索引

定义

- 主码 : 数据库中每条记录的唯一标识
- 辅码 : 数据库中可出现重复值的码（属性）
  - 辅码索引把一个辅码值与具有这个辅码值的多条记录的主码值关联起来
  - 多数检索都是利用辅码索引完成的
- 索引 : (key, pointer)
- 索引文件：用于记录这种联系的文件
  - 稠密索引 : 对每个记录建立一个索引项
    - 主文件不需按关键码次序排列
  - 稀疏索引 : 对一组记录建立一个索引
    - 主文件必须按照关键码次序存放

线性索引

按照索引码值的顺序进行排序的文件

- 优缺点
  - 可对变长的数据库记录访问
  - 支持对数据的高效检索
    - 二分检索
  - 线性索引太大，存储在磁盘中
    - 一次检索可能多次访问磁盘，影响检索的效率
    - 使用二级线性索引

静态索引

- 索引结构在文件创建时生成
- 一旦生成就固定下来，在系统运行(例如插入和删除记录)
  过程中，索引结构不做改变
- 只有当文件再组织时才允许改变索引结构

> 多分树 : 大大减少访问外存的次数

倒排索引

- 按属性值建立索引，索引表中的每一项包括 (attr, ptrList):
- 一个属性值
- 具有该属性值的各关键码或者记录地址
- 优缺点
  - 支持基于属性的高效检索
  - 花费储存倒排表的代价, 降低更新的效率

动态索引 (B树)

动态索引 : 文件创建时生成, 结构随着插入, 删除等操作而改变, 以保持最佳的检索性能, 例如B树和B+树

B树

B树的定义
m 阶 B 树是一棵 m 路查找树, 或者为空, 或者:

- 下界 : 根节点至少有两个子树, 其他非叶节点至少有 $\lceil m/2 \rceil$ 个子树
- 上界 : 每个节点至多有 m 个子树
- 有 k 个子树的节点有 $k-1$ 个关键码
- 所有叶节点都在同一层, 有 $\lceil m/2 \rceil - 1$ 到 $m-1$ 个关键码

B 树的一个包含 j 个关键码, $j+1$ 个指针的节点的一般形式为:

- $P_0, K_1, P_1, K_2, P_2, \ldots, K_j, P_j$
  - K_i 是关键码值, $K_1 < K_2 < \cdots < K_j$
  - P_i 是指向包括 K_i 到 K_{i+1} 的子树的指针

每个关键码对应一个记录指针域

B树的特点

- 访问局部性
- 高度平衡
- 父节点关键码值是子节点的分界
- 节点关键码至少一定比例是满的
  - 改进空间利用率, 减少检索, 更新操作的I/O次数

B树的查找 : 从根节点开始, 逐层查找, 如果指向空节点, 则查找失败

B树的检索长度 : 假如高度是 h, 则自顶向下检索到叶节点的过程可能需要 h 次读盘, 最多 $h+1$ 访外

B树的插入

- 找到最底层插入
- 若溢出, 分裂节点, 中间关键码连同新指针插入父节点
- 若父节点溢出, 继续分裂, 可能一直到根节点(则树升高一层)

> 节点分裂方法, 假如插入后节点状态为 $p=(m, A_0, K_1, A_1, \ldots, K_m, A_m)$, 则拆分成两个节点 $p=(\lceil m/2 \rceil-1, A_0, K_1, A_1, \ldots, K_{\lceil m/2 \rceil-1}, A_{\lceil m/2 \rceil-1})$ 和 $q=(m-\lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil+1}, A_{\lceil m/2 \rceil+1}, \ldots, K_m, A_m)$, 中间关键码 $K_{\lceil m/2 \rceil}$ 与指向新节点 q 的指针插入父节点

B树的删除

- 删除的关键码在叶节点
  - 若删除后节点关键码数小于 $\lceil m/2 \rceil-1$
    - 若兄弟节点关键码数大于 $\lceil m/2 \rceil-1$, 则从兄弟节点借关键码 (父节点分界关键码也要调整)
    - 若兄弟节点关键码数等于 $\lceil m/2 \rceil-1$, 则合并节点
  - 若删除后节点关键码数不小于 $\lceil m/2 \rceil-1$, 则直接删除
- 删除的关键码不在叶节点层
  - 先把此关键码替换为其后继关键码, 再删除该关键码

B+树

B树的变种, 所有关键码都在叶节点, 各层节点中的关键码是下一层相应节点中最大关键码(或最小关键码)的拷贝

B+树的特点

- 每个节点子节点数量在 $[\lceil m/2 \rceil, m]$, 根节点可以只有两个子节点 (空, 独根除外) 这一点与B树相同
- B+树和B树的差异
  - B+树中 n 棵子树的结点中含有 n 个关键码, 而B树中 n 棵子树的结点中含有 $n-1$ 个关键码
  - B+树叶子结点包含了完整的索引的信息，而B树所有结点共同构成全部索引信息
  - B+树所有的非叶结点可以看成是高层索引，结点中仅含有其子树中最大(或最小)关键码

B+树的查找 : 逐层找

- b+树叶节点一般链接起来, 形成双链表, 便于范围查询, 需要的话每一层节点也都可以这样链接

B+树的插入 : 与B树类似, 左 $\lceil m/2 \rceil$, 右 $\lfloor m/2 \rfloor$, 注意上层关键码的维护

> 两棵树分裂时都是左节点 $\lceil m/2 \rceil$ 个子节点

B+树的删除 : 与B树类似, 但被删除关键码的上层副本可以选择保留, 作为"分界关键码"存在

性能分析

m 阶 B树, 关键码为 N 个, 内部节点数为 p, 则存取次数 k 满足

$$k \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2} \right)$$

插入一个关键码的平均分裂节点次数为

$$s = \frac{p-1}{N-1} \leq \frac{N-1}{(\lceil m/2 \rceil - 1)(N-1)} = \frac{1}{\lceil m/2 \rceil - 1}$$

红黑树

定义

BST的染色版:

- 每个节点要么是红色, 要么是黑色
- 根节点, 叶节点是黑色
- 父子节点不能同时为红色
- 任意结点到叶节点的路径上黑色结点数相同

节点的阶

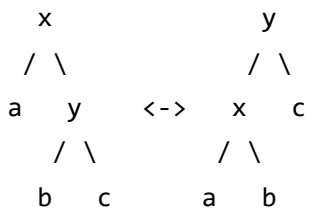
- 即从该节点到叶节点的黑色节点数, 不包括该节点本身, 包括叶节点
- 叶节点的阶为0, 根节点的阶为该树的阶

红黑树的性质

- 满二叉树 : 空树叶也看作节点
- k 阶红黑树树高 $\in [k+1, 2k+1]$
- k 阶红黑树内部节点数最少是 $2^k - 1$
- n 个内部节点的红黑树树高最大是 $2\log_2(n+1) + 1$
- 红黑树检索, 插入, 删除的最差时间都是 $O(\log n)$

操作

左旋与右旋 ->左旋(x), <-右旋(y)



保留BST的性质, 旋转后的树的中序遍历结果不变 $(a < A < b < B < c)$

红黑树的插入

- 插入节点为红色
- 若父节点为黑色, 不违反性质, **直接插入**
- 若父节点为红色, 则违反性质, 需要调整
  - 叔节点为红色, 则将父节点, 叔节点染黑, 祖父节点染红, **递归**调整祖父节点
  - 叔节点为黑色, 且当前节点为右子节点, 则左旋父节点, **转化**为左子节点的情况
  - 叔节点为黑色, 且当前节点为左子节点, 则右旋祖父节点, 旋转前的祖父节点染红, 父节点染黑, **结束**
- 时间复杂度 $O(\log n)$, 因为递归调整的次数不超过树高的一半

BST的删除

- 没有儿子, 直接删除
- 有一个儿子, 用儿子替代
- 有两个儿子, 找到右子树的最小节点(即从右子树根出发递归找左儿子), 用该节点替代, 再删除该节点(该节点最多只有一个右儿子) (或者找左子树的最大节点)

红黑树的删除 : 不考, 懒得写了