

8. 高级搜索树

(b4) B-树：插入

说再见，在这梦幻国度，最后的一瞥
清醒让我，分裂再分裂

邓俊辉

deng@tsinghua.edu.cn

算法

❖ template <typename T>

```
bool BTree<T>::insert( const T & e ) {
```

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

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

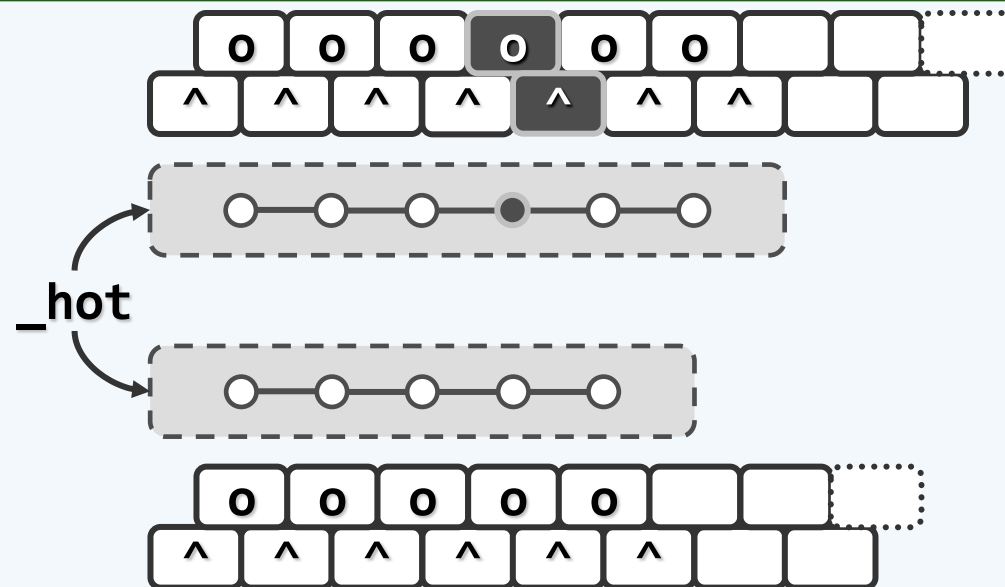
```
    Rank r = _hot->key.search( e ); //在节点_hot中确定插入位置
```

```
    _hot->key.insert( r + 1, e ); //将新关键码插至对应的位置
```

```
    _hot->child.insert( r + 2, NULL ); //创建一个空子树指针
```

```
    _size++; solveOverflow( _hot ); //如发生上溢，需做分裂
```

```
    return true; //插入成功
```



分裂

❖ 设上溢节点中的关键码依次为 k_0, \dots, k_{m-1}

❖ 取中位数 $s = \lfloor m/2 \rfloor$ ，以关键码 k_s 为界划分为

k_0, \dots, k_{s-1} , k_s , k_{s+1}, \dots, k_{m-1}

❖ 关键码 k_s 上升一层，并

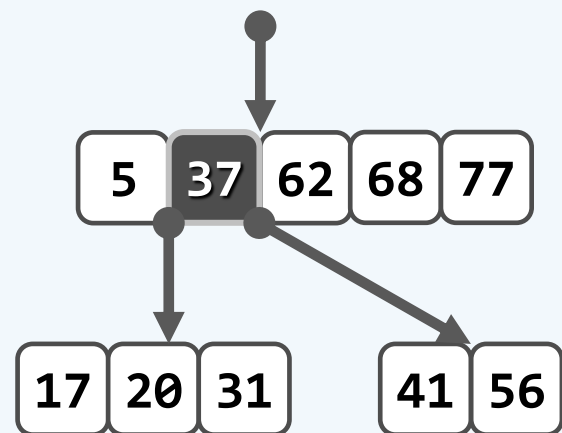
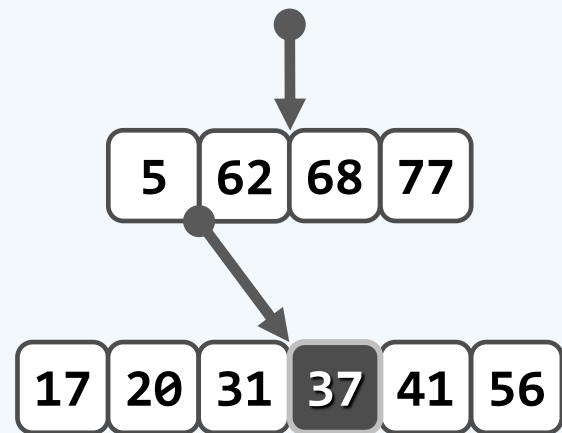
分裂 **split**：以所得的两个节点作为左、右孩子

❖ 确认：如此分裂后

左、右孩子所含 **关键码数** 依然符合 m 阶 B-树 的条件

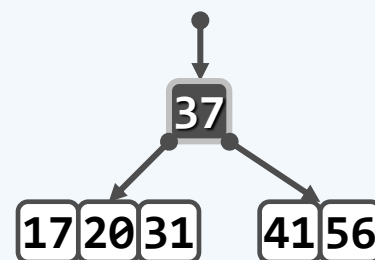
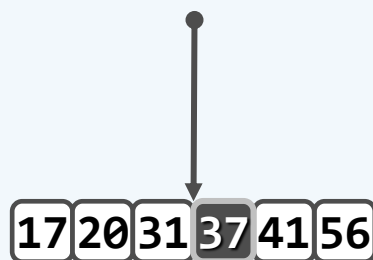
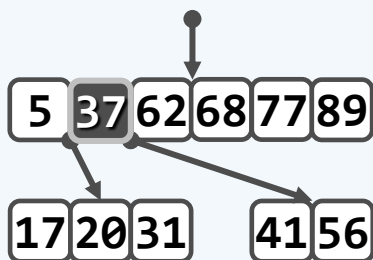
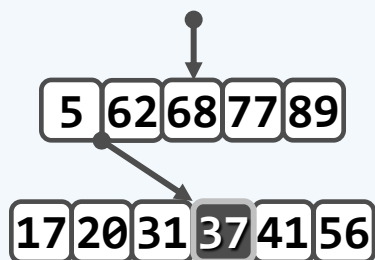
$$s = \lfloor m/2 \rfloor \geq \lceil m/2 \rceil - 1$$

$$m - s - 1 = m - \lfloor m/2 \rfloor - 1 = \lceil m/2 \rceil - 1$$



再分裂

- ❖ 若上溢节点的父亲本已饱和，则在接纳被提升的关键码之后，也将上溢
此时，大可套用前法，继续分裂

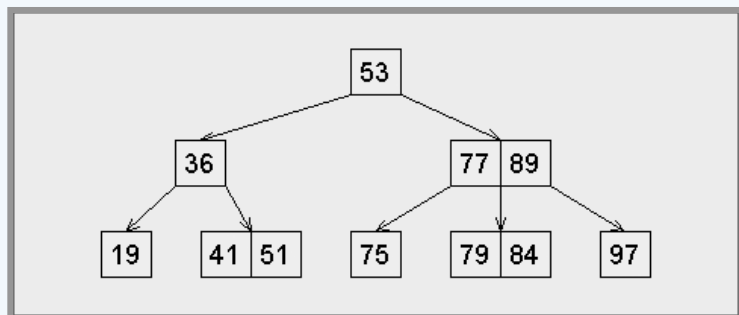


- ❖ 上溢可能持续发生，并逐层向上传播；纵然最坏情况，亦不过到根
- ❖ 若果真抵达树根，可令被提升的关键码自成节点，作为新的树根
这也是B-树增高的唯一可能（具体的概率多大？）
- ❖ 注意：新的树根仅有两个分支——正因如此，树根不必遵守下限 $\lceil m/2 \rceil$
- ❖ 总体执行时间线性正比于分裂次数，不超过 $O(h)$

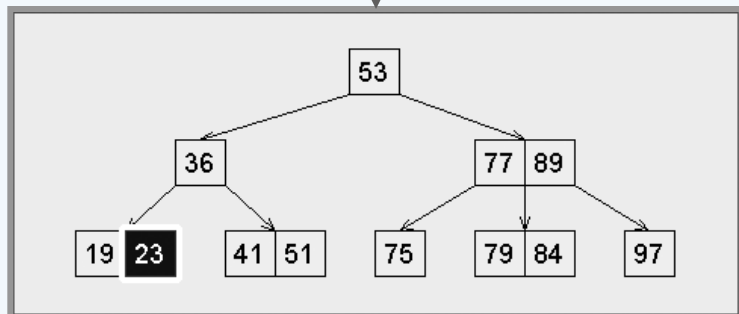
实例

❖ 2-3-树：

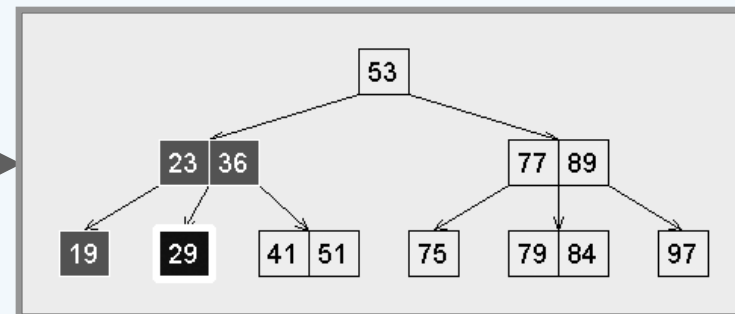
53 97 36 89 41 75 19 84 77 79 51



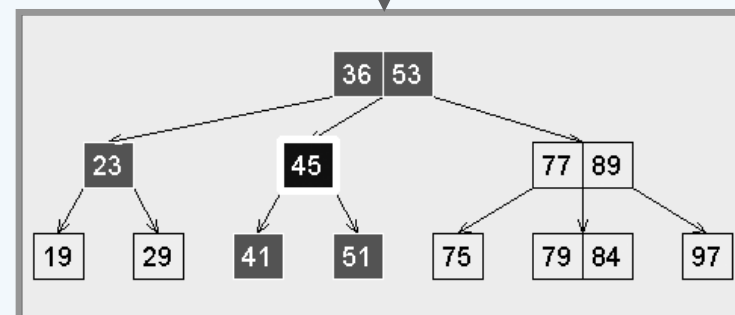
insert(23) //无需分裂



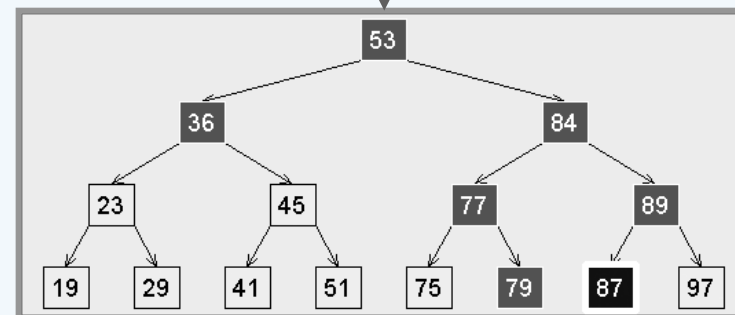
insert(29) //分裂一次



insert(45) //分裂两次



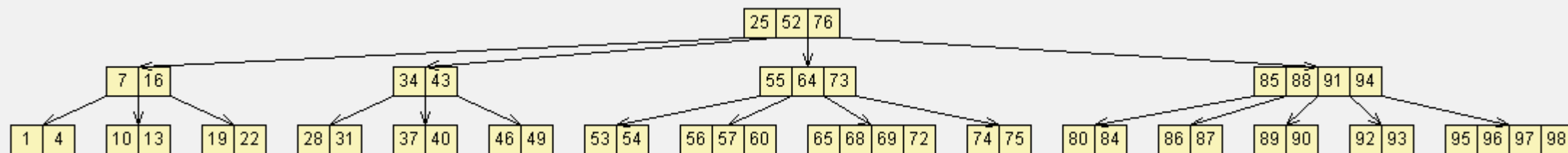
insert(87) //分裂到根



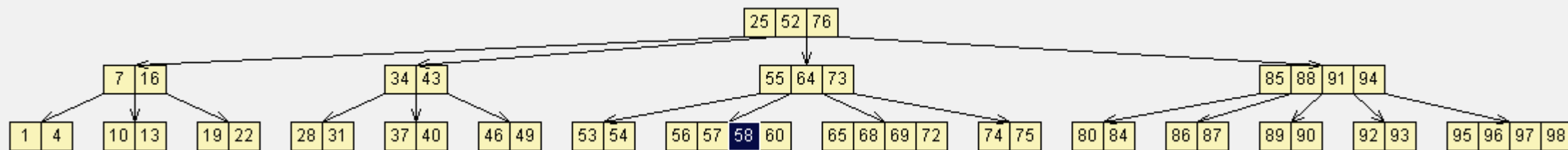
实例

3-5-树

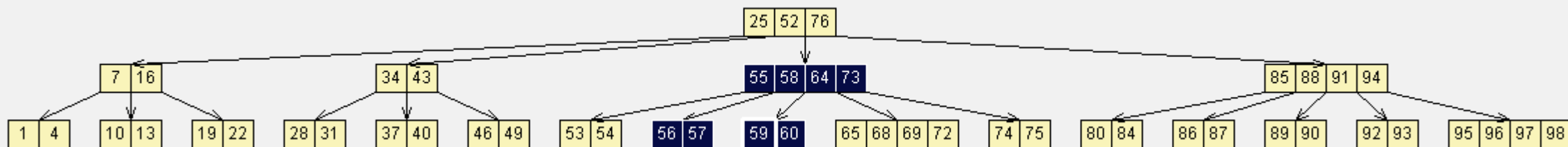
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 56 60 64 68 72 76 80 84 53 54 55 85 86 87 88 89 90 91 92 93 94 95 96 97 98 73 74 75 57 65 69



insert(58) //无需分裂

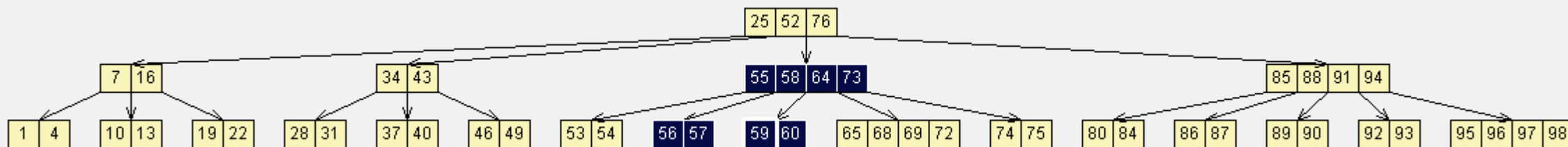


insert(59) //分裂 1 次

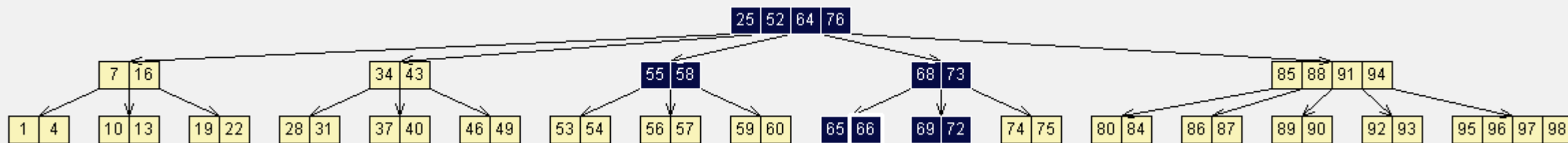


实例

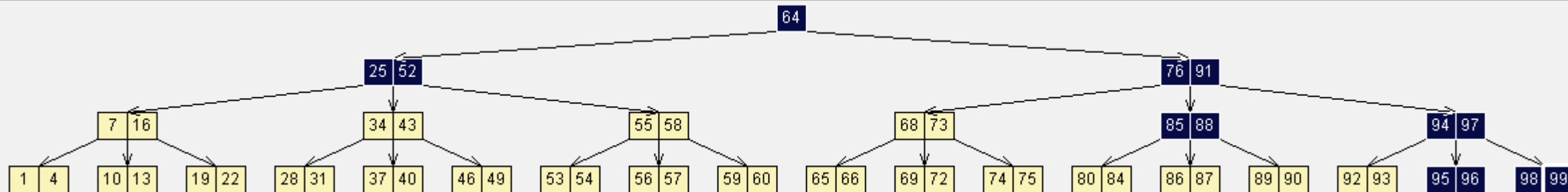
insert(59) //分裂 1 次



insert(66) //分裂 2 次



insert(99) //分裂到根



上溢修复

```
❖ template <typename T> void BTree<T>::solveOverflow( BTreeNodePosi(T) v ) {  
    if ( _order >= v->child.size() ) return; //递归基：不再上溢  
  
    Rank s = _order / 2; //轴点（此时_order = key.size() = child.size() - 1）  
  
    BTreeNodePosi(T) u = new BTreeNode<T>(); //注意：新节点已有一个空孩子  
  
    for ( Rank j = 0; j < _order - s - 1; j++ ) { //分裂出右侧节点u（效率低可改进）  
        u->child.insert( j, v->child.remove( s + 1 ) ); //v右侧_order-s-1个孩子  
        u->key.insert( j, v->key.remove( s + 1 ) ); //v右侧_order-s-1个关键码  
    }  
  
    u->child[_order - s - 1] = v->child.remove( s + 1 ); //移动v最靠右的孩子
```

/* TBC */

上溢修复

❖ if (u->child[0]) //若u的孩子们非空，则统一令其以u为父节点

for (Rank j = 0; j < `_order - s`; j++) u->child[j]->parent = u;

BTNodePosi(T) p = v->parent; //v当前的父节点p

if (! p) //若p为空，则创建之（全树长高一层，新根节点恰好两度）

{ _root = p = new BTNode<T>(); p->`child[0]` = v; v->parent = p; }

Rank `r` = 1 + p->`key`.search(v->`key[0]`); //p中指向u的指针的秩

p->`key`.insert(`r`, v->`key.remove`(`s`)); //轴点关键码上升

p->`child`.insert(`r + 1`, u); u->parent = p; //新节点u与父节点p互联

solveOverflow(p); //上升一层，如有必要则继续分裂——至多递归 $O(\log n)$ 层

}