实用算法设计——查找

主讲：娄文启

louwenqi@ustc.edu.cn

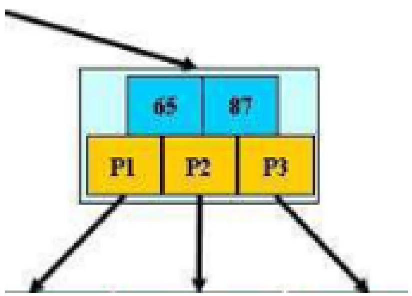
B树（B-tree）的定义

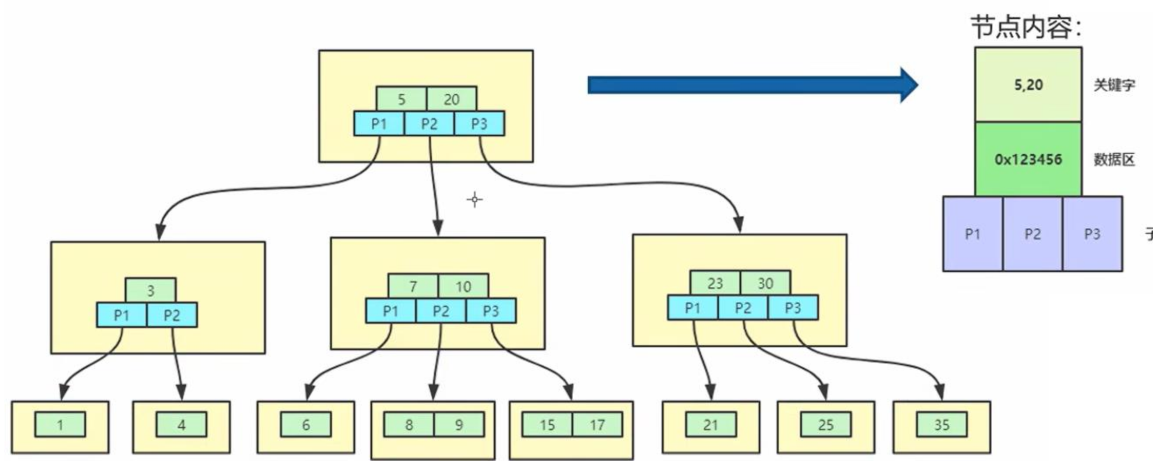
- B树是一种平衡的多路查找树，适用于外查找，常用于数据库和文件系统，由R.Bayer和E.mccreight 于1970年提出。
- 一棵m阶的B树的定义(递归)：或者是一棵空树，或者是满足下列特性的m叉树：
  - 树中每个结点至多有m棵子树;
  - 若根结点不是叶子结点，则至少有两棵子树；
  - 除根之外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树，至多有m棵子树
  - 所有的非终端结点中包含下列信息数据(n, P0, K1, P1, K2, P2,…, Kn , Pn)
  - 所有叶子结点都出现在同一层，叶子结点可看作Null节点 ；

- (n, P0, K1, P1, K2, P2,…, Kn , Pn)
  - n为关键字的个数
  - Ki是关键字，且Ki< Ki+1；
  - Pi为指向子树根结点的指针，且Pi-1所指向的子树中所有结点的关键字均小于Ki，Pn所指向的子树中所有结点的关键字均大于Kn ；
  - 最小度数(t)，[t-1，2t-1]关键字
- 在B树中，每个结点中关键字从小到大排列，并且当该结点的孩子是非叶子结点时，该n个关键字正好是(n+1)个孩子包含的关键字的值域的分划。

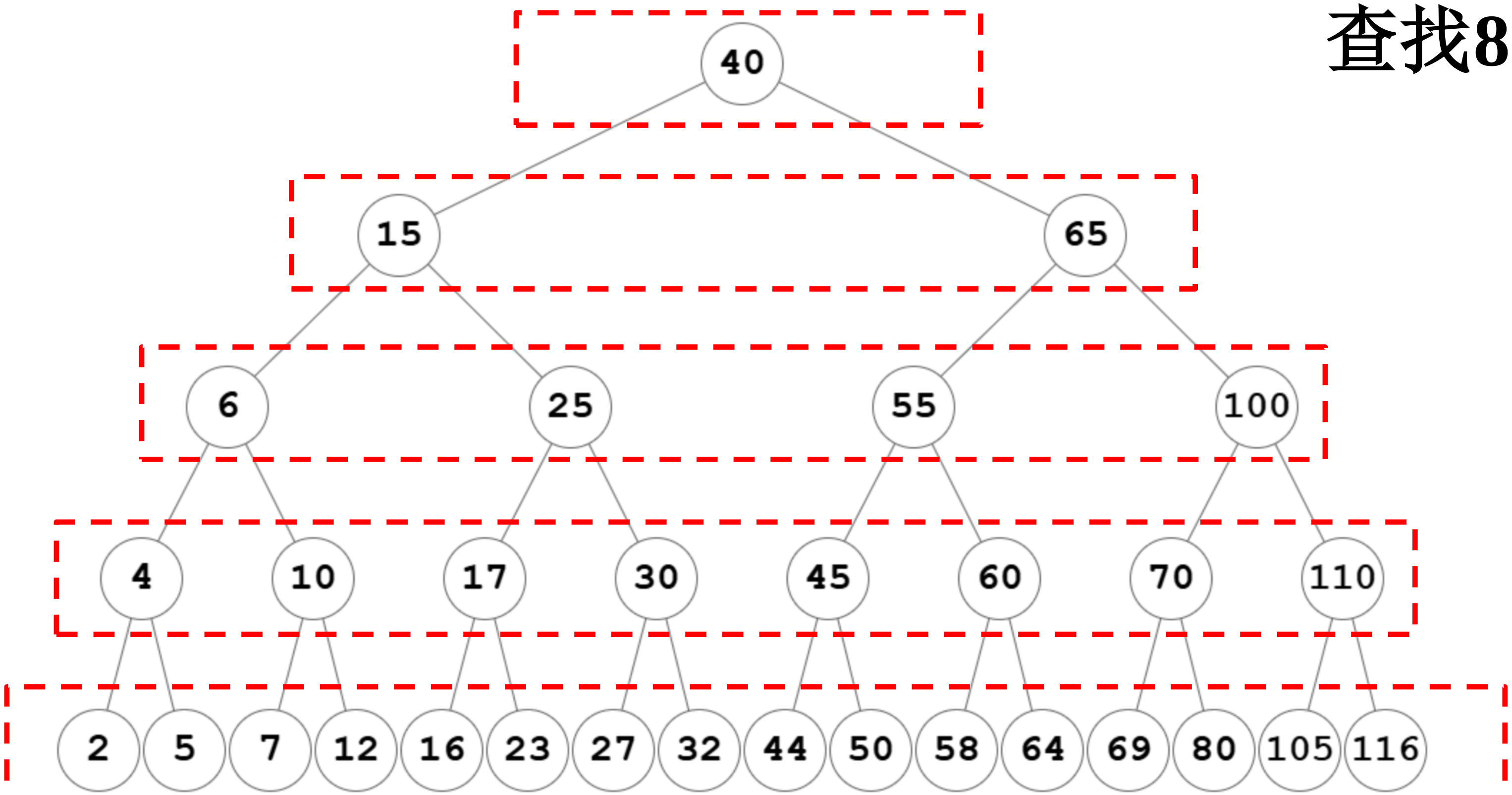
- 注意：关键字K_i=(key($Record_i$),Address($Record_i$))

- (n, P0, K1, P1, K2, P2,…, Kn , Pn)
- 注意：关键字K_i=(key($Record_i$),Address($Record_i$))





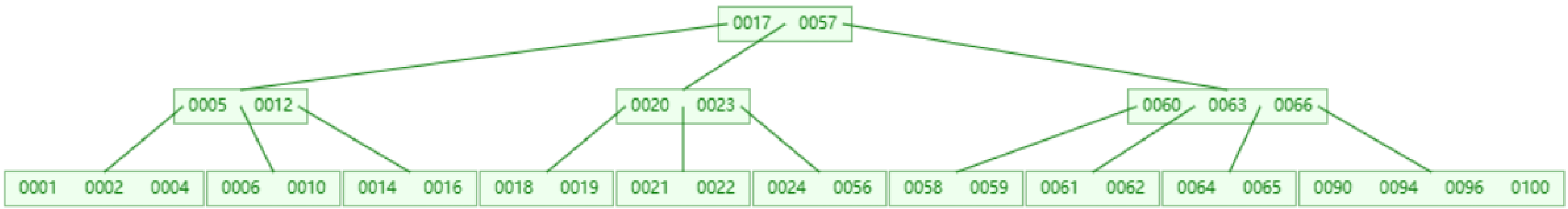
为何要引入B树



查找80？

本质上还是数据量和树高$\log_2 N$的关系
而内存是有限的

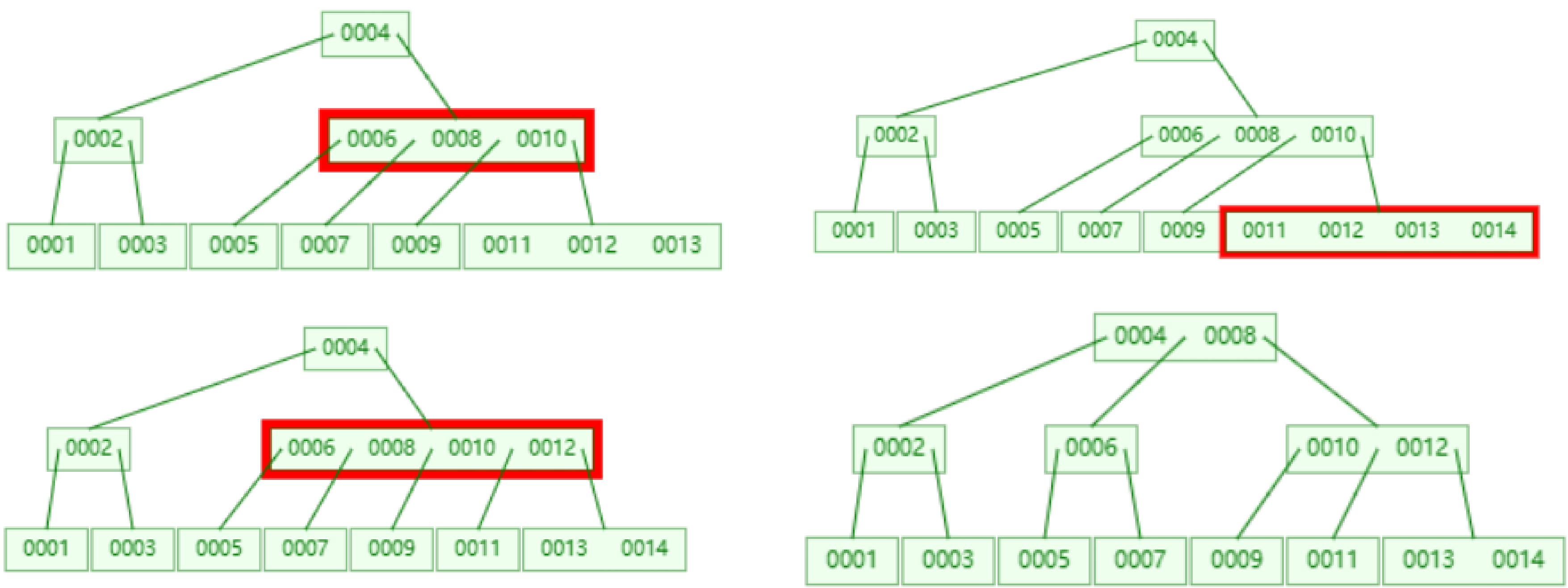
为何要引入B树



高度低、数据局部性好

为什么说B树是平衡的

- 高度平衡：B树通过每次插入和删除操作时的**分裂与合并机制**来自动调整树的高度，确保树不会严重倾斜。当一个节点的数据量超过预定的最大容量时，它会被分裂成两个或多个节点，以维持每个节点的数据量在一个预设范围内，从而控制树的高度不会无限增长



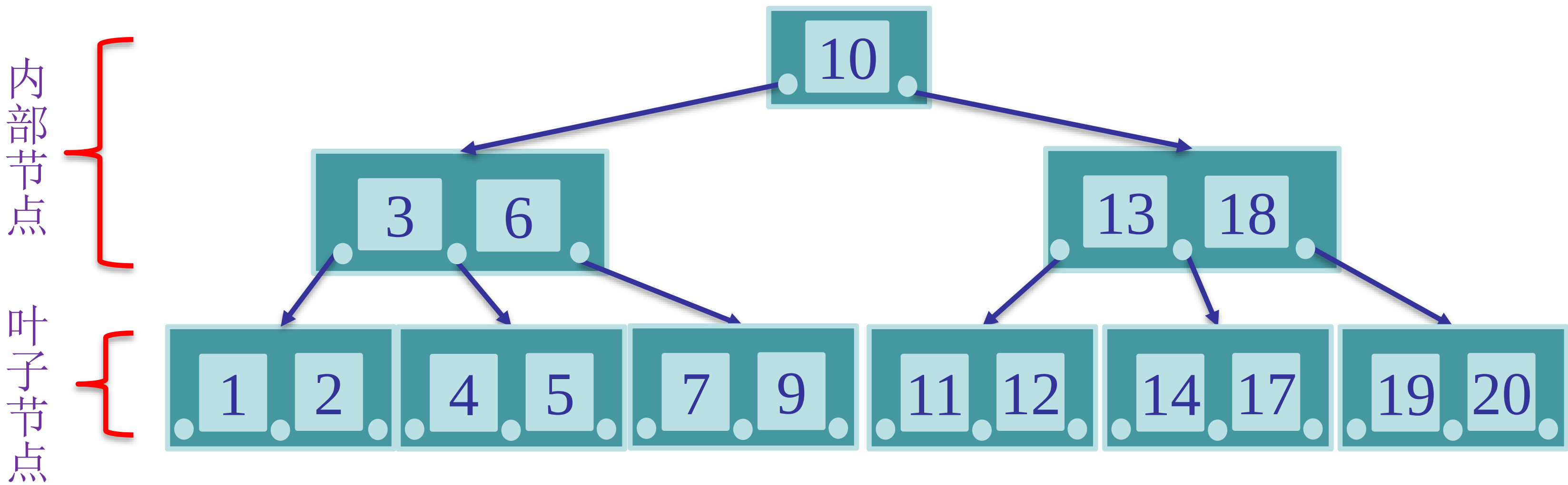
为什么说B树是平衡的

- 高度平衡：B树通过每次插入和删除操作时的**分裂与合并机制**来自动调整树的高度，确保树不会倾斜。当一个节点的数据量超过预定的最大容量时，它会被分裂成两个或多个节点，以维持每个节点的数据量在一个预设范围内，从而控制树的高度不会无限增长

- 数据分布均匀：在B树中，数据项是按照顺序分布在各个节点中的，每个节点可以存储多个数据项和对应的子节点指针，这有助于数据在树中的均匀分布，减少了查找路径上的跳跃，使得查找、插入和删除等操作更加高效。

高度平衡、数据分布均匀

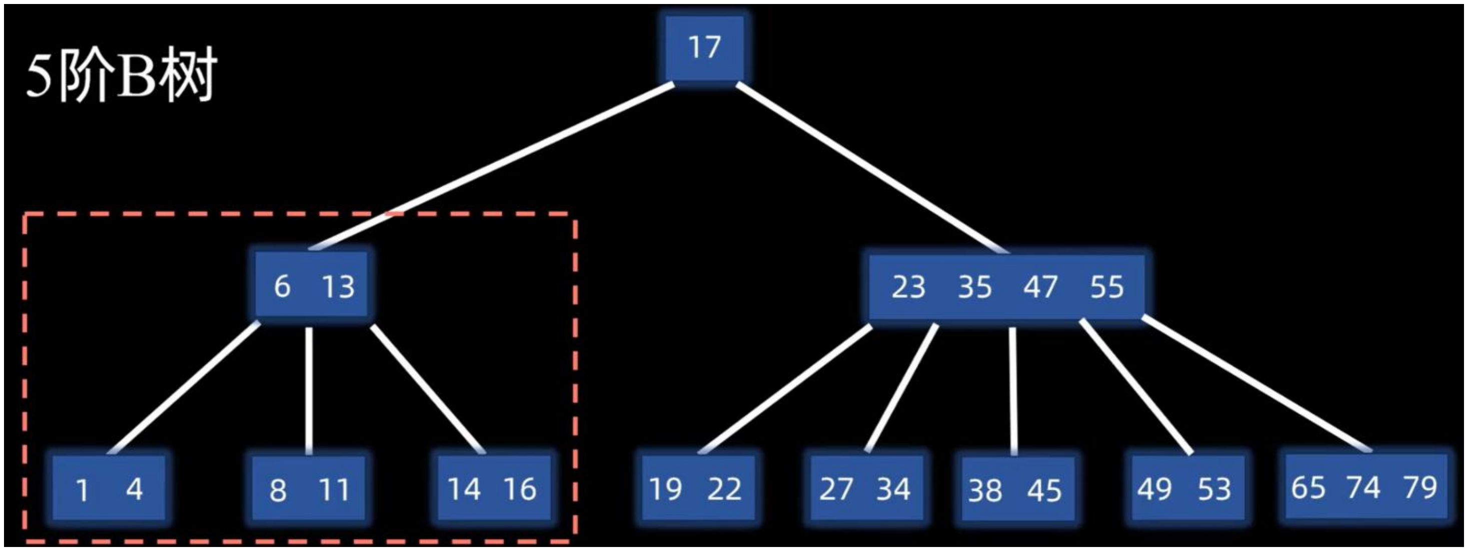
B 树示例



多叉平衡搜索树：

1. 访问节点是在**硬盘上**进行的，节点内的数据操作是在**内存中进行**的
2. 每次load进内存的数据量多了（**但这并不影响实际速度**）

B 树特性



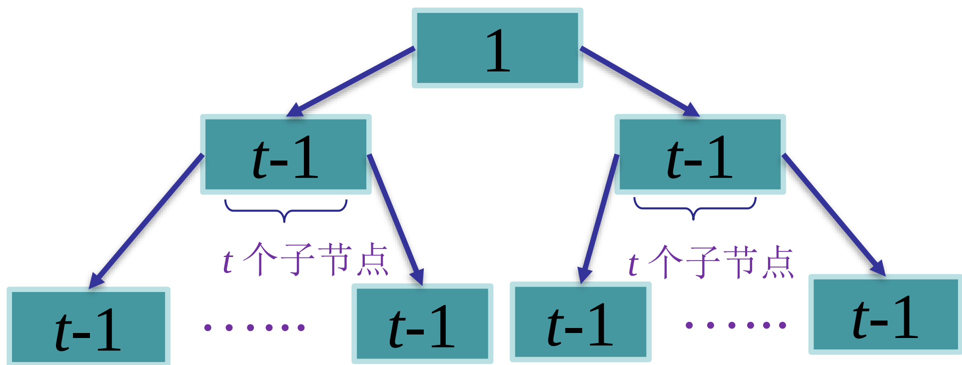
平衡：所有叶子节点都在同一层

有序：节点内有序（任一元素的左子树都小于它，右子树都大于它）

多路：对于m阶的B树，最多m个分支，除根节点外，最少$\lceil m/2 \rceil$个分支(上取整)；根节点最少1个

B 树高度

证明：如果 $n \geq 1$,那么对于任意一颗包含 n 个关键字，高度为 h，**最小度数** $t \geq 2$ 的 B树 T，有：$h \leq \log_t \frac{n+1}{2}$



深度	节点数
0	1
1	2
2	$2t$

有：$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$

即：$t^h \leq \frac{n+1}{2} \Rightarrow h \leq \log_t \frac{n+1}{2}$

B树——查找

- 如何查找？（纵向查节点+横向查关键字）
  - **Step1:**从根节点开始。
  - **Step2:**对节点内的关键字（有序）序列进行二分查找。
  - **Step3:**若匹配，则结束；否则，找到第一个比当前值小的**key**进入左子树，否则进入最大**key**的右子树
  - **Step4:**重复**Step2~3**，直到所对应的孩子指针为空，或已经是叶子节点。
- 查找效率：主要取决于待查关键字所在节点在B树中所处于的层次数。

B 树的存储结构中，节点的类型定义如下：

```
#define MAXM 10              // 定义 B树的最大的阶数
typedef int KeyType;         // KeyType为关键字类型
typedef struct node {
   int keynum;              // 节点当前拥有的关键字的个数
   KeyType key[MAXM];       // 存放关键字的数组，范围[1..keynum-1]
   struct node *parent;      // 父节点指针
   struct node *ptr[MAXM]; // 子节点指针数组，范围[0..keynum]
} BTNode;
```

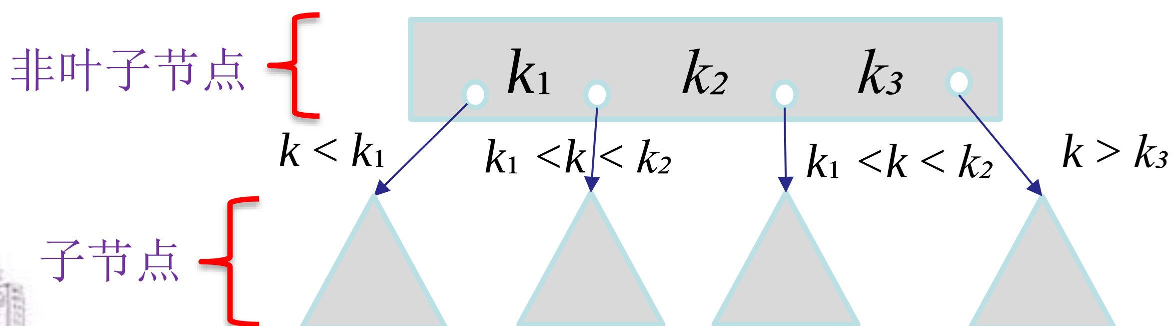
```
class BTreeNode{
  int *keys;  // 关键字数组
  int t;      // 最小度 (定义了节点关键字的数量限制)
  BTreeNode **C; // 节点对应孩子节点的数组指针
  int n;     // 节点当前的关键字数量
  bool leaf; // 当节点是叶子节点的时候为true 否则为false
}
```

B 树 查找

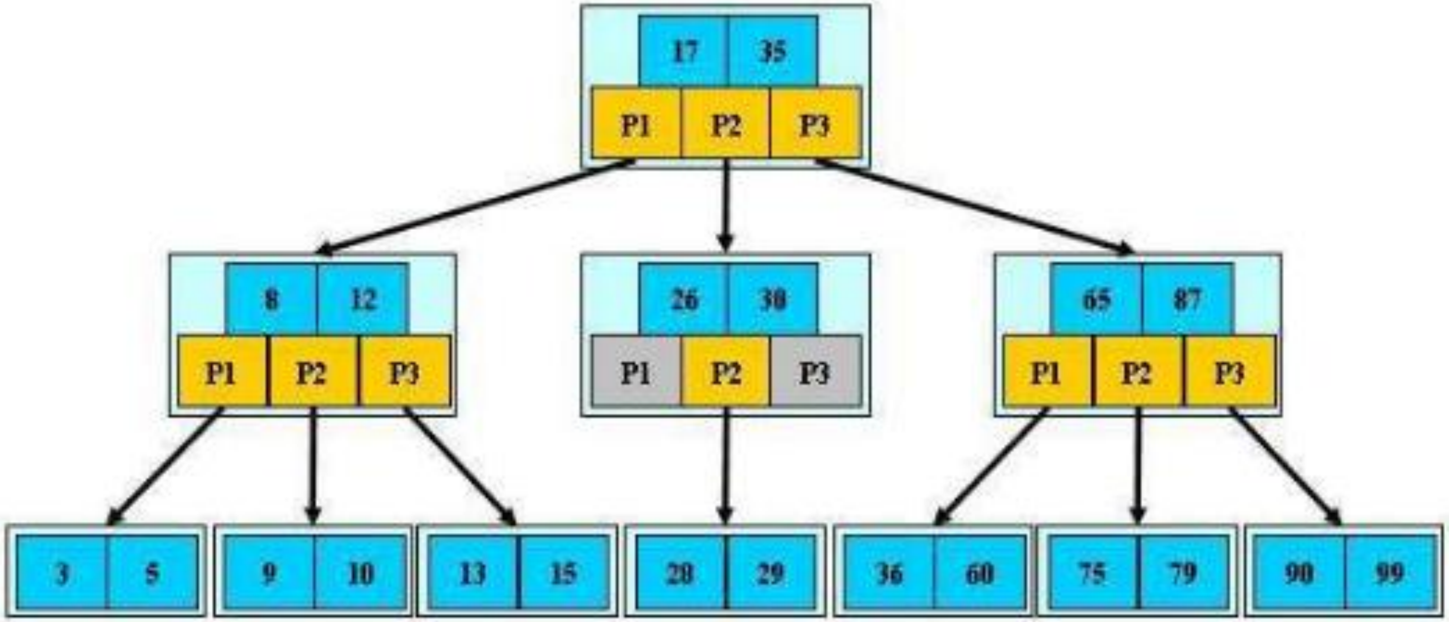
B 树中的节点包含有多个关键字。假设需要查找的是 k，那么从根节点开始，从上到下递归的遍历树。在每一层上搜索的范围为包含了搜索值的子树中。 子树值的范围由它的父节点的关键字确定。流程如下：

将 k 与根节点中的 key[i] 进行比较：

1. 若 k = key[i]，则查找成功；
2. 若 k ＜ key[1]，则沿着指针 ptr[0] 所指的子树继续查找；
3. 若 key[i] ＜ k ＜ key[i +1]，则沿着指针 ptr[i] 所指的子树继续查找；
4. 若 k ＞ key[n]，则沿着指针 ptr[n] 所指的子树继续查找；
5. 若没有找到关键字 k 且当前节点为叶子节点，则查找失败。



B 树查找过程



1）查找Key=35的记录的磁盘地址？

2）查找Key=87的记录的磁盘地址？

3）查找Key=92的记录的磁盘地址？

B 树 插入

将关键字 k 插入到 B树的过程分两步完成：

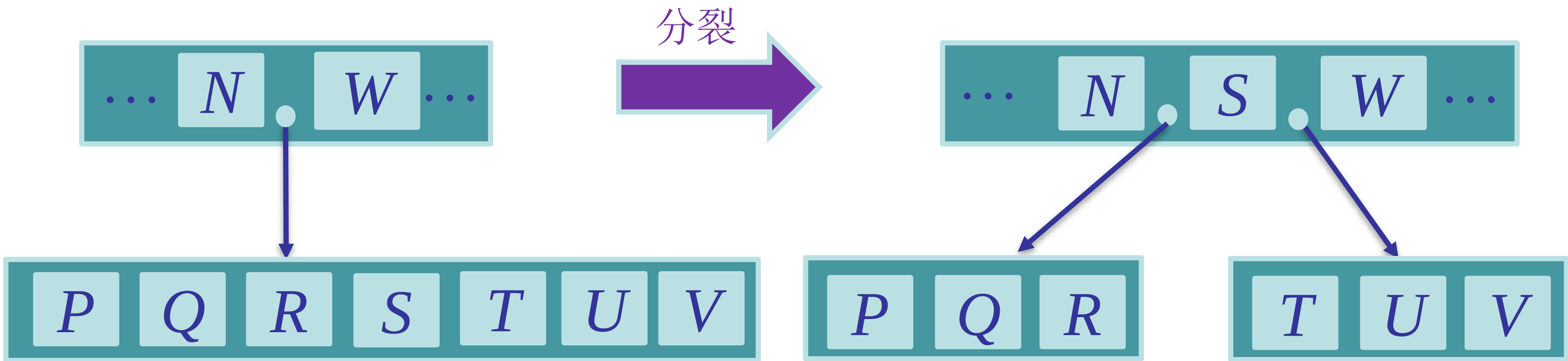
1. 查找该关键字的插入节点(注意B树的插入节点一定是叶子节点层的节点)
2. 插入关键字

在某个叶子节点中插入关键字分两种情况：

1. 插入节点有空位置，即关键字个数 $n < m - 1$，直接把关键字 k 有序插入到该节点的合适位置上。
2. 插入节点没有空位置，即原关键字个数 $n = m - 1 \Rightarrow$ 分裂

B 树 插入（先插入后分裂）

分裂过程： m = 7

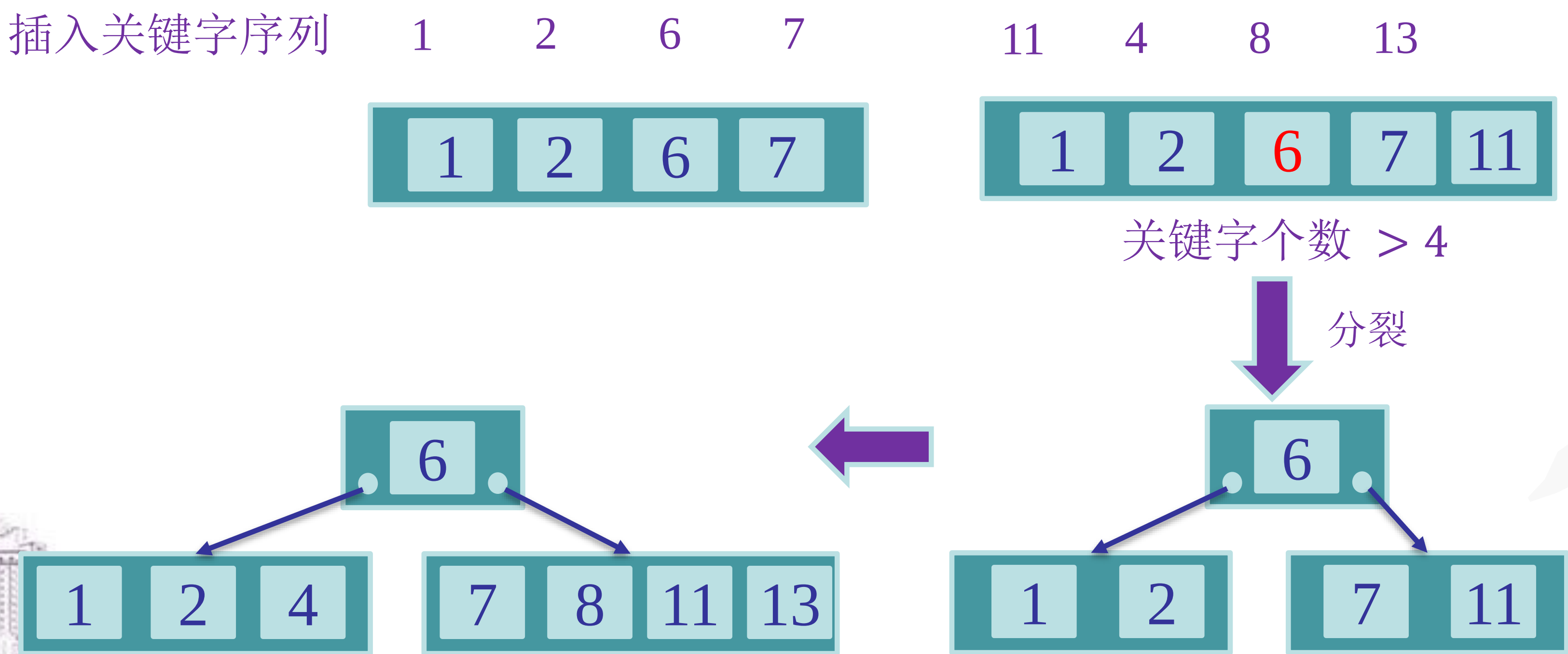


1. 从该节点的原有元素和新的元素中选择出**中位数元素**(中间关键字)
2. 小于这一中位数的元素放入左边节点，大于这一中位数的元素放入右边节点，中间关键字作为分隔值，左右节点各含有$\lceil 2/m \rceil$ - 1个关键字
3. 中间关键字被插入到父节点中，可能会造成**父节点分裂**，分裂父节点时可能又会使它的父节点分裂，以此类推。如果没有父节点（这一节点是根节点），就创建一个新的根节点（增加树的高度）

B 树 插入（先插入后分裂）

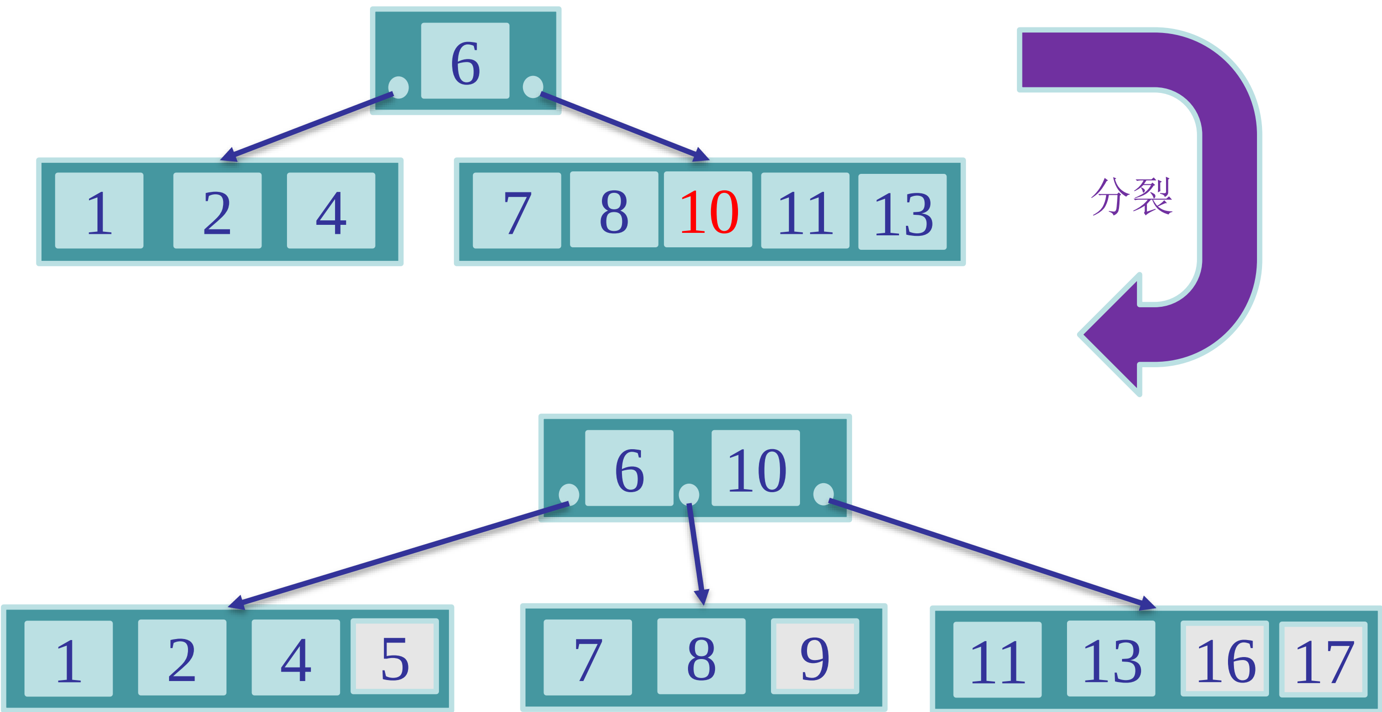
关键字序列为：（**1，2，6，7，11，4，8，13**，10，5, 17，9，16，20，3， 12，14，18，19，15)，创建一棵 5 阶 B树。

节点最多关键字个数为 m -1 = 4



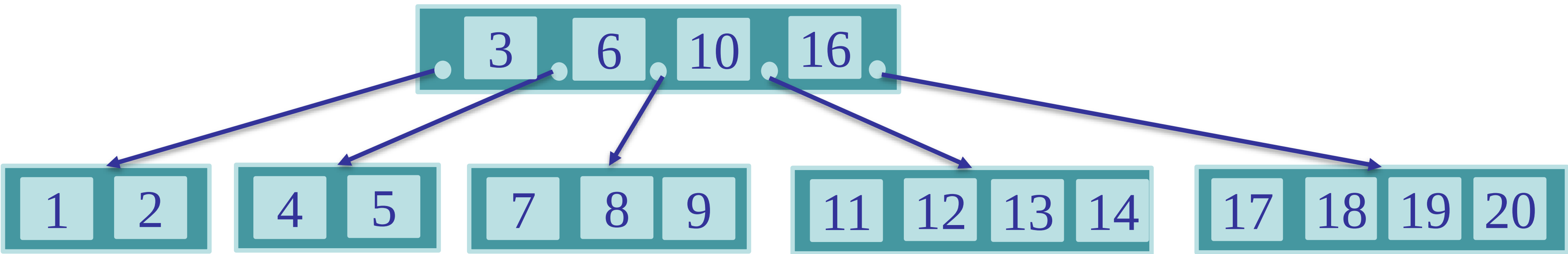
B 树 插入示例

插入关键字序列 **10** 5 17 9 16

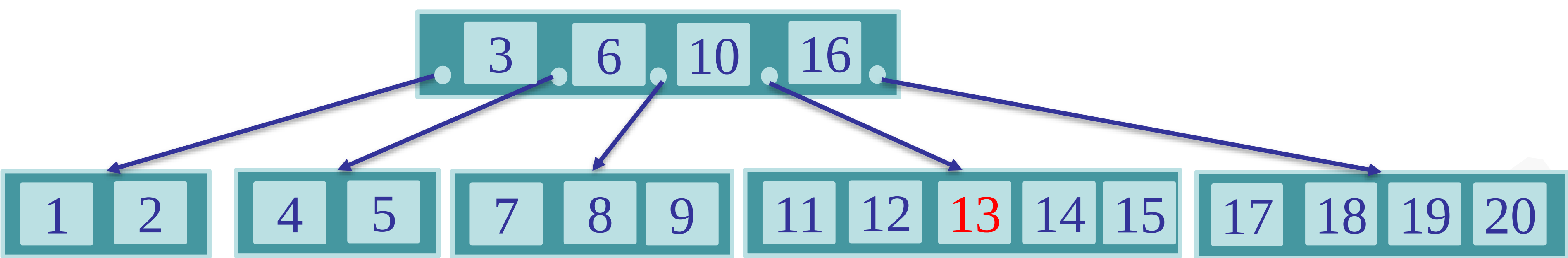


B 树 插入示例

插入关键字序列　20　3　12　14　18　19

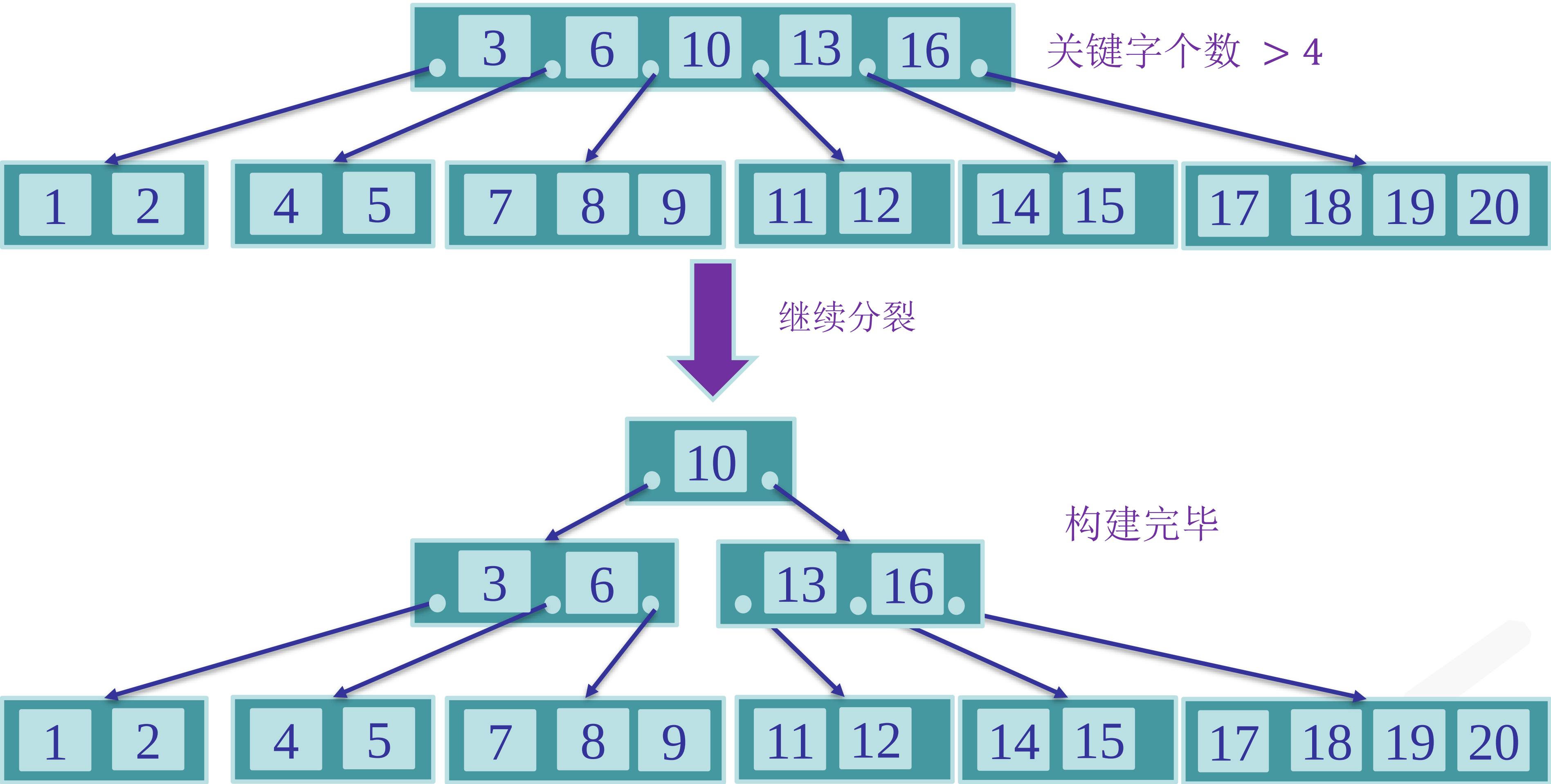


插入关键字序列　15



分裂

B 树 插入示例


3
6
10
13
16
关键字个数 ＞4
1
2
4
5
7
8
9
11
12
14
15
17
18
19
20
继续分裂
10
构建完毕
3
6
13
16
1
2
4
5
7
8
9
11
12
14
15
17
18
19
20

B 树 代码示例-构造函数

```
BTreeNode::BTreeNode(int _t, bool _leaf)
{
    // 复制参数中的最小度数以及叶子布尔值
    t = _t;
    leaf = _leaf;
    // 分配节点可以存放关键字的最大内存，以及孩子指针
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];
    // 初始化节点内部的孩子数目
    n = 0;
}
```

B 树 代码示例-search函数

```
BTreeNode *BTreeNode::search(int k)
{
    int i = 0;
    while (i < n && k > keys[i])// The first Key (>= k)
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;
    // Go to the appropriate child
    return C[i]->search(k);}
```

B 树 代码示例-Insert函数

```
void BTree::insert(int k){
    if (root == NULL){
        root = new BTreeNode(t, true);
        root->keys[0] = k;  // Insert key
        root->n = 1;  // Update number of keys in root}
    else // If tree is not empty{
        if (root->n == 2*t-1){
            BTreeNode *s = new BTreeNode(t, false); // build new root
            s->C[0] = root; // Make old root as child of new root
            // Split old root, move 1 key to the new root, location:0
            s->splitChild(0, root);
            // New root has two children now. Decide which of the
            // two children is going to have new key
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);
            // Change root
            root = s;}
        else root->insertNonFull(k);}}
```

B 树 代码示例-Insert函数

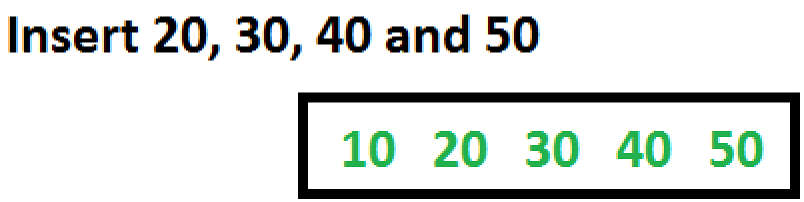
```
void BTreeNode::insertNonFull(int k){
    int i = n-1;
    if (leaf == true){
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k){
            keys[i+1] = keys[i];
            i--;}
        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;}
    else // If this node is not leaf{
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k) i--;
        // See if the found child is full
        if (C[i+1]->n == 2*t-1){
            // split it, new key location: i+1
            splitChild(i+1, C[i+1]);
            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two.
          if (keys[i+1] < k) i++;
        }
        C[i+1]->insertNonFull(k);}}
```

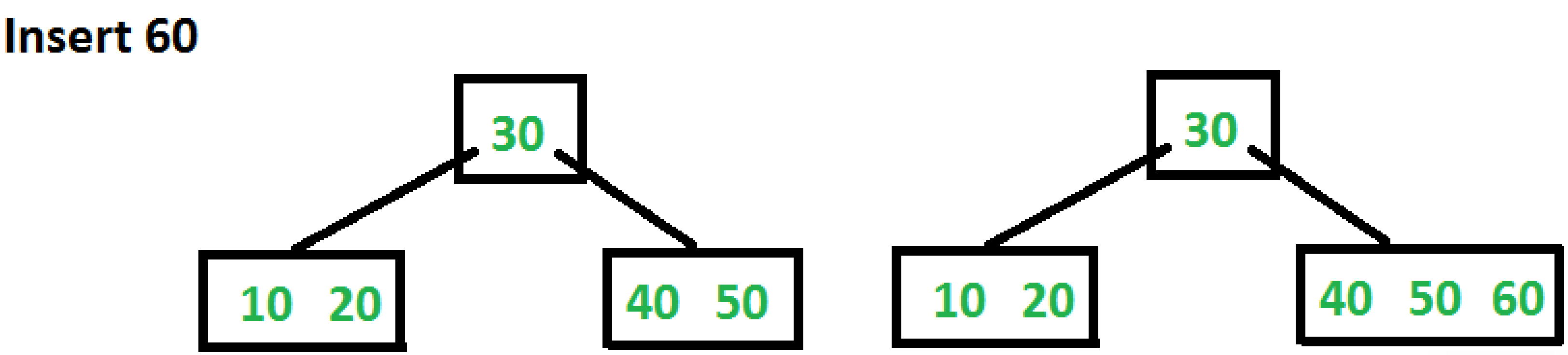
B 树 代码示例-Insert函数

```
void BTreeNode::splitChild(int i, BTreeNode *y){
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;
    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++) z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false){
        for (int j = 0; j < t; j++) z->C[j] = y->C[j+t];}

    // Reduce the number of keys in y
    y->n = t - 1;
    // create space of new child
    for (int j = n; j >= i+1; j--) C[j+1] = C[j];
    // Link the new child to this node
    C[i+1] = z;
    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--) keys[j+1] = keys[j];
    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];
    // Increment count of keys in this node
    n = n + 1;
}
```

B 树 先分裂再插入

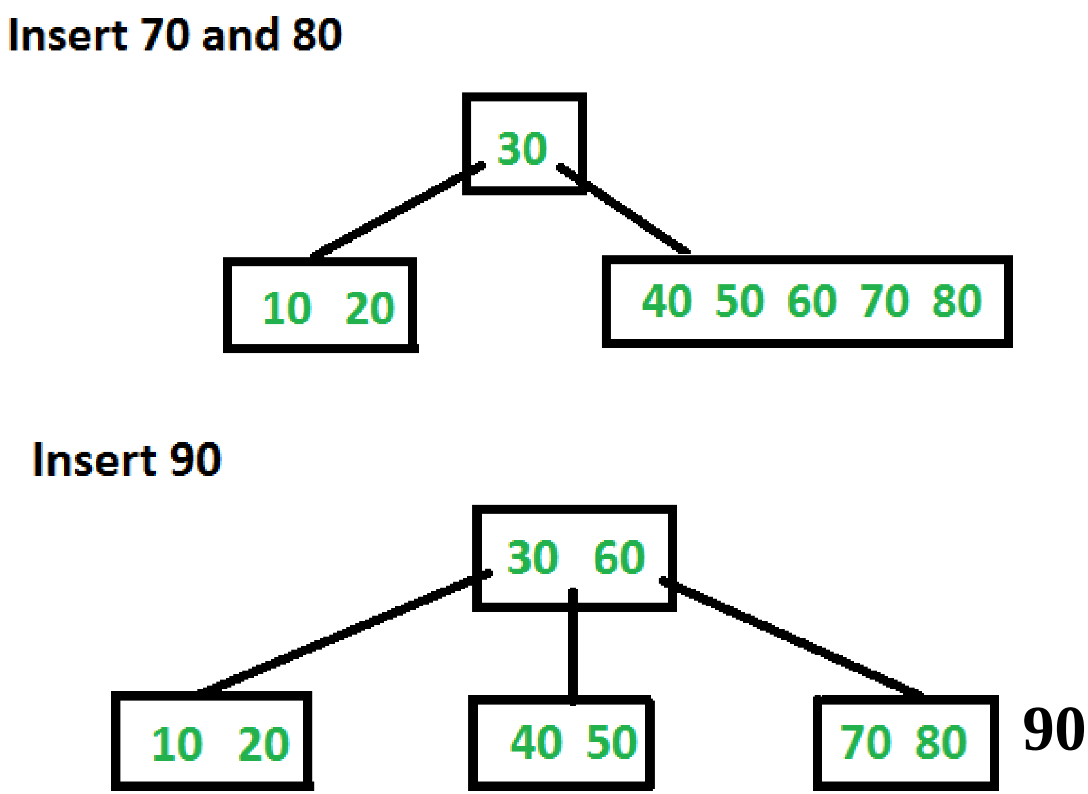
‘t’ as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.





B 树 先分裂再插入

't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.



B 树

分裂与插入的顺序有影响么？

按序插入{1，10，30，50，60，70，16，24，25}

接着插入{80，90}

B 树 删除

在 B 树上删除关键字 k 的过程分两步完成：

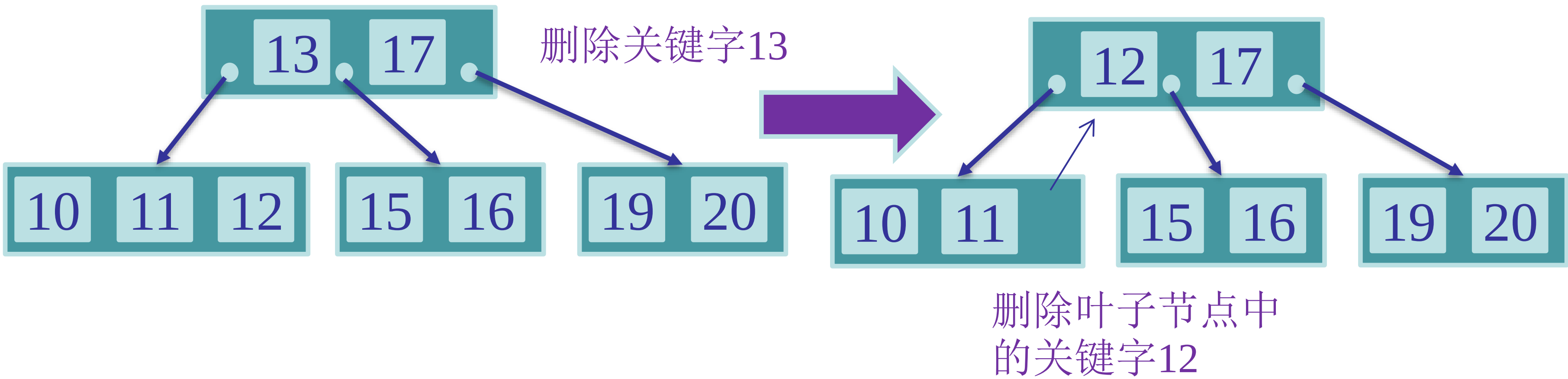
1. 查找关键字 k 所在的节点
2. 删除关键字 k
3. 调整树使其满足 B树的约束条件

删除关键字 k 分两种情况：

1. 在叶子节点层上删除关键字 k
2. 在非叶子节点层上删除关键字 k

B 树 删除

在非叶子节点上删除关键字 k $\Rightarrow$ 在叶子节点上删除关键字 k

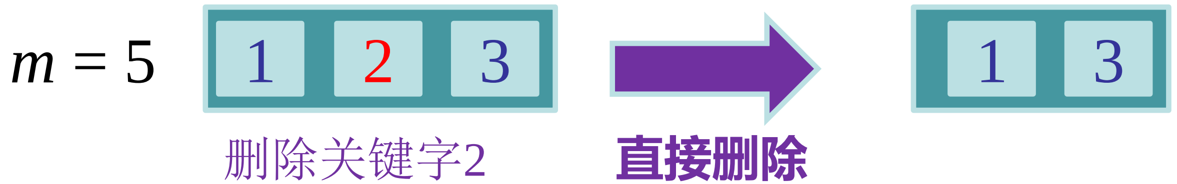


删除元素后，首先判断该元素是否有左右子节点， 如果有，则上移子节点中的某相近元素(「左孩子最右边的节点」或「右孩子最左边的节点」)到父节点中

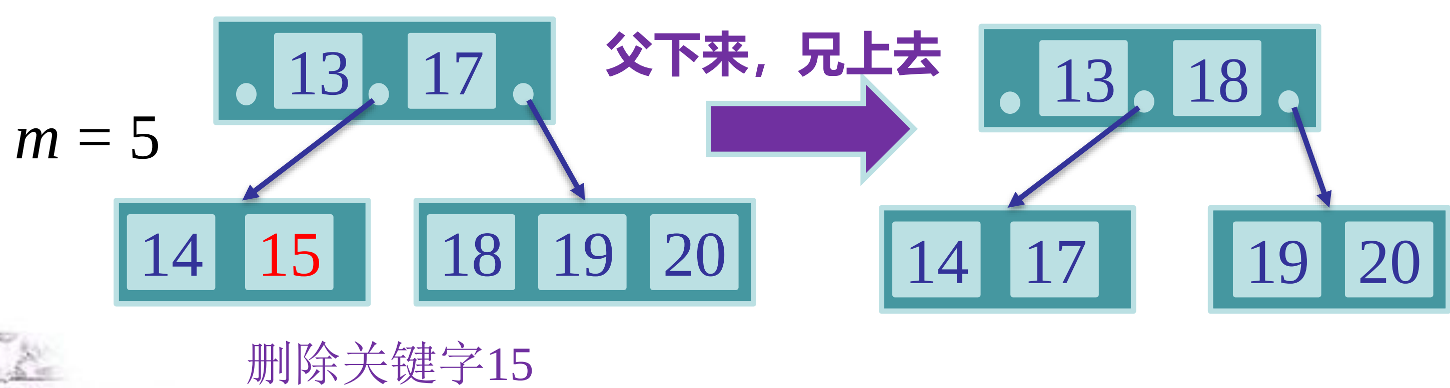
B 树 删除

在 B 树的叶子节点上删除关键字共有以下3种情况：

1. 假如叶子节点的关键字个数大于$\lceil m/2\rceil - 1$，说明删去该关键字后该节点仍满足 B 树的定义，则可直接删去该关键字



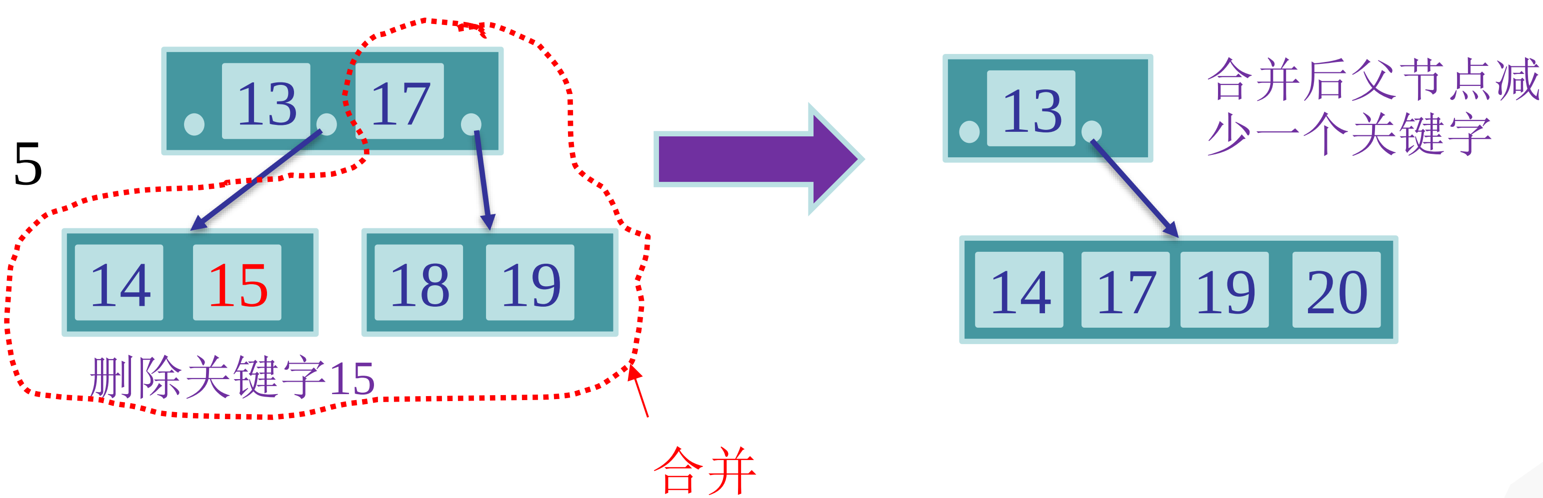
2. 假如叶子节点的关键字个数等于$\lceil m/2\rceil - 1$，说明删去该关键字后该节点不满足 B 树的定义，若可以从兄弟节点借。



B 树 删除

3. 假如叶子节点的关键字个数等于 $\lceil m/2 \rceil - 1$，说明删去该关键字后该节点不满足 B 树的定义，若不可以从兄弟节点借，即兄弟节点关键字个数等于$\lceil m/2 \rceil - 1$，该节点与其相邻的某一兄弟节点进行合并成一个节点

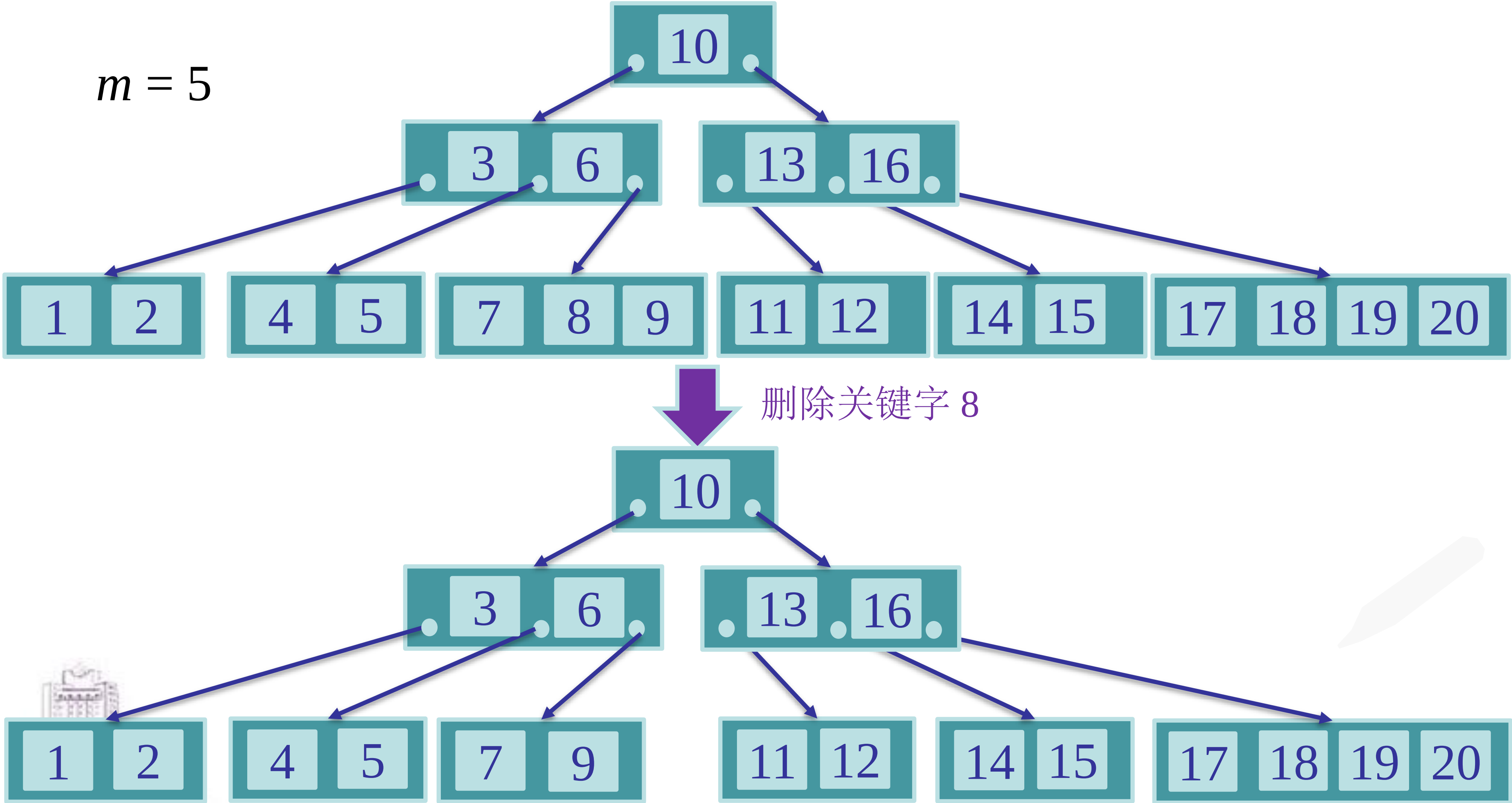
$m = 5$



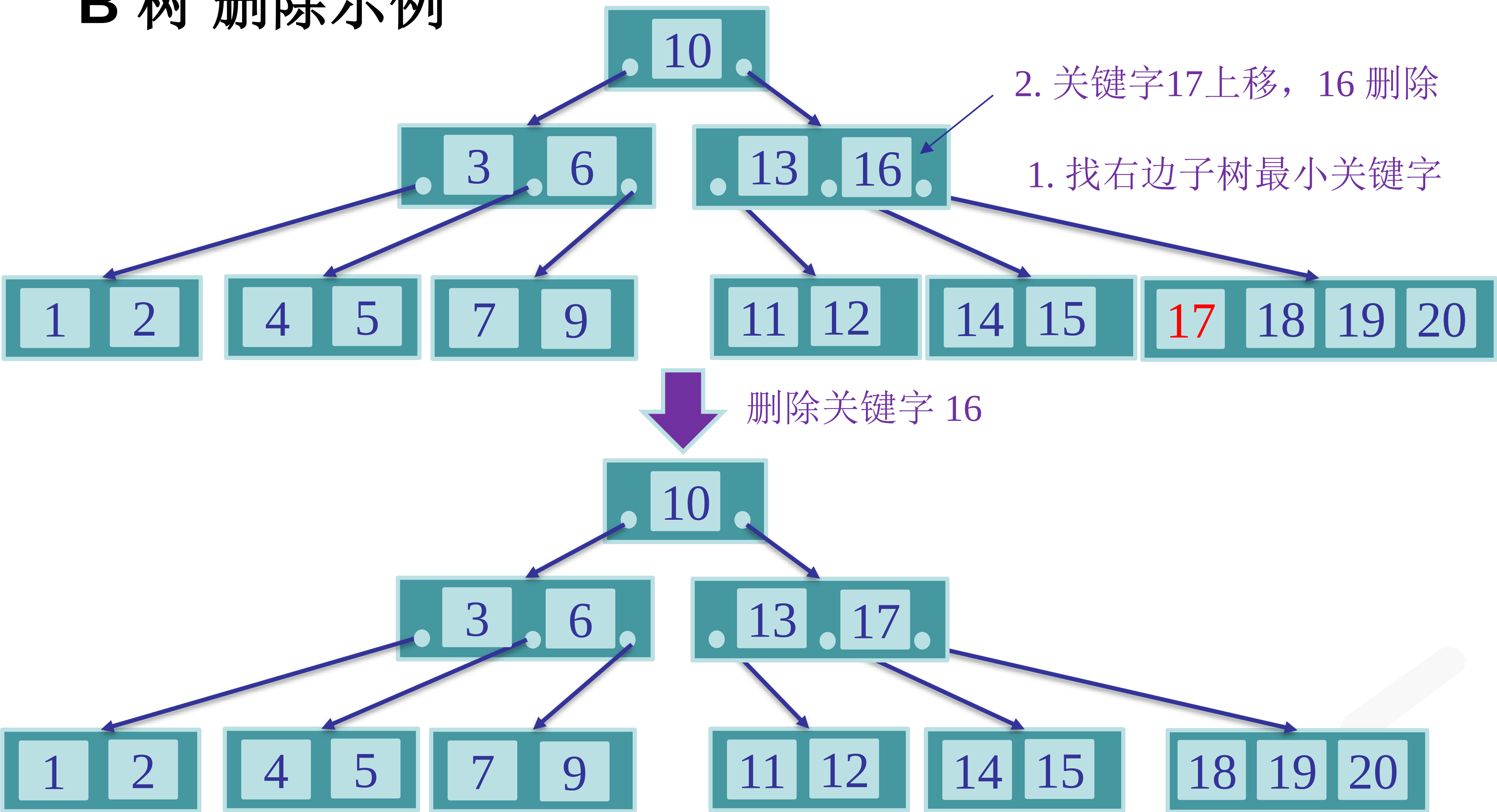
B树 删除示例

对于前例生成的 B 树，给出删除8，16，15，4等 4个关键字的过程

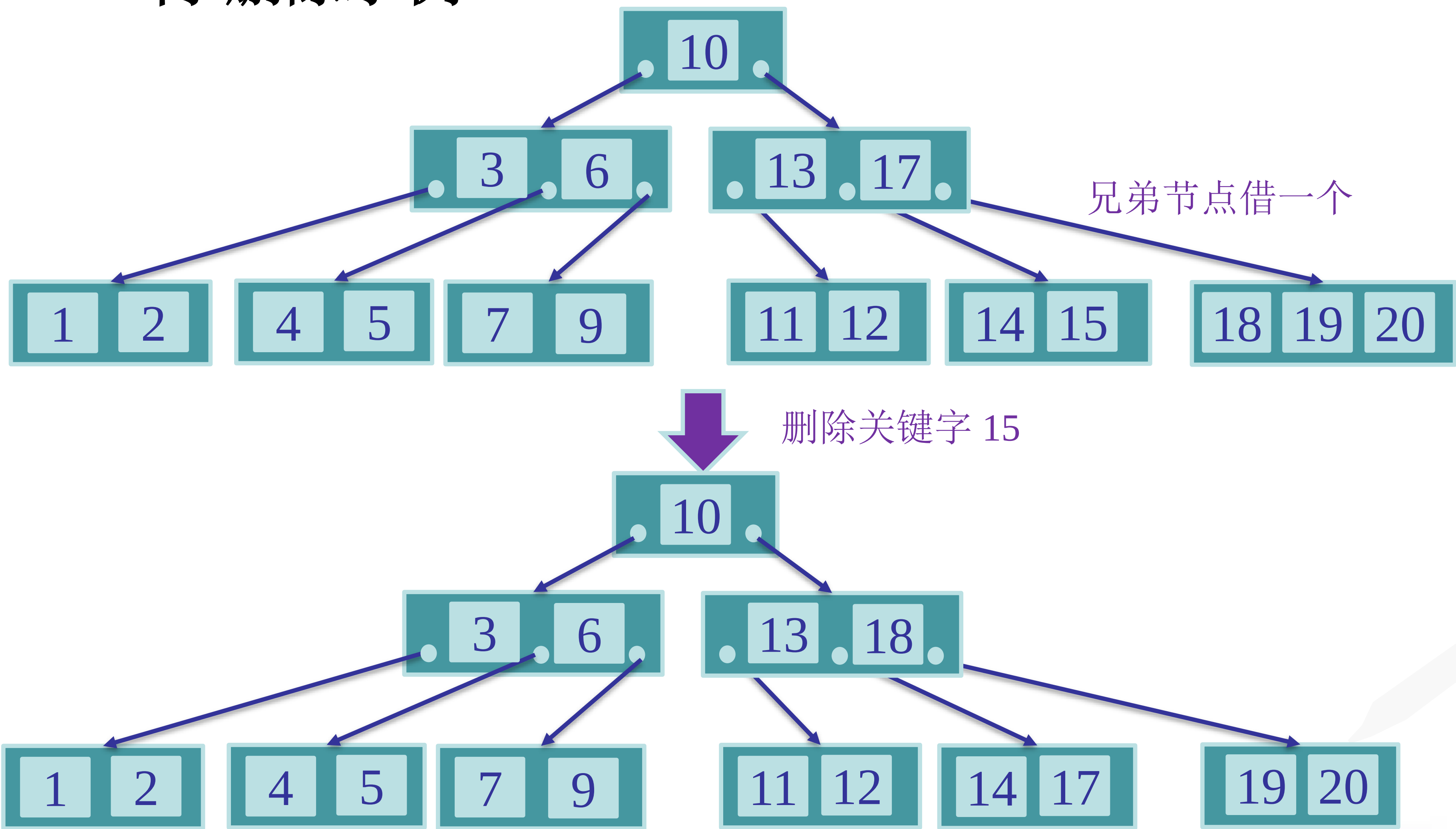
$m = 5$



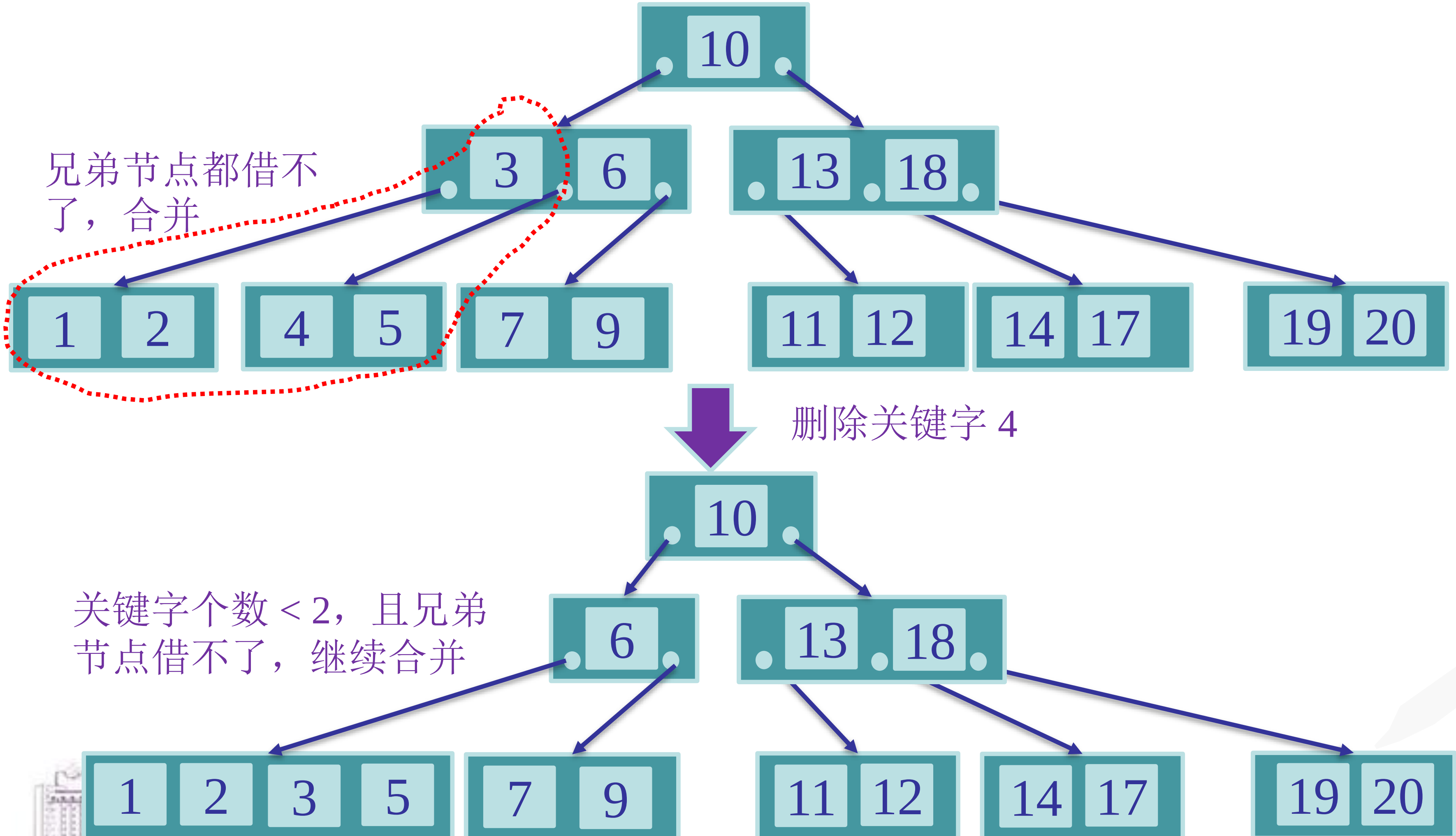
B 树 删除示例


10
3
6
13
16
2. 关键字17上移，16 删除
1. 找右边子树最小关键字
1 2
4 5
7 9
11 12
14 15
17 18 19 20
删除关键字 16
10
3
6
13
17
1 2
4 5
7 9
11 12
14 15
18 19 20

B 树 删除示例

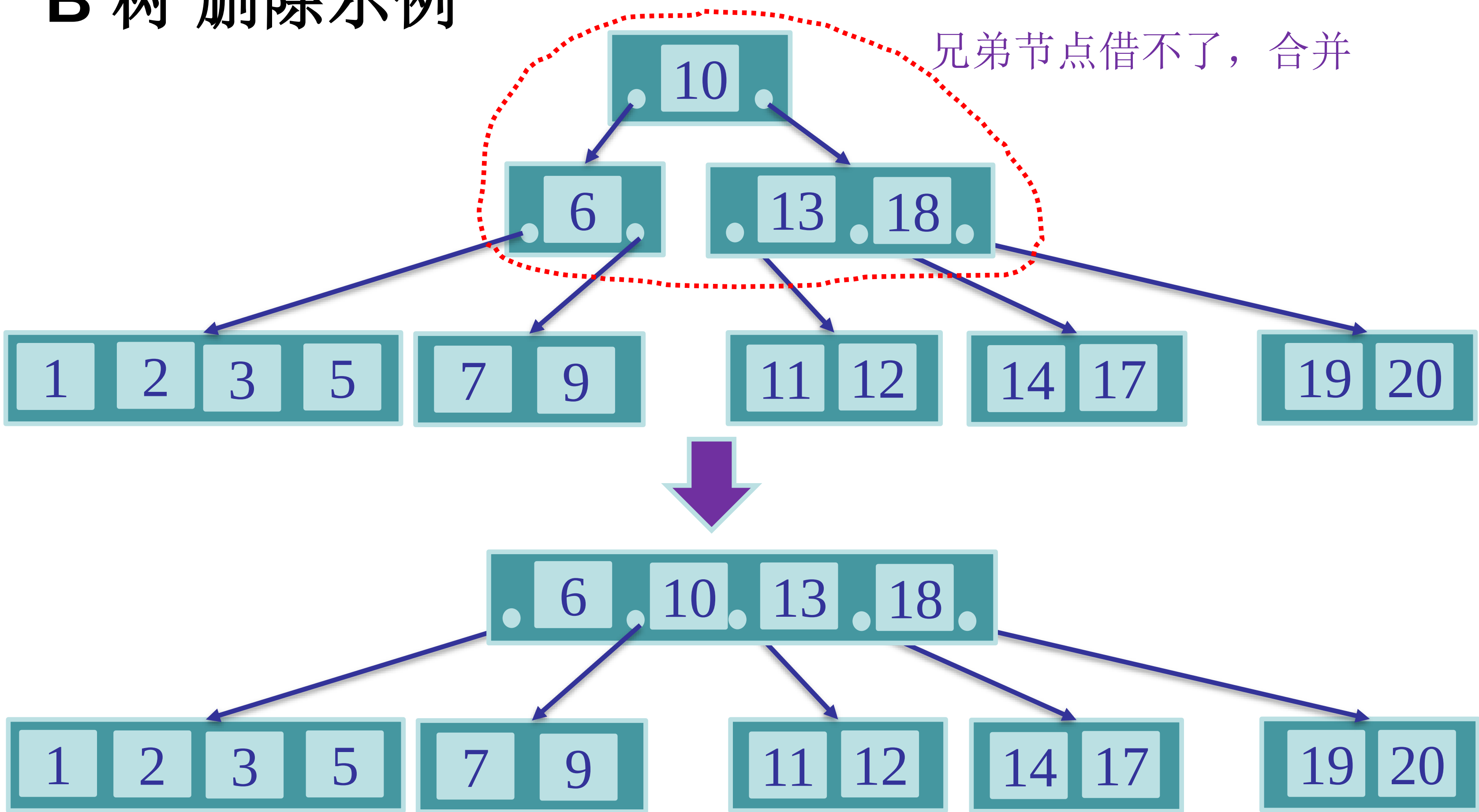

10
3 6
13 17
兄弟节点借一个
1 2
4 5
7 9
11 12
14 15
18 19 20
删除关键字 15
10
3 6
13 18
1 2
4 5
7 9
11 12
14 17
19 20

B 树 删除示例



10
3 6
13 18
兄弟节点都借不了，合并
1 2
4 5
7 9
11 12
14 17
19 20
删除关键字 4
10
6
13 18
关键字个数 < 2，且兄弟节点借不了，继续合并
1 2 3 5
7 9
11 12
14 17
19 20

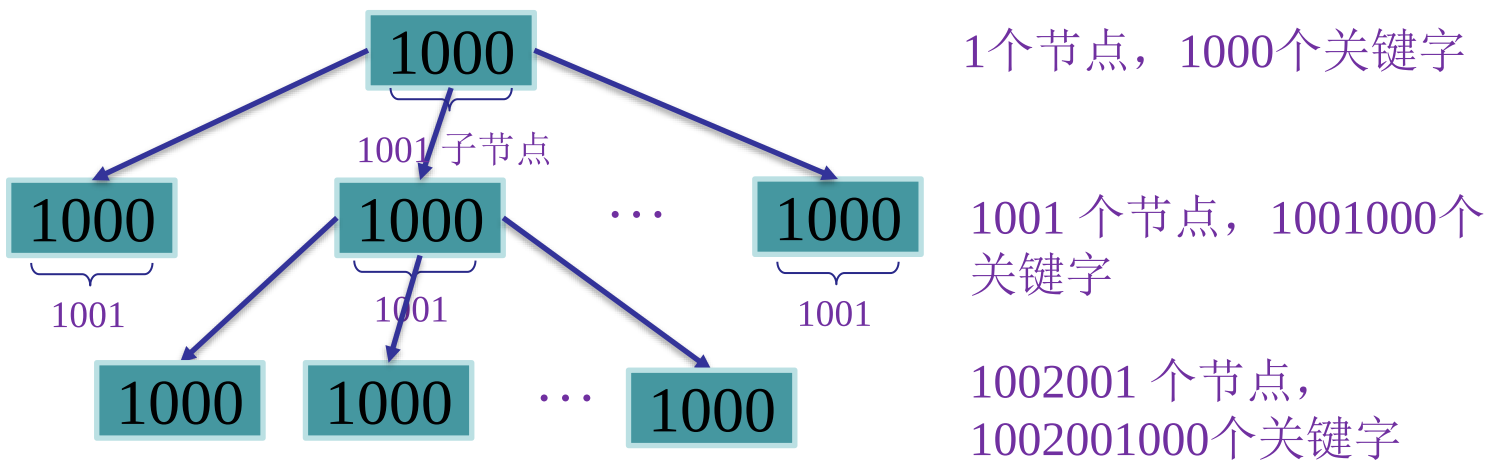
B 树 删除示例



兄弟节点借不了，合并
10
6
13
18
1
2
3
5
7
9
11
12
14
17
19
20
6
10
13
18
1
2
3
5
7
9
11
12
14
17
19
20

B 树 优势

下图为一颗1001阶，高度为2的 B 树，它可以存储超过10亿个关键字



考虑在磁盘中存储数据的情况，与内存相比，读写磁盘有以下不同点：
1.读写磁盘的速度相比内存读写慢很多。
2.每次读写磁盘的单位要比读写内存的最小单位大很多。

由于读写磁盘的这个特点，因此对应的数据结构应该尽量的满足局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用，为了满足局部性原理，所以应该将逻辑上相邻的数据在物理上也尽量存储在一起。这样才能减少读写磁盘的数量。

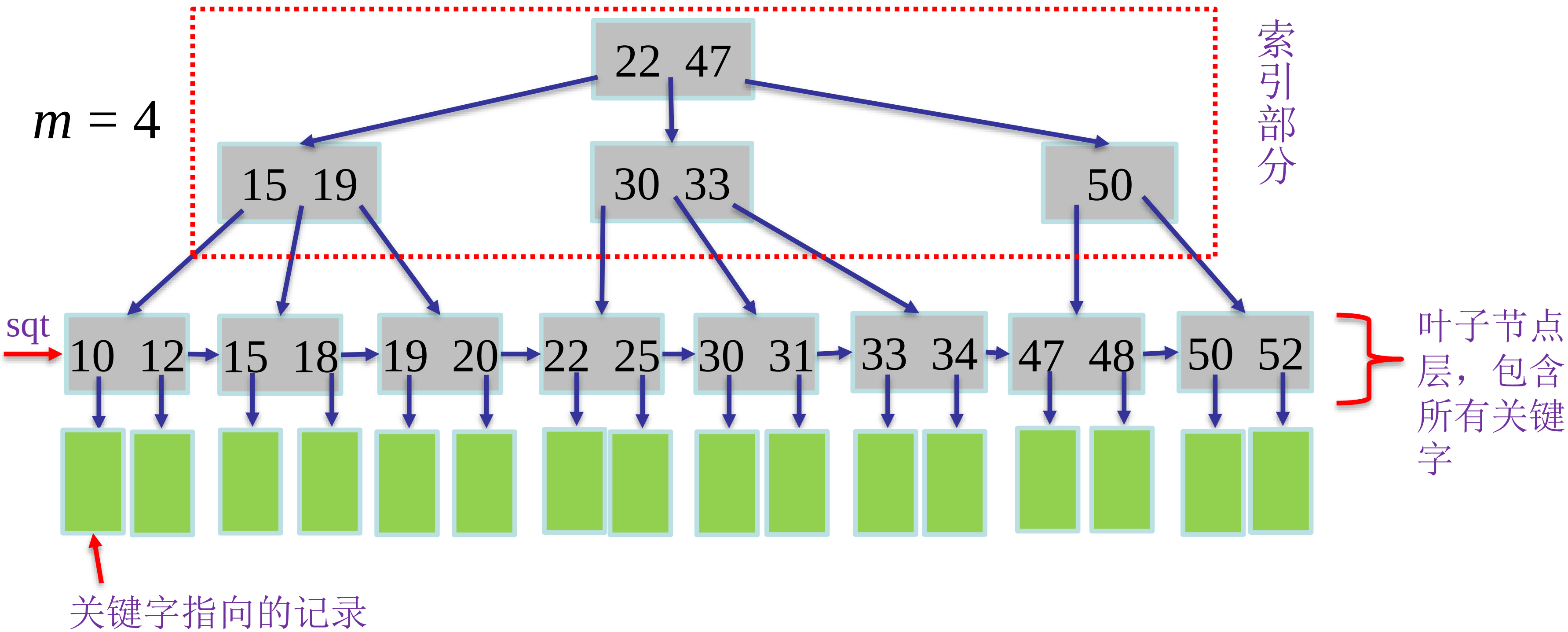
(4) B+ 树（B+ tree)

一棵 m 阶 B+ 树和 B 树的差异在于：

1. 有 n 棵子树的节点中含有 n -1 个关键字(即将区间分为 n 个子区间，每个子区间对应一棵子树)。
2. 所有叶子节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，且叶子节点本身依关键字的大小自小而大顺序链接。
3. 所有的非叶子节点可以看成是索引部分，节点中仅含有其子树（根节点）中的最大（或最小）关键字。
4. B+ 树各个节点类型的约束条件如下：

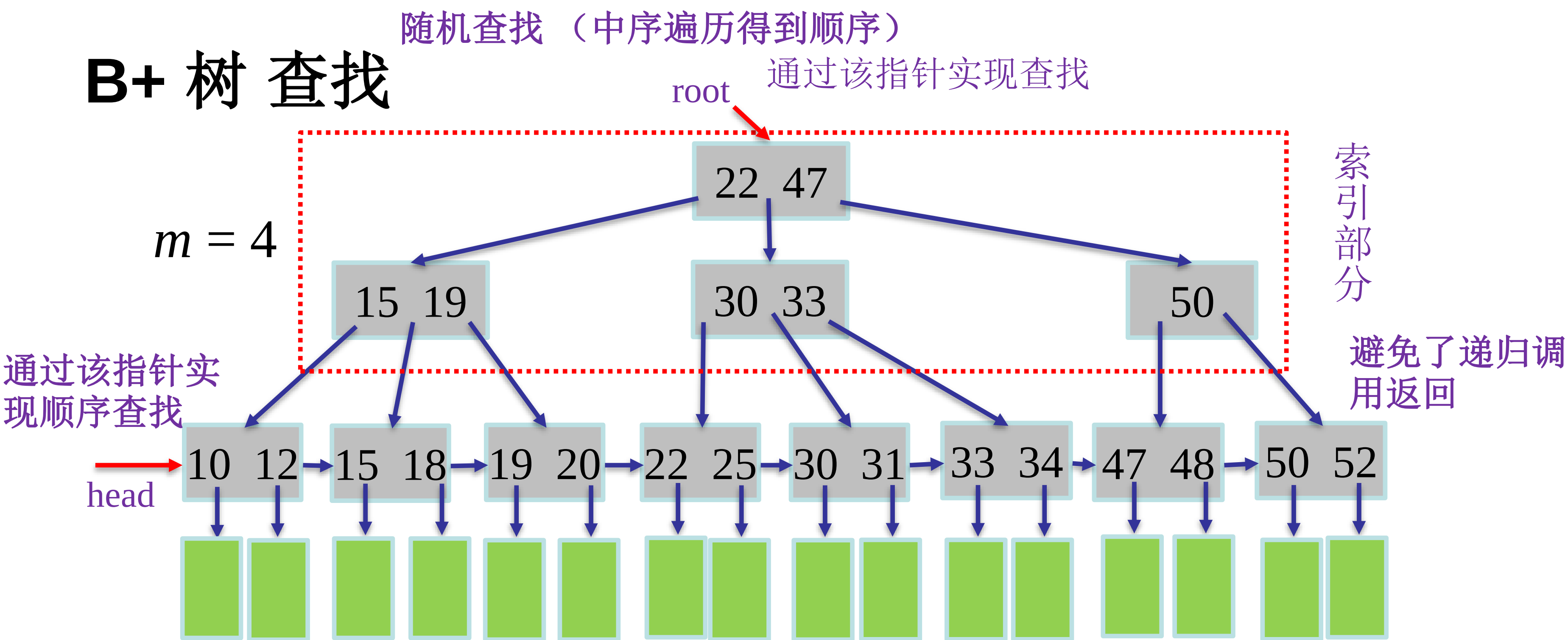
节点类型	最小关键字个数	最大关键字个数	最小子节点个数	最大子节点个数
根节点(为叶子节点)	0	$m-1$	0	0
根节点(为内部节点)	1	$m-1$	2	m
内部节点	$\lceil m/2 \rceil$ - 1	$m-1$	$\lceil m/2 \rceil$	m
叶子节点	$\lceil \boldsymbol{m/2} \rceil$	$m-1$	0	0

B+ 树相比于B 树的优势



1. 索引节点上只有索引而没有数据，所以索引节点上能存储比 B 树更多的索引，这样树的高度就会更矮。树的高度越矮，磁盘寻道的次数就会越少
2. 数据都集中在叶子节点，而所有叶子节点的高度相同，那么可以在叶子节点中增加前后指针，指向同一个父节点的相邻兄弟节点，这样可以更好地支持查询一个值的前驱或后继，使连续访问更容易实现。

B+ 树 查找



B+ 树的查找过程与 B 树类似，假设需要查找的键值是 k ，那么从根节点开始，从上到下递归地遍历树。在每一层上，搜索的范围被减小到包含搜索值的子树。

需要注意的是，在查找时，若非叶子节点上的关键字等于给定值，并不终止，而是继续向下直到叶子节点。因此，在 B+ 树中，不管查找成功与否，每次查找都是走了一条从根到叶子节点的路径。其余同 B 树的查找类似。

select * from tbl where t > 10

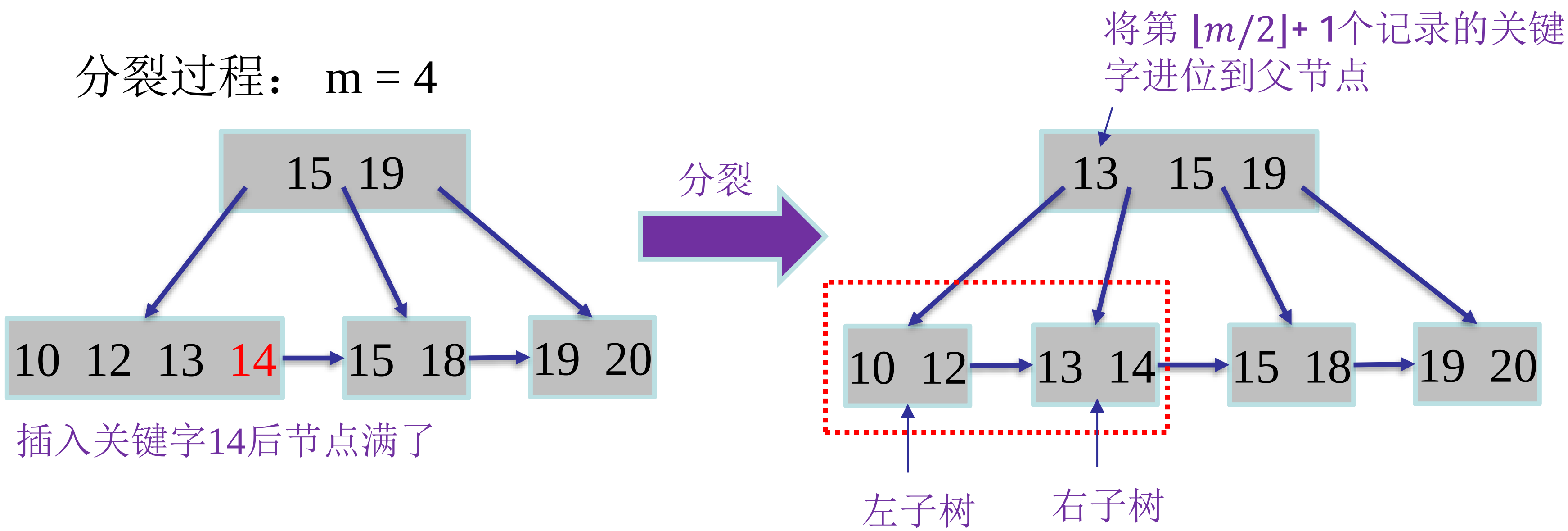
B+ 树 插入

将关键字 k 插入到 B+ 树的过程如下：

1. 若为空树，创建一个叶子节点，然后将记录插入其中，此时这个叶子节点也是根节点，插入操作结束。
2. 若不是空树，查找该关键字的插入节点(注意 B+ 树的插入节点一定是叶子节点层的节点)
3. 插入关键字，若当前节点节点中(父节点一定是索引类型节点)，执行步骤 4关键字的个数等于 m，将这个叶子节点分裂成左右两个叶子节点，将第 $\lfloor m/2 \rfloor$+ 1 个记录的关键字进位到父节点
4. 针对索引类型节点(内部节点)：若当前节点关键字的个数大于m -1，将这个索引类型节点分裂成两个索引节点，将第 $\lfloor m/2 \rfloor$个关键字进位到父节点中，重复这一步

B+ 树 插入

分裂过程： m = 4



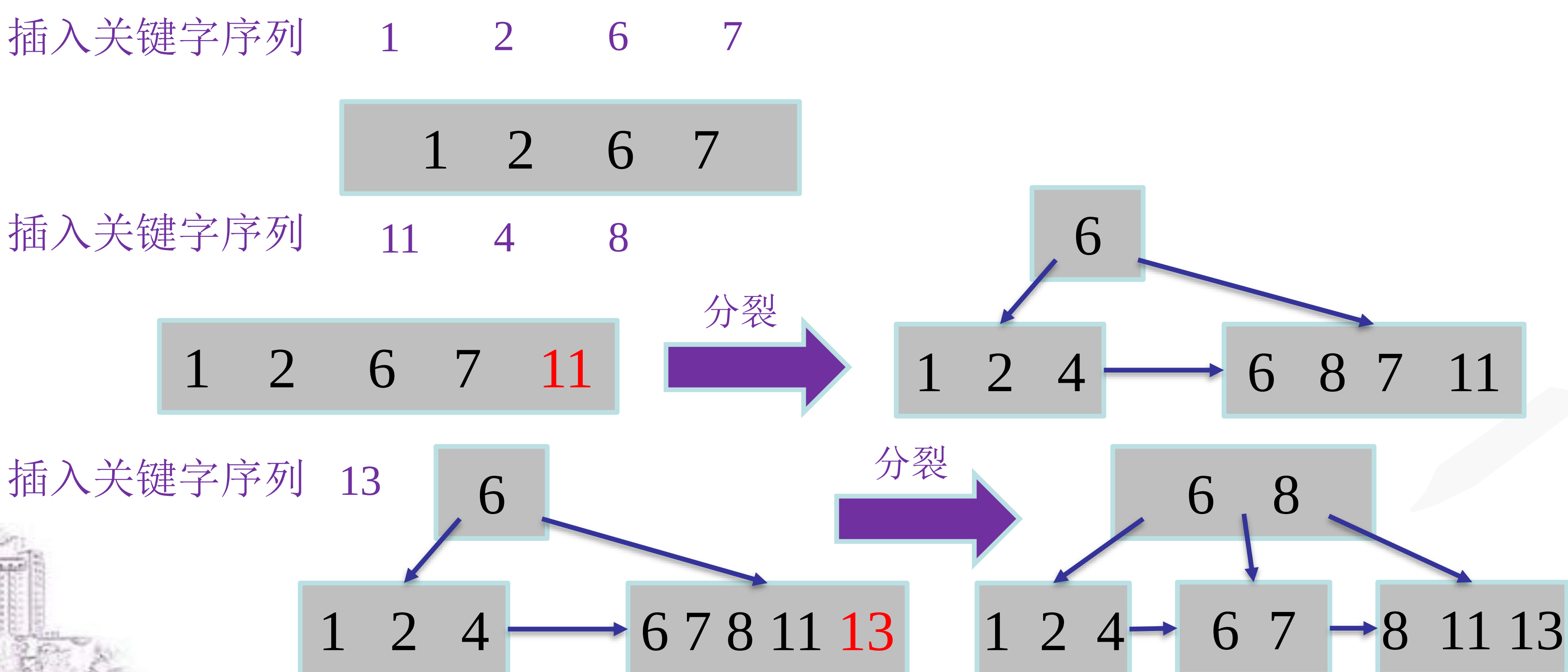
[10,12,13,14] 分裂成 [10,12]，[13,14]，因此需要在这两个节点之间新增一个索引值，这个值应该满足：

- 大于左子树的最大值
- 小于等于右子树的最小值

B+ 树 插入示例

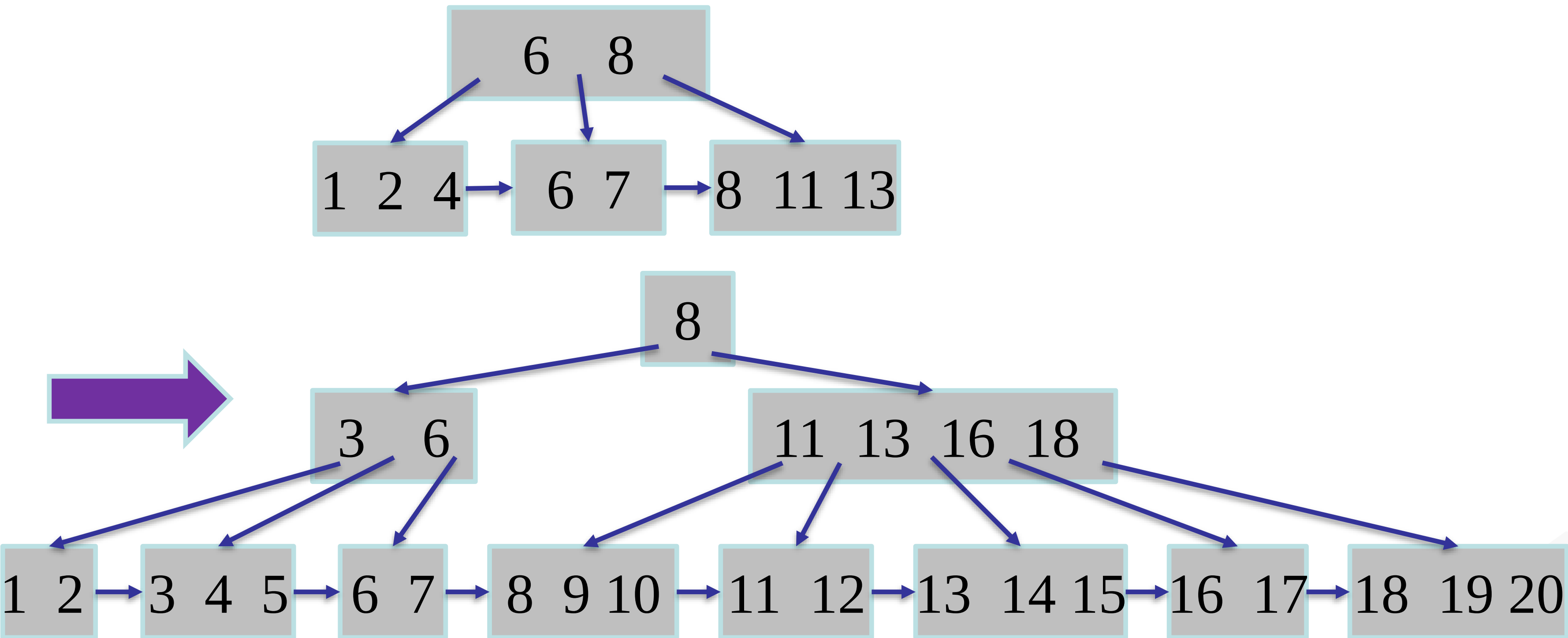
关键字序列为：(1，2，6，7，11，4，8，13，10，5，17，9，16，20，3， 12，14，18，19，15)，创建一棵 5 阶 B+ 树。

节点最多关键字个数为 m -1 = 4



B+ 树 插入示例

插入关键字序列 10 5 17 9 16 20 3 12 14 18 19 15



B+ 树 删除

B+ 树的删除也仅在叶子节点中进行，当叶子节点中的最大关键字被删除时，其在非叶子节点中的值可以作为一个分界关键字存在。若因删除而使节点中关键字的个数少于 $\lceil m/2 \rceil$ - 1 时，其和兄弟节点的合并过程与 B 树类似。

在 B+ 树上删除关键字 k 的过程分如下步骤：

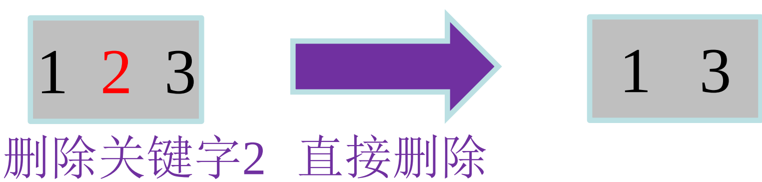
1. 查找关键字 k 所在的叶子节点，删除该叶子节点的数据。
2. 如果删除叶子节点之后的数据数量，满足 B+ 树的平衡条件，则直接返回。
3. 删除不满足 B+ 树的平衡条件就需要做平衡操作：如果该叶子节点的左右兄弟节点的数据量可以借用，就借用过来满足平衡条件。否则，就与相邻的兄弟节点合并成一个新的子节点了。
4. 若删除操作导致父节点也会不平衡，那么就按照前面的步骤也对父节点进行重新平衡操作，这样一直到某个节点平衡为止。

B+ 树 删除

在 B+ 树的叶子节点上删除关键字共有以下3种情况：

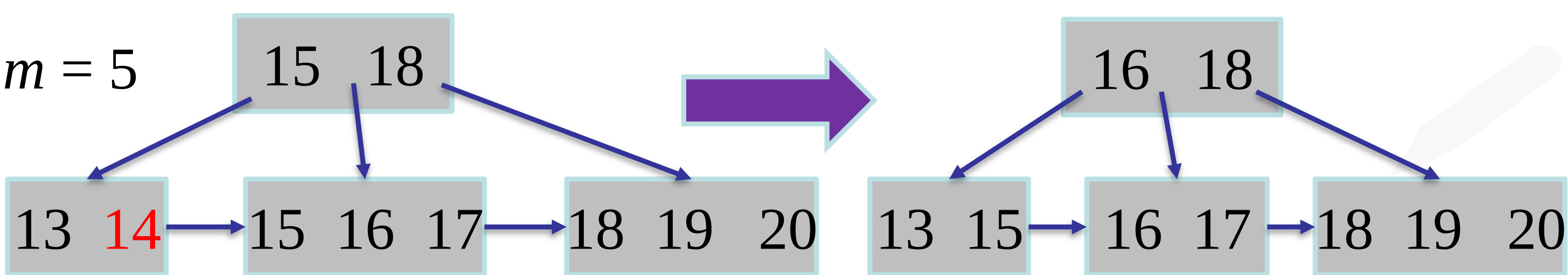
1. 假如叶子节点的关键字个数大于$\lceil m/2 \rceil - 1$，说明删去该关键字后该节点仍满足 B 树的定义，则可直接删去该关键字

$m = 5$



2. 假如叶子节点的关键字个数等于$\lceil m/2 \rceil - 1$，说明删去该关键字后该节点不满足 B 树的定义，若可以从兄弟节点借。

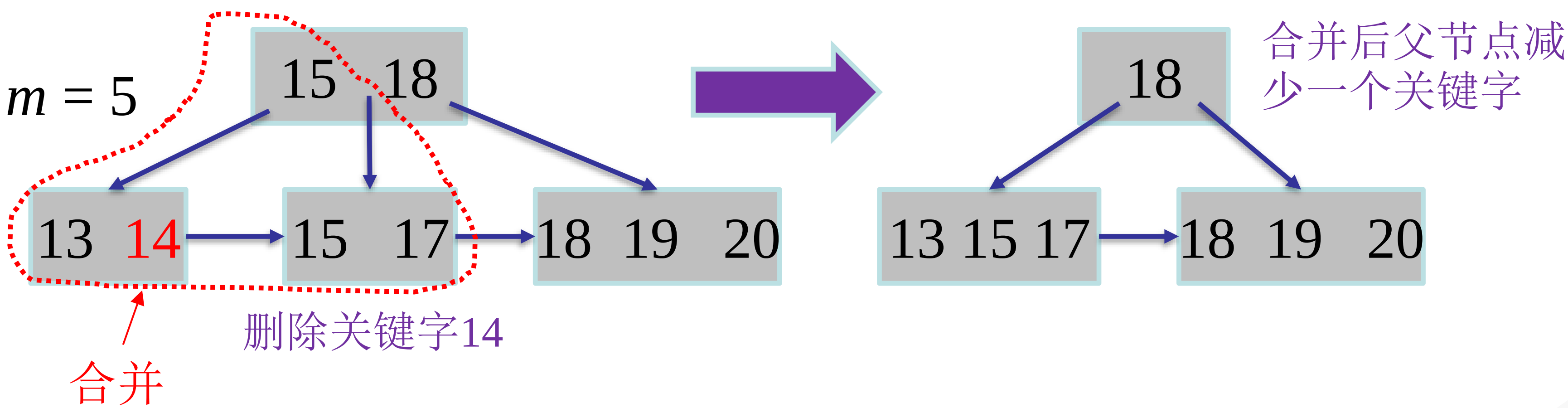
$m = 5$



删除关键字14

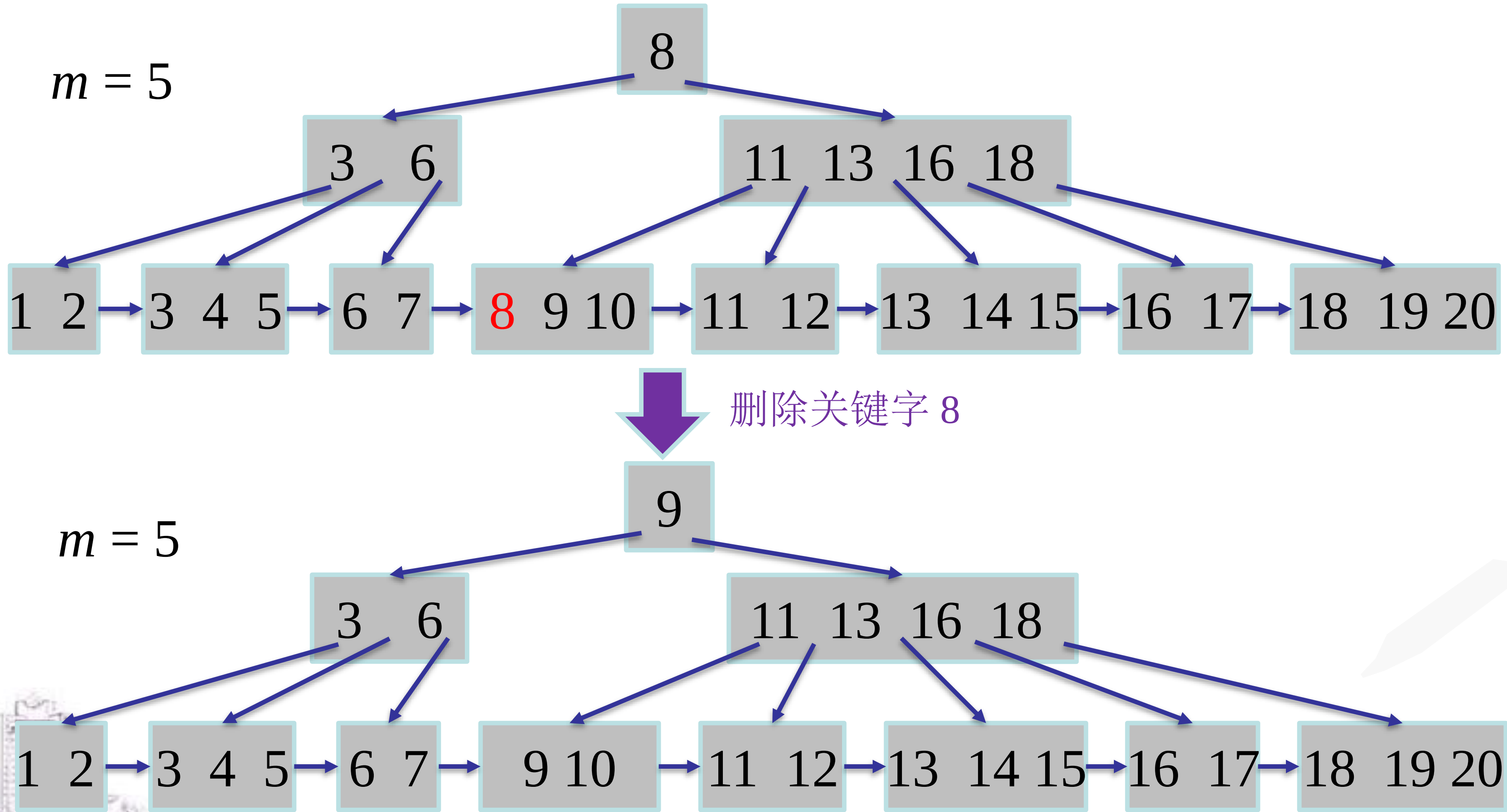
B+ 树 删除

3. 假如叶子节点的关键字个数等于 $\lceil m/2 \rceil - 1$，说明删去该关键字后该节点不满足 B 树的定义，若不可以从兄弟节点借，即兄弟节点关键字个数等于$\lceil m/2 \rceil - 1$，该节点与其相邻的某一兄弟节点进行合并成一个节点

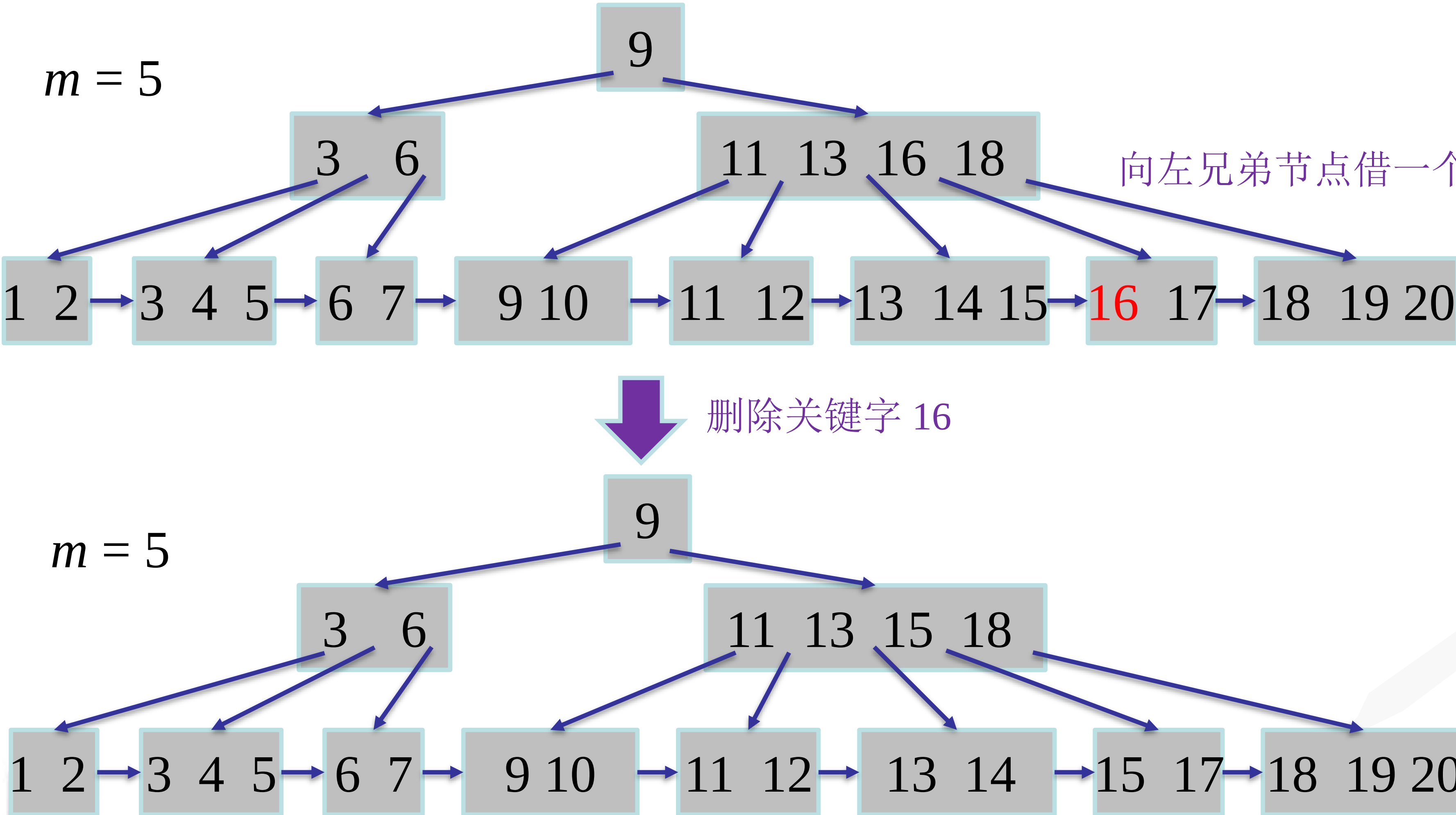


B+ 树 删除示例

对于前例生成的 B+ 树，给出删除8，16，15，9等 4个关键字的过程

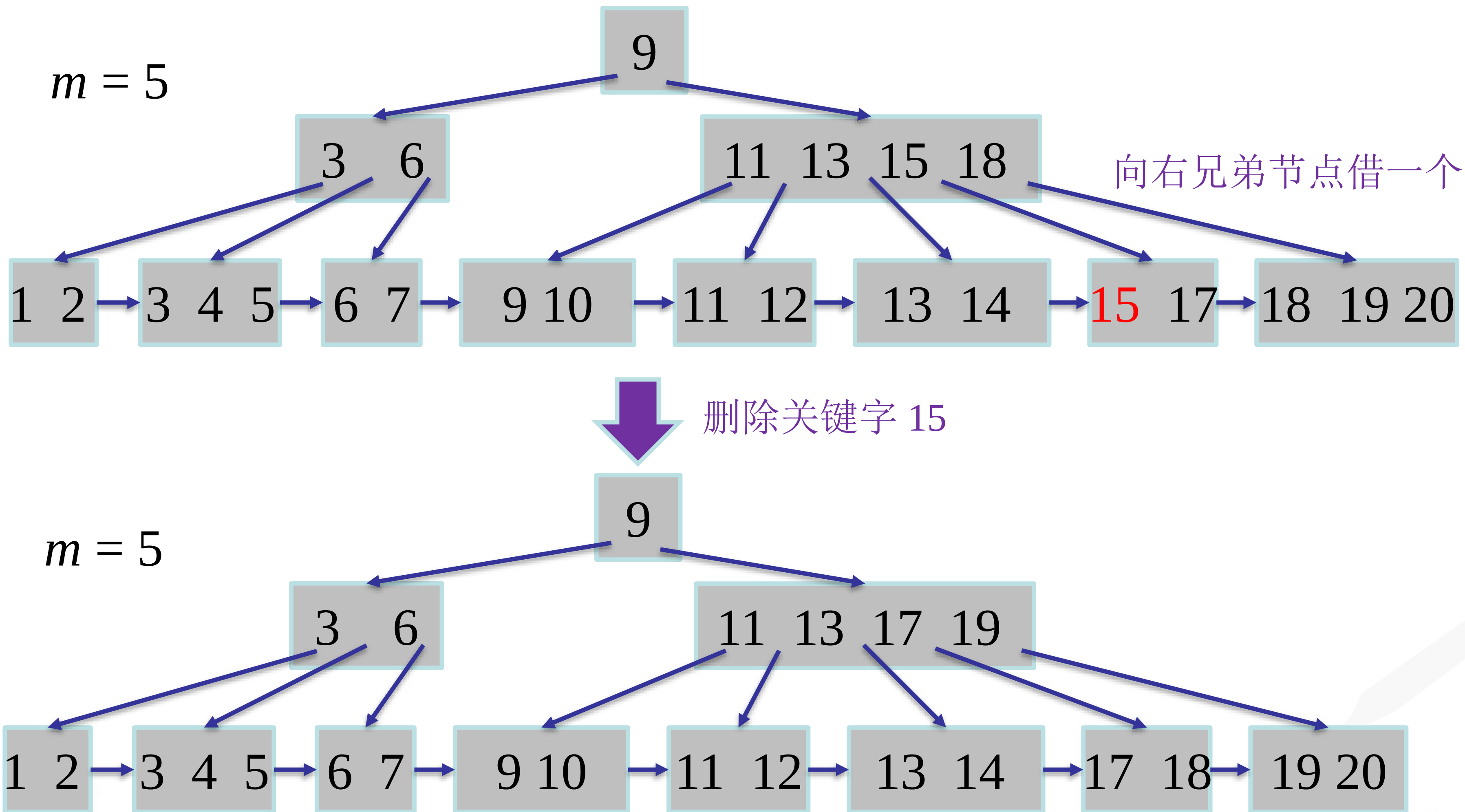


B+ 树 删除示例



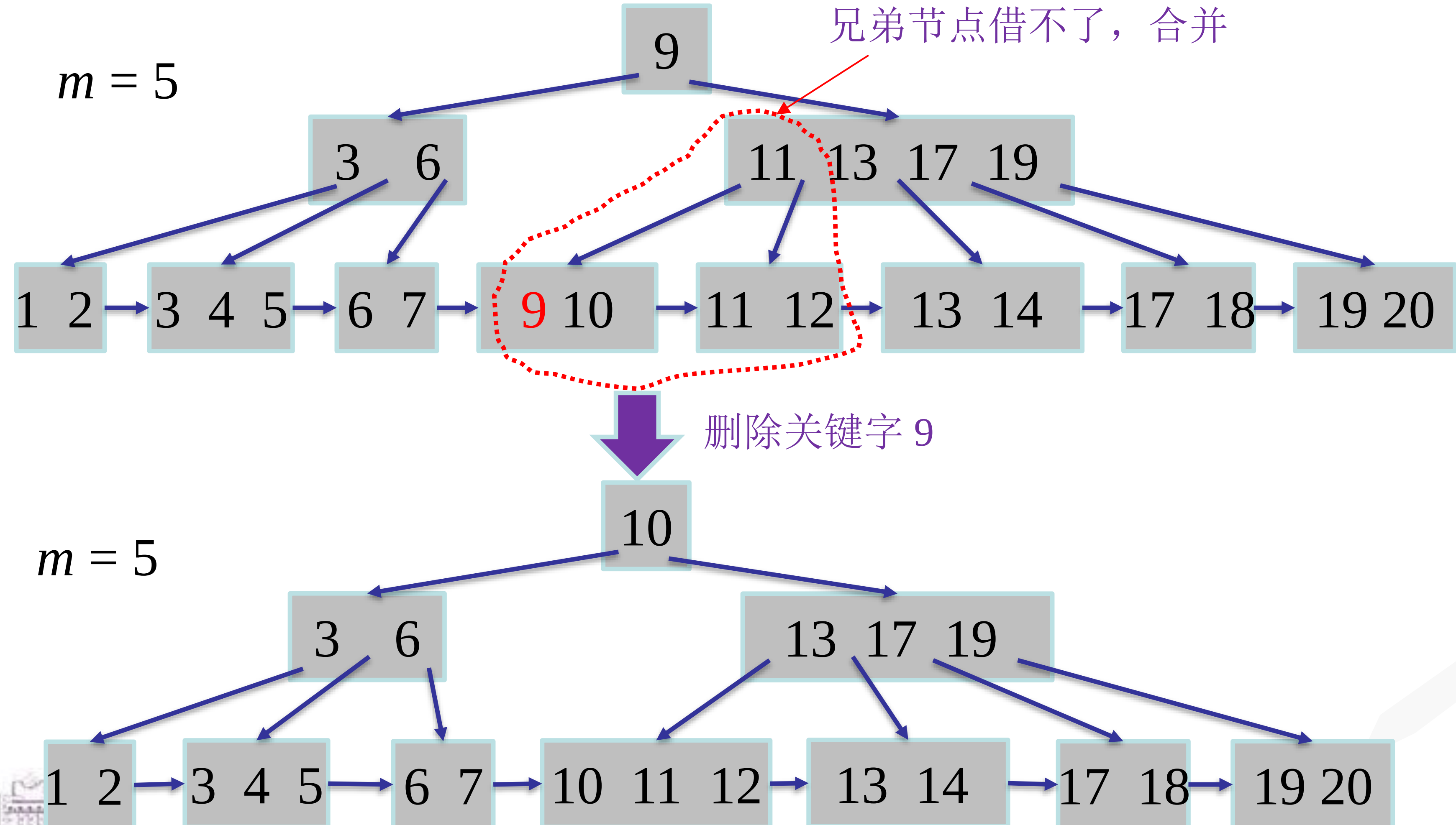
m = 5
9
3 6
11 13 16 18
向左兄弟节点借一个
1 2
3 4 5
6 7
9 10
11 12
13 14 15
16 17
18 19 20
删除关键字 16
m = 5
9
3 6
11 13 15 18
1 2
3 4 5
6 7
9 10
11 12
13 14
15 17
18 19 20

B+ 树 删除示例



$m = 5$
9
3 6
11 13 15 18
向右兄弟节点借一个
1 2
3 4 5
6 7
9 10
11 12
13 14
15 17
18 19 20
删除关键字 15
$m = 5$
9
3 6
11 13 17 19
1 2
3 4 5
6 7
9 10
11 12
13 14
17 18
19 20

B+ 树 删除示例



m = 5
9
兄弟节点借不了，合并
3 6
11 13 17 19
1 2
3 4 5
6 7
9 10
11 12
13 14
17 18
19 20
删除关键字 9
m = 5
10
3 6
13 17 19
1 2
3 4 5
6 7
10 11 12
13 14
17 18
19 20

总结：B^+树的特点 （VS. B树）

- B^+树的键可能出现多次。B树中同一键值不会出现多次，并且它有可能出现在叶结点，也有可能出现在非叶结点中。而B^+树的键一定会出现在叶结点中，并且有可能在非叶结点中也有可能重复出现，以维持B^+树的平衡。
- B^+树插入或删除的位置必定在叶子节点上。因为B树键位置不定，且在整个树结构中只出现一次，虽然可以节省存储空间，但使得在插入、删除操作复杂度明显增加。B^+树相比来说是一种较好的折中。
- B^+树的查询效率为常数。B树的查询效率与键在树中的位置有关，最大时间复杂度与B^+树相同(在叶结点的时候)，最小时间复杂度为1(在根结点的时候)。而B^+树的查询效率对某建成的树是固定的。

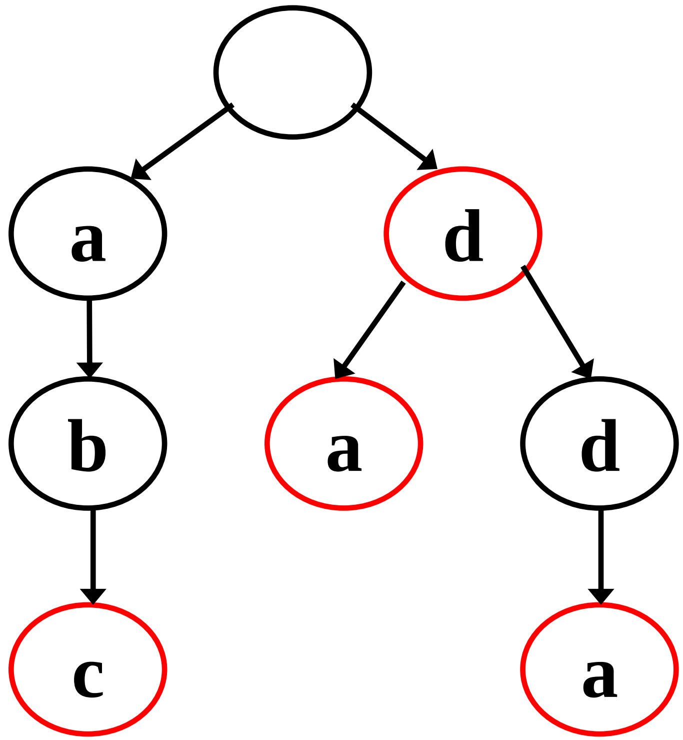
总结：B树和B$^+$树的应用场景

- 磁盘上海量数据的检索；
- 不少数据库都支持B树或B$^+$树算法处理索引。
  - 支持B树的数据库： Sql server ,Oracle及Sysbase
  - 支持B$^+$树的数据库： mysql，Berkeley DB , sqlite
- B$^+$树支持精确检索和模糊检索：
  - 可以用于高效率地执行精确的或者基于范围（使用操作<、<=、=、>=、>、<>、!=和BETWEEN）的比较。
  - 也可以用于LIKE模式匹配，前提是该模式以文字串而不是通配符开头

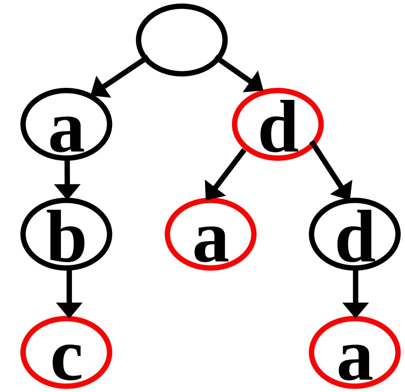
Trie树

- Trie树，又称单词查找树或键树或字典树，是一种树形结构，是一种哈希树的变种。
- 典型应用：用于统计和排序大量字符串等场景中（但不仅限于字符串），所以经常被搜索引擎用于文本词频统计。
- 优点：最大限度地减少无谓的字符串比较，查询效率比较高。
- 核心思想：以空间换时间，利用字符串的公共前缀来降低查询时间的开销，以达到提高效率的目的。
  - Trie树中第i层的结点数为26^i（i≥0）
  - 查找某个字符串的操作的复杂度最多只需O(n)，其中n为字符串的长度。（用空间来换时间）

Trie树（字典树）：示例



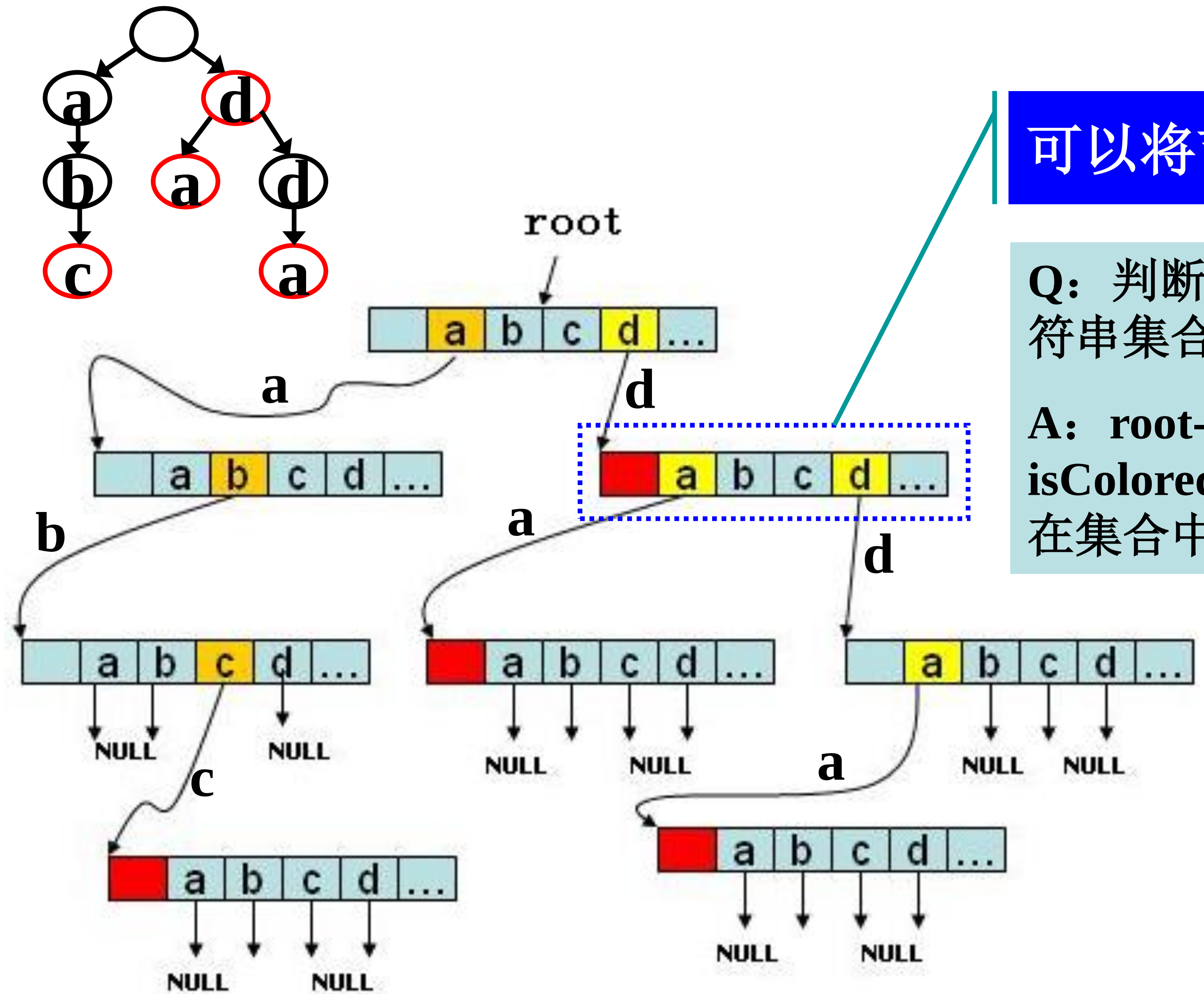
- 在这个Trie结构中，保存了abc、d、da、dda四个字符串，其中红色框表示字符串的结尾字符 。



- **Trie树有3个基本特性：**
  - 1）根节点不包含字符，除根节点外每一个节点都只包含一个字符。
  - 2）从根节点到某一节点的路径上经过的字符连接起来，即为该节点对应的字符串。
  - 3）每个节点的所有子节点包含的字符都不相同（即，互为兄弟的结点上的字符都不同）。

trie树——数据结构的定义

const int kind=26;//字母种类
struct Treenode { //Trie树的结点结构
int count ; //记录遍历到该结点形成的字符串出现的次数，在不同题中可记录不同的内容。
bool isColored; //是否标记为红色,红色表示记录此处是否构成一个串
TrieNode *next[kind];//26个指针，可由字符定位（类似Hash表的定义）
……
}TrieNode
TrieNode * root;



可以将节点看作Hash表

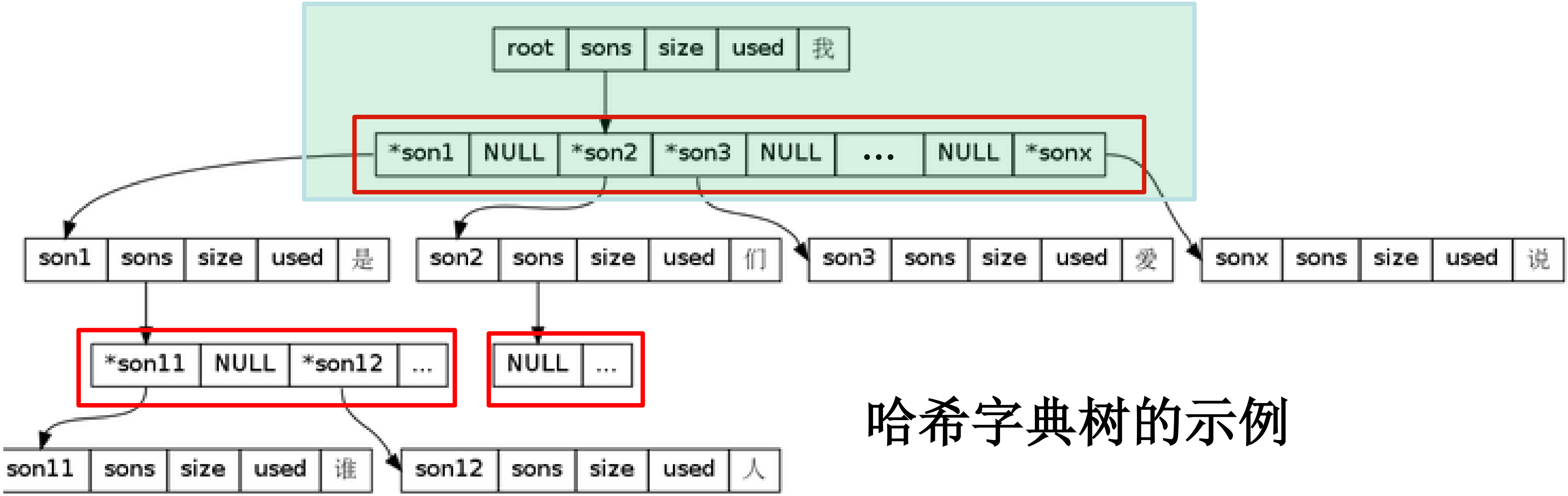
Q：判断字符串”d”是否在该字符串集合中？

A：root->next[‘d’-’a’]->isColored=1，因此字符串”d”在集合中。

- 在这个Trie结构中，保存了abc、d、da、dda四个字符串，如果是字符串会在节点的尾部进行标记。没有后续字符的branch分支指向NULL 。（用空间换取时间）

中文字典树（Trie树 + 哈希表）

- 英文字符只有**26**个大写和**26**个小写字母；而常用的汉字大概有**5**万个，因此，如果中文字典树也用**Trie**的方式存储，就会出现每个节点占用空间太大的问题。
- 在处理中文字符时，对传统的字典树进行改进
  - 使用 **Trie**树**+**哈希表
- **Trie**树中每个节点的子节点用一个哈希表存储，因而，既不用浪费太大的空间，而且查询速度上可以保留哈希的时间复杂度**O(1)**。
- 例如：
  “我”的下一个可能为“们，是，爱，说”，就可以为“我”建立一个大小为**11**的哈希表，作为子节点，将这四个字存入哈希表中。



哈希字典树的示例

- 每个节点包含：一个指针sons(指向哈希表)，一个整数used(表示子节点的个数)，一个素数size(表示哈希表的大小，值为大于2*used的素数)。
- 如上图所示“我”有四个子节点，所以used=4, size=11, sons指向一个大小为11的哈希表，哈希表中存储着子节点的指针或NULL。