# 数据结构

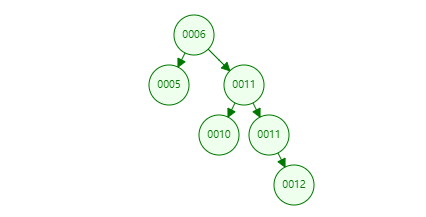
## 数据结构可视化网址

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

**树结构都是左节点小于父节点，右节点大于等于父节点**

示例：

11父节点下方10在左侧11在右侧



<https://www.cnblogs.com/tiancai/p/9024351.html> //参考地址

## 一、二叉树

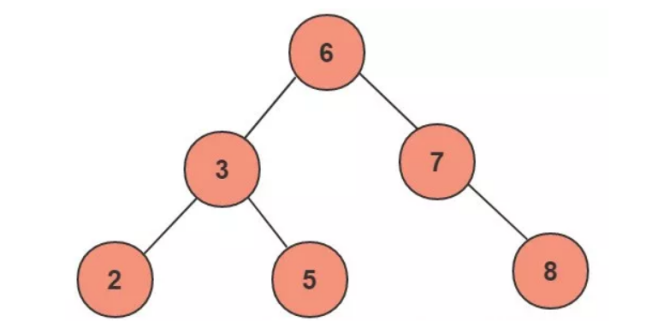
### （1）二叉树简介

二叉查找树也称为**有序二叉查找树，满足二叉查找树的一般性质，是指一棵空树具有如下性质：**

1、任意节点左子树不为空,则左子树的值均小于根节点的值；

2、任意节点右子树不为空,则右子树的值均大于于根节点的值；

3、任意节点的左右子树也分别是二叉查找树；

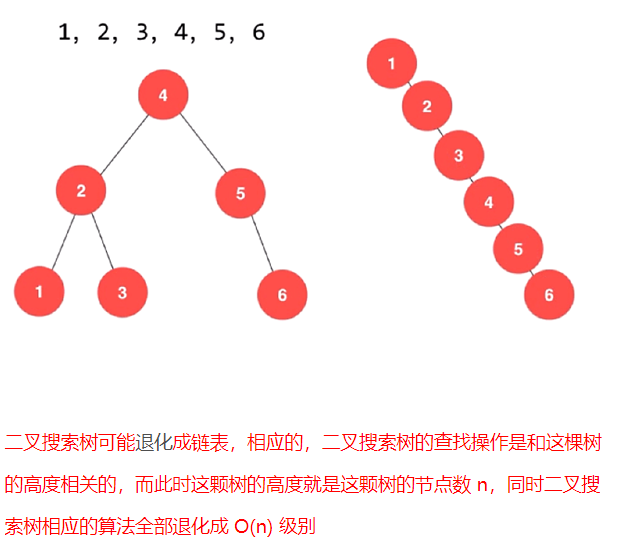
4、没有键值相等的节点；  
 

上图为一个普通的二叉查找树，按照中序遍历的方式可以从小到大的顺序排序输出：2、3、5、6、7、8。

对上述二叉树进行查找，如查键值为5的记录，先找到根，其键值是6，6大于5，因此查找6的左子树，找到3；而5大于3，再找其右子树；一共找了3次。如果按2、3、5、6、7、8的顺序来找同样需求3次。用同样的方法在查找键值为8的这个记录，这次用了3次查找，而顺序查找需要6次。计算平均查找次数得：顺序查找的平均查找次数为（1+2+3+4+5+6）/ 6 = 3.3次，二叉查找树的平均查找次数为（3+3+3+2+2+1）/6=2.3次。二叉查找树的平均查找速度比顺序查找来得更快。

### 局限性

一个二叉查找树是由n个节点随机构成，所以，对于某些情况，二叉查找树会退化成一个有n个节点的线性链。如下



如上图，如果我们的根节点选择是最小或者最大的数，那么二叉查找树就完全退化成了线性结构。显然这个二叉树的查询效率就很低，因此若想最大性能的构造一个二叉查找树，需要这个二叉树是平衡的（这里的平衡从一个显著的特点可以看出这一棵树的高度比上一个输的高度要大，在相同节点的情况下也就是不平衡），从而引出了一个新的定义-平衡二叉树AVL。

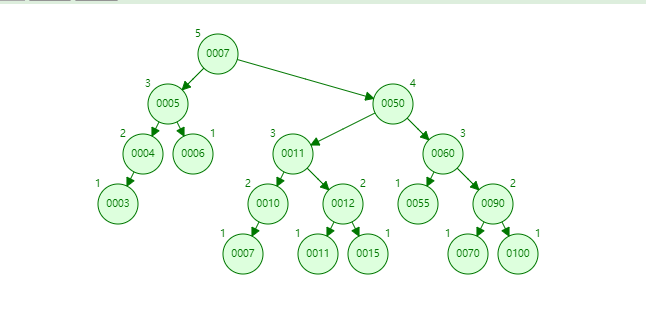
### （3）应用

海量数据并发查询，二叉树复杂度是O(K+LgN)。二叉排序树就既有链表的好处，也有数组的好处， 在处理大批量的动态的数据是比较有用。

## 二、AVL树（平衡二叉树AVL）

### 平衡二叉树简介

AVL树是带有平衡条件的二叉查找树，一般是用平衡因子差值判断是否平衡并通过旋转来实现平衡，左右子树树高不超过1，和红黑树相比，它是严格的平衡二叉树，平衡条件必须满足（**所有节点的左右子树高度差不超过1）。不管我们是执行插入还是删除操作，只要不满足上面的条件，就要通过旋转来保持平衡，而旋转是非常耗时的，由此我们可以知道AVL树适合用于插入删除次数比较少，但查找多的情况。**

****

**从上面是一个普通的平衡二叉树，这张图我们可以看出，任意节点的左右子树的平衡因子差值都不会大于1。**

### 局限性

由于维护这种高度平衡所付出的代价比从中获得的效率收益还大，故而实际的应用不多，更多的地方是用追求局部而不是非常严格整体平衡的红黑树。当然，如果应用场景中对插入删除不频繁，只是对查找要求较高，那么AVL还是较优于红黑树。

### （3）应用

1、Windows NT内核中广泛存在；

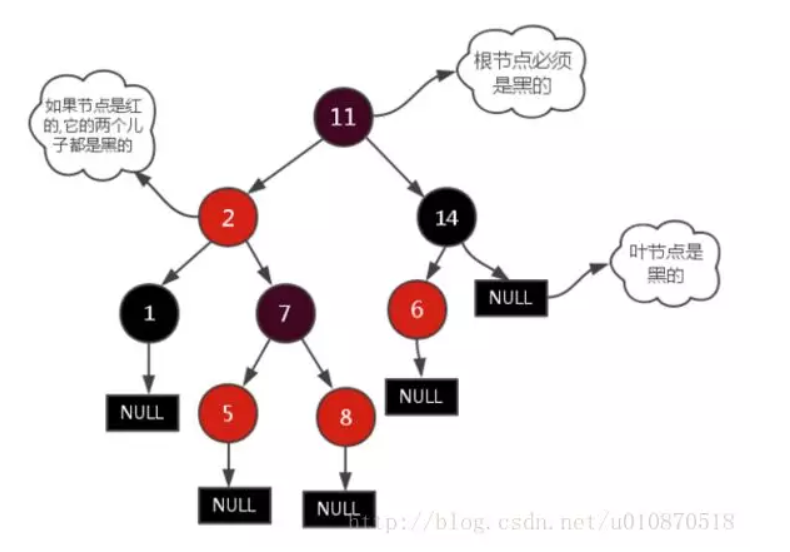
## 红黑树

### （1）简介

一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是red或black。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍。它是一种弱平衡二叉树(由于是若平衡，可以推出，相同的节点情况下，AVL树的高度低于红黑树)，相对于要求严格的AVL树来说，它的旋转次数变少，所以对于搜索、插入、删除操作多的情况下，我们就用红黑树。

### （2）性质

1、每个节点非红即黑；   
2、根节点是黑的；   
3、每个叶节点(叶节点即树尾端NULL指针或NULL节点)都是黑的；   
4、如果一个节点是红的,那么它的两儿子都是黑的；   
5、对于任意节点而言，其到叶子点树NULL指针的每条路径都包含相同数目的黑节点；   
6、每条路径都包含相同的黑节点；



### （3）应用

1、广泛用于C++的STL中，Map和Set都是用红黑树实现的；   
2、著名的Linux进程调度Completely Fair Scheduler，用红黑树管理进程控制块，进程的虚拟内存区域都存储在一颗红黑树上，每个虚拟地址区域都对应红黑树的一个节点，左指针指向相邻的地址虚拟存储区域，右指针指向相邻的高地址虚拟地址空间；   
3、IO多路复用epoll的实现采用红黑树组织管理sockfd，以支持快速的增删改查；   
4、Nginx中用红黑树管理timer，因为红黑树是有序的，可以很快的得到距离当前最小的定时器；   
5、Java中TreeMap的实现；

## 四、B树

### （1）简介

我们在MySQL中的数据一般是放在磁盘中的，读取数据的时候肯定会有访问磁盘的操作，磁盘中有两个机械运动的部分，分别是盘片旋转和磁臂移动。盘片旋转就是我们市面上所提到的多少转每分钟，而磁盘移动则是在盘片旋转到指定位置以后，移动磁臂后开始进行数据的读写。那么这就存在一个定位到磁盘中的块的过程，而定位是磁盘的存取中花费时间比较大的一块，毕竟机械运动花费的时候要远远大于电子运动的时间。当大规模数据存储到磁盘中的时候，显然定位是一个非常花费时间的过程，但是我们可以通过B树进行优化，提高磁盘读取时定位的效率。

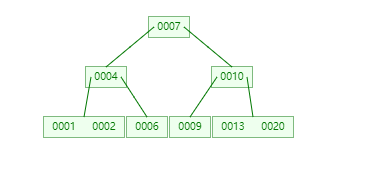
为什么B类树可以进行优化呢？我们可以根据B类树的特点，构造一个多阶的B类树，然后在尽量多的在结点上存储相关的信息，保证层数尽量的少，以便后面我们可以更快的找到信息，磁盘的I/O操作也少一些，而且B类树是平衡树，每个结点到叶子结点的高度都是相同，这也保证了每个查询是稳定的。

总的来说，B/B+树是为了磁盘或其它存储设备而设计的一种平衡多路查找树(相对于二叉，B树每个内节点有多个分支)，与红黑树相比，在相同的的节点的情况下，一颗B/B+树的高度远远小于红黑树的高度(在下面B/B+树的性能分析中会提到)。B/B+树上操作的时间通常由存取磁盘的时间和CPU计算时间这两部分构成，而CPU的速度非常快，所以B树的操作效率取决于访问磁盘的次数，关键字总数相同的情况下B树的高度越小，磁盘I/O所花的时间越少。

**注意B-树就是B树，-只是一个符号。**

### （2）B树的性质

1、定义任意非叶子结点最多只有M个儿子，且M>2；   
2、根结点的儿子数为[2, M]；   
3、除根结点以外的非叶子结点的儿子数为[M/2, M]；   
4、每个结点存放至少M/2-1（取上整）和至多M-1个关键字；（至少2个关键字）   
5、非叶子结点的关键字个数=指向儿子的指针个数-1；   
6、非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；   
7、非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；   
8、所有叶子结点位于同一层；



## 五、B+树

### （1）简介

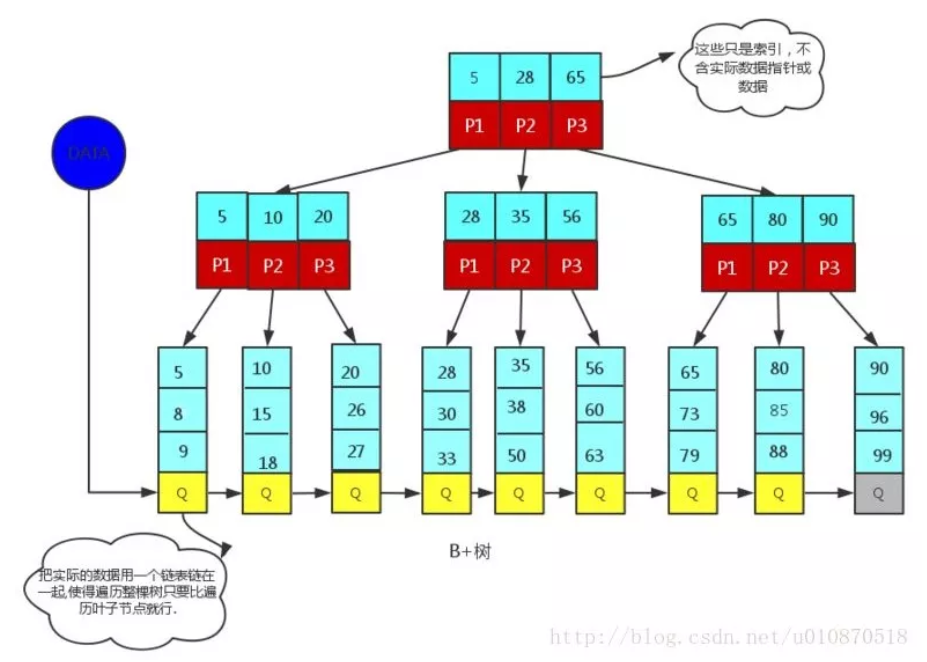
B+树是应文件系统所需而产生的一种B树的变形树（文件的目录一级一级索引，只有最底层的叶子节点（文件）保存数据）非叶子节点只保存索引，不保存实际的数据，数据都保存在叶子节点中，这不就是文件系统文件的查找吗?

我们就举个文件查找的例子：有3个文件夹a、b、c， a包含b，b包含c，一个文件yang.c，a、b、c就是索引（存储在非叶子节点）， a、b、c只是要找到的yang.c的key，而实际的数据yang.c存储在叶子节点上。

所有的非叶子节点都可以看成索引部分！

### （2）B+树的性质(下面提到的都是和B树不相同的性质)

1、非叶子节点的子树指针与关键字个数相同；   
2、非叶子节点的子树指针p[i],指向关键字值属于[k[i],k[i+1]]的子树.(B树是开区间,也就是说B树不允许关键字重复,B+树允许重复)；   
3、为所有叶子节点增加一个链指针；   
4、所有关键字都在叶子节点出现(稠密索引). (且链表中的关键字恰好是有序的)；   
5、非叶子节点相当于是叶子节点的索引(稀疏索引),叶子节点相当于是存储(关键字)数据的数据层；   
6、更适合于文件系统；



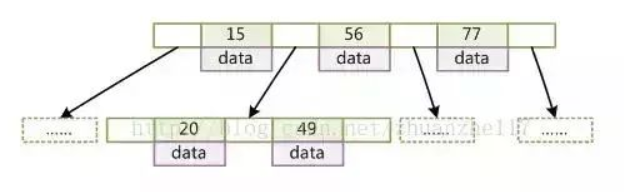
非叶子节点（比如5，28，65）只是一个key（索引），实际的数据存在叶子节点上（5，8，9）才是真正的数据或指向真实数据的指针。

### （3）应用

1、B和B+树主要用在文件系统以及数据库做索引，比如MySQL；

## B树和B+树的区别

1. B树，每个节点都存储key和data，所有节点组成这棵树，并且叶子节点指针为nul，叶子结点不包含任何关键字信息。



2.B+树，所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接

所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。 (而B 树的非终节点也包含需要查找的有效信息)

## B/B+树性能分析

n个节点的平衡二叉树的高度为H(即logn)，而n个节点的B/B+树的高度为logt((n+1)/2)+1；   
若要作为内存中的查找表，B树却不一定比平衡二叉树好，尤其当m较大时更是如此。因为查找操作CPU的时间在B-树上是O(mlogtn)=O(lgn(m/lgt))，而m/lgt>1；所以m较大时O(mlogtn)比平衡二叉树的操作时间大得多。因此在内存中使用B树必须取较小的m。（通常取最小值m=3，此时B-树中每个内部结点可以有2或3个孩子，这种3阶的B-树称为2-3树）。

## 七、为什么说B+树比B树更适合数据库索引？

1、 **B+树的磁盘读写代价更低：B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。**

2、**B+树的查询效率更加稳定：由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。**

3、由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

PS：我在知乎上看到有人是这样说的,我感觉说的也挺有道理的：

他们认为数据库索引采用B+树的主要原因是：B树在提高了IO性能的同时并没有解决元素遍历的我效率低下的问题，正是为了解决这个问题，B+树应用而生。B+树只需要去遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作或者说效率太低。

今天看了几篇文章，自己总结一下。

数据库使用B+树肯定是为了提升查找效率。

但是具体如何提升查找效率呢？

查找数据，最简单的方式是顺序查找。但是对于几十万上百万，甚至上亿的数据库查询就很慢了。

所以要对查找的方式进行优化，熟悉的二分查找，二叉树可以把速度提升到O(log(n,2))，查询的瓶颈在于树的深度，最坏的情况要查找到二叉树的最深层，由于，每查找深一层，就要访问更深一层的索引文件。在多达数G的索引文件中，这将是很大的开销。所以，尽量把数据结构设计的更为‘矮胖’一点就可以减少访问的层数。在众多的解决方案中，B-/B+树很好的适合。B-树定义具体可以查阅，简而言之就是中间节点可以多余两个子节点，而且中间的元素可以是一个域。相比B-树，B+树的父节点也必须存在于子节点中，是其中最大或者最小元素，B+树的节点只存储索引key值，具体信息的地址存在于叶子节点的地址中。这就使以页为单位的索引中可以存放更多的节点。减少更多的I/O支出。因此，B+树成为了数据库比较优秀的数据结构，MySQL中MyIsAM和InnoDB都是采用的B+树结构。不同的是前者是非聚集索引，后者主键是聚集索引，所谓聚集索引是物理地址连续存放的索引，在取区间的时候，查找速度非常快，但同样的，插入的速度也会受到影响而降低。聚集索引的物理位置使用链表来进行存储。