索引与算法

基本数据结构与算法

顺序数组

二分查找

顺序表自然支持二分查找，时间复杂度为o(log_n)

二叉查找树

规则：二叉查找树的左节点小于根节点，右节点大于根节点。
性能分析：查询复杂度、构造复杂度和删除复杂度的分析可知，三种操作的时间复杂度皆为o(log_n)~o(n)。

查找效率

最优情况:当二叉查找树是一个完全二叉树的时候，查找效率为o(log_n)
最差情况:当二叉树变成一个链表的时候，查询效率为o($n/2$)

构造复杂度

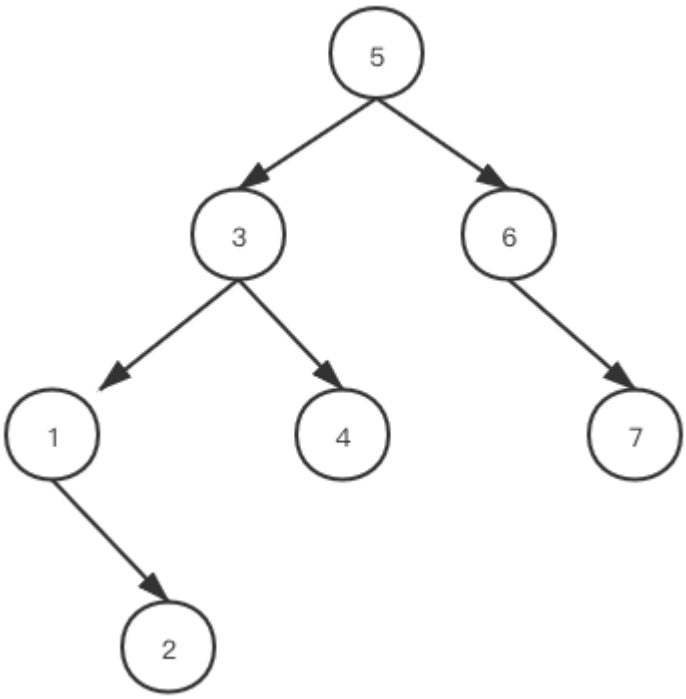
> 二叉搜索树的构造过程，也就是将节点不断插入到树中适当位置的过程。该操作过程，与查询节点元素的操作基本相同，不同之处在于：查询节点过程是，比较元素值是否相等，相等则返回，不相等则判断大小情况，迭代查询左、右子树，直到找到相等的元素，或子节点为空，返回节点不存在插入节点的过程是，比较元素值是否相等，相等则返回，表示已存在，不相等则判断大小情况，迭代查询左、右子树，直到找到相等的元素，或子节点为空，则将节点插入该空节点位置,因此时间复杂度是o(log_n)~o(n)。

删除复杂度

> 二叉搜索树的节点删除包括两个过程，查找和删除。查询的过程和查询复杂度已知，删除就是在查询的基础上进行单点的删除常数复杂度，因此删除的复杂度也是o(log_n)~o(n)。

平衡二叉树

平衡二叉树（Balanced BinaryTree）又被称为AVL树。它具有以下性质：它是一棵空树或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。



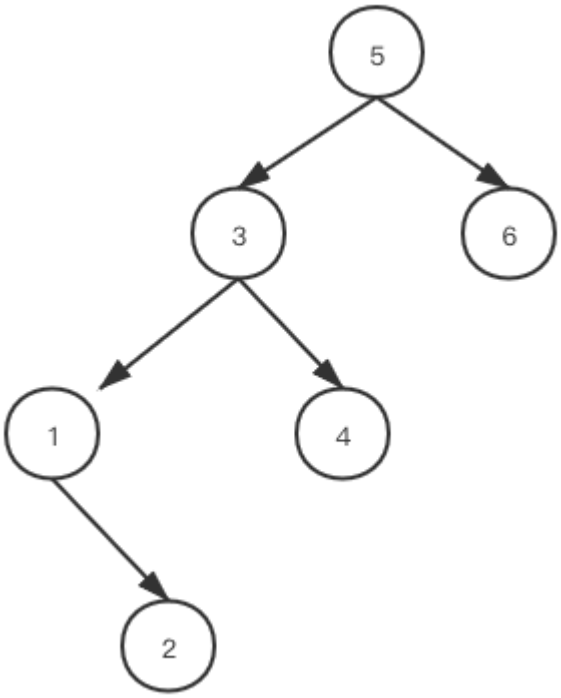
平衡因子的计算

平衡因子 = 父节点的左子树高度 - 父子树的右子树高度

> - 如上图中的5节点的左子树高度为3,右子树高度为2所以5节点的平衡因子是1
> - 3节点的左子树高度为2,右子树高度为1,平衡因子也是1
> - 6节点和1节点的左子树高度为0,右子树高度为1,平衡因子为-1

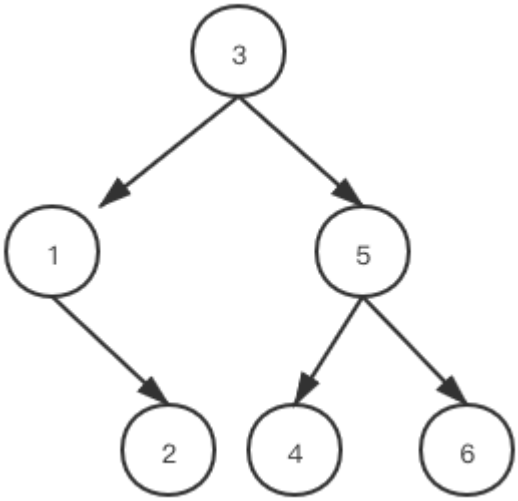
右旋

左左情况(LL) LL 情况是指根节点的平衡因子为 2，根节点的左子节点平衡因子为 0 或 1 。如下图:



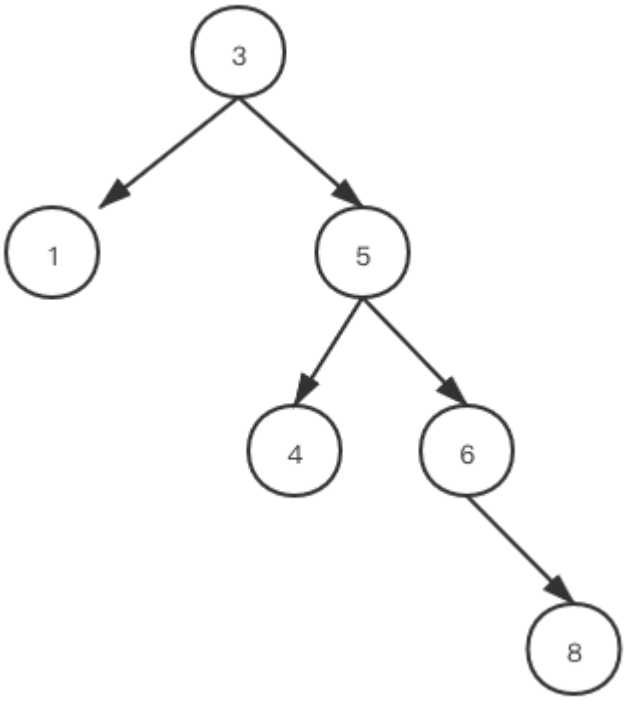
> 注：产生该情况的场景可能是在节点6右侧删除了节点7，也可能是增加了节点2导致的，对于该场景为了保证二叉树的平衡，需要采用右旋操作。

- 设置根节点 root 的左子节点为新的根节点 $root_{new}$
- 将 $root_{new}$ 节点的右子树作为 root 节点的左子树，将 root 节点作为 $root_{new}$ 的右子树，即降低“左子树”高度，提升“右子树”高度，使得新的左右子树高度趋于平衡；结果如下图：



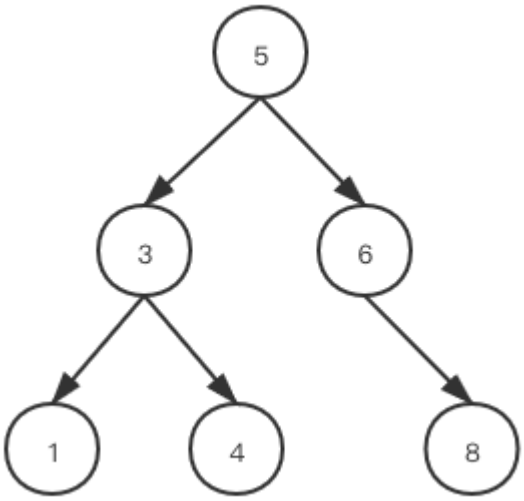
左旋

右右情况(RR) LL 情况是指根节点的平衡因子为 -2，根节点的左子节点平衡因子为 0 或 1 。如下图:



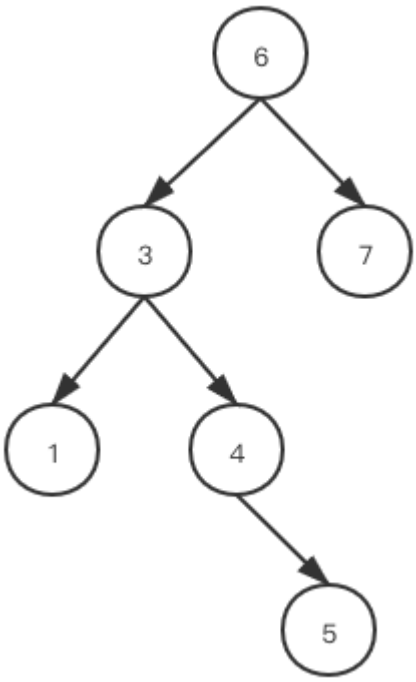
> 注：产生该情况的场景可能是在节点1右侧删除了节点2，也可能是增加了节点8导致的，对于该场景为了保证二叉树的平衡，需要采用左旋操作。

- 设置根节点 root 的右子节点为新的根节点 $root_{new}$
- 将 $root_{new}$ 节点的左子树作为 root 节点的右子树，将 root 节点作为 $root_{new}$ 的左子树，即降低“右子树”高度，提升“左子树”高度，使得新的左右子树高度趋于平衡；结果如下图：



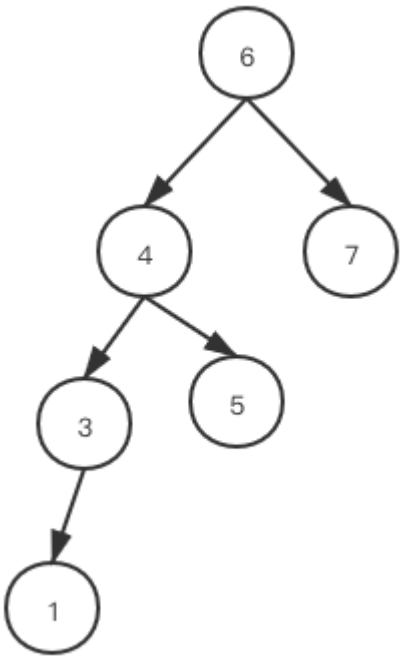
左旋+右旋

左右情况(LR) 该情况下根节点的平衡因子与左左情况相同，都为 2，不同之处在于左子节点的平衡因子为 -1，若按照之前直接进行右旋操作，则旋转操作后二叉树扔处于不平衡状态。如下图:

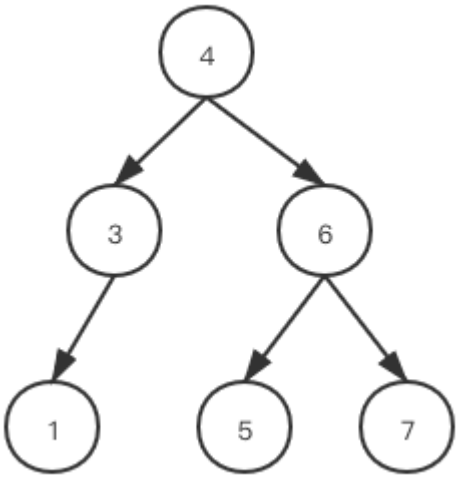


> 注：产生该情况的场景可能是在节点4右侧增加了节点5，也可能是节点7删除节点8导致的，对于该场景为了保证二叉树的平衡，需要采用左子树左旋+右旋操作。

- 对左子树进行左旋结果

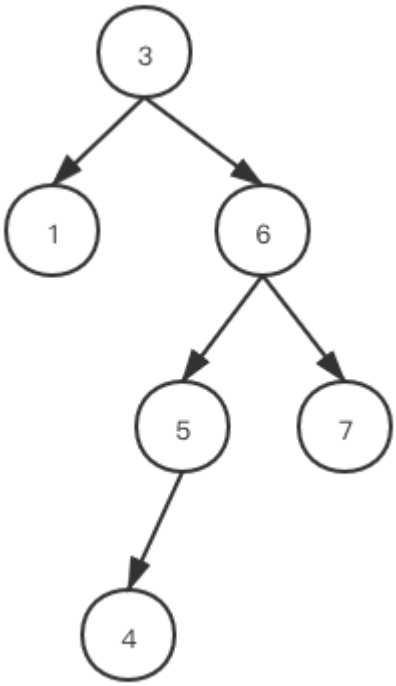


- 对二叉树执行右旋操作



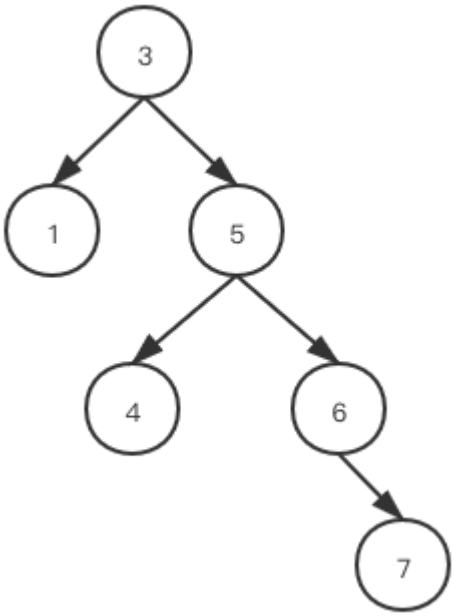
右旋+左旋

右左情况(RL) 该情况与上面左右情况对称，根节点的平衡因子为 -2，右子节点平衡因子为 1，需要首先对右子树进行右旋操作，调整二叉树为 RR 情况，再对二叉树执行左旋操作。如下图:

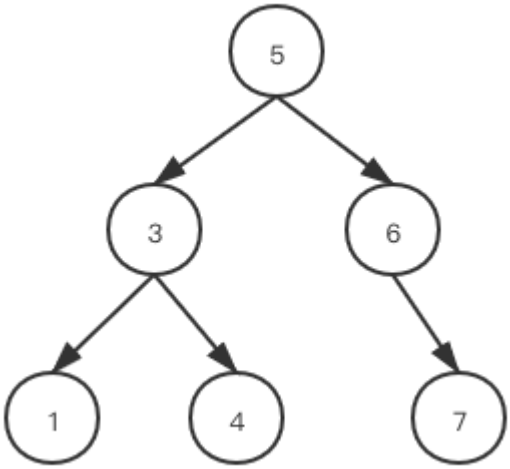


> 注：产生该情况的场景可能是在节点1右侧删除了节点2，也可能是节点5增加了节点4导致的，对于该场景为了保证二叉树的平衡，需要采用右子树右旋+左旋操作。

- 对右子树进行右旋结果



- 对二叉树执行右旋操作



性能分析

> 相对于二叉搜索树，平衡二叉树避免了向线性结构演化的倾向，查询、插入和删除节点的时间复杂度为 $O(log_2N)$~$O(log_{\frac{1+\sqrt5}2}N)$。因为每个节点上需要保存平衡因子，所以空间复杂度要略高于二叉搜索树。

Hash表

定义

> 根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。

存在意义

> 传统的数组和链表当作存储接口的时候并不能同时保证寻址（链表只能顺序查找）和存取（数组只能顺序存储）的效率，使用hash表可以同时均衡存取和寻址。

期望效果

> - 单向推导，不可以反推导出原始数据（数据的安全性）
> - 对于数据的微小变化，hash值可以充分变化
> - 从原始数据计算hash值的时候效率高
> - Hash映射函数的冲突概率要小

常用映射函数

> - 加法Hash：将值中的每一位进行相加然后取余一个很大的素数
> - 位运算Hash： Hash函数通过利用各种位运算（常见的是移位和异或）来充分的混合输入元素。
> - 乘法Hash：利用乘法的不相关性（FNV算法）
> - 除法Hash（效率太低没人用）
> - 查表hash：根据表格映射查找出hash值
> - 混合hash：结合上述几种方式的hash（MD5；Tiger）

如何解决冲突

> - 开放定址法（散列法） 顺序再散列：如果发生冲突则找该位置的下一个位置 二次探测再散列，如果碰到冲突，就在左右进行跳跃式探测
> - 再哈希法：同时构造多个hash函数，hash冲突的时候，在执行另外的hash函数进行再hash（不易冲突但是计算量大）
> - 链地址法：这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

应用场景

> - 版权校验：用于保证数据的唯一性，所有数据以第一次提交的为主（避免用户重复提交）
> - 大文件分块校验：将文件分块上传的时候怎么验证数据的完整性，将原始数据进行分块然后进行hash处理后生成hashlist并将数据和hashlist上传，到了服务器后将文件再次进行hash，将结果拼接成rootHash，与之前的hashlist拼成的roothash对比，如果一致则表明数据一致，如果不一致则对比每一块数据的hash判断哪块数据损坏了。
> - 负载均衡（一致性Hash） 参考：https://iwiki.woa.com/pages/viewpage.action?pageId=74061517&from=km_search
> - 其他应用：地图距离以及地图范围筛选操作的GeoHash（比如现在的外卖服务，使用geohash在数据库中进行城市范围或者片区范围内的小哥筛选like geohash字符串中的前n位数据），大数据过滤的布隆过滤器

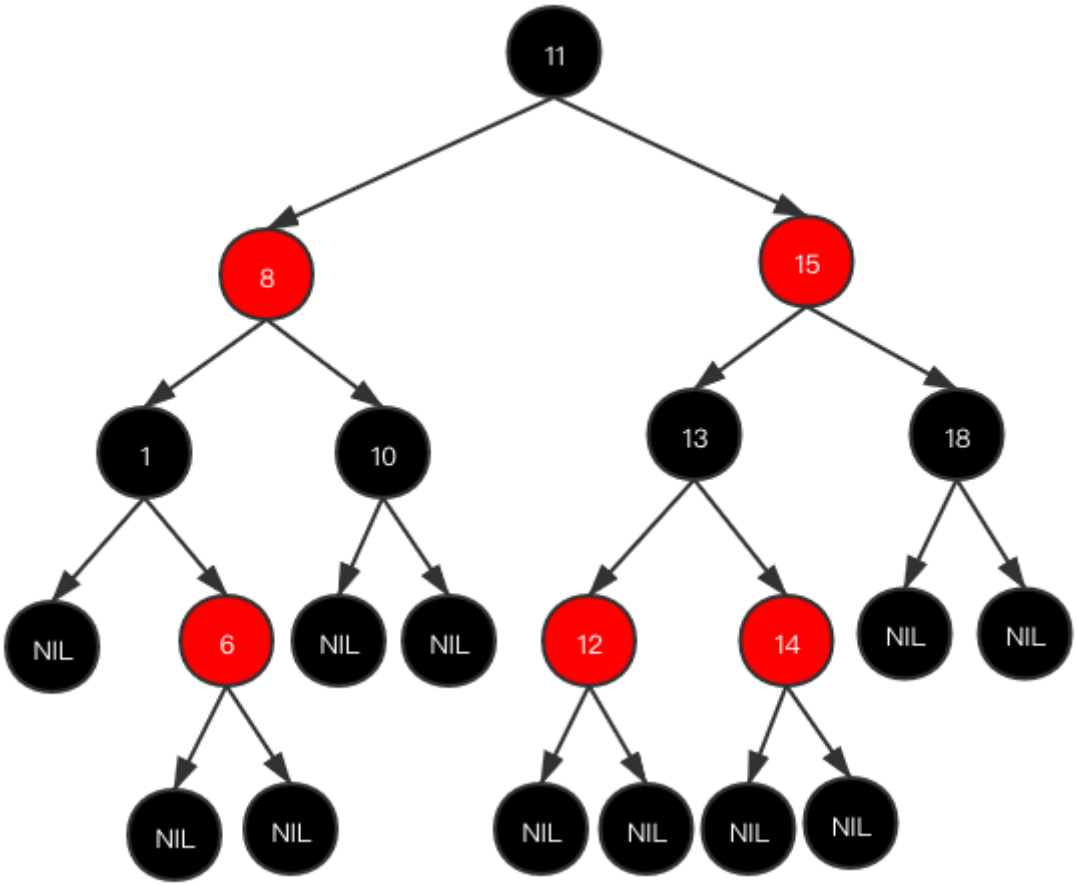
红黑树

性质

> - 每个节点要么是黑色，要么是红色。
> - 根节点是黑色。

> - 每个叶子节点（NIL）是黑色。
> - 每个红色结点的两个子结点一定都是黑色。
> - 任意一结点到每个叶子结点的路径都包含数量相同的黑结点。

下图就是一颗典型的红黑树



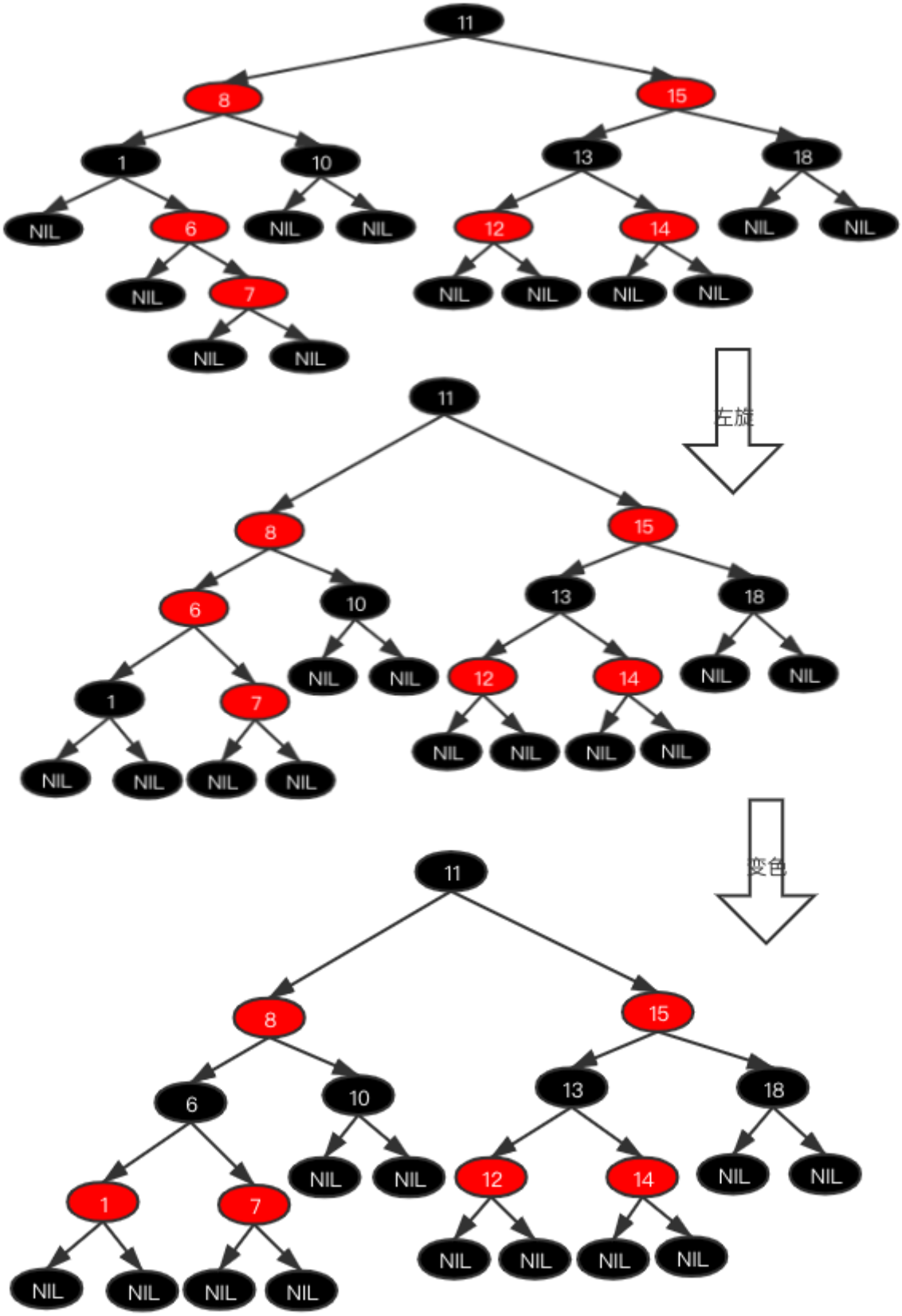
红黑树查找效率

红黑树的查找效率为O(logn),原因如下：

> - 根据限制规则5，假设红黑树从根节点到叶子节点有n个黑色节点，再根据限制规则4，每个红色结点的两个子结点一定都是黑色，因此红黑树从根节点到叶子节点最多有n+1个红色节点。因此，从根到叶子的最长的可能路径(\$2*n\$)*不多于最短的可能路径(*\$2n-2\$)的两倍长(n>1)。
> - 红黑树的高度不高于黑色节点的2倍。
> - 一棵拥有n个内部结点(不包括叶子结点)的红黑树的树高h<=2log(n+1)

红黑树操作

> 从上图的红黑树中，插入数字7，则违反了红黑树的规则，需要采用变色或者左旋右旋操作(类似与AVL树)将红黑树恢复。恢复过程如下：



应用

- java1.8以后的hashmap就是采用红黑树+链表实现的。
- C++ 的STL set/map容器

B树

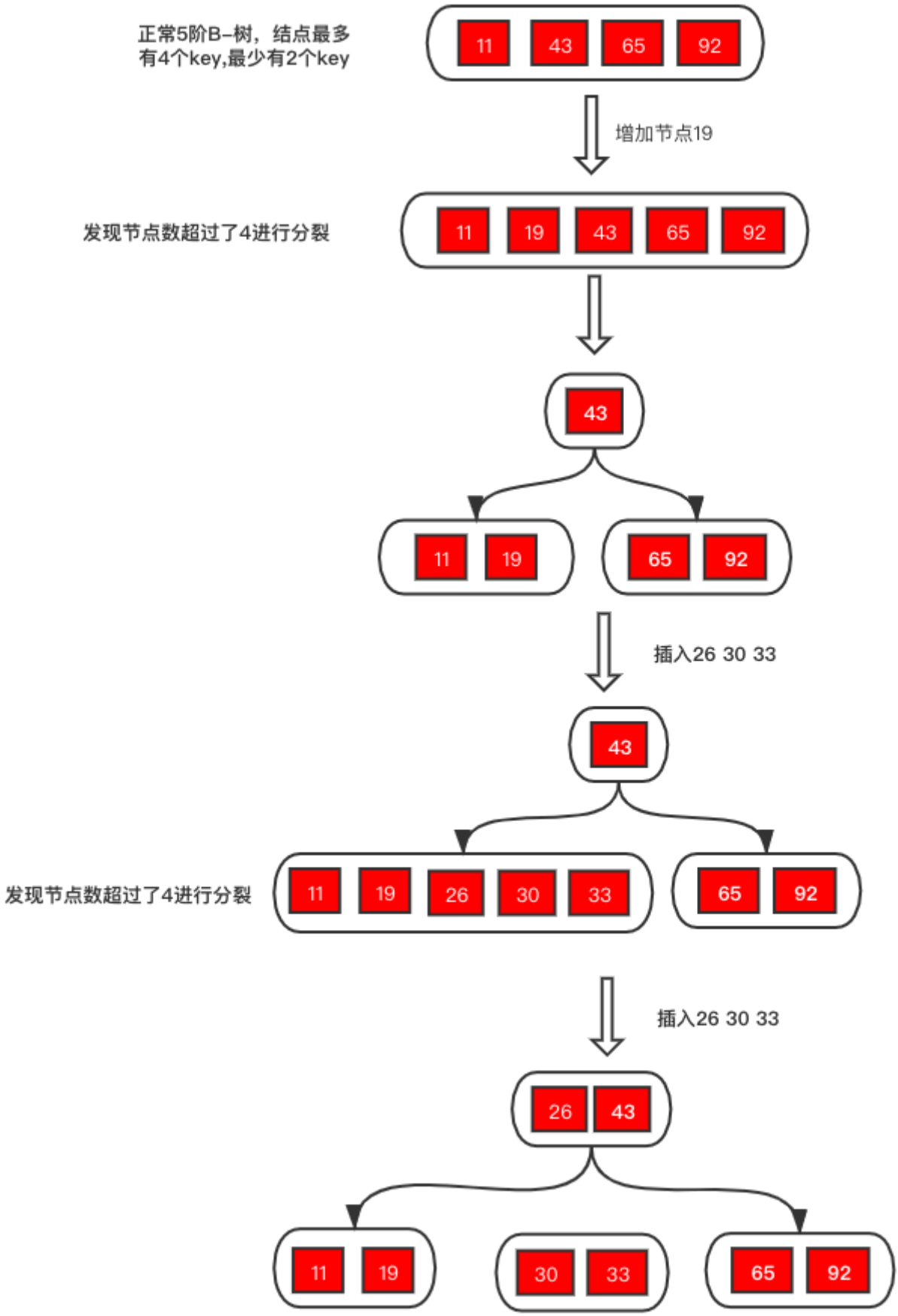
B树是一种多路平衡查找树，它的每一个节点最多包含k个孩子，k被称为B树的阶。

B树规则

> - 根结点至少有两个子女。
> - 每个中间节点都包含k-1个元素和k个孩子，其中 m/2 <= k <= m
> - 每一个叶子节点都包含k-1个元素，其中 m/2 <= k <= m
> - 所有的叶子结点都位于同一层。
> - 每个节点中的元素从小到大排列，节点当中k-1个元素正好是k个孩子包含的元素的值域分划。

B树的插入

> 判断当前结点key的个数是否小于等于m-1，如果满足，直接插入即可，如果不满足，将节点的中间的key将这个节点分为左右两部分，中间的节点放到父节点中即可。示例如下：

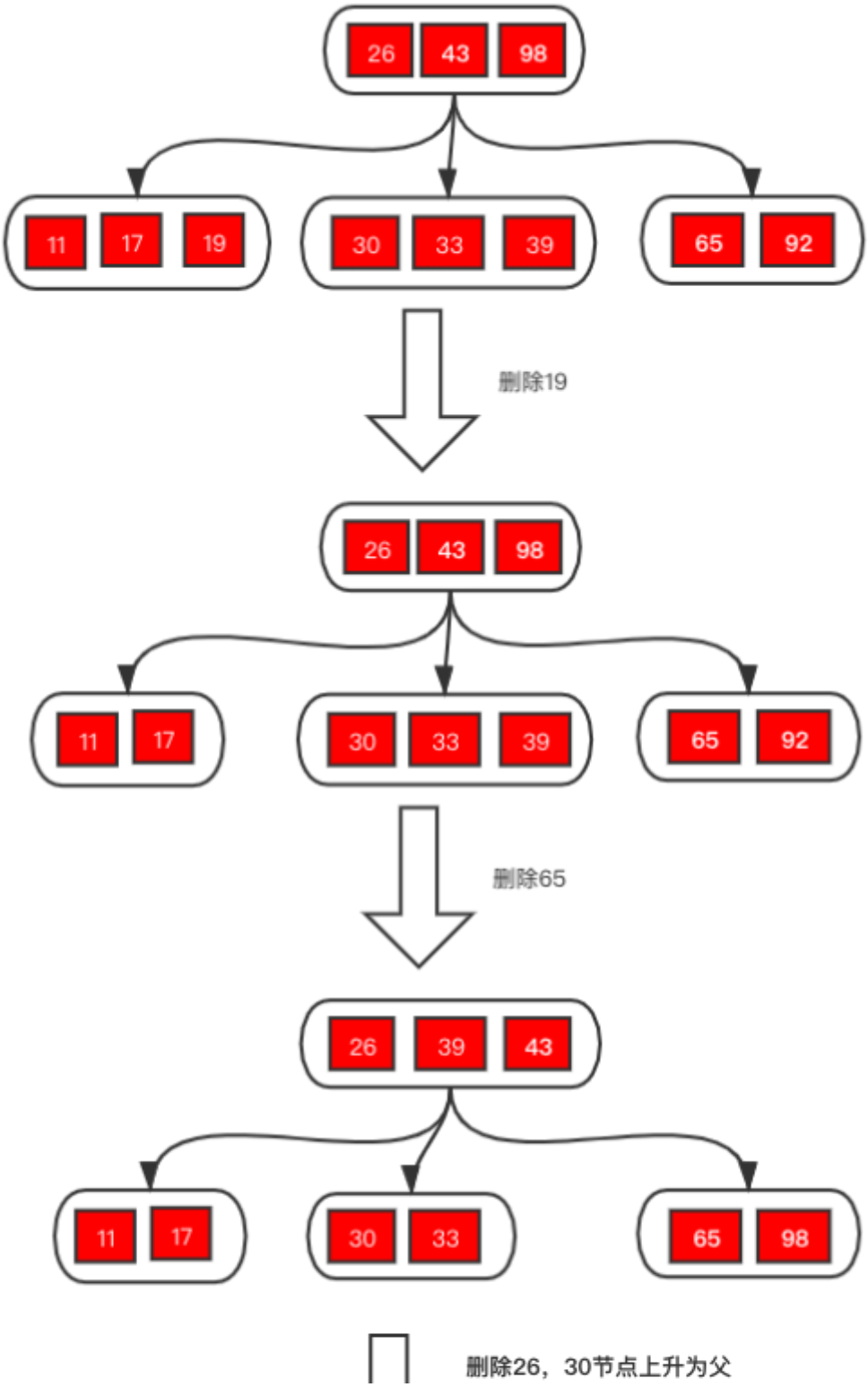

正常5阶B-树，结点最多
有4个key,最少有2个key
11 43 65 92
增加节点19
发现节点数超过了4进行分裂
11 19 43 65 92
43
11 19
65 92
插入26 30 33
43
发现节点数超过了4进行分裂
11 19 26 30 33
65 92
插入26 30 33
26 43
11 19
30 33
65 92

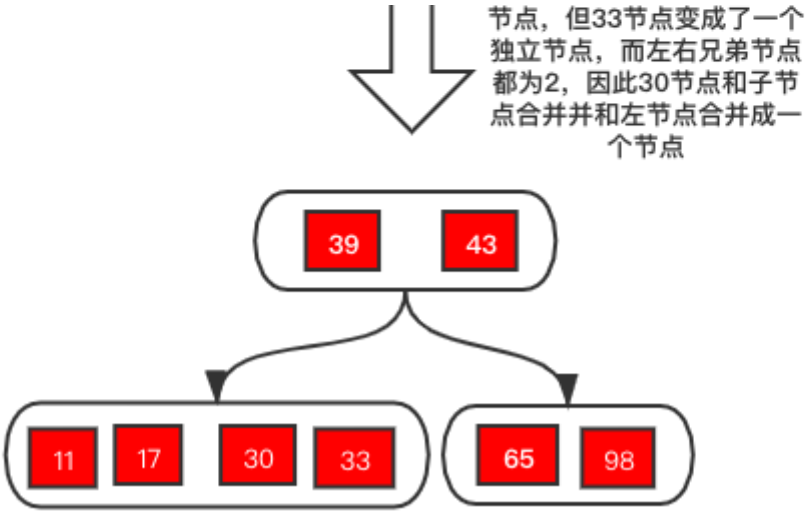
B树的删除

步骤：

1. 如果当前需要删除的key位于非叶子结点上，则用后继key（这里的后继key均指后继记录的意思）覆盖要删除的key，然后在后继key所在的子支中删除该后继key。此时后继key一定位于叶子结点上，这个过程和二叉搜索树删除结点的方式类似。删除这个记录后执行第2步
2. 该结点key个数大于等于Math.ceil(m/2)-1，结束删除操作，否则执行第3步。
3. 如果兄弟结点key个数大于Math.ceil(m/2)-1，则父结点中的key下移到该结点，兄弟结点中的一个key上移，删除操作结束。否则，将父结点中的key下移与当前结点及它的兄弟结点中的key合并，形成一个新的结点。原父结点中的key的两个孩子指针就变成了一个孩子指针，指向这个新结点。然后当前结点的指针指向父结点，重复第2步。

示例如下：





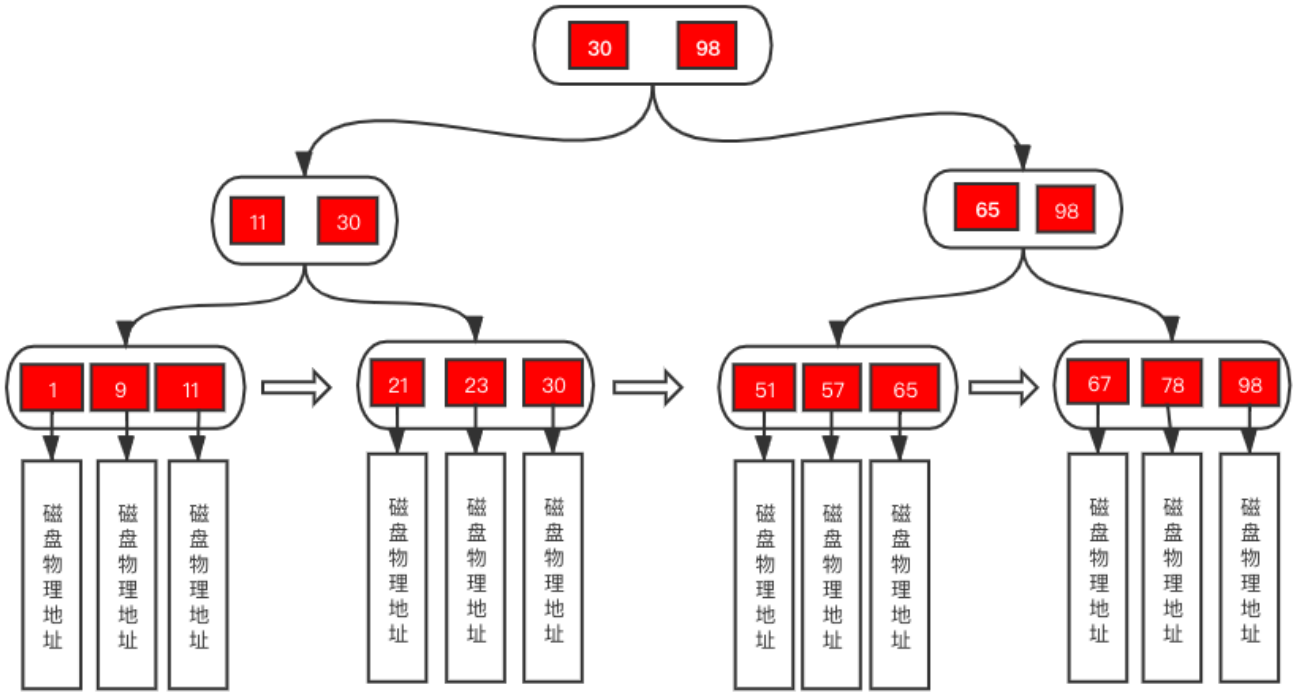
B+树

> B+树由B树衍生而来，因此插入删除等操作与B树类似(不赘述)

B+树规则

> - 有k个子树的中间节点包含有k个元素（B树中是k-1个元素），每个元素不保存数据，只用来索引，所有数据都保存在叶子节点。
> - 所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
> - 所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素。

典型B+树如下图:



B+树索引

各种数据结构的对比以及使用场景

BST树,AVL树与红黑树

- 插入,删除时间复杂度对比:
  BST树 < 红黑树 < AVL树
- 查询时间复杂度对比:
  BST树 > 红黑树 > AVL树
- 应用场景

> 1. BST树查询极端情况(树型为链表)时间复杂度$o(n)$,因此在计算机内部很少使用
> 2. 二叉平衡树在Windows NT内核中广泛存在;
> 3. 红黑树广泛用于C ++的STL中；着名的Linux的的进程调度，进程控制块；IO多路复用的epoll；Nginx的管理定时器，Java的的的中TreeMap中的中的实现;

Hash表与B树

- 扩展性差，需要提前预测数据量的大小，哈希表是基于数组的，数组创建后难于扩展某些哈希表被基本填满时，性能下降得非常严重，所以程序员必须要清楚表中将要存储多少数据（或者准备好定期地把数据转移到更大的哈希表中，这是个费时的过程）。因为动态Hash一直是一个比较难的问题，所以开始为了保证较合适的填充因子，所以不得不开一个比较大的空间来存储索引，此时数据内容的条数可能并不是很多。而B树，扩展性比较好。
- 不能有序遍历数据 没有一种简便的方法可以以任何一种顺序〔例如从小到大）遍历表中数据项。B树阔以轻松搞定（中序遍历即可：O（N * log(ceil(m/2) N)）），B+树阔以更轻松地搞定（扫一遍叶子结点即可：O（N））。
- B树磁盘IO次数少 而hash如果同一个桶里的数据如果比较多，难免增加不少IO次数。而磁盘IO次数往往决定了索引速度。

B树,B+树和红黑树比较

- B树和红黑树的区别

  > 在大规模数据存储的时候，红黑树往往出现由于树的深度过大而造成磁盘IO读写过于频繁，进而导致效率低下的情况。为什么会出现这样的情况，我们知道要获取磁盘上数据，必须先通过磁盘移动臂移动到数据所在的柱面，然后找到指定盘面，接着旋转盘面找到数据所在的磁道，最后对数据进行读写。磁盘IO代价主要花费在查找所需的柱面上，树的深度过大会造成磁盘IO频繁读写。根据磁盘查找存取的次数往往由树的高度所决定，所以，只要我们通过某种较好的树结构减少树的结构尽量减少树的高度，B树可以有多个子女，从几十到上千，可以降低树的高度。

- B树和B+树的区别

> 1. B树则所有节点都带有带有指向记录（数据）的指针（ROWID），B+树中只有叶子节点会带有指向记录(数据)的指针（ROWID）。因此B树遍历到节点就可以获取到数据，而B+树需要遍历到叶子节点才能获取到数据。此外,由于B+树没有在非叶子节点存储指向记录的指针,因此在相同内存下可以加载的索引就越多，可以定位到更多数据。
> 2. B+树中每个叶子节点都包含指向下一个叶子节点的指针。在范围查找的时候可以顺序便利，不需要再重新使用中序遍历。
> 3. B+树每个节点的指针和key一样多，B树每个节点指针比key多1。

对比总结

- Hash表，查询速度为O(1),但是 对于区间查找不适用，如果数据量不断增涨，扩容时间慢，当桶内由多个数据的时候，需要多次磁盘IO去获取数据。
- BST树，当数据库存储大量数据的时候，存在极端情况整个BST树极度不平衡，导致查找时间达到o(n)。
- AVL树：该树在查找插入和删除的消耗都是O(lgn),再插入数据的时候最多旋转1次，但是该树在删除的时候会导致树的失衡而进行旋转，量级为O(lgn)
- 红黑树：会造成树的高度过高，在内存的时候存取性能可以，但是在磁盘存取的时候会导致磁盘IO问题。为什么树太高的时候会影响磁盘IO？因为每次只能读取一个节点导致会进行h（树的高度）次单个节点的IO查找。
- B树：磁盘IO有个有个特点，就是从磁盘读取1B数据和1KB数据所消耗的时间是基本一样的，所以尽可能在一次磁盘IO中多读一点数据到内存。这一点很符合B树的一个节点多个key的构造，从结构上看，B树比较矮，进行的磁盘io数量会比较少。B树有以下几个优点：1.优秀检索速度，时间复杂度：B树的查找性能等于O（h*logn），其中h为树高，n为每个节点关键词的个数；2.尽可能少的磁盘IO，加快了检索速度；3.可以支持范围查找。
- B+树相对于B树一个节点能存很多索引，可以同时过滤更多数据。B+树的叶子节点是数据阶段用了一个链表串联起来，便于范围查找。

索引分类

在innodb中，根据不同的维度可以将索引分为聚集索引、辅助索引、联合索引、覆盖索引以及全文索引(倒排索引)。

聚集索引

聚集索引就是按照每张表的主键构造的一颗B+树，同时叶子结点中存放的是整张表的行记录数据。因此叶子节点也被称为数据页，非叶子节点被称为索引页。

例如创建表t_test

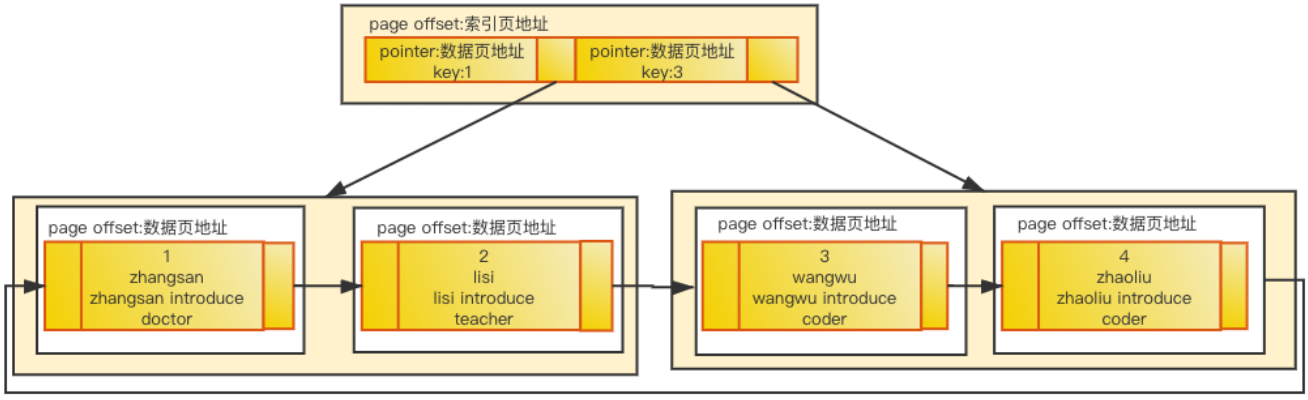
```
create table t_test(id int UNSIGNED AUTO_INCREMENT,name varchar(64) NOT
NULL,
    info text NOT NULL, title varchar(64) NOT NULL, PRIMARY KEY (`id`))
ENGINE = InnoDB DEFAULT CHARACTER SET = utf8mb4
```

插入数据

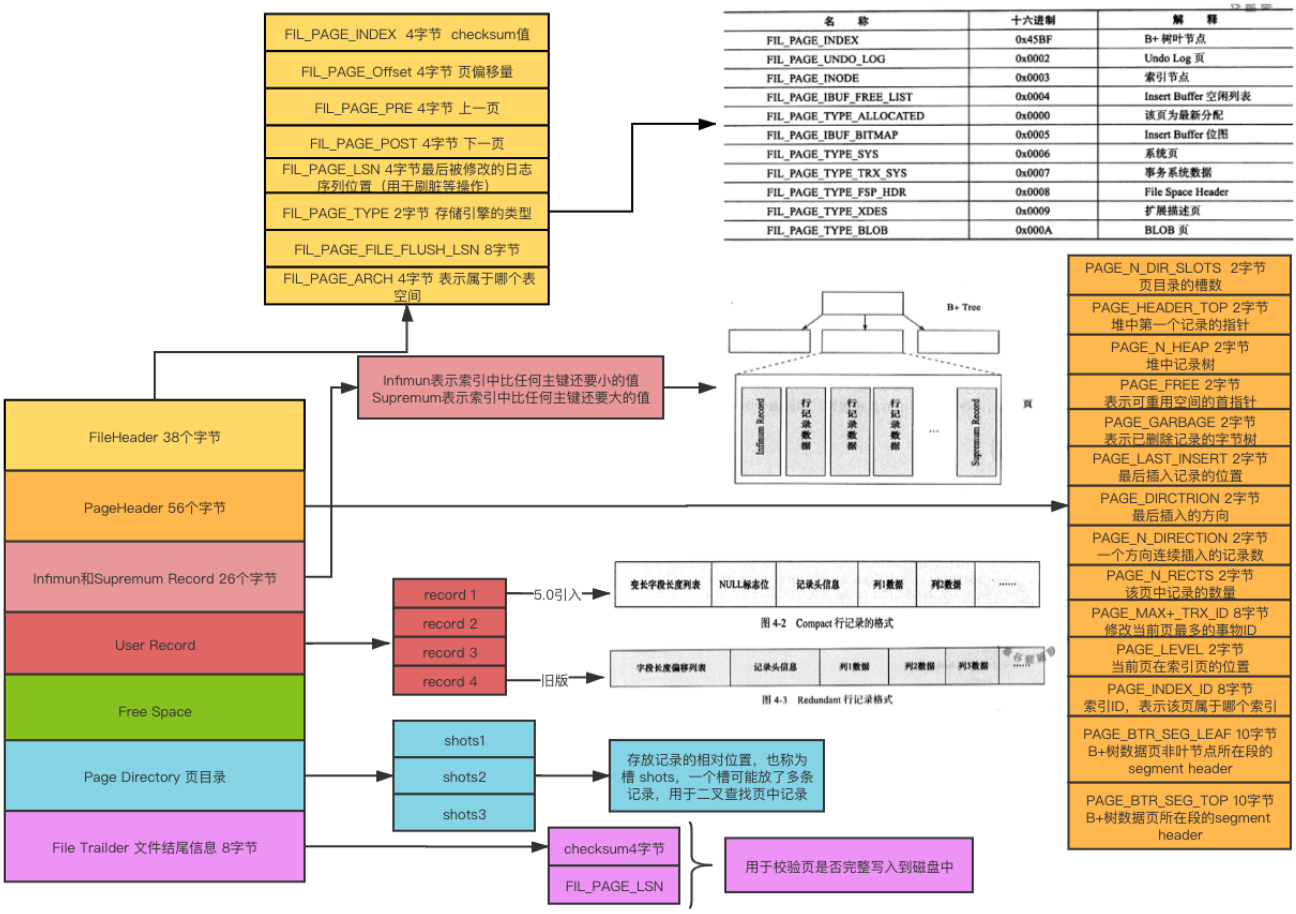
```
insert into table t(name, info,title)values("zhangsan","zhangsan
introduce", "doctor")
insert into table t(name, info,title)values("lisi","lisi introduce",
"teacher")
insert into table t(name, info,title)values("wangwu","wangwu introduce",
"coder")
```

```
insert into table t(name, info,title)values("zhaoliu","zhaoliu introduce",
"coder")
```

则产生的聚集索引(假设是一个三阶B+假设一条记录一页，因为innodb中B+树记录是指向记录所在的页) 如下图所示



innodb中非叶子节点存储在索引页中，根据B+树的结构pointer存储的是数据的数据页地址。数据页结构如下：



> 聚集索引是用双向链表链接起来的，按照主键排序的，是逻辑上连续的，是不是物理连续的。此外聚集索引由于叶子有双向链表，因此排序查找和范围查找效率非常快。

辅助索引

innodb的辅助索引结构和聚集索引差不多，但是辅助索引的叶子节点存储的不是数据，而是数据的主键索引键。辅助索引与聚集索引的关系如下图(摘自innodb存储引擎)：

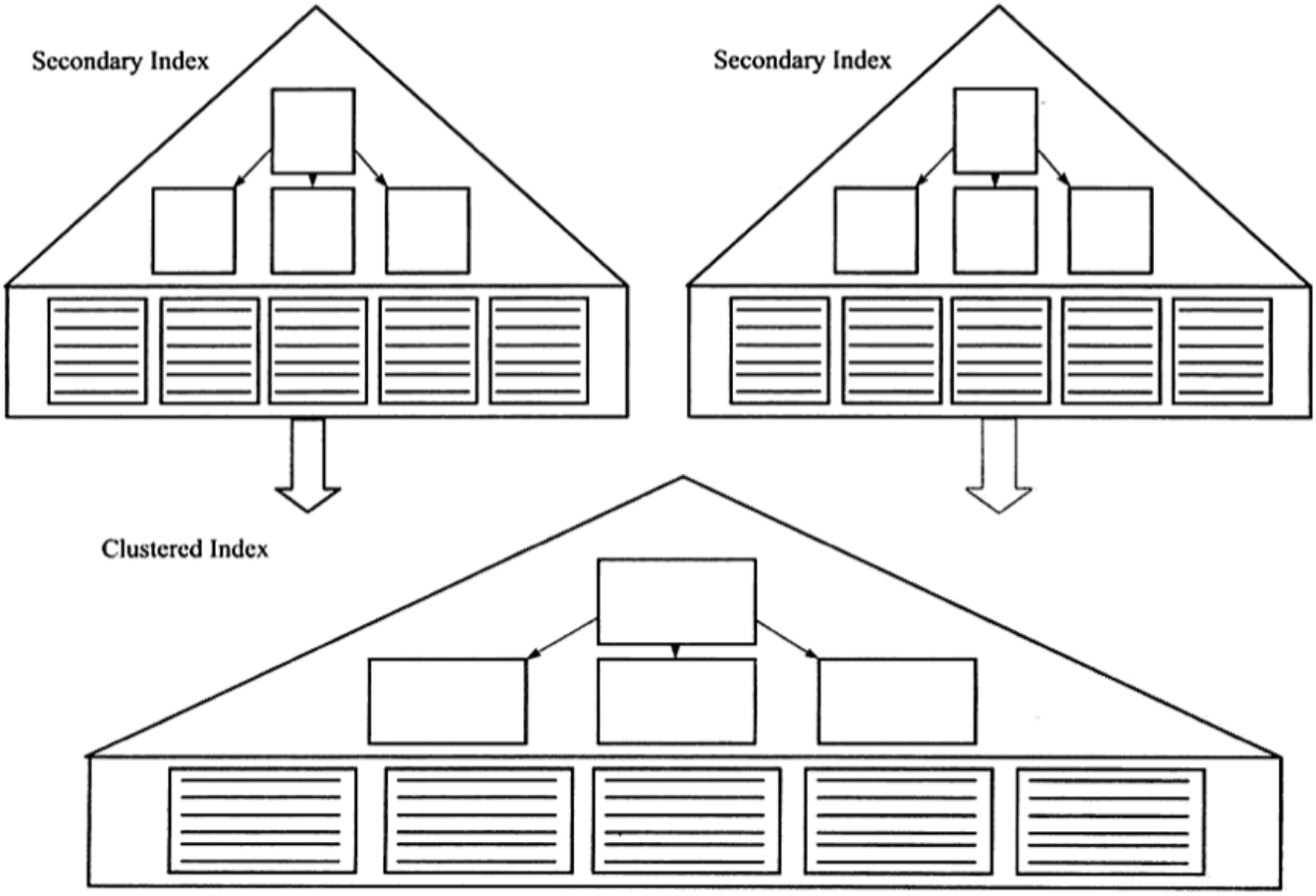
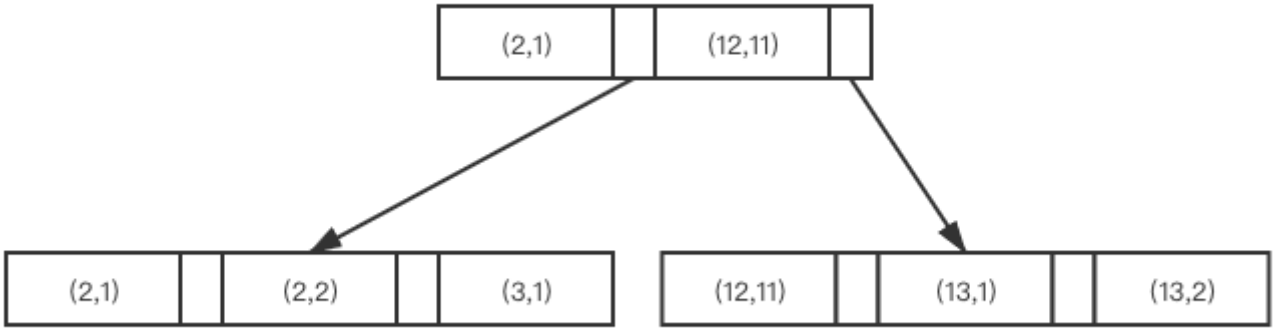


图 5-15　辅助索引与聚集索引的关系

联合索引

联合索引是指对表上的多个列进行索引。联合索引本质上也是使用B+树，不同的是他的键值大于等于2，按着列数一层一层的排序，先按第一列排序再按第二列排序以此类推。因此，联合索引有一个好处就是默认对后续的键进行排序，省去一次叶子节点的排序。多个键值的B+树：



覆盖索引

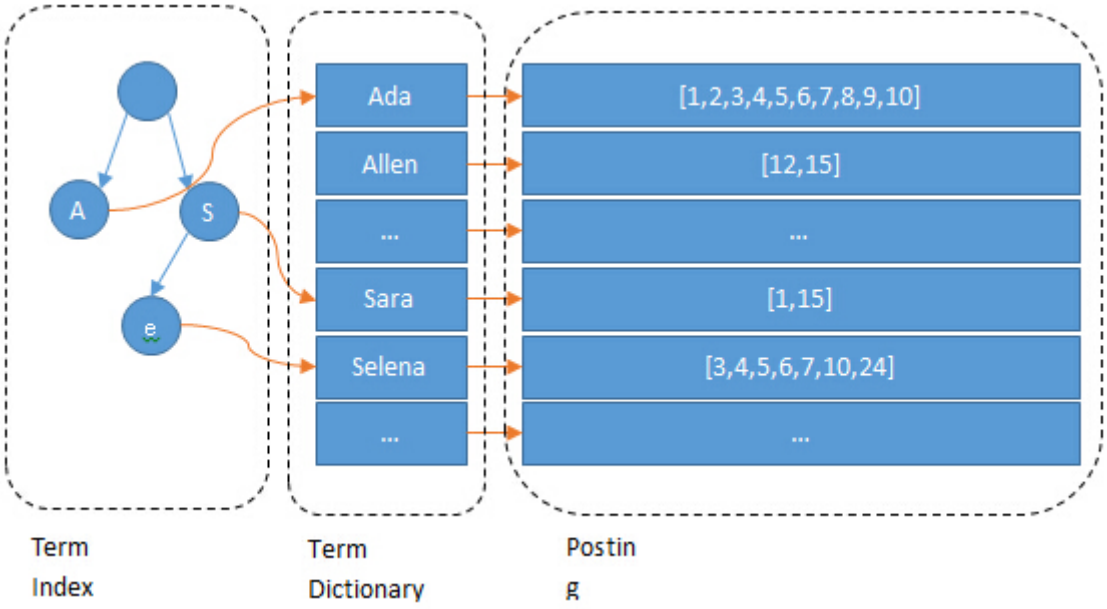
覆盖索引指的是将需要返回的数据全部放在索引列中，通过索引列就能返回所有数据，不需要拿到页号以后再去磁盘查询，可以减少大量IO操作。

倒排索引

倒排索引（英语：Inverted index），也常被称为反向索引、置入档案或反向档案，是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。有两种不同的反向索引形式：

> - 一条记录的水平反向索引（或者反向档案索引）包含每个引用单词的文档的列表。
> - 一个单词的水平反向索引（或者完全反向索引）又包含每个单词在一个文档中的位置。

后者的形式提供了更多的兼容性（比如短语搜索），但是需要更多的时间和空间来创建。倒排索引的瓶颈是word的查找，在elasticseach中是利用B树或者B+树进行存储，也可以hash+链表的存储方式，也可以java1.8以后的hashmap的实现方式。 倒排索引示意图如下：



自适应hash

InnoDB会监控对表上各索引页的查询，如果观察该数据被访问的频次符合规则，那么就建立哈希索引来加快数据访问的速度，这个哈希索引称之为$Adaptive Hash Index$(AHI),AHI是通过缓冲池的B+树页构建的，建立的速度很快，而且不对整颗树都建立哈希索引。(可以理解成热点的数据才会进入这个哈希表)。

> 这里有个限制需要注意，就是对同一个页的连续访问模式必须一致，比如已经对字段a和b添加了联合索引，但是交替使用一下两种查询是不会使用自适应hash的(虽然用到了联合索引)：
>
> - WHERE a = xxx
> - WHERE a = xxx and b = xxx

自适应hash的性能

> 读取、写入的速度可以提高2倍，辅助索引的连接操作性能可以提高5倍。 但是需要注意，这个优化是MySQL自优化，无法通过DBA来人为调整。

自适应hash的限制

> - 只能使用=号来作为条件。
> - 自适应哈希索引无法对order by进行优化
> - 不支持模糊查询

自适应hash的启动方式

> - innodb_adaptive_hash_index变量启用AHI; *在服务器启用时，禁用skip-innodb-adaptive-hash-index,还有一个参数为innodb_adaptive_hash_index_parts，这个5.7后InnoDB将自适应哈希索引进行了分区处理，每个区对应一个锁，如果大量地访问，那么可能会对性能产生影响(抢锁)，InnoDB将这个值设为8，最大可为512。

自适应hash的检测方式

> SHOW ENGINE INNODB STATUS 命令可以看自适应hash的使用情况

其他优化

索引分裂

1. 数据插入是根据Page Header中PAGE_LAST_INSERT、PAGE_DIRECTION、PAGE_N_DIRECTION三个字段来决定往哪个方向顺序插入的。
2. 索引分裂并不总是从页的中间记录进行分裂，因为这样可能导致空间的浪费，例如页中有数据记录:

   > 1 2 3 4 5 6 7 8 9

   若此时插入数据10，如果按着5分裂则结果是：

   > 1 2 3 4
   > 5 6 7 8 9 10

   这样会导致第一页中的剩余空间浪费；但是如果数据为

   > 1 5 7 8 11 14 15 17

   若此时插入的数据是21，按着8分裂的结果是

   > 1 5 7 8
   > 11 14 15 17 21

   分裂之后，两个页面的空间利用率是一样的；如果新的插入是随机在两个页面中挑选进行，那么下一次分裂的操作就会更晚触发；
3. 索引分裂步骤
   - 定位到待插入记录的前一个位置index，然后确认之后还有n条记录需要插入，则分裂点为index + n。具体例子如下：

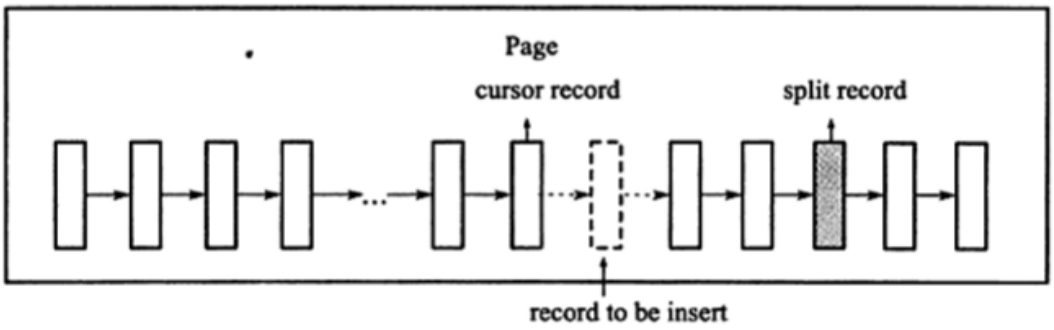


图 5-17　向右分裂的一种情况

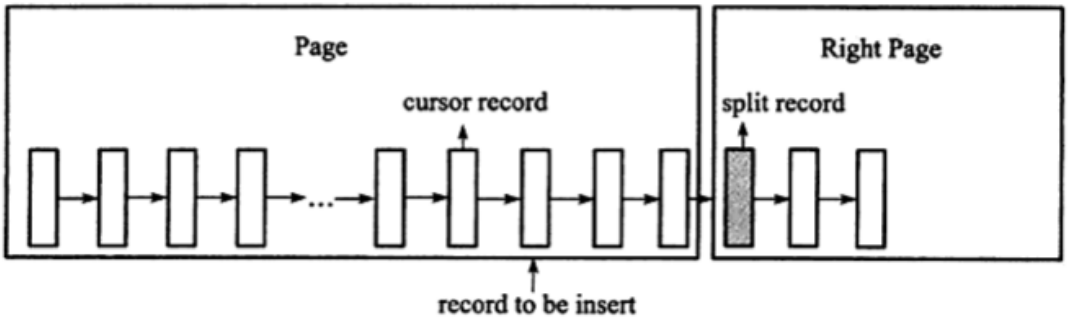


图 5-18　向右分裂后页中记录的情况

- 如果分裂点就是插入记录本身，则直接新开一个页面放入插入节点，示意图如下:

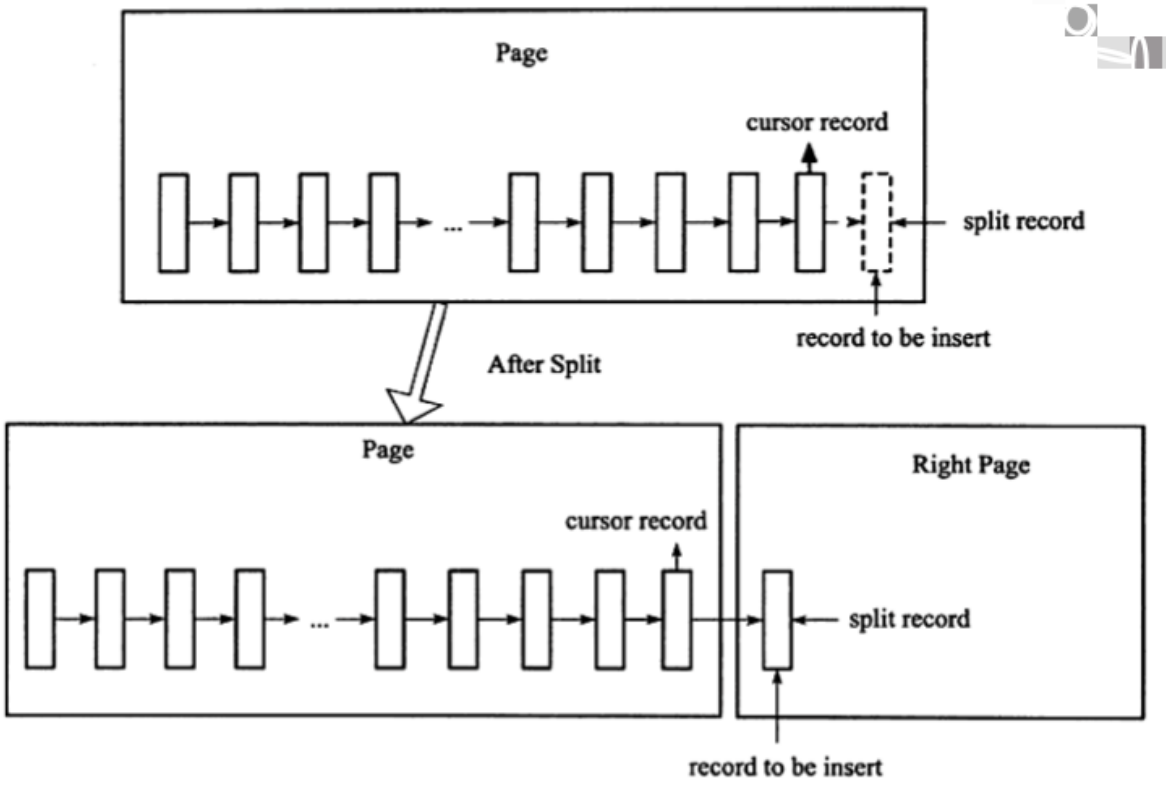


图 5-19　向右分裂的另一种情况

因此我们使用主键的时候,推荐使用递增主键，因为这样可以减少索引分裂，并且可以高效的利用索引页的空间。

Multi-Range Read 优化（MRR）

- 目的:减少磁盘随机访问，转化为较为顺序的数据访问。
- 适用范围:range, ref, eq_ref类型查询。
- MRR针对与范围查询和join查询操作的工作方式如下：

> 1. 将查询得到的辅助索引键值对放在缓存中，此时数据是按照辅助索引顺序排序的。
> 2. 将数据根据RowID进行排序
> 3. 根据RowID的顺序进行查找数据

Index Condition Pushdown 优化(ICP)

- 目的:减少SQL层对于记录的获取(fetch),从而提高数据库的整体性能。
- 适用范围:range, ref, eq_ref, ref_or_null。
- 工作方式:

> - 旧的工作方式:先在存储引擎层通过索引获取数据, 其次在SQL层利用Where语句进行过滤。
> - 新的工作方式:ICP优化则是在取出索引后,使用Where条件过滤,然后在去获取数据返回给SQL层,从而减少SQL层对于记录的获取。