# 数据库索引数据结构B树和B+树

# 绪论

- **B树**是一种自平衡树数据结构，是一种组织和维护 外存文件 系统非常有效的数据结构 。数据库系统普遍采用B/B+Tree作为索引结构。

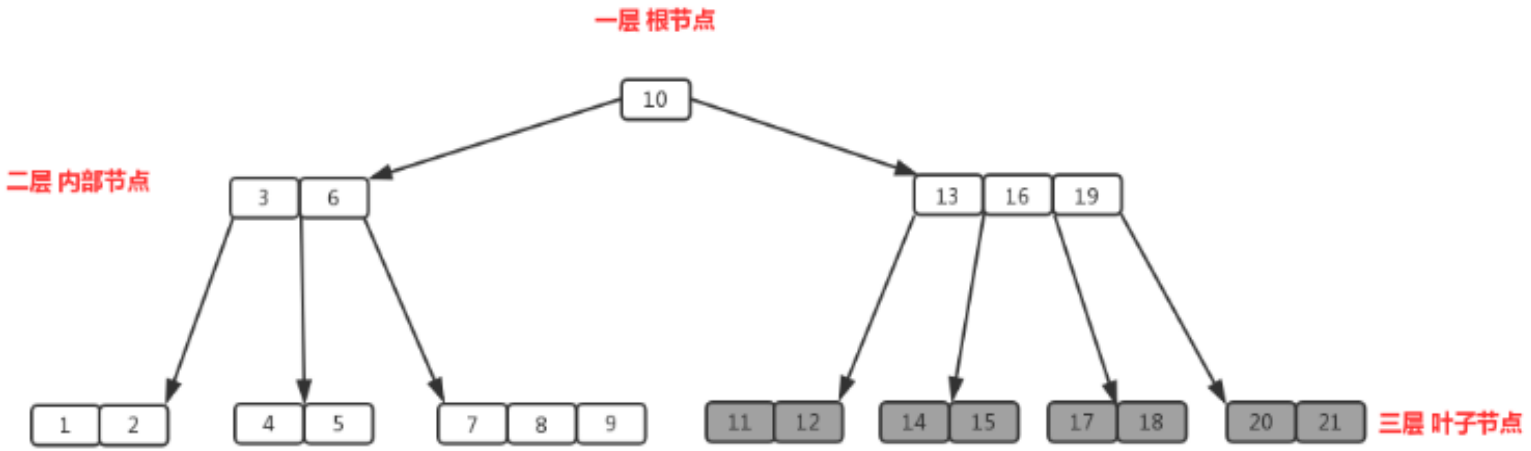
# B树定义

B树是一种平衡的多分树，通常m阶的B树，它必须满足如下条件：

- 每个节点最多只有m个子节点。
- 每个非叶子节点（除了根）具有至少[ m/2]子节点。
- 如果根不是叶节点，则根至少有两个子节点。
- 具有*k*个子节点的非叶节点包含*k* -1个键。

# B树的阶

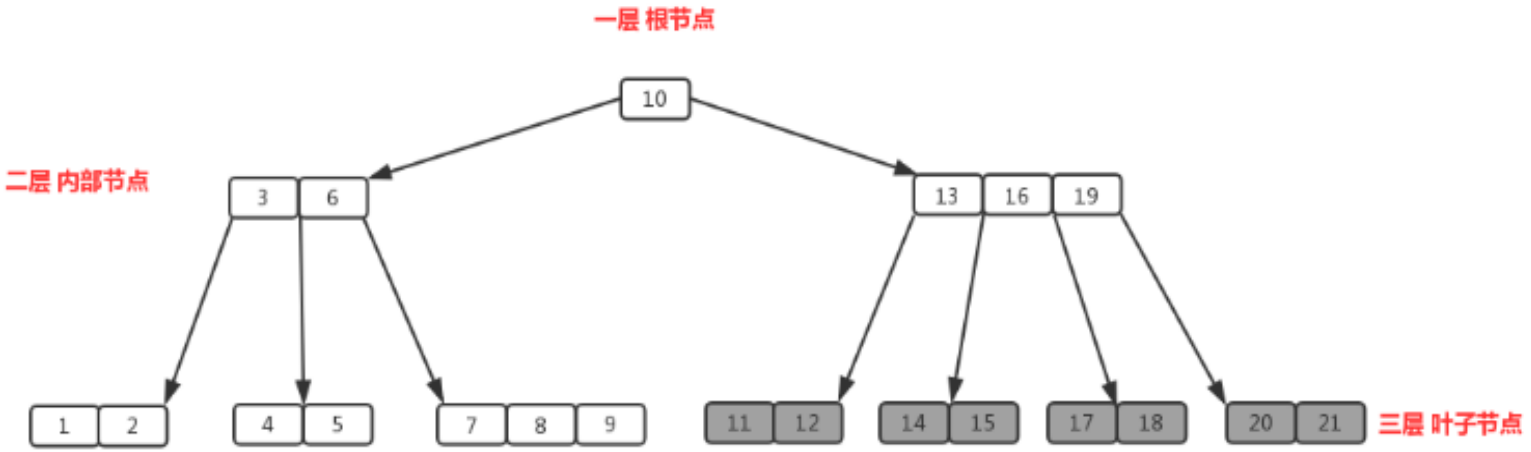
- B树中一个节点的子节点数目的最大值，用m表示




4阶B树

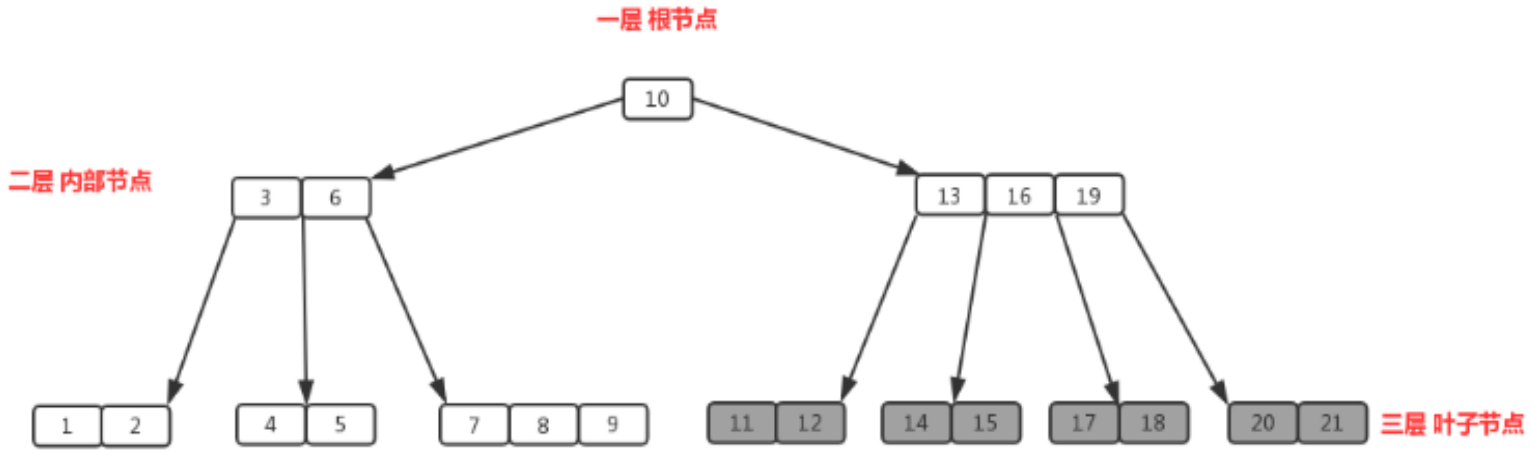
# 根节点

- 根节点拥有的子节点数量的上限和内部节点相同，如果根节点不是树中唯一节点，至少有俩个子节点。
- 在m阶B树中（根节点非树中唯一节点），有关系式2<= M <=m，M为子节点数量；包含的元素数量 1<= K <=m-1,K为元素数量。



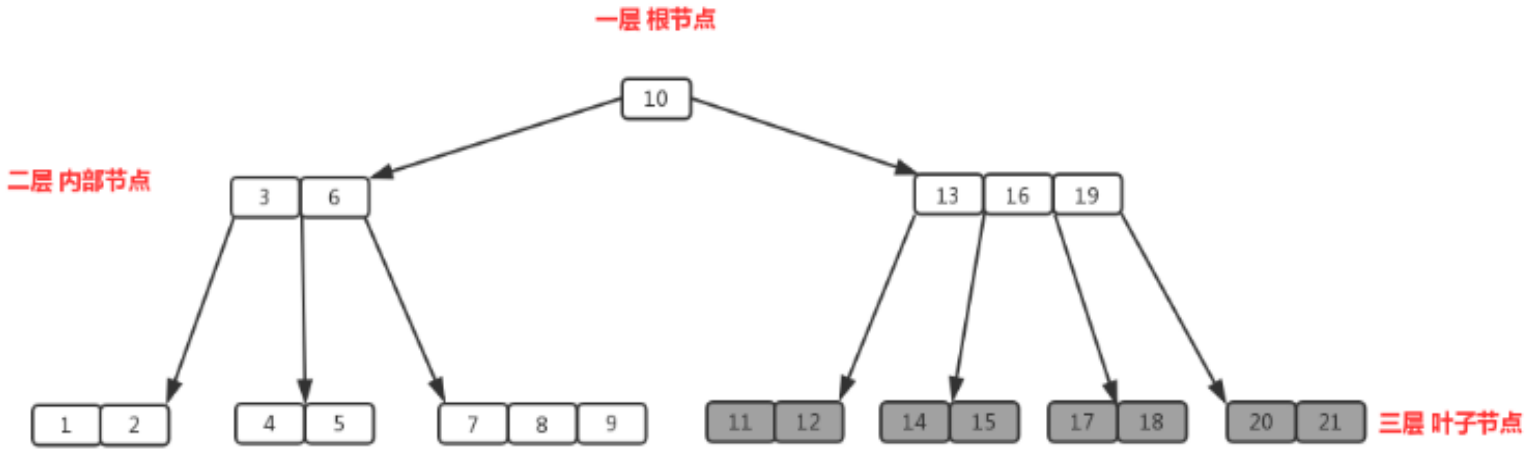
# 内部节点

- 内部节点是除叶子节点和根节点之外的所有节点，拥有父节点和子节点。
- 假定m阶B树的内部节点的子节点数量为M，则一定要符合（m/2）<= M <=m关系式；包含的元素数量（m/2）-1<= K <=m-1,K为元素数量。m/2向上取整。



# 叶子节点

- 在m阶B树中叶子节点的元素符合（m/2）-1<= K <=m-1。



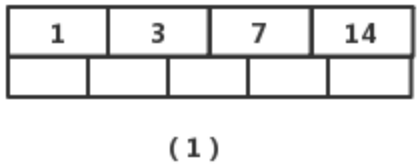
# 插入

- 针对m阶高度h的B树，插入一个元素时，首先在B树中是否存在，如果不存在，在叶子结点中插入该新的元素。
- 若该节点元素个数小于m-1，直接插入；
- 若该节点元素个数等于m-1，引起节点分裂；以该节点中间元素为分界，取中间元素（偶数个数，中间两个随机选取）插入到父节点中；
- 重复上面动作，直到所有节点符合B树的规则；最坏的情况一直分裂到根节点，生成新的根节点，高度增加1。

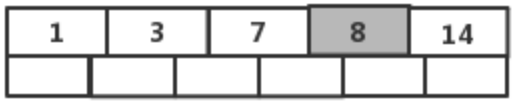
# 插入

5阶B树关键点:

- 2<=根节点子节点个数<=5
- 3<=内节点子节点个数<=5
- 1<=根节点元素个数<=4
- 2<=非根节点元素个数<=4

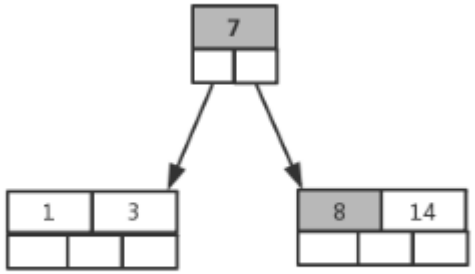



(1)




(2)

插入8

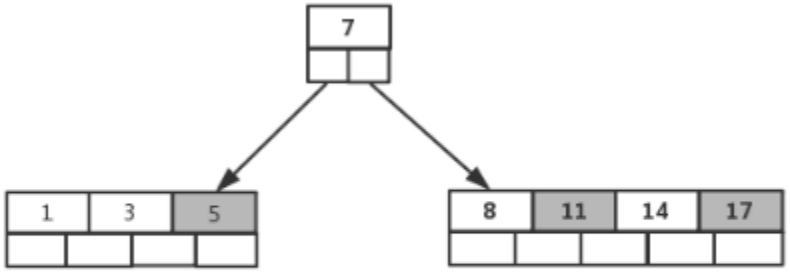



(3)

# 插入

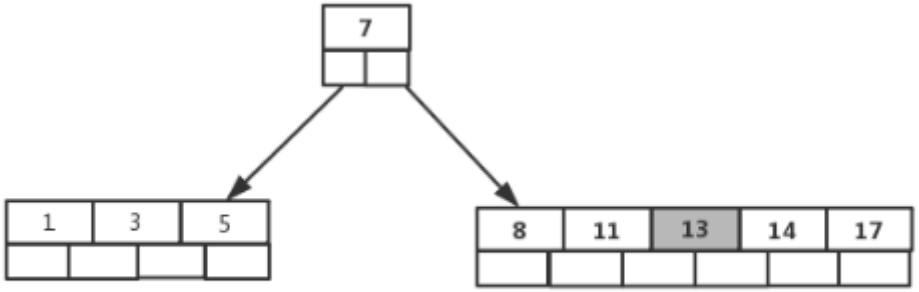
5阶B树关键点:

- 2<=根节点子节点个数<=5
- 3<=内节点子节点个数<=5
- 1<=根节点元素个数<=4
- 2<=非根节点元素个数<=4



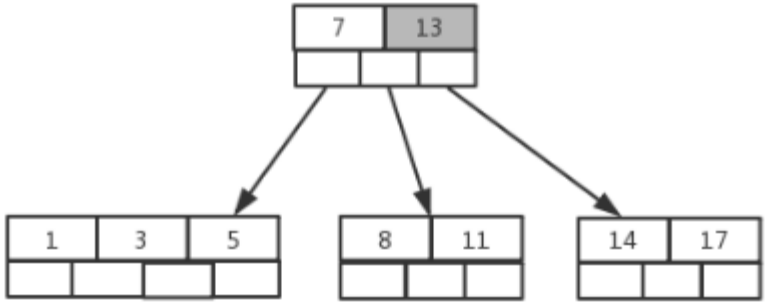

(4)

插入元素5，11，17




(5)

插入元素13

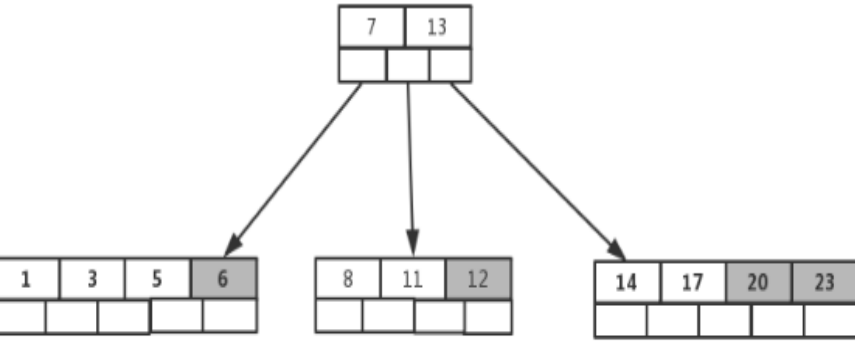



(6)

# 插入

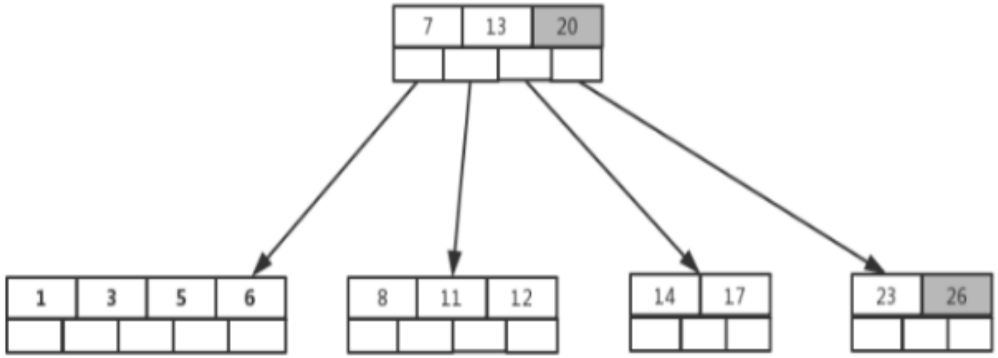
5阶B树关键点:

- 2<=根节点子节点个数<=5
- 3<=内节点子节点个数<=5
- 1<=根节点元素个数<=4
- 2<=非根节点元素个数<=4



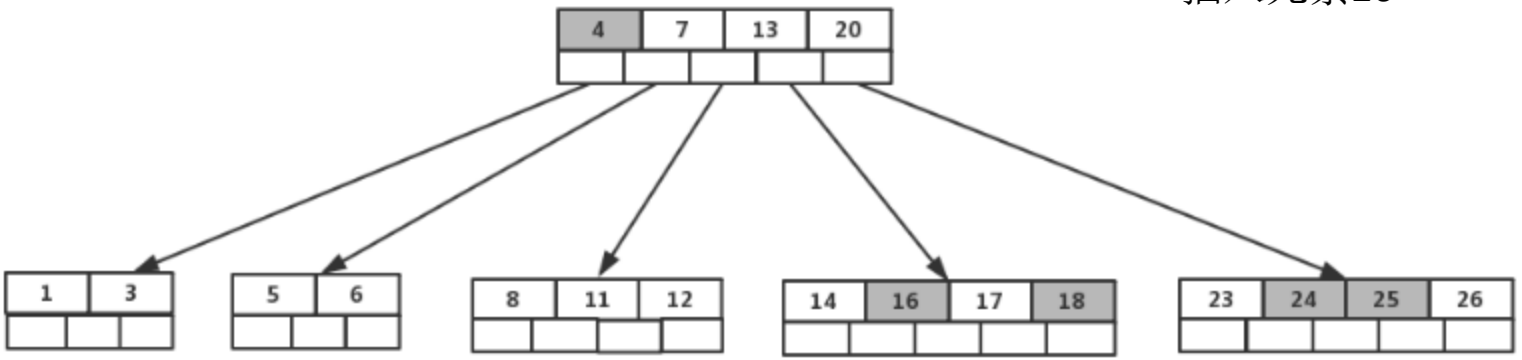

(7)

插入元素6，12，20，23




(8)

插入元素26



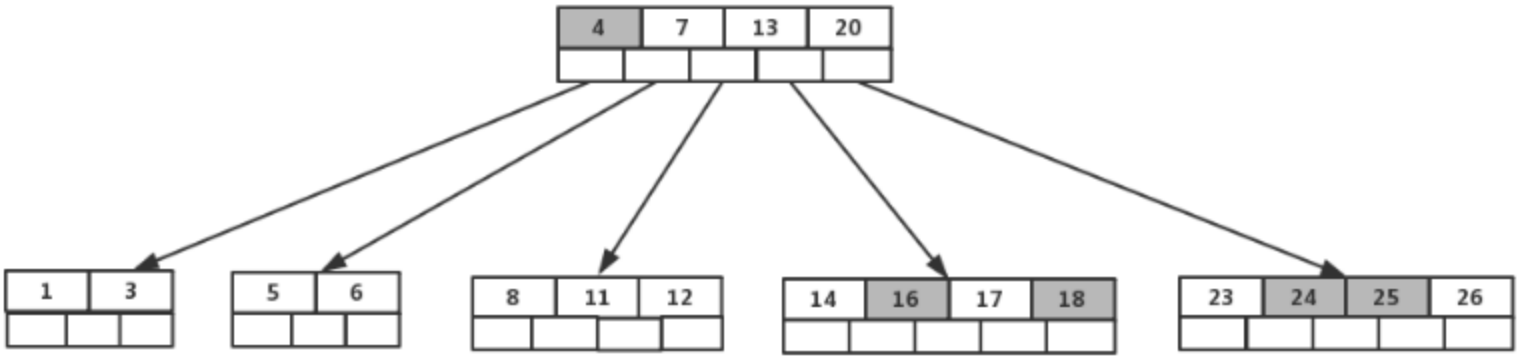

插入元素4，16，18，25

(9)

# 插入

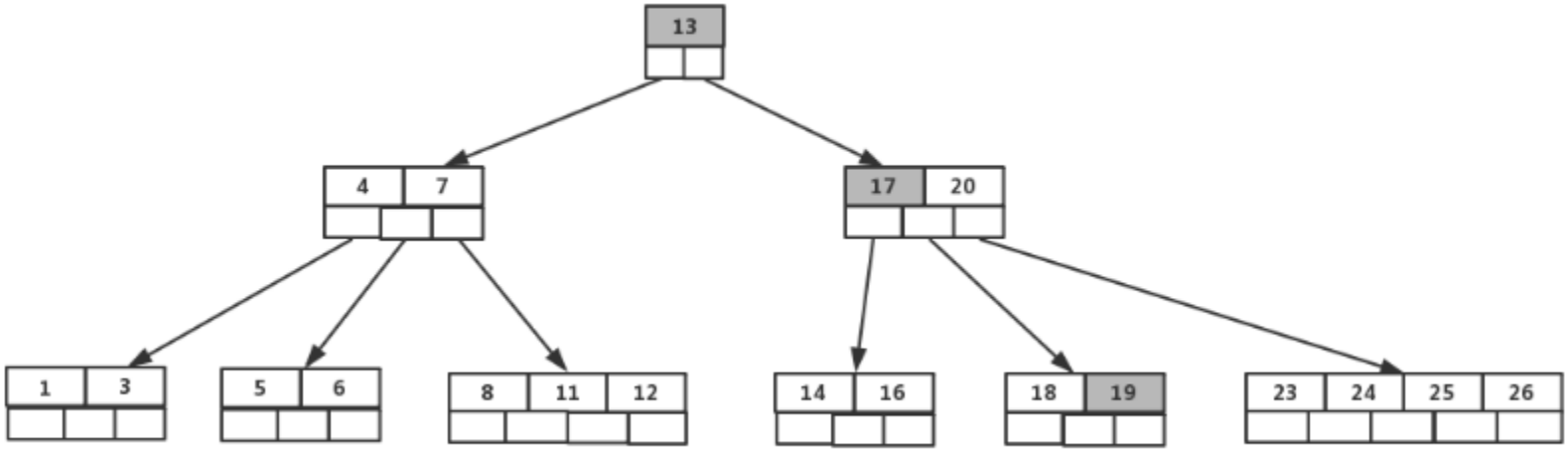
5阶B树关键点:

- 2<=根节点子节点个数<=5　　3<=内节点子节点个数<=5
- 1<=根节点元素个数<=4　　2<=非根节点元素个数<=4




(9)

插入元素19



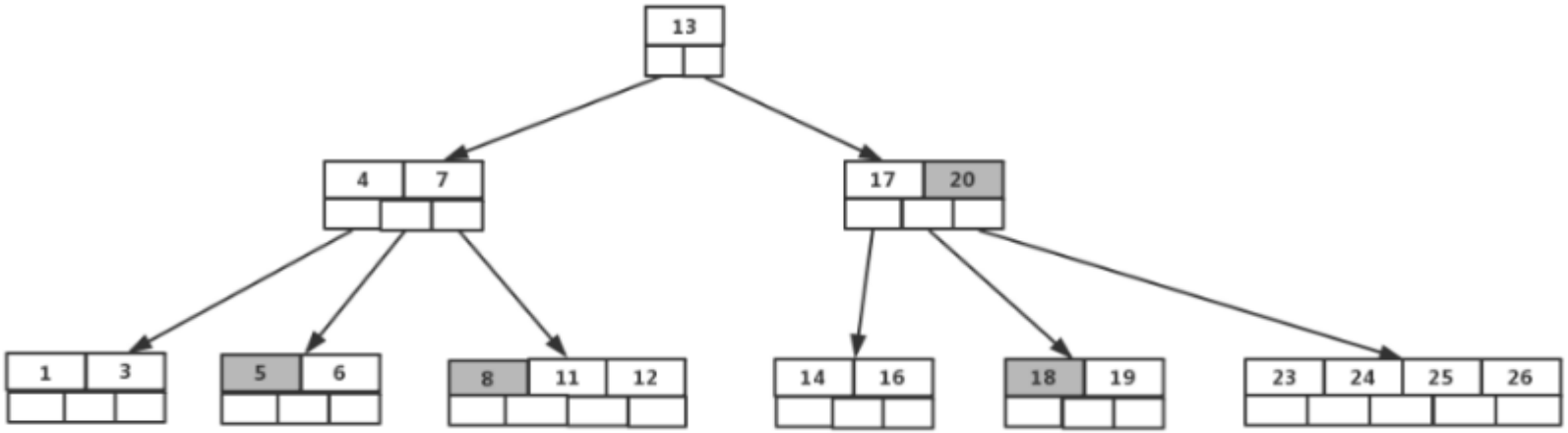

(10)

# 删除

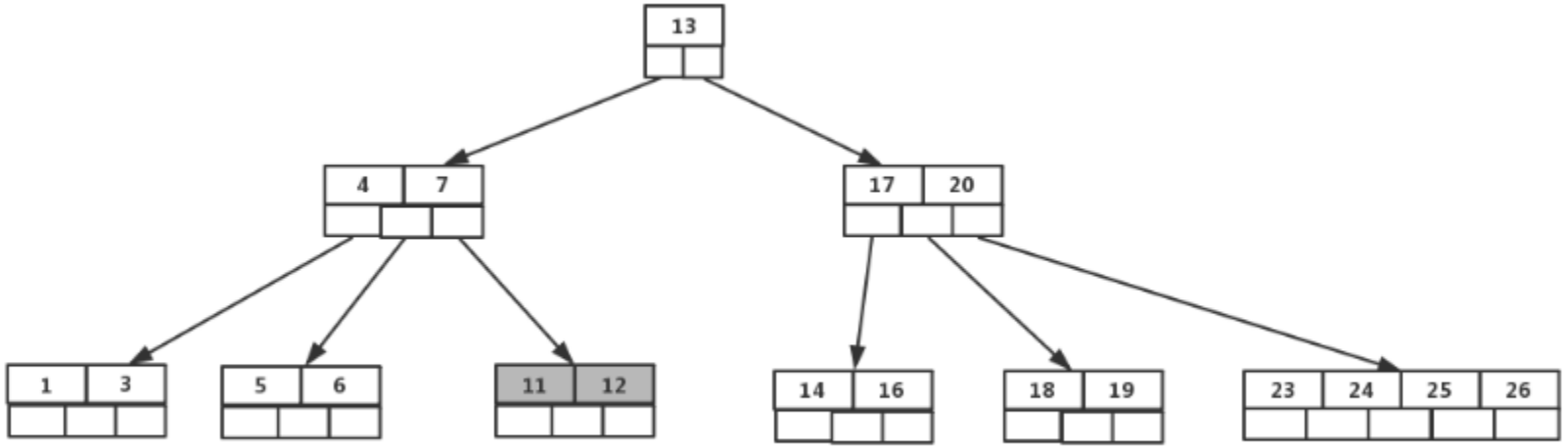
5阶B树关键点:

- 元素个数小于 2（m/2 -1）就合并，大于4（m-1）就分裂

如图依次删除依次删除【8】,【20】,【18】,【5】




（1）

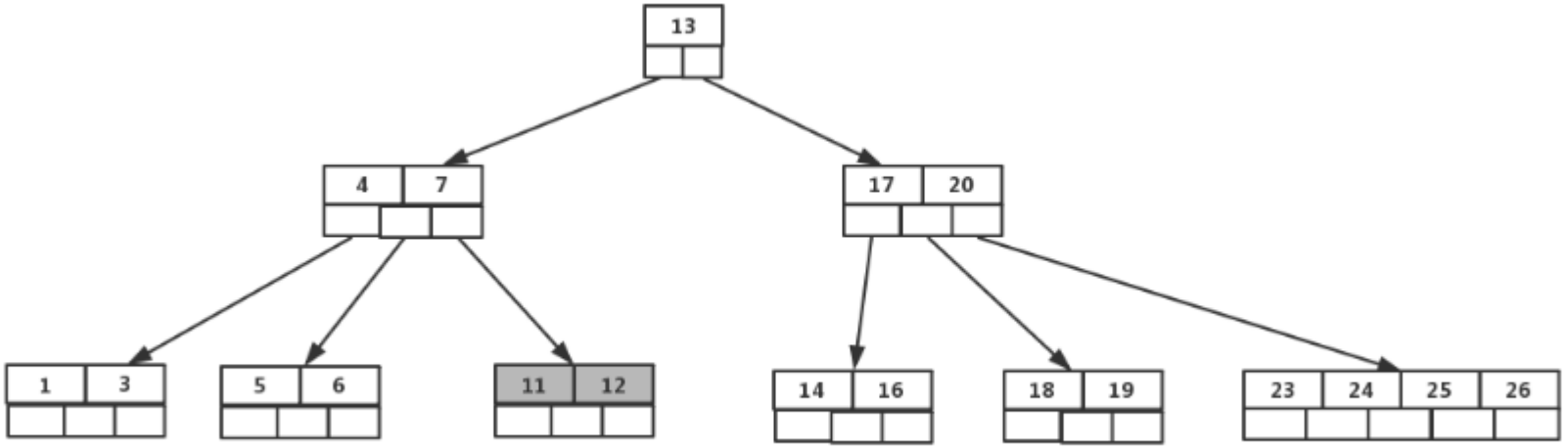



（2）

# 删除

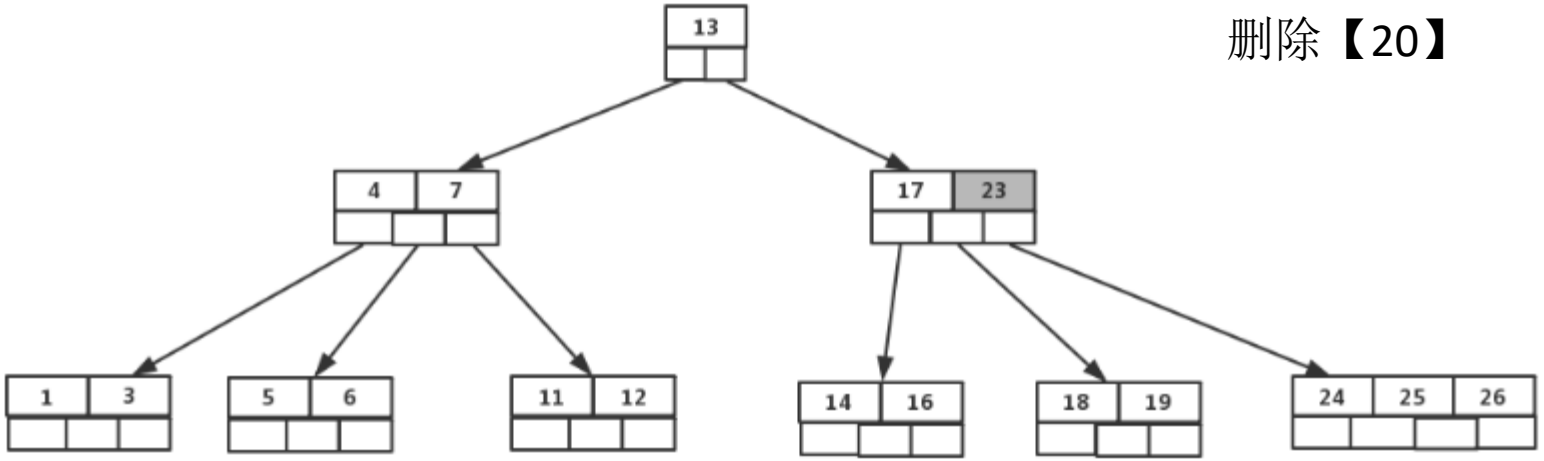
5阶B树关键点:

- 元素个数小于 2（m/2 -1）就合并，大于4（m-1）就分裂




（2）

删除【20】

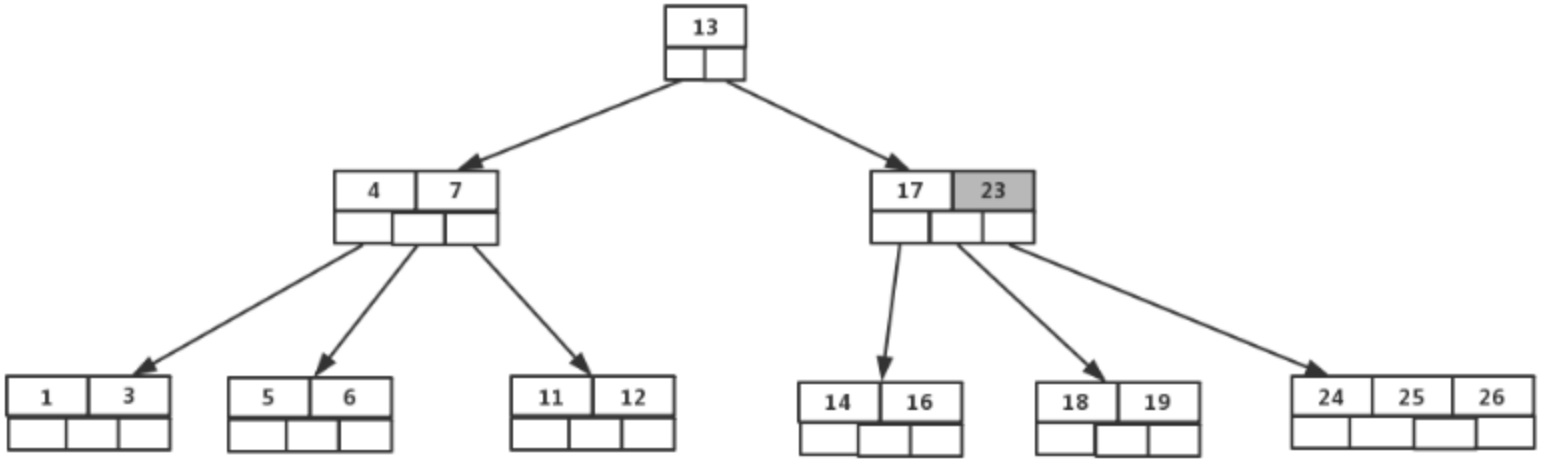



（3）

# 删除

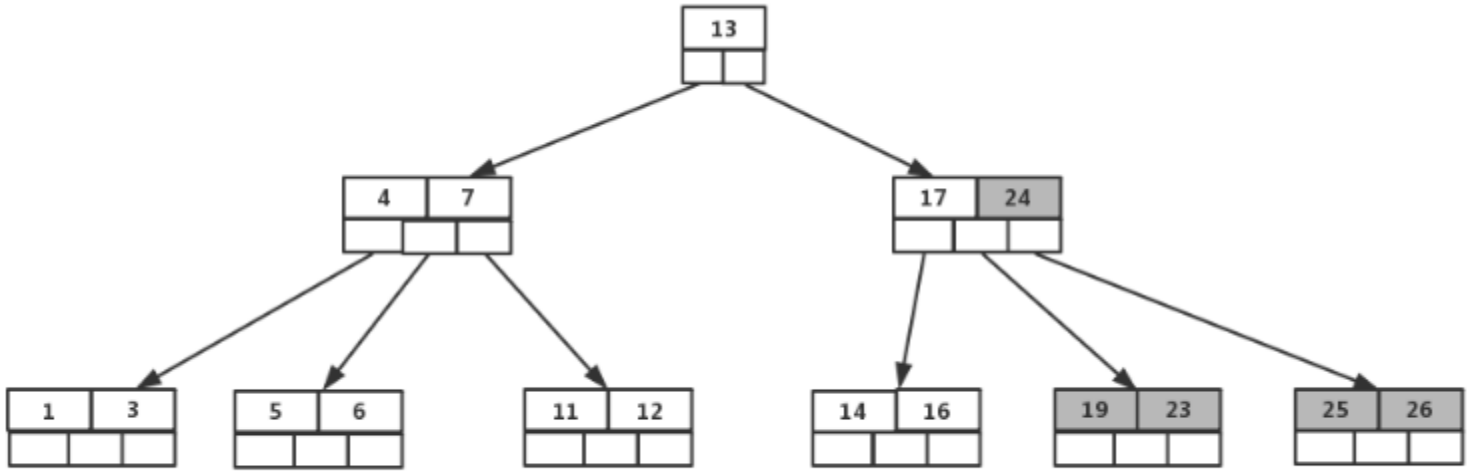
5阶B树关键点:

- 元素个数小于 2（m/2 -1）就合并，大于4（m-1）就分裂




（3）

删除【18】

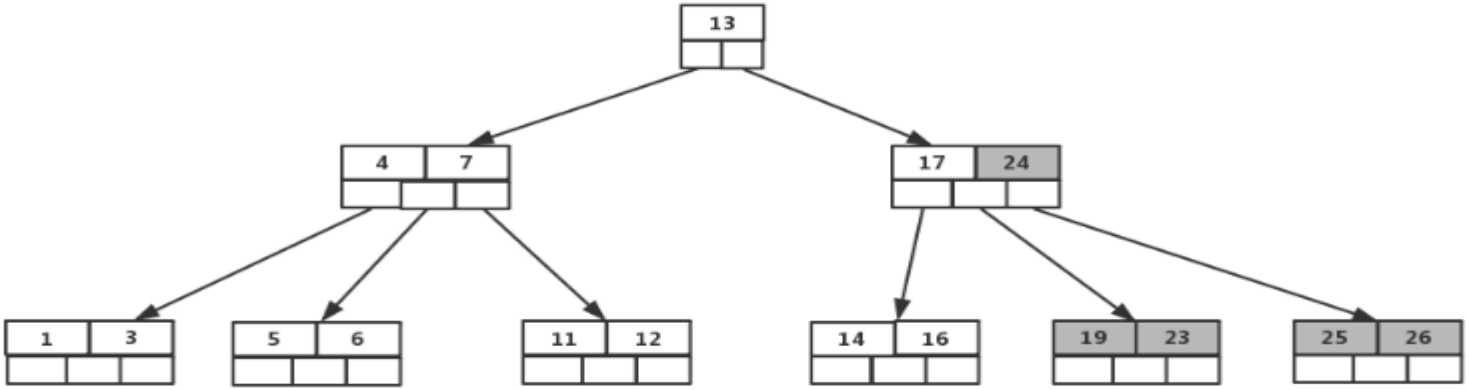



（4）

# 删除

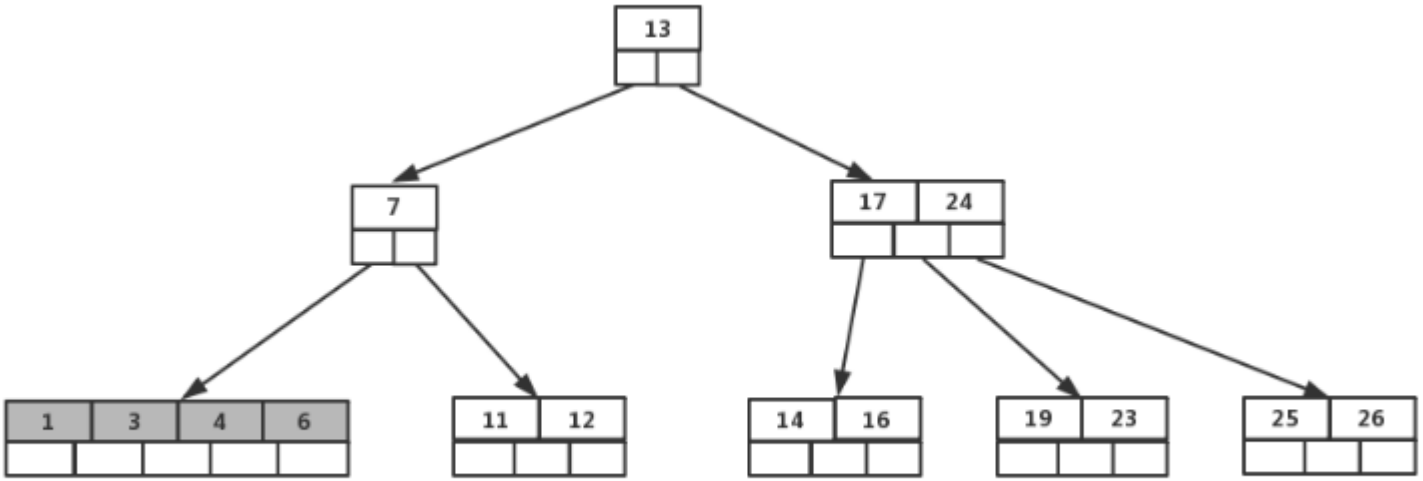
5阶B树关键点:

- 元素个数小于 2（m/2 -1）就合并，大于4（m-1）就分裂




（4）

删除【5】

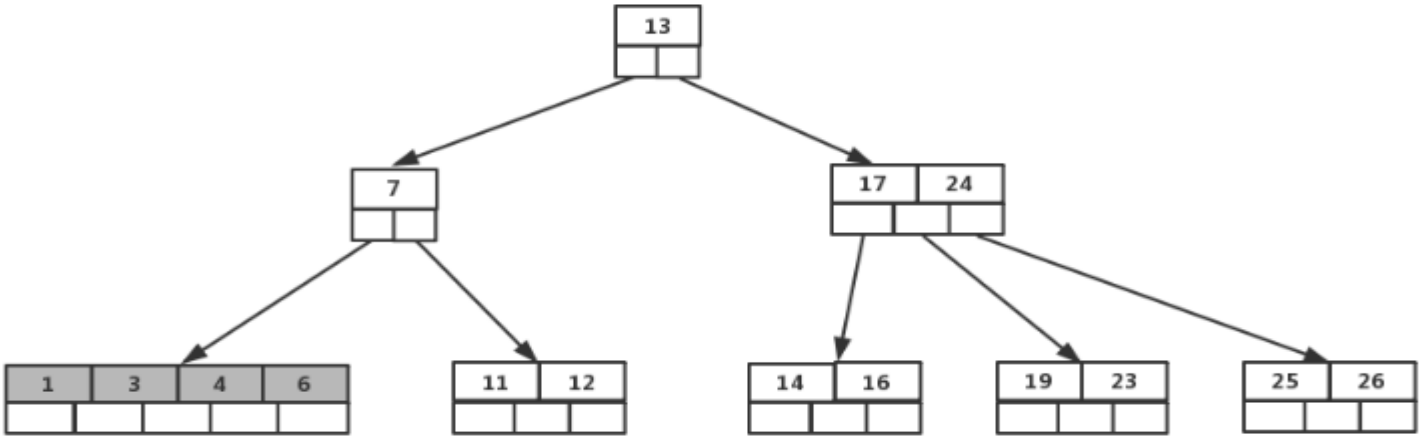



（5）

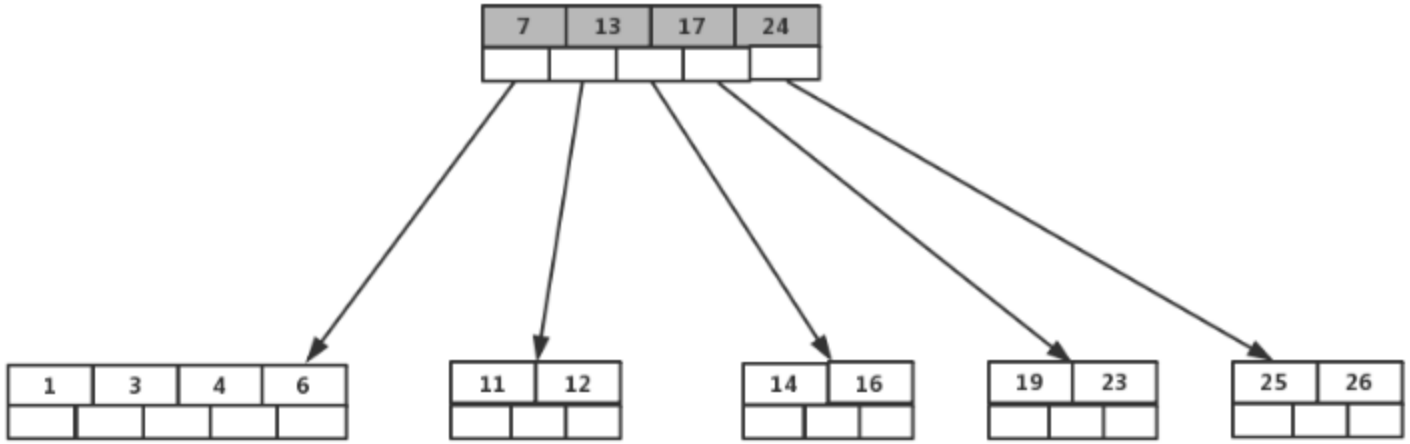
# 删除

5阶B树关键点:

- 元素个数小于 2（m/2 -1）就合并，大于4（m-1）就分裂

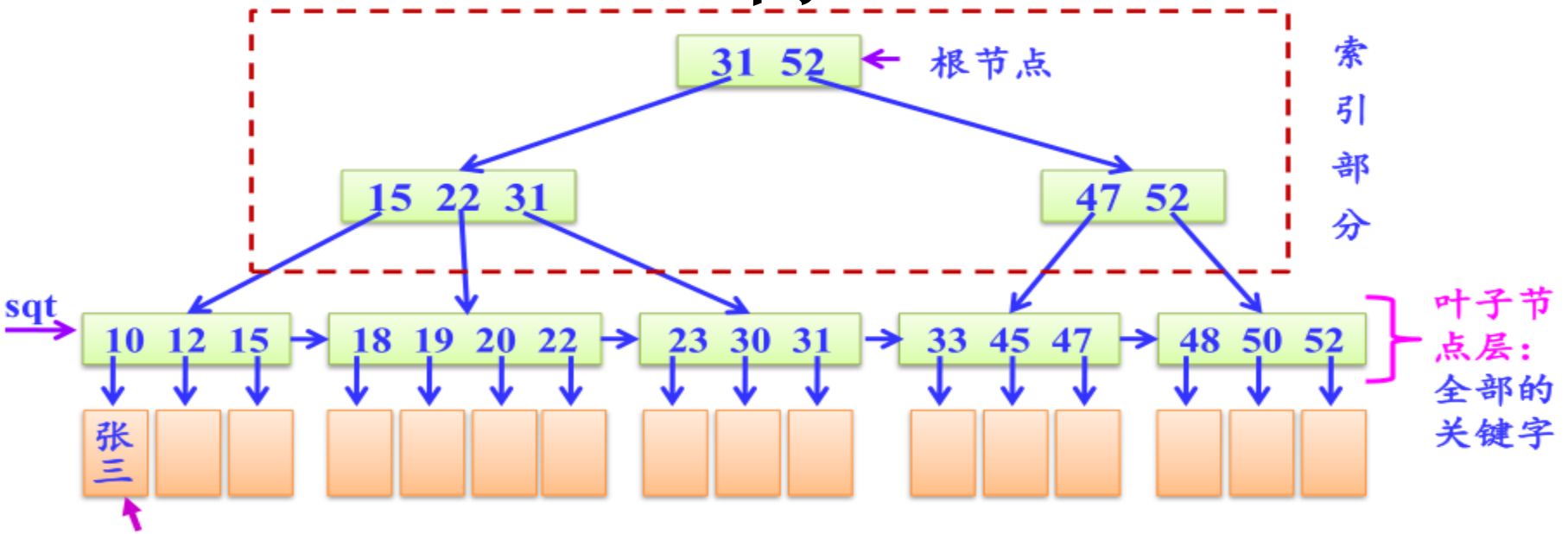



（5）




（6）

# B+树




B+树的定义：一棵$m$阶B+树满足下列要求：

❶ 每个分支节点至多有$m$棵子树（这里$m$=4）。

❷ 根节点或者没有子树，或者至少有两棵子树。

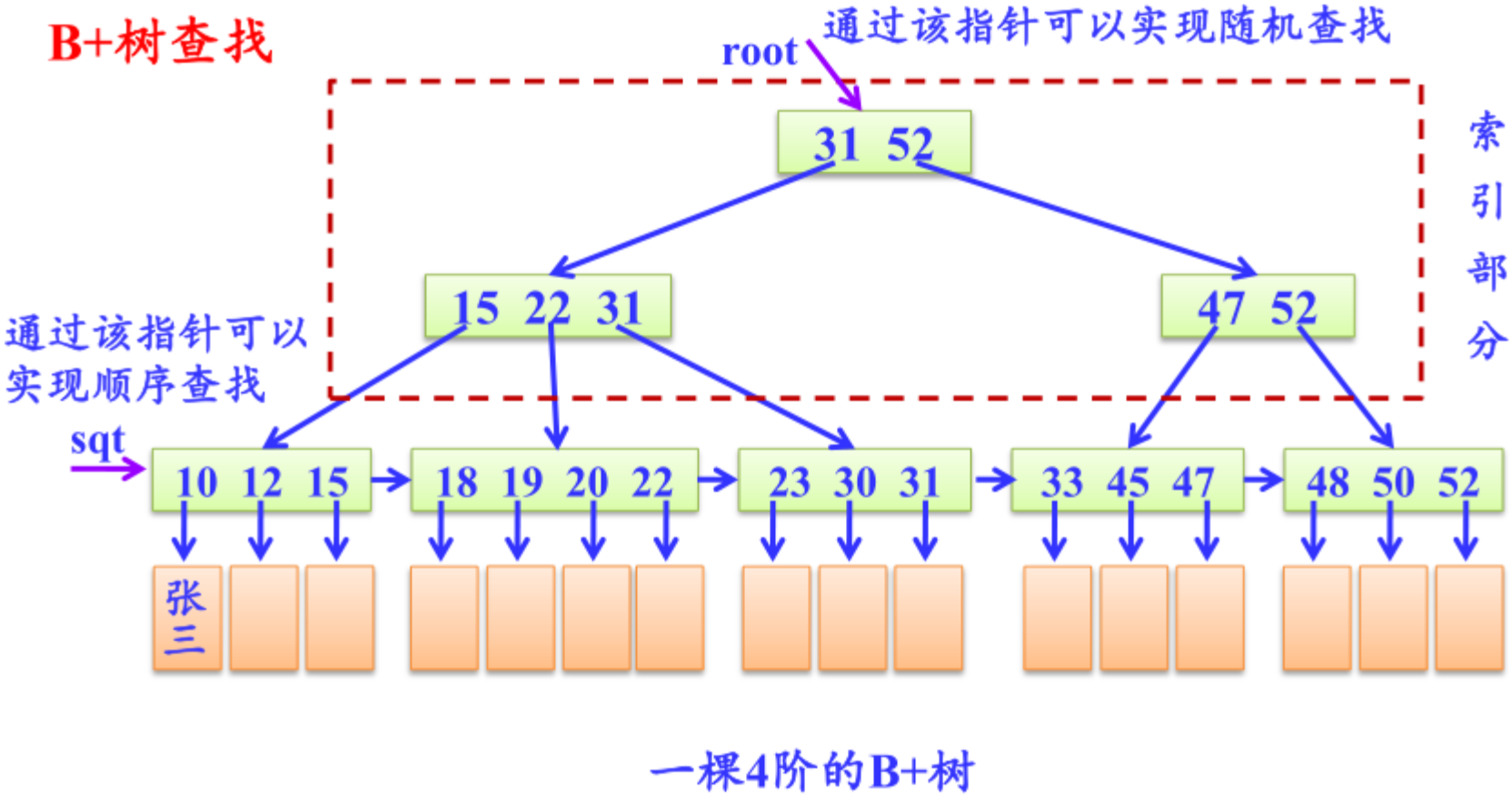
❸ 除根节点外，其他每个分支节点至少有$\lceil m/2 \rceil$棵子树。

❹ 有$n$棵子树的节点恰好有$n$个关键字。

❺ 所有叶子节点包含全部关键字及指向相应记录的指针，而且叶子节点按关键字大小顺序链接。并将所有叶子节点链接起来。

❻ 所有分支节点（可看成是索引的索引）中仅包含它的各个子节点（即下级索引的索引块）中最大关键字及指向子节点的指针。

# B+树


B+树查找
root
通过该指针可以实现随机查找
索引部分
31 52
15 22 31
47 52
通过该指针可以
实现顺序查找
sqt
10 12 15
18 19 20 22
23 30 31
33 45 47
48 50 52
张三

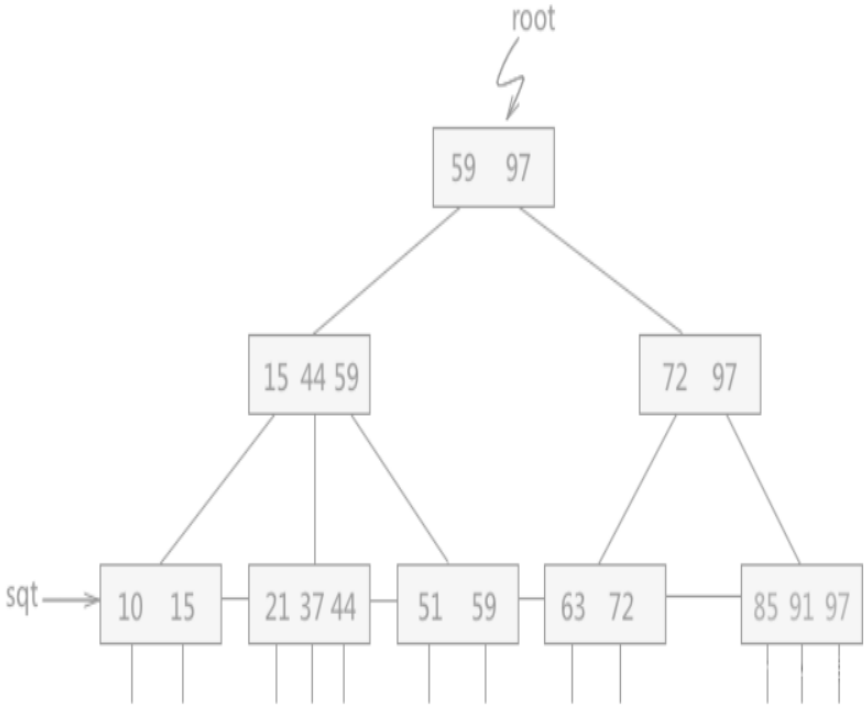
一棵4阶的B+树

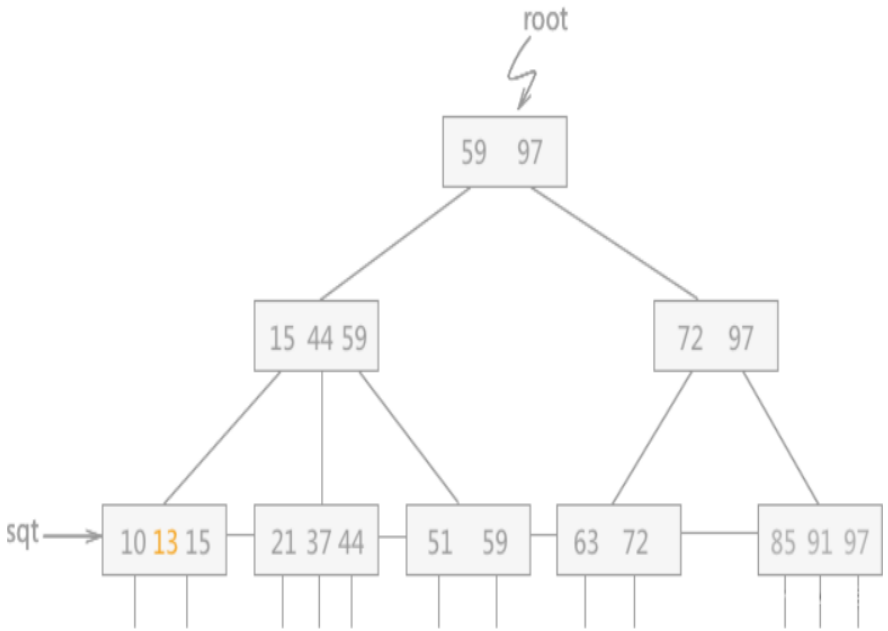
# B+树 插入

- 插入的操作全部都在叶子结点上进行，且不能破坏关键字自小而大的顺序；
- 由于 B+树中各结点中存储的关键字的个数有明确的范围，做插入操作可能会出现结点中关键字个数超过阶数的情况，此时需要将该结点进行“分裂”。

# B+树 插入

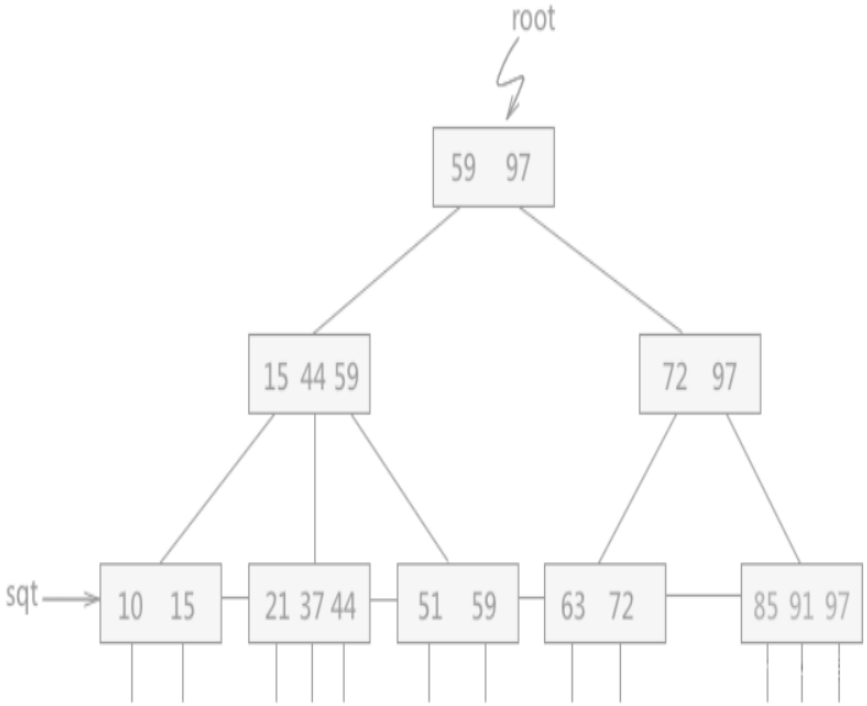
- 若被插入关键字所在的结点，其含有关键字数目小于阶数 M，则直接插入

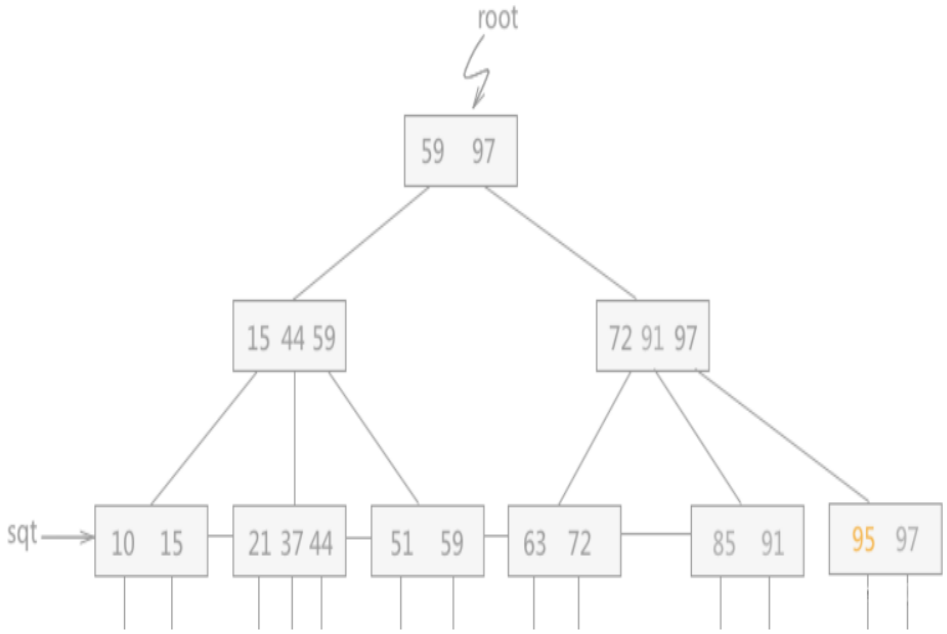


# B+树 插入

若被插入关键字所在的结点，其含有关键字数目等于阶数 M，则需要将该结点分裂为两个结点，一个结点包含⌊M/2⌋，另一个结点包含⌈M/2⌉。同时，将⌈M/2⌉的关键字上移至其亲结点。假设其双亲结点中包含的关键字个数小于 M，则插入操作完成。

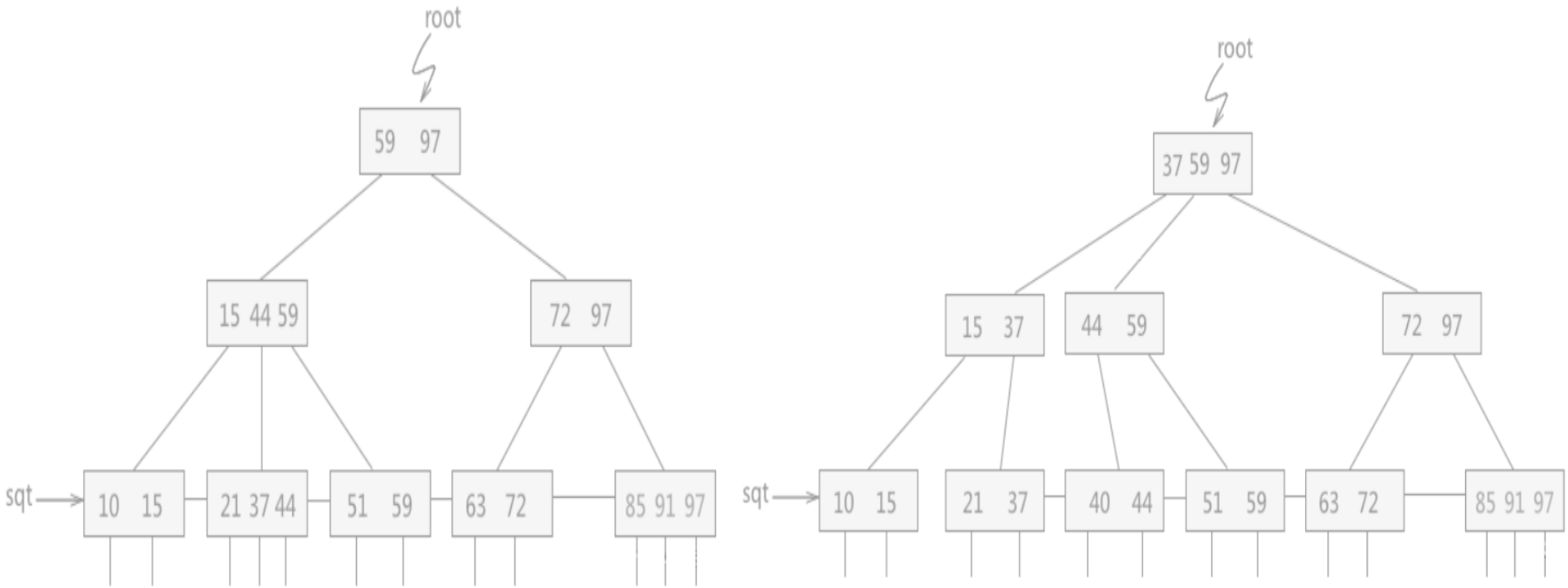



插入95

# B+树 插入

如果上移操作导致其双亲结点中关键字个数大于 M，则应继续分裂其双亲结点。



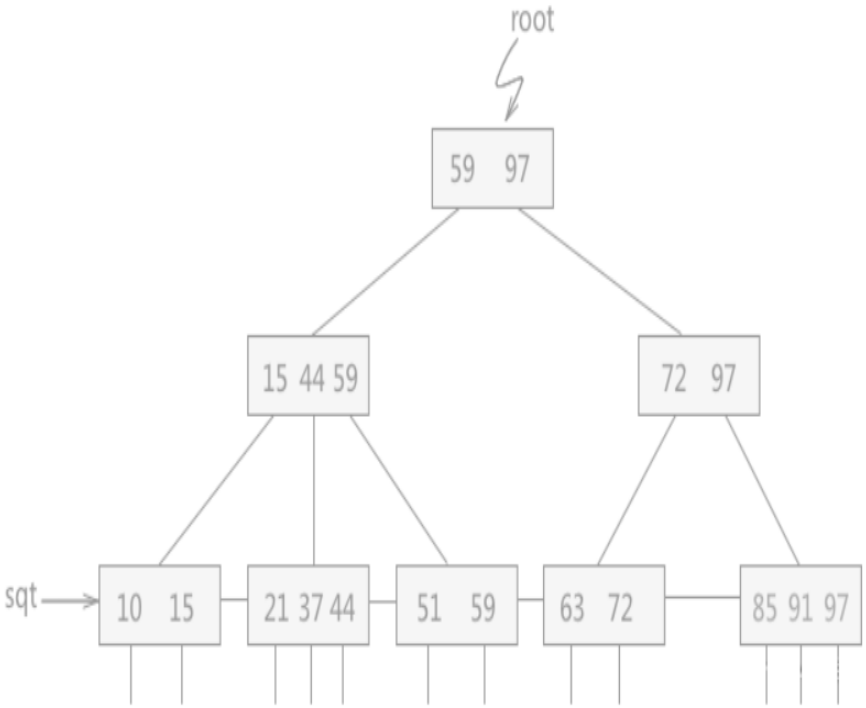

插入40

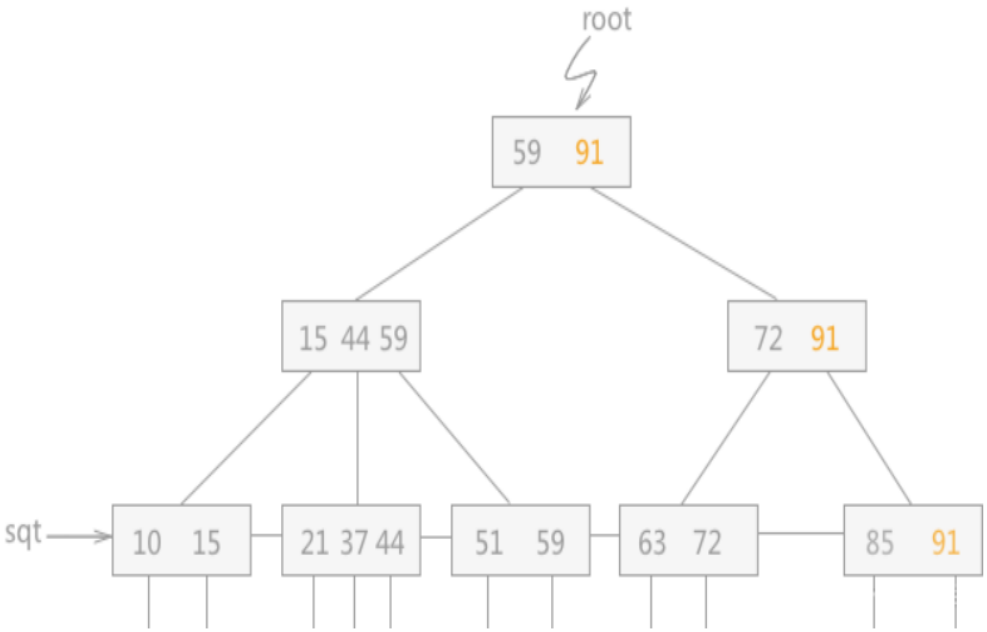
# B+树 删除

- 删除该关键字，如果不破坏 B+树本身的性质，直接完成操作；
- 如果删除操作导致其该结点中最大（或最小）值改变，则应相应改动其父结点中的索引值；
- 在删除关键字后，如果导致其结点中关键字个数不足，有两种方法：一种是向兄弟结点去借，另外一种是同兄弟结点合并。（注意这两种方式有时需要更改其父结点中的索引值。）

# B+树 删除

若该结点中关键字个数大于⌈M/2⌉，做删除操作不会破坏 B+树，则可以直接删除。当删除某结点中最大或者最小的关键字，就会涉及到更改其双亲结点一直到根结点中所有索引值的更改。

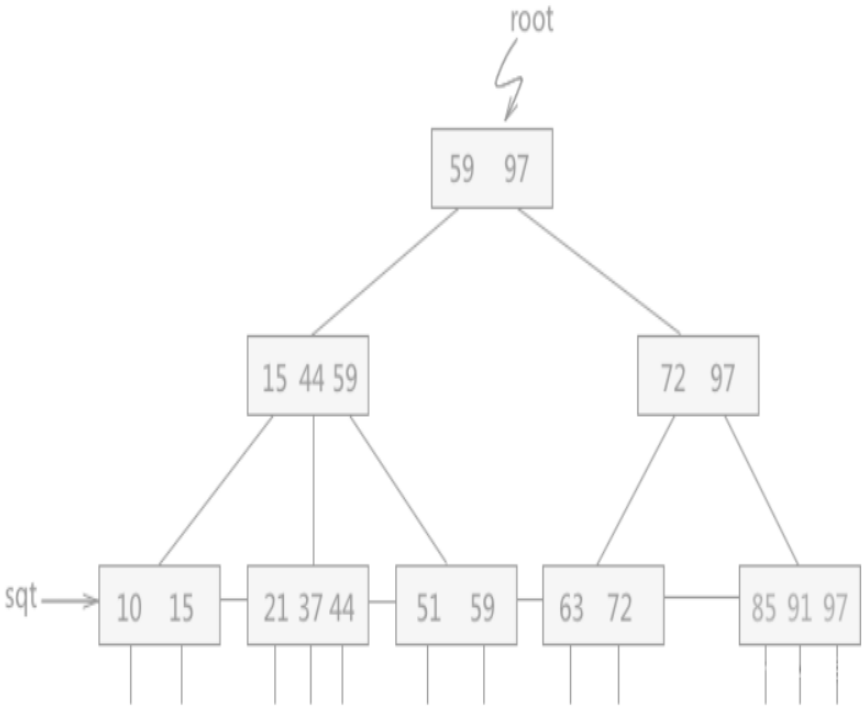


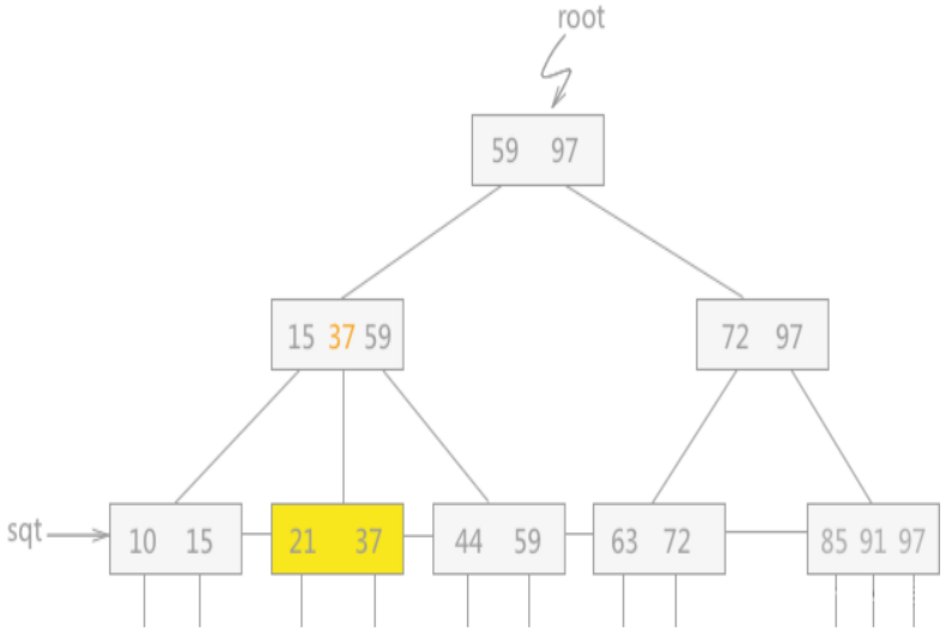




删除97

# B+树 删除

当删除该关键字，导致当前结点中关键字个数小于⌈M/2⌉，若其兄弟结点中含有多余的关键字，可以从兄弟结点中**借关键字**完成删除操作。

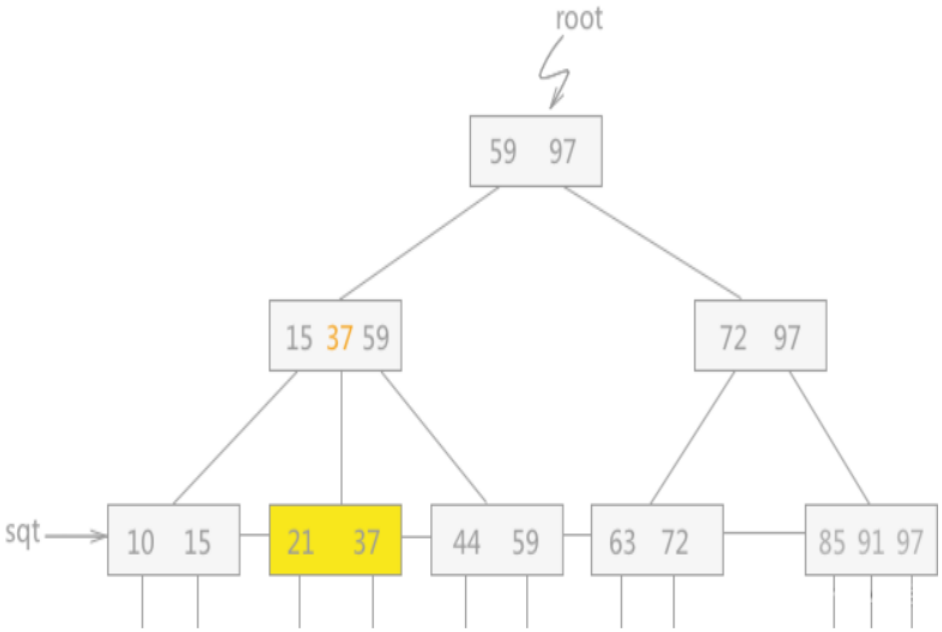


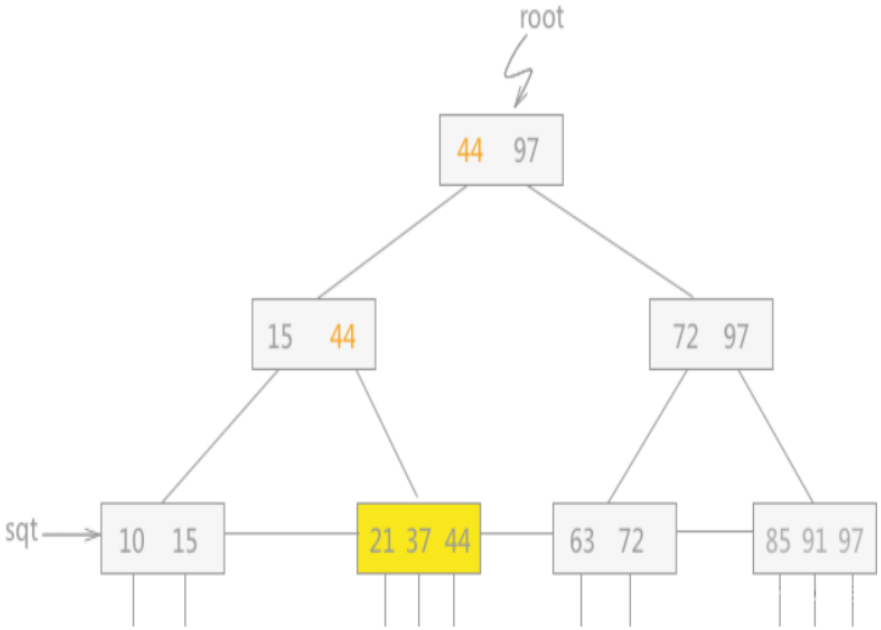




删除51

# B+树 删除

如果其兄弟结点没有多余的关键字，则需要同其兄弟结点**进行合并**。

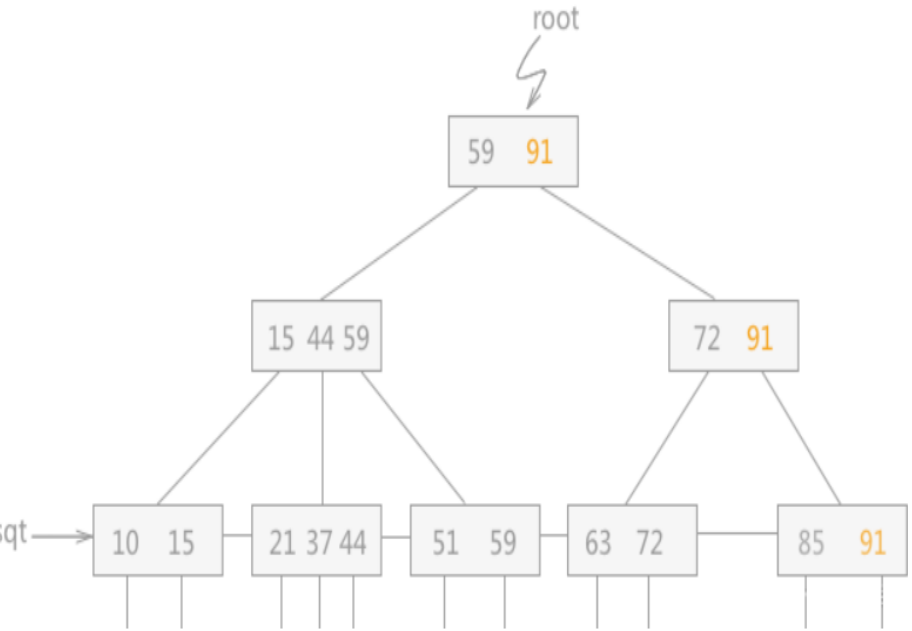


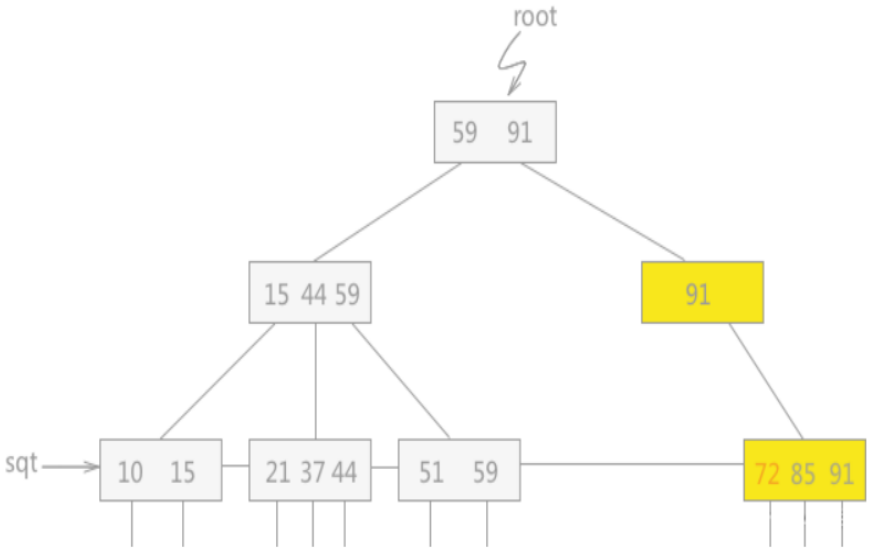

删除59

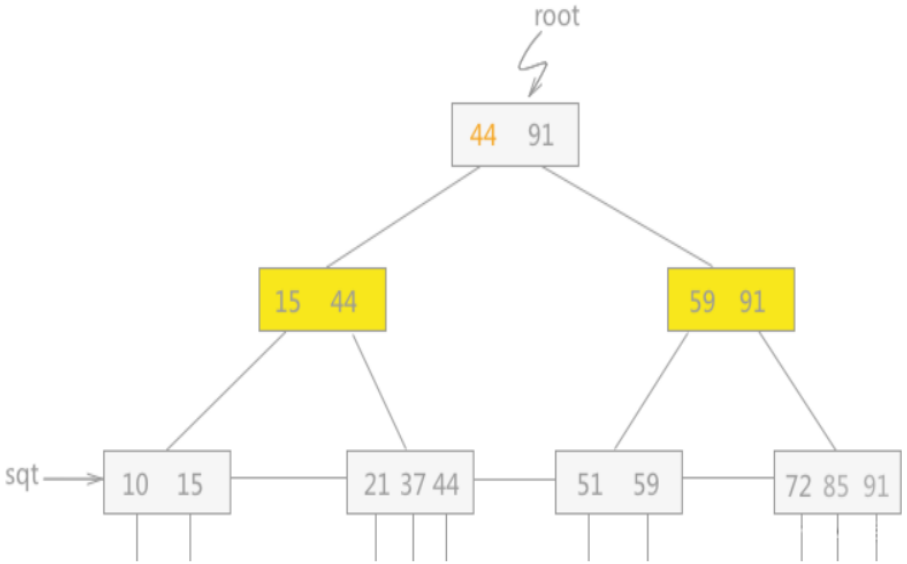
# B+树 删除

进行合并时，可能会产生因合并使其双亲结点破坏 B+树的结构，需要依照以上规律处理其双亲结点。

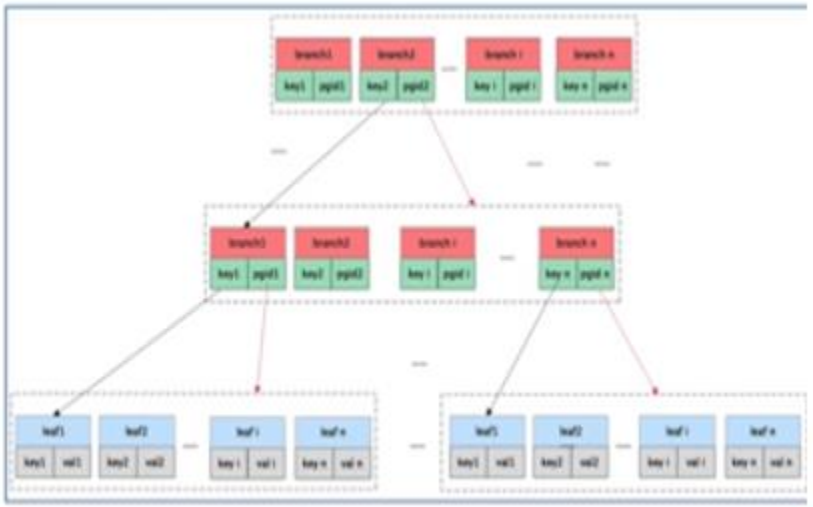



删除63



# B树和B+树对比



每页中的索引项中是否有必要额外存储该索引对应的记录原始数据呢？

| 方案选型 | 存储原始数据 | 不存原始数据 |
| --- | --- | --- |
| 特点1 | 查询到该索引项时，可以直接返回数据，而不必遍历到最底层节点 | 不存储时，所有的查询数据都需要遍历到最底层节点，才能返回数据 |
| 特点2 | 存储同等数据量级的数据时，树的高度会偏高，部分读写请求涉及的磁盘IO相对较多 | 存储同等数据量级的数据时，树的高度相对较低，因此涉及的磁盘IO较少 |
| 特点3 | 不同的查询请求，时间复杂度不均衡 | 不同的查询请求，时间复杂度均衡 |
| 数据结构 | B树 | B+树 |

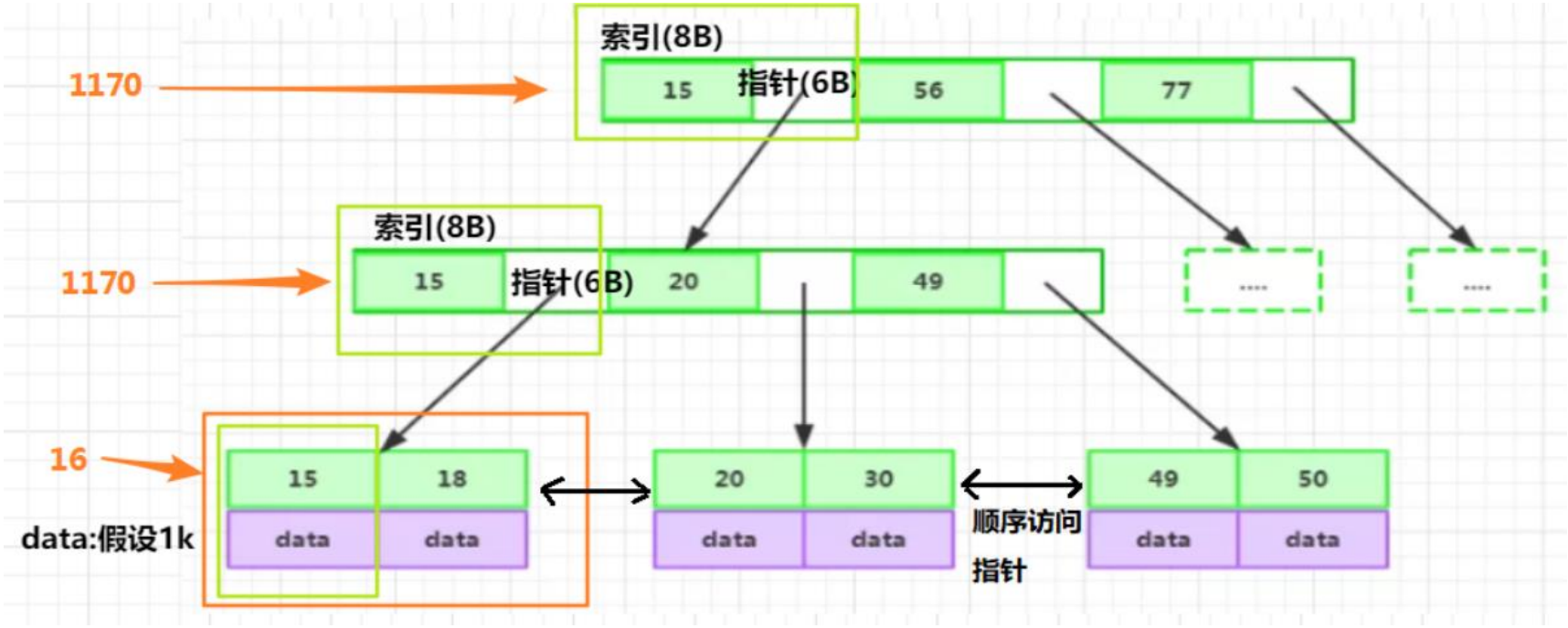
# 为什么B+树比B树更适合数据库索引？

1）B+树查询效率更加稳定

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。

2）B+树便于范围查询（最重要的原因，范围查找是数据库的常态）

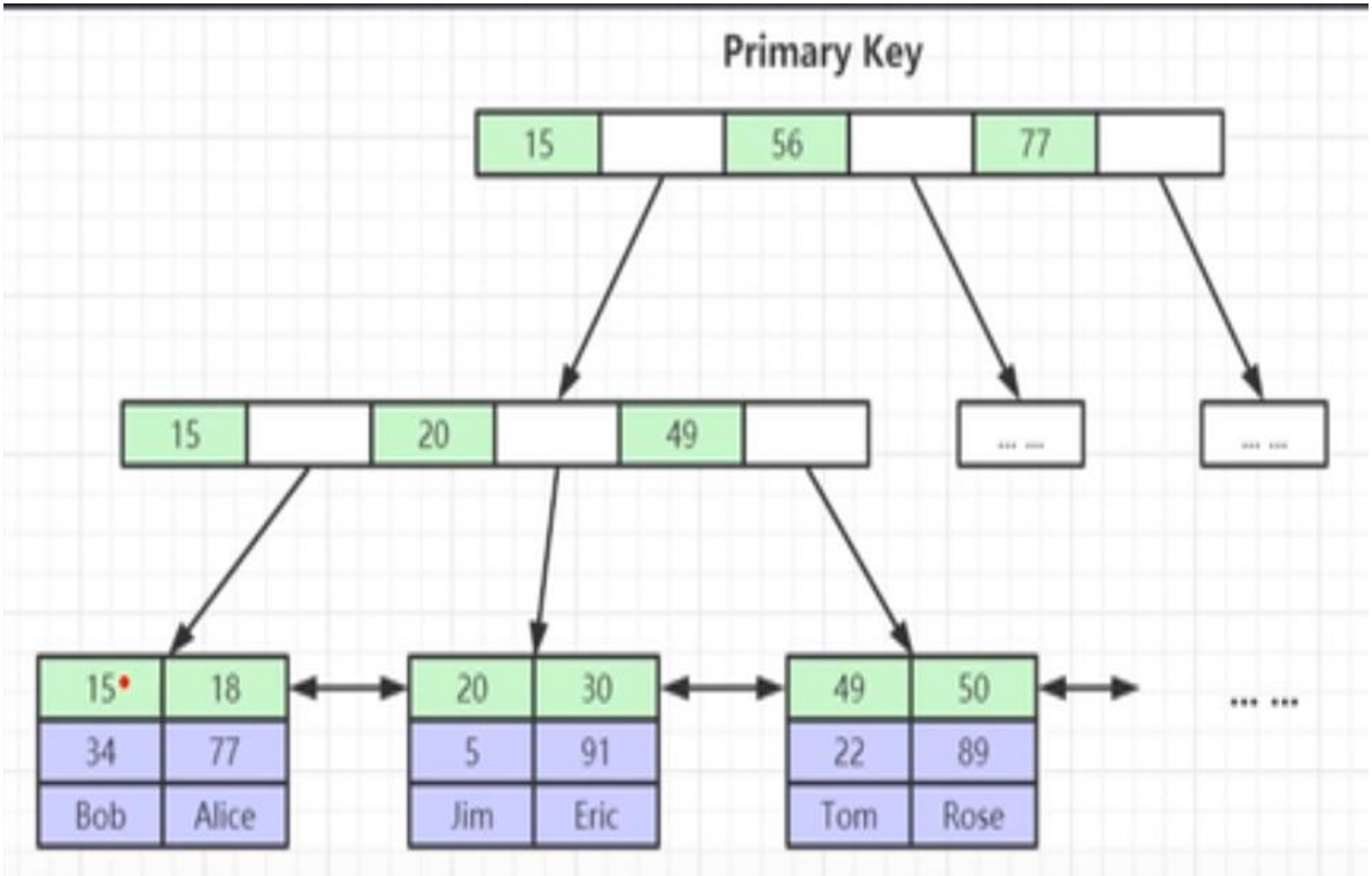
B+树只需要去遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作。



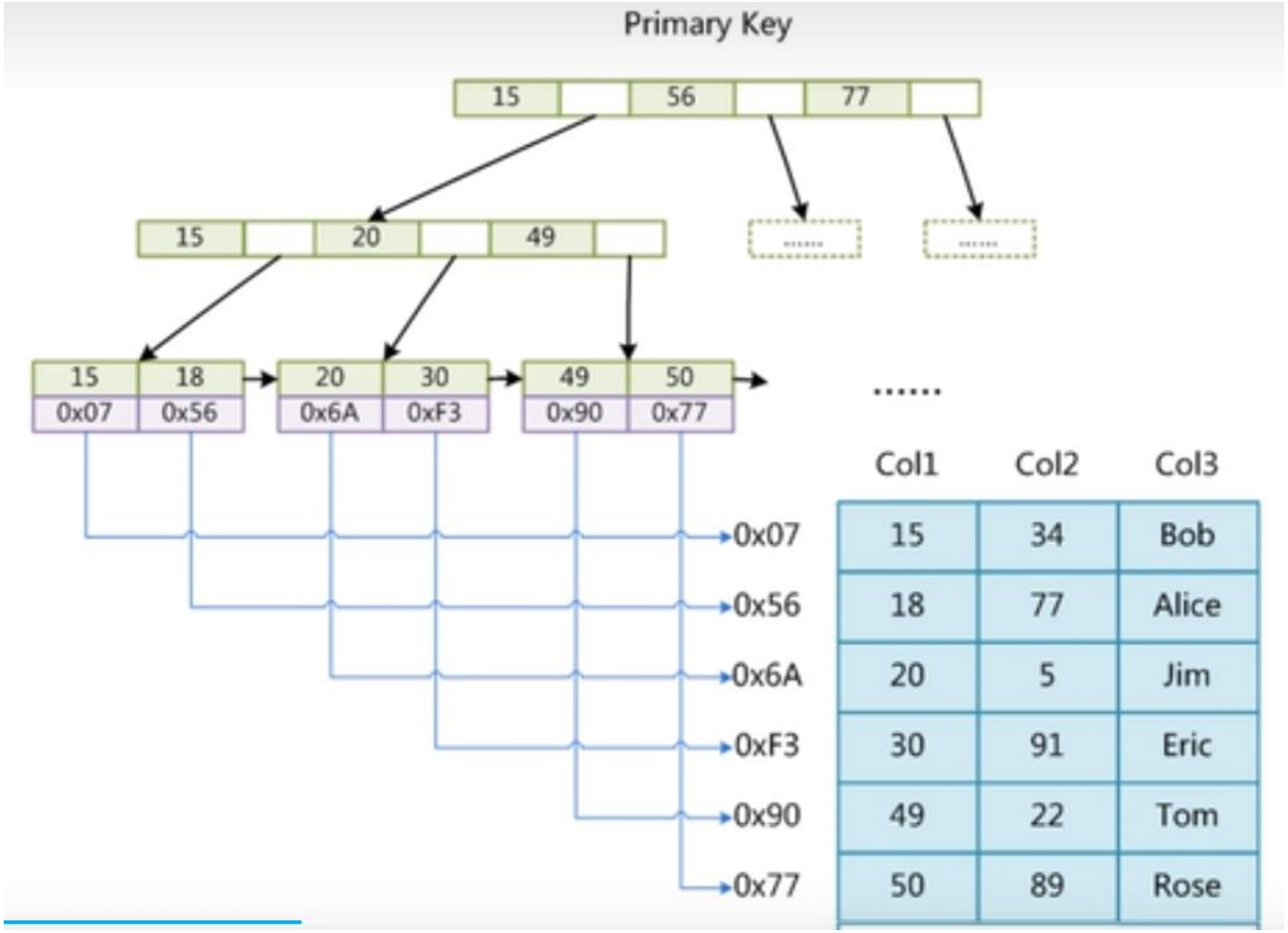

- **MySQL** 官方对非叶子节点(如最上层 **h = 1**的节点，**B+Tree**高度为**3)** 的大小是有限制的，可以得到大小为 **16384**，即 **16k**大小。
- 假如：**B+Tree**的表都存满了。索引的节点的类型为**BigInt**，大小为**8B**，指针为**6B**。
- 最后一层，假如 **存放的数据data为1k** 大小，那么
- 第一层最大节点数为： **16k / (8B + 6B) = 1170 (个)**；
- 第二层最大节点数也应为：**1170个**；
- 第三层最大节点数为：**16k / 1k = 16 (个)**。
- 则，一张B+Tree的表最多存放 **1170 * 1170 * 16 ≈ 2千万**。

- ✓ 聚集索引（聚簇索引）：表中的数据都会有一个主键，即使不创建主键，系统也会创建一个隐式的主键。这是因为**innodb是把数据存放在B+树中的**，而B+树的键值就是主键，在B+树的叶子节点中，存储了表中所有的数据。这种**以主键作为B+树索引的键值而构建的B+树索引，称之为聚集索引**。
- ✓ 非聚集索引（非聚簇索引）：以主键以外的列值作为键值构建的B+树索引，称之为非聚集索引。

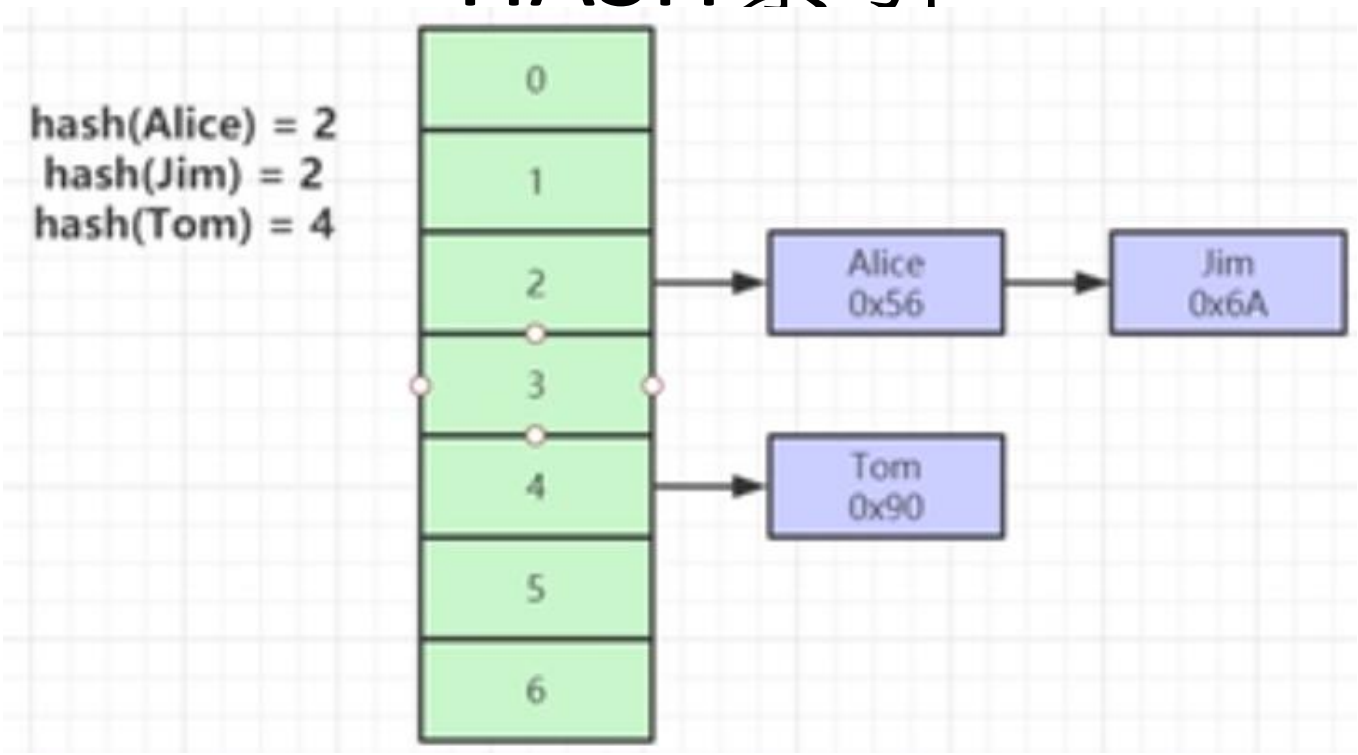
# 聚集索引


Primary Key
15
56
77
15
20
49
... ...
... ...
15
18
34
77
Bob
Alice
20
30
5
91
Jim
Eric
49
50
22
89
Tom
Rose
... ...

# 非聚集索引


Primary Key
15
56
77
15
20
49
.....
.....
15
18
20
30
49
50
0x07
0x56
0x6A
0xF3
0x90
0x77
.....
0x07
0x56
0x6A
0xF3
0x90
0x77
Col1
Col2
Col3
15
34
Bob
18
77
Alice
20
5
Jim
30
91
Eric
49
22
Tom
50
89
Rose

# HASH索引




对索引的key进行一次hash计算就可以定位出数据存储的位置
很多时候Hash索引要比B+树索引更高效
仅能满足‘’=“ ”in“，"IN" ,不支持范围查询
hash冲突问题