数据结构和算法（三）

1：今日课程内容介绍(了解)

今日课程中我们主要来学习以下几个知识模块：

- 散列表
- 哈希算法
- 树

2：今日课程目标(了解)

1：了解散列表的定义

2：掌握散列表的要求和特点

3：理解散列函数的设计方法

4：掌握散列冲突的解决方案

5：掌握散列表的应用：HashMap 源码分析

6：掌握哈希算法的应用场景

7：理解树的定义

8：掌握二叉树的定义及遍历方式

9：掌握二叉查找树的插入，删除，查询等相关操作

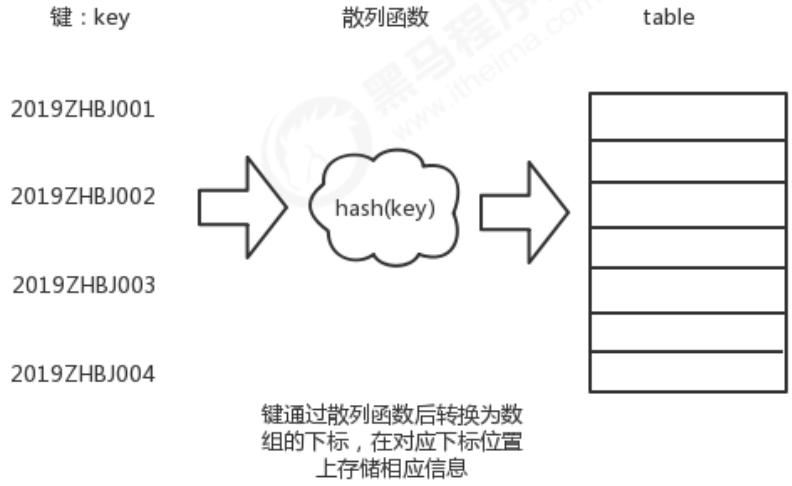
10：掌握 AVL 树的特点及实现

3：散列表(重要)

3.1.散列表概述(理解)

3.1.1、散列表概念

散列表(Hash Table)又名哈希表/Hash 表，是根据键（Key）直接访问在内存存储位置的数据结构，它是由数组演化而来的，利用了数组支持按照下标进行随机访问数据的特性；接下来我们以一个具体的例子来说明散列的思想及原理。



3.2.散列函数(列举并掌握)

3.2.1.散列函数的要求及特点

散列函数就是一个函数(方法)，能够将给定的 key 转换为特定的散列值，我们可以表示为：hashValue = hash(key)

散列函数要满足的几个基本要求：

1：散列函数计算得到的散列值必须是大于等于 0 的正整数，因为 hash 值需要作为数组的下标。

2：如果 key1==key2，那么经过 hash 后得到的哈希值也必相同即：hash(key1) == hash(key2)

3：如果 key1 != key2，那么经过 hash 后得到的哈希值也必不相同即：hash(key1) != hash(key2)

好的散列函数应该满足以下特点：

1：散列函数不能太复杂，因为太复杂度势必要消耗很多的时间在计算哈希值上，也会间接影响散列表性能。

2：散列函数计算得出的哈希值尽可能的能随机并且均匀的分布，这样能够将散列冲突最小化。

3.2.2.散列函数的设计方法

实际工作中，我们还需要综合考虑各种因素。这些因素有关键字的长度、特点、分布、还有散列表的大小等。散列函数各式各样，我举几个常用的、简单的散列函数的设计方法。

- **1：直接寻址法：**

比如我们现在要对 0-100 岁的人口数字统计表，那么我们对年龄这个关键字 key 就可以直接用年龄的数字作为地址。此时 hash(key) = key。这个时候，我们可以得出这么个哈希函数：hash(0) = 0，hash(1) = 1，……，hash(20) = 20。

地址	年龄	人数
00	0	500万
01	1	600万
02	2	450万
…	…	…
20	20	1500万
…	…	…

如果我们现在要统计的是 1980 年后出生年份的人口数，那么我们对出生年份这个关键字可以用年份减去 1980 来作为地址。此时 hash(key) = key-1980

地址	出生年份	人数
00	1980	1500万
01	1981	1600万
02	1982	1300万
…	…	…
20	2000	500万
…	…	…

也就是说，我们可以**取关键字 key 的某个线性函数值为散列地址，即：**

hash(key) = a x key + b，其中 a,b 为常量

这样的散列函数优点就是简单、均匀，也不会产生冲突，但问题是这需要事先知道关键字 key 的分布情况，适合查找表较小且连续的情况。由于这样的限制，在现实应用中，直接寻址法虽然简单，但却并不常用。

- **2：除留余数法**

除留余数法此方法为最常用的构造散列函数方法。**对于散列表长为 m 的散列函数公式为：**

hash(key) = key mod p (p ≤ m)

本方法的关键就在于选择合适的 p，p 如果选得不好，就可能会容易产生哈希冲突，比如：有 12 个关键字 key，现在我们要针对它设计一个散列表。如果采用除留余数法，那么可以先尝试将散列函数设计为 hash(key) = key mod 12 的方法。比如 29 mod 12 = 5，所以它存储在下标为 5 的位置。

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25	38	15	16	29	78	67	56	21	22	47

不过这也是存在冲突的可能的，因为 12 = 2×6 = 3×4。如果关键字中有像 18(3×6)、30(5×6)、42(7×6)等数字，它们的余数都为 6，这就和 78 所对应的下标位置冲突了。此时如果我们不选用 p=12 而是选用 p=11 则结果如下：

下标	1	2	3	4	5	6	7	8	9	10	0	1
关键字	12	24	36	48	60	72	84	96	108	120	132	144

使用除留余数法的一个经验是，若散列表表长为 m，通常 p 为小于或等于表长（最好接近 m)的最大质数或不包含小于 20 质因子的合数。总之实践结果证明：**当 P 取小于哈希表长的最大质数时，产生的哈希函数较好**。

3：平方取中法：

这是一种常用的哈希函数构造方法。这个方法是先取关键字的平方，然后根据可使用空间的大小，选取平方数是中间几位为哈希地址。

hash(key) = key 平方的中间几位

这种方法的原理是通过取平方扩大差别，平方值的中间几位和这个数的每一位都相关，则对不同的关键字得到的哈希函数值不易产生冲突，由此产生的哈希地址也较为均匀。

关键字	关键字的平方	哈希函数值
1234	1522756	227
2143	4592449	924
4132	17073424	734
3214	10329796	297

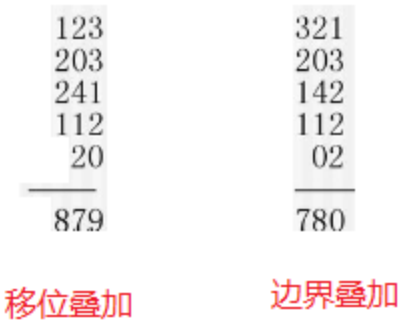
4：折叠法：

有时关键码所含的位数很多，采用平方取中法计算太复杂，则可将关键码分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为散列地址，这方法称为折叠法，折叠法可分为两种：

移位叠加：将分割后的几部分低位对齐相加。

边界叠加:从一端沿分割界来回折叠，然后对齐相加。

比如关键字为:12320324111220，分为 5 段，123，203，241，112，20，两种方式如下：



当然了散列函数的设计方法不仅仅只有这些方法，对于这些方法我们无需全部掌握也不需要死记硬背，我们理解其设计原理即可。

3.2.3 .散列冲突

两个不同的关键字（**key**），由于散列函数值相同，因而被映射到同一表位置上。该现象称为散列冲突或哈希碰撞。

3.2.4.散列冲突的解决方案

前面我们讲到即使再好的散列函数我们也无法避免不了散列冲突(哈希冲突，哈希碰撞)，那如果真的出现了散列冲突我们应该如何来解决散列冲突呢？在本节中我们来介绍**两类方法解决散列冲突：开放寻址法，链表法**

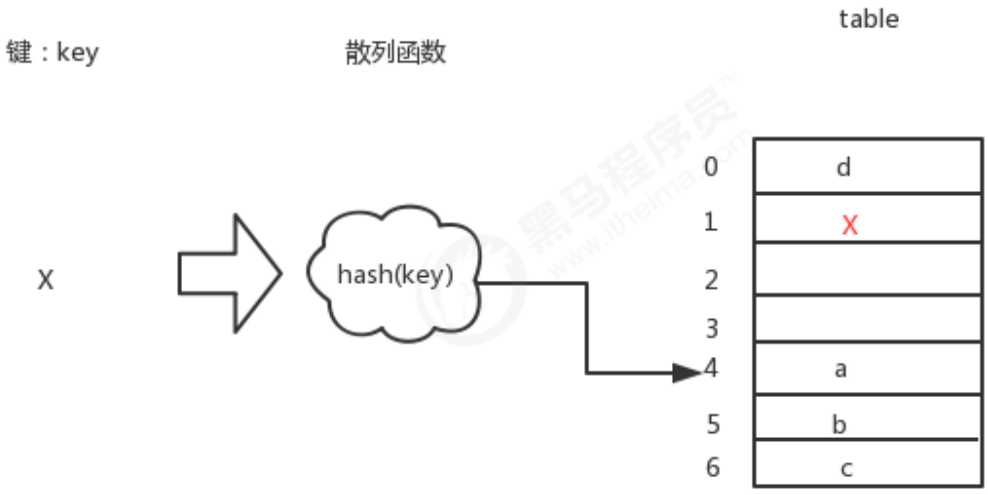
（1）、开放寻址法

开放寻址法的核心思想是：一旦出现了散列冲突，我们就重新去寻址一个空的散列地址。

①、线性检测

我们往散列表中插入数据时，如果某个数据经过散列函数散列之后，存储位置已经被占用了，我们就从当前位置开始，依次往后查找，看是否有空闲位置，直到找到为止。

图示说明如下：

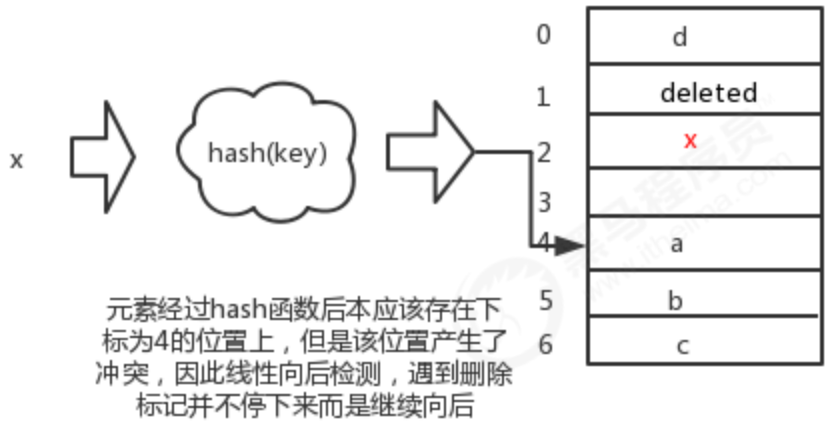


散列表的大小为 7，在元素 X 插入之前已经有 a，b，c，d 四个元素插入到散列表中了，元素 X 经过 hash(X)计算之后得到的哈希值为 4，但是 4 这个位置已经有数据了，所以产生了冲突，于是我们需要按照顺序依次向后查找，一直查找到数组的尾部都没有空闲位置了，所以再从头开始查找，直到找到空闲位置下标为 1 的位置，至此将元素 X 插入下标为 1 的位置。

我们刚刚所讲的是向散列表中插入数据，如果要从散列表中查找是否存在某个元素，这个过程跟插入类似，先根据散列函数求出要查找元素的 key 的散列值，然后比较数组中下标为其散列值的元素和要查找的元素，如果相等则表明该元素就是我们想要的元素，如果不等还要继续向后寻找遍历，如果遍历到数组中的空闲位置还没有找到则说明我们要找的元素并不在散列表中。

当然了散列表跟数组一样，不仅支持插入、查找操作，还支持删除操作。其中删除操作稍微有点特殊，删除操作不能简单的将要删除的位置设置为空，为什么呢？

上面刚讲到从散列表中查找是否存在某个元素一旦在对应 hash 值下标下的元素不是我们想要的就会继续在散列表中向后遍历，直到找到数组中的空闲位置，但如果这个空闲位置是我们刚刚删除的，那就会中断向后查找的过程，那这样的话查找的算法就会失效，本来应该认定为存在的元素会被认定为不存在，那删除的问题如何解决呢？我们可以将删除的元素特殊标记为 deleted，当线性检测遇到标记 deleted 的时候并不停下来而是继续向后检测，如下图所示：



使用线性检测的方式存在很大的问题：那就是当散列表中的数据越来越多的时候，散列冲突发生的可能性就越来越大，空闲的位置越来越少，那线性检测的时间就会越来越长，在极端情况下我们可能需要遍历整个数组，所以最坏的情况下时间复杂度为 O(n)，因此对于开放寻址解决冲突还有另外两种比较经典的的检测方式：**二次检测，双重散列**。

②、二次检测

所谓的二次检测跟线性检测的原理一样，只不过线性检测每次检测的步长是 1，每次检测的下标依次是： hash(key)+0，hash(key)+1，hash(key)+2，hash(key)+3.....，所谓的二次检测指的是每次检测的步长变为原来的二次方，即每次检测的下标为

$$hash(key)+0, hash(key)+1^2, hash(key)+2^2, hash(key)+3^2, ...$$

③、双重散列

所谓的双重散列，意思就是不仅要使用一个散列函数。我们使用一组散列函数 hash1(key)，hash2(key)，hash3(key).....我们先用第一个散列函数，如果计算得到的存储位置已经被占用，再用第二个散列函数，依次类推，直到找到空闲的存储位置。

装载因子：

总之不管采用哪种探测方法，当散列表中空闲位置不多的时候，散列冲突的概率就会大大提高。为了尽可能保证散列表的操作效率，一般情况下，我们会尽可能保证散列表中有一定比例的空闲位置。我们用**装载因子(load factor)**来表示空位的多少。散列表装载因子的计算公式为：

装载因子 = 散列表中元素的个数 / 散列表的长度

装载因子越大，说明空闲位置越少，冲突越多，散列表的性能会下降。那**如果装载因子过大了怎么办**？装载因子过大不仅插入的过程中要多次寻址，查找的过程也会变得很慢。当装载因子过大时，进行**动态扩容**，重新申请一个更大的散列表，将数据搬移到这个新散列表中。假设每次扩容我们都申请一个原来散列表大小两倍的空间。如果原来散列表的装载因子是 0.8，那经过扩容之后，新散列表的装载因子就下降为原来的一半，变成了 0.4。针对数组的扩容，数据搬移操作比较简单。但是，针对散列表的扩容，数据搬移操作要复杂很多。因为散列表的大小变了，数据的存储位置也变了，所以我们需要通过散列函数重新计算每个数据的存储位置。

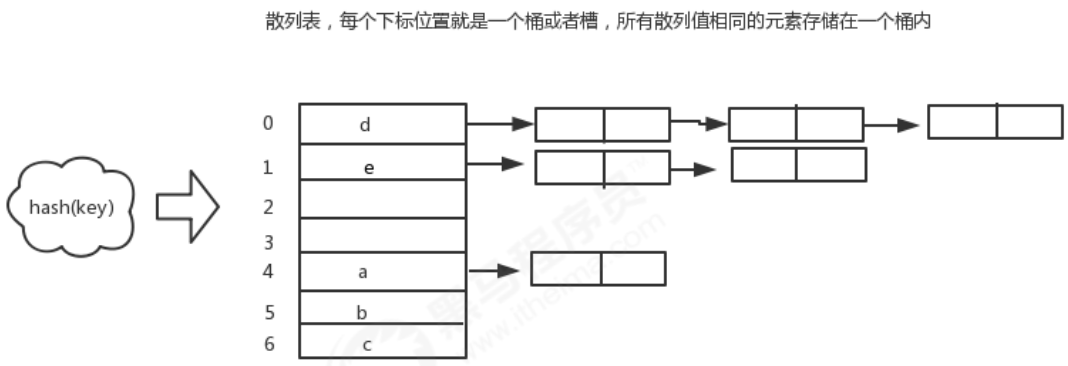
插入一个数据，最好情况下，不需要扩容，最好时间复杂度是 O(1)。最坏情况下，散列表装载因子过高，启动扩容，我们需要重新申请内存空间，重新计算哈希位置，并且搬移数据，所以时间复杂度是 O(n)。但是这个动态扩容的过程在 n 次操作中会遇见一次，因此平均下来时间复杂度接近最好情况，就是 O(1)。

当散列表的装载因子超过某个阈值时，就需要进行扩容。装载因子阈值需要选择得当。如果太大，会导致冲突过多；如果太小，会导致内存浪费严重。装载因子阈值的设置要权衡时间、空间复杂度。如果内存空间不紧张，对执行效率要求很高，可以降低负载因子的阈值；相反，如果内存空间紧张，对执行效率要求又不高，可以增加负载因子的值，甚至可以大于 1。

总结一下，当数据量比较小、装载因子小的时候，适合采用开放寻址法。这也是 Java 中的 ThreadLocalMap 使用开放寻址法解决散列冲突的原因。

（2）、链表法

相比开放寻址法，它要简单很多。我们来看这个图，在散列表中，数组的每个下标位置我们可以称之为“桶（bucket）”或者“槽（slot）”，每个桶(槽)会对应一条链表，所有散列值相同的元素我们都放到相同槽位对应的链表中。



基于链表的散列冲突处理方法比较适合存储大对象、大数据量的散列表，而且，比起开放寻址法，它更加灵活，支持更多的优化策略，比如用红黑树代替链表。

综合本章节所学的知识点，我们知道作为一个企业级的散列表应该具有如下特点：

- 支持快速的查询、插入、删除操作；
- 内存占用合理，不能浪费过多的内存空间；
- 性能稳定，极端情况下，散列表的性能也不会退化到无法接受的情况。

我们要实现这样一个散列表应该从如下几个方面来考虑设计思路：

- 设计一个合适的散列函数；
- 定义装载因子阈值，并且设计动态扩容策略；
- 选择合适的散列冲突解决方法。

3.3.散列表的应用(掌握并应用)

HashMap 的数据结构图如下图所示：

jdk1.8 中关于 HashMap 的实现跟 jdk1.7 的几点差别：

1：数据结构引入了红黑树，好处是可以提高查询效率(jdk1.7 中极端情况下查询是 O(n)，如果引入红黑树在极端情况下的查询可以降低为 O(log n))，当散列表某一桶内链表节点数>=8 时链表树化成红黑树，红黑树太小时退化成链表，退化的阈值为 6

2：计算 key 的 hash 值的方式不一样，但是思路和原理一样都是对 key 的 hashCode 进行扰动让高位和低位一起参与运算计算出更加均匀的 hash 码，降低 hash 冲突的概率。

3：插入数据时如果发送了 hash 冲突，优先判断该位置上是否是红黑树，如果是则存入红黑树中，如果是链表则插入链表尾节点上(jdk1.7 是插入到链表头节点上)，插入完成后还判断链表的节点数是否大于等于设定好的链表转红黑树的阈值，如果满足则将链表转换为红黑树。

4：两个版本都会产生扩容操作，只不过 jdk1.8 中扩容涉及到对红黑树的操作以及优化了在 hash 冲突时计算元素新下标的代码，使其非常简单高效！

3.4.哈希算法(理解并掌握)

3.4.1.定义

哈希算法又称为摘要算法，它可以将任意数据通过一个函数转换成长度固定的数据串，这个映射转换的规则就是哈希算法，而通过原始数据映射之后得到的二进制值串就是哈希值。 可见，摘要算法就是通过**摘要函数 f()**对**任意长度的数据 data** 计算出**固定长度的摘要 digest**，目的是为了发现原始数据**是否被人篡改过**。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个**单向函数**，计算 f(data)很容易，但通过 digest 反推 data 却非常困难。而且，对原始数据做一个 bit 的修改，都会导致计算出的摘要完全不同。

那有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要呢？完全有可能！因为任何摘要算法都是把无限多的数据集合映射到一个**有限的集合**中。这种情况就是我们说的**碰撞**。

3.4.2.要求

我们要想设计出一个优秀的哈希算法并不是很容易，一个优秀的哈希算法一般要满足如下几点要求：

1. **将任何一条不论长短的信息，计算出唯一的一摘要(哈希值)与它相对应，对输入数据非常敏感，哪怕原始数据只修改了一个 Bit，最后得到的哈希值也大不相同**
2. **摘要的长度必须固定，散列冲突的概率要很小，对于不同的原始数据，哈希值相同的概率非常小**
3. **摘要不可能再被反向破译。也就是说，我们只能把原始的信息转化为摘要，而不可能将摘要反推回去得到原始信息，即哈希算法是单向的**

4. **哈希算法的执行效率要尽量高效，针对较长的文本，也能快速地计算出哈希值**

这些要求都是比较理论的说法，我们那一种企业常用的哈希算法 MD5 来说明：现使用 MD5 对三段数据分别进行哈希求值：

1.MD5('数据结构和算法') = 31ea1cbbe72095c3ed783574d73d921e

2.MD5('数据结构和算法很好学')=0fba5153bc8b7bd51b1de100d5b66b0a

3.MD5('数据结构和算法不好学')=85161186abb0bb20f1ca90edf3843c72

从其结果我们可以看出：MD5 的摘要值(哈希值)是固定长度的，是 16 进制的 32 位即 128 Bit 位，无论要哈希的数据有多长，多短，哈希之后的数据长度是固定的，另外哈希值是随机的无规律的，无法根据哈希值反向推算文本信息，其次 2，3 表明尽管只有一字之差得到的结果也是千差万别，最后哈希的速度和效率是非常高的，这一点我们可能还体会不到，因为我们哈希的只是很短的一串数据，即便我们哈希的是整个这段文本，用 MD5 计算哈希值，速度也是非常的快，总之 MD5 基本满足了我们前面所讲解的这几个要求。

在本章节中我们学习了散列表数据结构，掌握了散列函数的特点及设计要求，明确了其中几种设计方案，知道了散列冲突的原理以及解决散列冲突的方案，对于散列表在企业中的应用我们重点分析了 HashMap 和 HashTable 的源码，最后我们介绍了哈希算法，重点是阐述了哈希算法的应用场景。

4：树(重要)

在前面章节的中我们学习了线性表数据结构：数组，链表，栈，队列；在这章中我们来学习一种非线性表叫做：树。我们先来看树的定义及相关概念

4.1.树的定义及相关概念(理解并掌握)

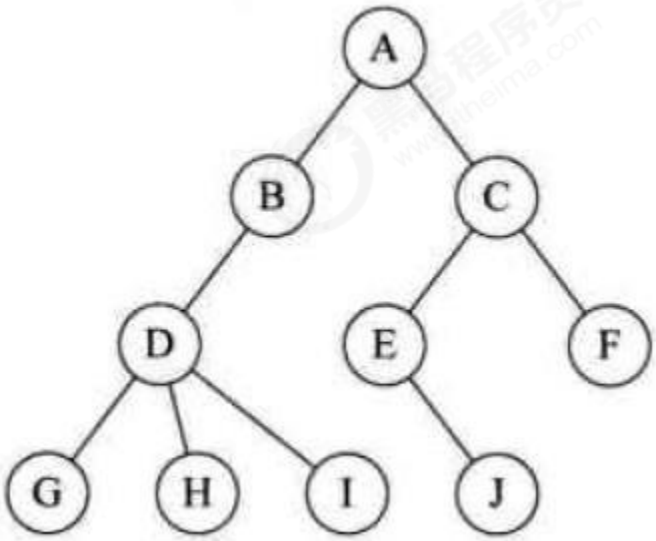
4.1.1、树的定义

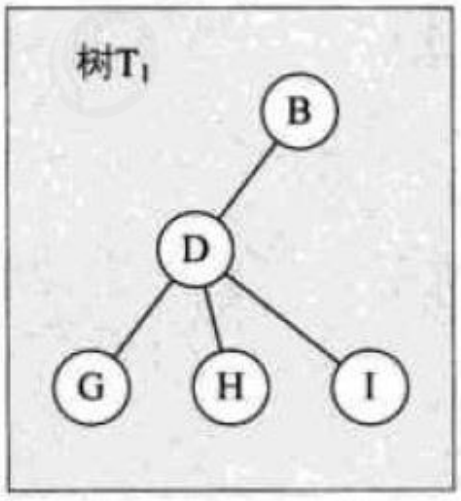
树在维基百科中的定义为：，**树**（英语：Tree）是一种无向图（undirected graph），其中任意两个顶点间存在唯一一条路径。或者说，只要没有回路的连通图就是树。在计算机科学中，**树**（英语：tree）是一种抽象数据类型（ADT）或是实现这种抽象数据类型的数据结构，用来模拟具有树状结构性质的数据集合。它是由 n（n>0）个有限节点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

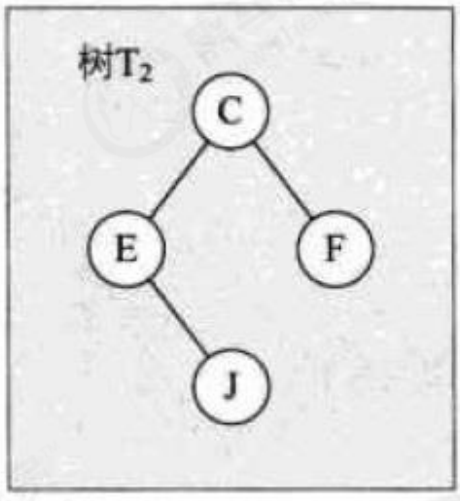
- 每个节点都只有有限个子节点或无子节点；
- 没有父节点的节点称为根节点；
- 每一个非根节点有且只有一个父节点；
- 除了根节点外，每个子节点可以分为多个不相交的子树；

- 树里面没有环路(cycle)

  这个定义不是特别好懂，那我们借助于一幅图来理解就会非常的清晰

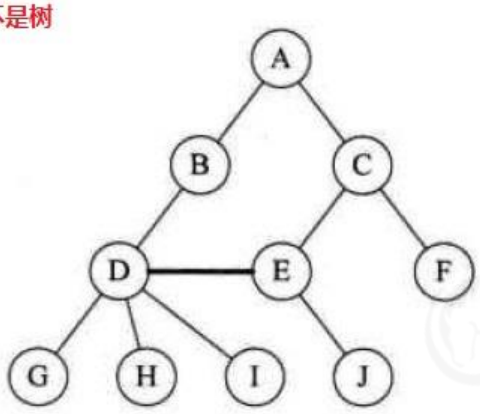




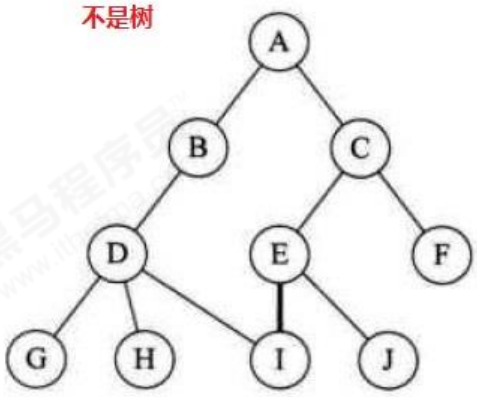


以上这些都是树，下面我们再看几个**不是树**的情况

不是树

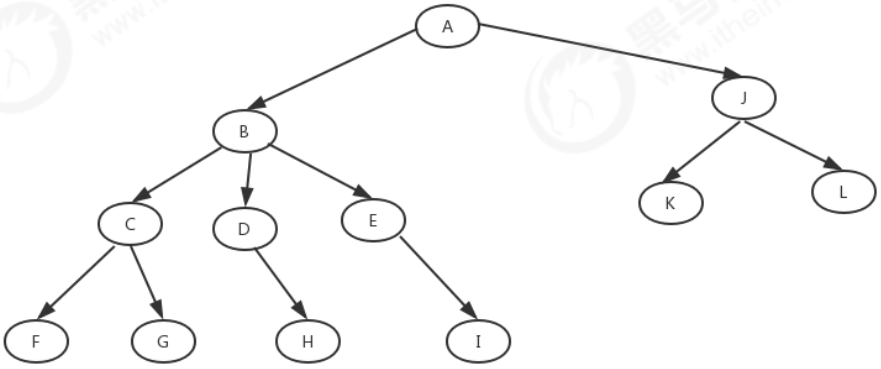


不是树



“树”这种数据结构真的很像我们现实生活中的“树”，这里面每个元素我们叫作“节点”；用来连线相邻节点之间的关系，我们叫作“父子关系”。

比如在下方这副图中：



其中：节点 B 是节点 C D E 的**父节点**，C D E 就是 B 的**子节点**，C D E 之间称为**兄弟节点**，我们把没有父节点的 A 节点叫做**根节点**，我们把没有子节点的节点称为**叶子节点**如：F G H I K L 均是叶子节点。

理解了树的定义之后我们再来理解一些关于树的概念

4.1.2、高度，深度及层

理解了树的定义之后我们来学习几个跟树相关的概念：**高度(Heigh)，深度(Depth)，层(Level)**，我们依次来看这几个概念：

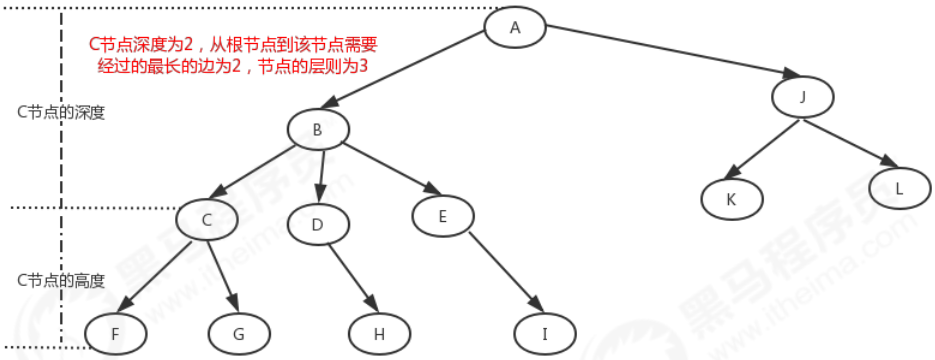
节点的高度：节点到叶子节点的最长路径(边数)，所有叶子节点的高度为 0。

节点的深度：根节点到这个节点所经历的边的个数，根的深度为 0。

节点的层数：节点的深度+1

树的高度：根节点的高度

我们用一幅图来继续说明如下：

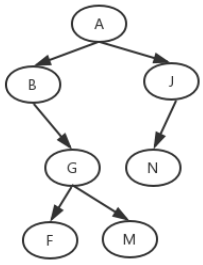


4.2.二叉树(理解并掌握)

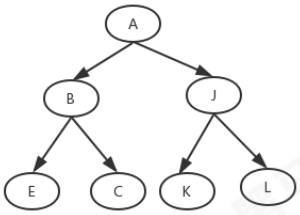
树这种数据结构形式结构是多种多样的，但是在实际企业开发中用的最多的还是二叉树，接下来我们学习二叉树

4.2.1、二叉树的定义

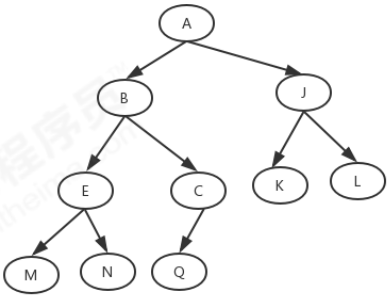
二叉树，顾名思义，每个节点最多有两个“叉”，也就是两个子节点，分别是**左子节点**和**右子节点**。不过，二叉树并不要求每个节点都有两个子节点，有的节点只有左子节点，有的节点只有右子节点，如下图所示均是二叉树



T1



T2



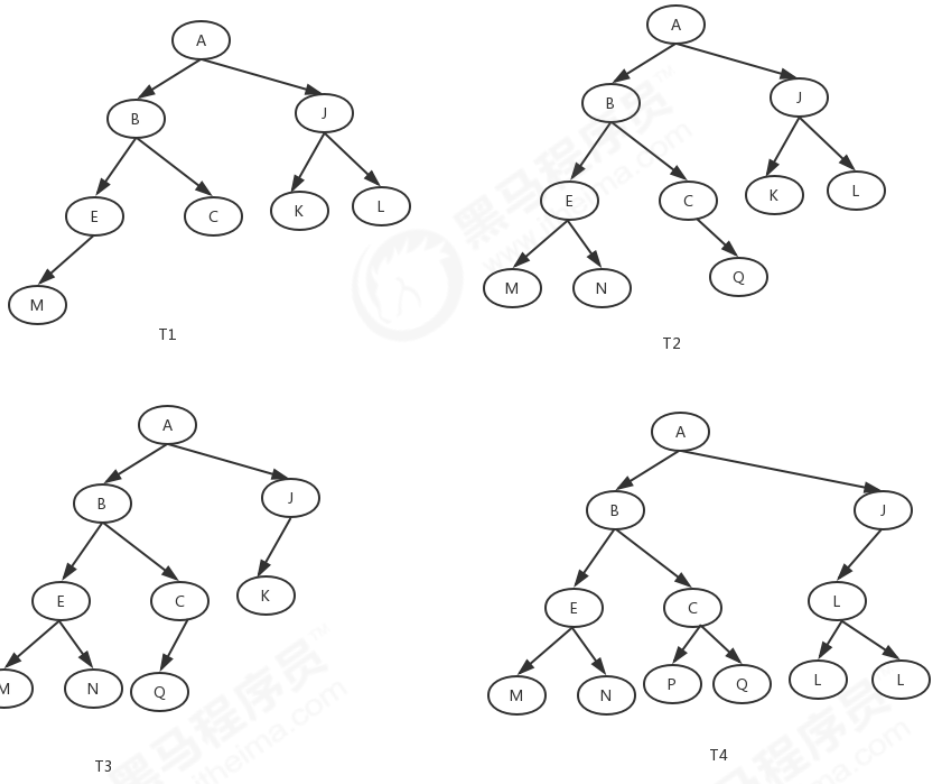
T3

当然了在这三棵树中，有两棵比较特殊的二叉树，分别是 T2 和 T3

T2：叶子节点全都在最底层，除了叶子节点之外，每个节点都有左右两个子节点，这种二叉树就叫作**满二叉树**。

T3：叶子节点都在最底下两层，最后一层的叶子节点都靠左排列，并且除了最后一层，其他层的节点个数都要达到最大，这种二叉树叫作**完全二叉树**。

满二叉树我们特别容易理解，完全二叉树我们可能就不是特别能够分清楚，下面我画几棵树，你分析一下看哪些是完全二叉树？



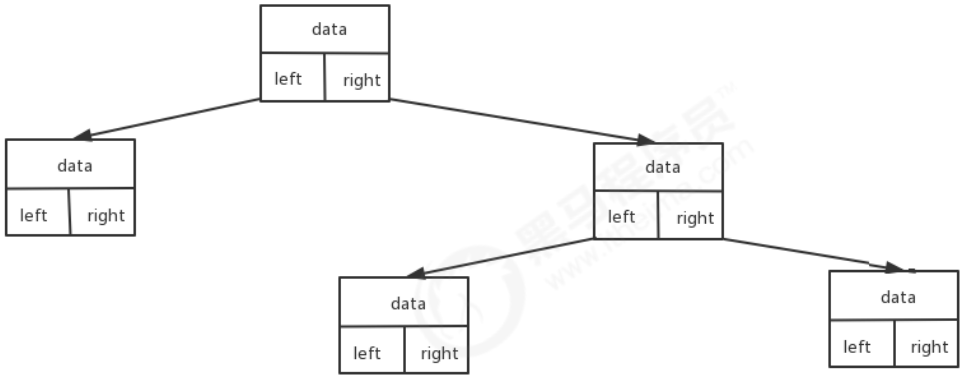
通过对比这几棵树，你分析出来哪些是完全二叉树哪些不是吗？

答案是：T1 是完全二叉树，T2,T3,T4 均不是完全二叉树，

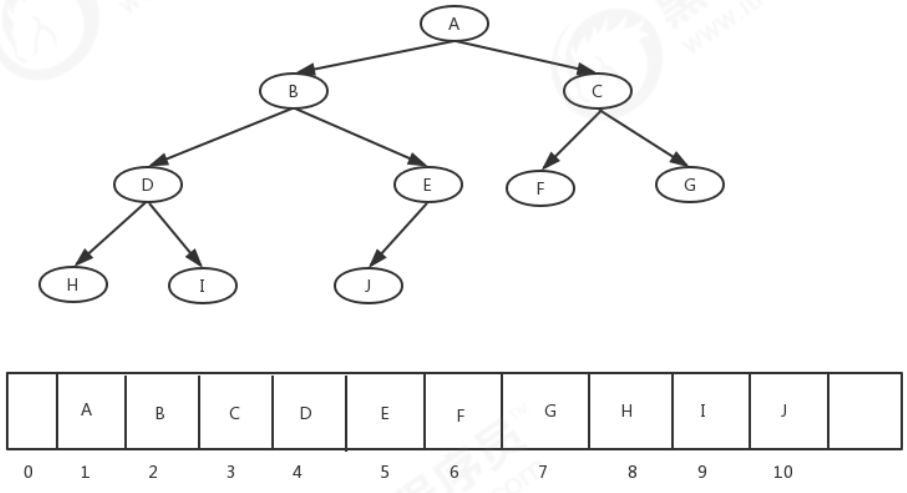
你可能会说，满二叉树的特征非常明显，但是完全二叉树的特征不怎么明显啊，单从长相上来看，完全二叉树并没有特别特殊的地方啊，那我们为什么还要特意把它拎出来讲呢？为什么偏偏把最后一层的叶子节点靠左排列的叫完全二叉树？如果靠右排列就不能叫完全二叉树了吗？这个定义的由来或者说目的在哪里？

要理解完全二叉树定义的由来，我们需要先了解，如何表示（或者存储）一棵二叉树？想要存储一棵二叉树，我们有两种方法，一种是基于指针或者引用的二叉链式存储法，一种是基于数组的顺序存储法。

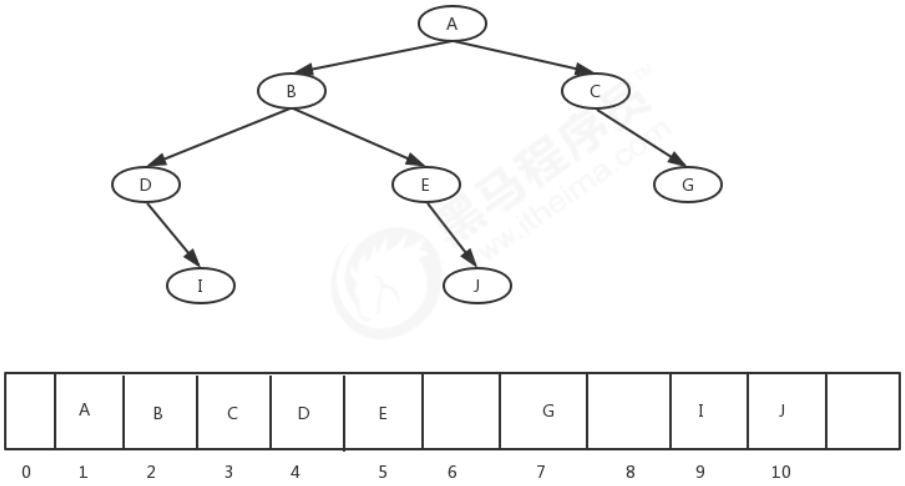
我们先来看比较简单、直观的链式存储法。从我们画的图中你应该可以很清楚地看到，每个节点有三个字
段，其中一个存储数据，另外两个是指向左右子节点的指针。我们只要找到根节点，就可以通过左右子节点的指针，把整棵树都串起来。这种存储方式我们比较常用。大部分二叉树代码都是通过这种结构来实现的



我们再来看，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，那左子节点存储在下标 2 * i = 2 的位置，右子节点存储在 2 * i + 1 = 3 的位置。以此类推，B 节点的左子节点存储在 2i = 2 * 2 = 4 的位置，右子节点存储在 2 * i + 1 = 2 * 2 + 1 = 5 的位置。如下图所示：



像我刚刚图中这棵树其实是一个完全二叉树，我们只是浪费了数组下标为 0 的位置，但如果对于如下这棵树，我们浪费的存储空间可就多了，



总结一下，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。通过这种方式，我们只要知道根节点存储的位置（一般情况下，为了方便计算子节点，根节点会存储在下标为 1 的位置），这样就可以通过下标计算，把整棵树都串起来。

所以，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。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会单独拎出来的原因，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。

4.2.2、二叉树的遍历

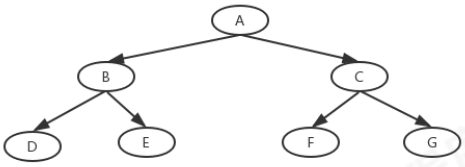
前面我们讲述了二叉树的存储结构接下来我们来学习二叉树的遍历方式，经典的三种遍历方式：**前序遍历，中序遍历，后续遍历**，我们依次来看

前序遍历：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。

终须遍历：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：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。

我们还是以图示的方式来表述这个过程：



前序遍历结果：A->B->D->E->C->F->G
中序遍历结果：D->B->E->A->F->C->G
后序遍历结果：D->E->B->F->G->C->A

前序遍历：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。
中须遍历：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：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。

实际上从遍历的过程我们可以总结出：二叉树的前序，中序，后序遍历是一个递归的过程，比如前序遍历，就是先答应根节点，然后递归的打印其左子树，然后递归的打印其右子树，那我们之前也分析过递归代码的编写，主要是要找出递推公式以及递归终止条件，那你能自行编写出二叉树前序，中序，后序遍历的代码吗？

另外遍历二叉树遍历的时间复杂度是多少呢？通过我们分析的二叉树的遍历流程我们可以发现，遍历二叉树的时间复杂度跟二叉树节点的个数 n 成正比，因此，**二叉树遍历的时间复杂度是 O(n)**。

4.2.3、二叉查找树

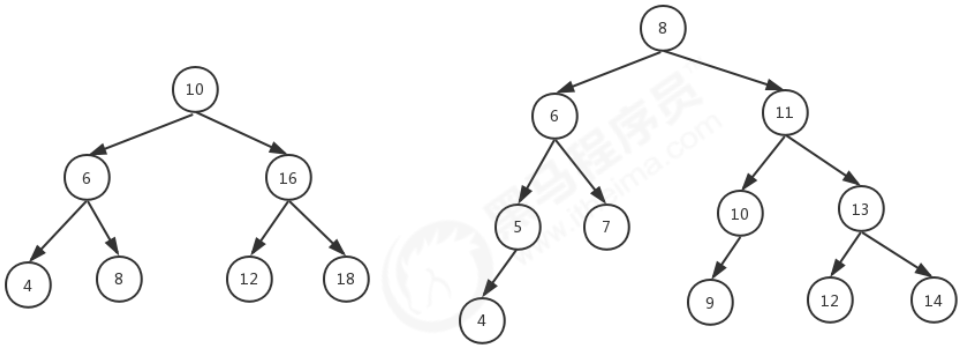
二叉查找树又名二叉搜索树，有序二叉树或者排序二叉树，是二叉树中比较常用的一种类型，我们先来看二叉查找树的结构定义

（1）、结构及特点

二叉查找树要求，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，详细可以分为以下 4 点：

1. **若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；**
2. **若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；**
3. **任意节点的左、右子树也分别为二叉查找树；**
4. **没有键值相等的节点。**

我们以一副图示表示一下该树的结构

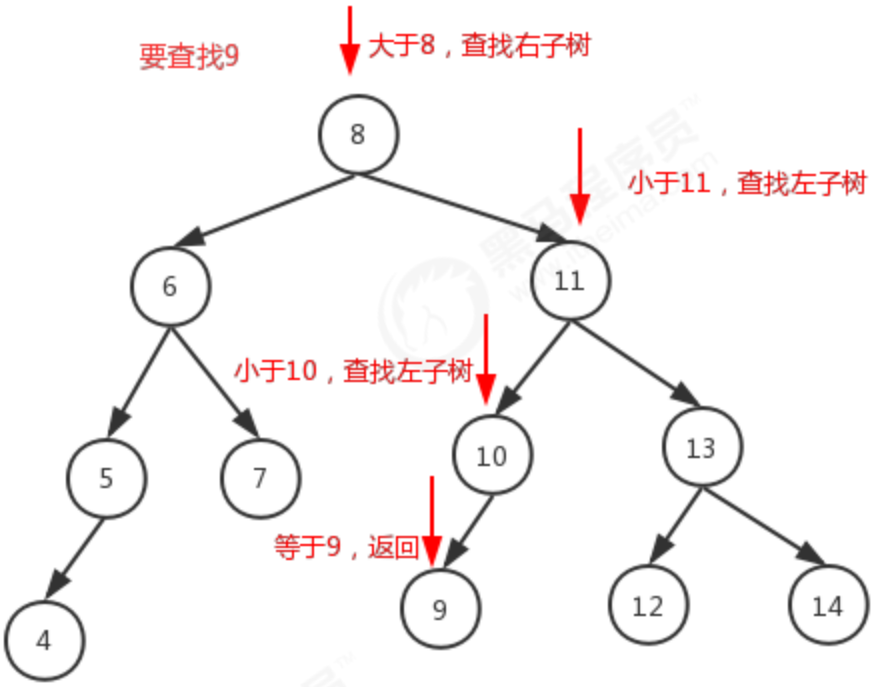


我们从二叉查找树这个名称就能够体会到该树的一个特点，能够支持快速查找，那除此之外还有什么特点呢？包括我们为什么还要来学习这种树呢？

二叉查找树支持动态数据的快速插入，删除，查找操作。这是它的特点，也正是我们要来学习它的原因，我们之前也学习过散列表这种数据结构，它也支持这几个操作，并且散列表实现这几个操作更加的高效，时间复杂度是 O(1),那既然有了如此高效的散列表为什么还要来学习二叉查找树，是不是有某些情况我们必须使用它？带着这几个问题我们依次展开二叉查找树这几个特点的相关学习。

（2）、查找操作

我们来看如何在一棵二叉查找树中查询某一个值的节点：我们从根节点开始，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。如图



代码实现如下：

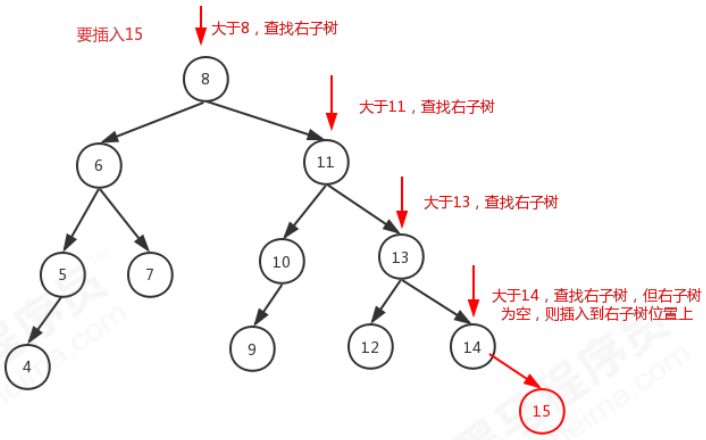
```
/**
 * 二叉搜索树
 */
public class SimpleBinarySearchTree {

    // 二叉查找树,指向根节点
    private Node tree;

    /**
     * 根据指定的值查找对应的节点
     * @param value
     * @return
     */
    public Node find(int value){
        Node parent = tree;

        while (parent !=null){

            if(parent.value > value){
                parent = parent.left;
            }else if(parent.value < value){
```

```
                parent = parent.right;
            }else {
                return parent;
            }
        }
        return parent;
    }


    private static class Node{
        private int value;
        private Node left;
        private Node right;

        protected Node(Node left,int value,Node right){
            this.left = left;
            this.value = value;
            this.right = right;
        }

        public int getValue() {
            return value;
        }

        public void setValue(int value) {
            this.value = value;
        }

        public Node getLeft() {
            return left;
        }

        public void setLeft(Node left) {
            this.left = left;
        }

        public Node getRight() {
            return right;
        }

        public void setRight(Node right) {
            this.right = right;
        }
    }
}
```

因为暂时还没写数据的插入方法，因此目前还没办法判断查询方法的有效性，接下来我们分析数据的插入。

（3）、插入操作

二叉查找树的插入操作和查询操作有点类似，也是要先从根节点开始依次比较要插入的数据和节点的数据的大小并以此来判断是将数据插入其左子树还是右子树，如果节点的右子树为空，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。



代码如下：

```
/**
     * 将 value 值存入容器
     * @param value
     * @return
     */
    public boolean put(int value){
        if(tree == null){
            tree = createNode(value);
            return true;
        }

        Node parent = tree;
        //遍历
        while (parent !=null){
            if(parent.value > value){
                if(parent.left ==null){
                    parent.left = createNode(value);
```

```
                    return true;
                }
                parent = parent.left;
            }else if(parent.value < value){
                if(parent.right ==null){
                    parent.right = createNode(value);
                    return true;
                }
                parent = parent.right;
            }
        }
        return false;
    }

    private Node createNode(Node left,int value,Node right){
        return new Node(left,value,right);
    }
      private Node createNode(int value){
        return createNode(null,value,null);
    }
```

编写代码测试如下：

```
@Test
    public void test1(){
        //创建容器
        SimpleBinarySearchTree tree = new SimpleBinarySearchTree();
        //向容器中添加值
        tree.put(2);
        tree.put(4);
        tree.put(6);
        tree.put(10);
        tree.put(15);
        tree.put(16);
        tree.put(17);
        tree.put(18);
        tree.put(5);
        tree.put(3);
        tree.put(9);
        tree.put(11);
        tree.put(12);
        // 从容器中取出节点值为12 的节点
        SimpleBinarySearchTree.Node node = tree.find(12);
        System.out.println("节点值为 12 的节点为:"+node);

    }
```

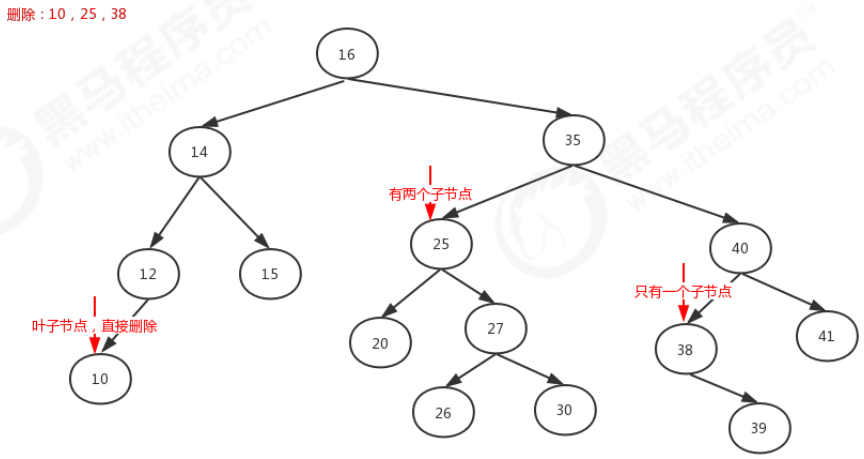
（4）、删除操作

二叉查找树的插入和查询相对来说比较简单易懂，但是删除操作相对复杂，总结下来有三种情况：

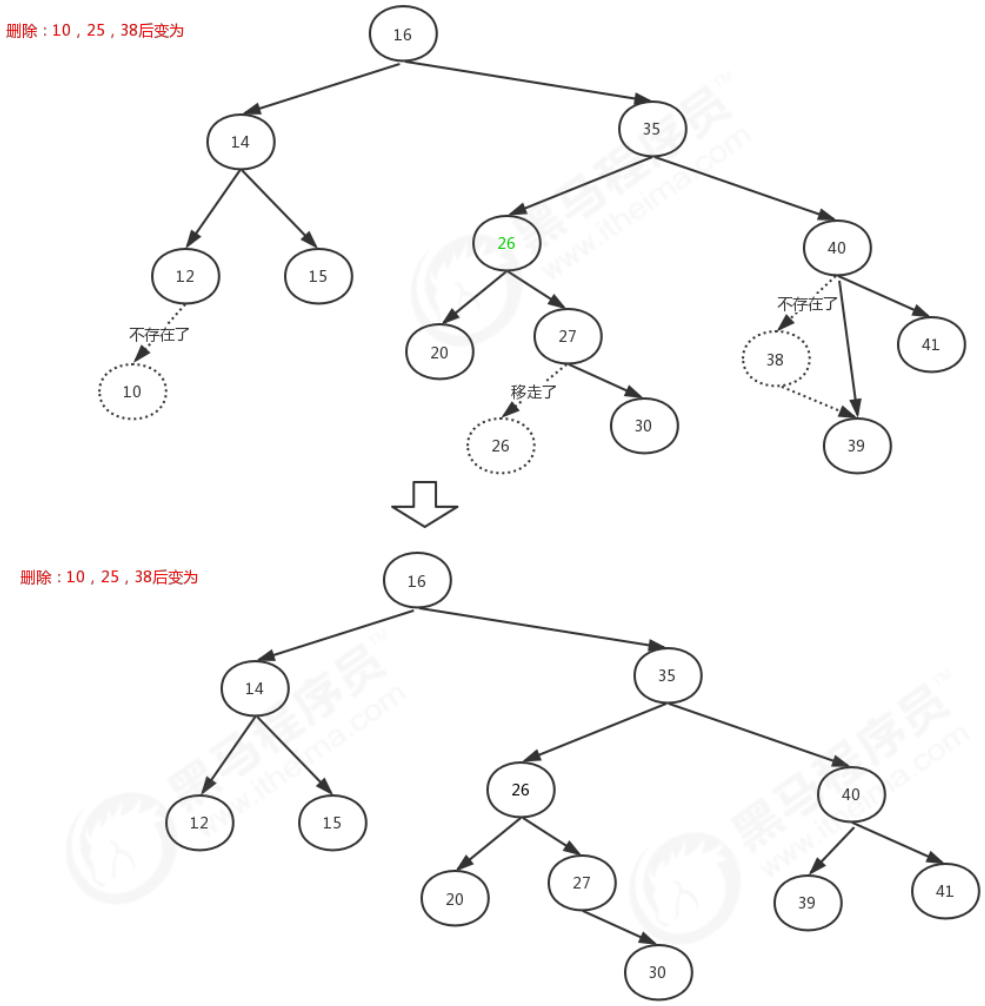
1：要删除的节点是叶子节点即没有子节点，我们只需将父节点中指向该节点的指针置为 null 即可，这是最简单的一种形式。比如删除图中的节点 10

2：要删除的节点只有一个子节点(只有左子节点或者只有右子节点)，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。比如删除图中的节点 38

3：要删除的节点有两个子节点，这是最复杂的一种情况，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。比如删除图中的节点 25



按照规则删除之后的树结构为：



代码实现如下：

```
/**
```

*1：要*删除的节点是叶子节点即没有子节点，我们只需将父节点中指向该节点的指针置为 null 即可，这是最简单的一种形式。比如删除图中的节点 *10*

*2：要*删除的节点只有一个子节点*(*只有左子节点或者只有右子节点*)*，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。比如删除图中的节点 *38*

*3：要*删除的节点有两个子节点，这是最复杂的一种情况，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）

```
     * @param value
     */
    public void remove(int value){
        // 记录要删除的节点
        Node p = tree;
        // 记录要删除节点的父节点
        Node p_parent = null;
        //先找到要删除的元素及其父元素
        while (p != null ) {
            if(p.value > value){
                p_parent = p;
                p = p.left;
            }else if( p.value < value){
                p_parent = p;
                p = p.right;
            }else {
                break;
            }
        }
        //如果没有找到则返回
        if(p == null){
            return;
        }

        // 要删除的节点有两个子节点 这种情况要用右子树中最小节点的值替换当前要删除元素的值，然后删除右侧最小节点
        if(p.left!=null && p.right !=null){
            // 找到该节点右子树的最小节点----->最左侧的叶子节点
            Node rigthTree = p.right;
            Node rightTree_p = p; // rigthTree 的父节点
            while(rigthTree.left !=null){
                rightTree_p = rigthTree;
                rigthTree = rigthTree.left;
            }
            // 用右子树中最小的节点替换当前要删除的节点
            p.value = rigthTree.value;
            // 删除右子树中最小的节点，考虑到删除操作的其他两种情况：要删除元素是叶子节点以及要删除元素只有一个子节点都属于元素的删除 这里
            // 思路和逻辑都是一样的，为统一代码逻辑编写在此处不直接删除
            p = rigthTree;
            p_parent = rightTree_p;
        }

        // 删除节点是叶子节点或者仅有一个子节点,都是要删除该节点，将父节点的指针指向当前节点的子节点
        Node child = null;
```

```
        //计算当前节点的子节点
        if(p.right != null ){ // 当前元素的右子节点不为空
            child = p.right;
        }else if( p.left != null ){ // 当前元素的左子节点不为空
            child =  p.left;
        }else {
            child = null;
        }
        //执行删除
        if(p_parent == null ){ // 要删除根节点
            tree = child;
        }else if(p_parent.left == p) { //更新父节点的左指针
            p_parent.left = child;
        }else {
            p_parent.right = child;
        }

    }
```

（5）、查找最大值/最小值

对于查找二叉查找树的最小值，我们只需要从根节点开始依次查找其左子节点直到最后的叶子节点，最后的叶子节点就是其最小值，同理查找最大值只需要从根节点开始依次查找其右子节点直到最后的叶子节点即为最大值。

代码实现如下“

```
/**
     * 获取最小节点
     * @return
     */
    public Node getMin(){
        if(tree ==null){
            return null;
        }
        Node p = tree;
        while (p.left != null){
            p = p.left;
        }
        return p;
    }

    /**
     * 获取最大节点
     * @return
     */
    public Node getMax(){
        if(tree ==null){
            return null;
```

```
        }
        Node p = tree;
        while (p.right != null){
            p = p.right;
        }
        return p;
    }
```

当然了对于二叉查找树我们还可以找到节点的前驱节点和后继节点，对于这两个概念我们解释说明如下：

后继节点：该节点右子树中最小的节点

前驱节点：该节点左子树中最大的节点

理解了这两个概念之后，留一个课后思考题：如何通过编程的方式获取某一个节点的前驱和后继节点呢？

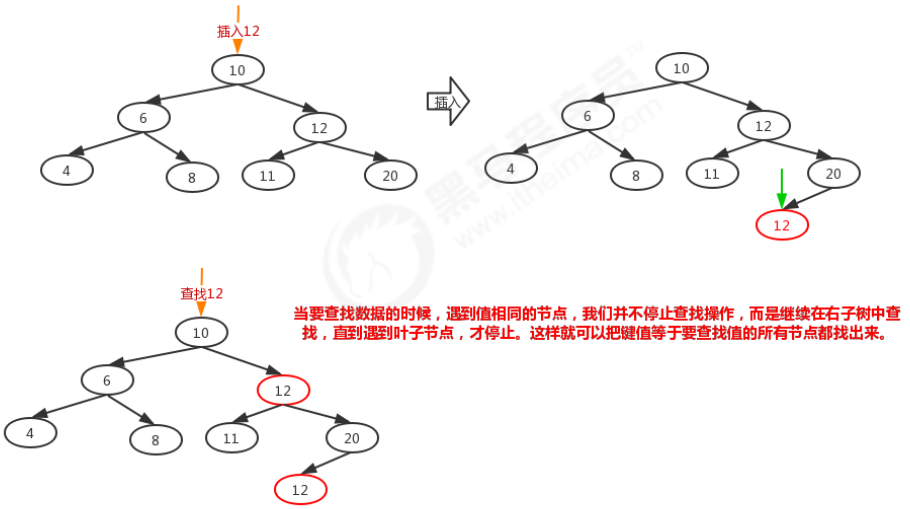
（6）、其他操作

二叉查找树除了上述的相关操作外，还有一个重要的特性，就是**如果中序遍历二叉查找树，能够得到一个有序的数据序列，时间复杂度是 O(n)，非常的高效，因此二叉查找树也叫二叉排序树**，

我们前面讲解的时候存储的都是 int 类型的数字，但是在实际的软件开发中我们一般都是存储的包含很多属性的对象，判断的时候利用对象中的某一个属性进行判断，对象中的其他属性我们称为卫星数据，对于有重复数据的二叉查找树，我们应该如何存储和查找呢？有两种方案：

1：二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。

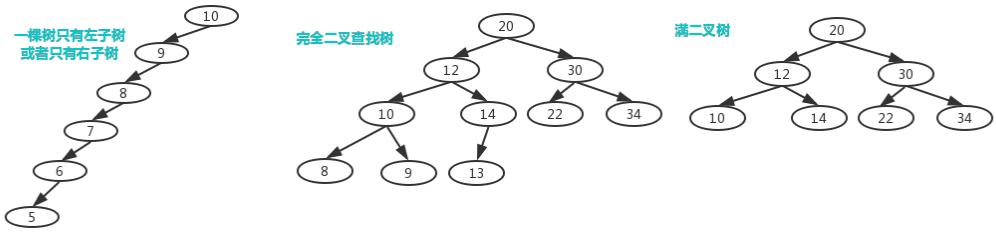
2：每个节点仍然只存储一个数据。在查找插入位置的过程中，如果碰到一个节点的值，与要插入数据的值相同，我们就将这个要插入的数据放到这个节点的右子树，也就是说，把这个新插入的数据当作大于这个节点的值来处理，当要查找数据的时候，遇到值相同的节点，我们并不停止查找操作，而是继续在右子树中查找，直到遇到叶子节点，才停止。这样就可以把键值等于要查找值的所有节点都找出来。对于删除操作，我们也需要先查找到每个要删除的节点，然后再按前面讲的删除操作的方法，依次删除。



（7）、时间复杂度分析

在这一节中我们分析一下二叉查找树的查找，插入，删除的相关操作的时间复杂度。

实际上由于二叉查找树的形态各异，时间复杂度也不尽相同，我画了几棵树我们来看一下插入，查找，删除的时间复杂度



对于图中第一种情况属于最坏的情况，二叉查找树已经退化成了链表，左右子树极度不平衡，此时查找的时间复杂度肯定是 O(n)。

对于图中第二种或者第三种情况是属于一个比较理想的情况，我们代码的实现逻辑以及图中所示表明**插入，查找，删除的时间复杂度其实和树的高度成正比，那也就是说时间复杂度为 O(height)**

那如何求一棵完全二叉树的高度？即求一棵包含 n 个节点的完全二叉树的高度？

对于一棵满二叉树而言：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。从上图中可以看出，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)。

但是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^(k-1)个之间（我们假设最大层数是 k）。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：

1+2+4+8+...+2^(k-2) +1 =<n <= 1+2+4+8+...+2^(k-2) +2^(k-1)

这是一个等比数列，根据等比数列求和公式 S= a1(1-q^(n-1))/ 1-q，其中 q 是公比，n 为数据个数。

所以：我们利用求和公式对上述式子进行计算后得知，k 的一个最大值是：$log_2 n + 1$

也就是说完全二叉树的高度小于等于：$log_2 n$

通过我们的分析我们发现一棵极度不平衡的二叉查找树，它的查找性能和单链表一样。我们需要构建一种不管怎么删除、插入数据，在任何时候，都能保持任意节点左右子树都比较平衡的二叉查找树，这种特殊的二叉查找树也可以叫做**平衡二叉查找树。平衡二叉查找树的高度接近 logn，所以插入、删除、查找操作的时间复杂度也比较稳定，是 O(logn)。**

（8）、和散列表的对比

之前我们学习过散列表的插入、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？个人认为原因如下：

1：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。

2：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也不稳定，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)。

3：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。

4：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。

5：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。

4.2.4、平衡二叉树(AVL)

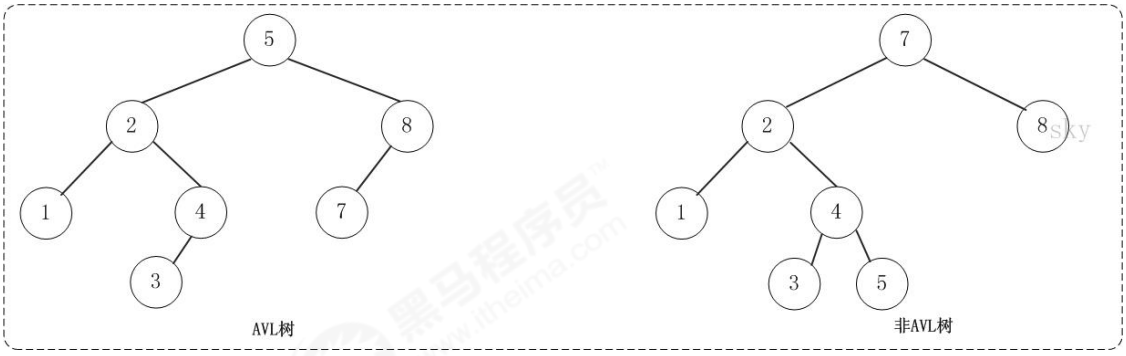
上一节我们讲到，二叉查找树只有在比较平衡的情况下，插入、删除、查找操作时间复杂度才是 O(logn)，不过在二叉查找树频繁的动态更新过程中，会逐渐退化直至最坏的情况变为链表，时间复杂度退化为 O(n)，所以我们要解决这种复杂度退化的问题就要找到一种平衡二叉树，平衡二叉查找树中“平衡”的意思，其实就是让整棵树左右看起来比较“对称”、比较“平衡”，不要出现左子树很高、右子树很矮的情况。这样就能让整棵树的高度相对来说低一些，相应的插入、删除、查找等操作的效率高一些。

发明平衡二叉查找树这类数据结构的初衷是，解决普通二叉查找树在频繁的插入、删除等动态更新的情况下，出现时间复杂度退化的问题。

（1）、定义

平衡二叉查找树：简称平衡二叉树。由前苏联的数学家 **A**delse-**V**elskil 和 **L**andis 在 1962 年提出的高度平衡的二叉树，根据科学家的英文名也称为 AVL 树。它具有如下几个性质：

1. 可以是空树。
2. 假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过 1。



上图中的两棵树，左边的是 AVL 树，它的任何节点的两个子树的高度差别都<=1；而右边的不是 AVL 树，因为 7 的两颗子树的高度相差为 2(以 2 为根节点的树的高度是 2，而以 8 为根节点的树的高度是 0)。

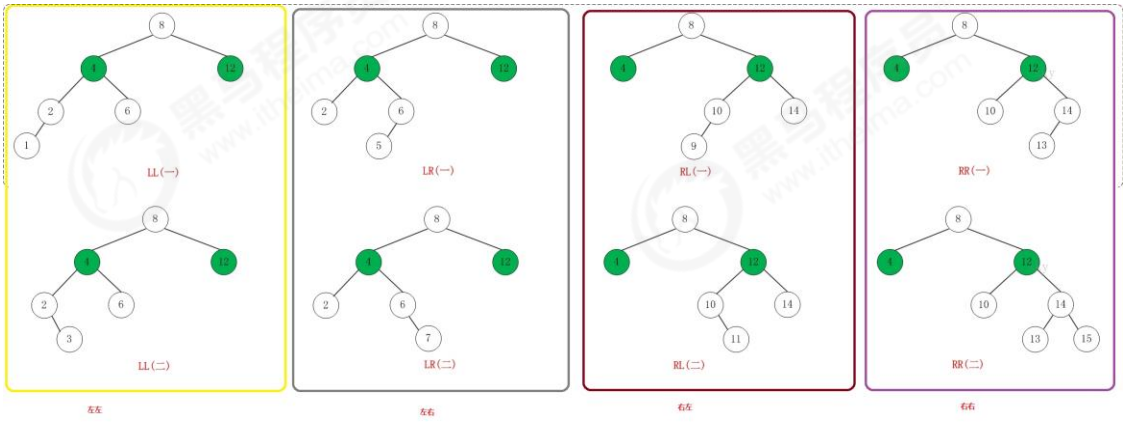
对于给定结点数为 n 的 AVL 树，最大高度为 O(log2n).也就说，从 n 个数中，查找一个特定值时，最多需要 log2n 次。因此，AVL 是一种特别适合进行查找操作的树。

（2）、失衡的四种情况及调整方法

在平衡二叉树中，当我们插入新的元素时，为了保证二叉搜索树的特性，很容易导致某些结点**失衡**，即该结点的平衡因子大于 1。

而在二叉树中，任意结点孩子最多只有左右两个，而且导致失去平衡的必要条件就是当前结点的两颗子树的高度差等于 2。因此，致使一个结点失衡的插入操作有以下 4 中。

- 在结点的左子树的左子树插入元素,LL 插入；
- 在结点的左子树的右子树插入元素,LR 插入；
- 在结点的右子树的左子树插入元素,RL 插入；
- 在结点的右子树的右子树插入元素,RR 插入。



  - LL(一）中结点 1 的插入导致结点 8 失衡，而插入的位置是在其左子树的左子树上，同样 LL(二）中，结点 3 插入同样导致结点 8 失衡，**这里需要注意子树是从受影响的结点算起，虽然 3 插在了右边，但他依旧是在 8（失衡结点）左子树的左子树上，因此属于 LL 插入。**
  - LR(一）,结点 5 插入导致结点 8 失衡，插入位置是在其左子树的右子树上，同样 LR(二)结点 7 的插入也是同理，因此这二者都属于 LR 插入。

后面两种失衡现象可以当做是前两者的镜像，原理都是一样的。

面对以上 4 种失衡的情况，在 AVL 树中将采用 LL(左左)，LR(左右)，RR(右右)和 RL(右左) 四种旋转方式进行调整。

为了对这四种情况进行说明，我们先来定义 AVL 树的结构：

AVL 树首先是二叉查找树，因此它的结点也必须是可比较。同时为了方便，加入一个表示当前结点高度的 height 字段，同时定义了返回节点高度和树的高度的方法，也定了一个返回两个高度中最大高度的方法。

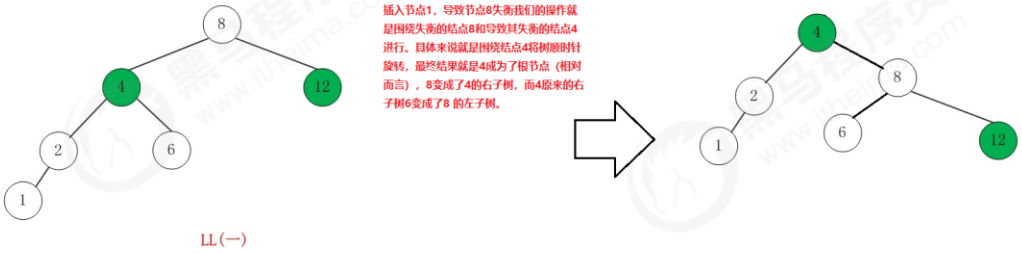
```
/**
 * AVL 树
 */
public class AvlTree<T extends Comparable>{

    // Avl 树 根节点
    private AvlNode tree;


    /**
     * 获取某一节点的高度
     * @param node
     * @return
     */
     private int height(AvlNode node) {
         return node == null ? 0:node.height;
    }
    /**
     * 获取 avl 树的高度
     * @return
     */
    public int height() {
        return height(tree);
    }

    /**
     * 返回两个高度中的最大值
     * @param height1
     * @param height2
     * @return
     */
    private int getMax(int height1,int height2){
        return height1 > height2 ? height1:height2;
    }
    /**
   * toString 方法我们使用中序遍历的方式打印树
   * @return
   */
  @Override
  public String toString() {
      System.out.println("前序遍历的结果:");
      preOrder(tree);
```

```
        System.out.println("\n");
        System.out.println("中序遍历的结果:");
        inOrder(tree);
        System.out.println("\n");
        System.out.println("后续遍历的结果:");
        postOrder(tree);
        return "";
    }

    /**
     * 中序遍历
     *   先打印左子树
     *   然后打印自身
     *   最后打印右子树
     * @param node
     */
    public void inOrder(AvlNode node){
        if(node == null ){
            return;
        }
        inOrder(node.left);
        System.out.print(node.getData()+"->");
        inOrder(node.right);
    }

    /**
     * 前序遍历
     *   先打印自身
     *   然后打印左子树
     *   最后打印右子树
     * @param node
     */
    public void preOrder(AvlNode node){
        if(node == null){
            return;
        }
        System.out.print(node.getData()+"->");
        preOrder(node.left);
        preOrder(node.right);
    }

    /**
     * 后续遍历
     *    先打印左子树
     *    然后打印右子树
     *    最后打印自身
     * @param node
     */
```

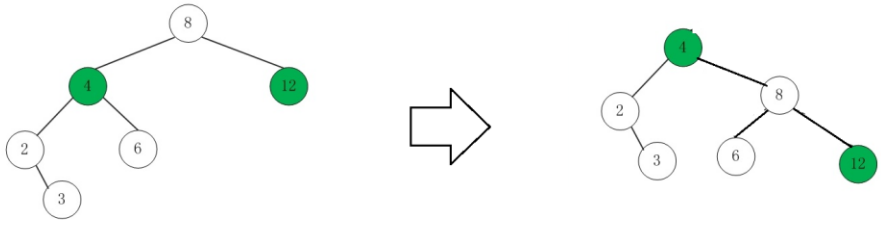
```
    public void postOrder(AvlNode node){
        if(node == null){
            return;
        }
        postOrder(node.left);
        postOrder(node.right);
        System.out.print(node.getData()+"->");
    }
    /**
     * AVL 树的节点
     */
    public static class AvlNode<T extends Comparable>{
        // 节点中存储的数据
        private T data;
        // 左子树节点
        private AvlNode<T> left;
        // 右子树节点
        private AvlNode<T> right;
        // 节点的高度
        private int height;

        protected AvlNode(T data,AvlNode left,AvlNode right,int heigh
t){
            this.data = data;
            this.left = left;
            this.right = right;
            this.height = height;
        }
        protected AvlNode(T data,AvlNode left,AvlNode right){
            this(data,left,right,0);
        }

        protected AvlNode(T data){
            this(data,null,null);
        }

        public T getData() {
            return data;
        }

        public void setData(T data) {
            this.data = data;
        }

        public AvlNode getLeft() {
            return left;
        }

        public void setLeft(AvlNode left) {
```

```
            this.left = left;
        }

        public AvlNode getRight() {
            return right;
        }

        public void setRight(AvlNode right) {
            this.right = right;
        }

        public int getHeight() {
            return height;
        }

        public void setHeight(int height) {
            this.height = height;
        }
    }
}
```

①、LL-左左旋转



LL(一)



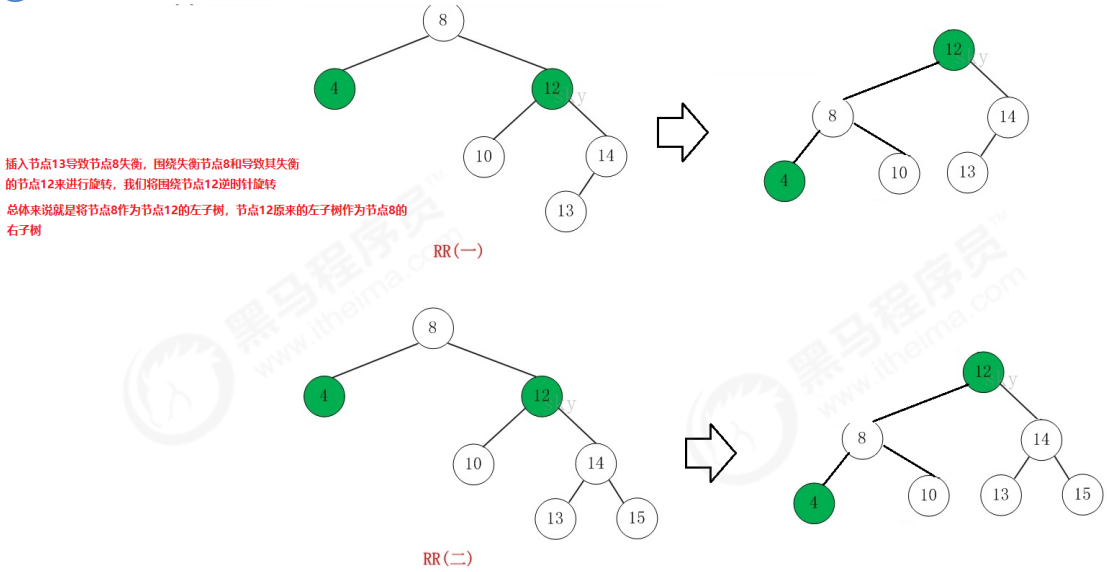
LL(二)

代码实现如下：

```
/**
     * LL 旋转
     * @param node    失衡节点
     * @return        左旋后的根节点
     */
```

```
    public AvlNode leftRotate(AvlNode node){
        // 定义临时变量保存 失衡节点的左子树 该节点也是左旋后的根节点
        AvlNode node_left = node.left;
        // 将失衡节点左子树的右子树作为失衡节点的左子树
        node.left = node_left.right;
        // 将失衡节点作为旋转后根节点的右子树
        node_left.right = node;
        // 重新计算失衡节点和旋转后根节点的高度
        node.height = getMax(height(node.left),height(node.right)) + 1;
        node_left.height = getMax(height(node_left.left),height(node_le
ft.right)) +1;
        return node_left;
    }
```

②、RR-右右旋转

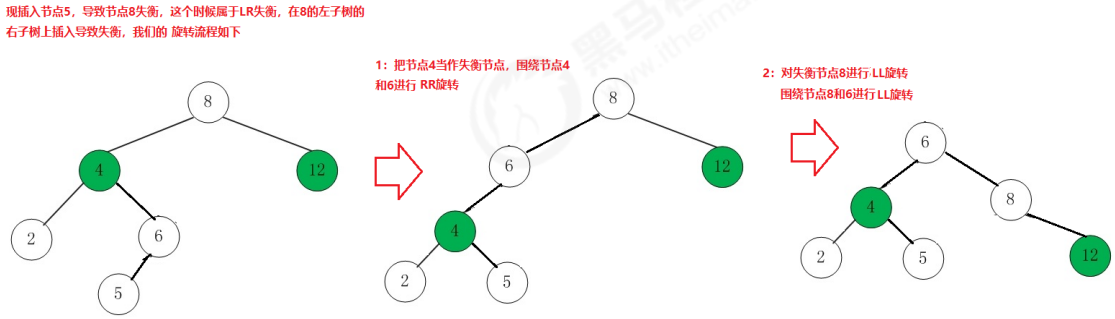


代码实现如下：

```
/**
     * RR 旋转
     * @param node   失衡节点
     * @return       右旋后的根节点
     */
    public AvlNode rightRotate(AvlNode node){
        // 定义临时变量保存右旋后的根节点 也就是失衡节点的右子树
        AvlNode new_root = node.right;
        // 将新的根节点的左子树当作失衡节点的右子树
        node.right = new_root.left;
        // 将失衡节点当作新的根节点的左子树
        new_root.left = node;
        // 重新计算失衡节点和新的根节点的高度
```

```
        node.height = getMax(height(node.left),height(node.right)) + 1;
        new_root.height = getMax(height(new_root.left),height(new_root.
right)) +1;
        return new_root;
    }
```

③、LR-左右旋转

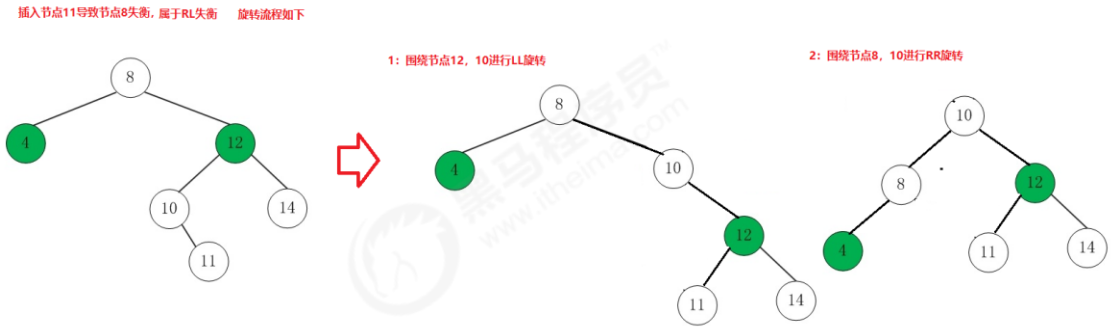


代码如下：

```
/**
     * LR 旋转
     * @param node   失衡节点
     * @return       旋转后根节点
     */
    public AvlNode leftRightRotate(AvlNode node){
        // 通过分析我们知道 LR 旋转可以拆分为一次 RR 旋转和一次 LL 旋转来完成
        node.left = rightRotate(node.left);
        return leftRotate(node);
    }
```

④、RL-右左旋转

有了前面分析 LR 旋转的过程，RL 的旋转过程跟其很类似就是方向不一样而已。



代码实现如下：

```
/**
     * RL 旋转
     * @param node 失衡节点
     * @return      旋转后根节点
     */
    public AvlNode rightLeftRotate(AvlNode node){
        // RL 旋转可以拆分成一次 LL 旋转和一次 RR 旋转来完成
        node.right = leftRotate(node.right);
        return rightRotate(node);
    }
```

（3）、插入操作

上一节中我们分析了平衡二叉树的几种失衡情况以及对应的调整策略，下面我们来分析一下如何向平衡二叉树中插入数据，

代码实现如下：

```
/**
     * 插入数据
     * @param value
     */
    public void insert(T value){
        this.tree = insert(tree,value);
    }
    private AvlNode insert(AvlNode node,T data){
        // 将 data 添加到 node 节点的子节点上
        if(node == null){
            node = new AvlNode<T>(data);
        }else {
            int compared = data.compareTo(node.getData());
            if(compared > 0){
                // 要添加的值大于当前节点的值,将 data 存储到当前节点的右子树
上 递归的插入
                node.right = insert(node.right,data);
                // 如插入后 avl 树变得不平衡,则应该重新调节该树的结构---旋转
                if(height(node.right) - height(node.left) == 2){
                    // 判断是 RR 还是 RL
                    if(data.compareTo(node.right.getData()) > 0){ // 证
明是 RR
                        node = rightRotate(node);
                    }else { // 证明是 RL
                        node = rightLeftRotate(node);
                    }
                }

            }else if(compared < 0){
```

```
                // 要添加的值小于当前节点的值,将 data 存储到当前节点的左子树
上 递归的插入
                node.left = insert(node.left,data);
                // 如插入后 avl 树变得不平衡,则应该重新调节该树的结构---旋转
                if(height(node.left) - height(node.right) ==2){
                    //判断是 LL 还是 LR
                    if(data.compareTo(node.left.getData()) > 0){ //证明
是 LR
                        node = leftRightRotate(node);
                    }else { // 证明是 LL
                        node = leftRotate(node);
                    }
                }
            }else {
                //要添加的值和该节点的值相同，不做处理
            }
        }
        // 计算节点 node 的高度
        node.height = getMax(height(node.left),height(node.right)) + 1;
        return node;
    }
```

编写测试代码测试如下：

```
/**
 * AVL 树的测试
 */
public class AvlTreeTest {

    @Test
    public void test1(){
        AvlTree tree = new AvlTree();
        // 添加节点
        tree.insert(10);
        tree.insert(8);
        tree.insert(3);
        tree.insert(12);
        tree.insert(9);
        tree.insert(4);
        tree.insert(5);
        tree.insert(7);
        tree.insert(1);
        tree.insert(11);
        tree.insert(17);
        // 打印结果
        System.out.println(tree);
    }
}
```

在这里我们分析了 AVL 树的插入操作，那对于 AVL 树的查询操作和删除操作应该如何实现呢？你可以试着去分析和实现一下。

4.3.小结

本章内容我们学习了树这种数据结构，最主要是学习了二叉树，掌握了二叉树的定义，二叉树的遍历方式，介绍了二叉查找树，对二叉查找树的查找，插入，删除等相关操作做了实现，接着又学习了 AVL 树，实现了 AVL 树的部分功能，最后分析了一下红黑树及其平衡的过程。对于我们日常的软件编程而言，我们知道了树对于动态数据的添加，删除和查询操作是非常友好的，基本上其时间复杂度为 O(n)，在某些情况下甚至于要好于 O(1)的散列表。

5：今日内容总结与作业安排

内容总结

在今天的课程中我们主要学习了两大块内容：

散列表

散列函数的特点及要求：

1：散列函数计算得到的散列值必须是大于等于 0 的正整数，因为 hash 值需要作为数组的下标。

2：如果 key1==key2，那么经过 hash 后得到的哈希值也必相同即：hash(key1) == hash(key2)

3：如果 key1 != key2，那么经过 hash 后得到的哈希值也必不相同即：hash(key1) != hash(key2)

散列函数设计的方法：

1：直接寻址法

2：除留余数发

3：数字分析法

4：平方取中法

5：折叠法

散列冲突的解决方案：

1：开放寻址-线性探测

2：开放寻址-二次检测

3：开放寻址-双重散列

4：链表法

散列表的应用：

1：HashMap 的源码解析

2：HashTable 的源码解析

哈希算法的应用场景：

1：安全加密

2：唯一 ID

3：数据校验

4：散列函数

5：负载均衡

6：数据分片

7：分布式存储

树

二叉树的定义

二叉树的遍历：前序，中序，后序遍历

二叉查找树

AVL 树

今日作业

1：熟读 HashMap 的底层源码

2：熟读 HashTable 的底层源码

3：实现二叉查找树的插入，删除，查询，最大值/最小值，前驱/后继的获取等相关操作

4：实现 AVL 树，实现插入操作，查询操作，分析一下删除操作如何实现