数据结构与算法

DATA STRUCTURE

第十九讲 B-Tree

胡浩栋

信息管理与工程学院

2017 - 2018 第一学期

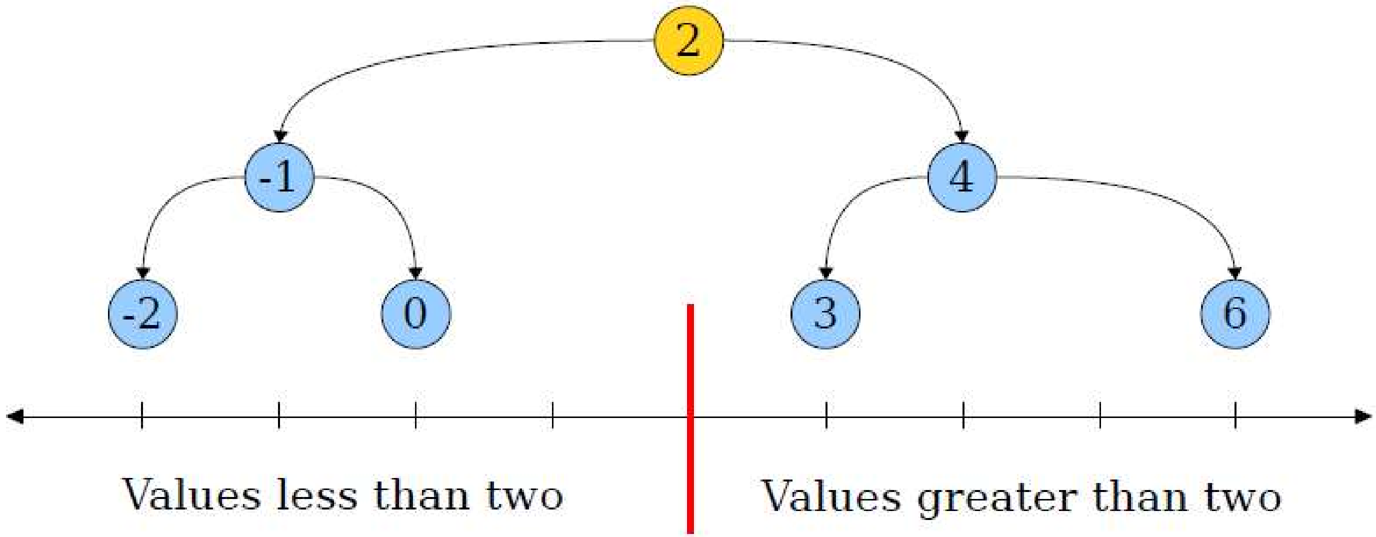
课堂内容

- B树和B+树

B树

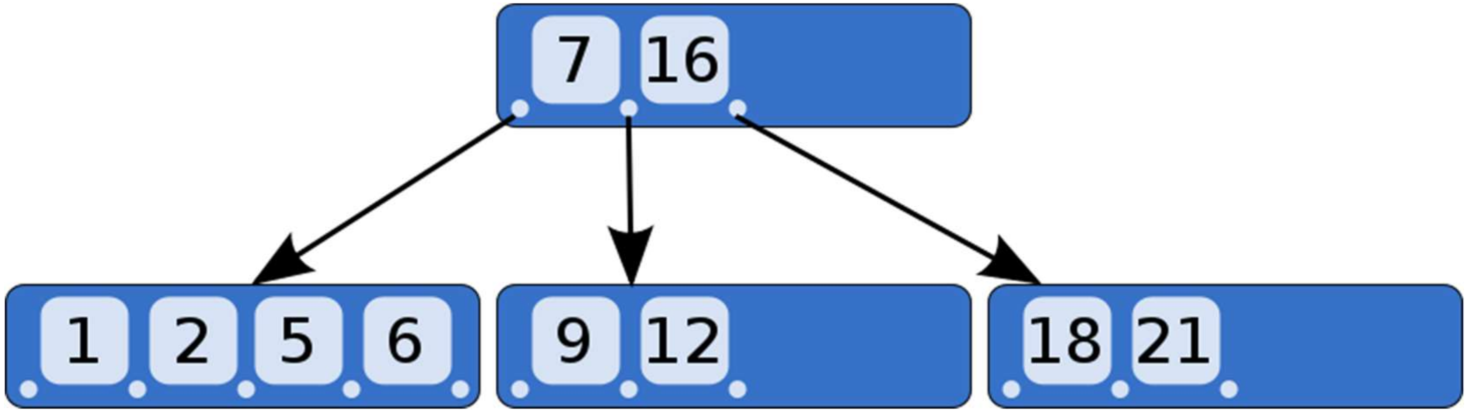
B-Tree

二叉查找树



- 链表推广到二叉树
- 任何节点都保存了一个元素，这个元素key把子树里的节点分成两部分

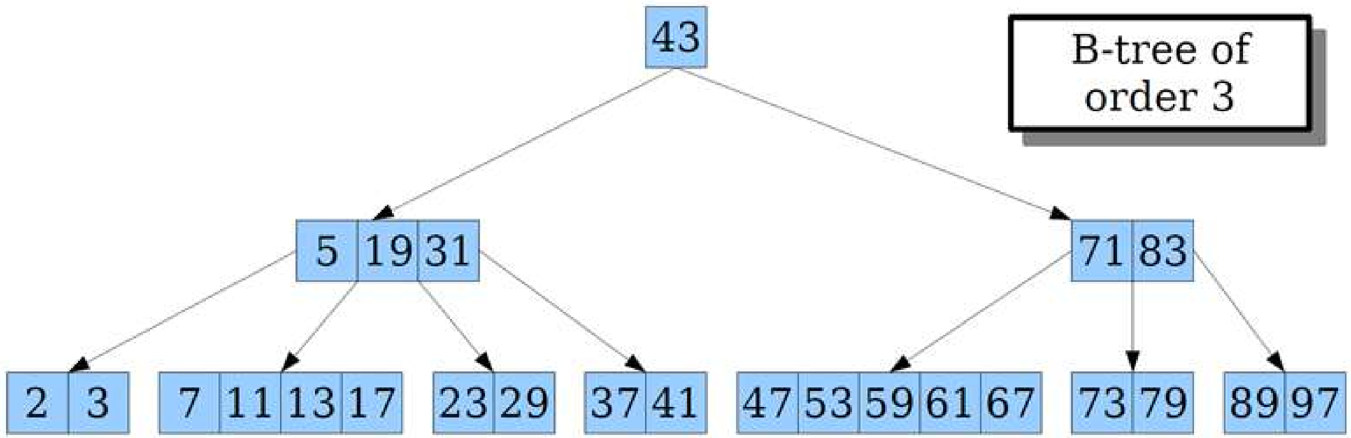
B-Tree



- 二叉查找树的推广
- 任何节点都保存了多个有序的元素，这些元素把子树里的节点分成多个子树

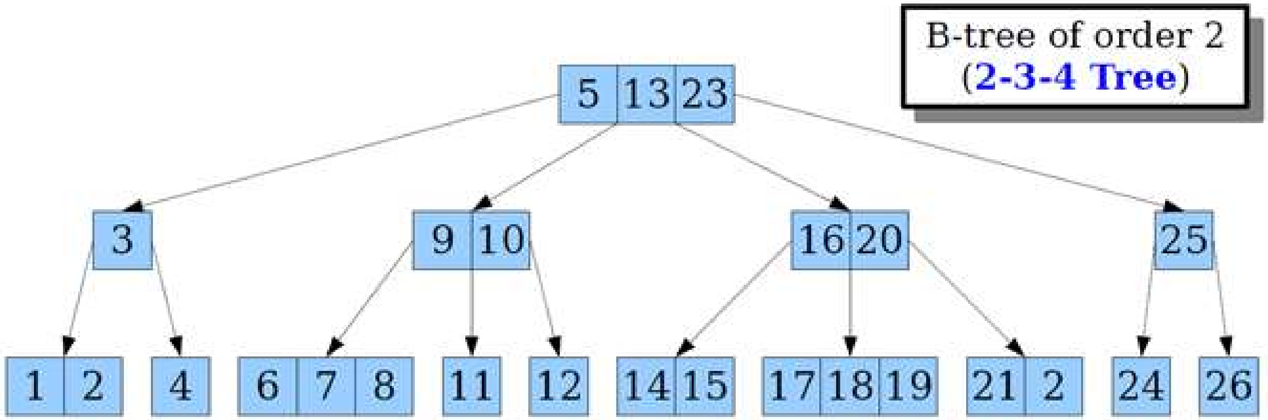
B树的定义

- b阶的B树
  - 根节点有最多$2b-1$个元素
  - 内部节点最多有$2b-1$个元素，同时至少$b-1$ 个元素
  - 所有叶节点高度都一样
  - 所有根节点到空节点的路径长度都一样

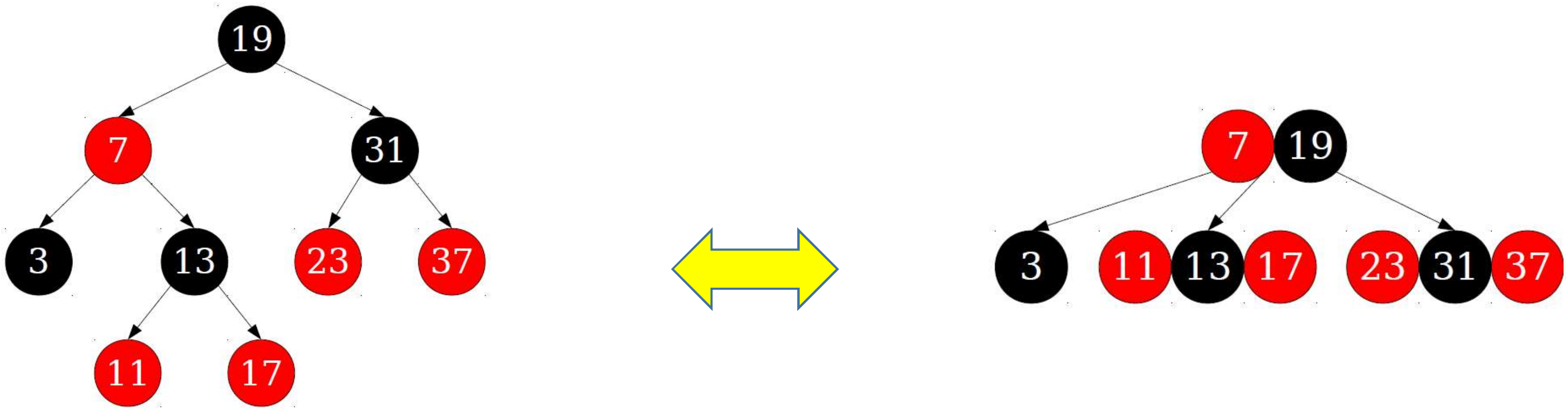


2-3-4树

- 一种特殊的2阶B树
- 每个节点有1，2，或者3个元素
- 和红黑树等价



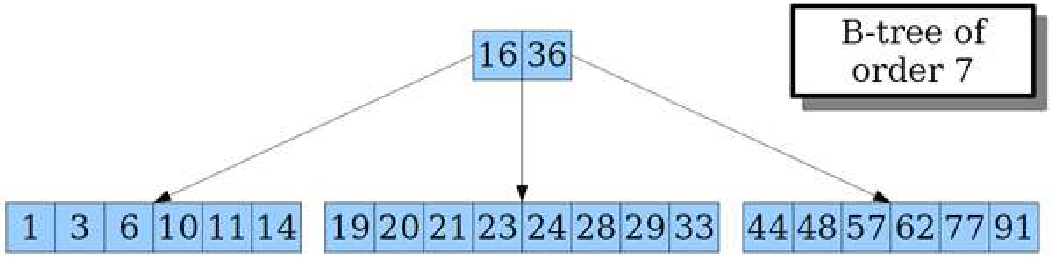
等价于红黑树



- 把红节点按位置收缩到父黑节点
- 注意红黑树里的任何操作情形，也有匹配的2-3-4树里的情形
- 2-3-4树里的情形更容易理解点

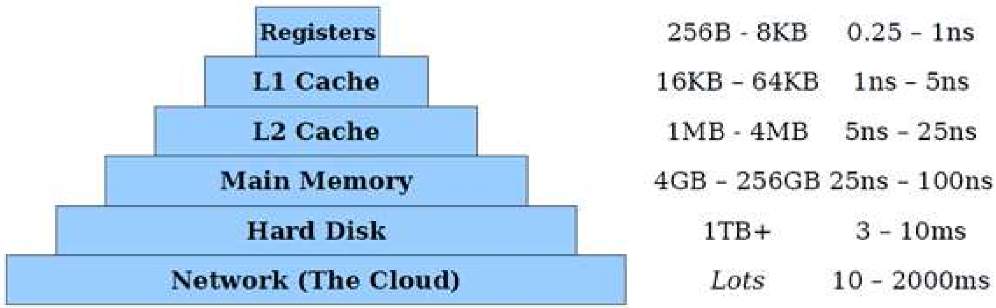
7阶B树

- 分支越多，树的高度越低
- 同时，节点里的元素越多，在节点内插入/删除的复杂度也高
- 阶取什么值最合适？



Memory Hierarchy

- 内存以及以上的速度快（ns），容量小
- 硬盘速度慢（ms），容量大
- 读取大量数据时候，瓶颈在内存和硬盘的交互
- 所以需要优化I/O读取的次数（热门的算法复杂度模型）



硬盘的读取特性

- 每次读取都是以块Block为单位
- Prefetching设计
- 顺序读写比随机读写快一个数量级

为什么使用B-tree

- B-tree的多分支降低了块读取次数
- 名字可能来源于块（B）lock
- 节点内元素的顺序存储，利用到了顺序读写和prefectching
- 特别适合文件系统和数据库

- 所以合适的阶应该是block size的常数倍，即O(BlockSize).

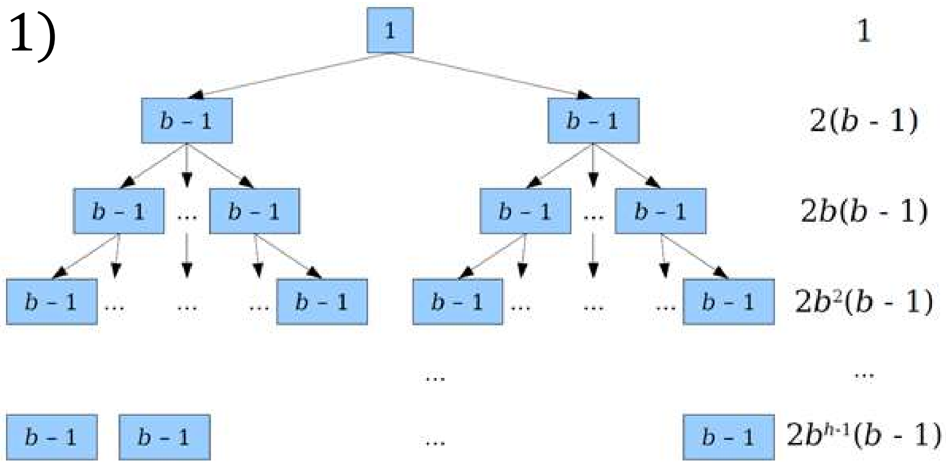
b阶B-Tree高度

定理：最大高度为$log_b((n+1)/2)$

即$O(log_b(n))$

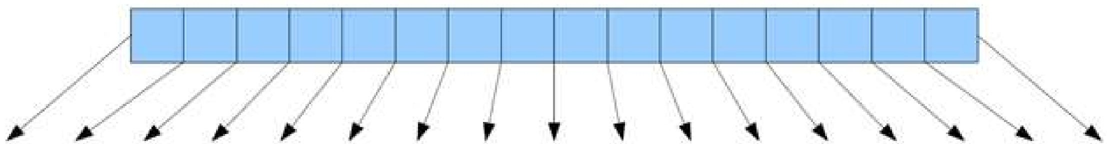
$$1+2(b-1)+2b(b-1)+\cdots+2bh^{-1}(b-1)$$
$$=1+2(b-1)(1+b+\cdots+bh^{-1})$$
$$=1+2(b-1)(bh-1)/(b-1)$$
$$=2bh-1$$
$$=n$$

$\Rightarrow \quad h=log_b((n+1)/2)$



查找

- 在根节点查找X
- 如果没有找到，就去相应子节点递归查找
- 在每个节点内部可以使用二分法查找$O(\log(\text{节点元素个数})) = O(\log b)$
- 从根节点查到叶节点$O(\log(\text{树的高度})) = O(\log_b n)$
- 如果按块的读取次数计算，复杂度是什么？

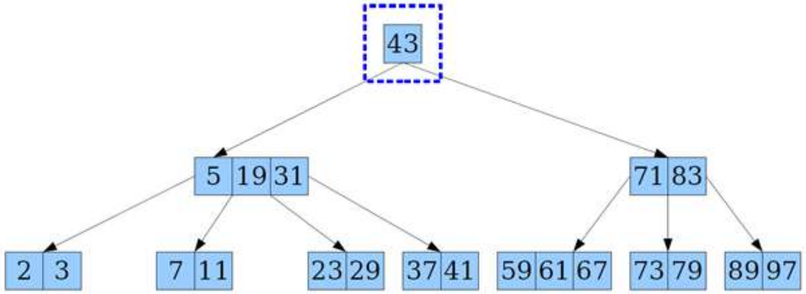


插入算法

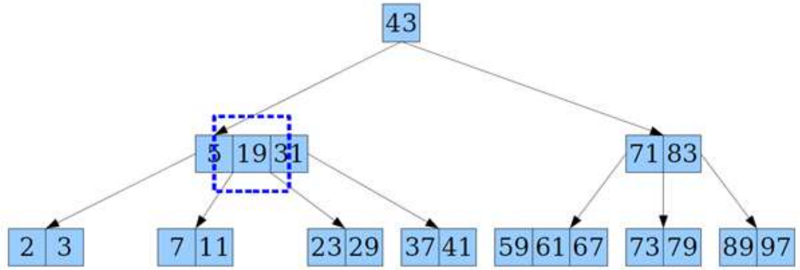
- 查找到要插入的叶节点
- 如果叶节点还没满，结束
- 要不然（包含$2b$个元素）
  1) 把节点分裂成两个节点 （每个b个元素）
  2) 中间的元素上浮并合并到父节点
  3) 如果父节点也满了，就重复以上步骤
- 复杂度
  - 节点操作是$O(b)$，最多上浮$O(\log_b n)$次
  - 按块的读取次数算是$O(\log_b n)$

例子：简单情形（3阶），先查找

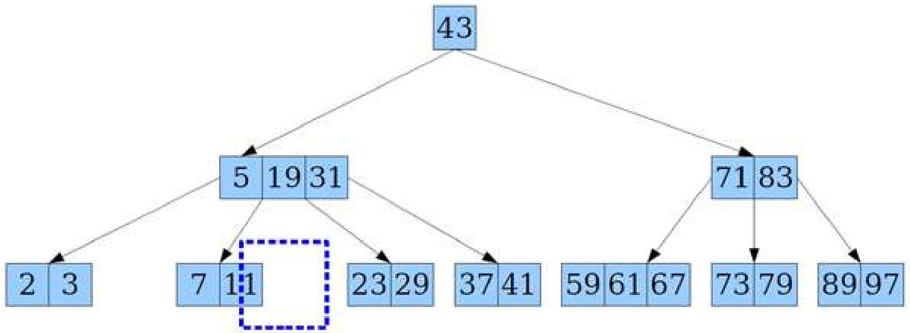
①



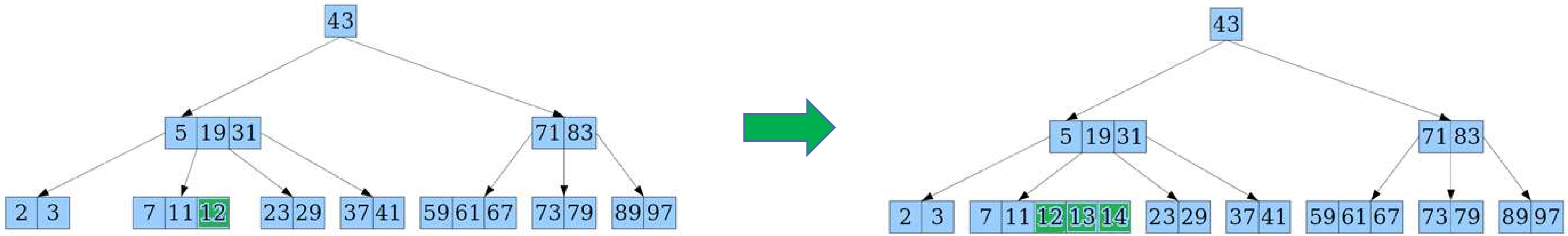
②



③

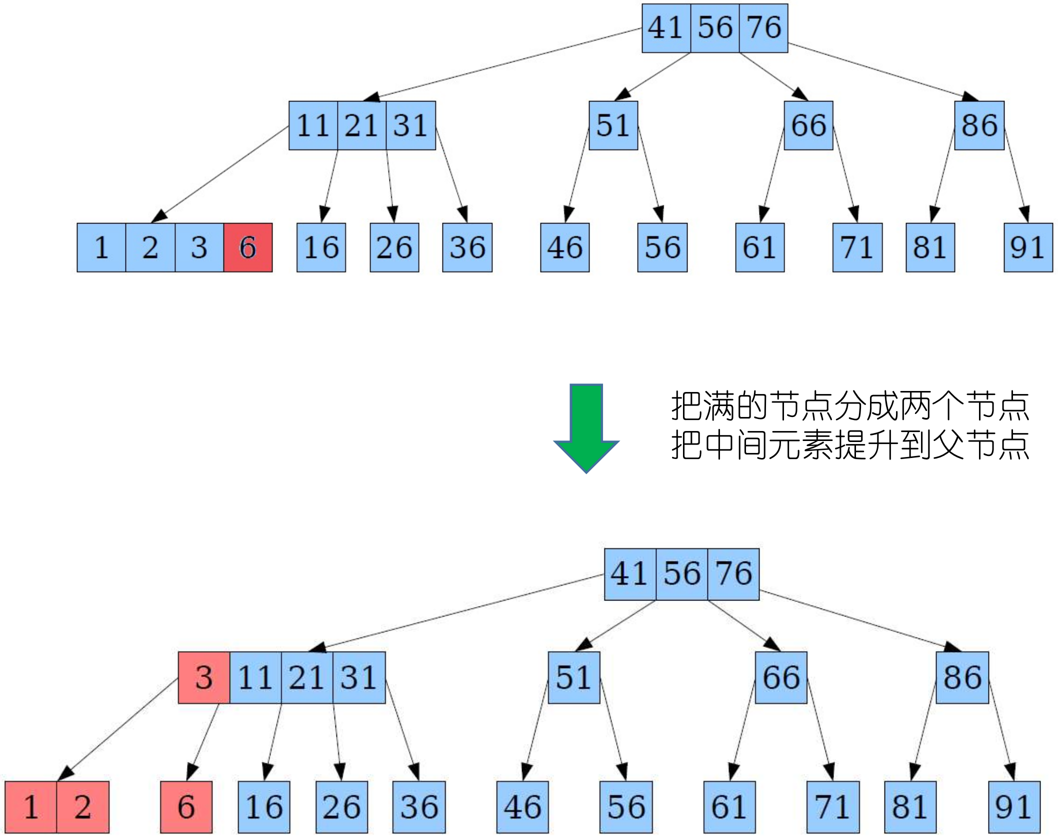


直接插入12， 13， 14

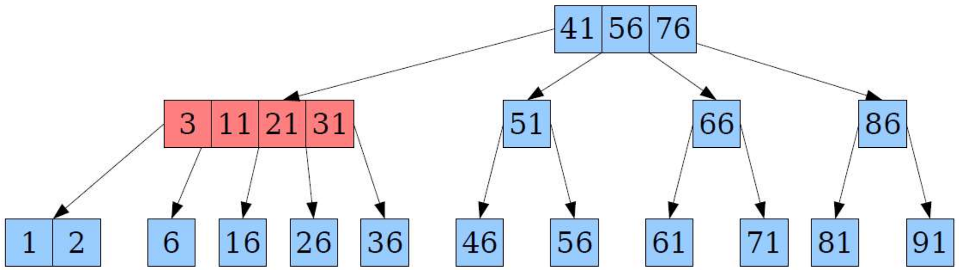


43
5 19 31
71 83
2 3
7 11 12
23 29
37 41
59 61 67
73 79
89 97
43
5 19 31
71 83
2 3
7 11 12 13 14
23 29
37 41
59 61 67
73 79
89 97

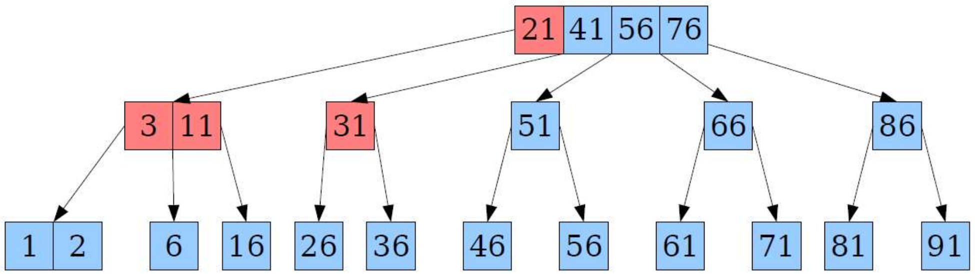
例子：插入节点满情形（2-3-4树）



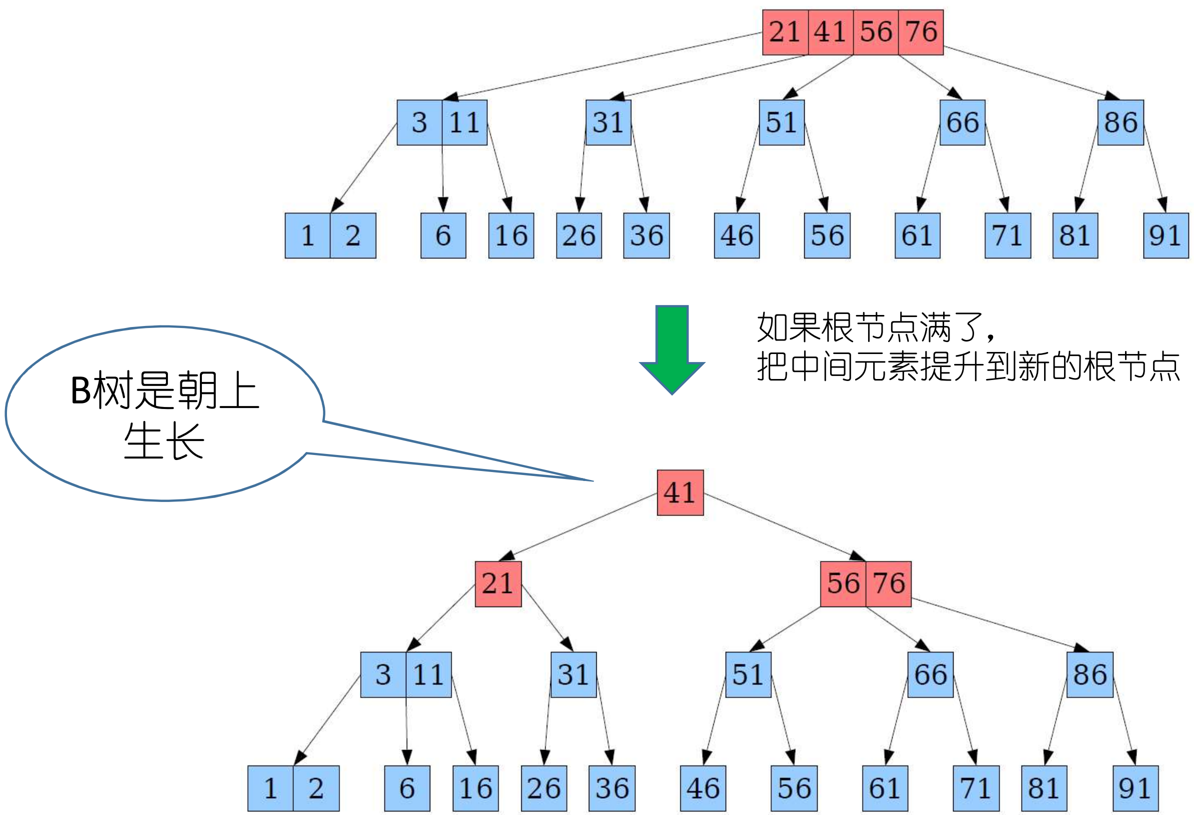
例子：父节点满情形（2-3-4树）



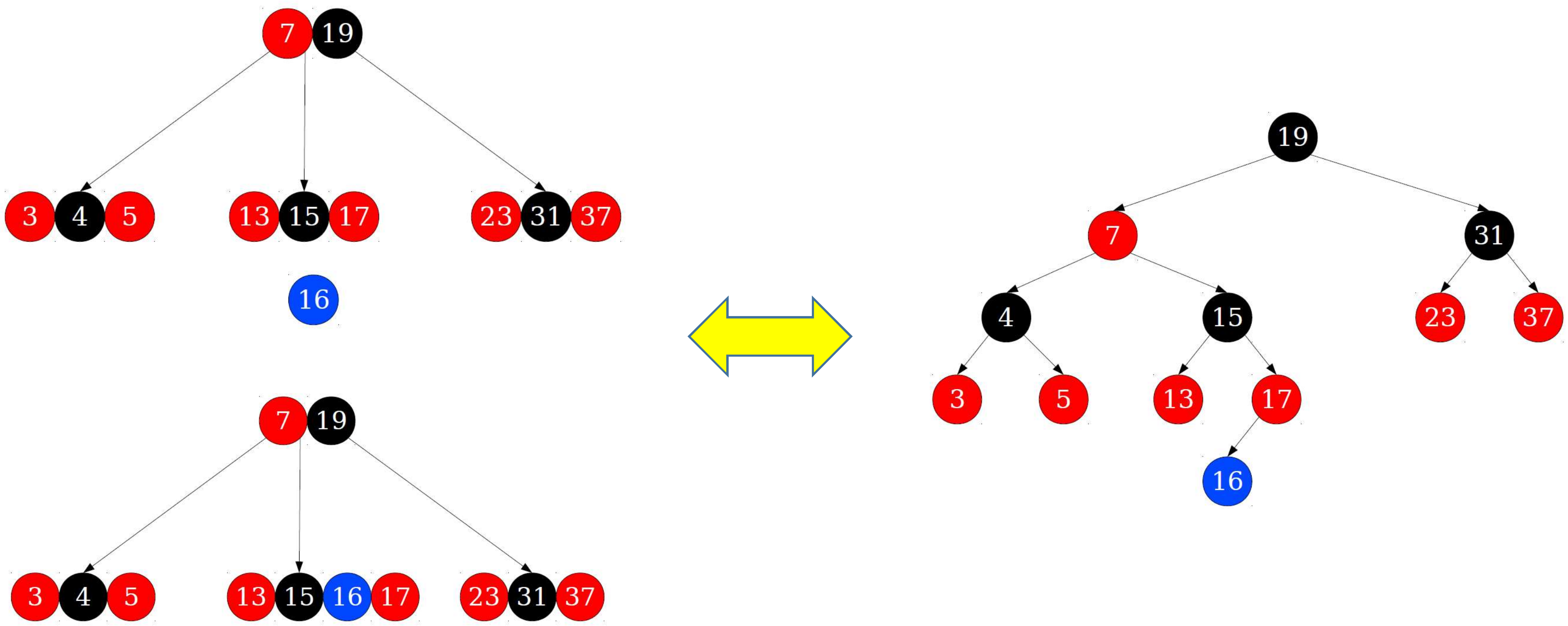
继续把满的父节点分开，
把中间元素提升到爷爷节点



例子：父节点满情形（2-3-4树）

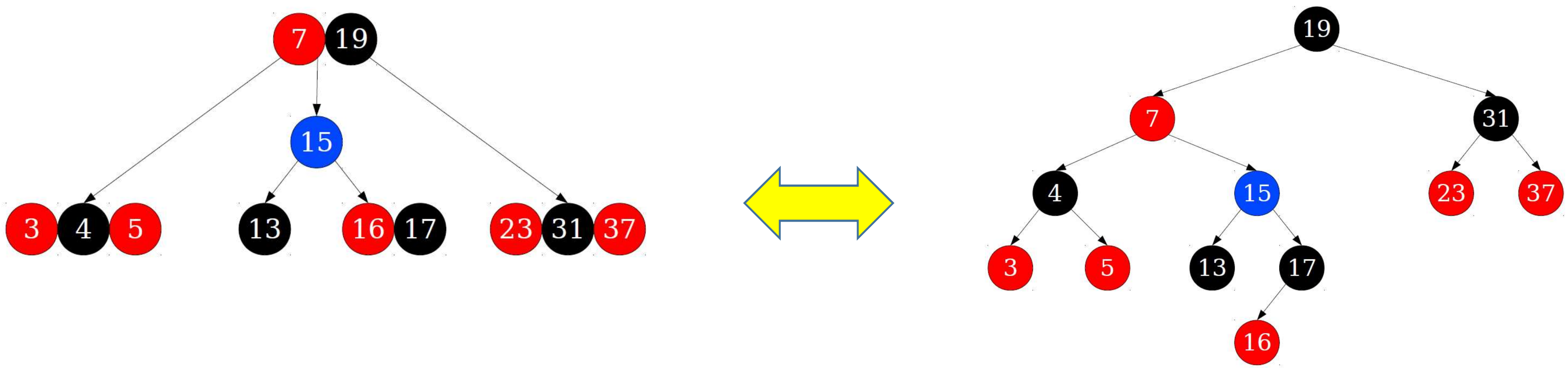


2-3-4树 vs 红黑树



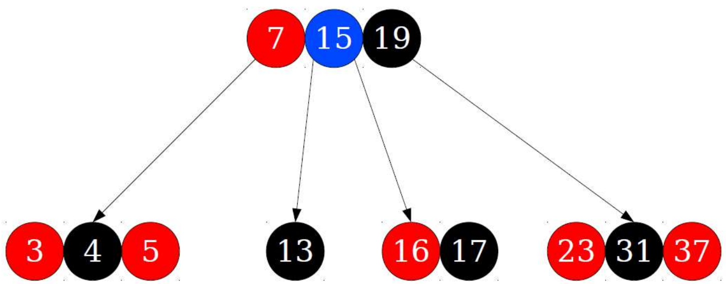
- 插入16后，2-3-4树叶节点多于3个元素，红黑树红颜色冲突

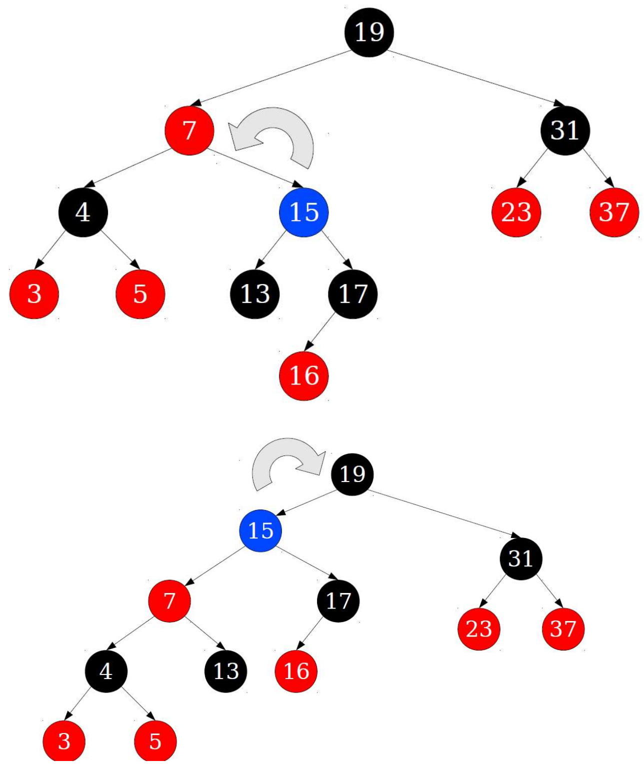
2-3-4树 vs 红黑树



- 2-3-4树： split节点，把中间元素提升成父节点
- 红黑树： 换颜色

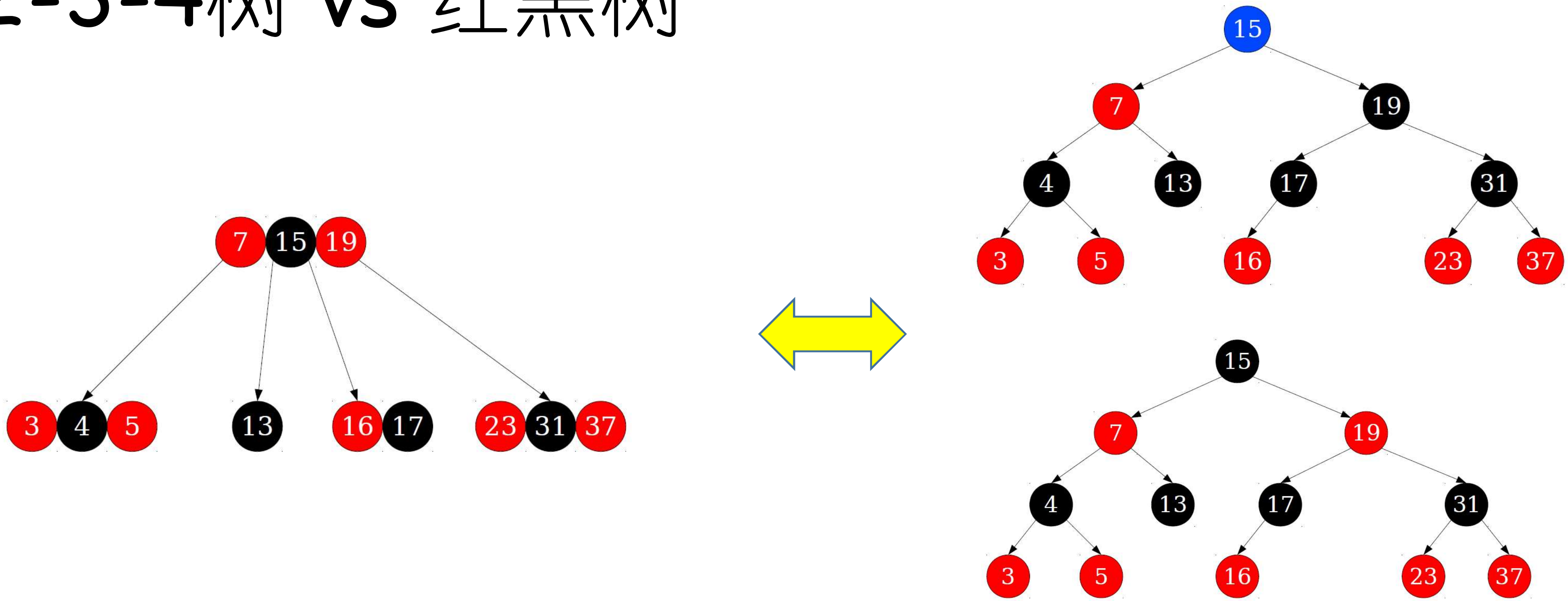
2-3-4树 vs 红黑树





- 2-3-4树：提升中间元素15，合并到父节点
- 红黑树：左旋，右旋

2-3-4树 vs 红黑树



- 2-3-4树： 换颜色，匹配红黑树，可缺省
- 红黑树： 换颜色

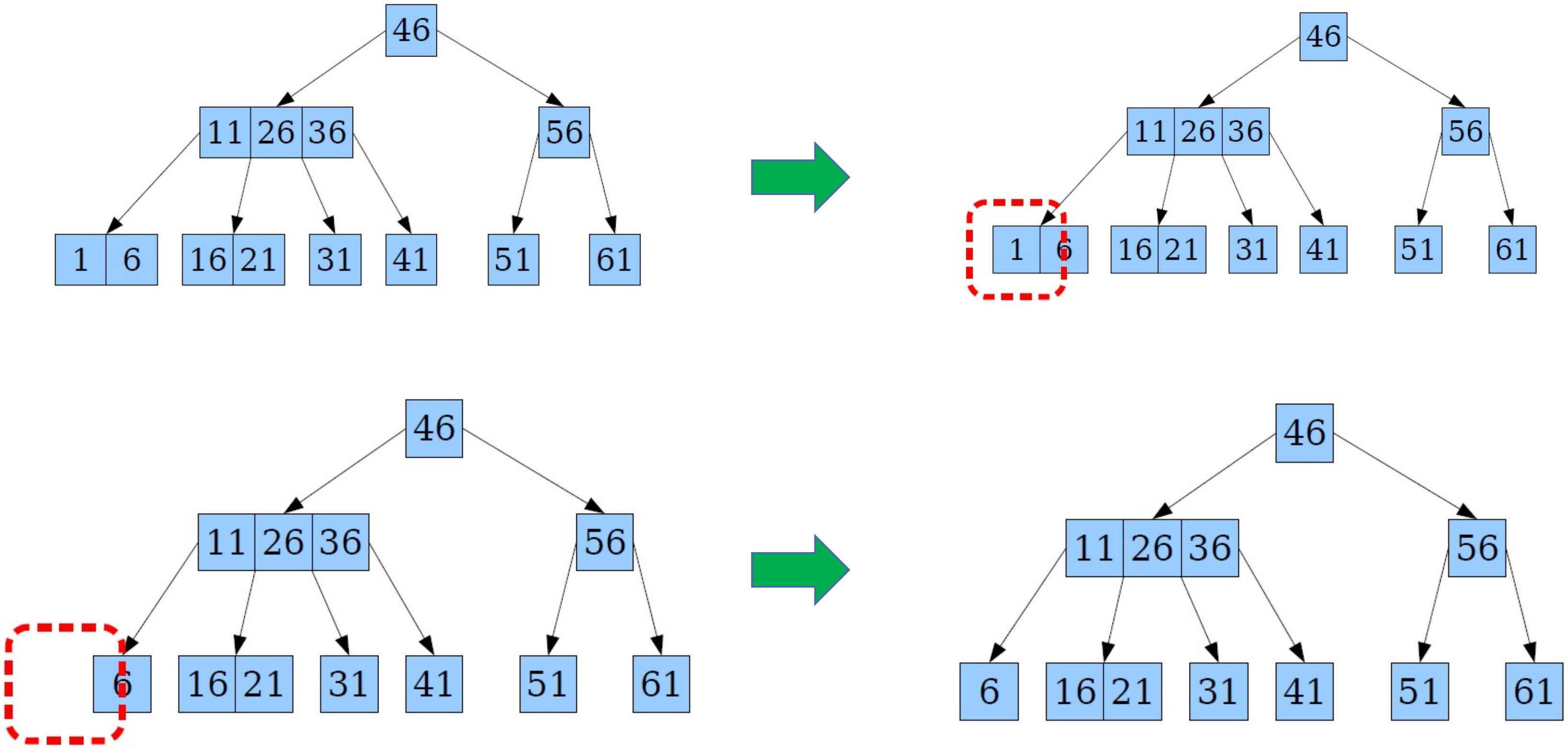
删除算法

- 如果被删除元素X是在内部节点，用X的直接后继S替换，相当于在叶节点删除S
- 如果被删除元素X在叶节点
  - 如果叶节点有足够多元素$> b-1$，直接删除

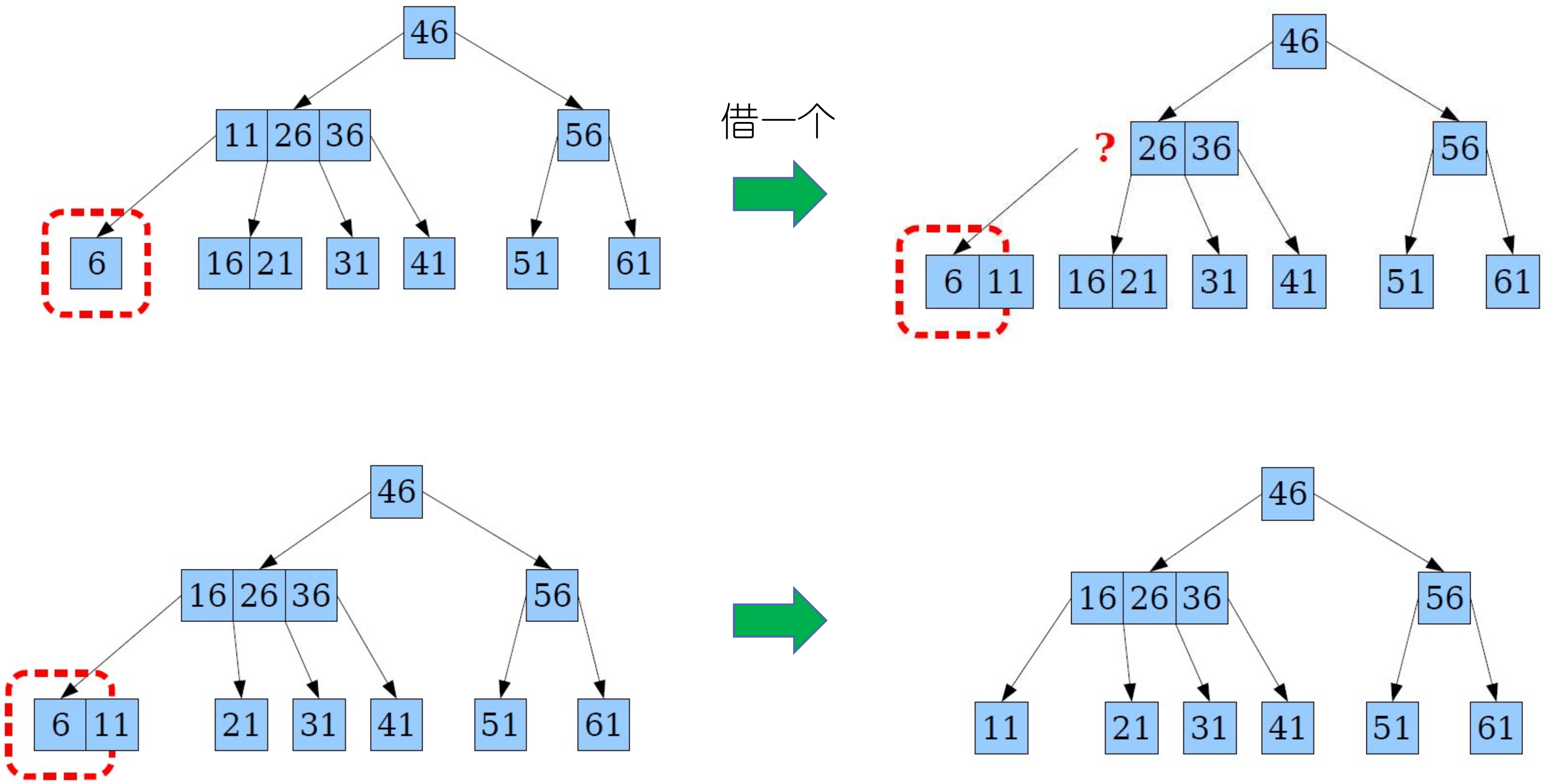
1) 要不然找下（前）一个兄弟节点借一个最小（大）元素
2) 即把借的元素替换父元素p，然后把p移到叶节点，类似左（右）旋
3) 如果兄弟节点只有$b-1$元素，把他们合并（包括p），然后从父节点删除p

  - 如果父节点元素个数不够，重复1）-- 3）
- 复杂度
  - 节点操作是$O(b)$，最多上浮$O(\log_b n)$次
  - 按块的读取次数算是$O(\log_b n)$

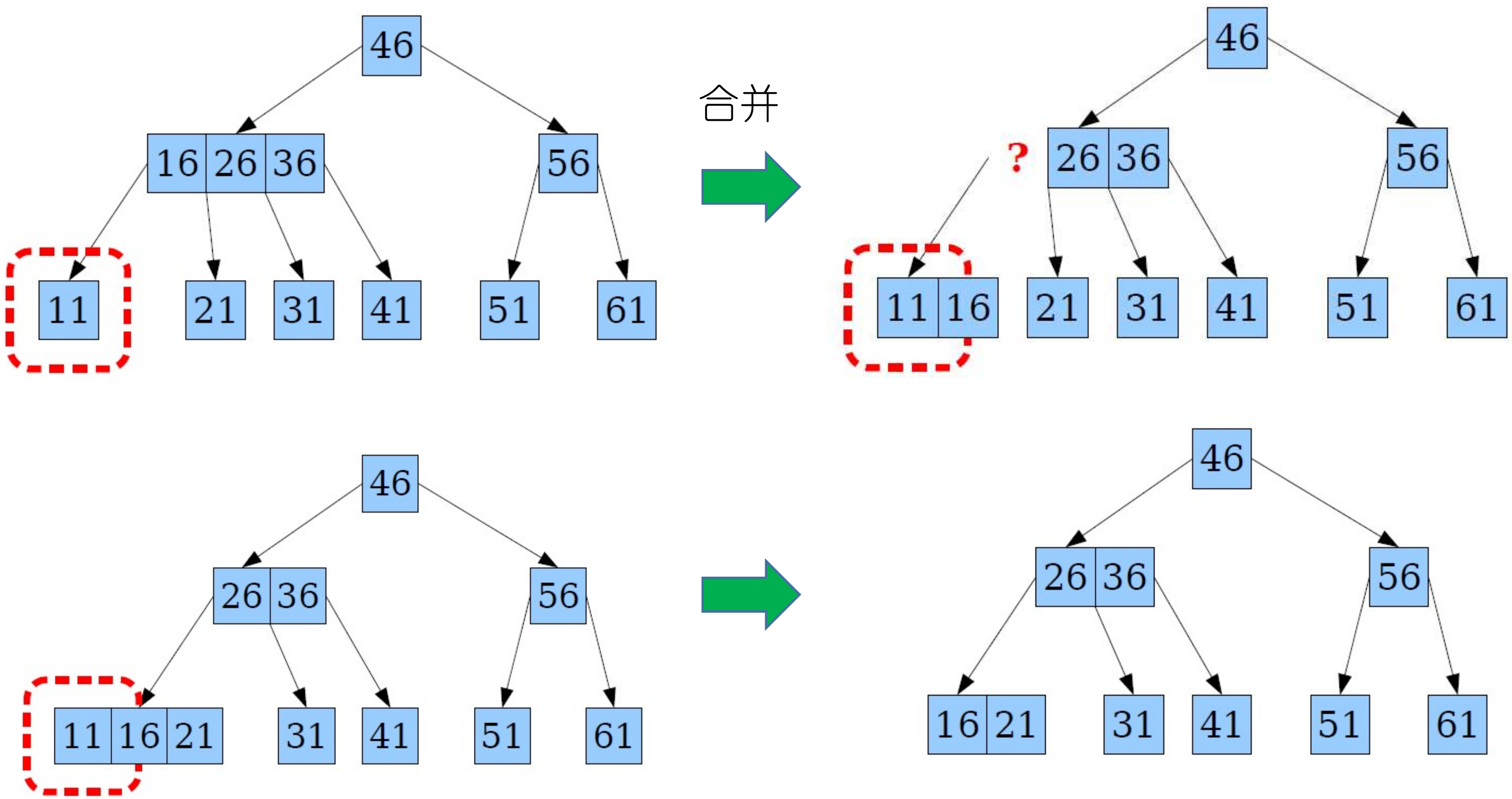
例子：简单情形（2-3-4树）


46
11 26 36
56
1 6
16 21
31
41
51
61
46
11 26 36
56
1 6
16 21
31
41
51
61
46
11 26 36
56
6
16 21
31
41
51
61
46
11 26 36
56
6
16 21
31
41
51
61

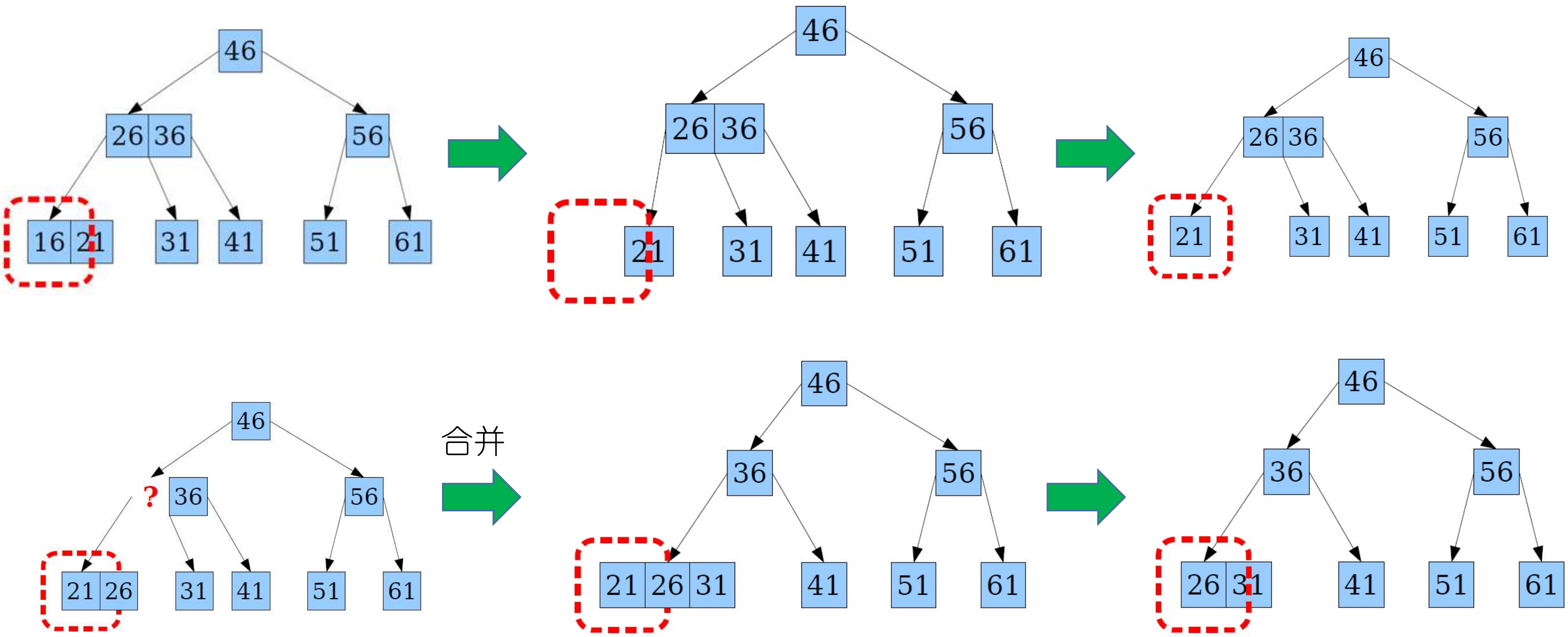
例子：删除节点不够情形1（2-3-4树）



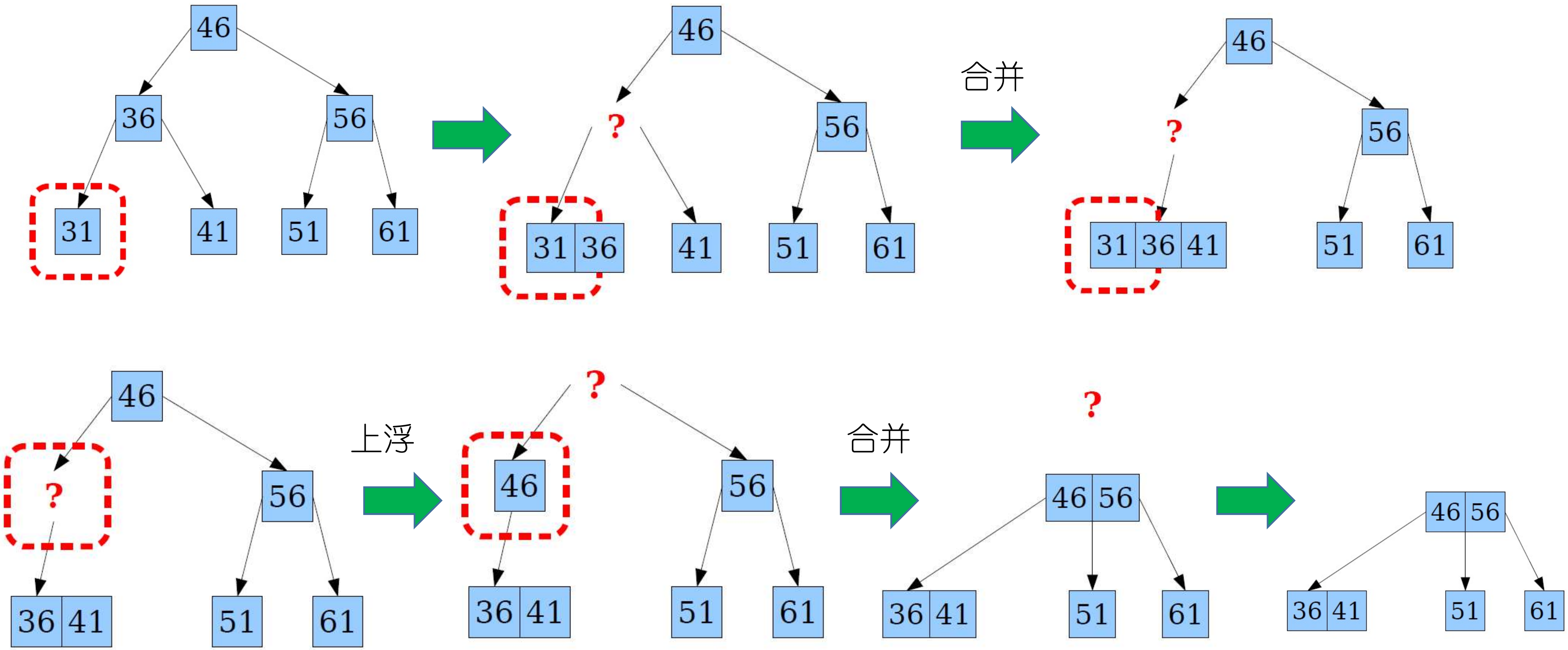
例子：删除节点不够情形2（2-3-4树）



例子：继续删除，上浮到父节点情形



例子：继续删除，上浮到父节点情形



总结

- 理解B-树的节点分裂，合并规则
- 理解2-3-4树和红黑树的等价
- 类比2-3-4树和红黑树的操作，更好地理解为什么要旋转，换颜色

B+树

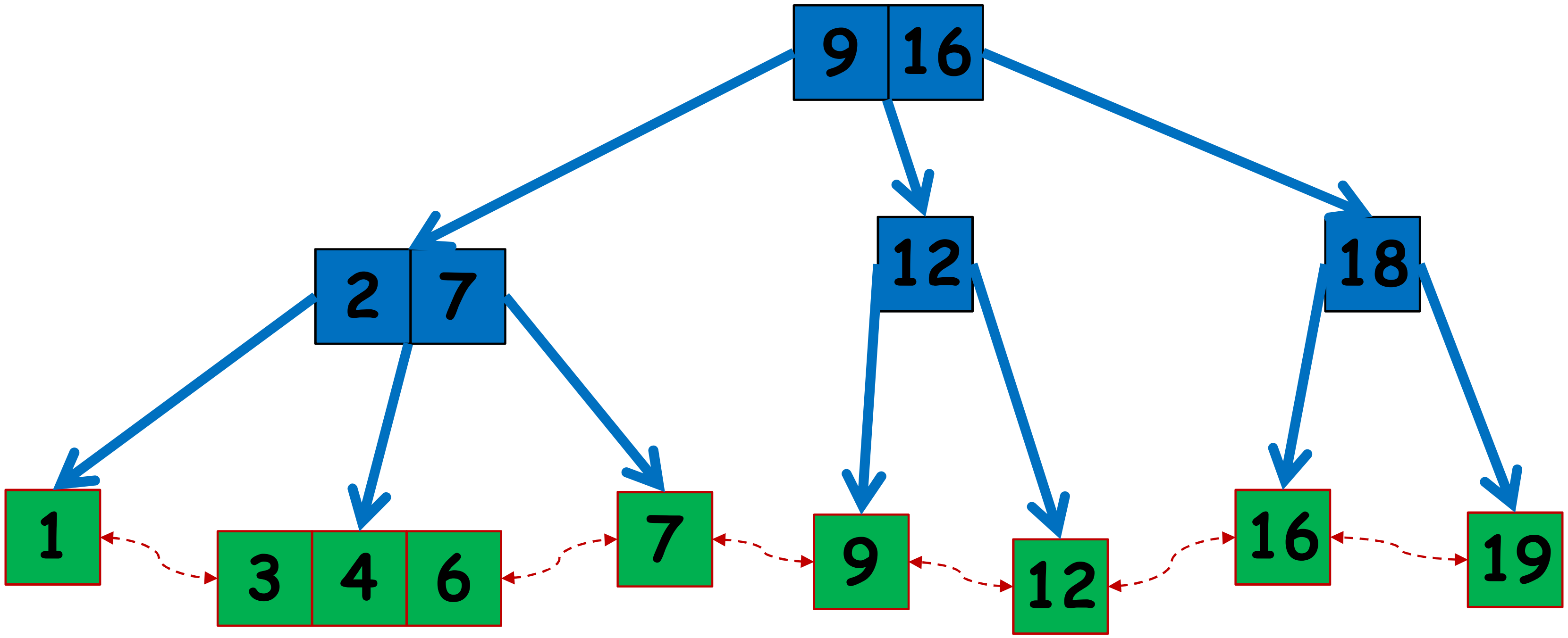
B+ Tree

和B树相似，区别是：

- 全部数据都存在叶节点;
- 内部节点只是索引元素,
- 内部节点的元素可以重复，索引元素的右子树元素(≥)
- 叶节点之间也有按顺序连接，以便中序遍历

- 数据库，包括mysql用B+树

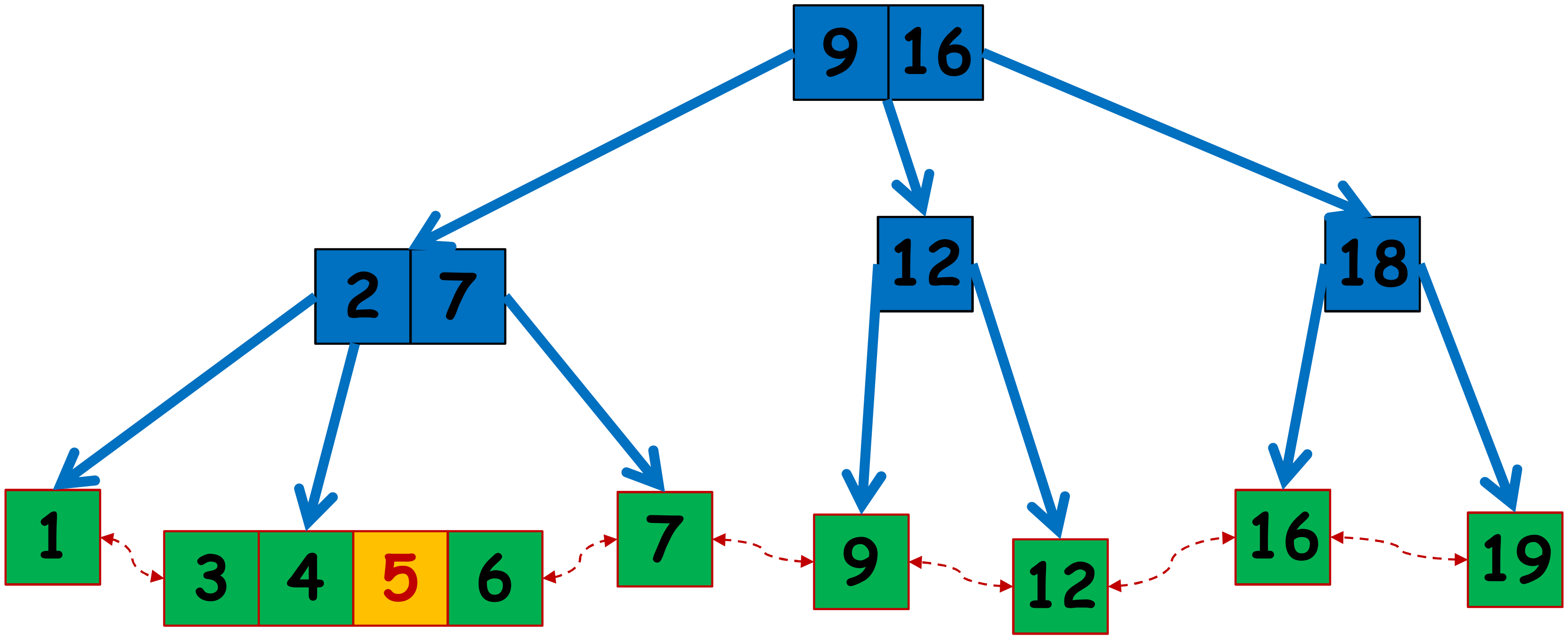
B+ Tree例子



B+树插入

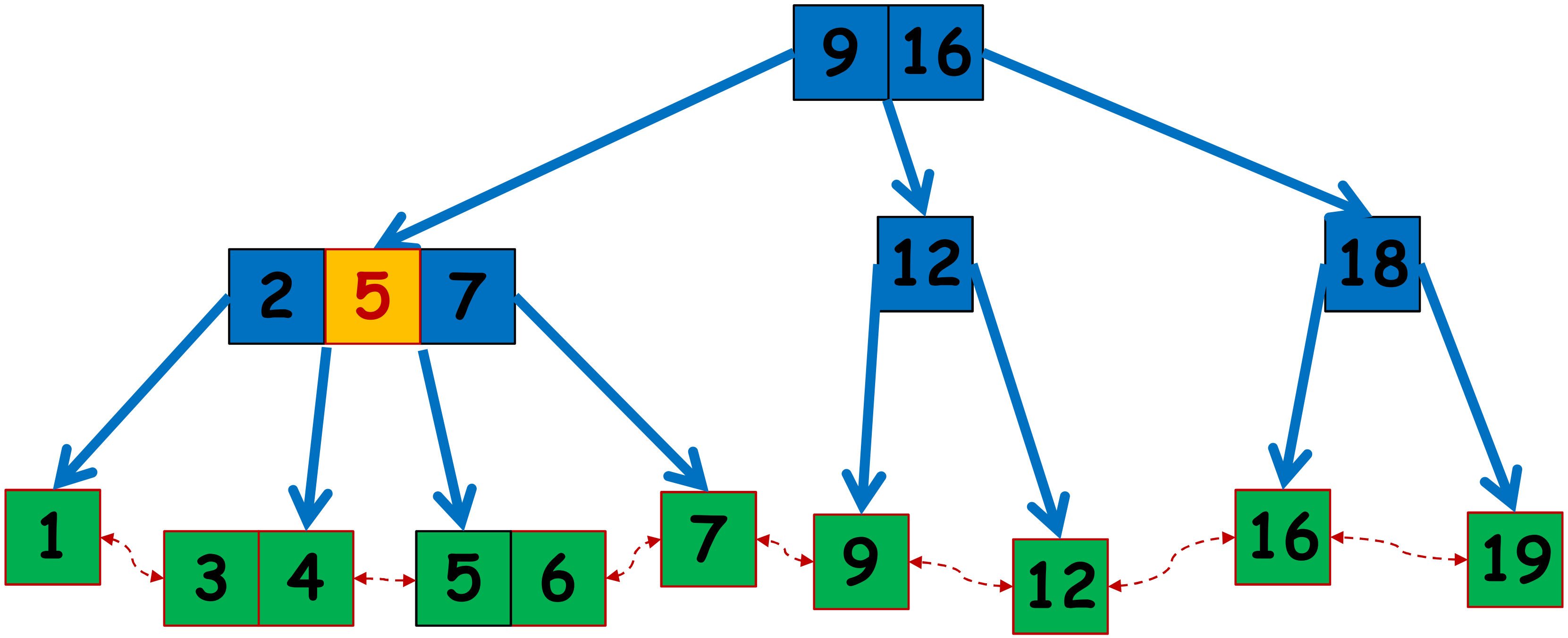
- 插入在叶节点
- 如果叶节点满了
  1) 把节点分裂成两个节点（每个b个元素）
  2) 中间的元素复制并合并到父节点
- 如果索引节点满了，
  1) 把节点分裂成两个节点
  2) 中间的元素上浮并合并到父节点

插入2阶树：叶节点满情形



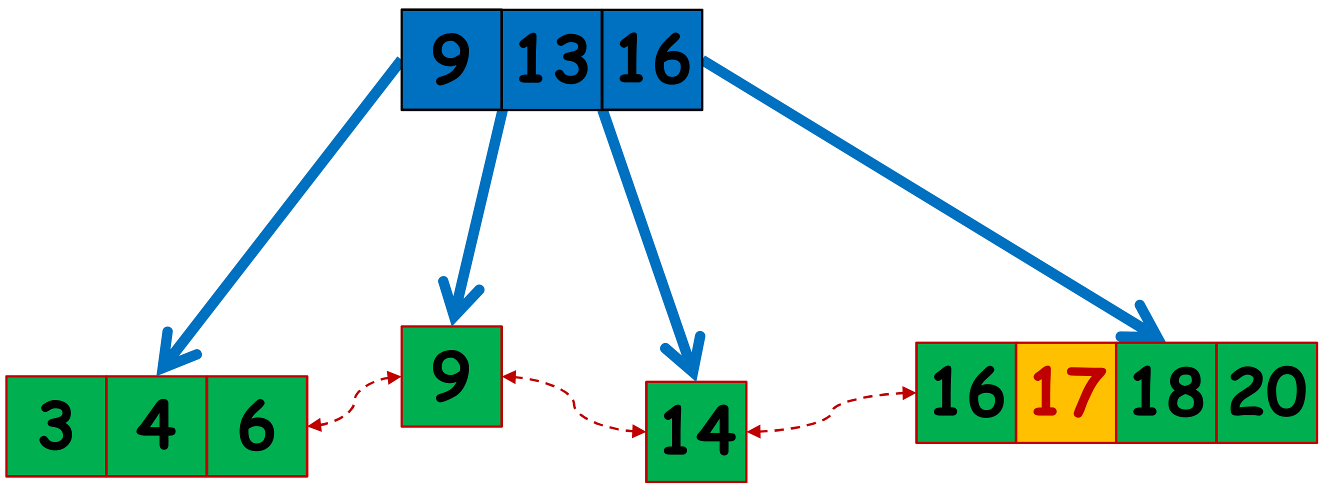
- 插入5

插入2阶树：叶节点满情形



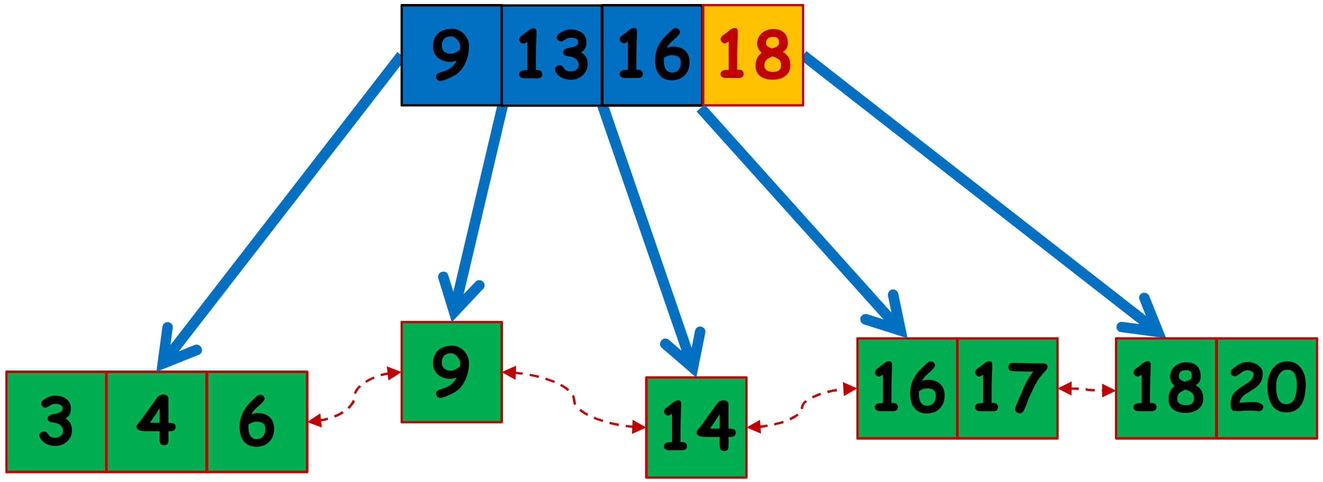
- 分裂叶节点，建立连接
- 中间元素上浮到父节点作索引

插入2阶树：索引节点满情形



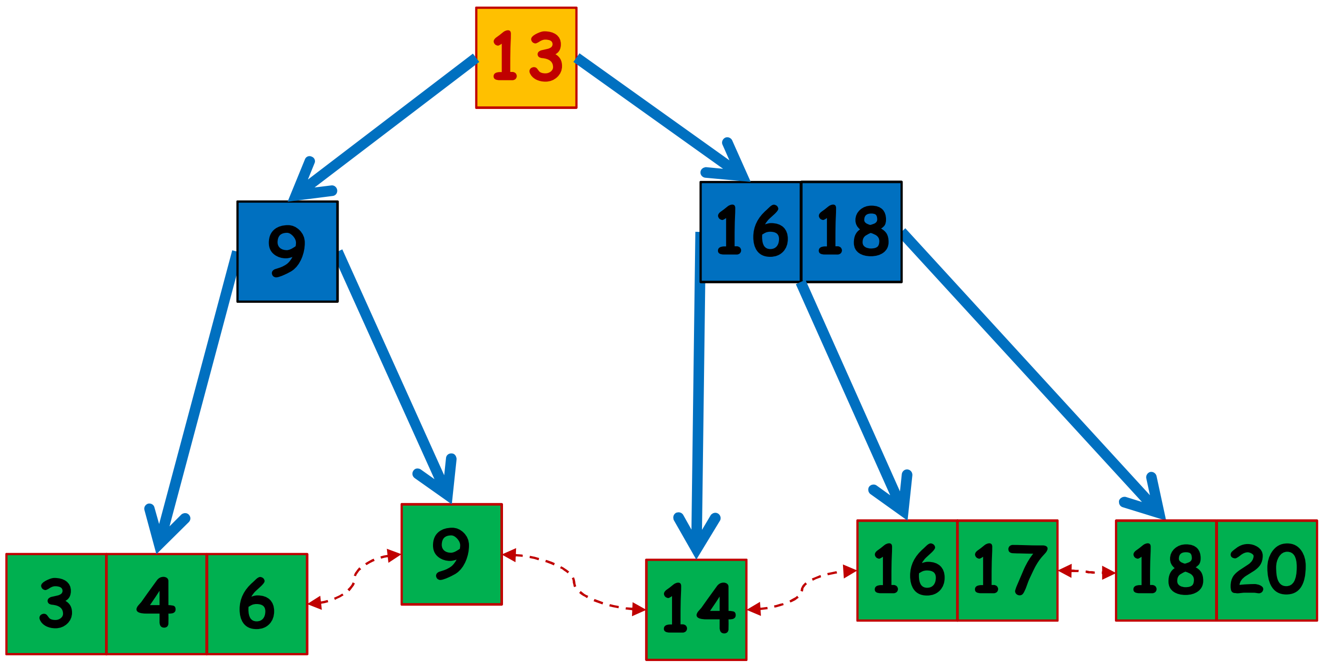
- 插入17

插入2阶树：索引节点满情形



- 分裂叶节点，建立连接
- 中间元素复制到父节点作索引

插入2阶树：索引节点满情形

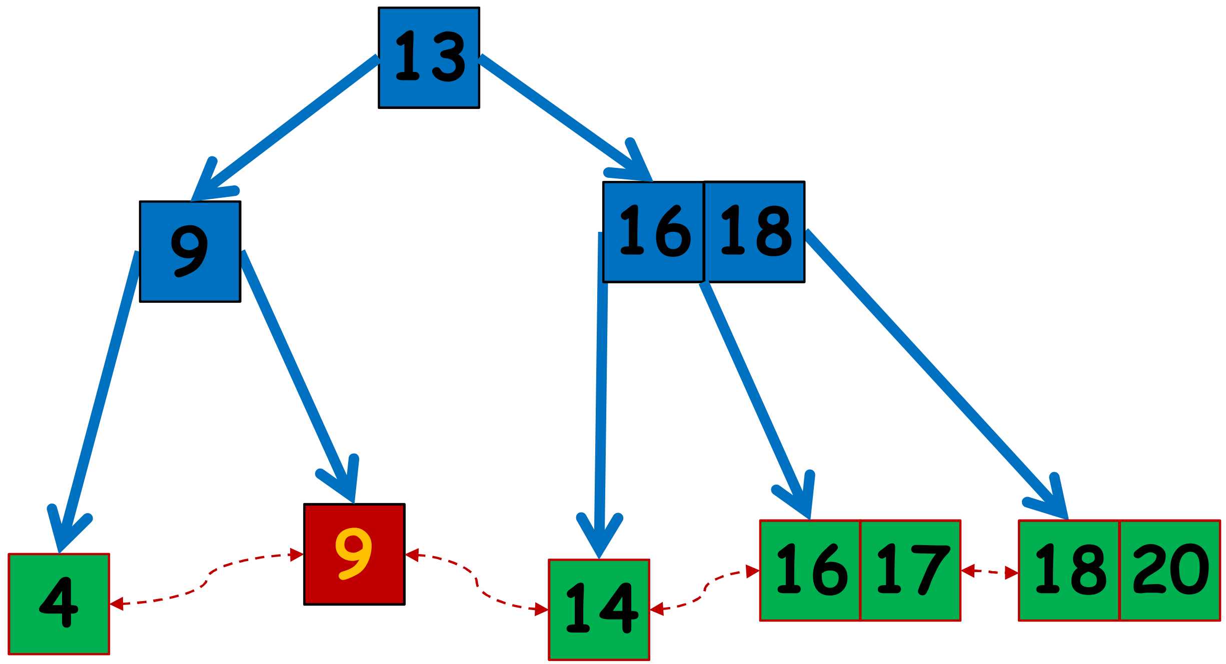


- 分裂中间节点
- 中间元素上浮作为根节点

B+树删除算法

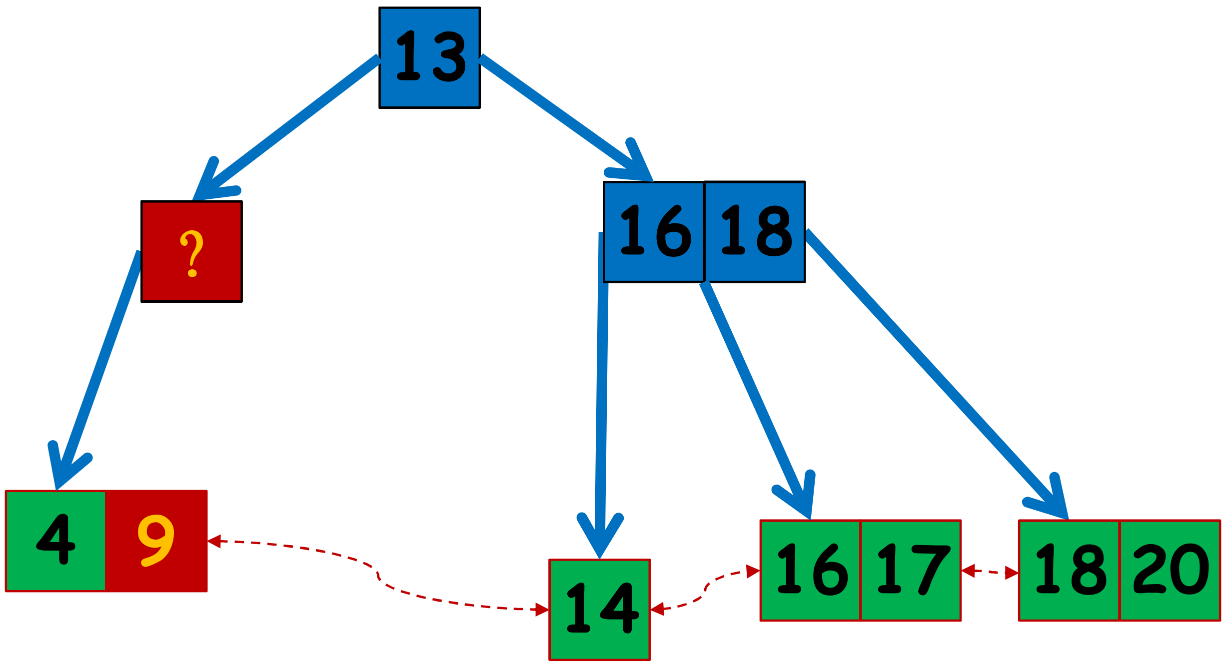
- 从叶节点删除元素
- 如果叶节点元素不够
  - 向兄弟节点借，借的元素要替代原来的父节点元素
  - 合并兄弟节点，删除父节点元素
- 如果索引节点元素不够
  - 类似B树

删除2阶树：



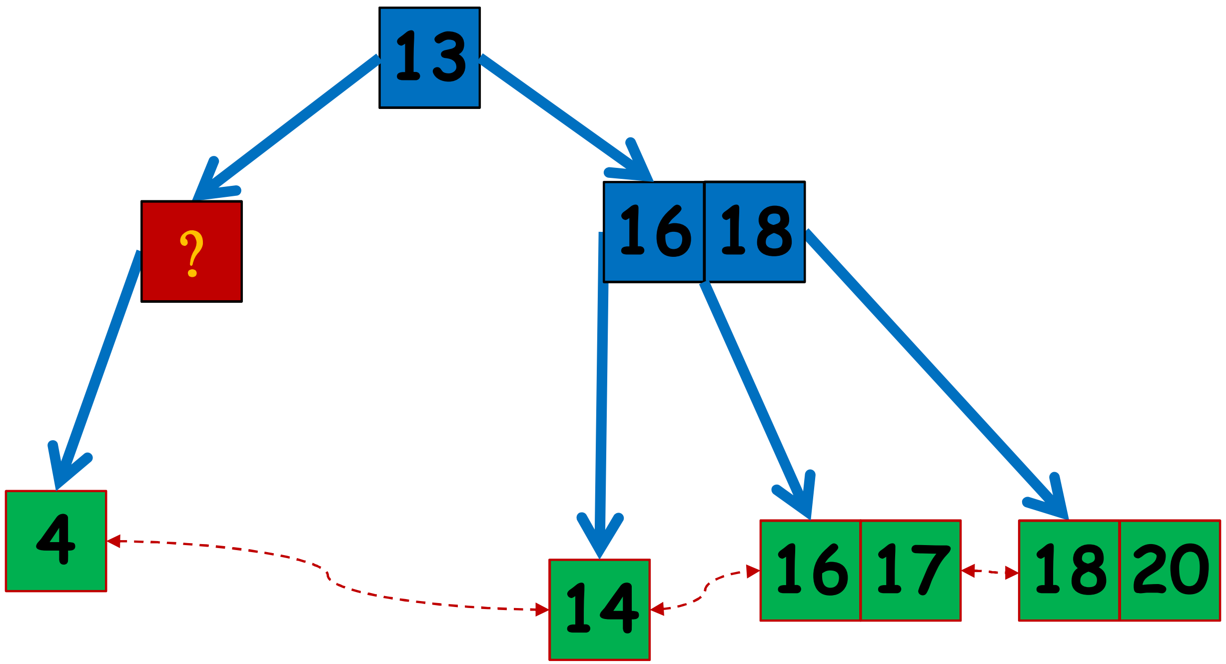
- 删除9

删除2阶树：



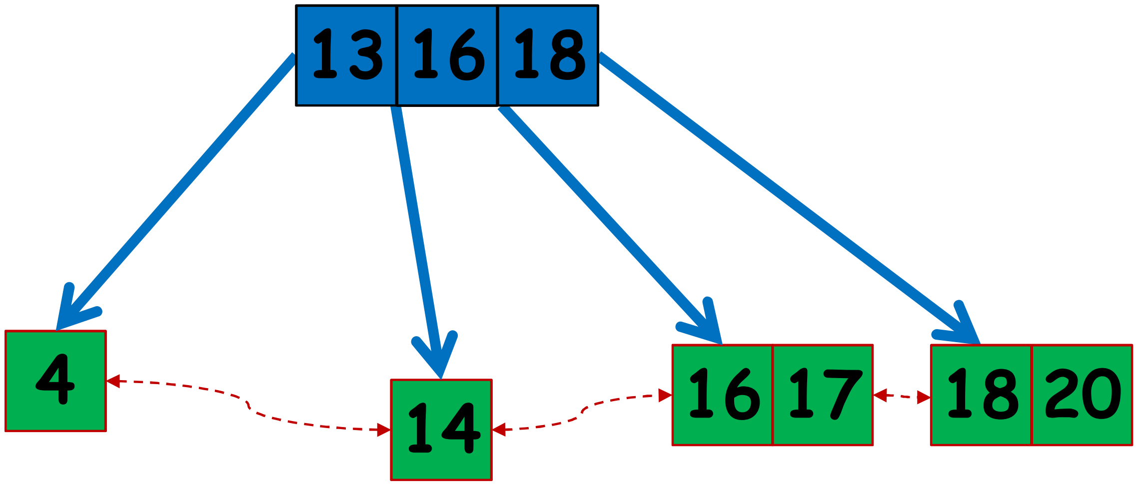
- 合并叶节点，移去父节点元素

删除2阶树：



- 索引节点不足，合并

删除2阶树：



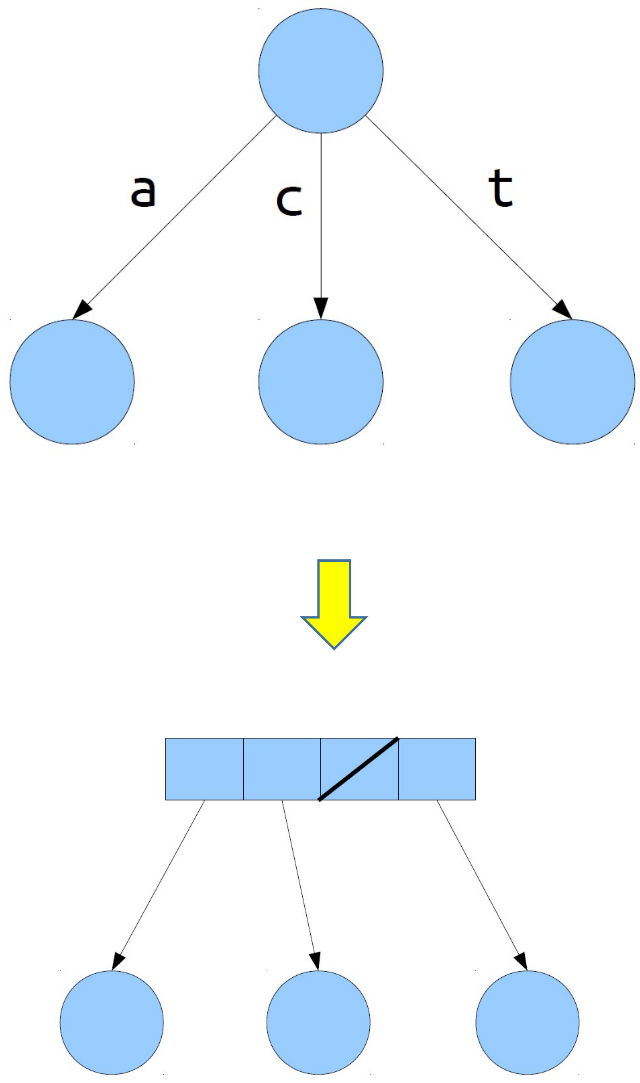
- 合并索引节点，下移13

Trie

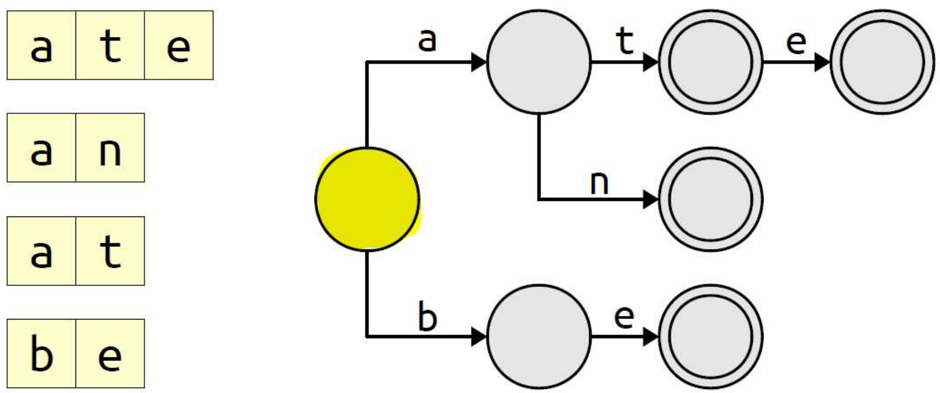
Trie 前缀树

- 每个节点都包含指向子节点的数组
- 这里我们假定字符集就是26字母

```
// trie node
struct TrieNode
{
    bool isKeyEnd; // true if leaf
    TrieNode * children[ALPHABET_SIZE];
};
```



Trie例子



查找算法

```
bool Trie::Search(char *szKey)
{
    int length = strlen(szKey);
    TrieNode *pNode = this->root;

    for (int level = 0; level < length; level++)
    {
        int index = *(szKey + level) - 'a';

        if (!pNode->children[index])
        {
            return false;
        }

        pNode = pNode->children[index];
    }

    return (pNode && pNode->isKeyEnd);
}
```

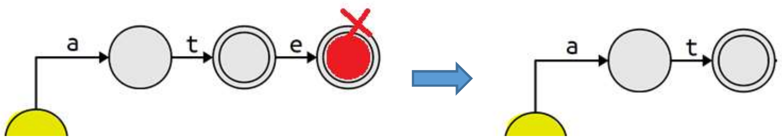
插入算法

```
void Trie::Insert(char * szKey)
{
    int length = strlen(szKey);
    TrieNode * pNode = this->root;

    for(int level = 0; level < length; level++)
    {
        int index = *(szKey + level) - 'a';

        if(!pNode->children[index])
        {
            // Add new node
            pNode->children[index] = getNode();
        }

        pNode = pNode->children[index];
    }

    // mark last node as leaf (non zero)
    pNode->isKeyEnd = true;
}
```

删除算法

- 如果要删除的字符串不在Trie里，结束
- 如果要删除的字符串是在Trie里只是某个prefix，只需要把叶节点的标志去掉

- 如果要删除的字符串是在Trie里和另外一个元素有公共的prefix，需要删除不是prefix那部分



- 最后一种情况，可以删除所有节点

删除算法

```
void Trie::Delete(char *szKey)
{
    DeleteHelper(this->root, szKey, 0, strlen(szKey));
}
```

```
bool Trie::DeleteHelper(TrieNode *pNode, char *szKey, int level, int len)
{
    if (!pNode || len == 0)
    {
        return false;
    }

    // Base case
    if (level == len)
    {
        if (pNode->isKeyEnd)
        {
            // Unmark leaf node
            pNode->isKeyEnd = false;

            // If empty, node to be deleted
            if (IsFreeNode(pNode))
            {
                return true;
            }
        }
        return false;
    }

    int index = *(szKey + level) - 'a';
    if (DeleteHelper(pNode->children[index], szKey, level+1, len))
    {
        // last node marked, delete it
        delete pNode->children[index];
        pNode->children[index] = nullptr;

        // recursively climb up, and delete eligible nodes
        return (!LeafNode(pNode) && IsFreeNode(pNode));
    }

    return false;
}
```

Q&A

Thanks!