

10-B5

高级搜索树

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 ) {
```

```
    BTreeNodePosi(T) v = search( e );
```

```
    if ( ! v ) return false; //确认e存在
```

```
    Rank r = v->key.search(e); //e在v中的秩
```

```
    if ( v->child[0] ) { //若v非叶子，则
```

```
        BTreeNodePosi(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）
```

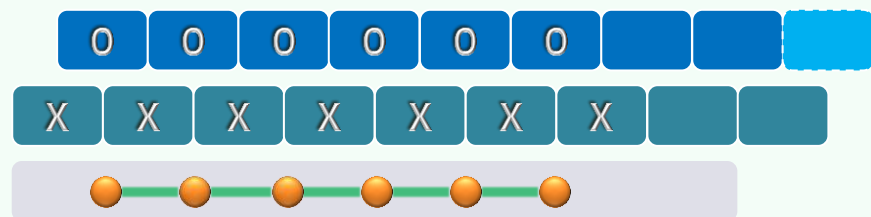
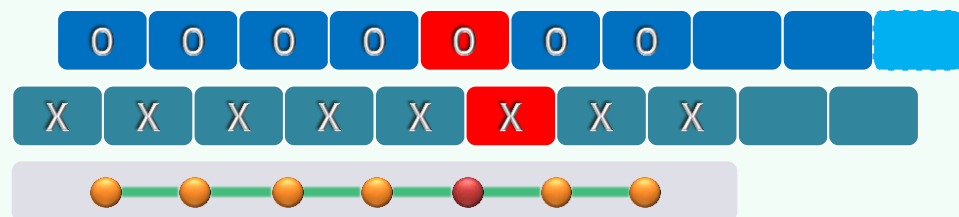
```
        v->key[r] = u->key[0]; v = u; r = 0; //并与之交换位置
```

```
    } //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[V]下溢时，必恰有 $\lceil m/2 \rceil - 2$ 个关键码和 $\lceil m/2 \rceil - 1$ 个分支

❖ 视其左、右兄弟[L]、[R]的规模，可分三种情况加以处理

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

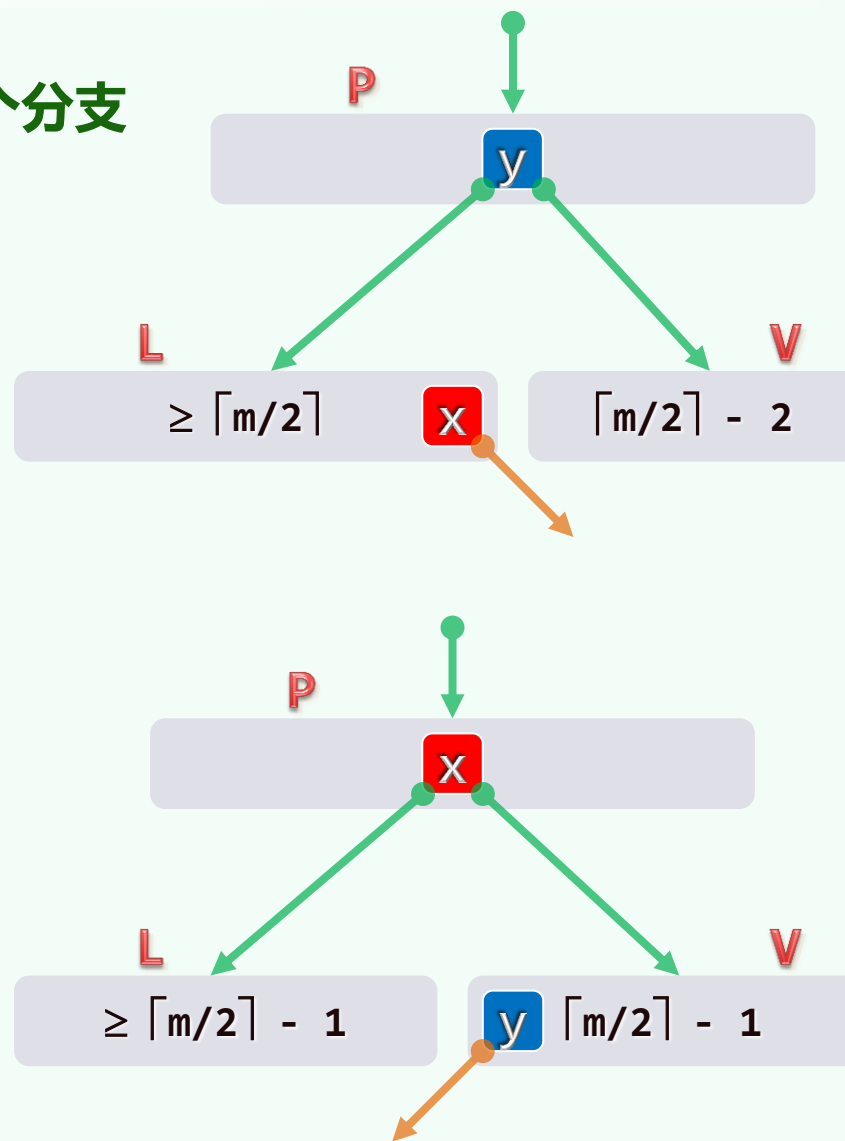
- 将[P]中的分界关键码[y]移至[V]中（作为最小关键码）
- 将[L]中的最大关键码[x]移至[P]中（取代原关键码[y]）

❖ 如此旋转之后，局部乃至全树都重新满足B-树条件

下溢修复完毕

2) 若[R]存在，且至少包含 $\lceil m/2 \rceil$ 个关键码

- 也可旋转，完全对称



合并

3) L 和 R 或不存在，或均不足 $\lceil m/2 \rceil$ 个关键码——即便如此

- L 和 R 仍必有其一（不妨以 L 为例），且
- 恰含 $\lceil m/2 \rceil - 1$ 个关键码

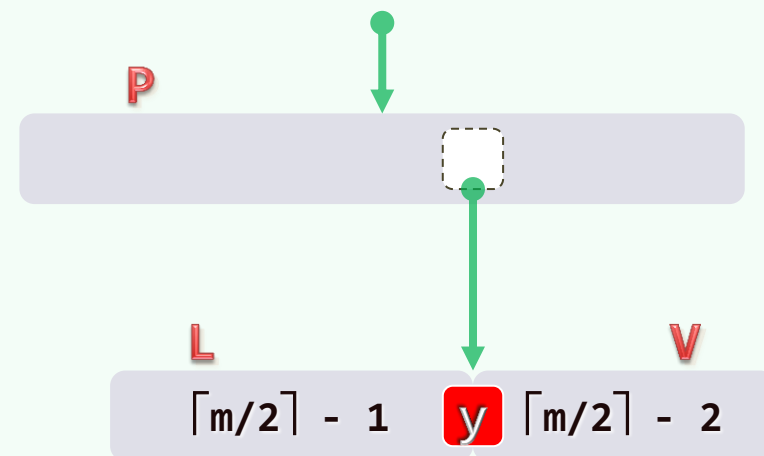
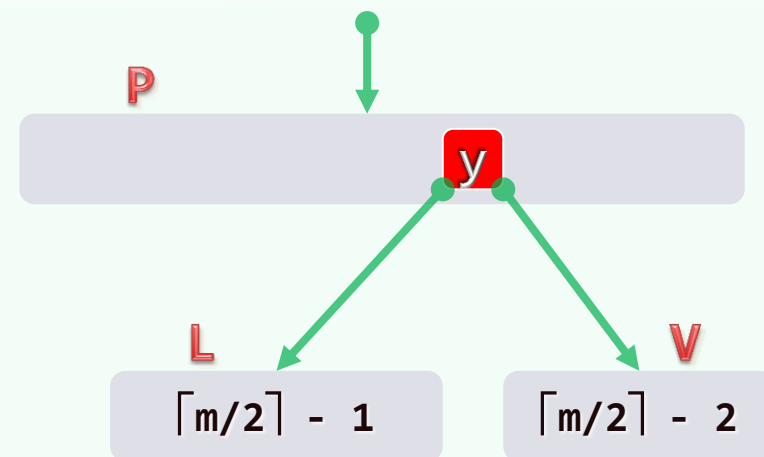
❖ 从 P 中抽出介于 L 和 V 之间的分界关键码 y

- 通过 y 做粘接，将 L 和 V 合成一个节点
- 同时合并此前 y 的孩子引用

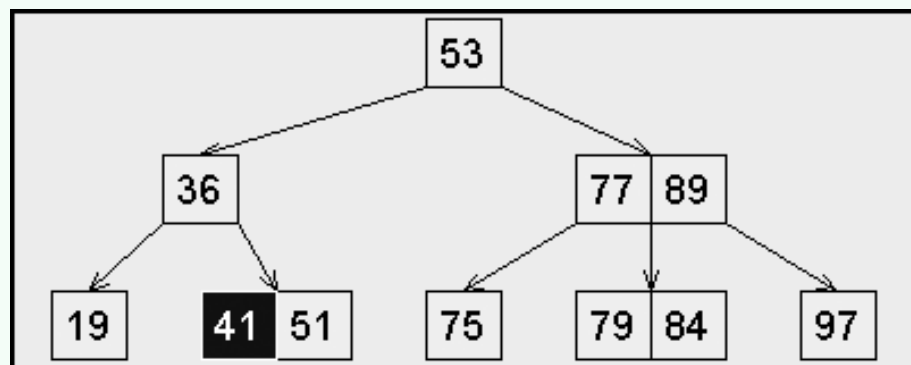
❖ 此处下溢得以修复，但可能继而导致 P 下溢

若果真如此，大可套用前法，继续旋转或合并

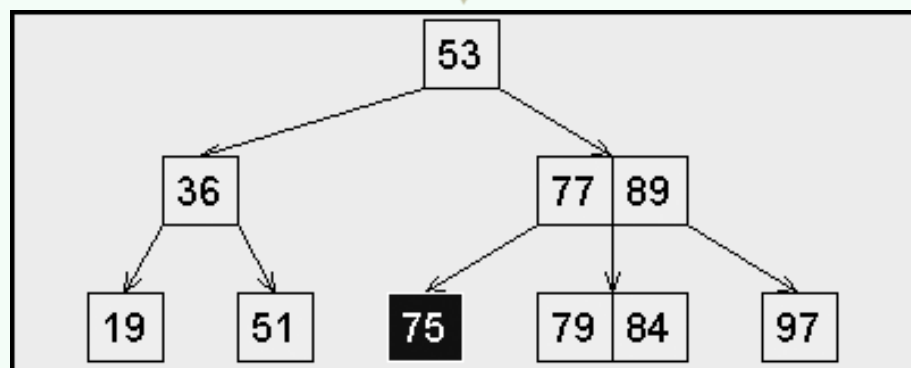
❖ 下溢可能持续发生并向上传播；但至多不过 $O(h)$ 层



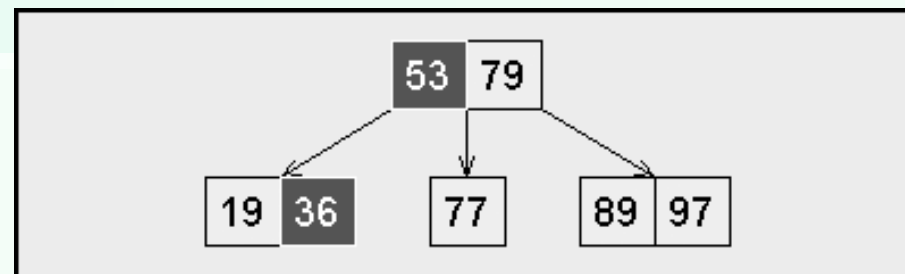
实例：(2,3)-树：底层节点



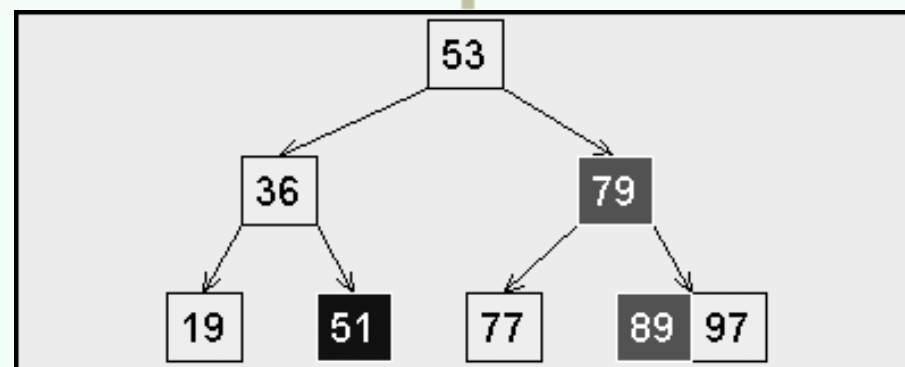
remove(41) //直接删除



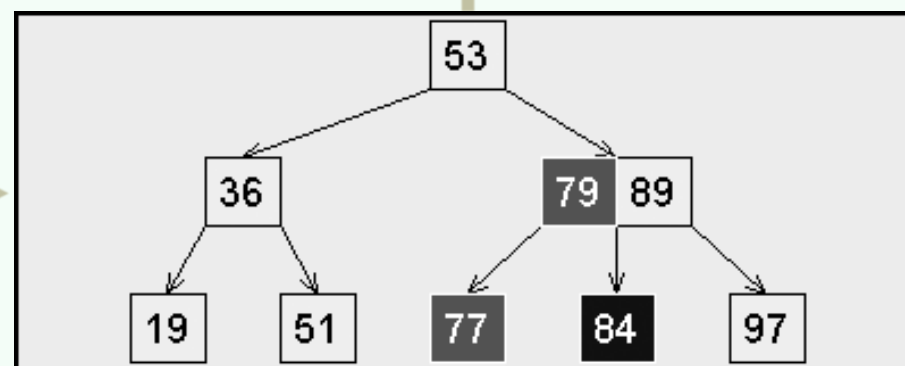
remove(75) //旋转



remove(51) //多次合并



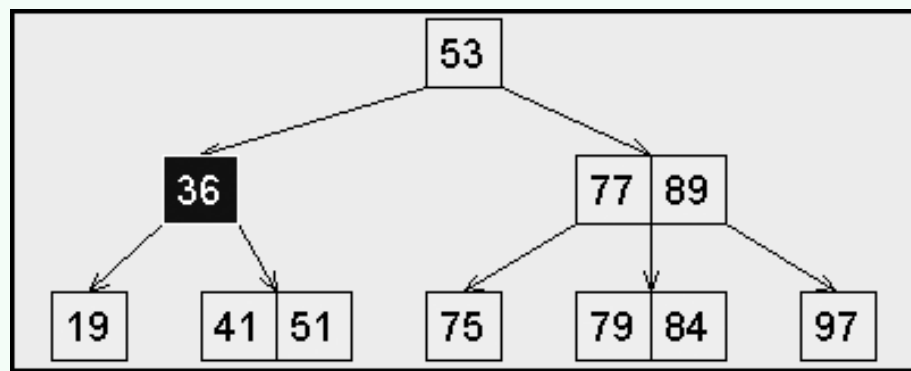
remove(84) //单次合并



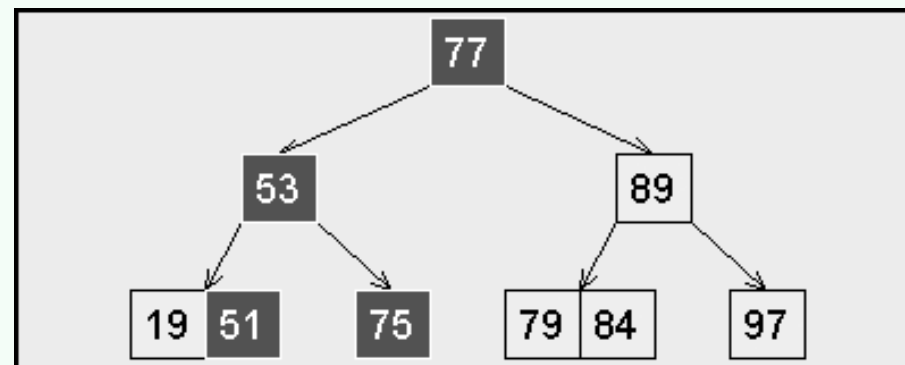
❖ 53 97 36 89 41 75 19 84 77 79 51

实例：(2,3)-树：非底层节点

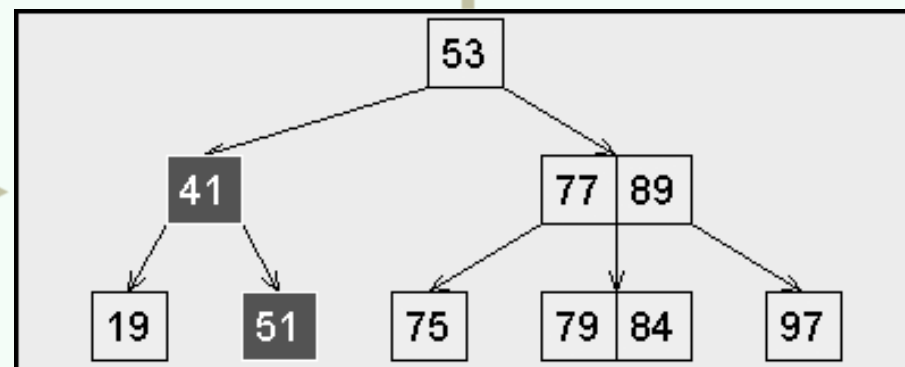
❖ 53 97 36 89 41 75 19 84 77 79 51



remove(36)



remove(41)



下溢修复

```
❖ template <typename T> void BTree<T>::solveUnderflow( BTreeNodePosi(T) v ) {  
    if ( ( _order + 1 ) / 2 <= v->child.size() ) return; //递归基：v并未下溢  
    BTreeNodePosi(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 $O(\log n)$ 层——典型尾递归  
    return;  
}
```

下溢修复：情况#1：旋转（向左兄弟借关键码）

```
❖ if (0 < r) { //若v不是p的第一个孩子，则
    BTreeNodePosi(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
    }
} //情况#2完全对称
```


下溢修复：情况#3：合并（1/2）

❖ 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：情况#3：合并（2/2）

❖ 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 */ }

探究

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

- 在**连续**的插入操作过程中，发生这种情况的概率有多大？
- 就**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，上溢修复既可做旋转，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**未被**普遍采用？

❖ B*-Tree

- **联合分裂**：节点上溢后未必独自分裂，而是由 k 个兄弟共同**均摊**溢出的关键码
- 得到 $k+1$ 个相邻节点，各至少含有 $\lfloor (m-1) \cdot k / (k+1) \rfloor$ 个关键码
- 如此，可将空间使用率从50%提高至 $k/(k+1)$