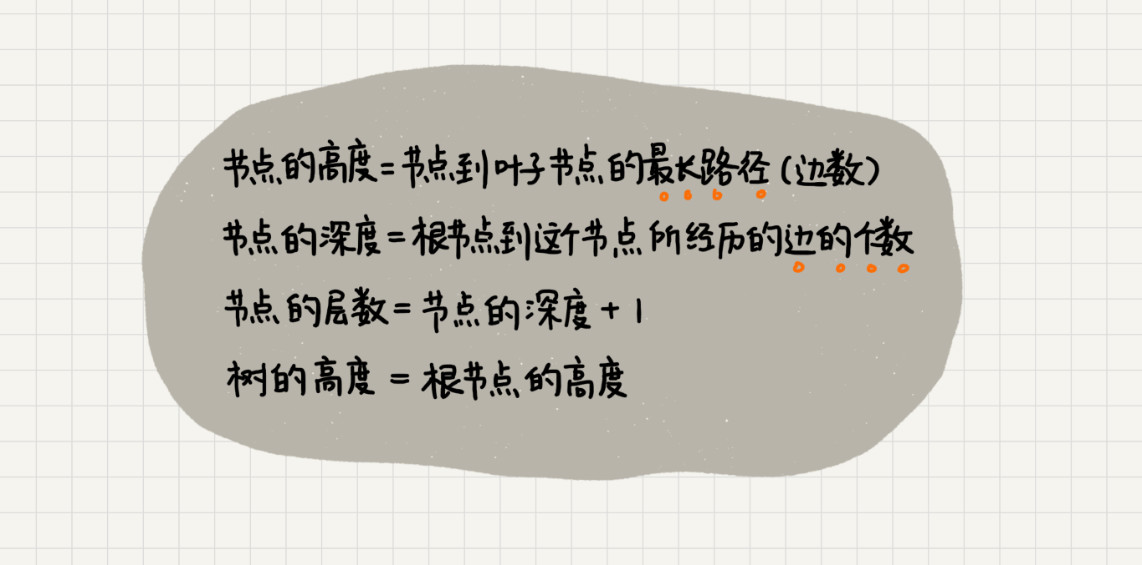
**一. 树**

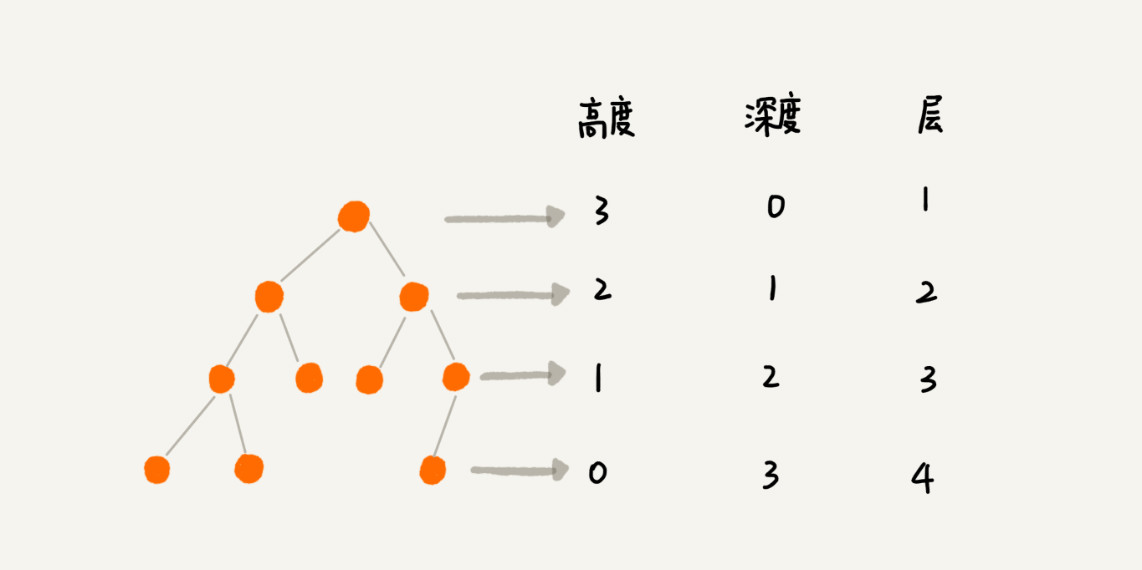
* 每个节点有零个或多个子节点
* 没有父节点的节点称为根节点
* 每一个非根结点有且只有一个父节点
* 除了根节点外，每个子节点可以分为多个不相交的字树

**二. 树的术语**

* 节点的度：一个节点含有的子树的个数
* 树的度：一棵树中，最大的节点的度
* 叶子节点：度为零的节点
* 父节点：若一个节点含有子节点，则这个节点成为其子节点的父节点
* 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
* 兄弟节点：具有相同父节点的节点互称为兄弟节点
* 节点的祖先：从根到该节点所经分支上的所有的节点
* 子孙：以某节点为根的子树中任意节点都成为该节点的子孙
* 森林：由m(m>=0)棵不相交的树的结合成为森林

除此之外，关于“树”，还有三个比较相似的概念：高度（Height）、深度（Depth）、层（Level）。它们的定义是这样的：

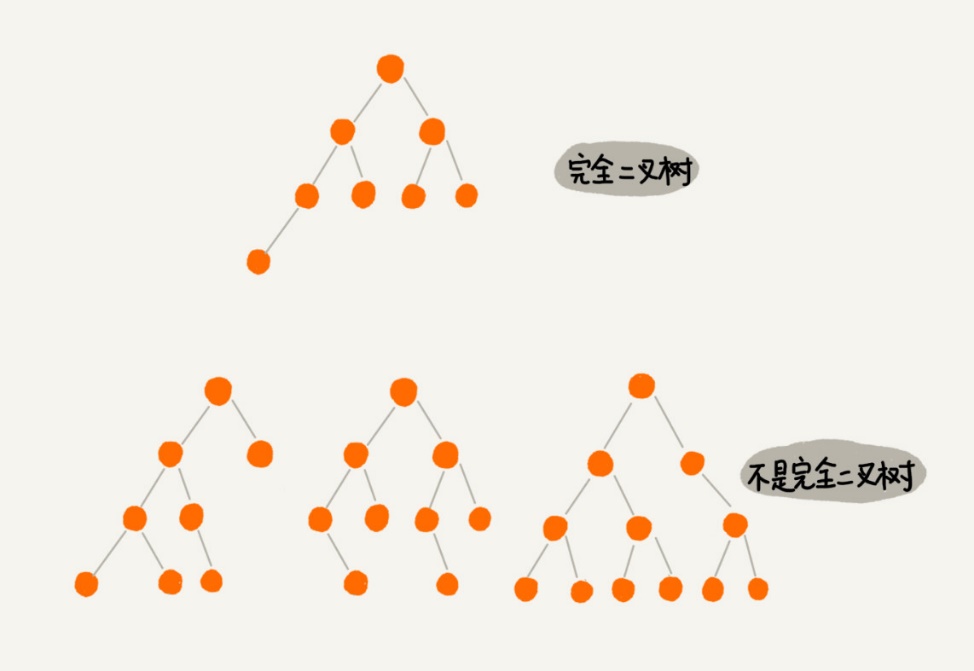




**三. 二叉树(Binary Tree)**

二叉树每个节点最多含有两个子树

1. 完全二叉树：除了最后一层以外，其余层的节点个数都要达到最大，最后一层的叶子节点从左向右连续紧密排列。



1. 满二叉树：叶子节点全部都在最底层，除了叶子节点以外，每个节点都有左右两个节点
2. 平衡二叉树(AVL树)：当且仅当任何节点的两棵子树高度差不大于1的二叉树
3. 二叉搜索树 Binary Search Tree: 在树中的任意一个节点，其左子树中的每个节点的值，都要小于这个节点的值，而右子树节点的值都大于这个节点的值。

Ps: 有序树：树中任意节点的子节点之间有顺序关系。包括二叉树，霍夫曼树（带权路径最短的二叉树，最优二叉树），B树（优化读写操作的自平衡二叉查找树，拥有多余两个子树）

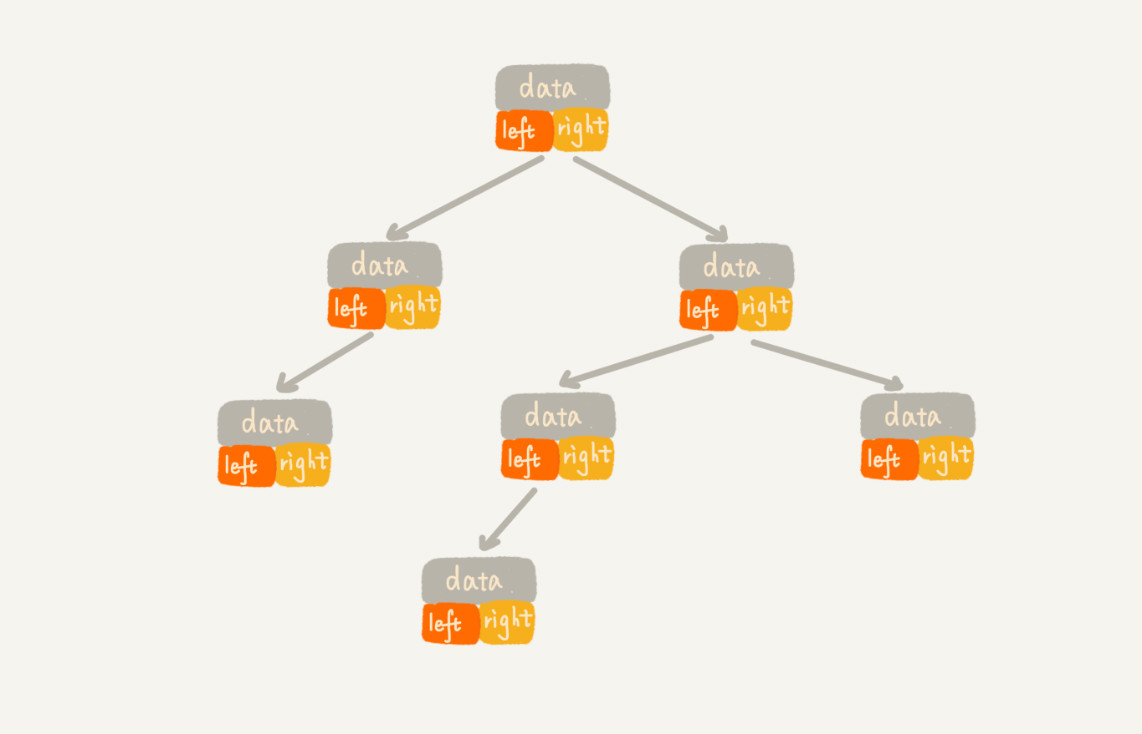
性质：

* 在二叉树第i层上至多有2^（i-1）个节点(i>0)
* 在最大层次为k的二叉树中至多有2^k – 1个节点(k>0)
* 具有n个节点的完全二叉树的向上取整有层
* 对于任意一棵二叉树，如果叶节点数为N0，而度数为2的节点总数为N2，则N0=N2+1

**四. 二叉树存储**

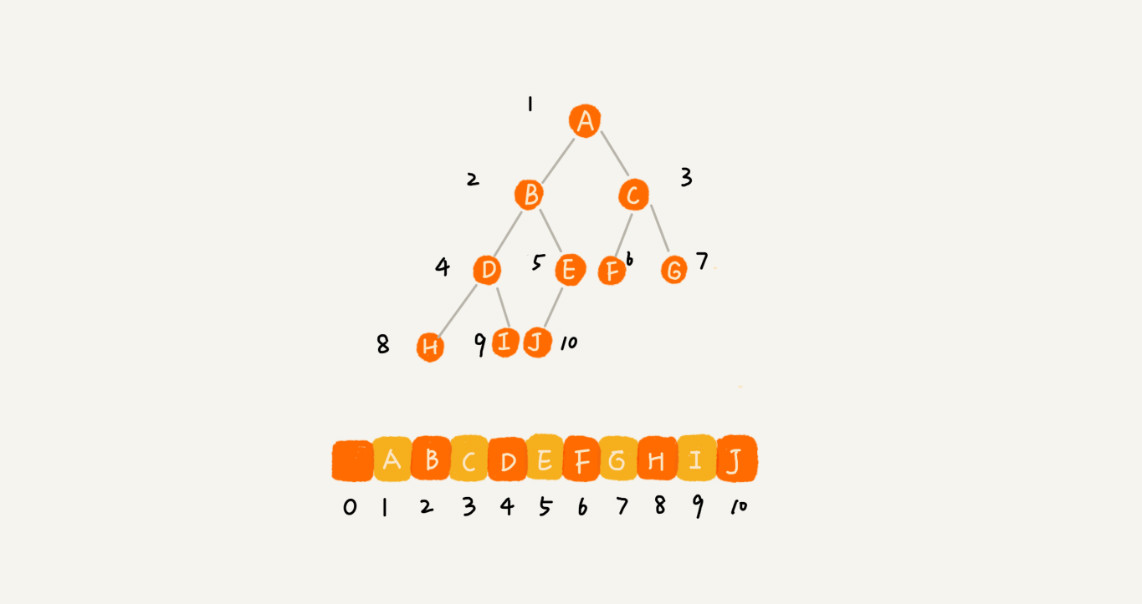
1. 链式存储法：

每个节点有三个字段，其中一个存储数据，另外两个是指向左右子节点的指针。我们只要拎住根节点，就可以通过左右子节点的指针，把整棵树都串起来。这种存储方式我们比较常用。大部分二叉树代码都是通过这种结构来实现的。



2. 顺序存储法:

基于数组的顺序存储法。我们把根节点存储在下标 i = 1 的位置。如果节点 X 存储在数组中下标为 i 的位置，下标为 2 \* i 的位置存储的就是左子节点，下标为 2 \* i + 1 的位置存储的就是右子节点。反过来，下标为 i/2 的位置存储就是它的父节点。

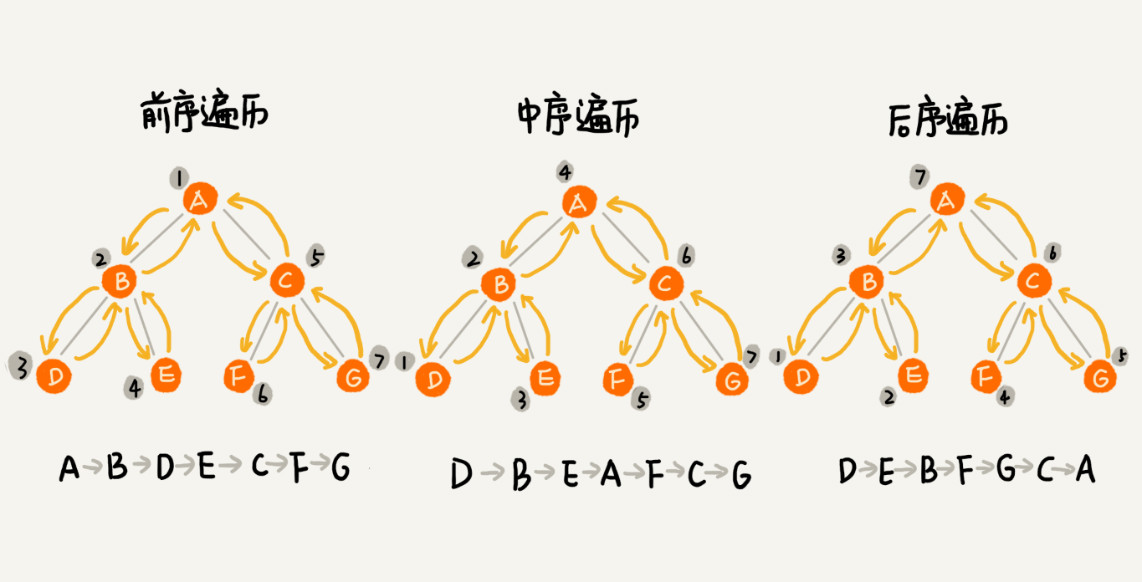


如果某棵二叉树是一棵完全二叉树，那用数组存储无疑是最节省内存的一种方式，仅浪费一个下标为0的存储位置。因为数组的存储方式并不需要像链式存储法那样，要存储额外的左右子节点的指针。这也是为什么完全二叉树要求最后一层的子节点都靠左的原因。

**五. 二叉树遍历**

经典的方法有三种，前序遍历、中序遍历和后序遍历。

* 前序遍历是指，对于树中的任意节点来说，先打印这个节点，然后再打印它的左子树，最后打印它的右子树。
* 中序遍历是指，对于树中的任意节点来说，先打印它的左子树，然后再打印它本身，最后打印它的右子树。
* 后序遍历是指，对于树中的任意节点来说，先打印它的左子树，然后再打印它的右子树，最后打印这个节点本身。



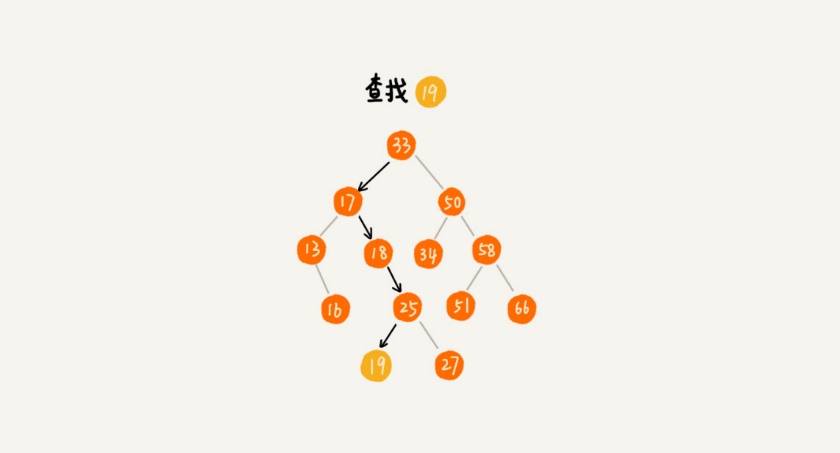
* 按层次遍历：借助队列思想，从上往下从左到右按层次遍历 ，O(n)。根节点先入队列，然后队列不空，取出对头元素，如果左子节点存在就入列队，否则什么也不做，右子节点同理。直到队列为空，则表示树层次遍历结束。树的层次遍历，其实也是一个广度优先的遍历算法。

Ps: 根据先序加中序或者后序加中序的结果可以确定一棵树

**六. 二叉查找树(Binary Search Tree)**

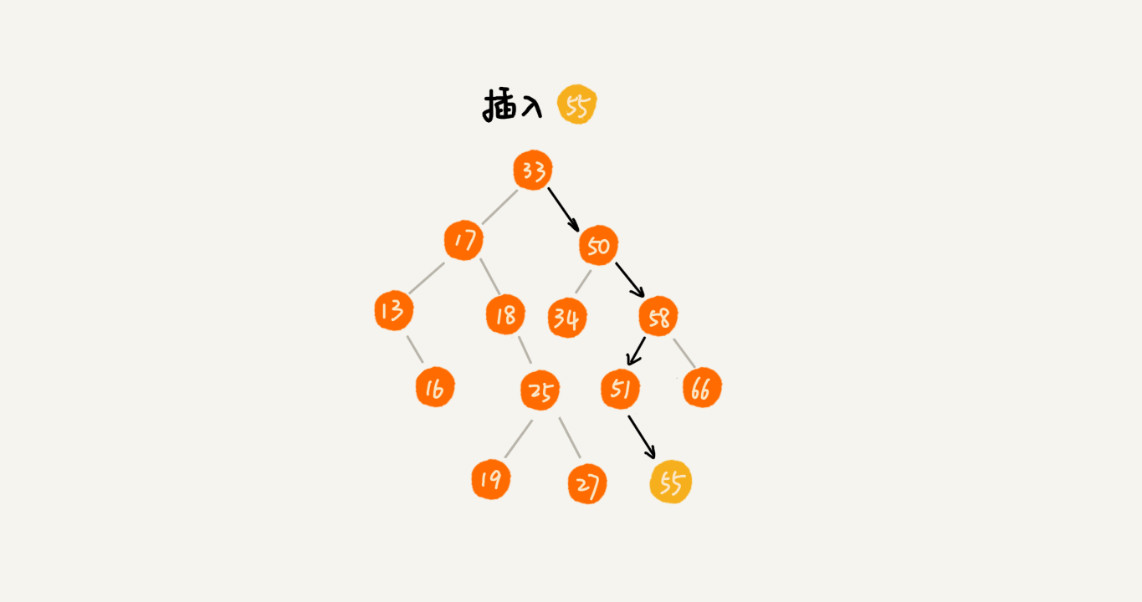
1. 二叉查找树的查找操作：

首先，我们看如何在二叉查找树中查找一个节点。我们先取根节点，如果它等于我们要查找的数据，那就返回。如果要查找的数据比根节点的值小，那就在左子树中递归查找；如果要查找的数据比根节点的值大，那就在右子树中递归查找。



2. 二叉查找树的插入操作:

如果要插入的数据比节点的数据大，并且节点的右子树为空，就将新数据直接插到右子节点的位置；如果不为空，就再递归遍历右子树，查找插入位置。同理，如果要插入的数据比节点数值小，并且节点的左子树为空，就将新数据插入到左子节点的位置；如果不为空，就再递归遍历左子树，查找插入位置。



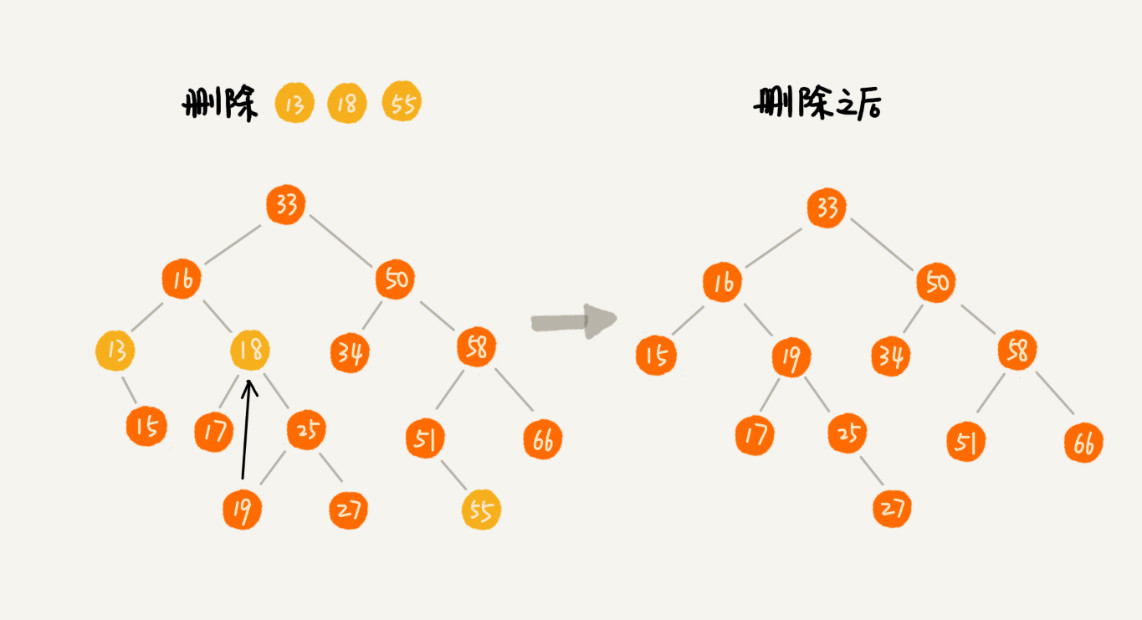
3. 二叉查找树的删除操作:

要删除节点的子节点个数的不同，我们需要分三种情况来处理。

第一种情况是，如果要删除的节点没有子节点，我们只需要直接将父节点中，指向要删除节点的指针置为 null。比如图中的删除节点 55。

第二种情况是，如果要删除的节点只有一个子节点（只有左子节点或者右子节点），我们只需要更新父节点中，指向要删除节点的指针，让它指向要删除节点的子节点就可以了。比如图中的删除节点 13。注：实际编写代码遇到，要删除的结点与其仅有的子节点含有的数值相同，这种情况需要删除两个结点需要一个elif判定语句。

第三种情况是，如果要删除的节点有两个子节点，这就比较复杂了。我们需要找到这个节点的右子树中的最小节点，把它替换到要删除的节点上。然后再删除掉这个最小节点，因为最小节点肯定没有左子节点（如果有左子结点，那就不是最小节点了），所以，我们可以应用上面两条规则来删除这个最小节点。比如图中的删除节点 18。



除了插入、删除、查找操作之外，二叉查找树中还可以支持快速地查找最大节点和最小节点、前驱节点和后继节点。

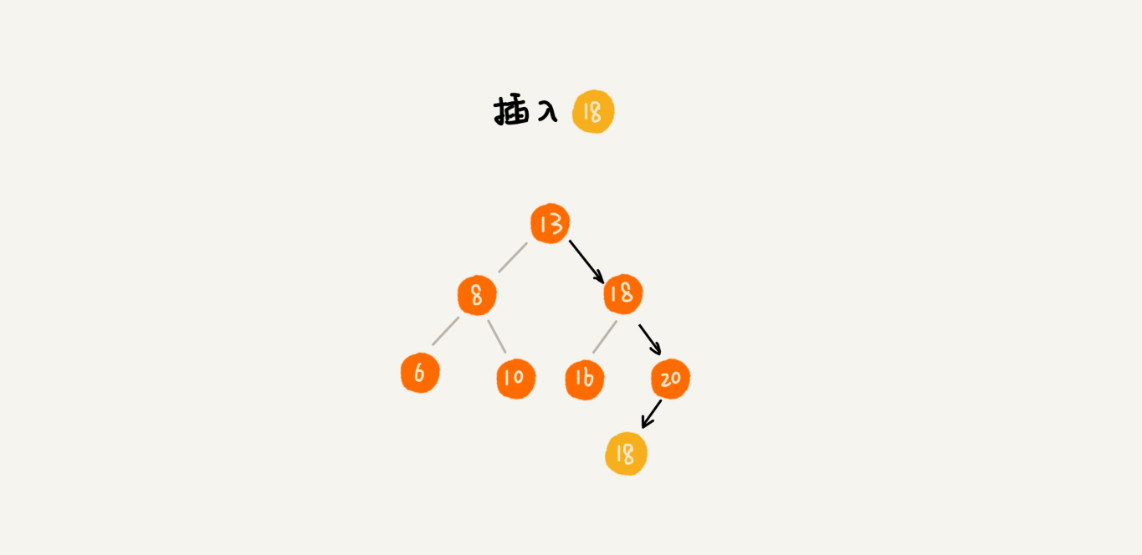
中序遍历二叉查找树，可以输出有序的数据序列，时间复杂度是 O(n)，非常高效。

**七. 支持重复数据的二叉查找树**

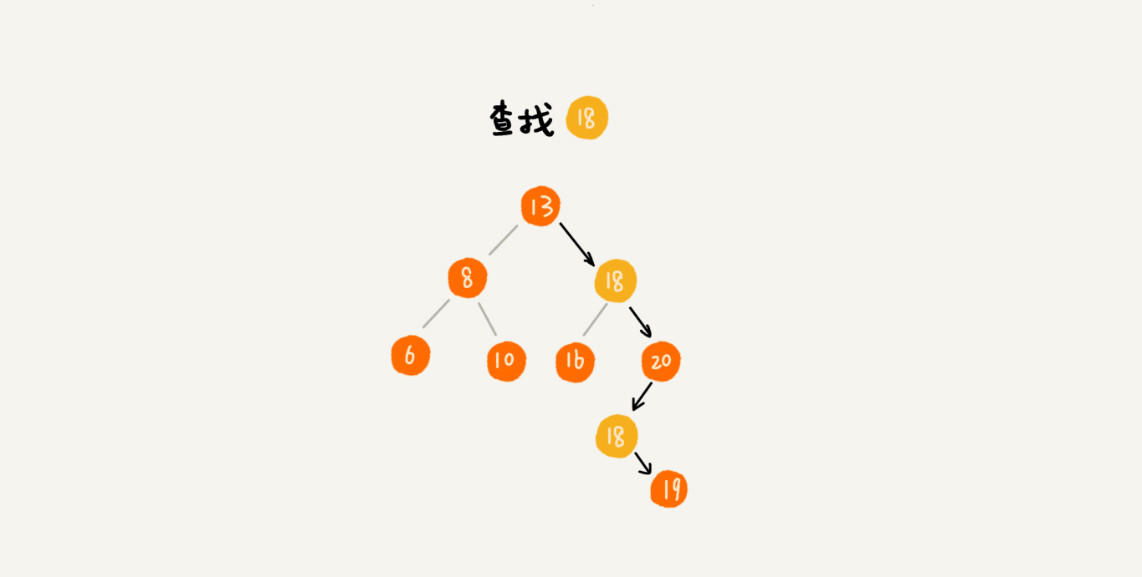
前面我们讲的二叉查找树的操作，针对的都是不存在键值相同的情况。那如果存储的两个对象键值相同，这种情况该怎么处理呢？我这里有两种解决方法。

第一种方法比较容易。二叉查找树中每一个节点不仅会存储一个数据，因此我们通过链表和支持动态扩容的数组等数据结构，把值相同的数据都存储在同一个节点上。

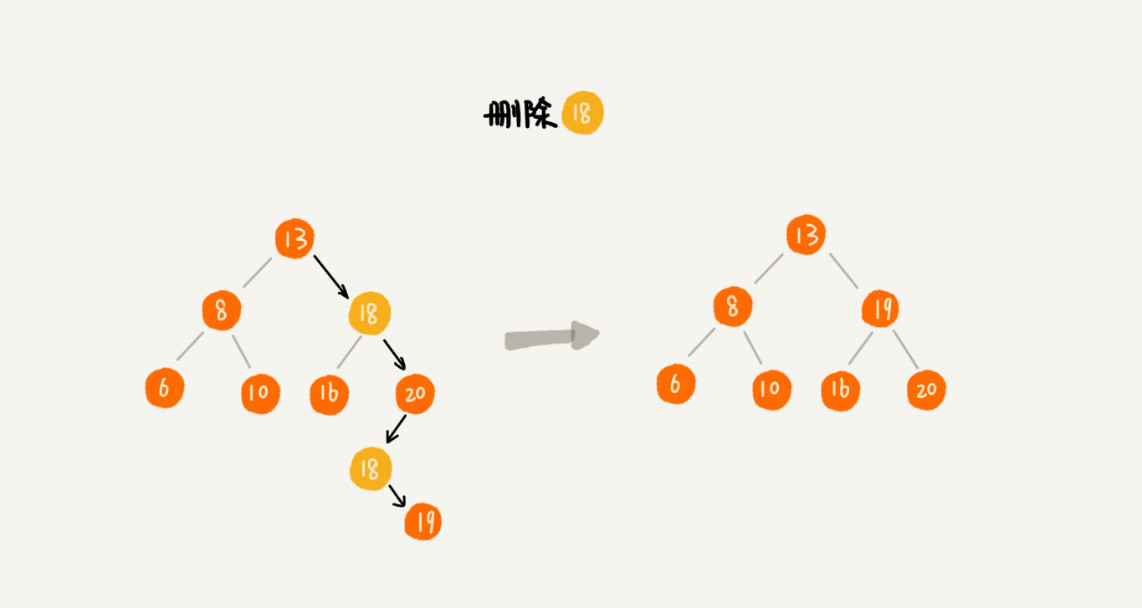
第二种方法比较不好理解，不过更加优雅。每个节点仍然只存储一个数据。在查找插入位置的过程中，如果碰到一个节点的值，与要插入数据的值相同，我们就将这个要插入的数据放到这个节点的右子树，也就是说，把这个新插入的数据当作大于这个节点的值来处理。



当要查找数据的时候，遇到值相同的节点，我们并不停止查找操作，而是继续在右子树中查找，直到遇到叶子节点，才停止。这样就可以把键值等于要查找值的所有节点都找出来。



对于删除操作，我们也需要先查找到每个要删除的节点，然后再按前面讲的删除操作的方法，依次删除。



**八. 二叉查找树的时间复杂度分析**

我们现在来分析一个最理想的情况，二叉查找树是一棵完全二叉树（或满二叉树）。这个时候，插入、删除、查找的时间复杂度是多少呢？

从我前面的例子、图，以及还有代码来看，不管操作是插入、删除还是查找，时间复杂度其实都跟树的高度成正比，也就是 O(height)。既然这样，现在问题就转变成另外一个了，也就是，如何求一棵包含 n 个节点的完全二叉树的高度？

树的高度就等于最大层数减一，为了方便计算，我们转换成层来表示。从图中可以看出，包含 n 个节点的完全二叉树中，第一层包含 1 个节点，第二层包含 2 个节点，第三层包含 4 个节点，依次类推，下面一层节点个数是上一层的 2 倍，第 K 层包含的节点个数就是 2^(K-1)。不过，对于完全二叉树来说，最后一层的节点个数有点儿不遵守上面的规律了。它包含的节点个数在 1 个到 2^(L-1) 个之间（我们假设最大层数是 L）。如果我们把每一层的节点个数加起来就是总的节点个数 n。也就是说，如果节点的个数是 n，那么 n 满足这样一个关系：

**n >= 1+2+4+8+...+2^(L-2)+1；n <= 1+2+4+8+...+2^(L-2)+2^(L-1)**

借助等比数列的求和公式，我们可以计算出，L 的范围是[log2(n+1), log2n +1]。完全二叉树的层数小于等于 log2n +1，也就是说，完全二叉树的高度小于等于 log2n。所以理想状态时间复杂度O(logn)

**九. 既然有了这么高效的散列表，使用二叉树的地方是不是都可以替换成散列表呢？有没有哪些地方是散列表做不了，必须要用二叉树来做的呢？**

我们在散列表那节中讲过，散列表的插入、删除、查找操作的时间复杂度可以做到常量级的 O(1)，非常高效。而二叉查找树在比较平衡的情况下，插入、删除、查找操作时间复杂度才是 O(logn)，相对散列表，好像并没有什么优势，那我们为什么还要用二叉查找树呢？我认为有下面几个原因：

第一，散列表中的数据是无序存储的，如果要输出有序的数据，需要先进行排序。而对于二叉查找树来说，我们只需要中序遍历，就可以在 O(n) 的时间复杂度内，输出有序的数据序列。

第二，散列表扩容耗时很多，而且当遇到散列冲突时，性能不稳定，尽管二叉查找树的性能不稳定，但是在工程中，我们最常用的平衡二叉查找树的性能非常稳定，时间复杂度稳定在 O(logn)。

第三，笼统地来说，尽管散列表的查找等操作的时间复杂度是常量级的，但因为哈希冲突的存在，这个常量不一定比 logn 小，所以实际的查找速度可能不一定比 O(logn) 快。加上哈希函数的耗时，也不一定就比平衡二叉查找树的效率高。

第四，散列表的构造比二叉查找树要复杂，需要考虑的东西很多。比如散列函数的设计、冲突解决办法、扩容、缩容等。平衡二叉查找树只需要考虑平衡性这一个问题，而且这个问题的解决方案比较成熟、固定。

最后，为了避免过多的散列冲突，散列表装载因子不能太大，特别是基于开放寻址法解决冲突的散列表，不然会浪费一定的存储空间。综合这几点，平衡二叉查找树在某些方面还是优于散列表的，所以，这两者的存在并不冲突。我们在实际的开发过程中，需要结合具体的需求来选择使用哪一个。

**十. 确定树的高度**

确定二叉树高度有两种思路：第一种是深度优先思想的递归，分别求左右子树的高度。当前节点的高度就是左右子树中较大的那个+1；第二种可以采用层次遍历的方式，每一层记录都记录下当前队列的长度，这个是队尾，每一层队头从0开始。然后每遍历一个元素，队头下标+1。直到队头下标等于队尾下标。这个时候表示当前层遍历完成。每一层刚开始遍历的时候，树的高度+1。最后队列为空，就能得到树的高度。Leetcode 104