8.高级搜索树

（b2）B-树：结构

妻子好合，如鼓瑟琴

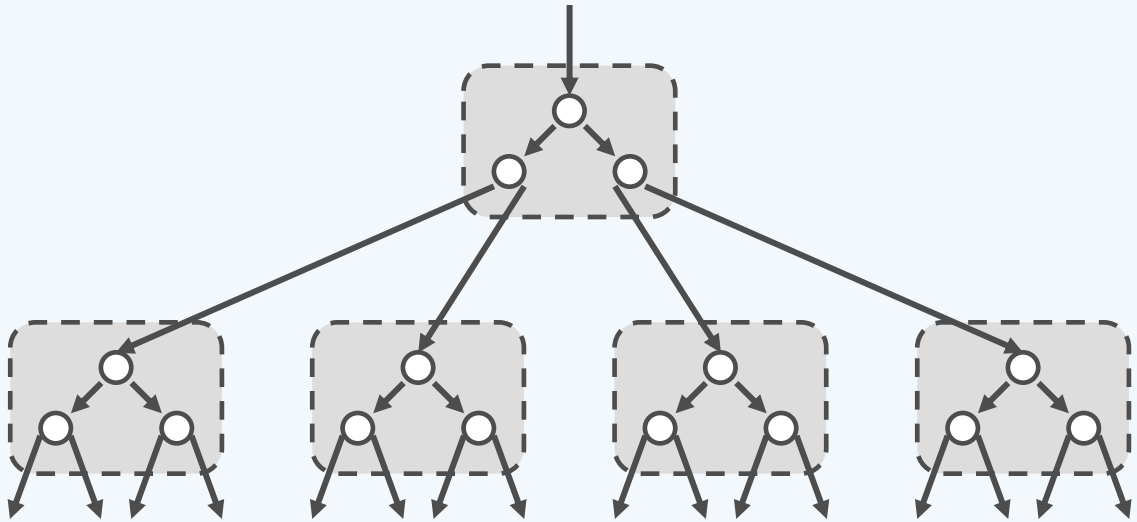
兄弟既翕，和乐且湛

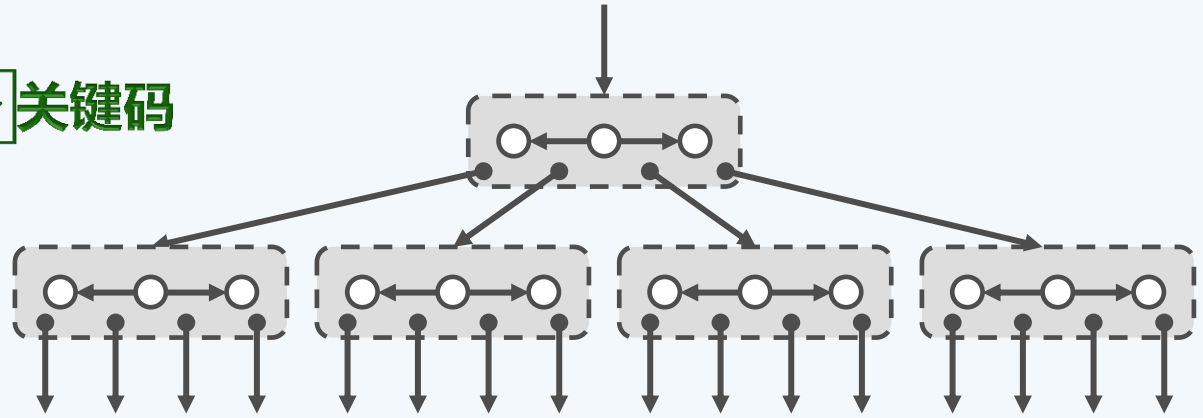
邓俊辉

deng@tsinghua.edu.cn

B-Tree

- 1970, R. Bayer & E. McCreight
- 平衡的多路（multi-way）搜索树
- 经适当合并，得超级节点
  - 每2代合并：4路
  - 每3代合并：8路
  - ...
  - 每d代合并：m = 2^d路，m – 1个关键码
- 逻辑上与BBST完全等价
  ——既然如此，为何还要引入B-树？





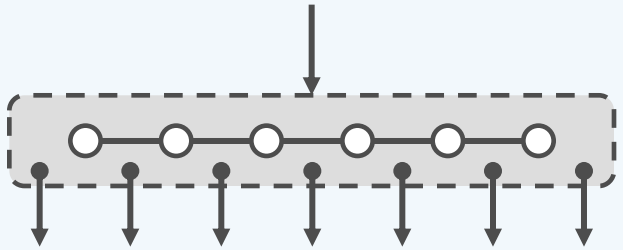
B-Tree

- 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) = 30 次I/O操作，每次只读出一个关键码，得不偿失
- B-树又能如何？

  充分利用外存对批量访问的高效支持，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



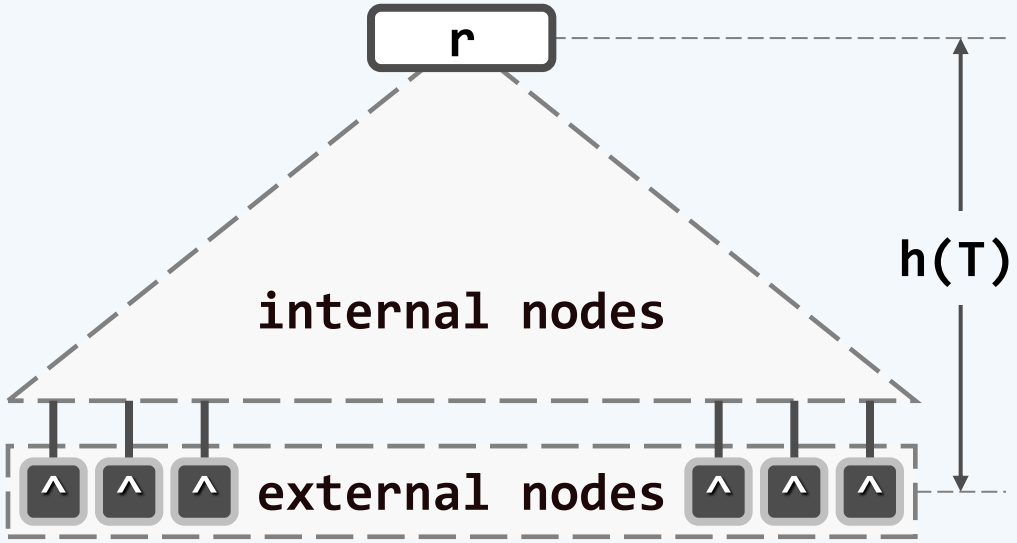
- 具体多大一组？视磁盘的数据块大小而定，m = #keys / pg

  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) <= 4 次I/O

B-Tree

- 所谓m阶B-树，即m路平衡搜索树（m ≥ 2）
- 外部节点的深度统一相等

  所有叶节点的深度统一相等
- 树高h = 外部节点的深度



B-Tree

❖ 内部节点各有

不超过 $m - 1$ 个关键码：$K_1 < K_2 < \ldots < K_n$

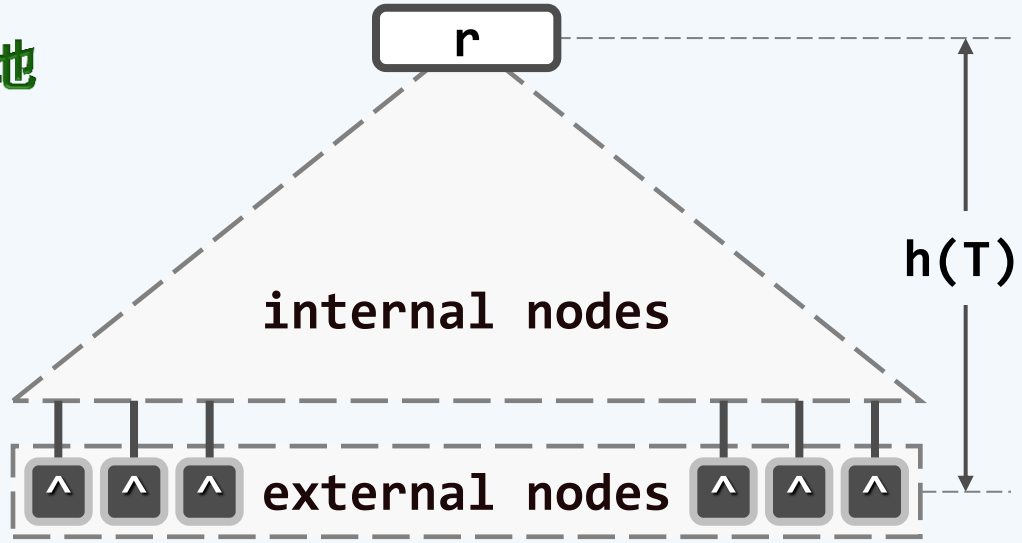
不超过 m 个分支：$A_0 , A_1 , A_2 , \ldots , A_n$

❖ 内部节点的分支数 $n + 1$ 也不能太少，具体地

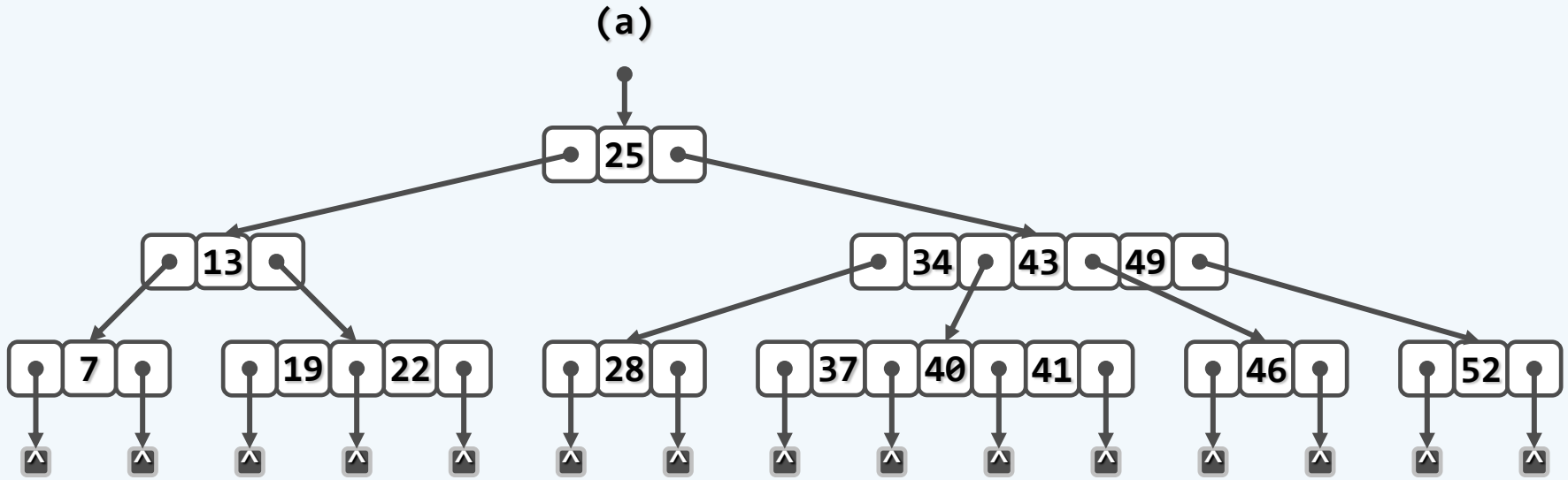
树根：$2 \le n + 1$

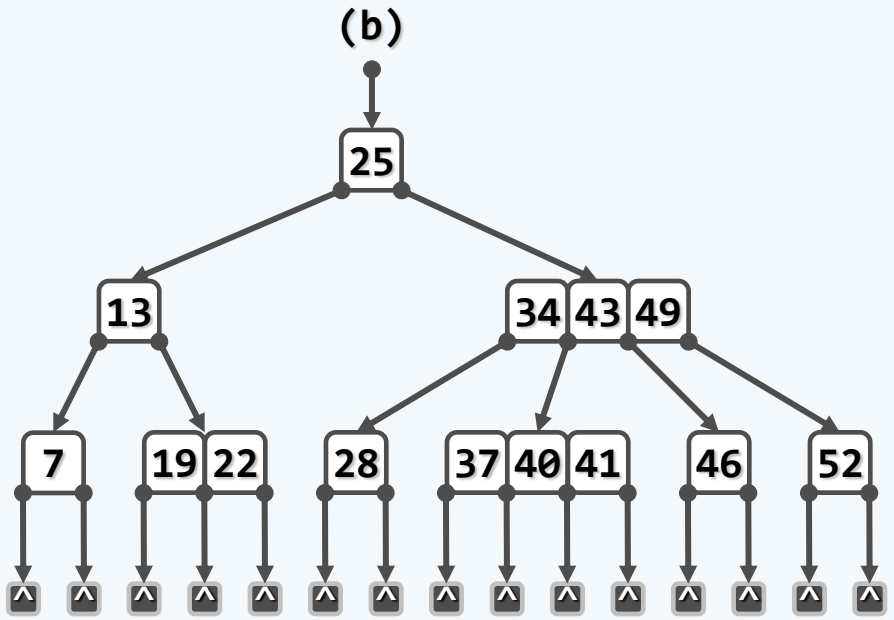
其余：$\lceil m/2 \rceil \le n + 1$

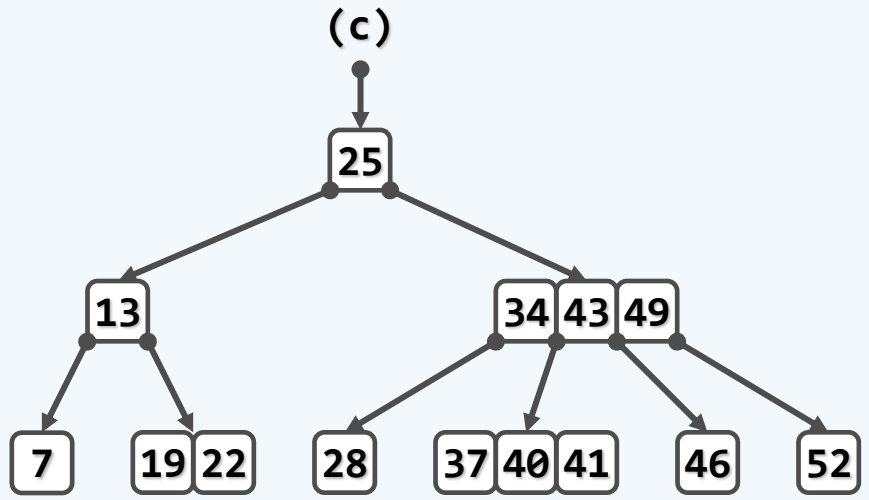
❖ 故亦称作 $(\lceil m/2 \rceil, m)$-树



紧凑表示


(a)
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


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

实例

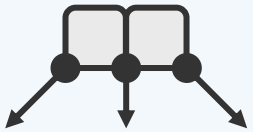
- demo/b_tree/

- m = 3

  2-3-树，(2,3)-树，最简单的B-树 //John 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

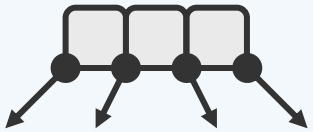


- m = 4

  2-3-4-树，(2,4)-树

  各节点的分支数，可能是2、3或4

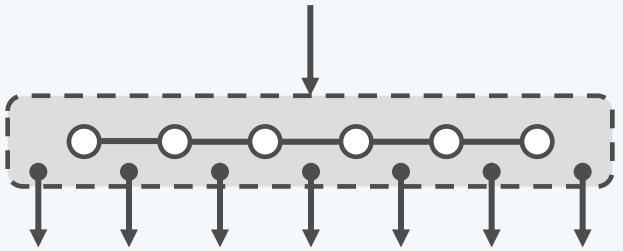
  各节点所含key的数目，可能是1、2或3

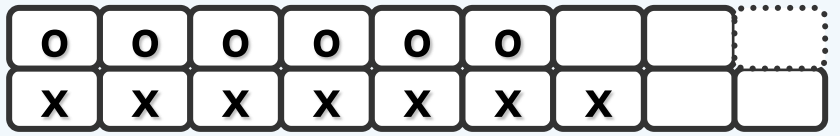


- 留意把玩4阶B-树，对稍后理解红黑树大有裨益

BTNode

```
❖ template <typename T> struct BTNode { //B-树节点
     BTNodePosi(T) parent; //父
     Vector<T> key; //数值向量
     Vector< BTNodePosi(T) > child; //孩子向量（其长度总比key多一）
     BTNode() { parent = NULL; child.insert( 0, NULL ); }
     BTNode( T e, BTNodePosi(T) lc = NULL, BTNodePosi(T) rc = NULL ) {
        parent = NULL; //作为根节点，而且初始时
        key.insert( 0, e ); //仅一个关键码，以及
        child.insert( 0, lc ); child.insert( 1, rc ); //两个孩子
        if ( lc ) lc->parent = this; if ( rc ) rc->parent = this;
     }
```





BTree

- `#define BTNodePosi(T) BTNode<T>* //B-树节点位置`
- `template <typename T> class BTree { //B-树`

```
protected:
    int _size; int _order; 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
};
```