8.高级搜索树

(b5) B-树：删除

射影，变了形，反而结晶

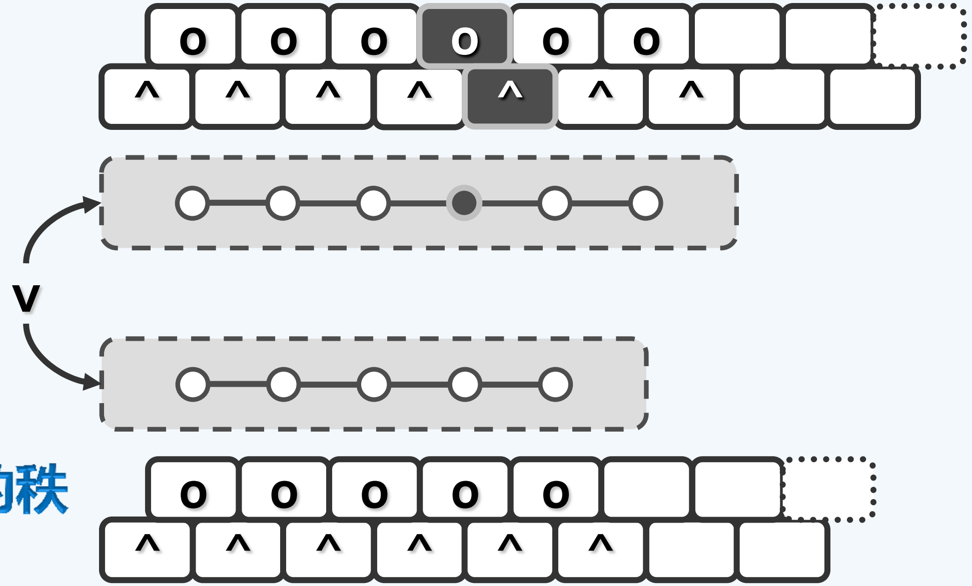
或动了情，也要合并，或归了零

也不愿不生不死不悔的倒影

邓俊辉

deng@tsinghua.edu.cn

算法




```
❖ template <typename T>
  bool BTree<T>::remove( const T & e ) {
     BTNodePosi(T) v = search( e );
     if ( ! v ) return false; //确认e存在
     Rank r = v->key.search(e); //确定e在v中的秩
     if ( v->child[0] ) { //若v非叶子，则
        BTNodePosi(T) u = v->child[r + 1]; //在右子树中一直向左，即可
        while ( u->child[0] ) u = u->child[0]; //找到e的后继（必属于某叶节点）
        v->key[r] = u->key[0]; v = u; r = 0; //并与之交换位置
     } //至此，v必然位于最底层，且其中第r个关键码就是待删除者
     v->key.remove( r ); v->child.remove( r + 1 ); _size--;
     solveUnderflow( v ); return true; //如有必要，需做旋转或合并
  }
```

旋转

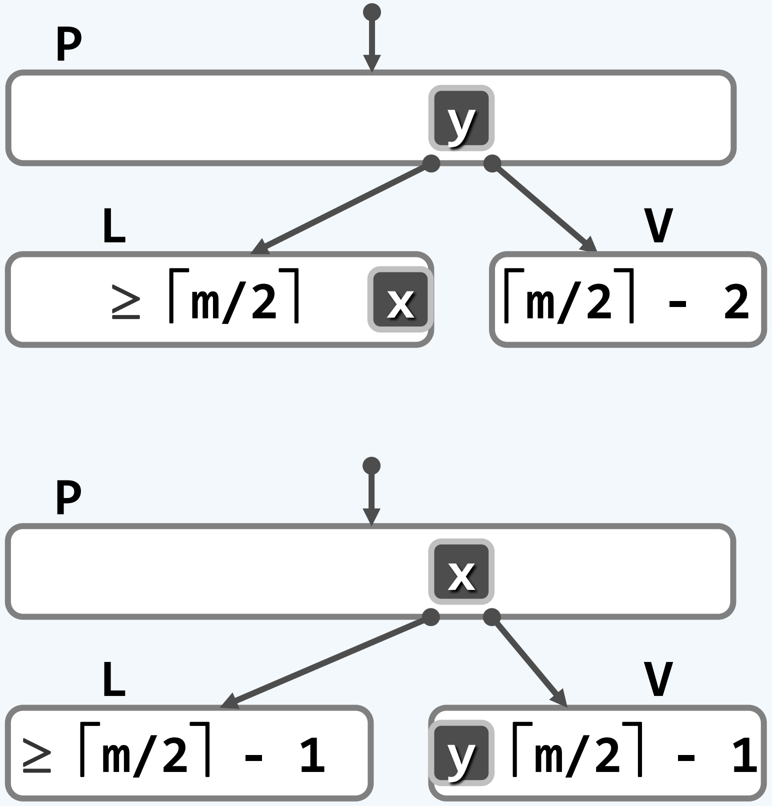
❖ 节点 $\boxed{V}$ 下溢时，必恰好包含：$\boxed{\lceil m/2 \rceil - 2}$ 个关键码 + $\boxed{\lceil m/2 \rceil - 1}$ 个分支

❖ 视其左、右兄弟L、R所含关键码的数目，可分 $\boxed{\text{三种情况}}$ 处理

1)若 $\boxed{L}$ 存在，且至少包含 $\boxed{\lceil m/2 \rceil}$ 个关键码

将关键码 $\boxed{y}$ 从 $\boxed{P}$ 移至 $\boxed{V}$ 中（作为最小关键码）

将关键码 $\boxed{x}$ 从 $\boxed{L}$ 移至 $\boxed{P}$ 中（取代其中原关键码 $\boxed{y}$）



❖ 如此 $\boxed{\text{旋转}}$ 之后，$\boxed{\text{局部}}$ 乃至 $\boxed{\text{整树}}$ 都重新满足B-树条件

下溢修复完毕

2)若 $\boxed{R}$ 存在，且至少包含 $\boxed{\lceil m/2 \rceil}$ 个关键码

完全对称

合并

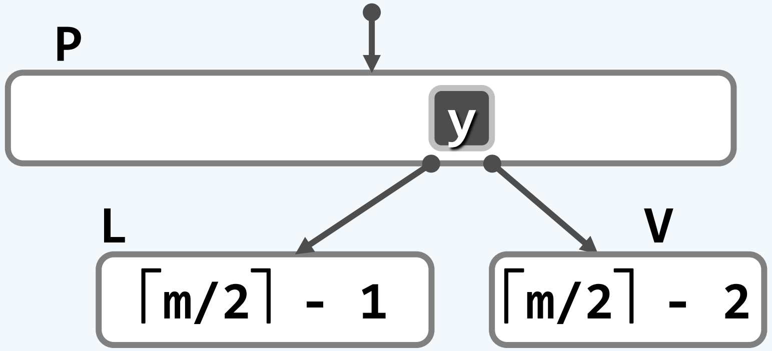
3) L和R或者不存在，或者所含的关键码均不足⌈m/2⌉个

注意，L和R仍必有其一，且恰含 ⌈m/2⌉ - 1 个关键码（不妨以L为例）

❖ 从P中抽出介于L和V之间的关键码y

通过y做粘接，以L和V合成一个节点

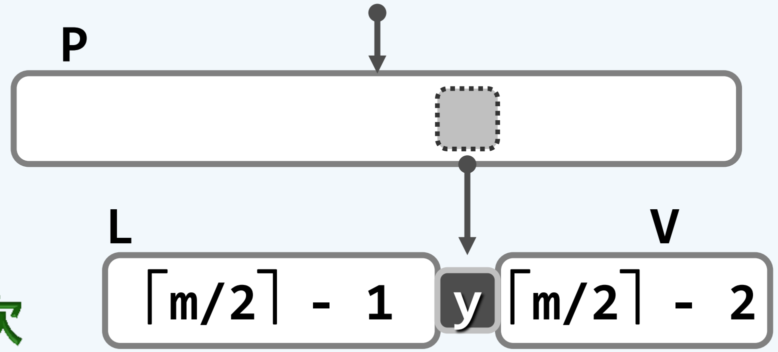
同时合并此前y的孩子引用



❖ 如此合并之后，原高度处的下溢得以修复

但可能导致更高处的P下溢

此时，大可套用前法，继续旋转或合并

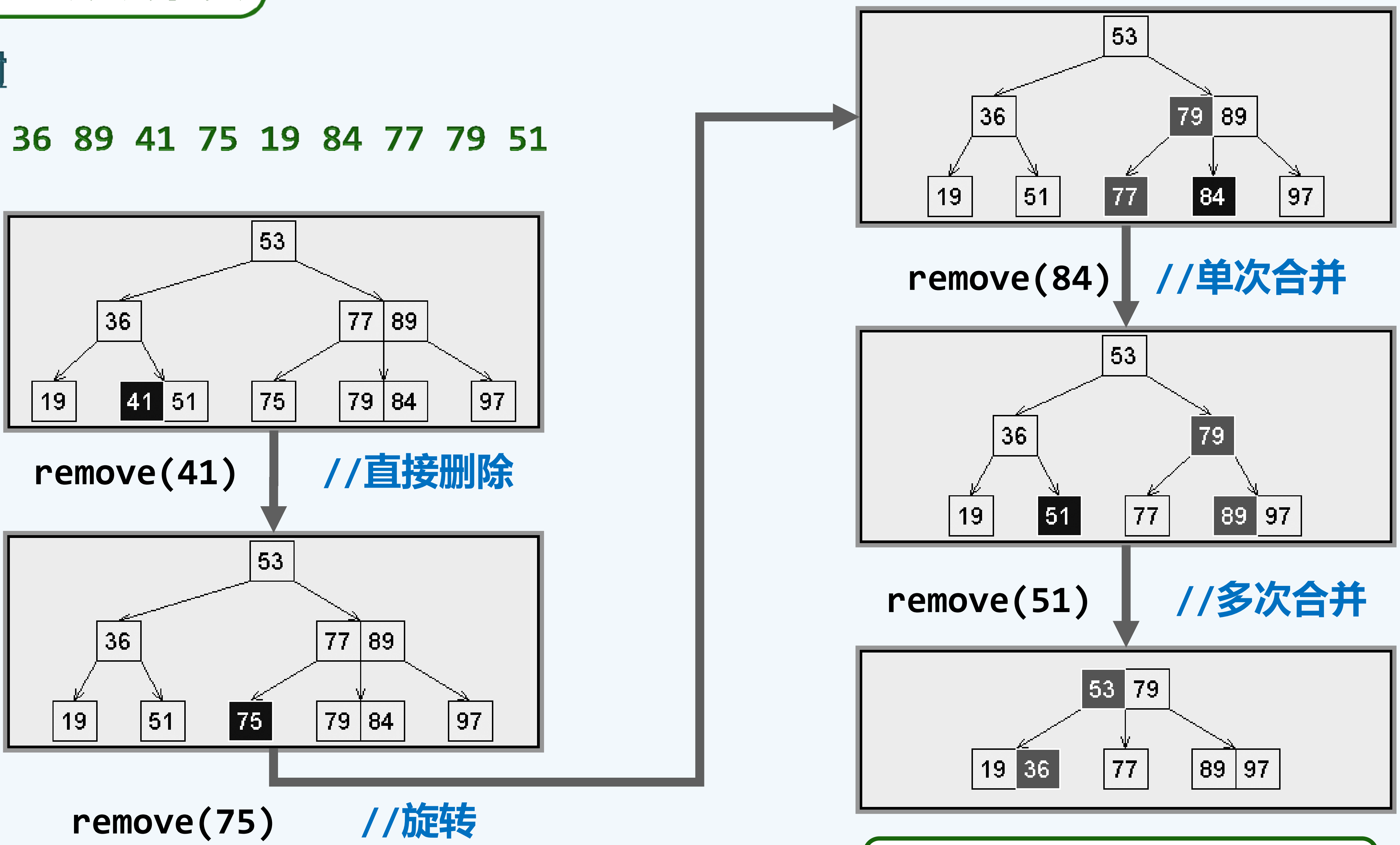


❖ 下溢可能持续发生，并逐层向上传播；但至多不过$O(h)$次

实例：底层节点

❖ 2-3-树

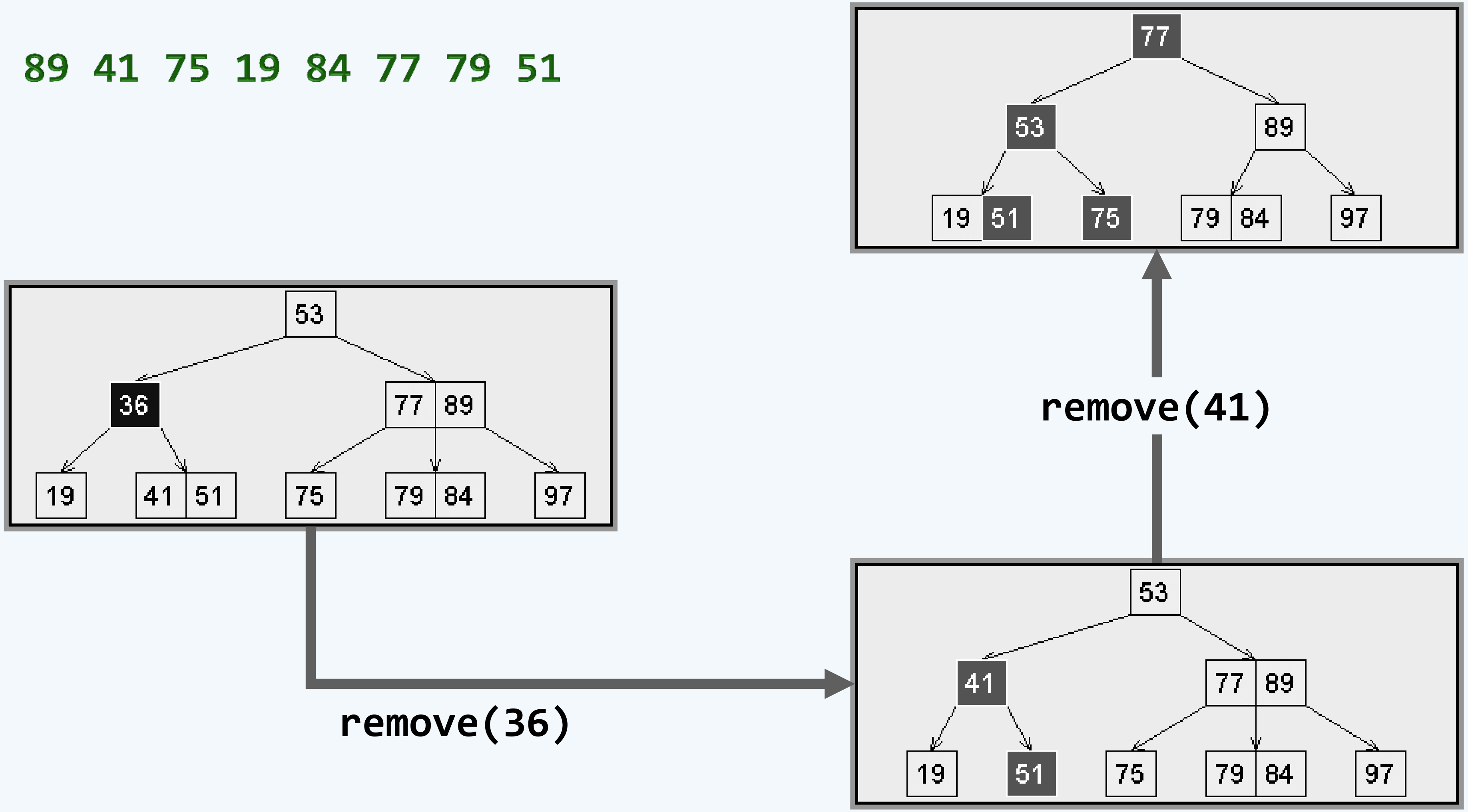
53 97 36 89 41 75 19 84 77 79 51



实例：非底层节点

❖ 2-3-树

53 97 36 89 41 75 19 84 77 79 51



下溢修复

❖
```
template <typename T> void BTree<T>::solveUnderflow( BTNodePosi(T) v ) {
   if ( (_order + 1) / 2 <= v->child.size() ) return; //递归基：v并未下溢
   BTNodePosi(T) p = v->parent; if ( !p ) { /* 递归基：已到根节点 */ }
   Rank r = 0; while ( p->child[r] != v ) r++; //确定v是p的第r个孩子
   if ( 0 < r ) { /* 情况1：若v的左兄弟存在，且... */ }
   if ( p->child.size() - 1 > r ) { /* 情况2：若v的右兄弟存在，且... */ }
   if ( 0 < r ) { /* 与左兄弟合并 */ } else { /* 与右兄弟合并 */ } //情况3
   solveUnderflow( p ); //上升一层，继续分裂——至多递归O(logn)层——典型尾递归
   return;
}
```

下溢修复：旋转

❖ 情况1：向左兄弟借关键码——情况2完全对称

❖
```
if (0 < r) { //若v不是p的第一个孩子，则
   BTNodePosi(T) ls = p->child[r - 1]; //左兄弟必存在
   if ( (_order + 1) / 2 < ls->child.size() ) { //若该兄弟足够“胖”，则
      v->key.insert( 0, p->key[r-1] ); //p借出一个关键码给v（作为最小关键码）
      p->key[r - 1] = ls->key.remove( ls->key.size() - 1 ); //ls的最大关键码转入p
      v->child.insert( 0, ls->child.remove( ls->child.size() - 1 ) );
         //同时ls的最右侧孩子过继给v（作为v的最左侧孩子）
      if ( v->child[0] ) v->child[0]->parent = v;
      return; //至此，通过右旋已完成当前层（以及所有层）的下溢处理
   }
```

下溢修复：合并

```
❖ if (0 < r) { //与左兄弟合并
    BTNodePosi(T) ls = p->child[r-1]; //左兄弟必存在
    ls->key.insert( ls->key.size(), p->key.remove(r - 1) );
    p->child.remove( r ); //p的第r - 1个关键码转入ls，v不再是p的第r个孩子
    ls->child.insert( ls->child.size(), v->child.remove( 0 ) );
    if ( ls->child[ ls->child.size() - 1 ] ) //v的最左侧孩子过继给ls做最右侧孩子
       ls->child[ ls->child.size() - 1 ]->parent = ls;
    /* ... TBC ... */
  } else { /* 与右兄弟合并，完全对称 */ }
```

下溢修复：合并

```
❖ if (0 < r) { //与左兄弟合并
    /* ...... */
    while ( !v->key.empty() ) { //v剩余的关键码和孩子，依次转入ls
      ls->key.insert( ls->key.size(), v->key.remove(0) );
      ls->child.insert( ls->child.size(), v->child.remove(0) );
      if ( ls->child[ ls->child.size() - 1 ] )
        ls->child[ ls->child.size() - 1 ]->parent = ls;
    }
    release(v); //释放v
  } else { /* 与右兄弟合并，完全对称 */ }
```

习题解析

- 举例说明，在最坏情况下，一次插入操作会引发 $\boxed{\Omega(\log n)}$ 次分裂

  在$\boxed{\text{连续}}$的插入操作过程中，发生这种情况的概率有多大？

  就$\boxed{\text{连续}}$意义而言，其间每次插入操作平均会引发多少次分裂？

- 就原理而言，与下溢修复一样，上溢修复即可做旋转，也可做分裂

  试扩充 `BTree::solveOverflow()` 接口，加入这种策略

  这一对称的策略，因何$\boxed{\text{未被}}$普遍采用？

习题解析

- B*-tree

从 独自分裂 到 联合分裂

节点上溢后未必独自分裂，也可由 k 个饱和的兄弟 均摊 新关键码

得到 k + 1 个相邻节点，各含有至少 $\lfloor (m - 1) * k / (k + 1) \rfloor$ 个关键码

如此，可将空间使用率从 50% 提高至 k / (k + 1)