B Trees

Zhengwei Qi

8.高级搜索树

(b1) B-树：动机

640K ought to be enough for anybody.

- B. Gates, 1981

邓俊辉

deng@tsinghua.edu.cn

越来越小的内存

- RAM: 存储器？不就是无限可数个寄存器吗？
- Turing: 存储器？不就是无限长的纸带吗？
- 但事实上
  - 系统存储容量的增长速度
  - << 应用问题规模的增长速度

越来越小的内存

- 典型的
  数据库规模 / 内存容量

  1980 : 10MB / 1MB = 10

  2000 : 1TB / 1GB = 1000

- 今天典型的数据集
  须以TB为单位度量

  345 TB ^ Global climate

  300 TB ^ Nuclear

  250 TB ^ Turbulent combustion

  50 TB ^ Parkinson's disease

  10 TB ^ Protein folding

- 亦即，相对而言...内存容量是在...不断减小！
- 为什么不把内存做得更大?
- 物理上，存储器的容量越大/小，访问速度就越慢/快

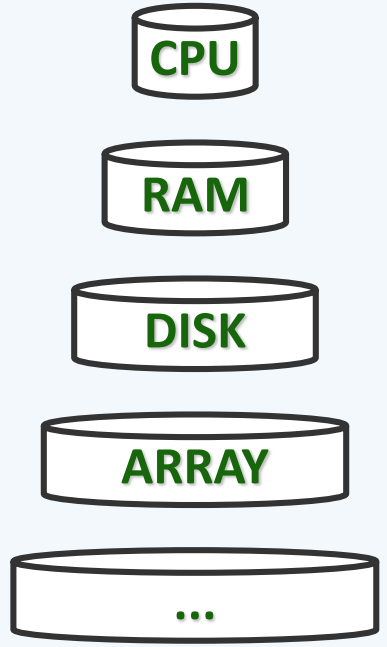
高速缓存

- 事实1：不同容量的存储器，访问速度差异悬殊
- 以磁盘与内存为例：ms / ns > 10^5
- 若一次内存访问需要一秒，则一次外存访问就相当于一天
- 为避免1次外存访问，我们宁愿访问内存10次、100次，甚至...
- 多数存储系统，都是分级组织的——Caching

  最常用的数据尽可能放在更高层、更小的存储器中

  实在找不到，才向更低层、更大的存储器索取
- 算法的I/O复杂度 ∝ 数据在不同存储级别之间的传输次数

  算法的实际运行时间，往往主要取决于此



存储金字塔

Arch:
纳秒级

新型IO:
微秒级

传统软件针对毫秒级IO进行优化

传统IO:
毫秒级

Operation	Time
L1 cache reference	1.5 ns
L2 cache reference	5 ns
Branch misprediction	6 ns
Uncontended mutex lock/unlock	20 ns
L3 cache reference	25 ns
Main memory reference	100 ns
Decompress 1 KB with Snappy [Sna]	500 ns
"Far memory"/Fast NVM reference	1,000 ns (1us)
Compress 1 KB with Snappy [Sna]	2,000 ns (2us)
Read 1 MB sequentially from memory	12,000 ns (12 us)
SSD Random Read	100,000 ns (100 us)
Read 1 MB bytes sequentially from SSD	500,000 ns (500 us)
Read 1 MB sequentially from 10Gbps network	1,000,000 ns (1 ms)
Read 1 MB sequentially from disk	10,000,000 ns (10 ms)
Disk seek	10,000,000 ns (10 ms)
Send packet California→Netherlands→California	150,000,000 ns (150 ms)

The increasing gap in performance between DRAM and disks, and the growing imbalance between throughput and capacity of modern disk drives makes the storage subsystem a common performance bottleneck in large-scale systems

Source: The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition

高速缓存

❖ 事实2：从磁盘中读写1B，与读写1KB几乎一样快

❖ 批量式访问：以页（page）或块（block）为单位，使用缓冲区 //<stdio.h>...

❖


```
#define BUFSIZ 512 //缓冲区默认容量
int setvbuf( //定制缓冲区
    FILE* fp, //流
    char* buf, //缓冲区
    int _Mode, //_IOFBF | _IOLBF | _IONBF
    size_t size); //缓冲区容量
int fflush(FILE* fp); //强制清空缓冲区
```

❖ 效果：单位字节的平均访问时间大大缩短

8.高级搜索树

（b2）B-树：结构

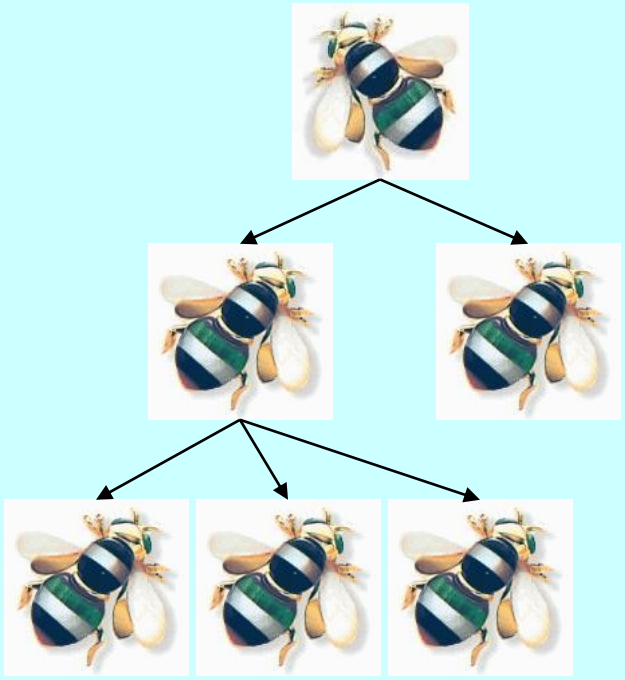
妻子好合，如鼓瑟琴

兄弟既翕，和乐且湛

邓俊辉

deng@tsinghua.edu.cn

B-Trees



Organization and Maintenance of Large Ordered Indexes

R. BAYER and E. MCCREIGHT

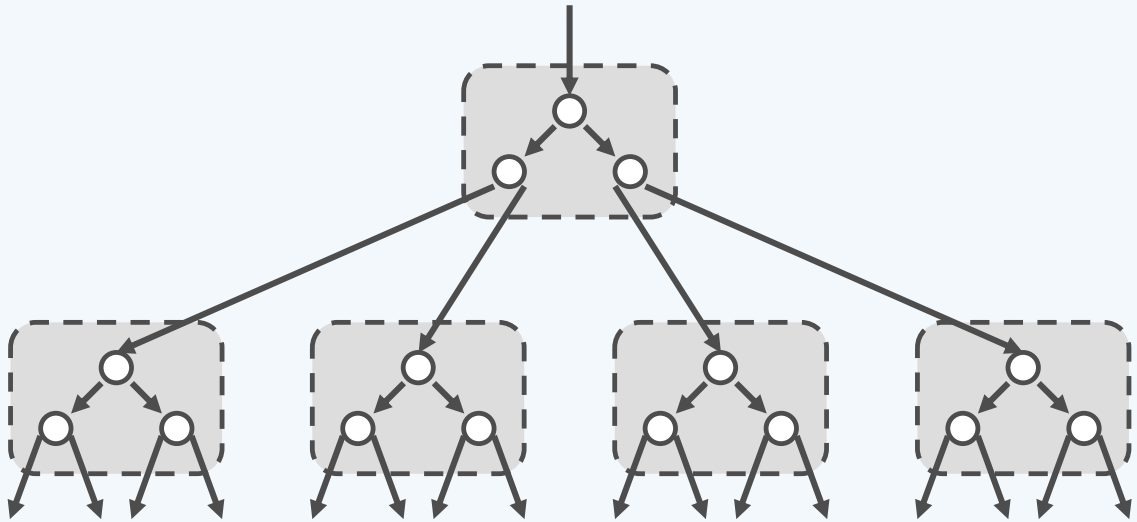
Received September 29, 1971

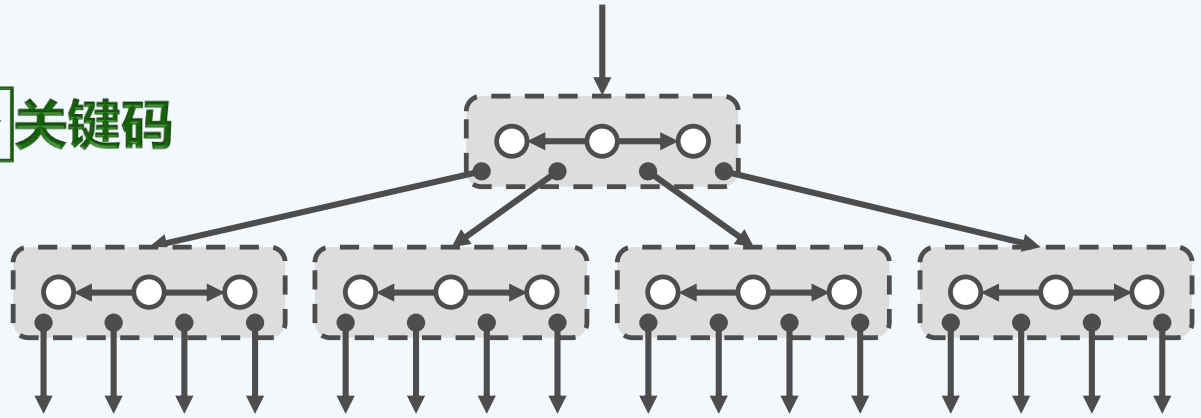
Summary. Organization and maintenance of an index for a dynamic random access file is considered. It is assumed that the index must be kept on some pseudo random access backup store like a disc or a drum. The index organization described allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ where I is the size of the index and k is a device dependent natural number such that the performance of the scheme becomes near optimal. Storage utilization is at least 50% but generally much higher. The pages of the index are organized in a special data-structure, so-called B-trees. The scheme is analyzed, performance bounds are obtained, and a near optimal k is computed. Experiments have been performed with indexes up to 100000 keys. An index of size 15000 (100000) can be maintained with an average of 9 (at least 4) transactions per second on an IBM 360/44 with a 2311 disc.

The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees. (Comer 1979, p. 123 footnote 1)

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？





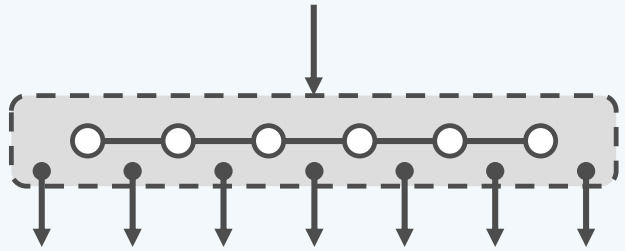
B-Tree

- 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

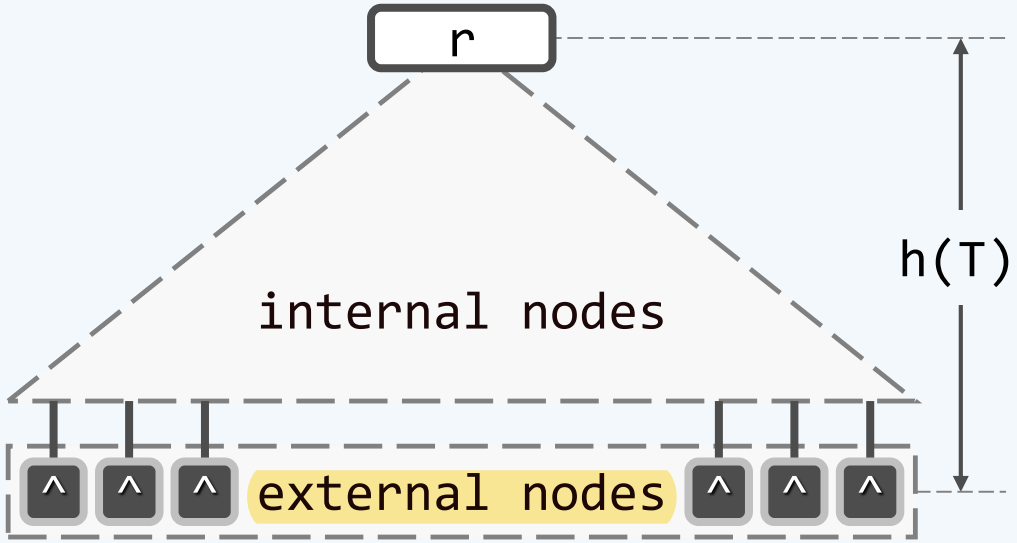


- 具体多大一组？视磁盘的数据块大小而定，m = #keys / pg

  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

- 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



❖ 内部节点各有

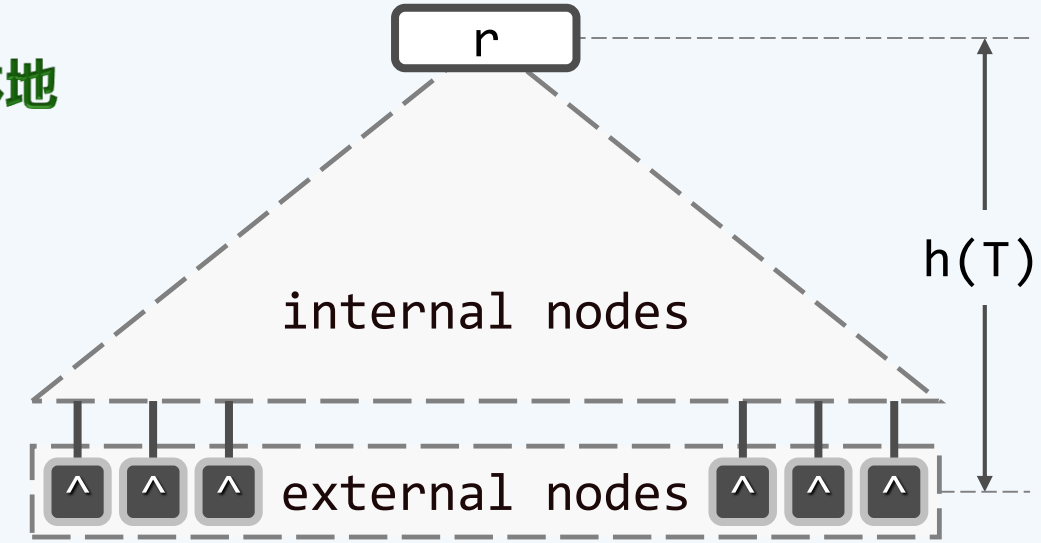
不超过 $m - 1$ 个关键码：$K_1 < K_2 < \ldots < K_n$

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

B-Trees

2. *B*-Trees

Def. 2.1. Let $h \geqq 0$ be an integer, k a natural number. A directed tree T is in the class $\tau(k, h)$ of *B-trees* if T is either empty ($h = 0$) or has the following properties:

i) Each path from the root to any leaf has the same length h, also called the *height* of T, i.e., $h =$ number of nodes in path.

ii) Each node except the root and the leaves has at least $k+1$ sons. The root is a leaf or has at least two sons.

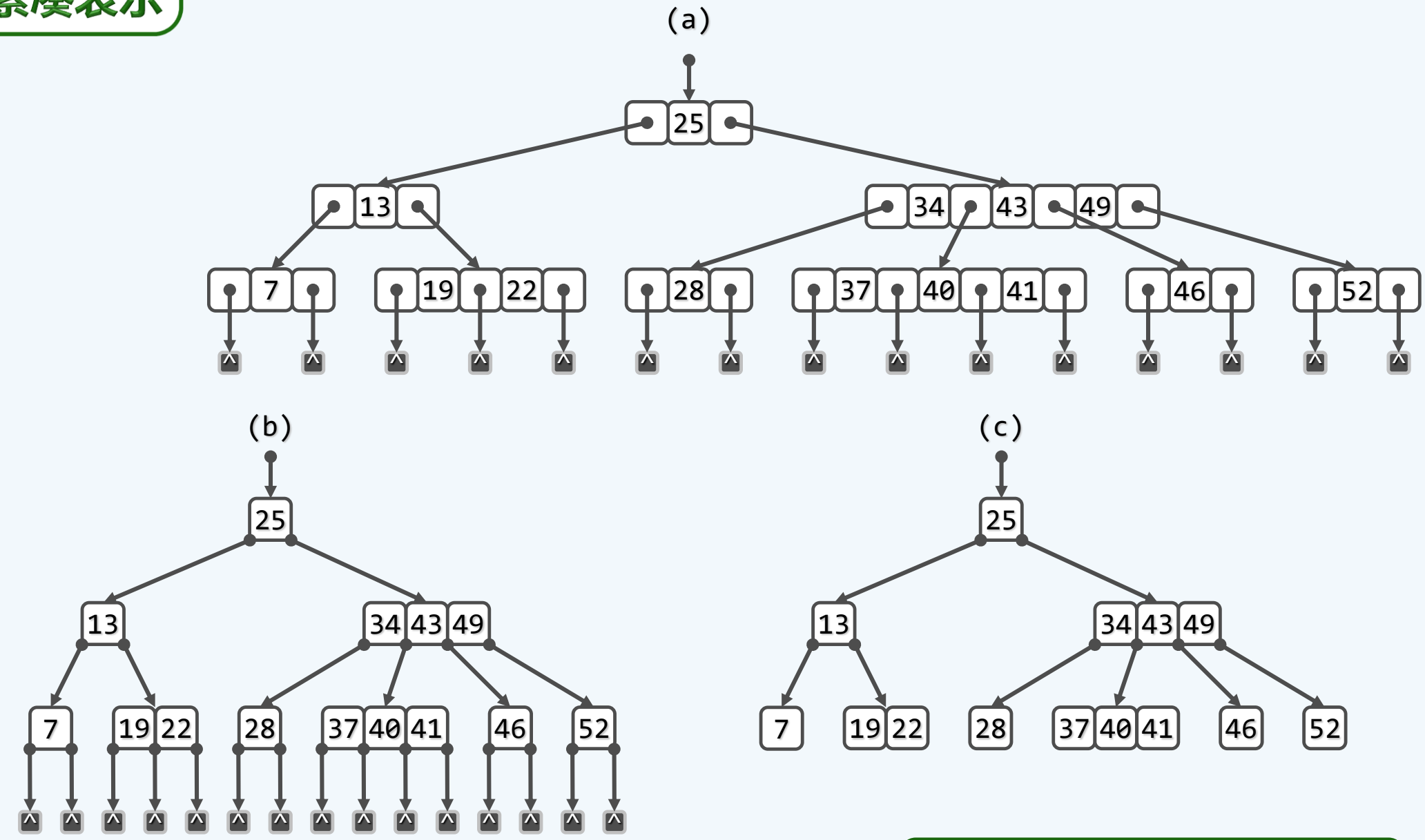
iii) Each node has at most $2k+1$ sons.

Number of Nodes in B-Trees. Let $N_{\min}$ and $N_{\max}$ be the minimal and maximal number of nodes in a B-tree $T \in \tau(k, h)$. Then

$$N_{\min} = 1 + 2\left((k+1)^0 + (k+1)^1 + \cdots + (k+1)^{h-2}\right) = 1 + \frac{2}{k}\left((k+1)^{h-1} - 1\right)$$

Rudolf Bayer R. Bayer, E. McCreight: Organization and Maintenance of Large Ordered Indexes Acta Informatica, Vol. 1, Fasc. 3, 1972 pp. 173-189

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

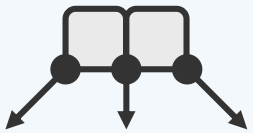
实例

❖ 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

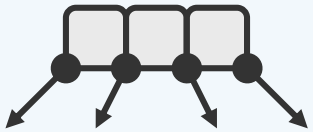


❖ m = 4

2-3-4-树，(2,4)-树

各节点的分支数，可能是2、3或4

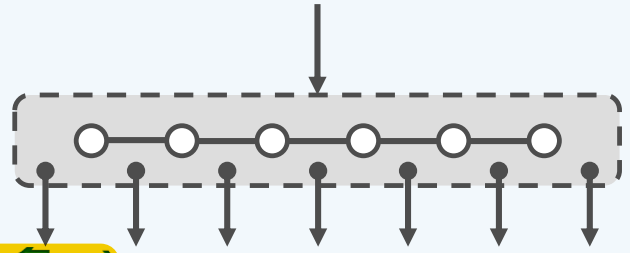
各节点所含key的数目，可能是1、2或3

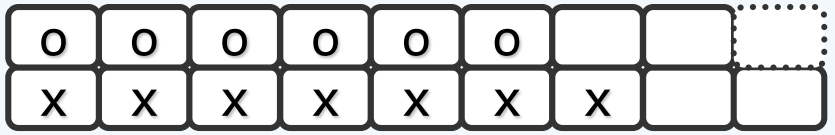


❖ 留意把玩4阶B-树，对稍后理解红黑树大有裨益

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
  };
```

8.高级搜索树

（b3）B-树：查找

高至天低至深海

每寸搜索着这天下

寻觅着那个"它"

邓俊辉

deng@tsinghua.edu.cn

算法

- 将根节点作为当前节点 //常驻RAM

  只要当前节点非外部节点

  在当前节点中顺序查找 //RAM内部

  若找到目标关键码，则

  返回查找成功

  否则 //止于某一对下层引用

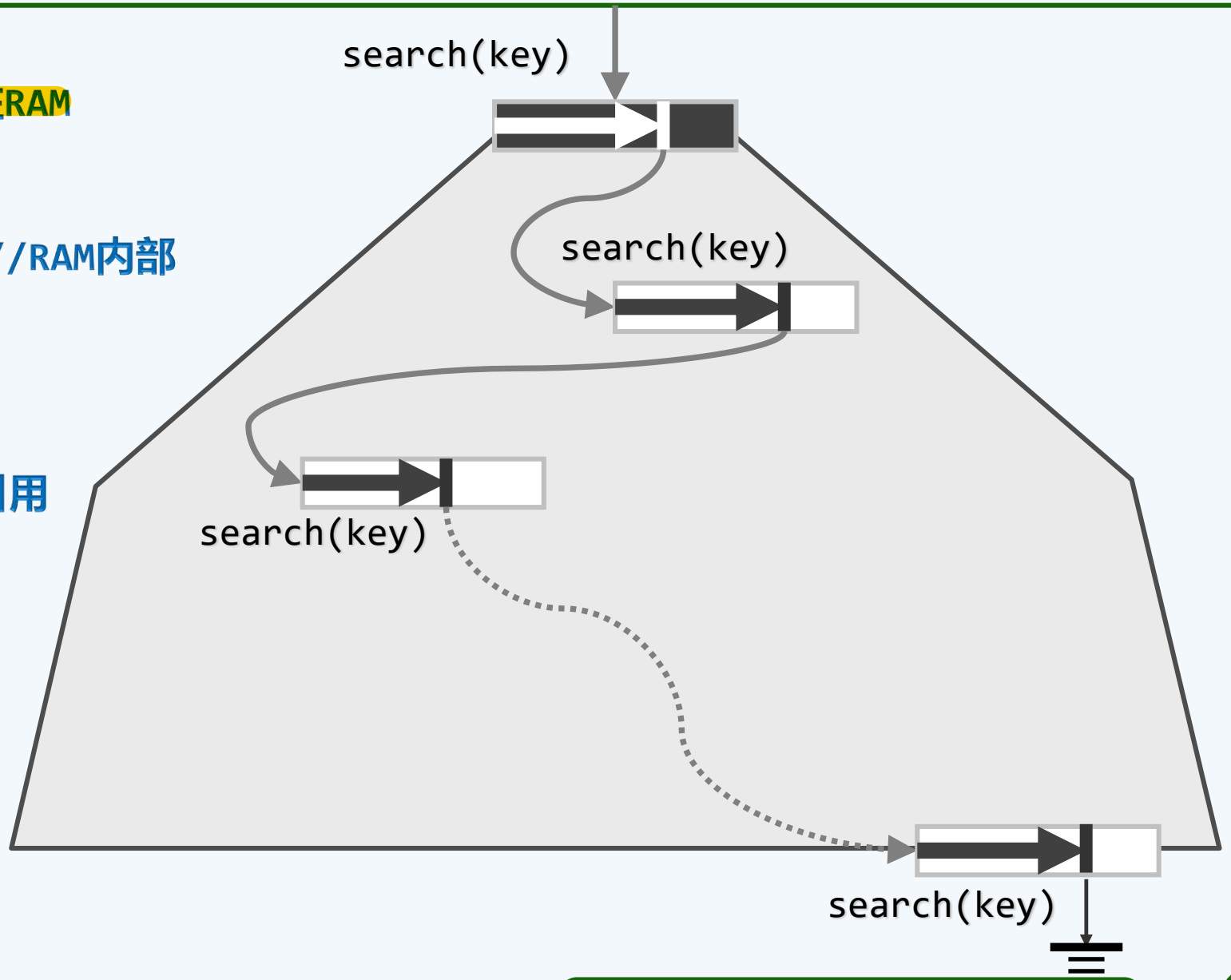
  沿引用，转至对应子树

  将其根节点读入内存

  //I/O，最为耗时

  更新当前节点

返回查找失败



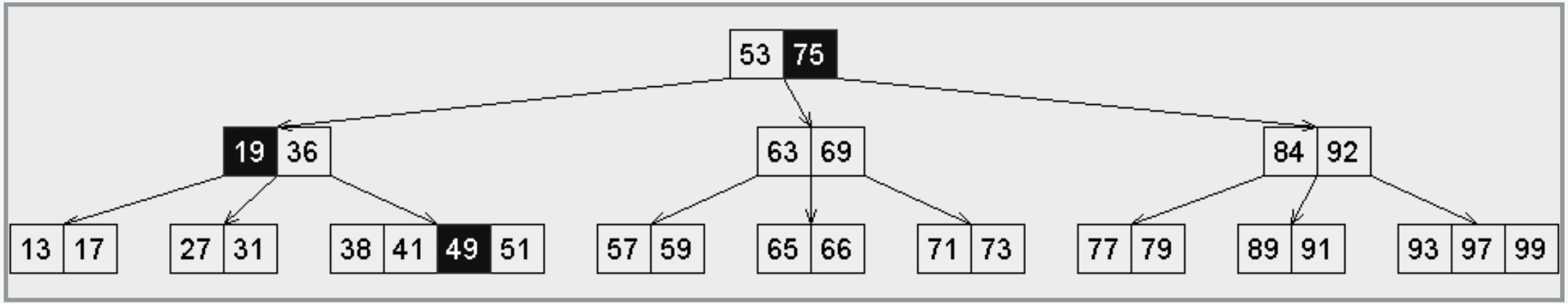
实例

❖ (3,5)-树: 53 97 36 89 41 75 19 84 77 79 51 57 99 91

92 93 17 73 13 66 59 49 63 65 71 69 27 31 38

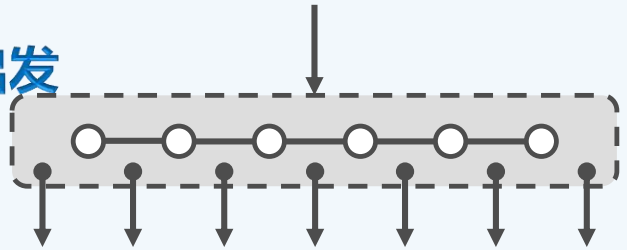
成功查找: 75, 19, 49

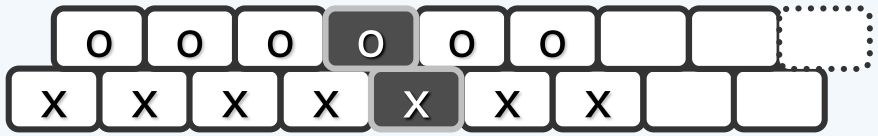
失败查找: 5, 45



实现

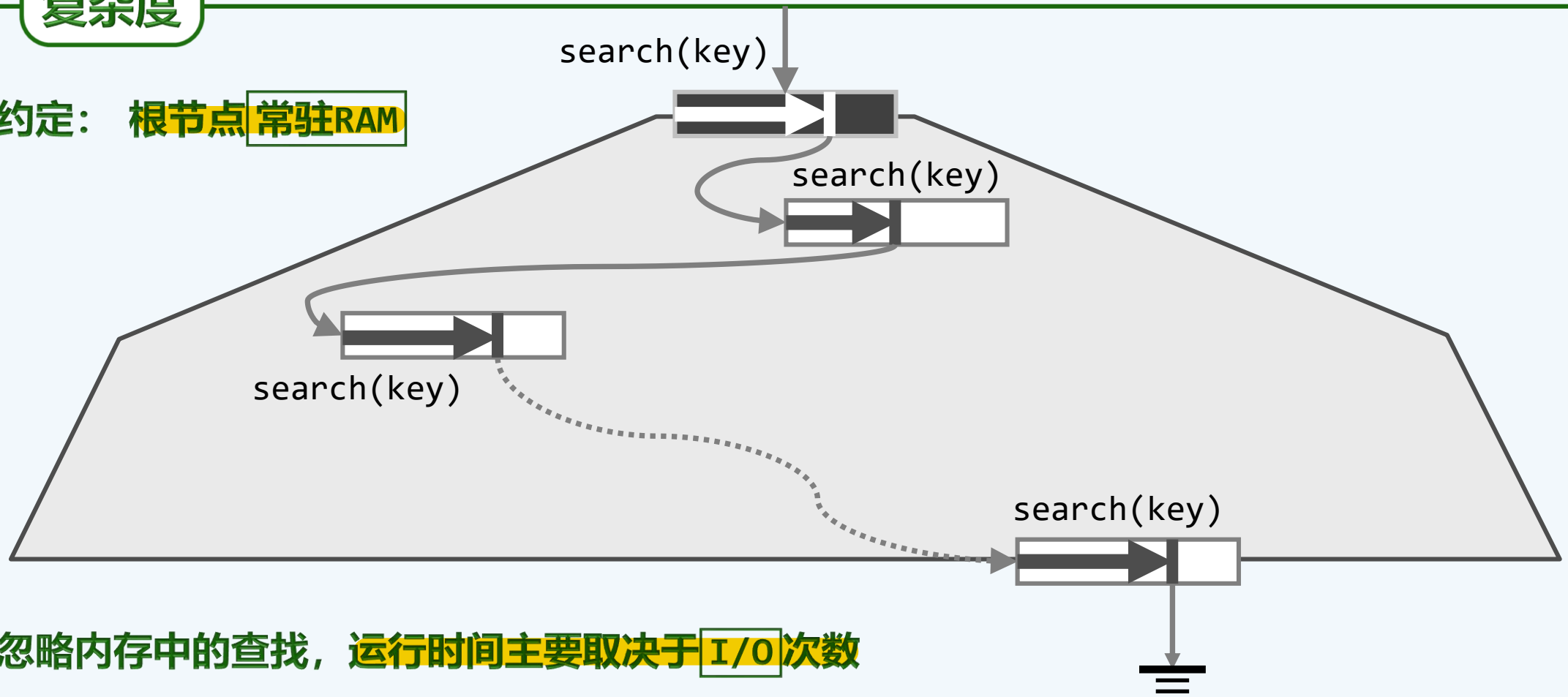
```
❖ template <typename T> BTNodePosi(T) BTree<T>::search( const T & e ) {
    BTNodePosi(T) v = _root; _hot = NULL; //从根节点出发
    while ( v ) { //逐层查找
      Rank r = v->key.search( e ); //在当前节点对应的向量中顺序查找
      if ( 0 <= r && e == v->key[ r ] ) return v; //若成功，则返回；否则...
      _hot = v; v = v->child[ r + 1 ]; //沿引用转至对应的下层子树，并载入其根I/O
    } //若因!v而退出，则意味着抵达外部节点
    return NULL; //失败
  }
```





复杂度

❖ 约定： 根节点常驻RAM



❖ 忽略内存中的查找，运行时间主要取决于I/O次数

❖ 在每一深度至多一次I/O

❖ 故运行时间 = Θ(终止节点的深度) = O(h)

最大树高

❖ 含N个关键码的m阶B-树，最大高度 = ？

❖ 为此，内部节点应尽可能“瘦”，各层节点数依次为

$n_0 = 1,\ n_1 = 2,\ n_2 = 2 \times \lceil m/2 \rceil,\ \ldots$

$n_k = 2 \times \lceil m/2 \rceil^{k-1}$

❖ 考查外部节点所在层

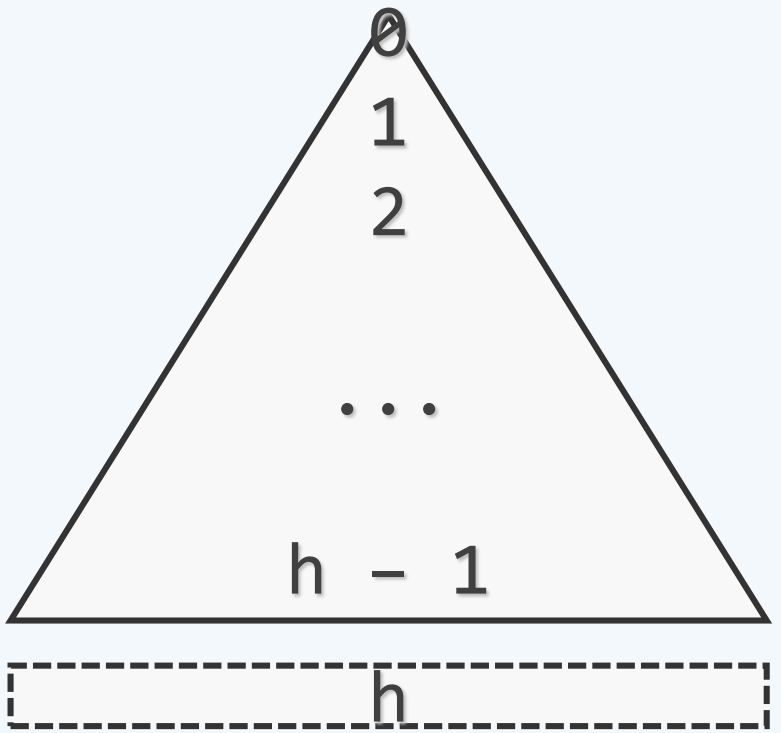
$N + 1 = nh \geq 2 \times (\lceil m/2 \rceil)^{h-1}$

$h \leq 1 + \log_{\lceil m/2 \rceil} \lfloor (N + 1)/2 \rfloor = O(\log_m N)$

❖ 相对于BBST：

$\log_{\lceil m/2 \rceil}(N/2) \ / \ \log_2 N = 1/(\log_2 m - 1)$

若取m = 256，树高（I/O次数）约降低至1/7



最小树高

❖ 含N个关键码的m阶B-树，最小高度 = ？

❖ 为此，内部节点应尽可能“胖”

各层节点数依次为

$n_0 = 1,\ n_1 = m,\ n_2 = m^2$

$n_3 = m^3,\ \ldots,\ n_{h-1} = m^{h-1},\ \boxed{n_h = m^h}$

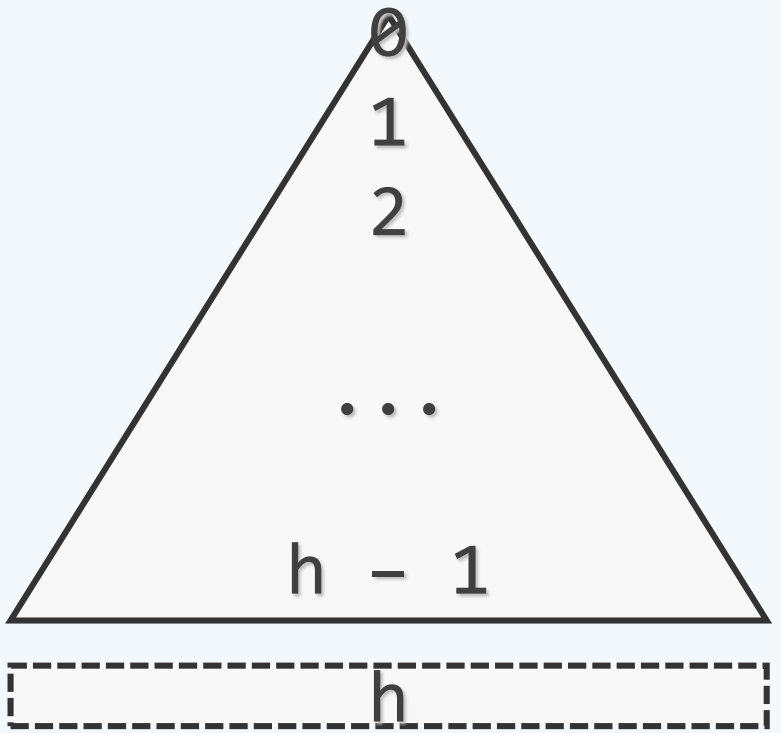
❖ 考查外部节点所在层：

$\boxed{N + 1 = n_h} \le m^h$

$h \ge \log_m(N + 1) = \Omega(\log_m N)$

❖ 相对于BBST：$(\log_m N - 1)/\log_2 N = \log_m 2 - \log_N 2 \approx 1/\log_2 m$

若取m = 256，树高（I/O次数）约降低至1/8



意义与价值

- BBST，究竟可能多高?

- 考查高度为h的BBST（如AVL）可包含节点的数目f

```
h =  33 时, f = 2 ^ 033 = 8.6 * 10 ^ 09          //全球人口
h =  77 时, f = 2 ^ 077 = 1.5 * 10 ^ 23          //全球人口的体细胞总数
h = 133 时, f = 2 ^ 133 = 1.1 * 10 ^ 40          //国际象棋可能的局面总数
h = 260 时, f = 2 ^ 260 = 1.8 * 10 ^ 78          //目前可观测宇宙中基本粒子总数
```

- 由此可见，B-树的价值，的确更多地体现在实用方面

  通过选取适当的节点规模（m），弥合存储层级之间巨大的速度差异

- 另外，在算法方面，B-树也有其独特的价值与地位...

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

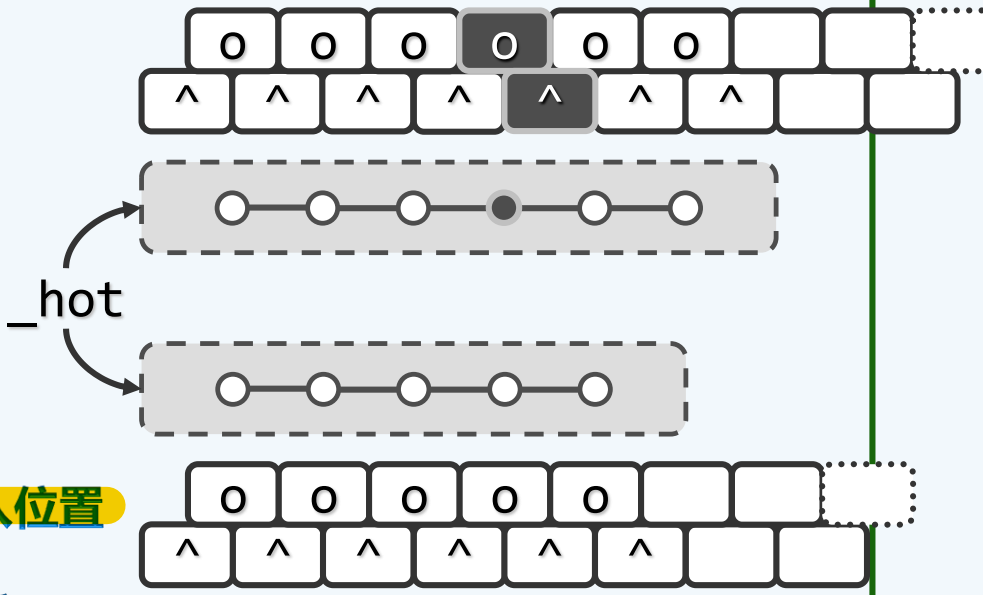
清醒让我，分裂再分裂

邓俊辉

deng@tsinghua.edu.cn

算法

```
template <typename T>
bool BTree<T>::insert( const T & e ) {
    BTNodePosi(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 $k_0, \ldots, k_{m-1}$

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

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

❖ 关键码 k_s 上升一层，并

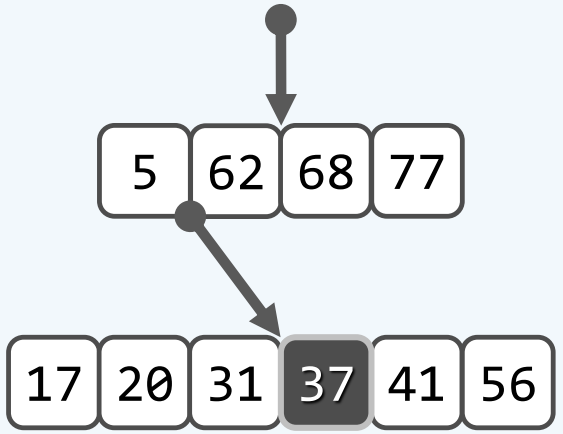
分裂 split：以所得的两个节点作为左、右孩子

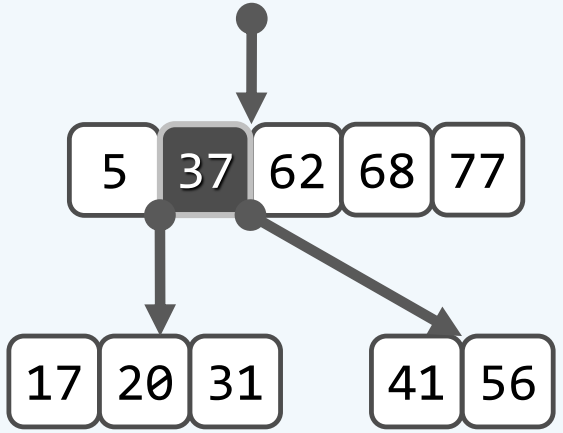
❖ 确认： 如此分裂后

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

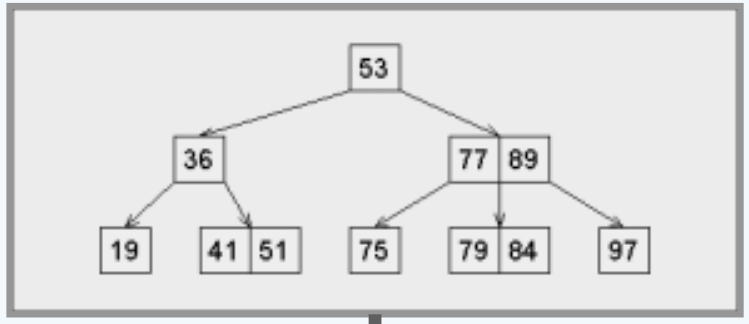
- 上溢可能持续发生，并逐层向上传播；纵然最坏情况，亦不过到根
- 若果真抵达树根，可令被提升的关键码自成节点，作为新的树根

  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⌈m/2⌉
- 总体执行时间线性正比于分裂次数，不超过$O(h)$

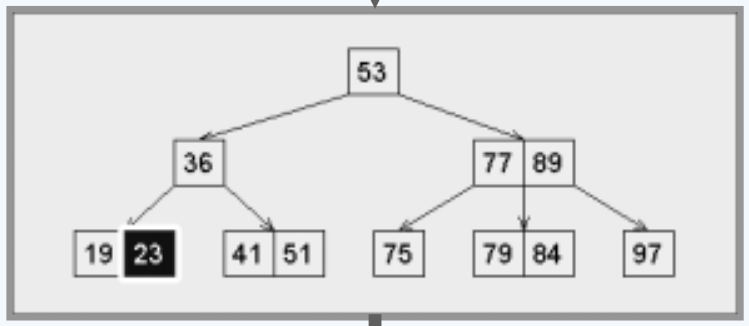
实例

❖ 2-3-树:

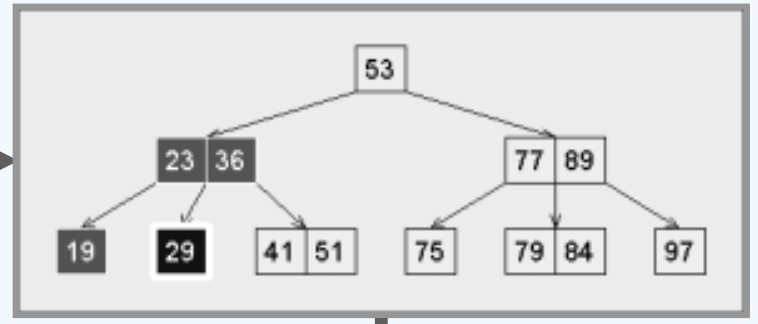
53 97 36 89 41 75 19 84 77 79 51



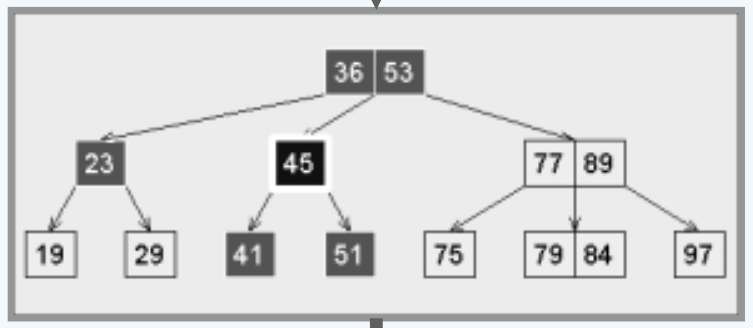
insert(23) //无需分裂



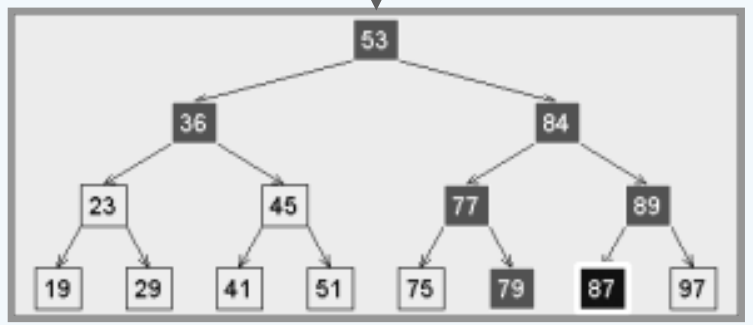
insert(29) //分裂一次



insert(45) //分裂两次



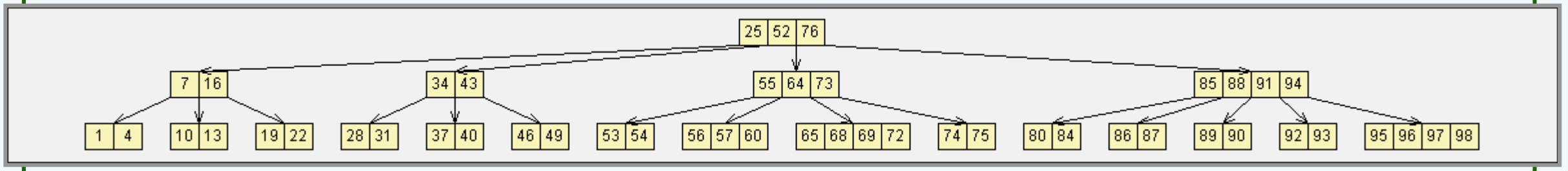
insert(87) //分裂到根



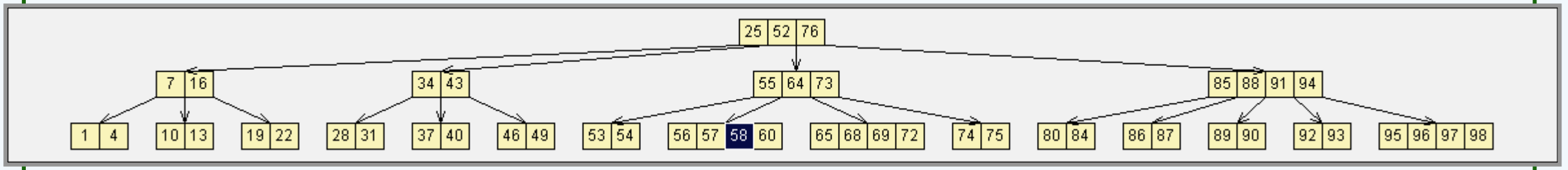
实例

3-5-树

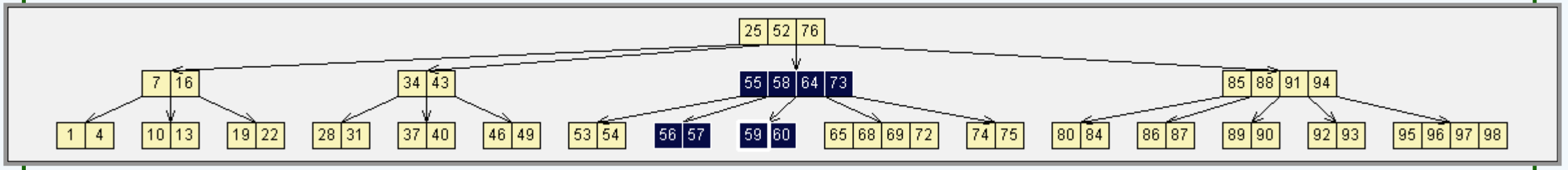
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 56 60 64 68 72 76 80 84 53 54 55 85 86 87 88 89 90 91 92 93 94 95 96 97 98 73 74 75 57 65 69



insert(58) //无需分裂

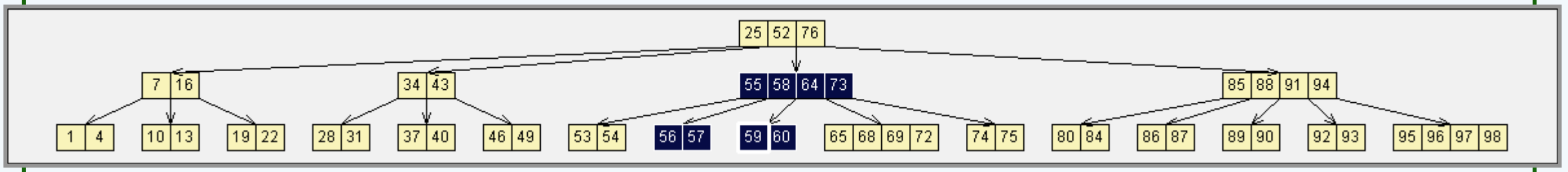


insert(59) //分裂 1 次

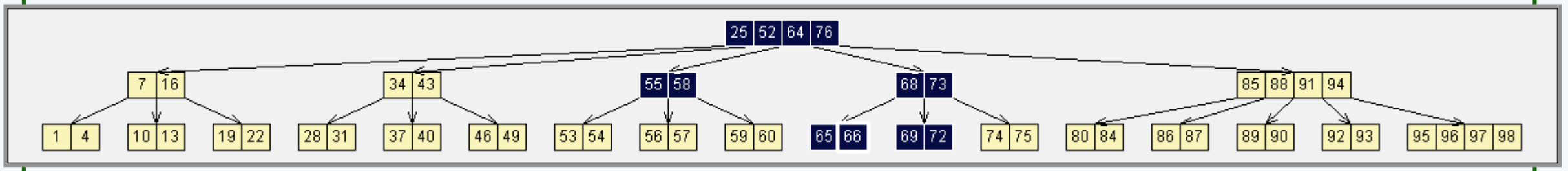


实例

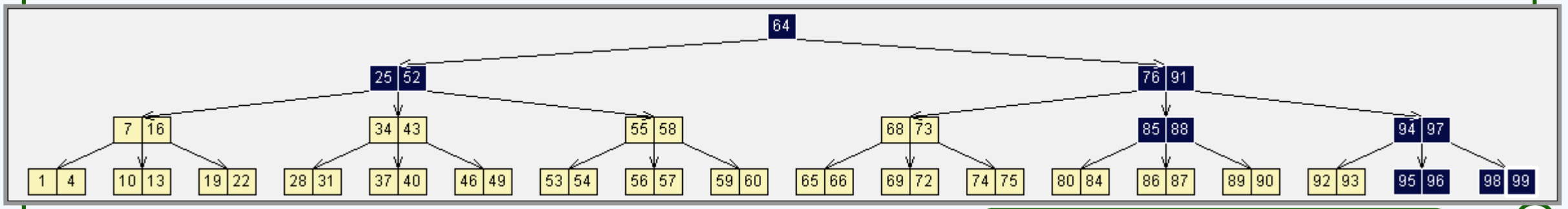
insert(59)　//分裂 1 次



insert(66)　//分裂 2 次



insert(99)　//分裂到根

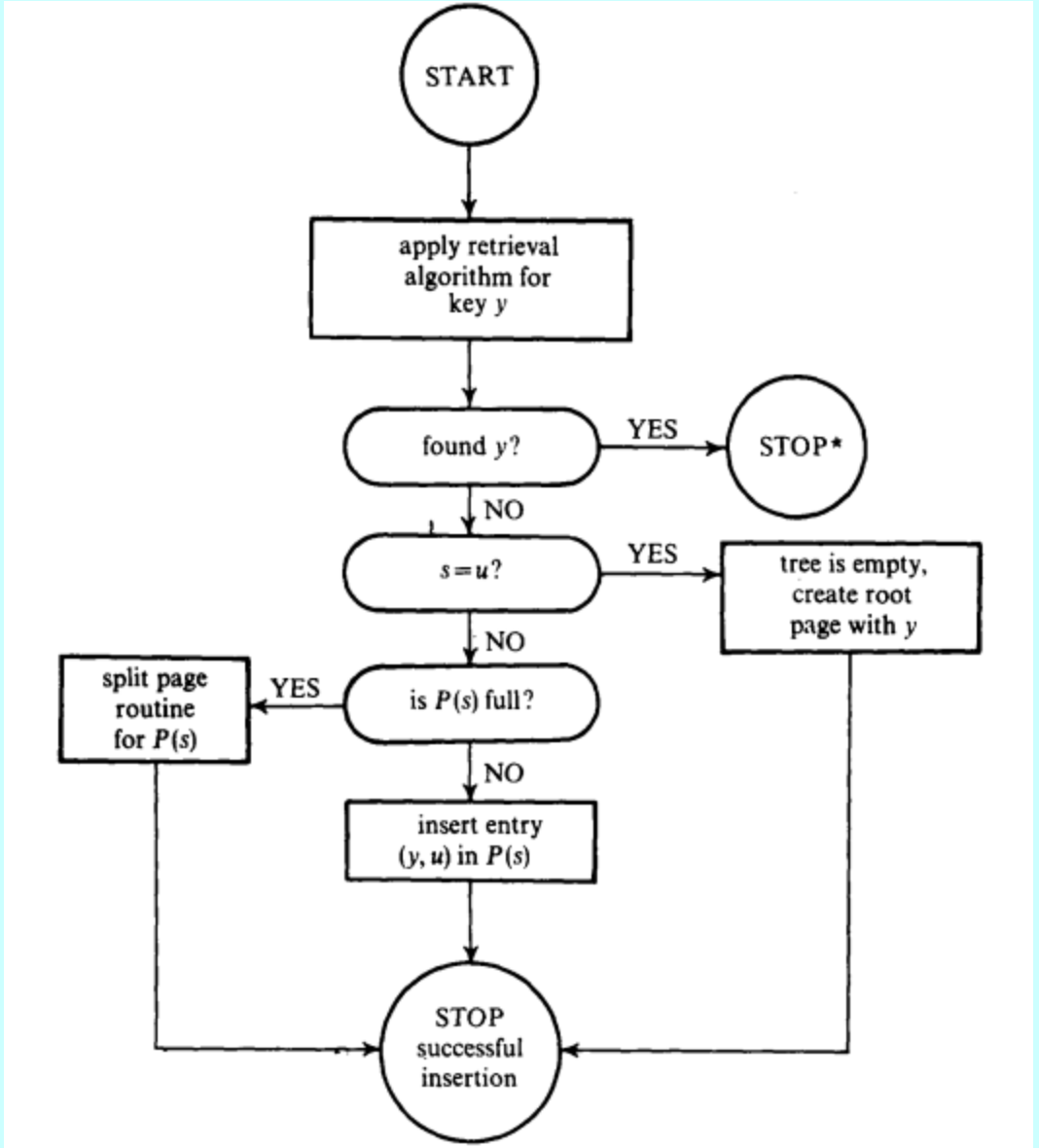


上溢修复

```
❖ template <typename T> void BTree<T>::solveOverflow( BTNodePosi(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）
    BTNodePosi(T) u = new BTNode<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(logn)层
  }
```



* Key y is already in index, take appropriate action.

Fig. 4. Insertion algorithm

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

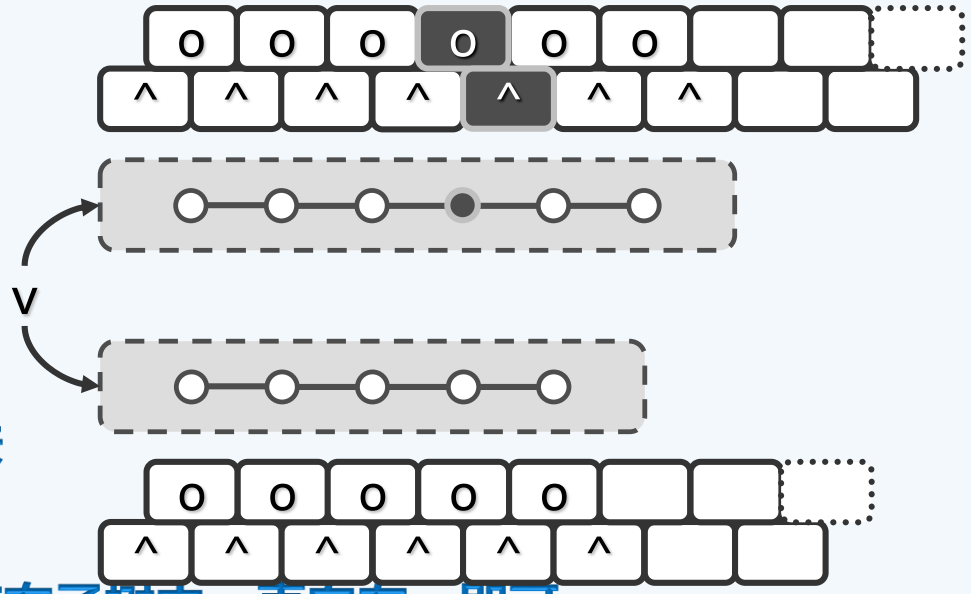
也不愿不生不死不悔的倒影

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

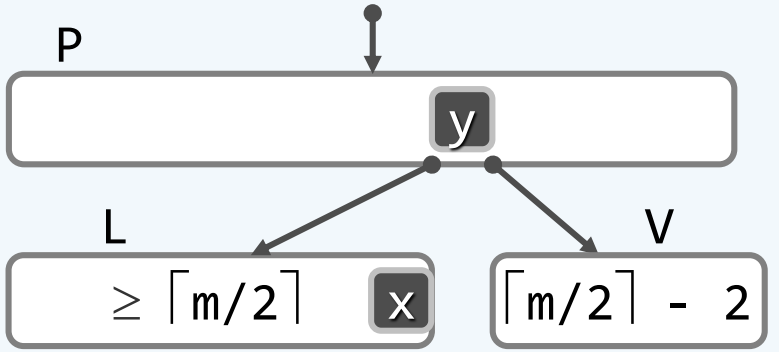
❖ 节点V下溢时，必恰好包含：⌈m/2⌉ – 2个关键码 + ⌈m/2⌉ – 1个分支

❖ 视其左、右兄弟L、R所含关键码的数目，可分三种情况处理

1)若L存在，且至少包含⌈m/2⌉个关键码

将关键码y从P移至V中（作为最小关键码）

将关键码x从L移至P中（取代其中原关键码y）

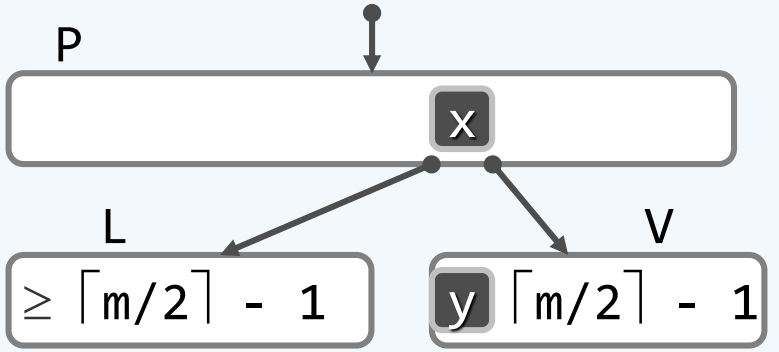


❖ 如此旋转之后，局部乃至整树都重新满足B-树条件

下溢修复完毕

2)若R存在，且至少包含⌈m/2⌉个关键码

完全对称



合并

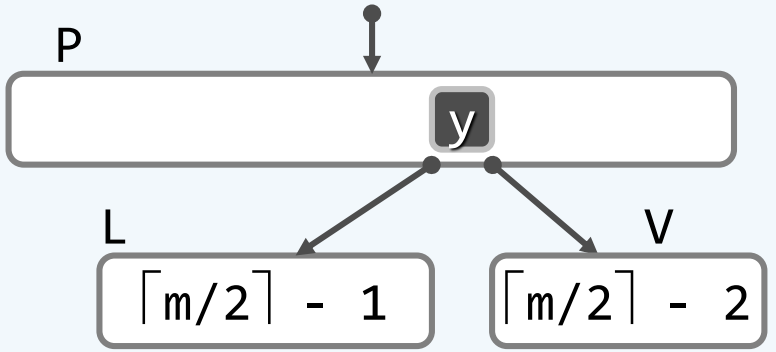
3) L和R或者不存在，或者所含的关键码均不足⌈m/2⌉个

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

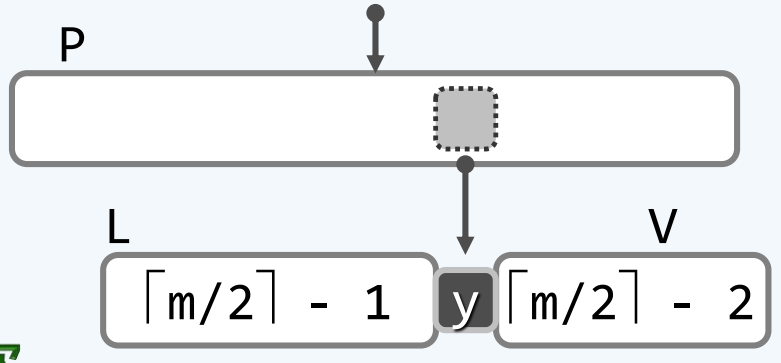
同时合并此前y的孩子引用



❖ 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

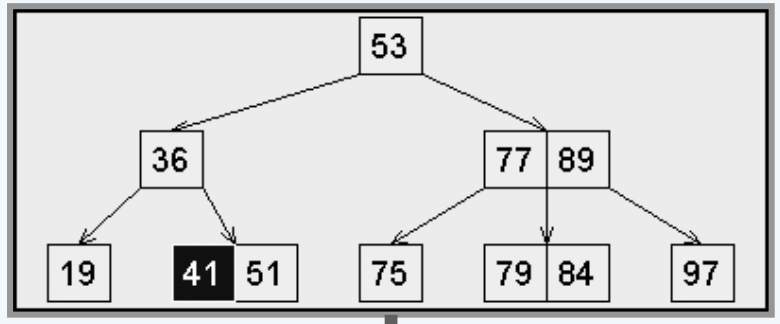


❖ 下溢可能持续发生，并逐层向上传播；但至多不过O(h)次

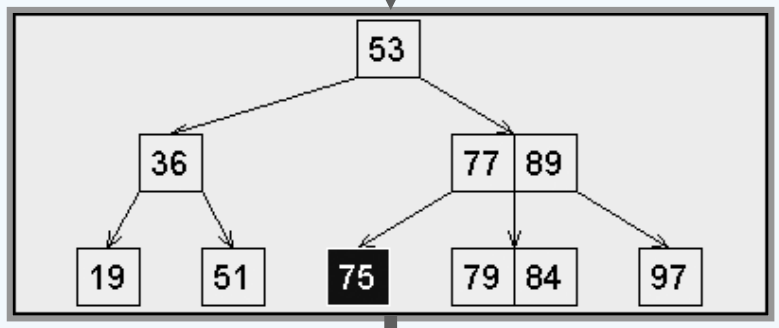
实例：底层节点

❖ 2-3-树

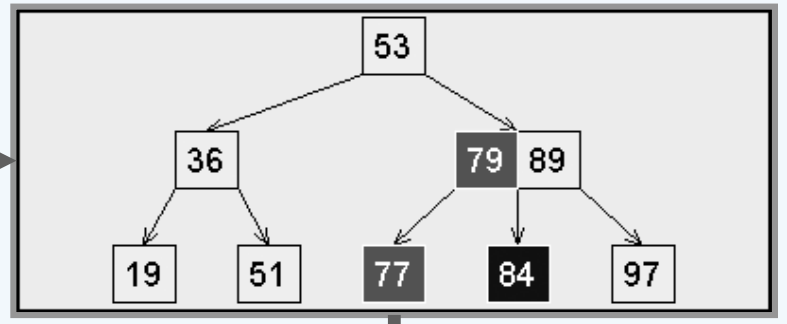
53 97 36 89 41 75 19 84 77 79 51



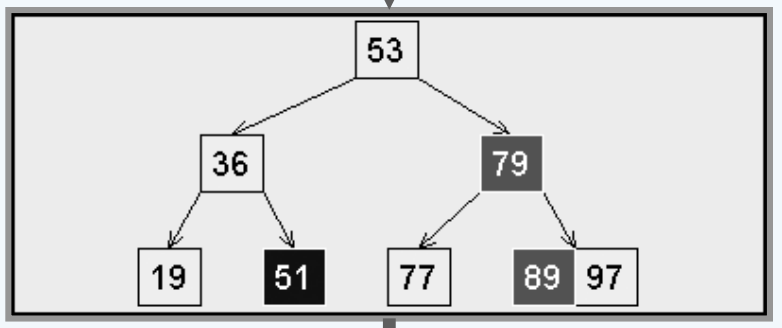
remove(41) //直接删除



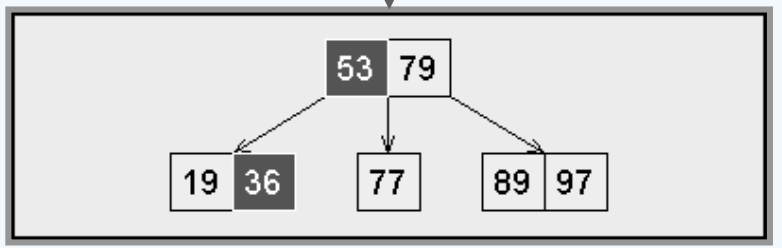
remove(75) //旋转



remove(84) //单次合并



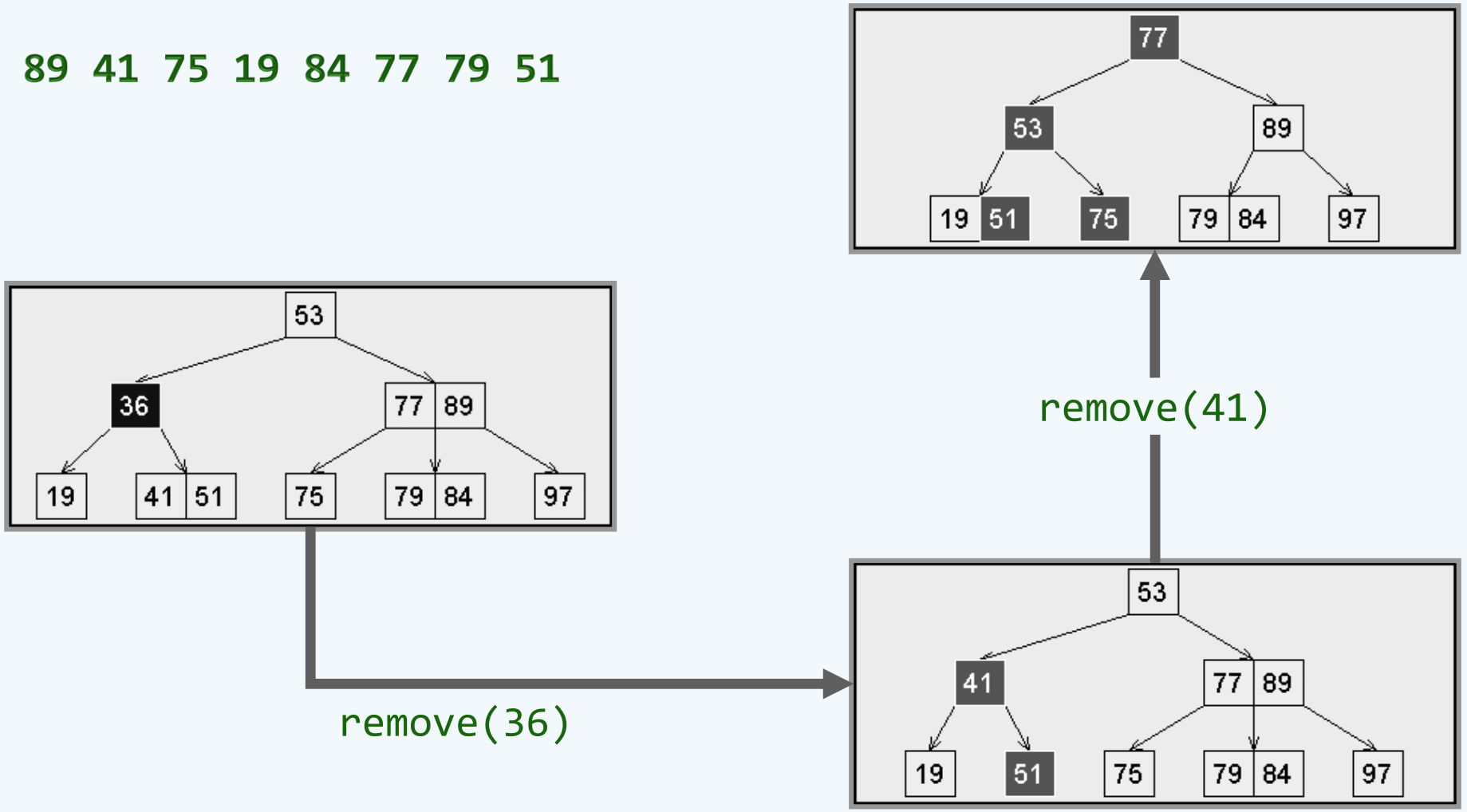
remove(51) //多次合并



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

- 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 ⌊(m - 1) * k / (k + 1)⌋ 个关键码

如此，可将空间使用率从 50% 提高至 k / (k + 1)

Reference

- *Rudolf Bayer R. Bayer, E. McCreight: Organization and Maintenance of Large Ordered Indexes Acta Informatica, Vol. 1, Fasc. 3, 1972 pp. 173-189.*
- *Data Structures and Algorithm Analysis in C++* Section 4.7

Next

- Red-hat 树
- 数据结构（C++语言版）第三版 Chapter 8.3