C++STL通用的数据结构：vector(动态数组), list(链表), queue(队列), stack(栈), set(红黑树) ,map(红黑树), deque(双向队列)， hash\_map(hashtable)

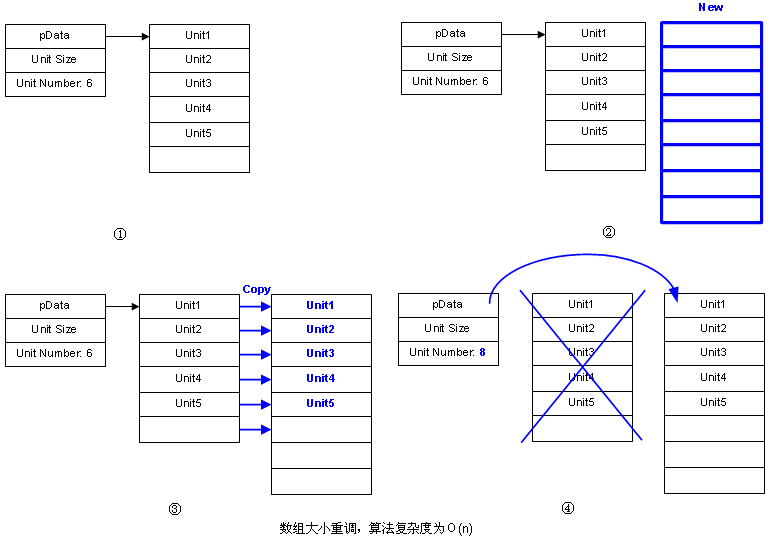
<http://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

## 动态数组

数组指向一块连续的内存空间，它可以非常快速的通过arr[i]的方式来获取数组的内容，但它的插入，删除和扩容是复杂．为了保证数据在内存中的连续存放，插入的时候需要将插入节点后的数据统一的向下移动，删除就需要上移．下图是一个数组扩容的一个过程，必须先重新分配空间再memcpy到新的空间，最后再进行旧的内存清空．

任意节点的访问效率：O(1)

添加删除：O(n)



动态数组的我们平时也是经常使用如C++中的vector和java中的ArrayList，它就是一个动态数组，我们也可以认真读nginx的数组代码：

struct ngx\_array\_s {

void \*elts; /\*数据指针\*/

ngx\_uint\_t nelts; /\*已用节点数\*/

size\_t size; /\*单个节点所需内存大小\*/

ngx\_uint\_t nalloc; /\*已分配的节点数\*/

ngx\_pool\_t \*pool; /\*内存池\*/

};

/\*动态数组创建\*/

ngx\_array\_t \*ngx\_array\_create(ngx\_pool\_t \*p, ngx\_uint\_t n, size\_t size);

/\*动态数组销毁\*/

void ngx\_array\_destroy(ngx\_array\_t \*a);

/\*添加新节点，同时返回数组的新节点指针\*/

void \*ngx\_array\_push(ngx\_array\_t \*a);

使用例子：

   arr = ngx\_array\_create(pool, 10, sizeof(ngx\_uint\_t));  
   for (n=0; n < 5; n++) {  
      ele = (int\*) ngx\_array\_push(arr);  
      \*ele = n;  
      printf("new element %d added\n", n);  
   }

   ngx\_array\_destroy(arr);

## 位图

位图（bitmap）是一种非常常用的结构，在索引，数据压缩等方面有广泛应用。本文介绍了位图的实现方法及其应用场景。

**位图实现**

http://dongxicheng.org/wp-content/uploads/2011/05/bitmap.jpg

**自己实现**

在位图中，每个元素为“0”或“1”，表示其对应的元素不存在或者存在。

#define INT\_BITS sizeof(int)

#define SHIFT 5 // 2^5=32 ////移动5个位,左移则相当于乘以32,右移相当于除以32取整

#define MASK 0x1f // 2^5=32

#define MAX 1024\*1024\*1024 //max number

int bitmap[MAX / INT\_BITS];

/\*

\* 设置第i位

\* i >> SHIFT 相当于 i / (2 ^ SHIFT),

\* i&MASK相当于mod操作 m mod n 运算

\*/

void set(int i) {

bitmap[i >> SHIFT] |= 1 << (i & MASK);

}

/\*获取第i位\*/

int test(int i) {

return bitmap[i >> SHIFT] & (1 << (i & MASK));

}

/\*清除第i位\*/

int clear(int i) {

return bitmap[i >> SHIFT] & ~(1 << (i & MASK));

}

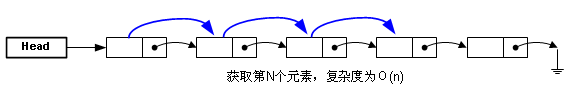
http://blog.csdn.net/QIBAOYUAN/article/details/5914662

## 链表

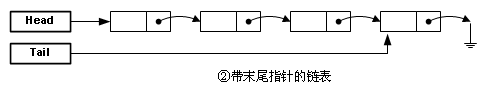
查询慢，查询的时候需要做遍历过程．O(n)

但相对数组，它的删除，插入和扩容都是要快．O(1)

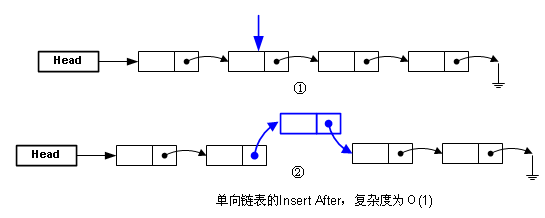
在任意位置插入和删除元素的效率都是很高．但随机访问支持不好，需要遍历元素．



链表的查询实在是太慢，为了快速的在最后一个节点添加数据，通常会设置一个tail指针指向最后节点：



它的优势是插入的时候快：

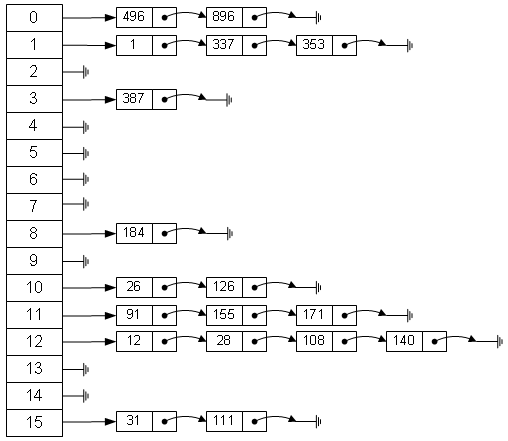


Collection  
├List       接口  
│├LinkedList       链表  
│├ArrayList         顺序结构动态数组类  
│└Vector         向量  
│ └Stack       栈

对应STL中的List容器

## 哈希

结合了动态数组和链表的优点，保证了数据快速查询，快速插入删除．它就是一个”链表的数组”．对应JAVA中的Map接口：Hashtable



数据需要平均的散列在各个数组节点，散列的方法通常有：

1. 除法取模法． Index = value % size (arr)，如：31%16 = 15
2. 斐波那契（Fibonacci）散列法

index = (value \* 2654435769) >> 28

1)，对于16位整数而言，这个乘数是40503  
2)，对于32位整数而言，这个乘数是2654435769  
3)，对于64位整数而言，这个乘数是11400714819323198485

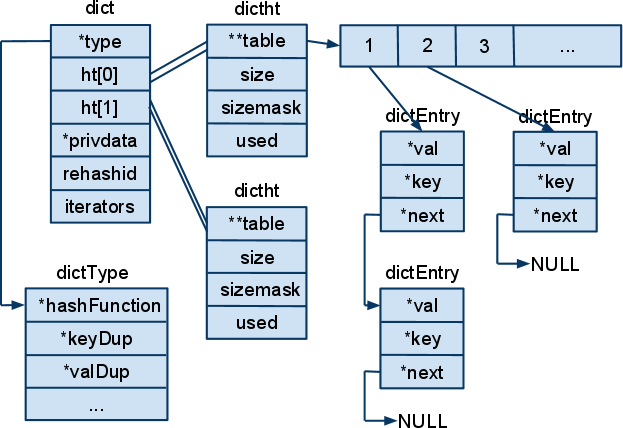
1. BKDR　Hash法

4．Redis中用到的哈希算法：字符串哈希函数使用的是Bernstein方法，整数是Thomas Wang的方法。

…..

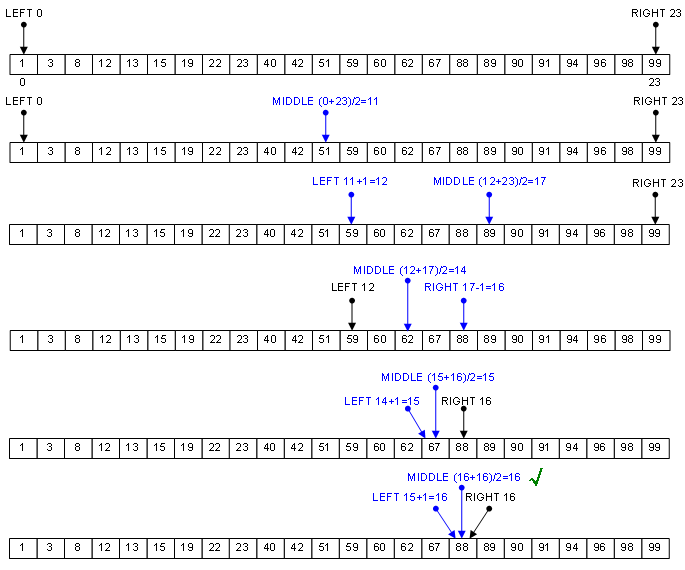
方法很多不一一列出．

Rehash：将hash的长度做成动态，方便对内存的管理，很多时候是不能准确的将要分配的槽位。如redis中就分配了两个hash，其中的ht[1]就是用来过渡保存rehash的计算值。

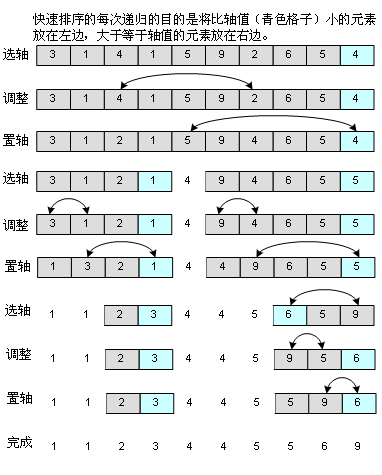


## 排序

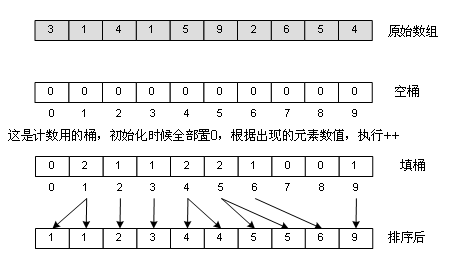
二分法排序 O(log(n))



快速排序法：

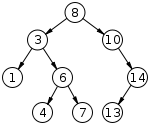


桶排序（Bucket Sort）  
这是迄今为止最快的一种排序法，其时间复杂度仅为Ο(n)，也就是线性复杂度！不可思议吧？但它是有条件的。举个例子：一年的全国高考考生人数为500万，分数使用标准分，最低100，最高900，没有小数，你把这500万元素的数组排个序。我们抓住了这么个非常特殊的条件，就能在毫秒级内完成这500万的排序，那就是：最低100，最高900，没有小数，那一共可出现的分数可能有多少种呢？一共有900-100+1=801，那么多种，想想看，有没有什么“投机取巧”的办法？方法就是创建801个“桶”，从头到尾遍历一次数组，对不同的分数给不同的“桶”加料，比如有个考生考了500分，那么就给500分的那个桶（下标为500-100）加1，完成后遍历一下这个桶数组，按照桶值，填充原数组，100分的有1000人，于是从0填到999，都填1000，101分的有1200人，于是从1000到2019，都填入101……如图：

  
很显然，如果分数不是从100到900的整数，而是从0到2亿，那就要分配2亿个桶了，这是不可能的，所以桶排序有其局限性，适合元素值集合并不大的情况。

对应bitmap数据结构实现方式

## 二叉查询树(Binary Search Tree)



O(log n)

特点：右小左大

二叉排序树又称为二叉查找树。它满足如下性质的二叉树：

（1）若它的左子树非空，则左子树上所有结点的值均小于根结点的值。

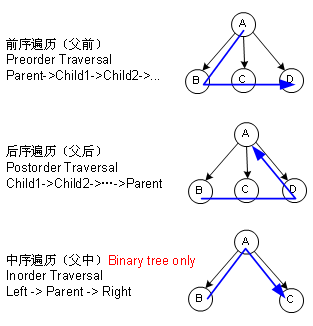
（2）若它的右子树非空，则右子树上所有结点的值均大于根结点的值。

（3）它的左、右子树本身又各是一棵二叉排序树。

应用于:

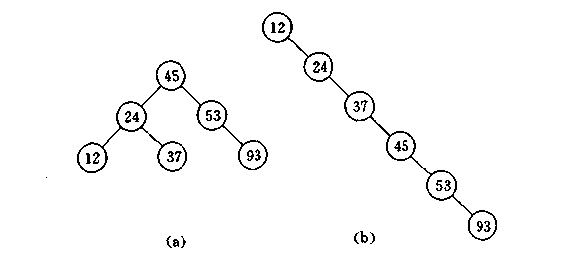
二分法排序查询 ,可将查询时间控制在对数Log2(N)时间范围内.N为树高．

遍历：



问题:

如果树不是按我们预期的来排列，同样的一组数据,组织不同，它的查询性能也是会大打折扣．如图所示:

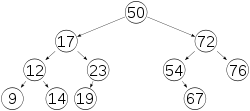


那么平衡的二叉树将帮我们解决这个问题．

## AVL树

平衡二叉树(Balanced binary tree)是由阿德尔森-维尔斯和兰迪斯(Adelson-Velskii and Landis)于1962年首先提出的，所以又称为AVL树。

平衡的二叉排序树BBST (Balancing Binary Search Tree)



AVL树的特性：

（1）若它的左子树非空，则左子树上所有结点的值均小于根结点的值。

（2）若它的右子树非空，则右子树上所有结点的值均大于根结点的值。

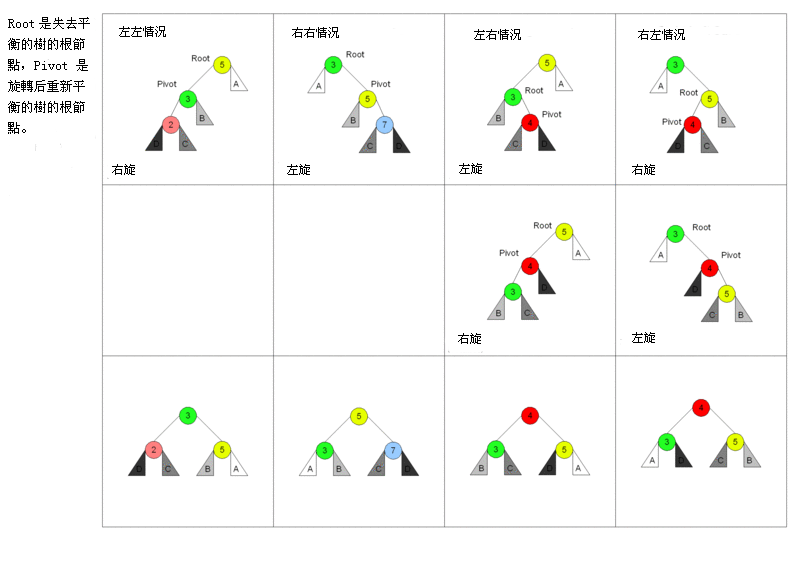
（3）左右子树深度之差的绝对值不超过1;

节点的**平衡因子**BF(Balance Factor) 为该结点的左子树的深度减去它的右子树的深度： 平衡因子BF=左子树深度－右子树深度

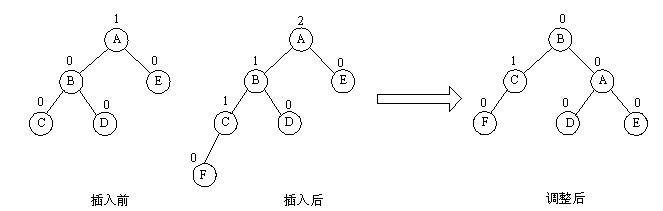
The **balance factor** of a node is the height of its left subtree minus the height of its right subtree (sometimes opposite)

带有平衡因子1、0或 -1的节点被认为是平衡的。带有平衡因子 -2或2的节点被认为是不平衡的，并需要重新平衡这个树。平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。

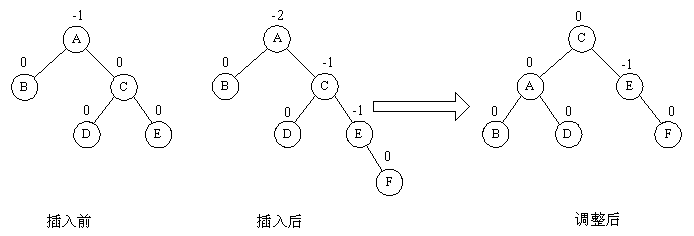
旋转再平衡过程：



平衡因子变化情况：



右旋



左旋

操作逻辑：

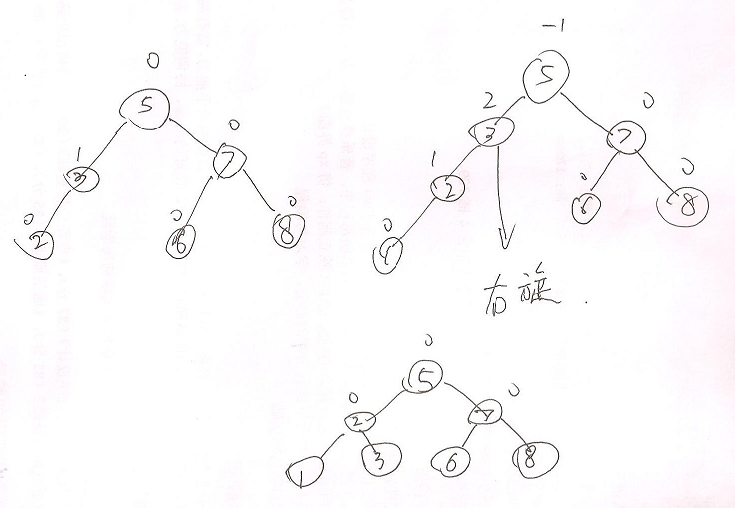
(1).添加和删除时首先要向上逐层调整平衡因子，直到根节点

(2).判断是否需要平衡化？重点看平衡因子：

if (current->bf == -2)     L\_Balance(c\_root);

else if (current->bf == 2)  R\_Balance(c\_root);

重复一二步骤，做一次或多次的"AVL旋转"。



节点为实际数值，节点外为平衡因子．

查询：O(log n)

添加删除：O(n)

## 红黑树

红黑树并不追求“完全平衡”——它只要求部分地达到平衡要求，降低了对旋转的要求，从而提高了性能。AVL是要求高度的平衡．

在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:

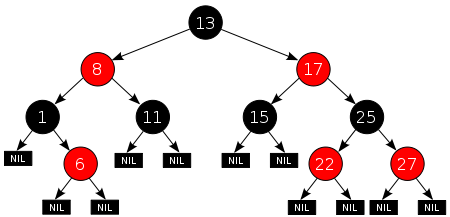
性质1. 节点是红色或黑色。

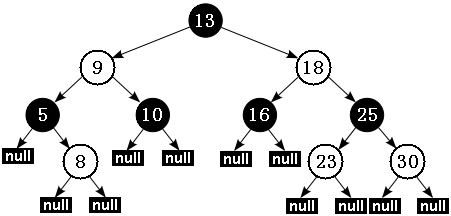
性质2. 根是黑色。

性质3. 所有叶子都是黑色（叶子是NIL节点）。

性质4. 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

性质5. 从任一节点到其每个叶子的所有[简单路径](http://zh.wikipedia.org/wiki/%E9%81%93%E8%B7%AF_(%E5%9B%BE%E8%AE%BA))都包含相同数目的黑色节点。





红黑树和AVL树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。

红黑树和AVL树的区别在于它使用颜色来标识结点的高度，它所追求的是局部平衡而不是AVL树中的非常严格的平衡。AVL树的复杂比起红黑树来说简直是小巫见大巫。红黑树是真正的变态级数据结构。

插入操作：

每次都红色节点

**AVL vs 红黑树**

在极端的条件下 avl的一次插入可能导致从叶到根的调整，所以可能为logn级的，旋转操作本身当然都是O(1)的 红黑树最多只需要3次换色，旋转就可重新实现局部平衡 

红黑树能够以**O(***log2* **n)** 的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。

红黑树并不适应所有应用树的领域。如果数据基本上是静态的，那么让他们待在他们能够插入，并且不影响平衡的地方会具有更好的性能。如果数据完全是静态的，例如，做一个哈希表，性能可能会更好一些。

应用：

STL中的SET就是以红黑树()

Nginx里面，处理超时连接的模型比之lighttp高效了很多：

当前所有可能被触发的定时器被保存在红黑树这种数据结构中，通过红黑树，你可以很快的得到距离当前最快发生的定时器事件的时间差。

将这个时间差作为select/poll/epoll等函数的参数，也就是，最多等待这么长时间就返回。  
当函数返回时，得到函数调用总共花费了多少时间，根据这个时间取出红黑树的根节点比较查看是否应该触发该定时器时间，如果可以则将该定时器从红黑树中删除，然后继续查看新的成为树根的定时器节点，这个过程一直进行下去直到没有定时器满足被触发的条件，也就是还没有到被触发的事件。

Nginx里面，新接收了一个连接，会保存这个连接上来的时间，并且以这个时间来加入红黑树定时器中。

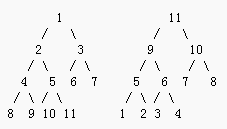
可以看到，因为引入了红黑树这个数据结构，所有的定时器都可以按照顺序来依次取出，这样不用轮询所有事件来查看是否超时了；而以距离当前最快发生的定时器事件时间差作为轮询的定时，又可以不用使用alarm信号来触发定时，一举两得。

这个处理超时事件的模型，与我之前分析过的libevent的模型，大体相同，只不过，在那里红黑树被堆代替了。

## 二叉堆

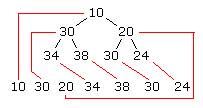
二叉堆是一种特殊的堆，二叉堆是完全二叉树或者是近似完全二叉树。

当父结点的键值总是大于或等于任何一个子节点的键值时为最大堆。 当父结点的键值总是小于或等于任何一个子节点的键值时为最小堆。



左图: 1 2 3 4 5 6 7 8 9 10 11

右图: 11 9 10 5 6 7 8 1 2 3 4



**插入**

首先把要添加的元素加到数组的末尾，然后和它的父节点（位置为当前位置除以2取整，比如第4个元素的父节点位置是2，第7个元素的父节点位置是3）比较，如果新元素比父节点元素小则交换这两个元素，然后再和新位置的父节点比较，直到它的父节点不再比它大，或者已经到达顶端，及第1的位置。

只需要将节点插在二叉树的最后一个叶子结点位置，然后比较它对它父亲节点的大小，如果大则停止；如果小则交换位置，然后对父亲节点递归该过程直至根节点。复杂度为O(log n)。

**删除**

要从堆中删除一个节点，用最后一个节点替换掉根节点，然后调整节点顺序以维持堆特性。

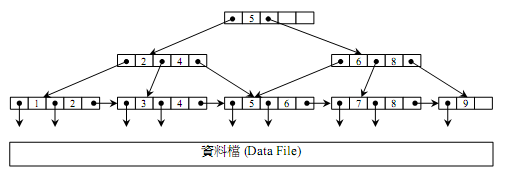
## B+树：

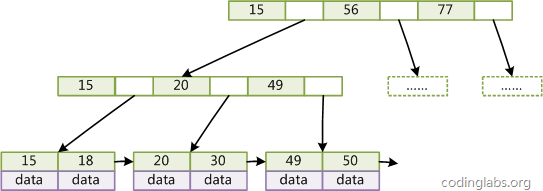
通常用于数据库和操作系统的文件系统中。B+树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+ 树元素自底向上插入，这与二叉树恰好相反。较为出名的一个B+树实例是NTFS文件系统中的文件索引。

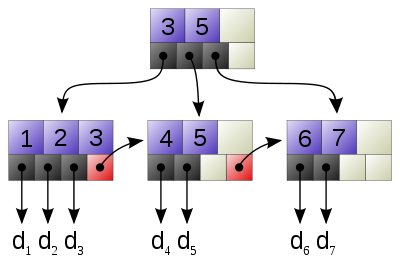
与内存相比，硬盘必须花成倍的时间来存取一个数据元素，这是因为硬盘的机械部件读写数据的速度远远赶不上纯电子媒体的内存。与一个结点两个分支的二元树相比，B-tree/B+tree利用多个分支（称为子树）的结点，减少获取记录时所经历的结点数，从而达到节省存取时间的目的。

特点：

* 所有的叶子结点中包含了全部关键字的信息，及指向含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，结点中仅含其子树（根结点）中的最大（或最小）关键字。
* 在实现时，通常在B+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。可以通过最小关键字顺序查询，也可以通过根结点随机查询．

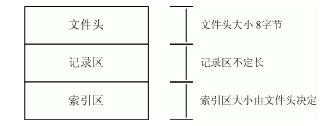


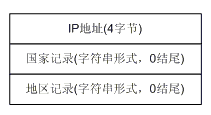


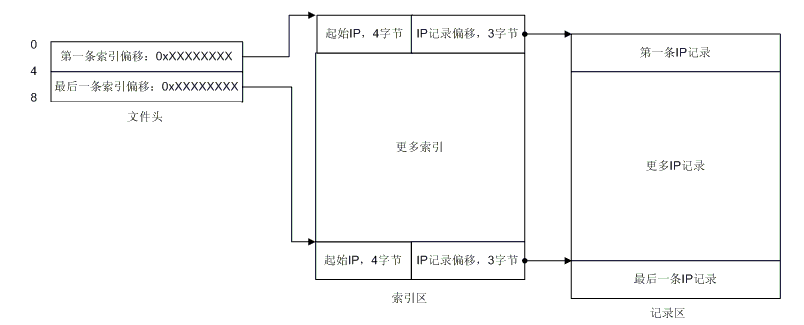


**纯真IP查询**

<http://lumaqq.linuxsir.org/article/qqwry_format_detail.html>

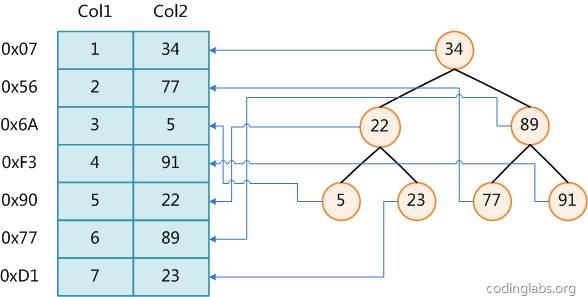






## 索引:

什么是索引，索引是为方便快速的查询到数据，如果直接对表检索性能必然是低下，如果我们把数据放入到二叉树中，它的查询时间复杂将是O(log n)，时间上大为改进。如下所示，如果要查询23的记录，如果顺序查询我们需要7次的查询时间，如果通过二叉树只需要2次查询就完成定位。



发展历史:

全表扫描

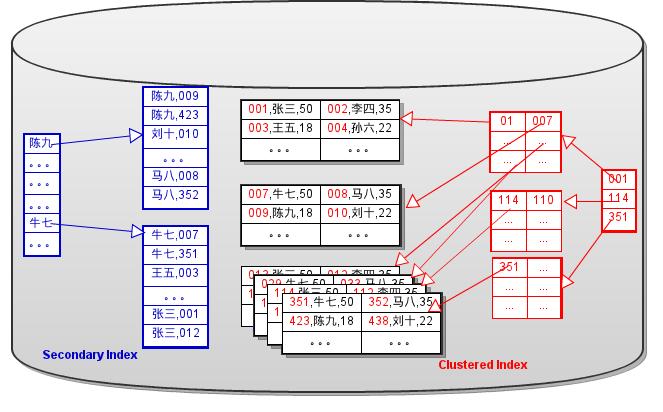
Dense Index

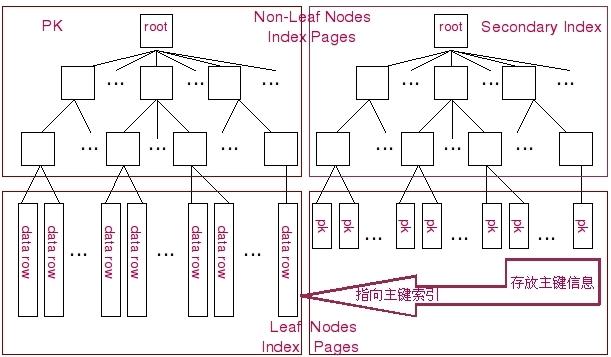
折半块查询

Sparse Index

多层Sparse Index

簇索引Clustered Index 辅助索引





虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

**第一个重大区别是InnoDB的数据文件本身就是索引文件。**MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

**聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。**

Innodb请谨记，使用业务无关的自增长字段做为主键。

## MySQL

has index:cardno

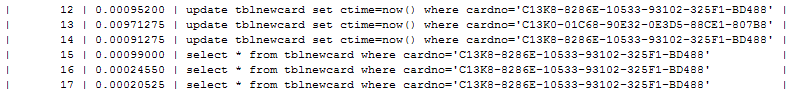
记录数:45000

测试情况：

innodb:

SELECT第一次加载到内存的时间相对要长：

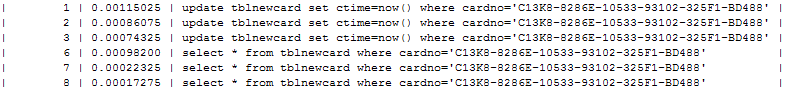
UPDATE每次的执行时间一样



myisam:

SELECT第一次加载到内存的时间相对要长：

UPDATE每次的执行时间一样

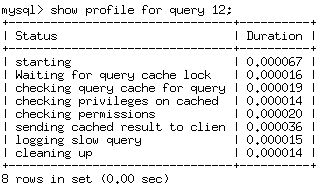


Query Cache.

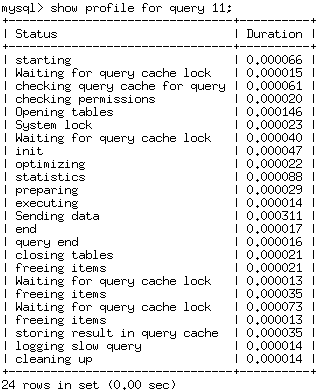
为减轻数据库的压力,MySQL会将一些SELECT的结果进行缓存,它们以SQL语句为Key进行缓存,当有相同的SQL语句出现的时候就会直接从Query Cache中提取.

但数据库表的DDL/DML会影响Query Cache.一旦发生表结构变动,或者任何的记录添加或修改,MySQL会清空Query Cache中的相关缓存记录.所以对于频繁更新的数据库表,不适合做Query Cache. 数据集发生变化,缓存数据就一定会出现结果不同,如按性别按渠道查询.

从QC缓存提取数据:



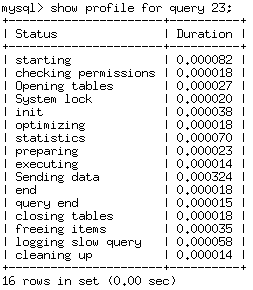
非QC缓存：



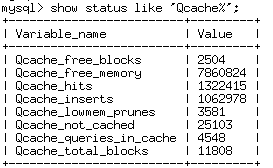
关闭QC:

mysql> set session query\_cache\_type = off;

Query OK, 0 rows affected (0.00 sec)



Query Cache命中次数



Qcache\_inserts 被加入到缓存中的查询数目

Qcache\_hits 缓存命中数目

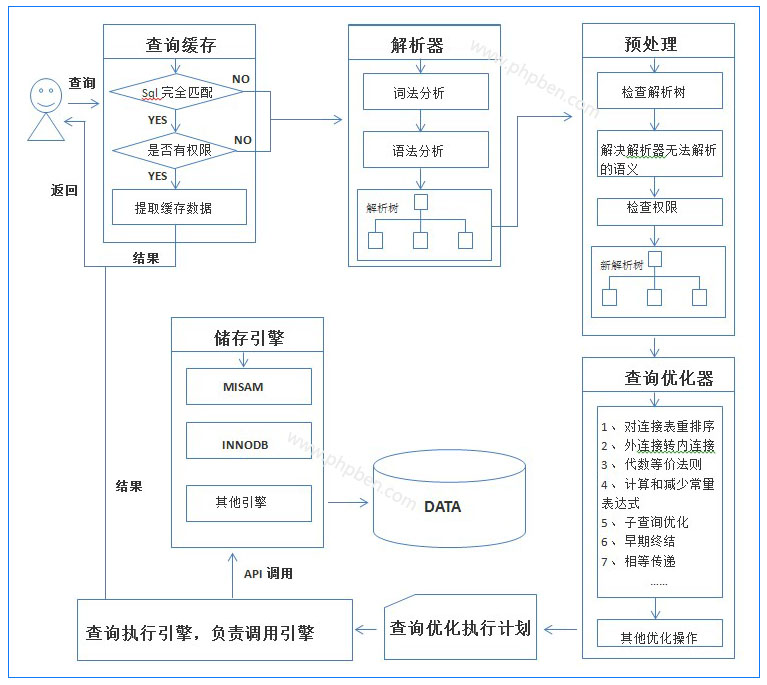
Qcache\_queries\_in\_cache 当前缓存中的查询数目

Qcache\_not\_cached 没有被缓存的查询结果数目（不能被缓存的如INSERT或由于Query\_cache\_type关闭）

从上表命中的次数还是可观的.

在高并发，写入量大的MySQL系统，建系把该功能禁掉。

无缓存的查询SQL可以提长15%以上



## 准备知识：

### 大Ｏ符号：

|  |  |
| --- | --- |
| **符号** | **名称** |
| \Omicron(1)\! | [常数](http://zh.wikipedia.org/wiki/%E5%B8%B8%E6%95%B0)（阶，下同） |
| \Omicron(\log n)\! | [对数](http://zh.wikipedia.org/wiki/%E5%AF%B9%E6%95%B0) |
| \Omicron(n)\! | [线性](http://zh.wikipedia.org/wiki/%E7%B7%9A%E6%80%A7)，次线性 |

### 对数：

3^4=3\cdot3\cdot3\cdot3=81，

4=\log_381\!，

自然对数，底数为１０:

10^2=100

log10(100)=2

二进制对数，底数为２：

2^3=8

Log28=3 (二叉查询)

## 参考:

由浅入深理解索引的实现

<http://www.mysqlops.com/2011/11/24/understanding_index.html>

<http://www.mysqlops.com/2011/12/20/understanding_index2.html>

二叉查找树

<http://zh.wikipedia.org/wiki/%E4%BA%8C%E5%8F%89%E6%9F%A5%E6%89%BE%E6%A0%91>

AVL树

<http://zh.wikipedia.org/wiki/AVL%E6%A0%91>

红黑树

<http://zh.wikipedia.org/wiki/%E7%BA%A2%E9%BB%91%E6%A0%91>

通过分析 JDK 源代码研究 TreeMap 红黑树算法实现

<http://www.ibm.com/developerworks/cn/java/j-lo-tree/index.html?ca=drs->

从B树、B+树、B\*树谈到R 树

<http://blog.csdn.net/v_july_v/article/details/6530142>

纯真IP数据库格式详解:

<http://lumaqq.linuxsir.org/article/qqwry_format_detail.html>

数据结构

<http://www.cnblogs.com/yc_sunniwell/category/232388.html>

MySQL索引背后的数据结构及算法原理

<http://www.codinglabs.org/html/theory-of-mysql-index.html>