B-Tree与B+Tree

来源：https://www.cnblogs.com/mysql-dba/p/6689597.html

在我们公司的DB规范中，明确规定：

```
1、建表语句必须明确指定
2、无特殊情况，主键必须
```

对于这项规定，很多研发小伙伴不理解。本文就来深入简出地分析MySQL索引设计背后的数据结构和算法，从而可以帮你释疑如下问题：

1、为什么innodb表需要主键？

2、为什么建议innodb表主键是单调递增？

3、为什么不建议innodb表主键设置过长？

一 | B-Tree基础知识

B-tree（多路搜索树，并不是二叉的）是一种常见的数据结构。使用B-tree结构可以显著减少定位记录时所经历的中间过程，从而加快存取速度。B通常认为是Balance的简称。这个数据结构一般用于数据库的索引，综合效率较高。目前很多数据库产品的索引都是基于B+tree结构。MySQL也采用B+tree,它是B-tree的一个变种，其实特性基本上差不多，理解了B-tree也就懂了B+tree。

1、一颗M阶B-Tree具有的特性【熟记于心】

1）根结点的孩子数>=2(前提是树高度大于1)

2）除根结点与叶子结点，其他结点的孩子数为[ceil(m/2),m]个。ceil函数表示上取整数

3）所有叶子结点都出现在同一层，叶子结点不存储数据。

4）各个结点包含n个关键字信息：(P0,K1,P1,K2,P2.....Kn,Pn)

其中：

4.1）Ki(i=1,2.....n)为关键字，且K(i-1)<Ki，即从小到大排序

4.2）关键字的个数n必须满足：[ceil(m/2)-1,m-1]

4.3）Pi指向子树，且指针P(i-1)所指向的子树结点中所有关键字均小于Ki。即：父结点中任何关键字的左孩子都小于它，右孩子大于它。

2、B-Tree插入操作

1）插入新元素，如果叶子结点空间足够，则插入其中，遵循从小到大排序；

2）如果该结点空间满了，进行分裂。将该结点中一半关键字分裂到新结点中，中间关键字上移到父结点中。

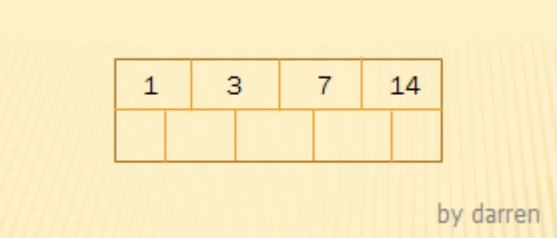
【举例】：如果单从上面特性及插入规则看得不明白，请结合以下分步骤图例：

将下面数字插入到一棵5阶B-Tree中：

[3,14,7,1,8,5,11,17,13,6,23,12,20,26,4,16,18,24,25,19]

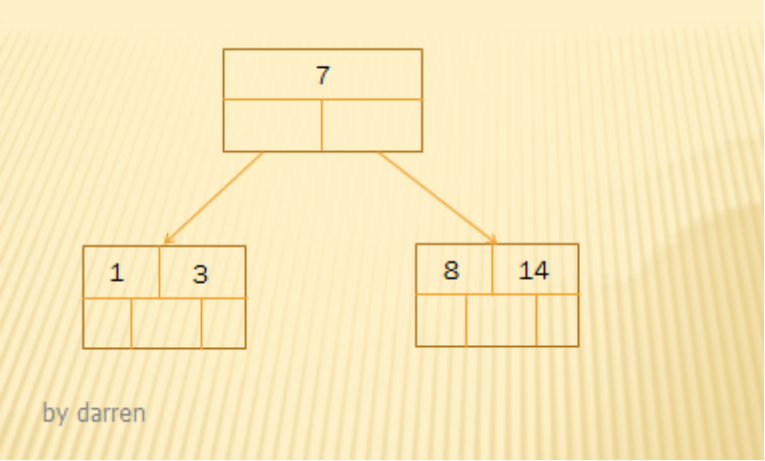
首先根据B-Tree特性知道，每个结点的关键字数量范围是： 2<=n<=4

【第一步】：插入3,14,7,1



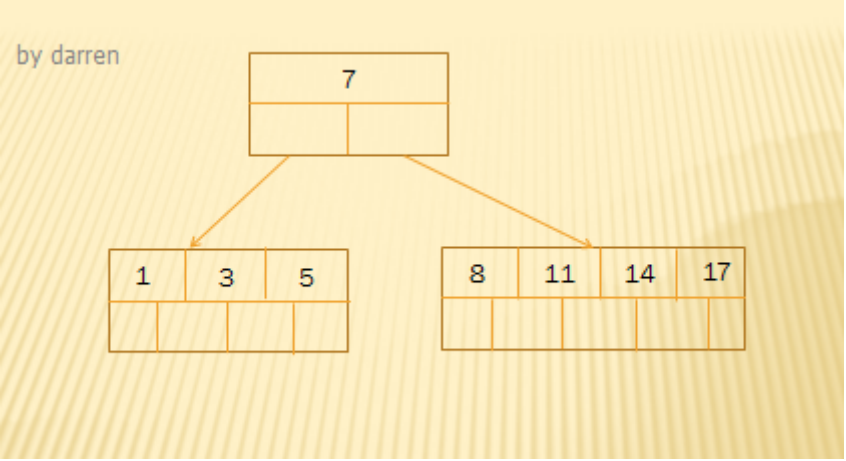
到这里，第一个结点中关键字数量刚好满了。

【第二步】：插入8



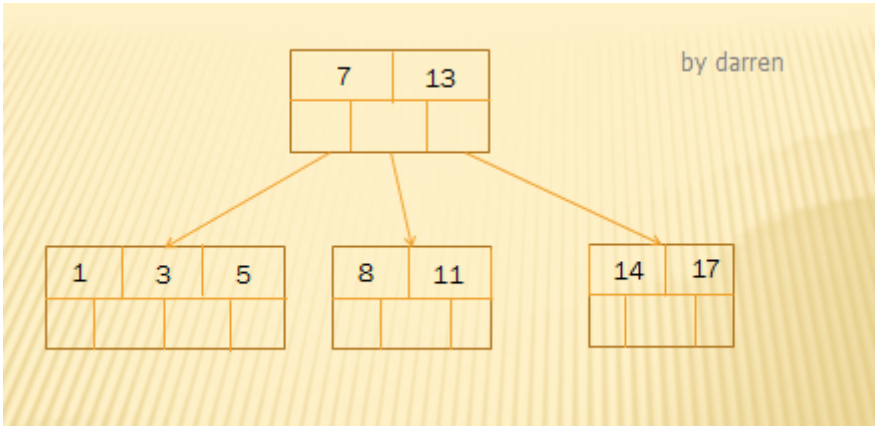
由于8是大于7的，故应该插入右子树，一个结点中最多存储4个关键字，按照插入规则，将中间关键字7上移形成父结点，其他按照50%分裂成两个结点，如上图。

【第三步】：插入5,11,17



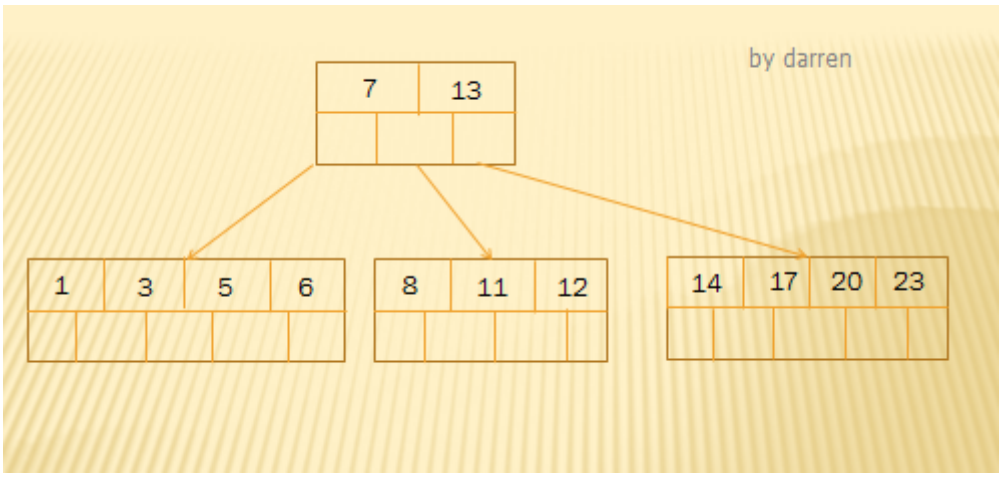
由于5小于7，插入左子树，11,17大于7,插入右子树。叶子结点没有满4个关键字，故可以直接插入5,11,17

【第四步】：插入13



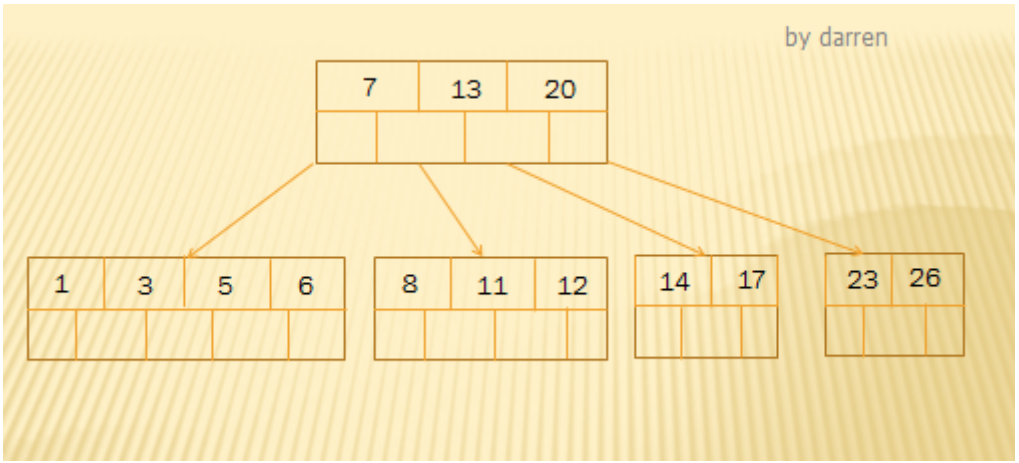
13大于7，应该插入右子树结点中，由于该结点中满4个关键字了，需要进行分裂。13刚好是中间关键字，上移到父结点中；其他按照50%分裂成两个结点。

【第五步】：插入6,23,12,20



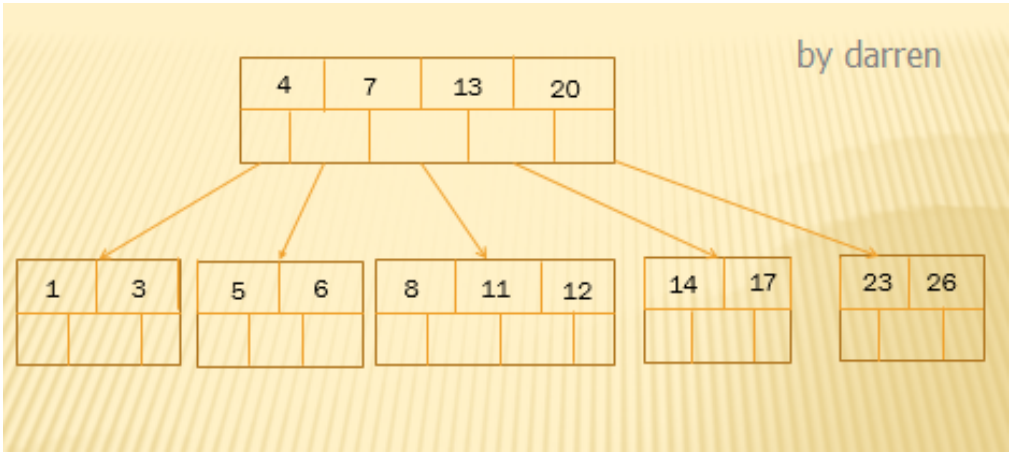
以上几个数字按照规则直接插入即可，无需分裂操作。

【第六步】：插入26



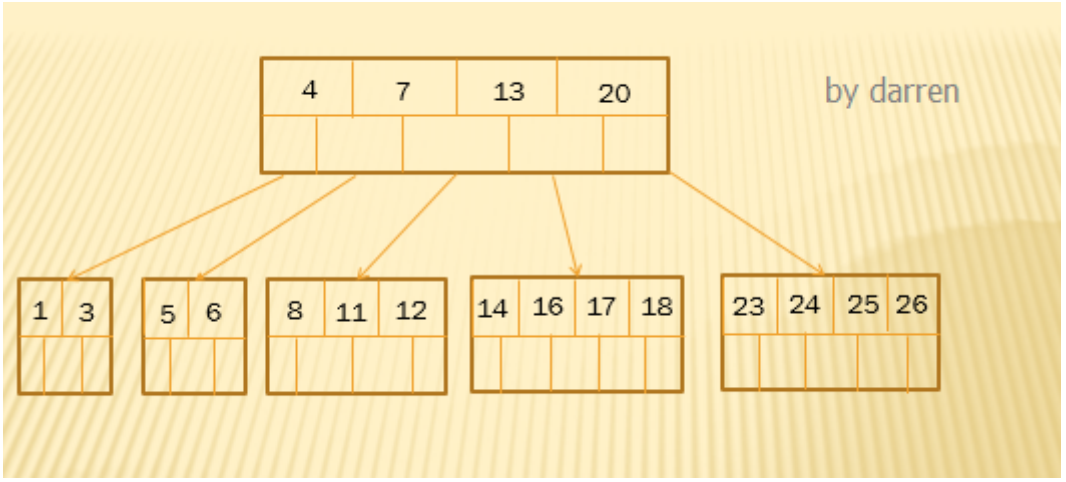
由于26大于13，应该插入13的右子树结点中，但是该结点已经满了，需要分裂，将中间20上移到父结点中，其他按照50%分裂成两个结点。

【第七步】：插入4



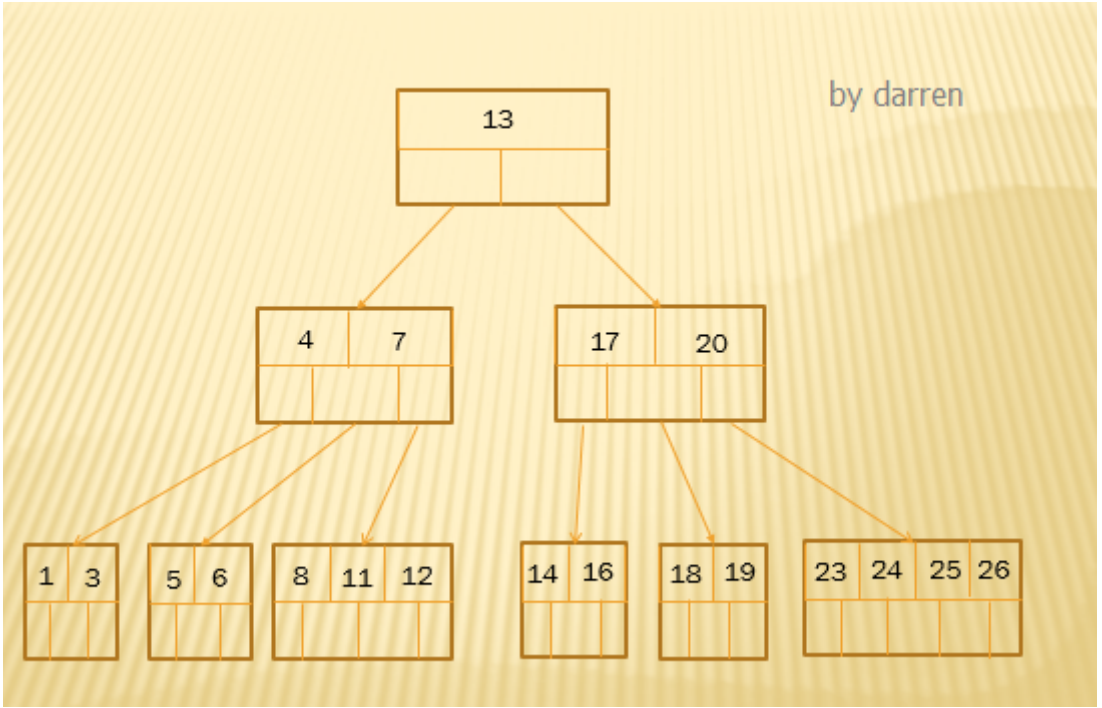
由于4小于7，应该插入7的左结点中，但该结点满了，需要进行分裂，将中间关键字4上移到父结点中，其他按照50%分裂成两个结点。

【第八步】：插入16，18，24，25



以上4个数字按大小直接插入到相应位置即可，无需分裂操作。

【第九步】：插入19



插入19，需要放到18的后面，但是由于该结点已满，需要分裂操作，将中间关键字17上移到父结点中，其它按照50%分裂成14，16以及18，19两个结点；

别以为到这就结束了，再看17被上移到父结点中，由于父结点已经满了，所以这时对父结点进行分裂，将中间关键字13上移形成新的父结点，其他按照50%分裂成4，7和17，20两个结点，到此，数据插入全部完成，形成了一棵B-Tree。

3、删除操作

删除操作稍稍复杂一些，这里就不举例展开了。大概思路如下：

1）查找B-tree中需删除的元素，如果该元素在B-tree中存在，则将该元素在其结点中进行删除。

2）删除该元素后，判断该元素是否有左右孩子结点，如果有，则上移孩子结点中的相近元素到父节点中（相近元素指的是：刚被删除元素的相邻后继元素，比如删除D，相近元素就是F等）

3）然后接着判断：如果结点中元素个数小于ceil(m/2)-1，首先找其相邻兄弟结点元素是否足够（结点中元素个数大于ceil(m/2)-1），如果足够，向父节点借一个元素，同时将借的元素的孩子结点中相邻后继元素上移到父结点中；

如果其相邻兄弟都不够，即借完之后其结点元素个数小于ceil(m/2)-1，那进行合并，具体是：将父结点中元素下移到要合并结点中（该元素一般是位于两个合并结点的中间元素），然后进行合并。

4）以上操作按顺序进行递归执行

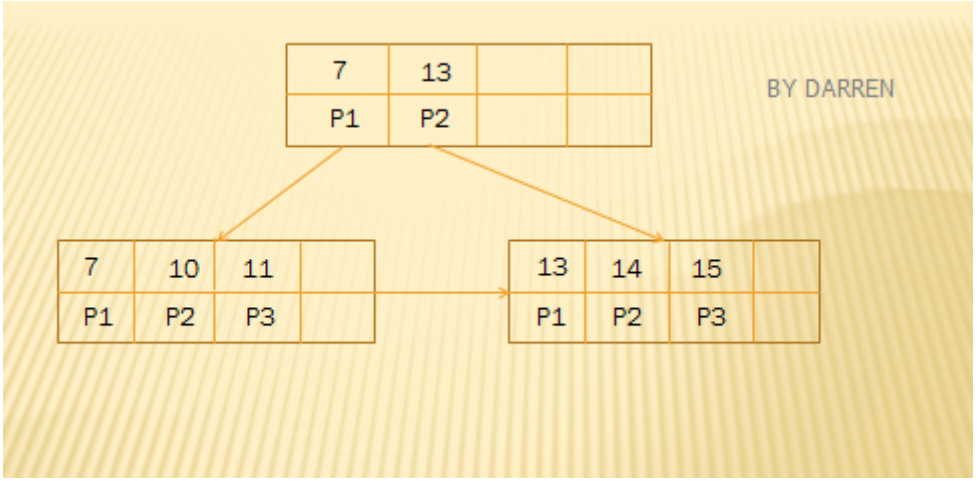
总之，对于索引文件，无论是插入还是删除B-Tree结点，不断地分裂和合并结点来维持B-Tree结构是非常昂贵的操作。

4、B+tree介绍：

MySQL索引采用B+Tree，它是应文件系统所需而产生的一种B-tree的变形树，他们的差异在于：

1 2 3 4	1） 非叶子结点的子树指 2） B+树父结点中的记录 3） 所有叶子结点通过一 4） 所有关键字都在叶子

如，下面是一棵典型的B+tree（假设每个结点最多有4个关键字）



其他特性与操作与B-tree基本相同。到此，B-tree和B+tree基础知识已经了解了，下面的内容都是基于以上的概念。

二 | MySQL索引实现

MySQL索引实现是在存储引擎端，不同存储引擎对索引实现方式是不同的，比如Innodb和MyISAM，下面我们重点介绍Innodb引擎索引的实现方式。

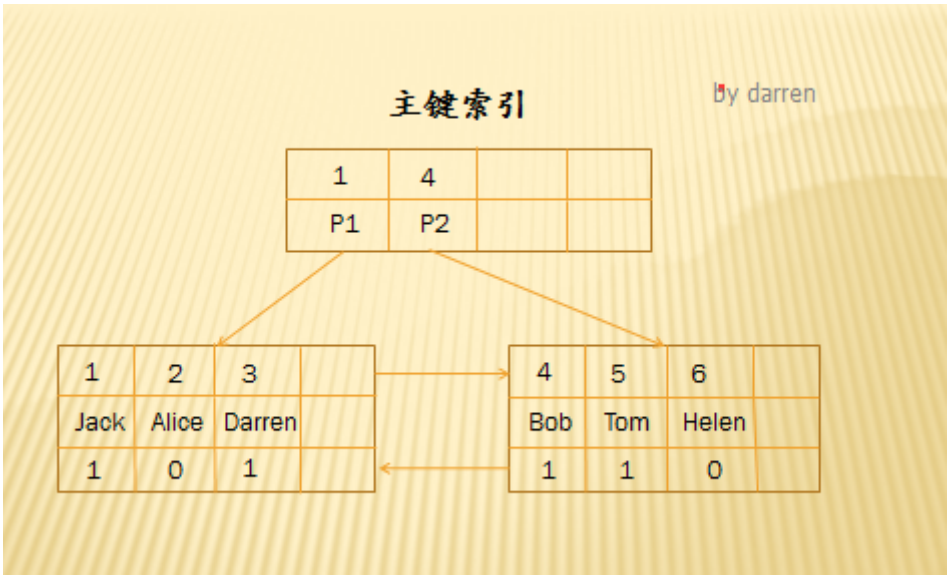
1、Innodb索引实现方式：

对于InnoDB表，数据文件ibd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。

举例说明，下面是students表，id是主键，name上有辅助索引，有6行数据记录。

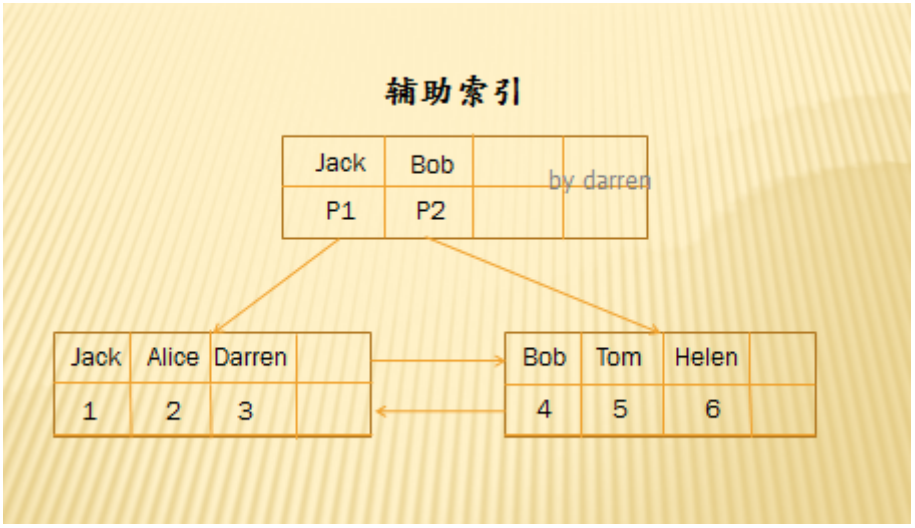
id	name	sex
1	Jack	1
2	Alice	0
3	Darren	1
4	Bob	1
5	Tom	1
6	Helen	0

假如在一棵5阶B+Tree(关键字范围[2,4]),它的主键索引组织结构如下：



上图是InnoDB主键索引的B+tree，叶节点包含了完整的数据记录，像这种索引叫做聚集索引。因为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会优先自动选择一个可以唯一标识数据记录的列作为主键，比如唯一索引列，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，长度为6个字节，类型为longint。

辅助索引结构：



对于secondary index，非叶子结点保存的是索引值，比如上面的name字段。叶子结点保存的不再是数据记录了，而是主键值。

从上面的B+Tree可以总结到：

MySQL聚集索引使得按主键的搜索非常高效的。

辅助索引需要搜索两遍索引：

第一：检索辅助索引获得主键值

第二：用主键值到主键索引中检索获得记录

到这里，再来分析本文开头提出的问题：

问题1、为什么Innodb表需要主键？

1）innodb表数据文件都是基于主键索引组织的，没有主键，mysql会想办法给我搞定，所以主键必须要有；

2）基于主键查询效率高；

3）其他类型索引都要引用主键索引；

问题3、为什么不建议Innodb表主键设置过长？

因为辅助索引都保存引用主键索引，过长的主键索引使辅助索引变得过大；

三 | InnoDB对B+Tree的改进

在上面的例子中：**将下面数字插入到一棵5阶B-Tree中：**

[3,14,7,1,8,5,11,17,13,6,23,12,20,26,4,16,18,24,25,19]

插入这些无序数据一共经历了6次分裂，对于磁盘索引文件而言，每次分裂都是很昂贵的操作；

如果将以上数据排好序，再次插入是不是效果会好，我试验了下，虽然每次都是插入到最右结点，涉及迁移数据量会少，但是分裂的次数依然挺多，需要7次分裂。

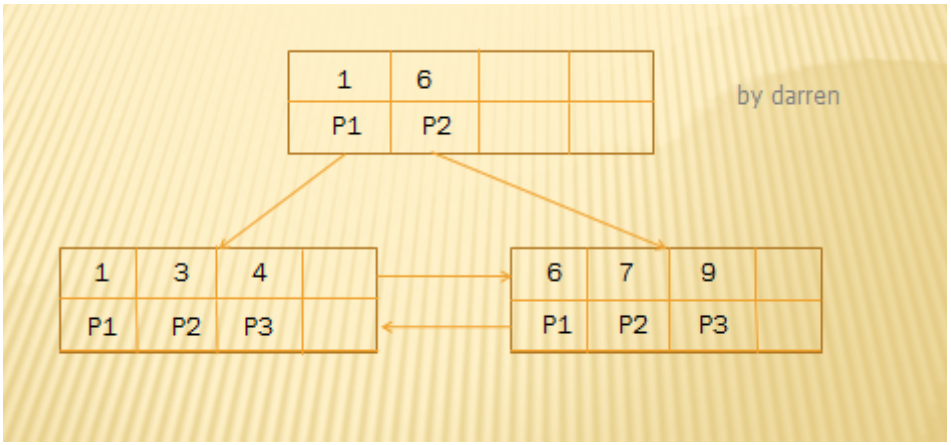
每次分裂都是按照50%进行，这样存在明显的缺点就是导致索引页面的空间利用率在50%左右；而且对于递增插入效率也不好，平均每两次插入，最右结点就得进行一次分裂。那Innodb是如何进行改进的呢？

Innodb其实只是针对递增/递减情况进行了改进优化，不再采用50%的分裂策略，而是使用下面的分裂策略：

```
1
2
3
```

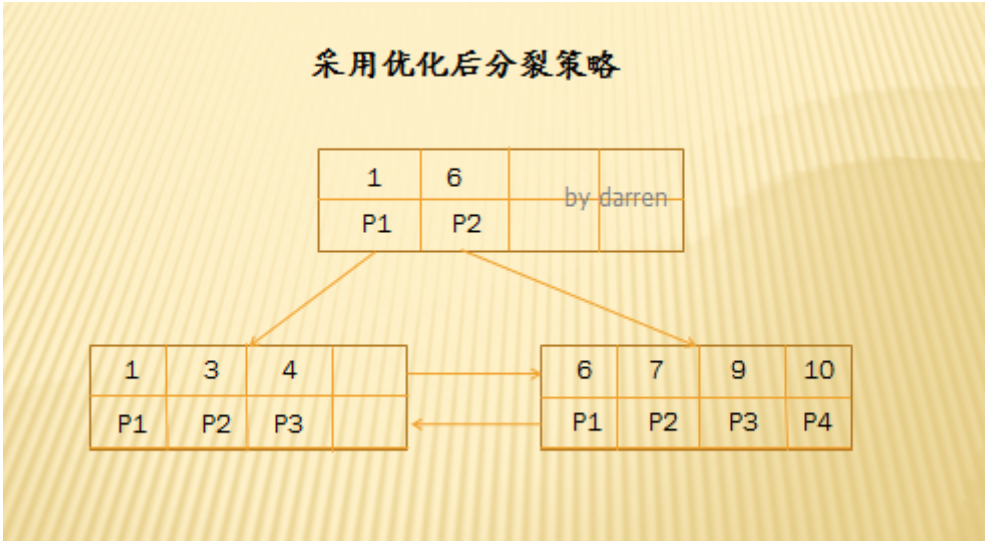
对于递增/递减索引插入
1、插入新元素，判断叶
2、如果叶子结点空间满
中；如果父结点空间满了

比如下面一棵5阶B+Tree：



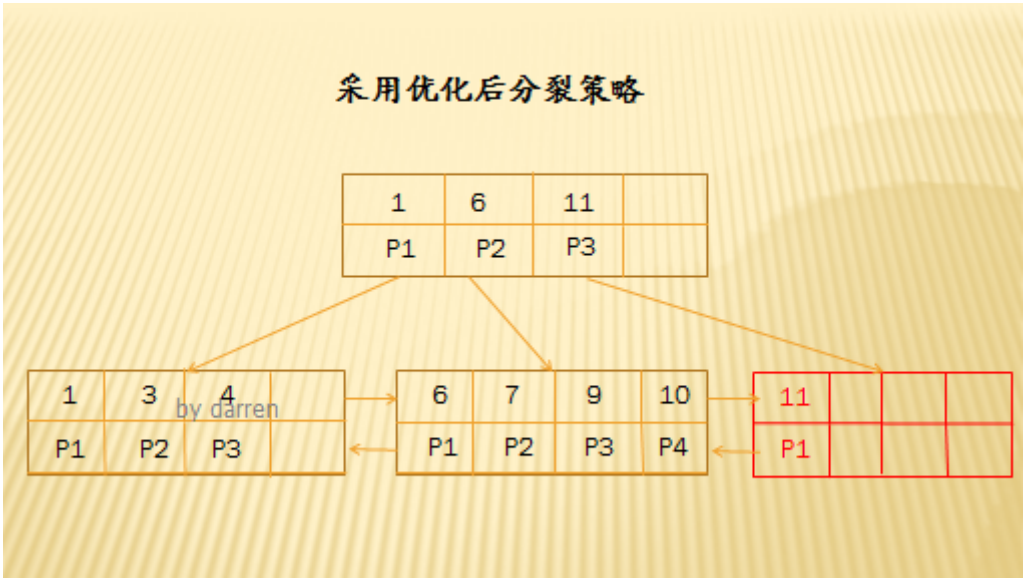
现在连续插入10,11,14,15,17，采用优化后分裂策略的分步图例如下：

【第一步】：插入10



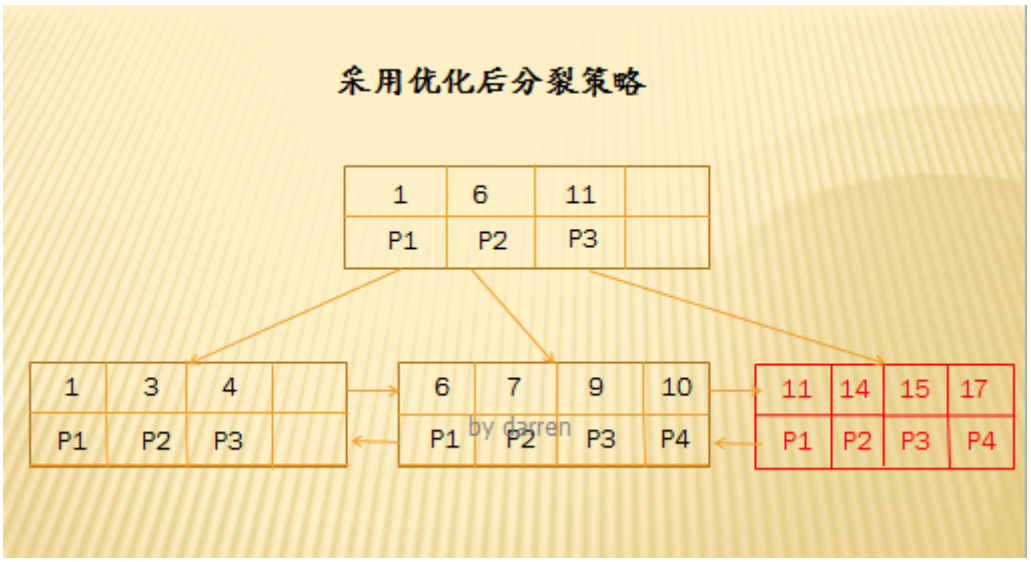
由于最右结点还有空间，直接插入即可。

【第二步】：插入11



插入11时，由于最右结点空间已满，如果使用50%分裂策略，则需要分裂操作了，但是使用优化后的分裂策略，当该结点空间已满，还要判断该结点的父结点是否满了，如果父结点还有空间，那么插入到父结点中，所以11插入到父结点中了，同时形成一个子结点。

【第二步】：插入14,15,17



优化后的分裂策略仅仅针对递增/递减情况，显著的减少了分裂次数并且大大提高了索引页面空间的利用率。

如果是随机插入，可能会引起更高代价的分裂概率。所以InnoDB存储引擎会为每个索引页维护一个上次插入的位置变量，以及上次插入是递增/递减的标识。InnoDB能够根据这些信息判断新插入数据是否满足递增/递减条件，若满足，则采用改进后的分裂策略；若不满足，则进行50%的分裂策略。

到此，我们可以回答本文开头提出的另一个问题了：

问题2：为什么建议InnoDB表主键是单调递增？

如果InnoDB表主键是单调递增的，可以使用改进后的B+tree分裂策略，显著减少B-Tree分裂次数和数据迁移，从而提高数据插入效率。

不仅如此，它还大大提高索引页空间利用率。