索引

目的

- 索引通过存储（查找键, 数据位置）来快速定位记录
  - 稠密索引：数据位置→记录的位置（页面，槽位）。稠密索引中，每一条记录对应一个索引条目，索引条目中存储了记录的键值和指针。
  - 稀疏索引：数据位置→页面的位置（页面）。稀疏索引中，索引条目只包含部分记录的键值和指针，而不是每一条记录都建立索引。
  - 索引一般存储以页面为单位组织（内存索引除外）
- 索引也需要支持并发控制
- 插入记录时，先插入数据页面，然后插入索引，而且两者要保持事务一致性

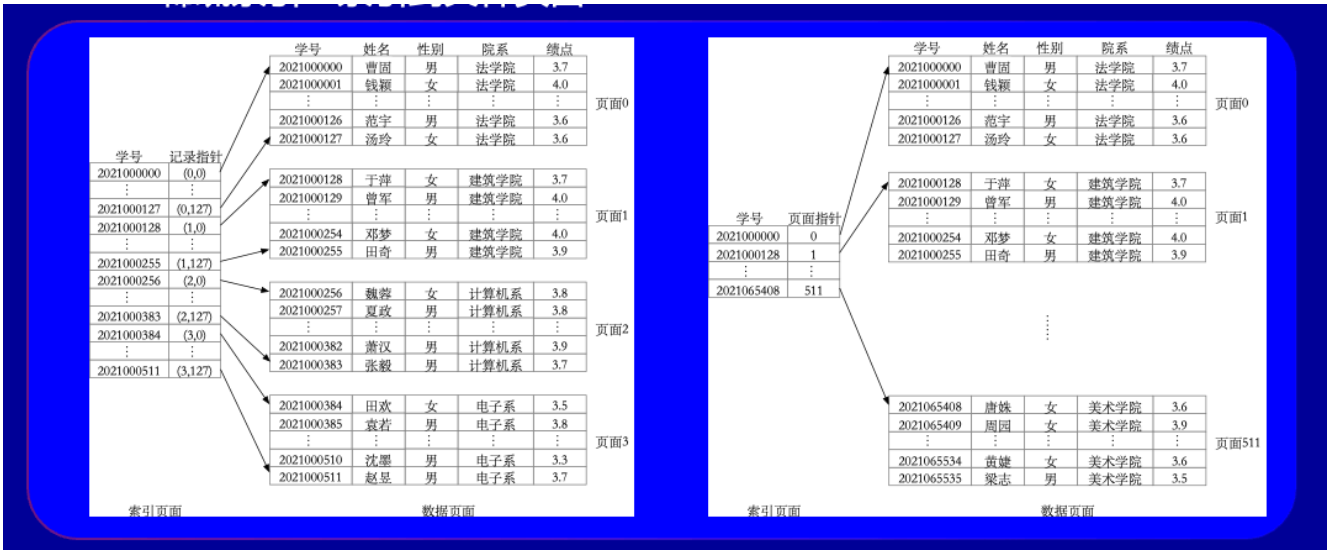
分类

聚集索引与非聚集索引（按物理存储分类）

- 聚集索引：数据按照查找键排序
  - 按照查找键连续存储，聚集索引上范围查询效率更高，一张表只能有一个
  - 维护增删时需要移动数据文件中的记录，开销稍大
- 非聚集索引：辅助索引、二级索引
  - 一张表可以有多个，数据不一定按照索引列顺序存储
- 例子

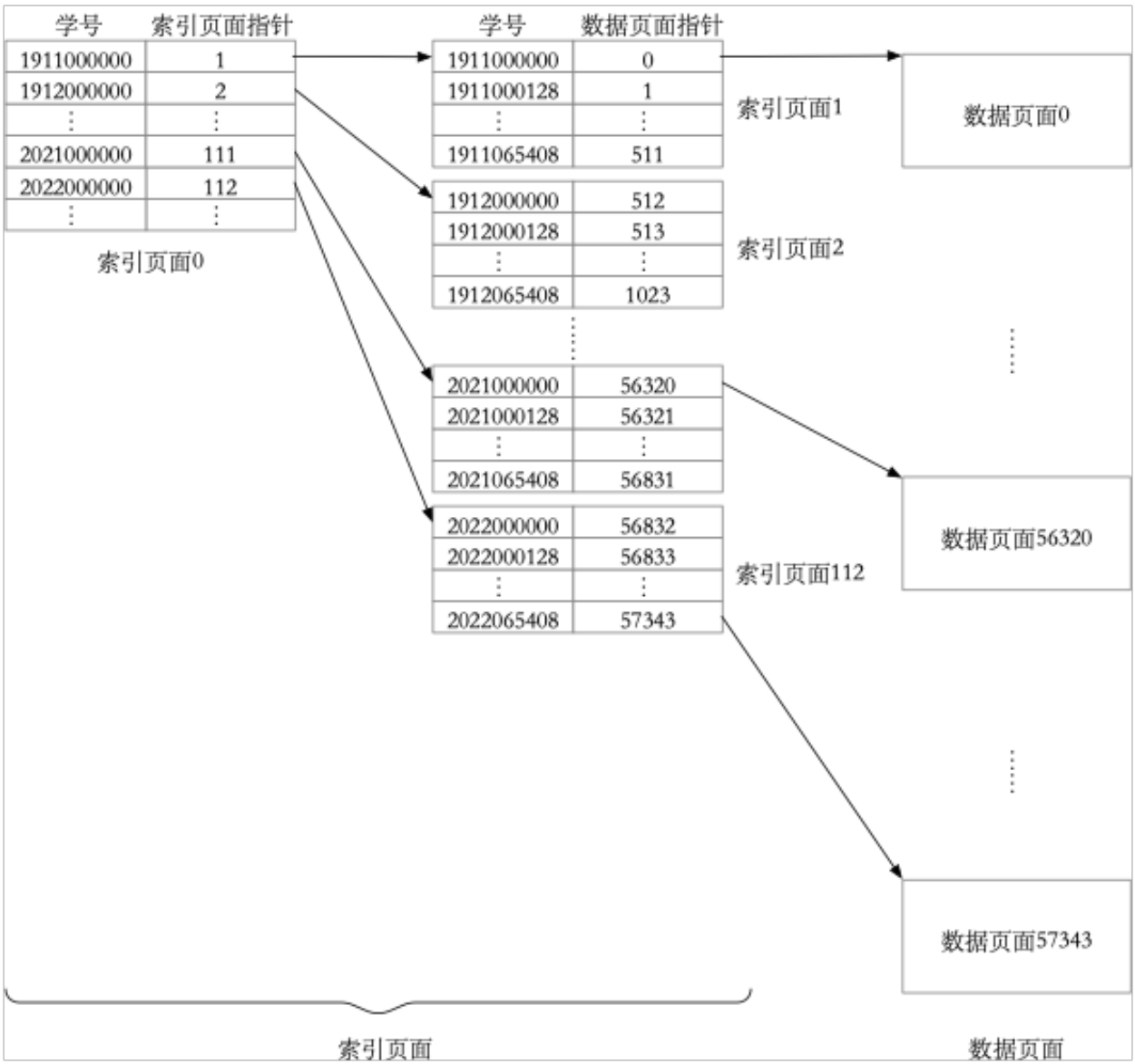
稠密索引与稀疏索引（按指针记录的粒度分类）

- 稠密索引：索引到数据记录。稠密索引中，每一条记录对应一个索引条目，索引条目中存储了记录的键值和指针。
- 稀疏索引：索引到文件页面。
- 稀疏索引中，索引条目只包含部分记录的键值和指针，而不是每一条记录都建立索引。
- 例子



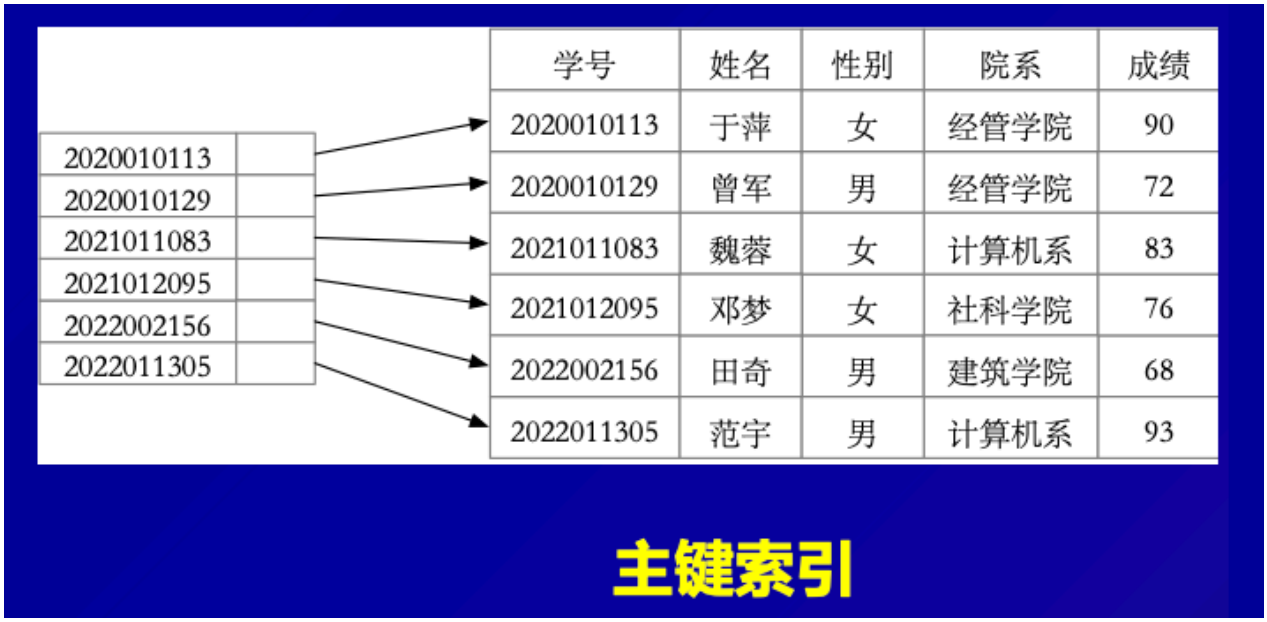
单级索引与多级索引（按索引层数分类）

- 单级索引
  - 一层索引：如哈希
- 多级索引
  - 索引文件上再建索引，如：B+tree
  - 多级索引目的
    - 多层过滤
    - 进一步降低I/O次数
    - 提升查找速度
  - 例子



主键索引、唯一索引和普通索引（按字段特性分类）

- 主键索引
  - 建立在主键字段上的索引
  - 一张表最多只有一个主键索引
  - 索引列的值不允许有空值
  - 例子



- 唯一索引
  - 建立在 UNIQUE 字段上的索引
  - 一张表可以有多个唯一索引
  - 索引列的值必须唯一，但是允许有空值

单列索引与多列索引（按字段个数分类）

- 单列索引
  - 建立在单个字段上的索引
- 多列索引
  - 建立在多个字段上的索引
- 例子

组织表

- **用B+树索引组织数据页面，数据页面使用顺序组织方式**
- **叶子节点为数据页面**
- **主键上的B+树聚集索引**
  - 主键上范围查询时更多连续I/O
  - 插入/删除数据时要维护主索引，开销增大
- **索引是否回表：回表指的是通过索引找到页面记录才返回结果**
  - Index only scan不回表：仅查找索引列，根据索引就可以返回结果，如学号是否存在
  - Index scan回表：根据索引列查找其他列的内容，如根据学号查找姓名

索引组织表中，按照主键组织数据，数据记录会经常移动，物理位置改变

数据文件	数据记录指针	代表
索引组织表	主键	MySQL、SQLite
堆表	(页面号, 槽号)	PostgreSQL、DB2

索引类型

- **不同类型索引支持不同的数据类型、条件类型**

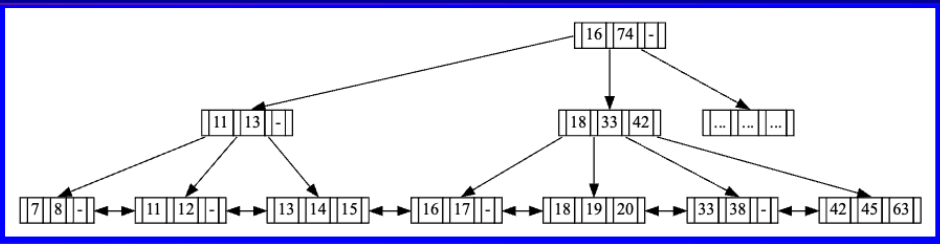
查询类型	索引类型
点查询（score=90）	哈希索引、B+树索引
范围查询（score>60）	B+树索引
多列条件查询(gender=male & rating=5)	位图索引
空间范围查询(115.7°<lag<117.4°& 39.4°<long<41.6°) 最近邻查询(KNN)	多维索引(R树、Quadtree、KD树)

- **是否支持持久化、可以否快速恢复**
  - 内存索引、磁盘索引

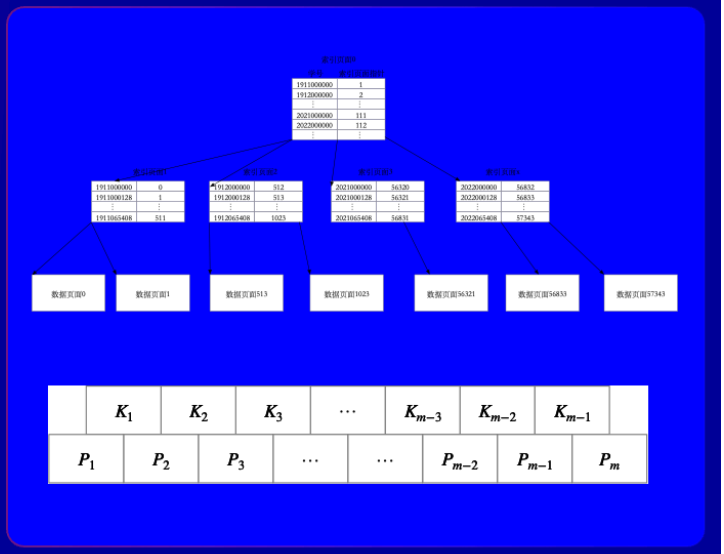
B+树

B+树索引结构

- **结构：平衡多叉查找树**
  - 根到所有叶子节点路径长度相同
- **扇出m：每个节点最多分叉数**
  - 子节点数目范围 $\left[\left\lceil\frac{m}{2}\right\rceil, m\right]$
  - 查找键数目范围 $\left[\left\lceil\frac{m}{2}\right\rceil - 1, m-1\right]$

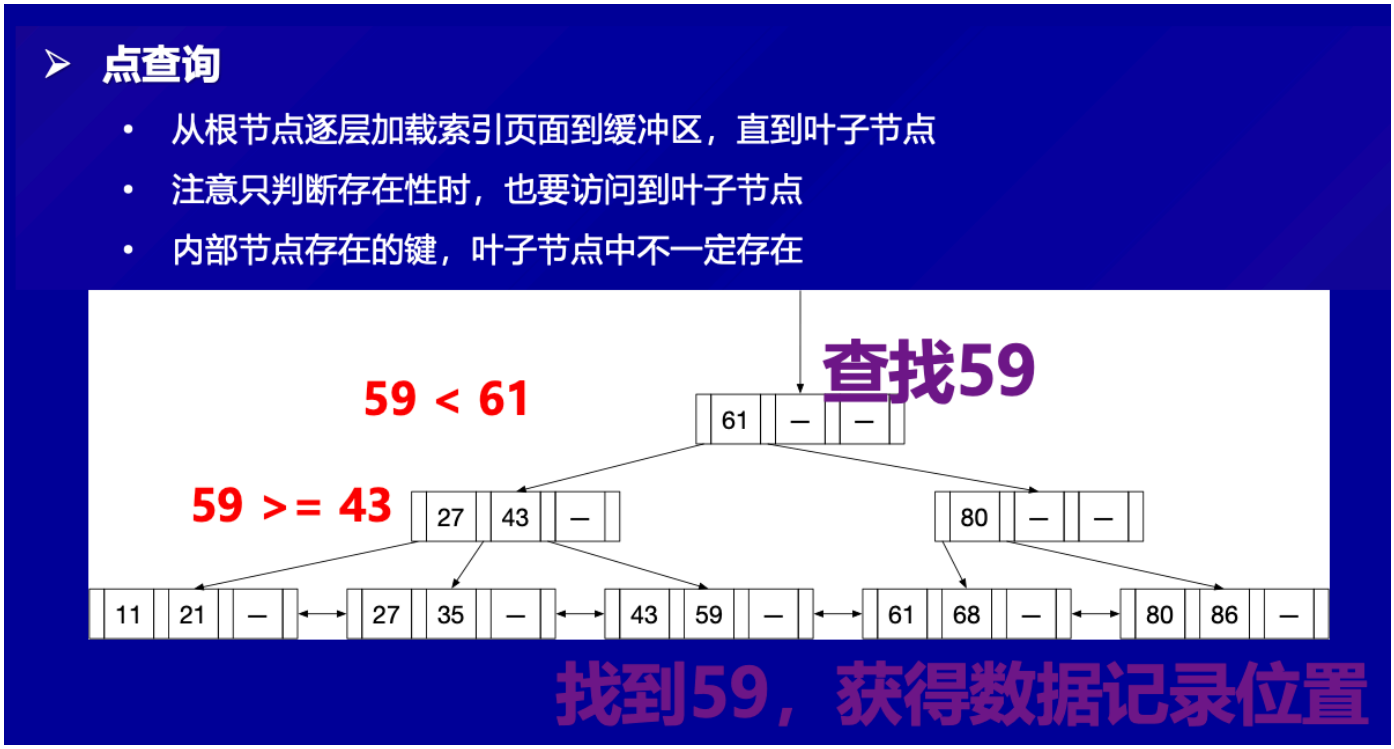


- **节点：页面（磁盘索引）**
  - 页面缓冲管理器
- **节点内指针**
  - 内部节点：索引页面号
  - 叶子节点：(数据记录页面号，数据记录槽号)
- $\mathbf{Tree}(P_i) < K_i \leq \mathbf{Tree}(P_{i+1})$
- **扇出m由页面大小和键值大小决定**
  - m通常200～300
  - 10亿条记录仅需要4～5层
  - I/O次数少

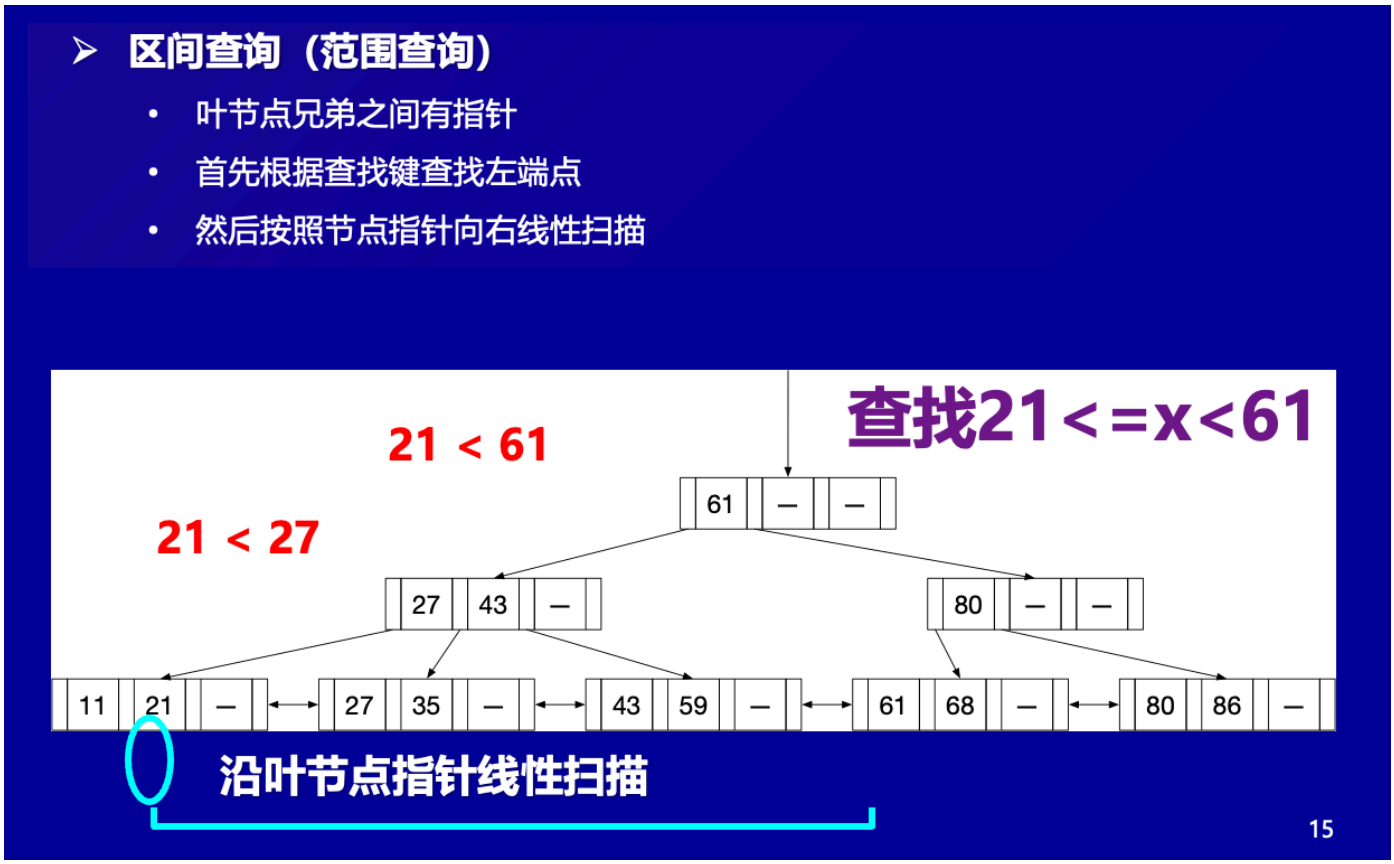


查找算法

点查询



区间查询



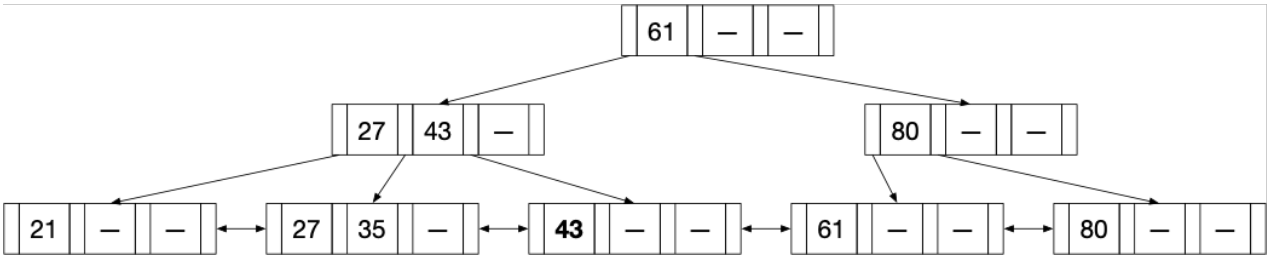
删除算法

- 找到待删除索引记录所在叶子节点
- 从叶子节点中删除索引记录
  - 如果节点中键数目大于等于**⌈m/2⌉-1,则结束**

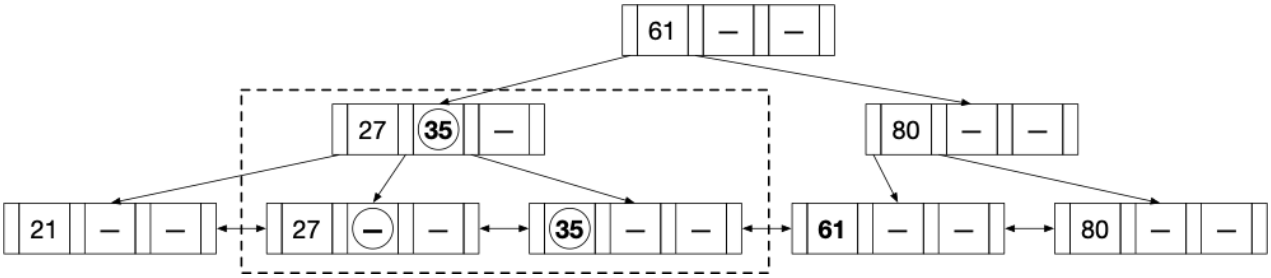
- 如果节点中键数目小于等于**⌈m/2⌉-2，发生下溢**
  - 尝试从兄弟节点借入索引记录,并更新父节点的键值
  - 若兄弟节点无法借入节点，则与兄弟节点合并，并从父节点删除兄弟节点的键值和指针，然后从父节点拉取下溢节点与兄弟节点之间的查找键

- 叶子节点下溢删除情况（可以从兄弟节点借入索引记录）

删除前（删除43）

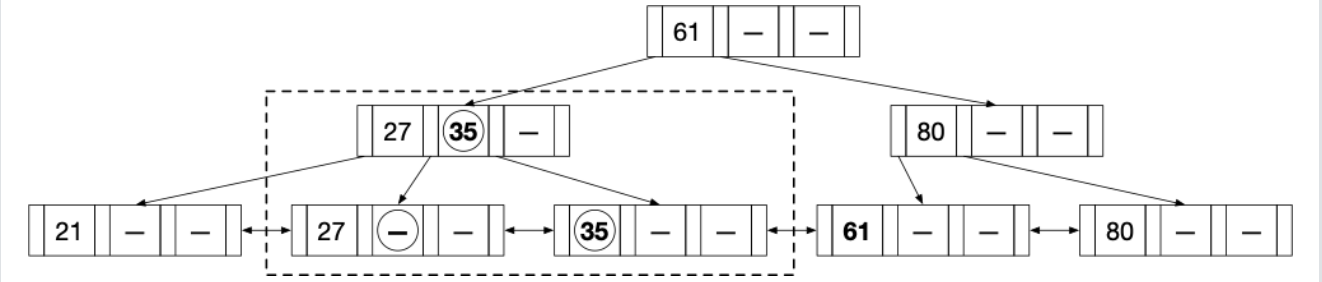


删除后

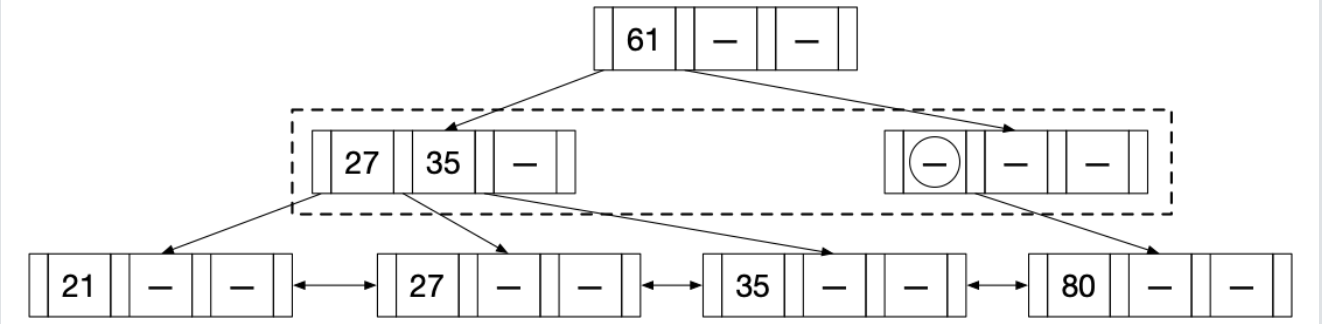


- 叶子节点下溢删除情况（无法从兄弟节点借入索引记录）

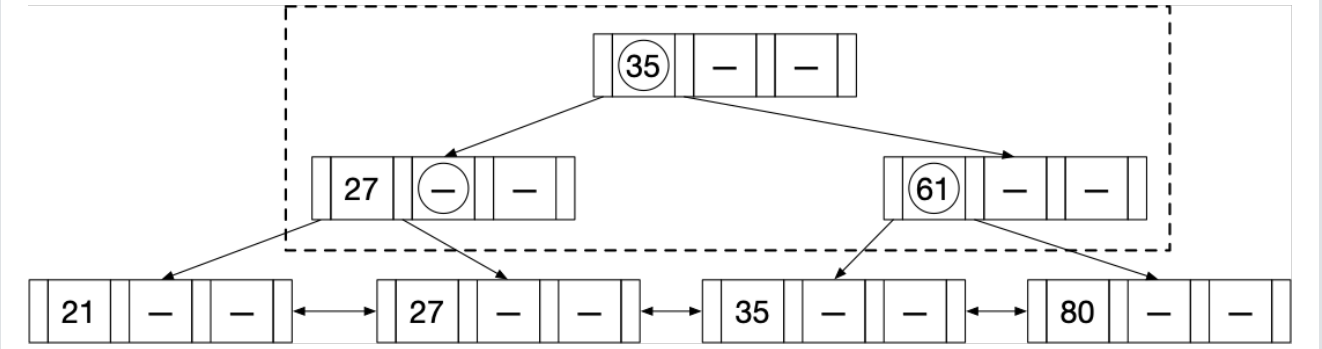
删除前（删除61）



删除后



此时父节点发生下溢，处理方法相同



创建索引指导

- 以下是是否在属性 A 上创建索引的主要因素：
  - 搜索频率：搜索 A 的频率，包括用于评估条件和联接的频率?
  - 更新频率：A 的更新频率（包括插入和删除）？
  - 候选键：A 的值必须是唯一的吗?
  - 大小：包含 A 的关系有多大?
- 如果满足以下所有条件，则应创建索引：
  - 该表包含大量元组（大表示超过 100,000 个）
  - 该属性包含广泛的值
  - 该属性包含大部分非 null 值
  - 应用程序查询经常在搜索/联接条件中使用该属性
  - 大多数查询检索一小部分 Tuples（比如 4%）
- 如果满足以下条件之一，则不应创建索引：

- 表不包含大量元组
- 应用程序查询很少在搜索/连接条件中使用该属性
- 大多数查询检索超过 4% 的元组
- 应用程序经常更改表内容。
- 维护成本：如果需要频繁更改，维护索引的成本可能会很高。

- 例题：考虑表 R（A， B， C， D） 和以下信息： A 是主键，不存在其他候选键。 大约 B 的不同值的数量是 C 的两倍，并且这个比率不会改变。 在针对 R 的每 100 次操作中，有 10 次是插入;10 个是删除，其中 5 个基于 A 的条件，5 个基于条件 B ;70次将是选择查询，其中 30 个基于 A 上的条件，20 个基于条件 B 和 20 个基于条件 C ;10 将进行更新，其中 5 个基于 B 的条件，5 个基于 C 的条件。所有条件都是相等条件。 讨论应该使用哪个属性来构建主索引，以及应该使用哪个属性来构建二级索引。
  - 从给定的信息中，可以得出以下内容： 在每 100 次操作中： 45 个需要基于 A 进行检索（10 个插入、5 个删除、30 个选择） 30 个需要基于 B 进行搜索（5 个删除、20 个选择、5 个更新） 25 个需要基于 C 进行搜索（20 个选择，5 个更新） 0 需要根据 D 进行搜索。A 应具有二级索引（具有主索引在 A 上并不能提高效率，因为最多一个元组将满足每个条件）。 D 不应有任何索引，因为它不在任何搜索中。 B 应该有一个二级索引，C 应该有一个主索引。尽管涉及 B 的所有条件的百分比略大于 C 的百分比，但 Tuples 的数量满足涉及 C 的每个条件是涉及 B 的 2倍，因为 B 的不同值数是C的两倍。这意味着，如果 B 和C 具有二级索引，基于 C 的每个搜索都将费用是每次基于 B 的搜索的两倍。因此在C上拥有主索引将比在B上具有主索引能减小更多花费。
  - ？？？