大数据算法

第五讲 外存查找结构

哈尔滨工业大学
王宏志
wangzh@hit.edu.cn

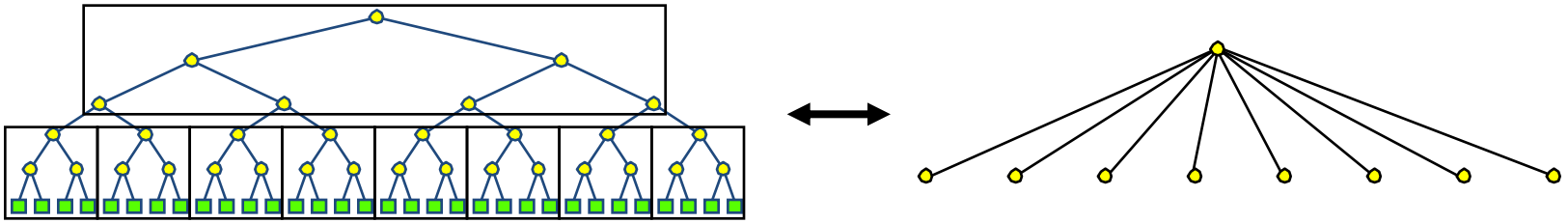
本讲内容

5.1 B树

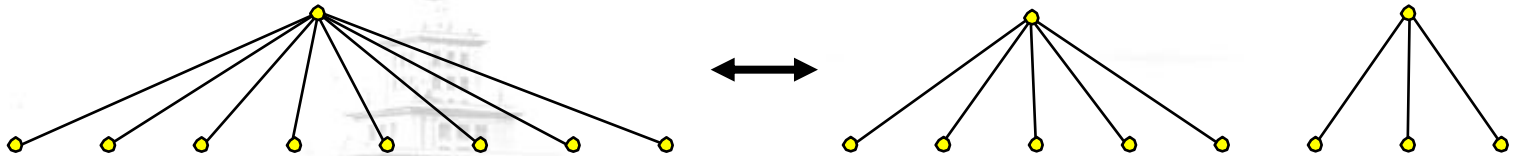
5.2 KD树

B-树

- BFS-块自然对应于扇出为$\Theta(B)$的树



- B-树通过允许节点度变化来保持平衡
  - 再平衡通过分裂和合并结点实现

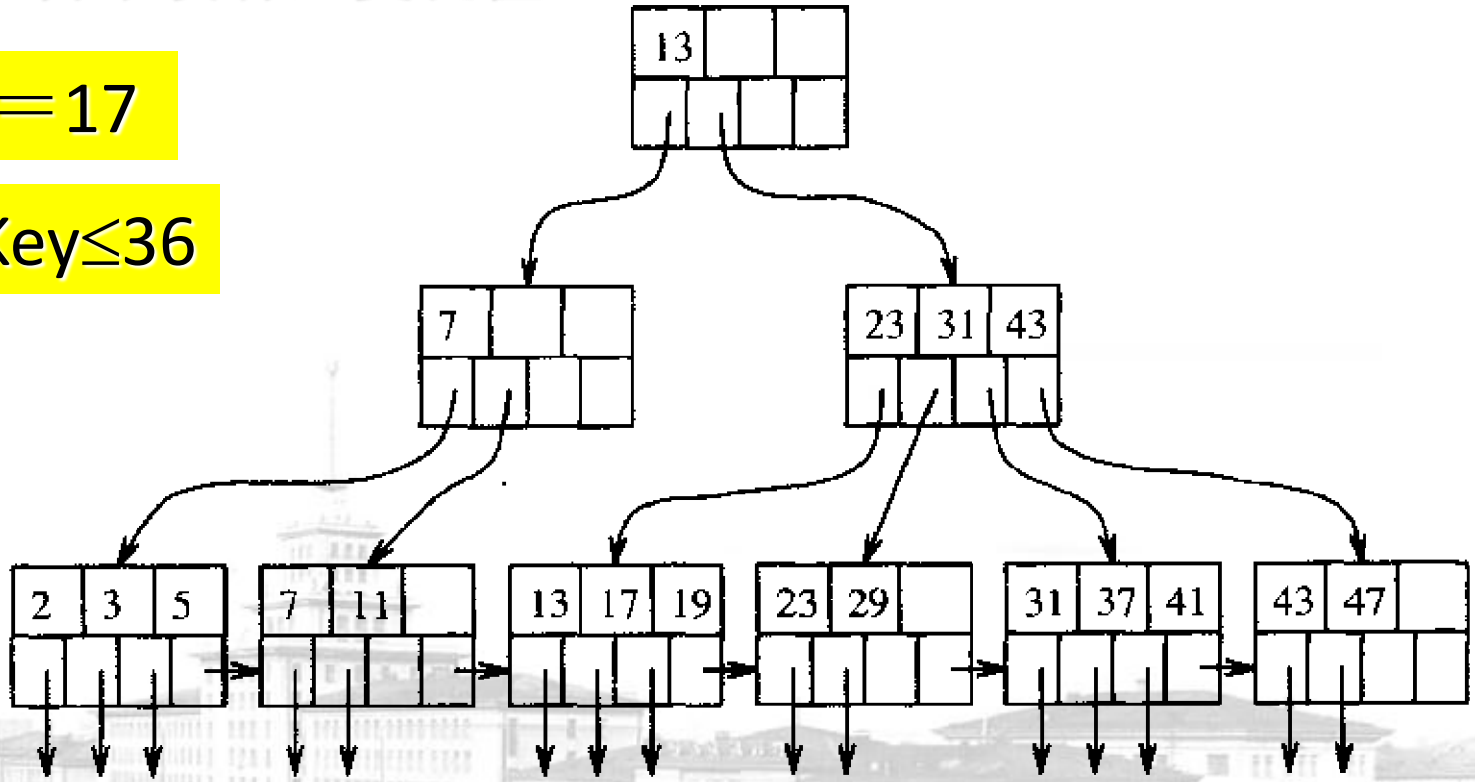


B-树上的查询

- 假定
  - B-树中没有重复键值

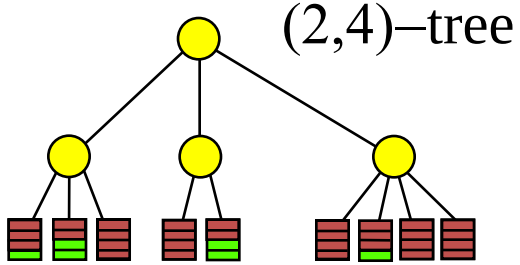
Key＝17

17≤ Key≤36



(a,b)-树

- T 是一个(a,b)-树 ($a \geq 2$ 且$b \geq 2a-1$)
  - 所有的叶子在同一层上并且包括a到 b个元素
  - 除了根结点，所有的结点的度在a到b之间
  - 根结点的度在2到b之间



- (a,b)-树使用线性空间，并且高度为 $O(\log_a N)$

  ⇓

  选择 $a,b = \Theta(B)$ 每个结点/叶子保存在一个磁盘块中

  ⇓

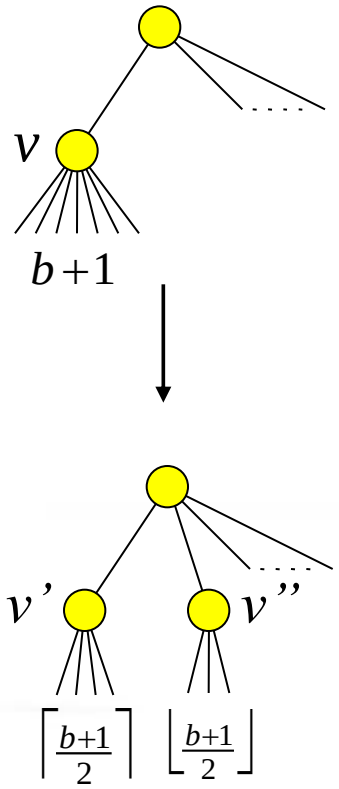
  $O(N/B)$ 空间和 $O(\log_B N + T/B)$ 查询

(*a*,*b*)-树的插入

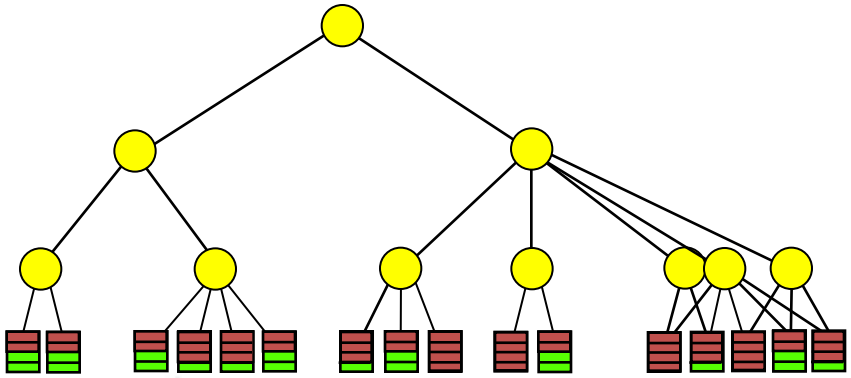
- 插入:

在叶子*v*上搜索和插入元素
DO v有b+1个元素/儿子
拆分*v*:
创建结点v’和v”，分别有 $\lceil \frac{b+1}{2} \rceil \le b$ 和 $\lfloor \frac{b+1}{2} \rfloor \ge a$ 个元素
将元素或指针插入parent(*v*)
(如有必要则创建新结点)
v=parent(*v*)

- 插入涉及 $O(\log_a N)$ 个结点



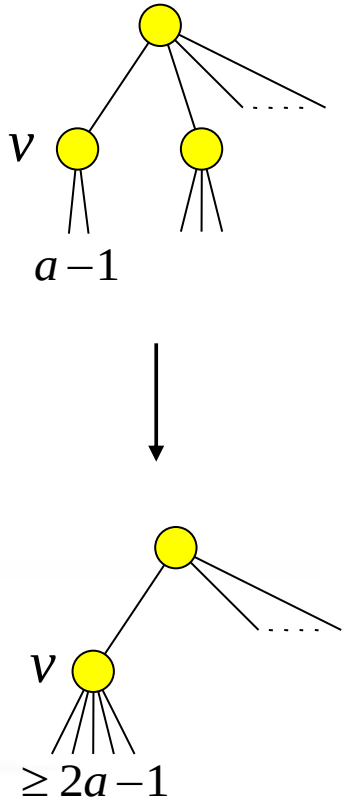
(2,4)-树的插入



(*a*,*b*)-树的删除

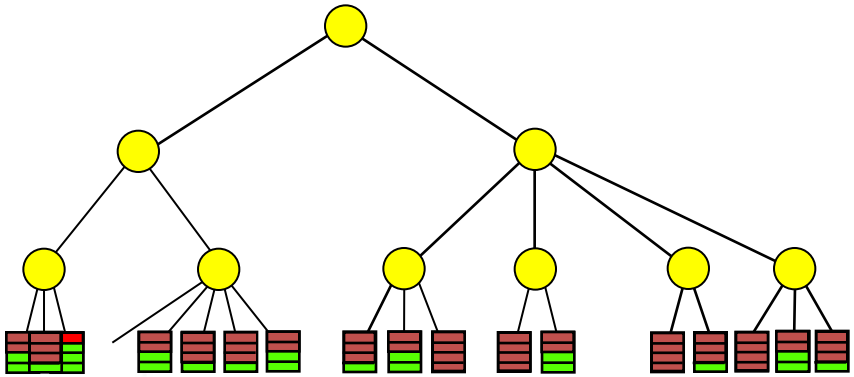
- 删除:

从叶子*v*搜索和删除元素
DO v有a-1个元素/儿子
　将*v*和兄弟*v*'合并:
　　将*v*'的儿子移到*v*
　　从parent(*v*)中删除元素/引用
　　(如果需要则删除根)
　If *v*有>*b*(且≤ *a*+*b*-1<2*b*)个儿子
　　分裂*v*
v=parent(*v*)



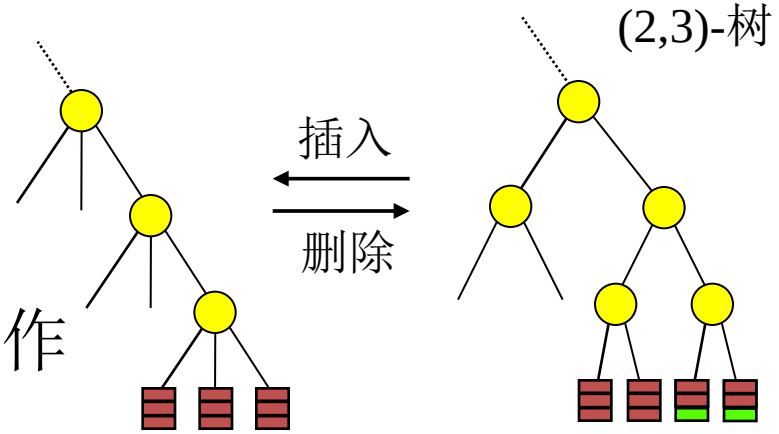
- 删除涉及 $O(\log_a N)$ 个结点

(*2,4*)-Tree Delete



(*a*,*b*)-树

- (a,b)-树属性:
  - 如果 b=2a-1，
    每个更新操作
    能够引起很多次再平衡操作



  - 如果 b≥2a，更新仅会引起O(1) 次再平衡操作
  - 如果 b>2a，只需要 $O(\frac{1}{b/2-a}) = O(1/a)$ 次再平衡操作,平摊分析起来都有一些困难
  - 如果 b=4a，很容易去证明更新会引起$O(\frac{1}{a}\log_a N)$ 次再平衡操作来平摊
    - 分裂之后，插入一个叶子的时候包含≅ 4a/2=2a 个元素
    - 融合之后，删除一个叶子的时候包含 ≅ 2a 个到 ≅ 5a 个之间的元素 (如果超过3a继续分裂 ⇒ 在3/2a 到 5/2a之间)

摘要/结论: B-树

- B-树: (a,b)-树 其中$a,b = \Theta(B)$
  - $O(N/B)$ 空间
  - $O(\log_B N+T/B)$ 查询
  - $O(\log_B N)$ 更新

- 元素都在叶子中的B-树有时被称为B^+-树

- 建立需要$O(\frac{N}{B}\log_{M/B}\frac{N}{B})$次I/O
  - 排列元素和创建叶子
  - 从底向上一层层的建树

摘要/结论: B-树

- B-树具有分支参数 b 和叶子参数 k (b,k≥8)
  - 所有的叶子在同一层上，并且包括1/4k到k个元素
  - 除了根结点，所有的结点的度为1/4b 到 b
  - 根结点的度为2到b

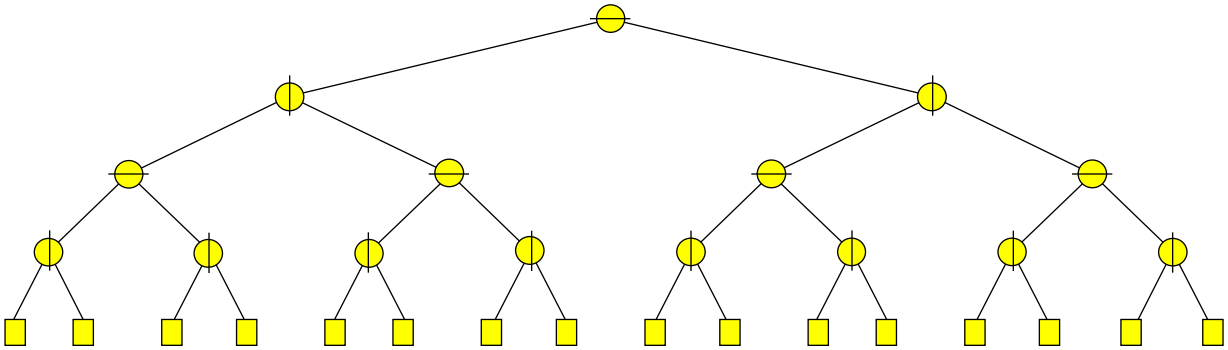
- B-树具有叶子参数 $k = \Omega(B)$
  - O(N/B) 空间
  - 高度 $O(\log_b \frac{N}{B})$
  - 平摊 $O(1/k)$次叶子再平衡的操作
  - 平摊$O(\frac{1}{b \cdot k} \log_b \frac{N}{B})$ 次内部结点再平衡操作
- B-树具有分支参数 B^c, $0<c\leq 1$, 和叶子参数 B
  - 空间 O(N/B)，更新代价$O(\log_B N)$，查询代价 $O(\log_B N + T/B)$

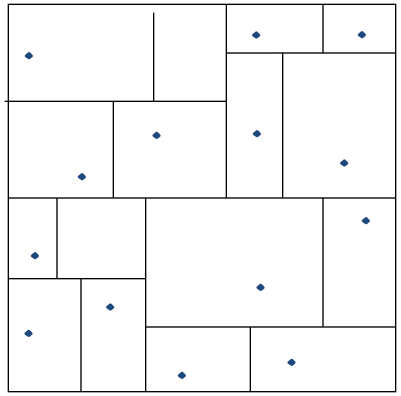
本讲内容

5.1 B树

5.2 KD树

kd-树



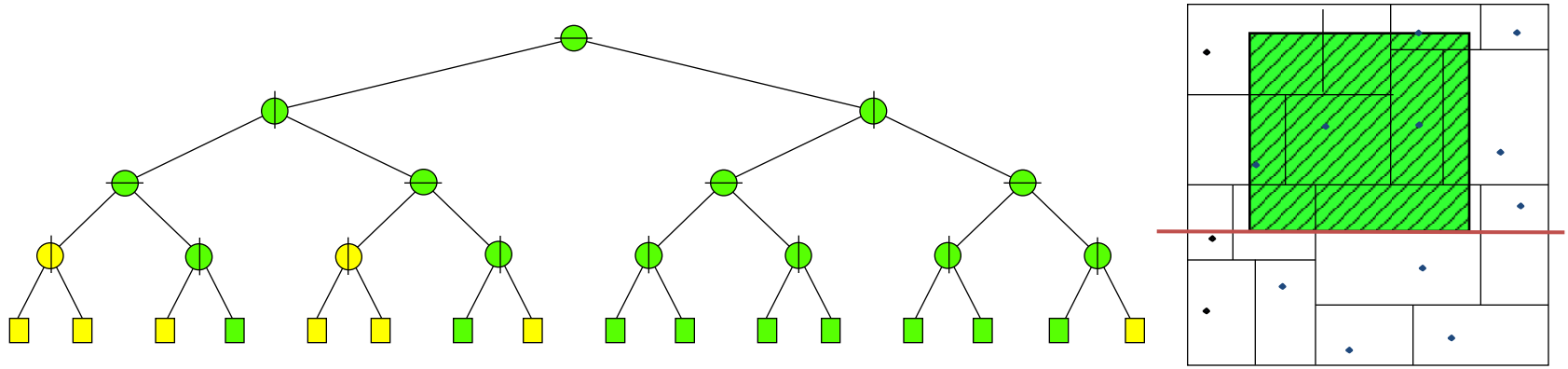


- kd-树:
  - 使用垂直/水平线把点集递归分割成两部分
  - 在偶数层使用水平线，奇数层使用垂直线
  - 每个叶子为一个点

⇓

线性结构和对数高度

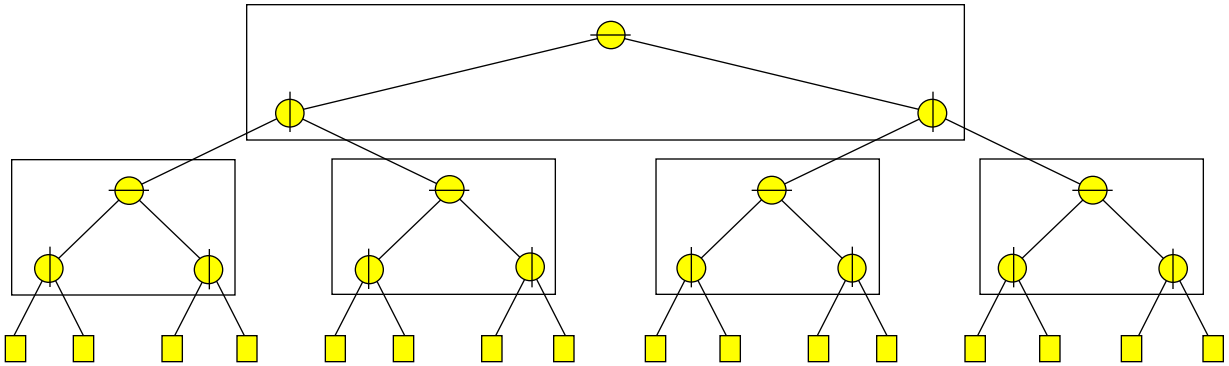
kd-树: 查询

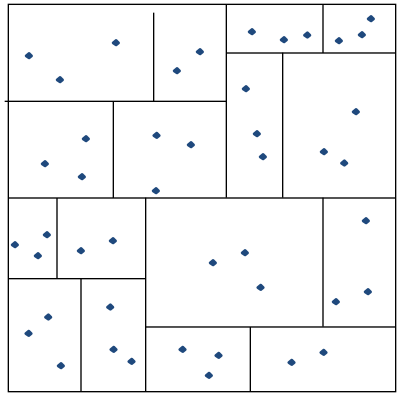


- 查询
  - 递归访问结点响应的交叉查询的区域
  - 报告在树/结点中且在查询中完全包含的点
- 查询分析
  - 水平线相交$Q(N) = 2+2Q(N/4) = O(\sqrt{N})$ 个区域
  - 查询覆盖T 个区域

  ⇒最坏的情况 I/O为$O(\sqrt{N}+T)$

kdB-树



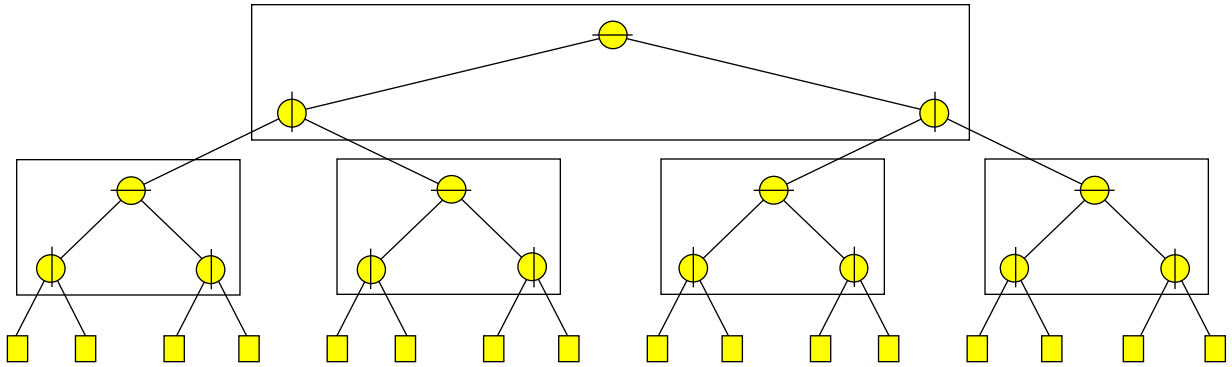


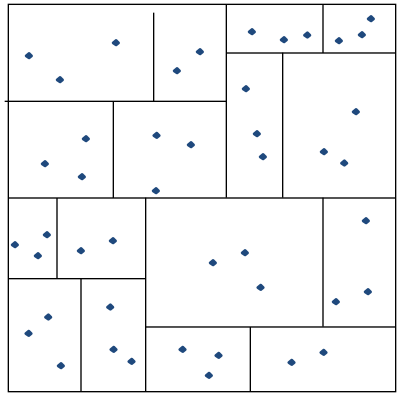
- kdB-树:
  - 当叶子包含$B/2$ 到 B 个点时停止分割
  - 内部结点的BFS-块
- 像以前一样查询
  - 像以前一样分析但是每个区域现在包含$\Theta(B)$ 个点

⇓

$O(\sqrt{N/B} + T/B)$ 次 I/O的查询

kdB-树的构建

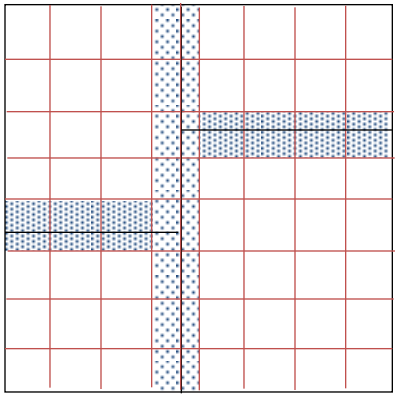




- 简单的 $O(\frac{N}{B}\log_2\frac{N}{B})$ 算法
  - 找到 *y*-坐标的中间 (构造根)
  - 基于中值来分布点
  - 递归构建子树
  - 从顶到下构建BFS-块
- 改进 $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ 算法的思想
  - 使用*O*(*N*/*B*) I/Os 一次构建 $\log\sqrt{M/B}$ 层

kdB-树的构建

- 使用 $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ 次 I/O(根据*x*- 和 *y*-坐标 对N个点排序)
- 建立 $\log\sqrt{M/B}$层 ($\sqrt{M/B}$ 个结点):*O*(*N*/*B*) 次I/O



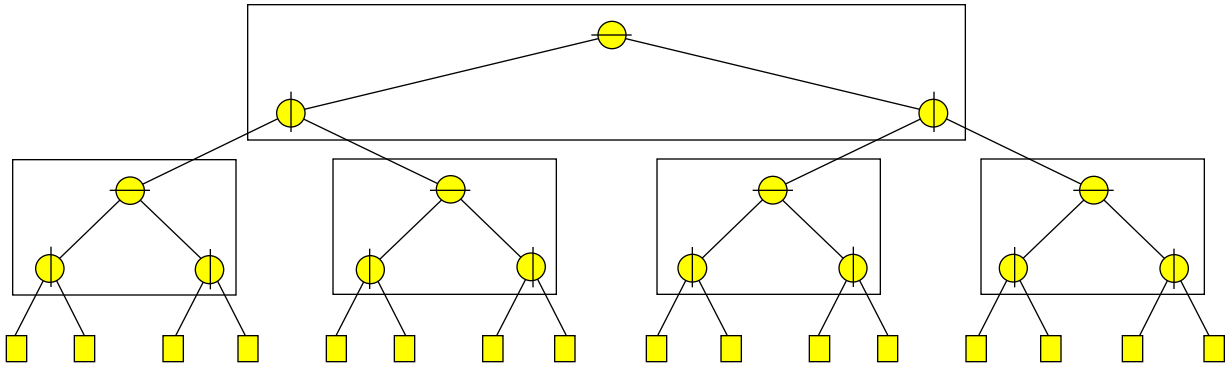
1.使用每个板中的 $N/\sqrt{M/B}$ 个点构建$\sqrt{M/B}$ X$\sqrt{M/B}$ 的网格

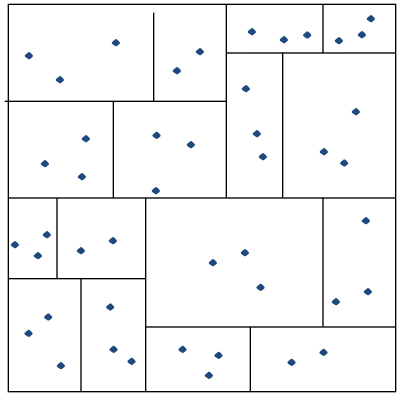
2. 计算每个网格中的点的个数，并且存储在内存中
3. 利用*x*-坐标中点找到板*s*
4. 扫描板*s* 来找到*x*-坐标中点，并且构建结点
5. 分裂包含*x*-坐标中点的板并且更新计数
6. 使用网格在*x*-坐标中点的每侧进行递归(步骤3)

$\Rightarrow$ 在算法过程中，网格增长到 $M/B+\sqrt{M/B}\cdot\sqrt{M/B}=\Theta(M/B)$

$\Rightarrow$ 每个结点的构建代价：$O(N/(\sqrt{M/B}\cdot B))$次I/O

kdB-树

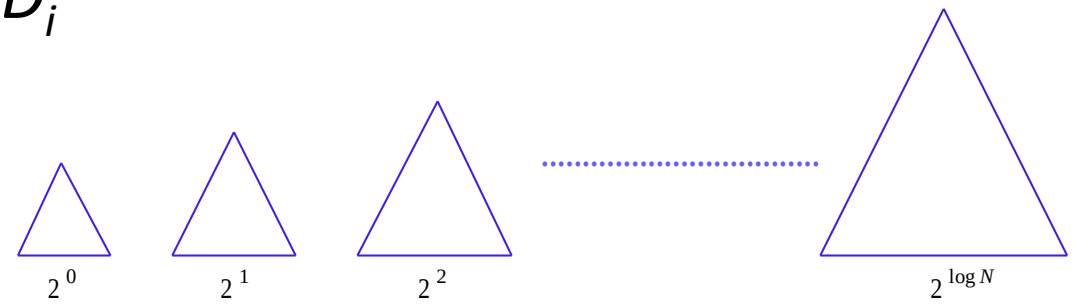




- kdB-树:
  - 线性空间
  - 查询: $O(\sqrt{N/B} + T/B)$ 次I/O
  - 构造: $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ 次I/O
  - 点搜索 $O(\log_B N)$ 次I/O
- 动态?
  - 删除相对容易: $O(\log_B^2 N)$ 次I/O (部分重构)

kdB-树使用对数方法插入

- 把点集S 分成子集$S_0, S_1, \ldots S_{\log N}$, $|S_i| = 2^i$ 或者 $|S_i| = 0$
- 在S_i上建立kdB-树 D_i



- 查询: 每个查询$D_i \Rightarrow \sum_{i=0}^{\log N} O(\sqrt{2^i/B} + T_i/B) = O(\sqrt{N/B} + T/B)$
- 插入: 找到第一个空的D_i 并且构造 D_i 使用 $1+\sum_{j=0}^{i-1} 2^j = 2^i$ 个$S_0, S_1, \ldots S_{i-1}$中的元素
  - $O(\frac{2^i}{B}\log_{M/B}\frac{2^i}{B})$ 次I/O $\Rightarrow$ 每个被移动的点 $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$
  - 点被移动$O(\log N)$ 次

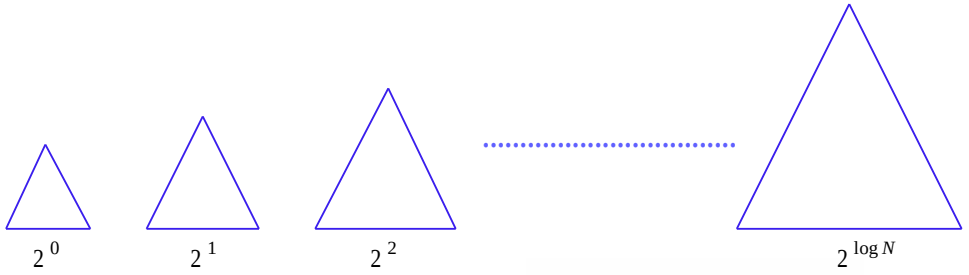
  $\Rightarrow$ 平摊I/O代价: $O(\frac{1}{B}\log_{M/B}\frac{N}{B}\log N) = O(\log_B^2 N)$

kdB-树的插入和删除

- 插入: 使用对数的方法忽略删除
- 删除: 简单的从相关D_i中删除点p
  - 因为p已经被插入，i 能够在#插入的基础上被计算
  - # 插入通过存储在分开的B-树中每个点插入的数字来计算

⇓

$O(\log_B N)$额外的更新代价

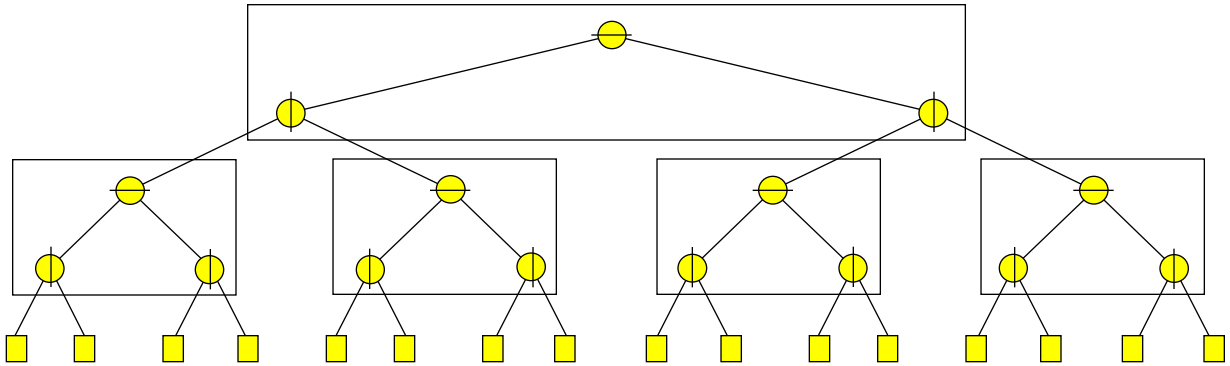


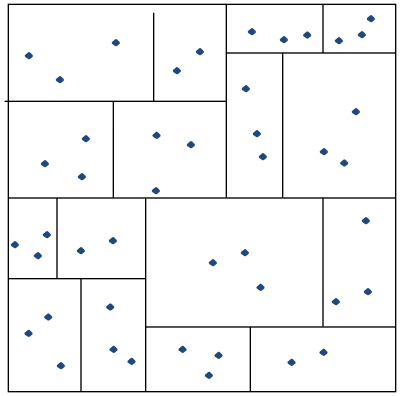
- 为了获得$O(\log N)$ 结构 D_i
  - 在每个$\Theta(N)$ 更新之后执行全局重构

⇓

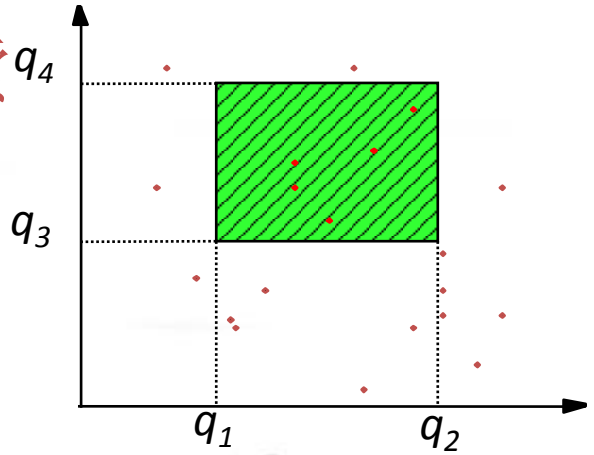
额外的更新代价：$O(\frac{1}{B}\log_{M/B}\frac{N}{B}) = O(\log_B N)$

总结:kdB-树





- 在$O(N/B)$空间内二维范围搜索
  - 查询:$O(\sqrt{N/B}+T/B)$ 次I/O
  - 构造:$O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ 次I/O
  - 更新: $O(\log_B^2 N)$ 次I/O



- 最优查询为线性空间结构

致谢

- 本讲义部分内容来自于Lars Arge的讲义