个节点的值都大于等于（或小于等于）所有子节点的值。

场景：当我们只关心所有数据中的最大值或者最小值，存在多次获取最大值或者最小值，多次插入或删除数据时，就可以使用堆。

（注意：很多博客说堆是完全二叉树，其实并非如此，堆不一定是完全二叉树，只是为了方便存储和索引，我们通常用完全二叉树的形式来表示堆，事实上，广为人知的斐波那契堆和二项堆就不是完全二叉树,它们甚至都不是二叉树。（二叉）堆是一个数组，它可以被看成是一个 近似的完全二叉树。）

堆的优势：有序数组vs堆

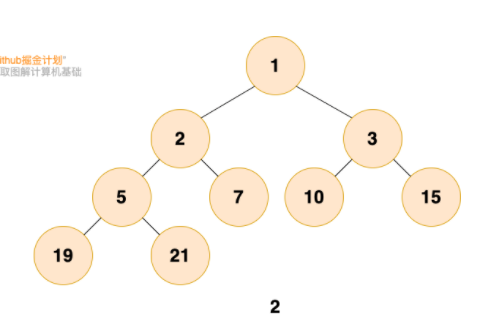
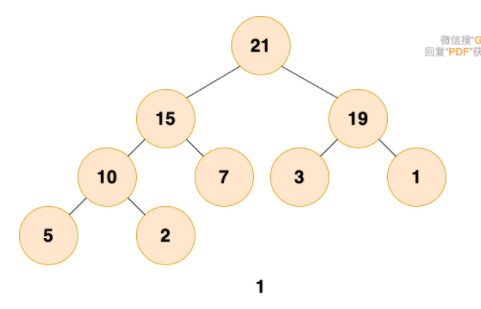
有小伙伴可能会想到用有序数组，初始化一个有序数组时间复杂度是 O(nlog(n))，查找最大值或者最小值时间复杂度都是 O(1)，但是，涉及到更新（插入或删除）数据时，时间复杂度为 O(n)，即使是使用复杂度为 O(log(n)) 的二分法找到要插入或者删除的数据，在移动数据时也需要 O(n) 的时间复杂度。

相对于有序数组而言，堆的主要优势在于更新数据效率较高。 堆的初始化时间复杂度为 **O(nlog(n))**，堆可以做到**O(1)**时间复杂度取出最大值或者最小值，**O(log(n))**时间复杂度插入或者删除数据

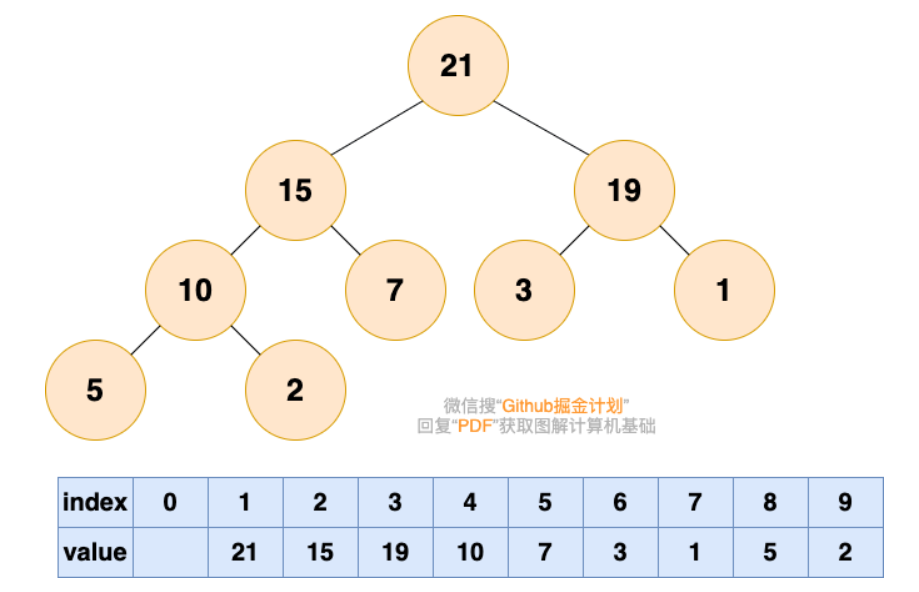
堆的分类：

最大堆：堆中的每一个节点的值都大于等于子树中所有节点的值

最小堆：堆中的每一个节点的值都小于等于子树中所有节点的值



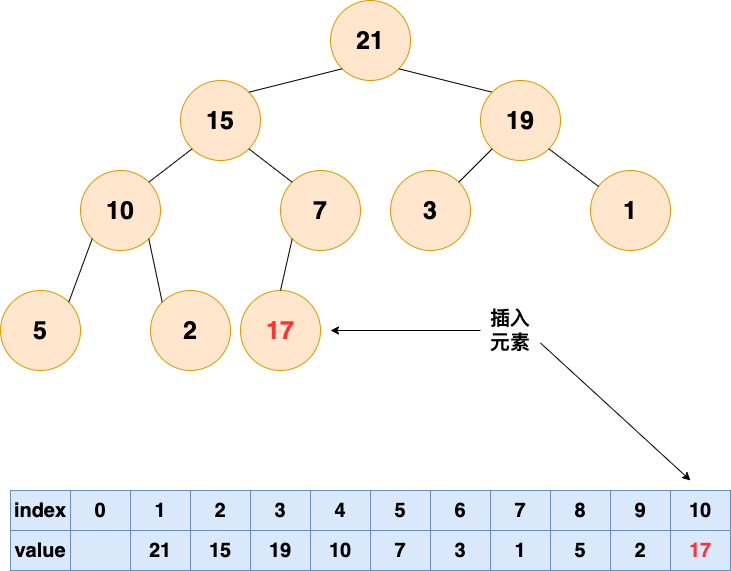
堆的存储：之前介绍树的时候说过，由于完全二叉树的优秀性质，利用数组存储二叉树即节省空间，又方便索引（若根结点的序号为1，那么对于树中任意节点i，其左子节点序号为 2\*i，右子节点序号为 2\*i+1）。为了方便存储和索引，（二叉）堆可以用完全二叉树的形式进行存储。存储的方式如下图所示：



堆的操作：插入元素，删除堆顶元素

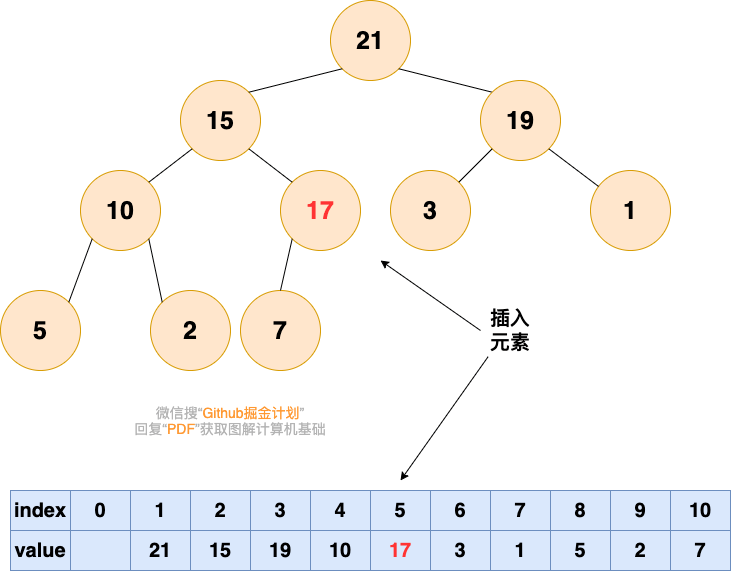
（1）删除元素

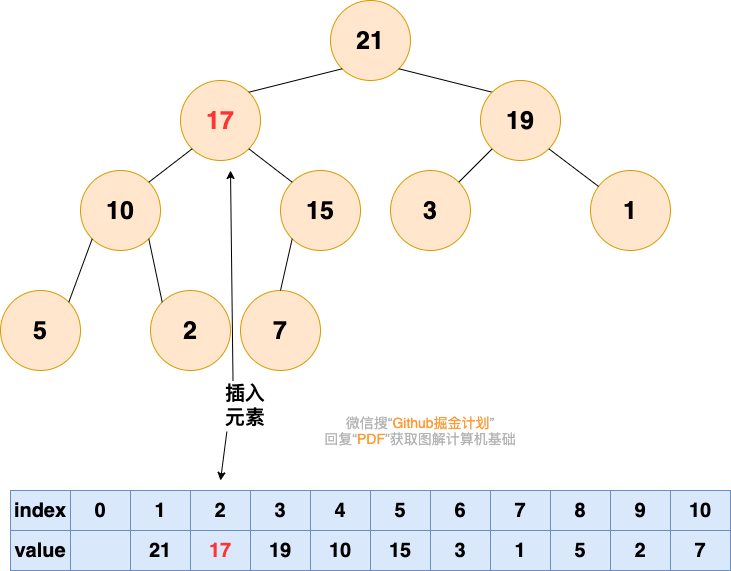
1.将要插入的元素放到最后



有能力的人会逐渐升职加薪，是金子总会发光的！！！

2.从底向上，如果父结点比该元素大，则该节点和父结点交换，直到无法交换





（2）删除堆顶元素

根据堆的性质可知，最大堆的堆顶元素为所有元素中最大的，最小堆的堆顶元素是所有元素中最小的。当我们需要多次查找最大元素或者最小元素的时候，可以利用堆来实现。

删除堆顶元素后，为了保持堆的性质，要对堆的结构进行调整，将这个过程称之为"**堆化**"，堆化的方法分为两种：

* 一种是自底向上的堆化，上述的插入元素所使用的就是自底向上的堆化，元素从最底部向上移动。
* 另一种是自顶向下堆化，元素由最顶部向下移动。

**自底向上堆化**：对于大顶堆-----先删除堆顶元素，然后各代儿子谁大谁上；

对于小顶堆-----先删除堆顶元素，然后各代儿子谁小谁上

缺点：会产生“气泡”，浪费存储空间

**自顶而下堆化**：“石沉大海”---那么第一件事情，就是把石头抬起来，从海面扔下去。这个石头就是堆的最后一个元素，我们将最后一个元素移动到堆顶。 然后开始将这个石头沉入海底，不停与左右子节点的值进行比较，和较大的子节点交换位置，直到无法交换位置。

总结：

* 插入元素 ：先将元素放至数组末尾，再自底向上堆化，将末尾元素上浮
* 删除堆顶元素 ：删除堆顶元素，将末尾元素放至堆顶，再自顶向下堆化，将堆顶元素下沉。也可以自底向上堆化，只是会产生“气泡”，浪费存储空间。最好采用自顶向下堆化的方式

堆排序：

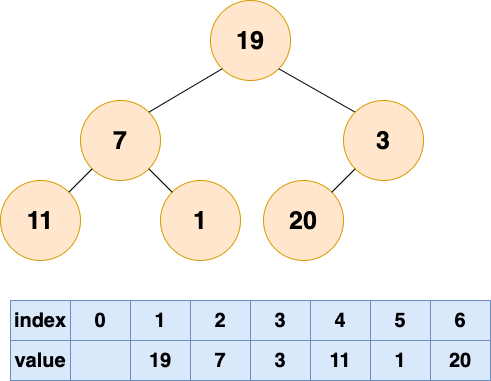
堆排序的过程分为两步：

* 第一步是建堆，将一个无序的数组建立为一个堆
* 第二步是排序，将堆顶元素取出，然后对剩下的元素进行堆化，反复迭代，直到所有元素被取出为止。

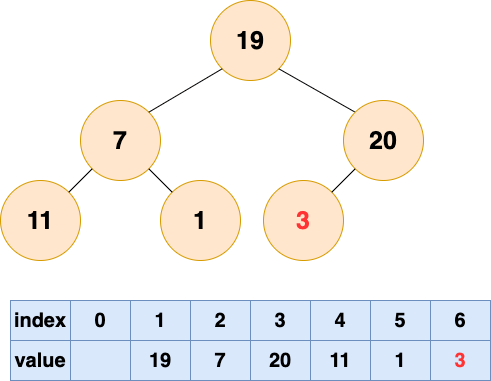
建堆：建堆的过程就是一个对所有非叶节点的自顶向下堆化过程。

首先要了解哪些是非叶节点，最后一个节点的父结点及它之前的元素，都是非叶节点。也就是说，如果节点个数为n，那么我们需要对n/2到1的节点进行自顶向下（沉底）堆化。

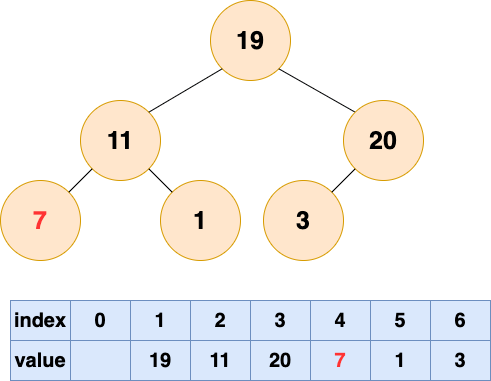
建堆具体过程如下图：



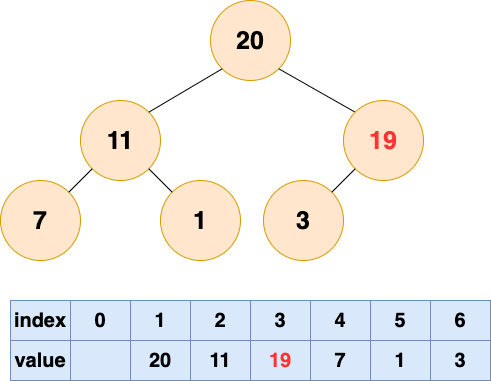
将初始的无序数组抽象为一棵树，图中的节点个数为6，所以4,5,6节点为叶节点，1,2,3节点为非叶节点，所以要对1-3号节点进行自顶向下（沉底）堆化，注意，顺序是从后往前堆化，从3号节点开始，一直到1号节点。 3号节点堆化结果：



2号节点堆化结果：



1号节点堆化结果：



至此，数组所对应的树已经成为了一个最大堆，建堆完成！

排序：

由于堆顶元素是所有元素中最大的，所以我们重复取出堆顶元素，将这个最大的堆顶元素放至数组末尾，并对剩下的元素进行堆化即可。

现在思考两个问题：

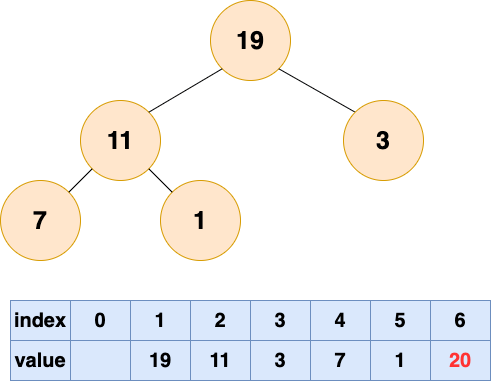
* 删除堆顶元素后需要执行自顶向下（沉底）堆化还是自底向上（上浮）堆化？
* 取出的堆顶元素存在哪，新建一个数组存？

先回答第一个问题，我们需要执行自顶向下（沉底）堆化，这个堆化一开始要将末尾元素移动至堆顶，这个时候末尾的位置就空出来了，由于堆中元素已经减小，这个位置不会再被使用，所以我们可以将取出的元素放在末尾。

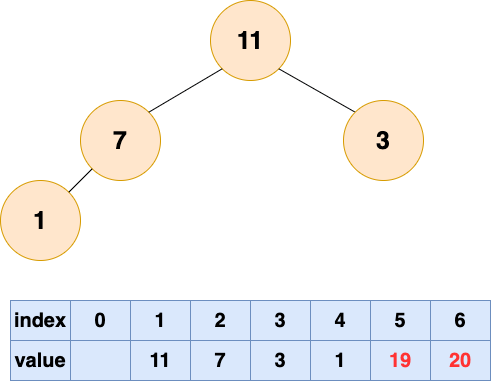
机智的小伙伴已经发现了，这其实是做了一次交换操作，将堆顶和末尾元素调换位置，从而将取出堆顶元素和堆化的第一步(将末尾元素放至根结点位置)进行合并。

详细过程如下图所示：

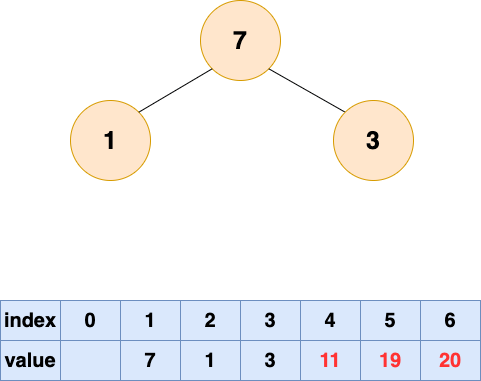
取出第一个元素并堆化：



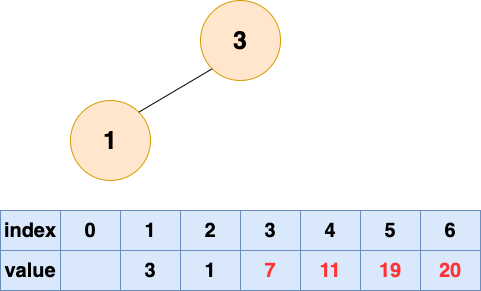
取出第二个元素并堆化：



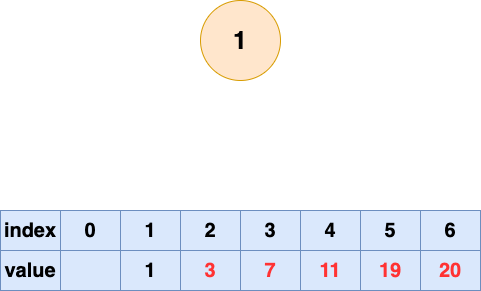
取出第三个元素并堆化：



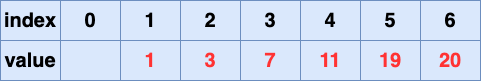
取出第四个元素并堆化：



取出第五个元素并堆化：



取出第六个元素并堆化：



堆排序完成！

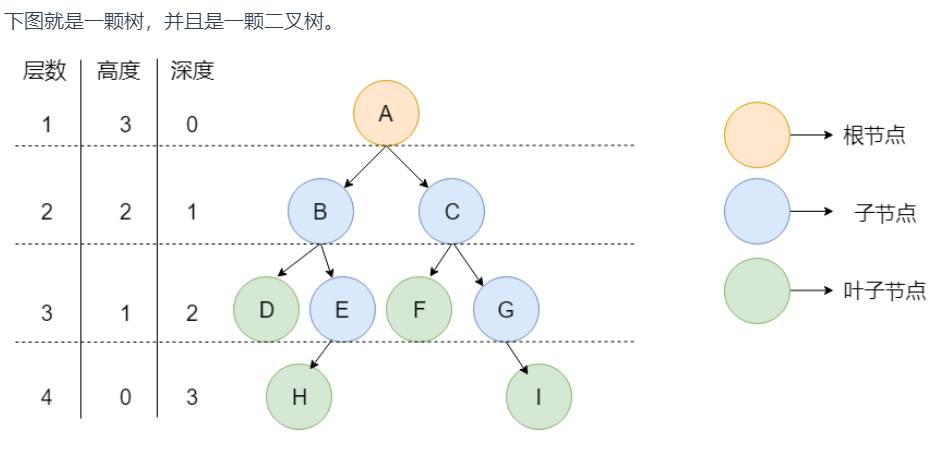
===========================================================

----------------------------------------树-------------------------------------

任何一颗非空树只有一个根节点。

一棵树具有以下特点：

* 一棵树中的任意两个结点有且仅有唯一的一条路径连通。
* 一棵树如果有 n 个结点，那么它一定恰好有 n-1 条边。
* 一棵树不包含回路。



* 节点 ：树中的每个元素都可以统称为节点。
* 根节点 ：顶层节点或者说没有父节点的节点。上图中 A 节点就是根节点。
* 父节点 ：若一个节点含有子节点，则这个节点称为其子节点的父节点。上图中的 B 节点是 D 节点、E 节点的父节点。
* 子节点 ：一个节点含有的子树的根节点称为该节点的子节点。上图中 D 节点、E 节点是 B 节点的子节点。
* 兄弟节点 ：具有相同父节点的节点互称为兄弟节点。上图中 D 节点、E 节点的共同父节点是 B 节点，故 D 和 E 为兄弟节点。
* 叶子节点 ：没有子节点的节点。上图中的 D、F、H、I 都是叶子节点。
* 节点的高度 ：该节点到叶子节点的最长路径所包含的边数。
* 节点的深度 ：根节点到该节点的路径所包含的边数
* 节点的层数 ：节点的深度+1。
* 树的高度 ：根节点的高度。

二叉树

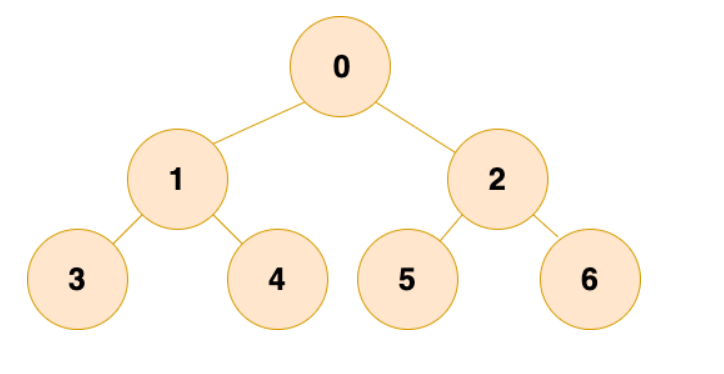
**二叉树**（Binary tree）是每个节点最多只有两个分支（即不存在分支度大于 2 的节点）的树结构。

**二叉树** 的第 i 层至多拥有 2^(i-1) 个节点，深度为 k 的二叉树至多总共有 2^(k+1)-1 个节点（满二叉树的情况），至少有 2^(k) 个节点

分类：

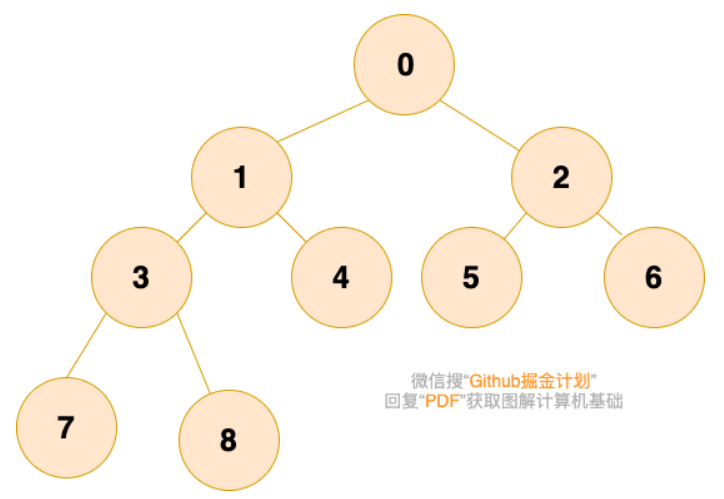
1. 满二叉树

一个二叉树，如果每一个层的结点数都达到最大值，则这个二叉树就是 **满二叉树**。也就是说，如果一个二叉树的层数为 K，且结点总数是(2^k) -1 ，则它就是 **满二叉树**。



1. 完全二叉树

除最后一层外，若其余层都是满的，并且最后一层或者是满的，或者是在右边缺少连续若干节点，则这个二叉树就是 **完全二叉树** 。



完全二叉树有一个很好的性质：父结点和子节点的序号有着对应关系。

当根节点的值为 1 的情况下，若父结点的序号是 i，那么左子节点的序号就是 2i，右子节点的序号是 2i+1。这个性质使得完全二叉树利用数组存储时可以极大地节省空间，以及利用序号找到某个节点的父结点和子节点。

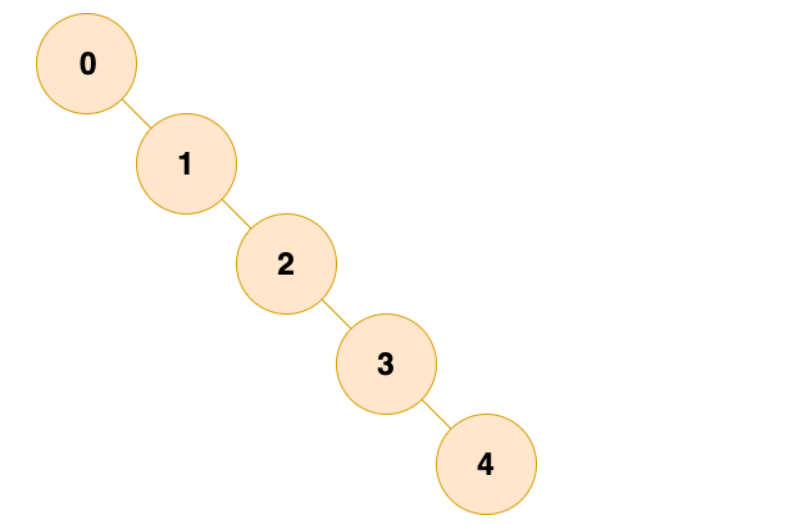
1. 平衡二叉树

平衡二叉树 是一棵**二叉排序树**，且具有以下性质：

* **可以是一棵空树**
* **如果不是空树，它的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。**

平衡二叉树的常用实现方法有 **红黑树、AVL 树、替罪羊树、加权平衡树、伸展树** 等。

斜树：



只不过这棵树已经退化为一个链表了，我们管它叫 斜树。

如果这样，那我为啥不直接用链表呢?谁说不是呢？

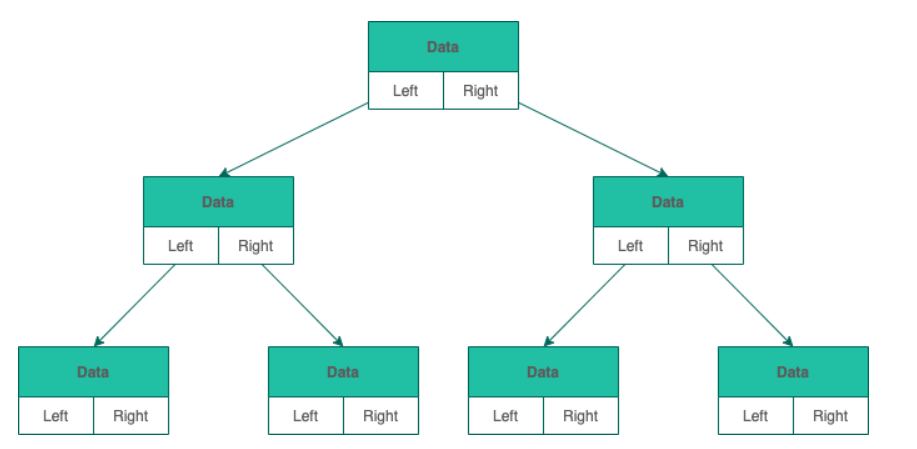
二叉树相比于链表，由于父子节点以及兄弟节点之间往往具有某种特殊的关系，这种关系使得我们在树中对数据进行搜索和修改时，相对于链表更加快捷便利。

二叉树的存储：分为 **链式存储** 和 **顺序存储**

链式存储：和链表类似，二叉树的链式存储依靠指针将各个节点串联起来，不需要连续的存储空间。

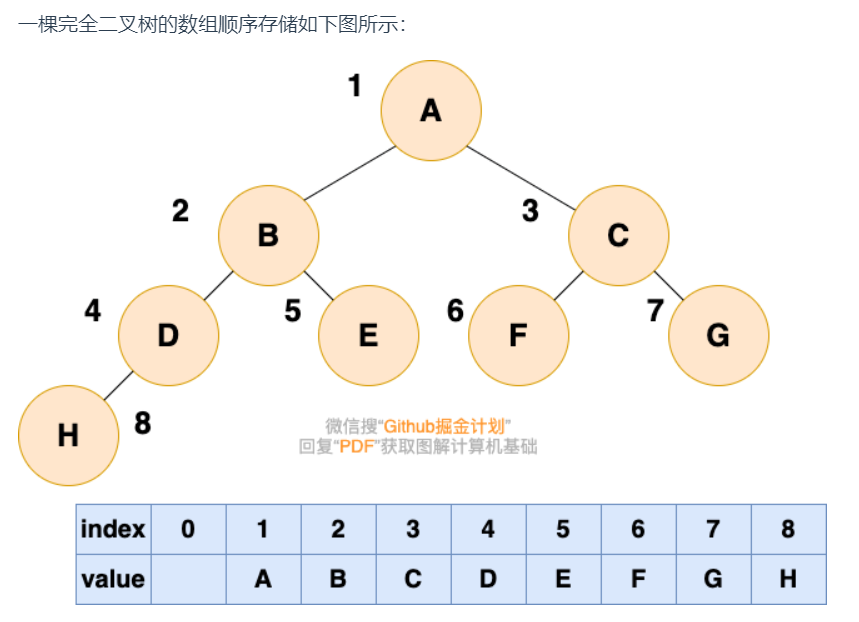
每个节点包括三个属性：

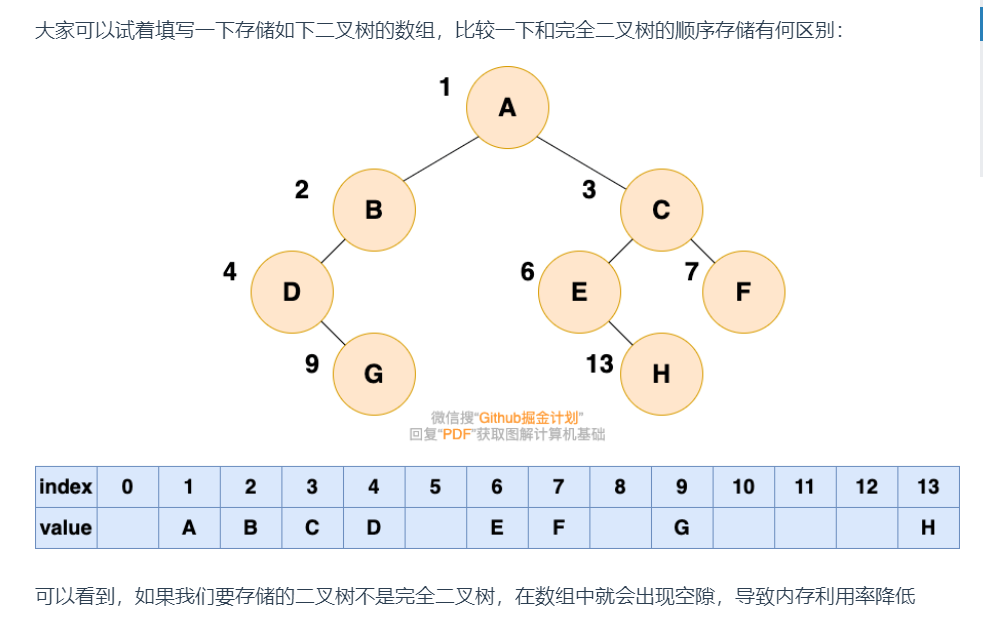
* 数据 data。data 不一定是单一的数据，根据不同情况，可以是多个具有不同类型的数据。
* 左节点指针 left
* 右节点指针 right。



顺序存储：

顺序存储就是利用数组进行存储，数组中的每一个位置仅存储节点的 data，不存储左右子节点的指针，子节点的索引通过数组下标完成。根结点的序号为 1，对于每个节点 Node，假设它存储在数组中下标为 i 的位置，那么它的左子节点就存储在 2 \_ i 的位置，它的右子节点存储在下标为 2 \_ i+1 的位置。





**二叉树的遍历**

1. 先序遍历---根左右

public void preOrder(TreeNode root){

if(root == null){

return;

}

system.out.println(root.data);

preOrder(root.left);

preOrder(root.right);

}

1. 中序遍历---左根右

public void inOrder(TreeNode root){

if(root == null){

return;

}

inOrder(root.left);

system.out.println(root.data);

inOrder(root.right);

}

1. 后序遍历---左右根

public void postOrder(TreeNode root){

if(root == null){

return;

}

postOrder(root.left);

postOrder(root.right);

system.out.println(root.data);

}

红黑树

红黑树特点 :

1. 每个节点非红即黑；
2. 根节点总是黑色的；
3. 每个叶子节点都是黑色的空节点（NIL节点）；
4. 如果节点是红色的，则它的子节点必须是黑色的（反之不一定）；
5. 从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）。

即：

1. 非红即黑
2. 根黑
3. 叶黑
4. 红子黑
5. 路径下黑相同

红黑树的应用 ：TreeMap、TreeSet以及JDK1.8的HashMap底层都用到了红黑树。

为什么要用红黑树？ 简单来说红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

布隆过滤器

布隆过滤器是一种由二进制向量（或者说位数组）和一系列随机映射函数（哈希函数）两部分组成的数据结构。

具有一定的错误识别率（无法避免），存放在布隆过滤器的数据不容易删除。

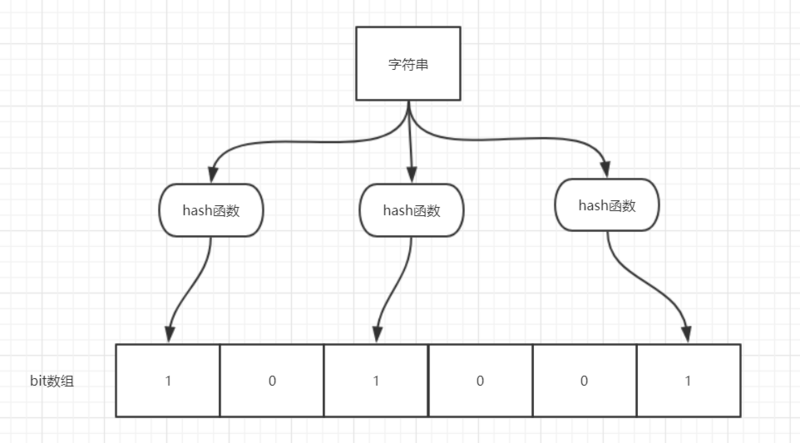
总结：一个名叫 Bloom 的人提出了一种来检索元素是否在给定大集合中的数据结构，这种数据结构是高效且性能很好的，但缺点是具有一定的错误识别率和删除难度。并且，理论情况下，添加到集合中的元素越多，误报的可能性就越大。

当一个元素加入布隆过滤器中的时候，会进行如下操作：

1. 使用布隆过滤器中的哈希函数对元素值进行计算，得到哈希值（有几个哈希函数得到几个哈希值）。
2. 根据得到的哈希值，在位数组中把对应下标的值置为 1。

当我们需要判断一个元素是否存在于布隆过滤器的时候，会进行如下操作：

1. 对给定元素再次进行相同的哈希计算；
2. 得到值之后判断位数组中的每个元素是否都为 1，如果值都为 1，那么说明这个值在布隆过滤器中，如果存在一个值不为 1，说明该元素不在布隆过滤器中。



如图所示，当字符串存储要加入到布隆过滤器中时，该字符串首先由多个哈希函数生成不同的哈希值，然后将对应的位数组的下标设置为 1（当位数组初始化时，所有位置均为 0）。当第二次存储相同字符串时，因为先前的对应位置已设置为 1，所以很容易知道此值已经存在（**去重非常方便**）。

如果我们需要判断某个字符串是否在布隆过滤器中时，只需要对给定字符串再次进行相同的哈希计算，得到值之后判断位数组中的每个元素是否都为 1，如果值都为 1，那么说明这个值在布隆过滤器中，如果存在一个值不为 1，说明该元素不在布隆过滤器中。

不同的字符串可能哈希出来的位置相同，这种情况我们可以适当增加位数组大小或者调整我们的哈希函数。

结论：**布隆过滤器说某个元素存在，小概率会误判。布隆过滤器说某个元素不在，那么这个元素一定不在。**

场景：

1. 判断给定数据是否存在：比如判断一个数字是否存在于包含大量数字的数字集中（数字集很大，5 亿以上！）、 防止缓存穿透（判断请求的数据是否有效避免直接绕过缓存请求数据库）等等、邮箱的垃圾邮件过滤、黑名单功能等等。
2. 去重：比如爬给定网址的时候对已经爬取过的 URL 去重。