08-B3

高级搜索树

B-树：结构

邓俊辉

deng@tsinghua.edu.cn

妻子好合，如鼓瑟琴；兄弟既翕，和乐且湛

等价变换

❖ 平衡的多路搜索树

R. Bayer & E. McCreight

1970

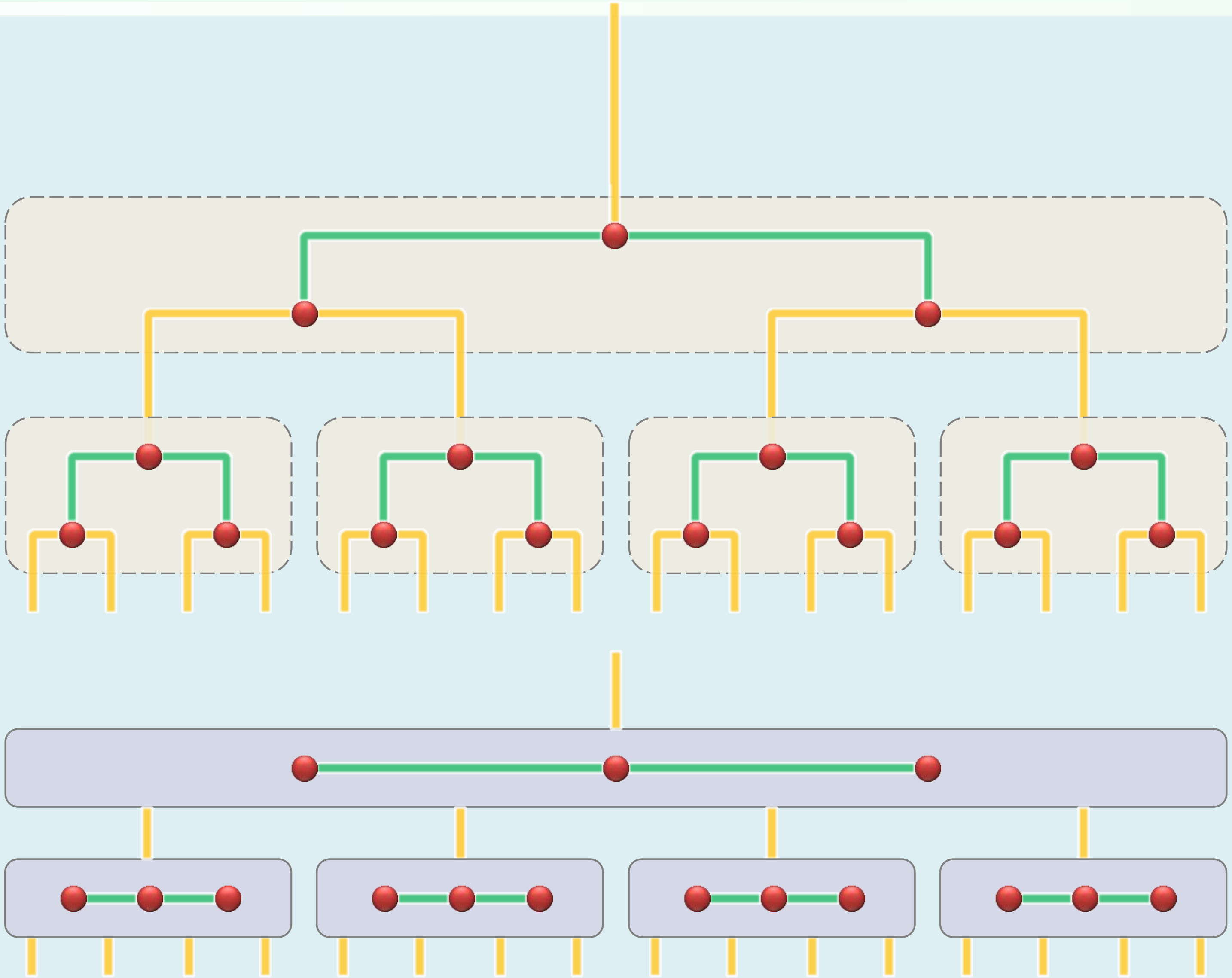
❖ 每d代合并为**超级节点**

- m = 2^d 路

- m-1 个关键码

❖ 逻辑上与BBST**完全等价**

既如此，B-树之意义何在?



I/O优化：多级存储系统中使用B-树，可针对外部查找，大大减少I/O次数

❖ 难道，AVL还不够？比如，若有n = 1G个记录...

- 每次查找需要 $\log_2 10^9 \approx 30$ 次I/O操作
- 每次只读出单个关键码，得不偿失

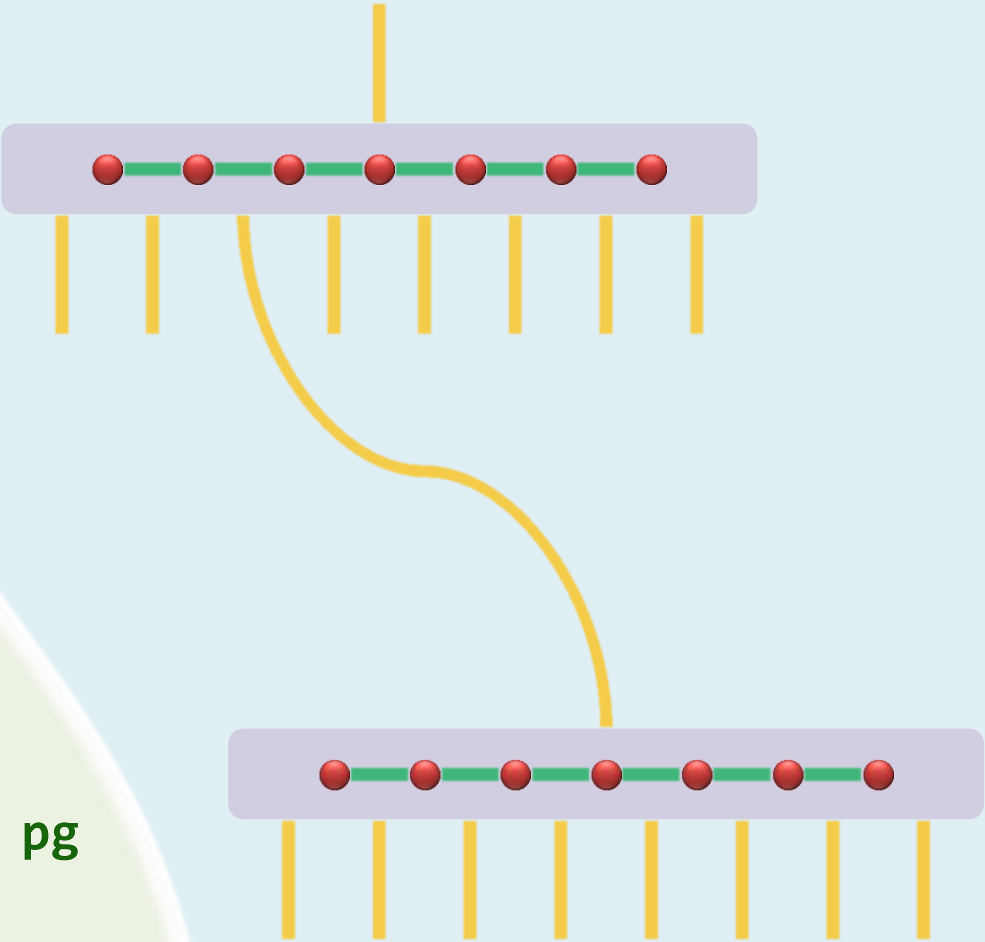
❖ B-树又能如何？

- 充分利用外存的批量访问，将此特点转化为优点
- 每下降一层，都以超级节点为单位，读入一组关键码

❖ 具体多大一组？视磁盘的数据块大小而定，m = #keys / pg

- 比如，目前多数数据库系统采用 m = 200~300

❖ 回到上例，若取m = 256，则每次查找只需 $\log_{256} 10^9 \le 4$ 次I/O



外部节点 + 叶子

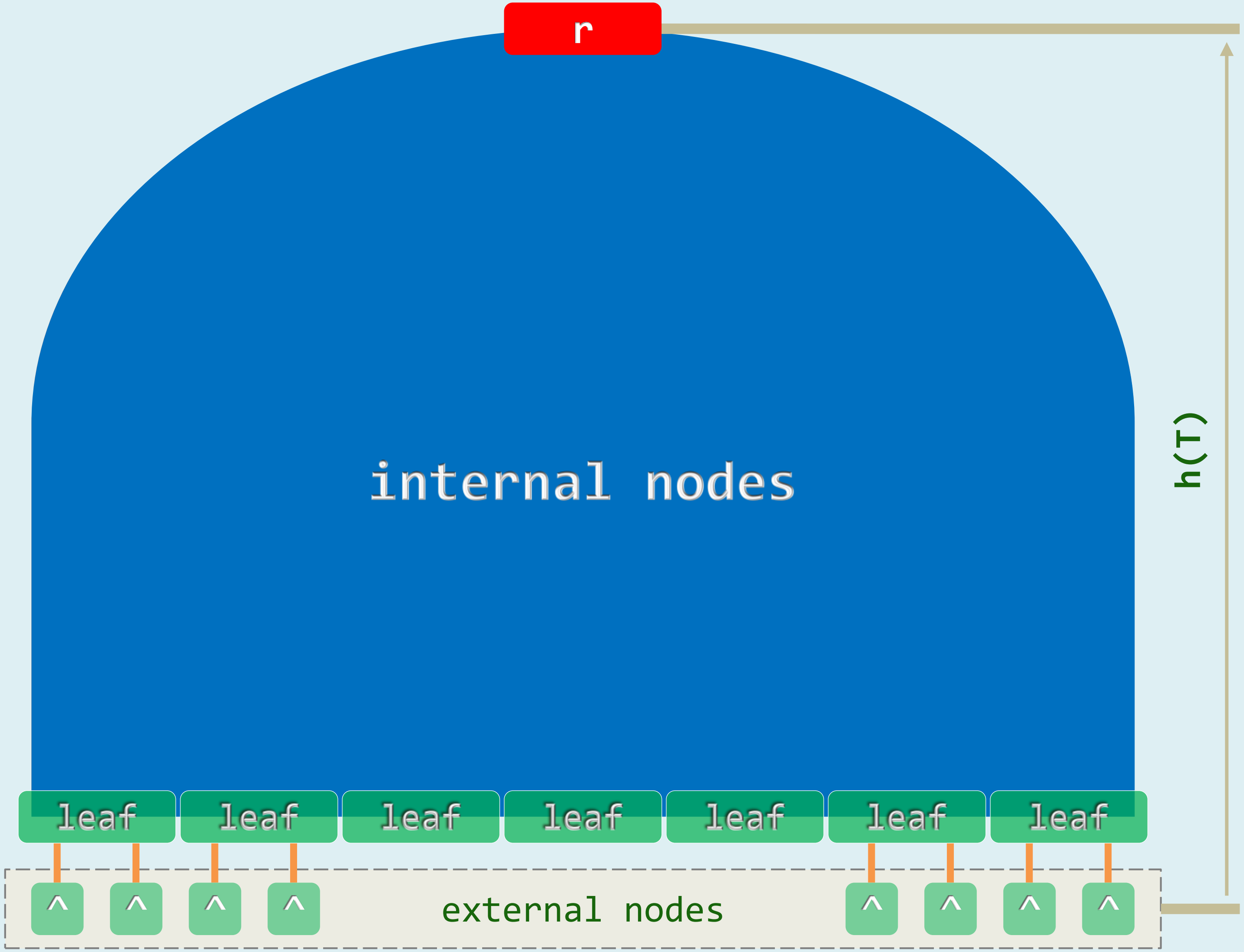
❖ 所谓m阶B-树，即

m路完全平衡搜索树（m ≥ 3）

❖ 外部节点的深度统一相等

约定以此深度作为树高h

❖ 叶节点的深度统一相等（h-1）



内部节点

- 各含 n ≤ m-1 个关键码：

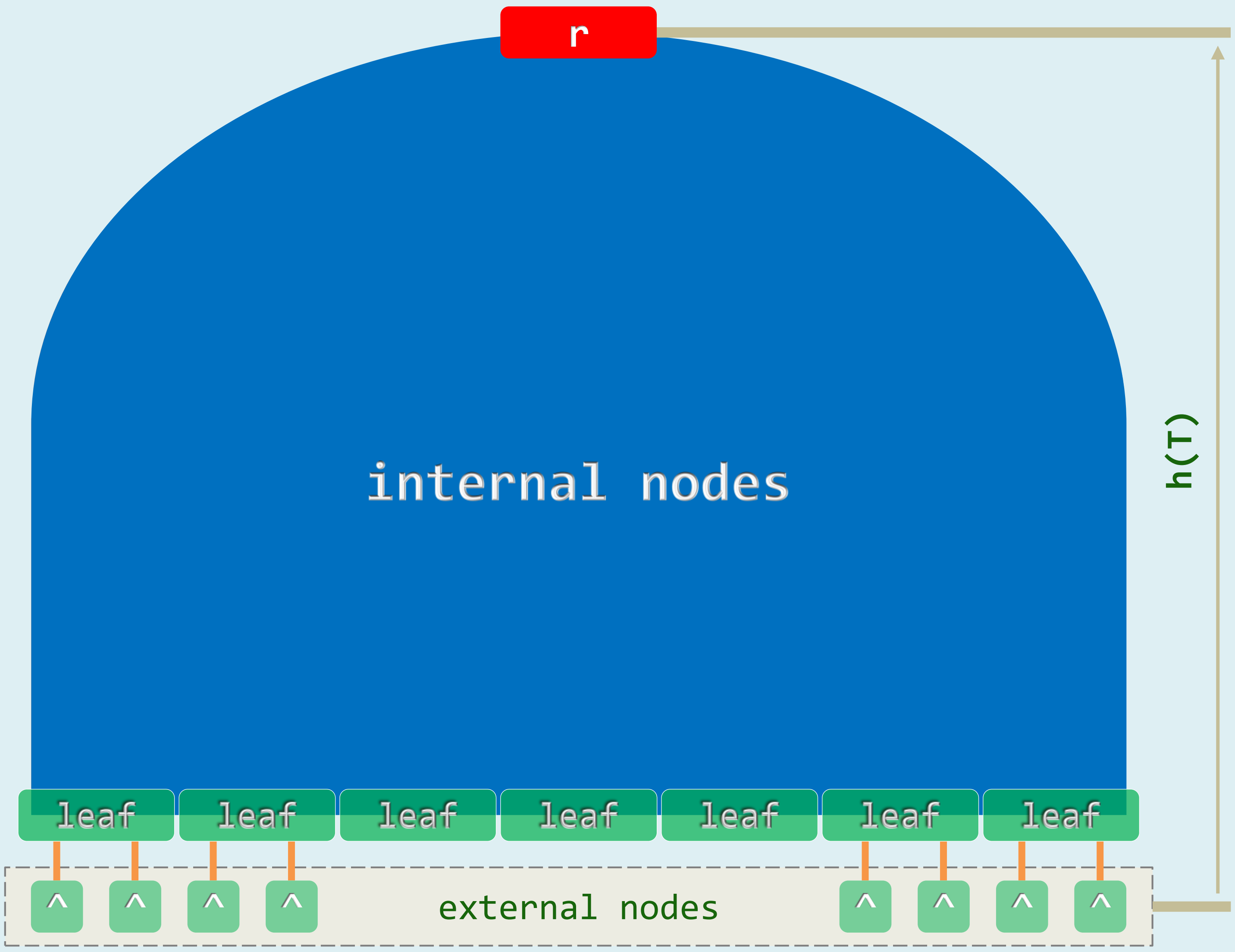
  $$K_1 < K_2 < K_3 < \cdots < K_n$$

- 各有 $n+1 \le m$ 个分支：

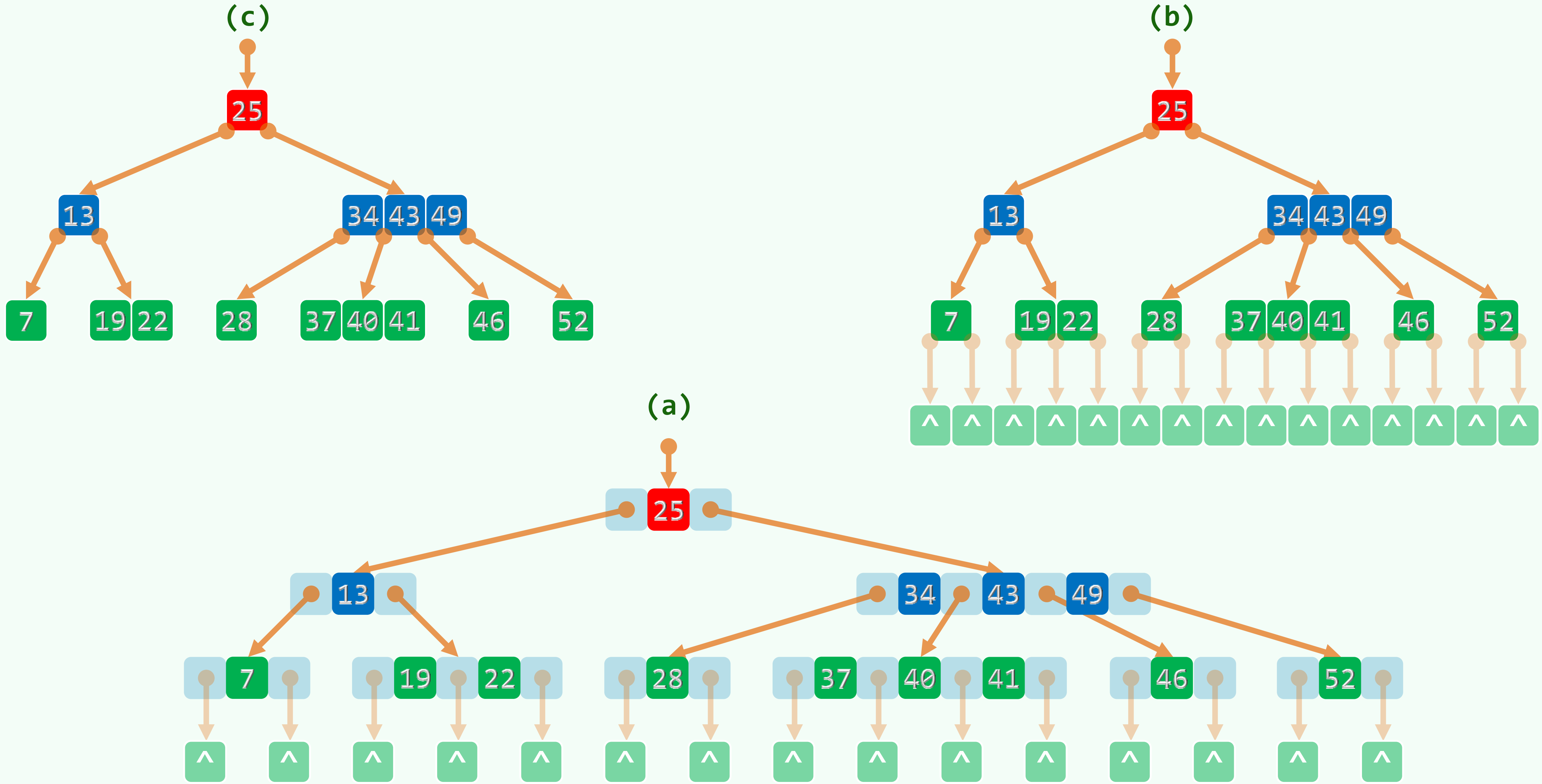
  $$A_0,\ A_1,\ A_2,\ A_3,\ \ldots,\ A_n$$

- 反过来，分支数也不能太少
  - 树根：$2 \le n+1$
  - 其余：$\lceil m/2 \rceil \le n+1$

- 故亦称作 $(\lceil m/2 \rceil, m)$- 树
  - (3,5)-树
  - (9,18)-树
  - ...



紧凑表示


(c)
25
13
34 43 49
7
19 22
28
37 40 41
46
52
(b)
25
13
34 43 49
7
19 22
28
37 40 41
46
52
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
(a)
25
13
34 43 49
7
19 22
28
37 40 41
46
52
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^

实例

❖ m = 3: 2-3-树，(2,3)-树

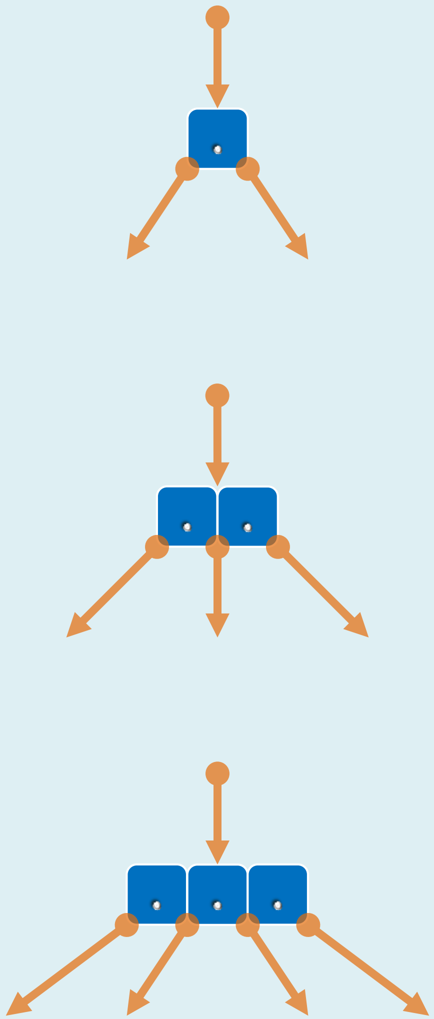
最简单的B-树 //J. Hopcroft, 1970

- 各（内部）节点的分支数，可能是2或3
- 各节点所含key的数目，可能是1或2

❖ m = 4: 2-3-4-树，(2,4)-树

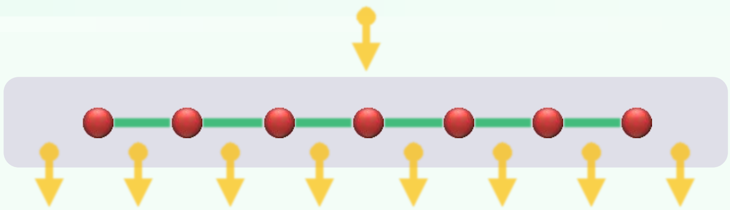
- 各节点的分支数，可能是2、3或4
- 各节点所含key的数目，可能是1、2或3

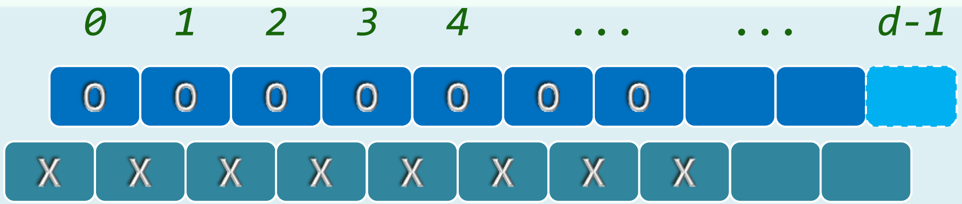
❖ 留意把玩4阶B-树，稍后对于理解红黑树大有裨益

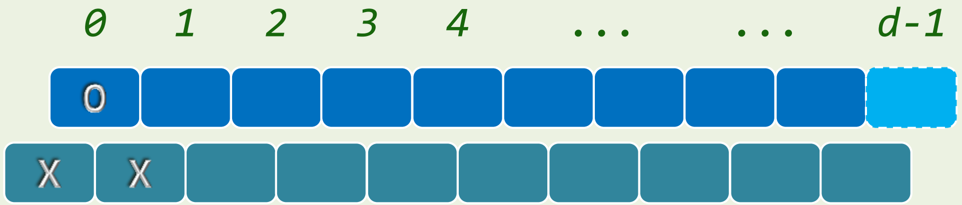


BTNode

```
template <typename T> struct BTNode { //B-树节点
   BTNodePosi<T> parent; //父
   Vector<T> key; //关键码（总比孩子少一个）
   Vector< BTNodePosi<T> > child; //孩子
   BTNode() { parent = NULL; child.insert( NULL ); }
   BTNode( T e, BTNodePosi<T> lc = NULL, BTNodePosi<T> rc = NULL ) {
      parent = NULL; //作为根节点
      key.insert( e ); //仅一个关键码，以及
      child.insert( lc ); if ( lc ) lc->parent = this; //左孩子
      child.insert( rc ); if ( rc ) rc->parent = this; //右孩子
   }
};
```







BTree

```
template <typename T> using BTNodePosi = BTNode<T>*; //B-树节点位置
template <typename T> class BTree { //B-树
protected:
   int _size; int _m; BTNodePosi<T> _root; //关键码总数、阶次、根
   BTNodePosi<T> _hot; //search()最后访问的非空节点位置
   void solveOverflow( BTNodePosi<T> ); //因插入而上溢后的分裂处理
   void solveUnderflow( BTNodePosi<T> ); //因删除而下溢后的合并处理
public:
   BTNodePosi<T> search( const T & e ); //查找
   bool insert( const T & e ); //插入
   bool remove( const T & e ); //删除
};
```