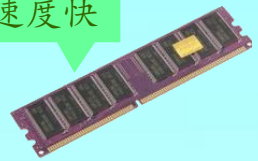
本节主题：

B-树

问题的提出

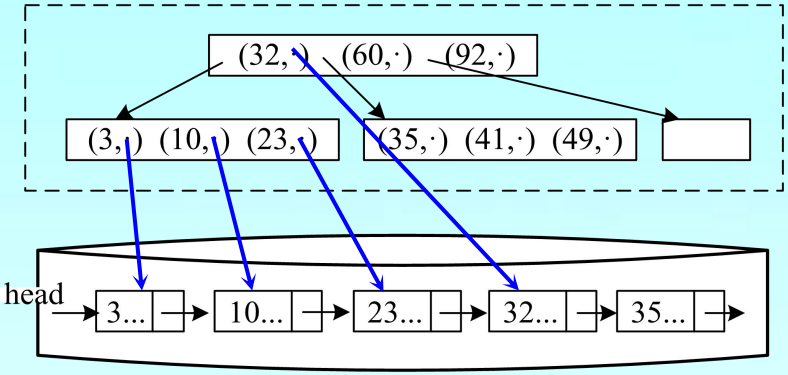
- 顺序查找、二分查找
  - 静态查找
- 二叉排序树、平衡二叉树
  - 内部动态查找——数据量小，在内存中完成
- B树
  - 用作外部查找
  - 也称多路平衡查找树
  - 是一种组织和维护外存文件系统非常有效的数据结构
- B树的应用背景
  - 大文件
  - （多级）顺序索引文件

容量小
价格高
速度快



多级的顺序索引文件

- 为每个页块建立一个索引项
- 必要时为索引再建索引
- 索引项只包括关键字及指向文件中区块的指针



容量大
价格低
速度慢
非易失

主(数据)文件

- 以页块为单位存储在外存中
- 包括关键字，以及全部数据

B-树定义

- B-树中所有节点中孩子节点的最大值称为**B-树的阶**
  - 阶数常用m表示，从查找效率考虑，要求m≥3
  - m越大，综合效率越高
  - m受存储管理中块大小的限制，也不可太大

n	p_0	k_1	p_1	k_2	p_2	…	k_n	p_n

- n为该节点中的关键字个数
- 除根节点外，对其他所有节点，$\lceil m/2 \rceil - 1 \le n \le m-1$
- $k_i (1 \le i \le n)$为该节点的关键字且满足$k_i < k_{i+1}$；
- $p_i (0 \le i \le n)$为该节点的孩子节点指针，且对p_i节点上的关键字k，$k_i \le k < k_{i+1}$。

- 定义：一棵**m阶B-树**或者是一棵空树，或者是满足下列要求的m叉树：

（1）树中每个节点至多有m个孩子节点（即至多有m-1个关键字）；

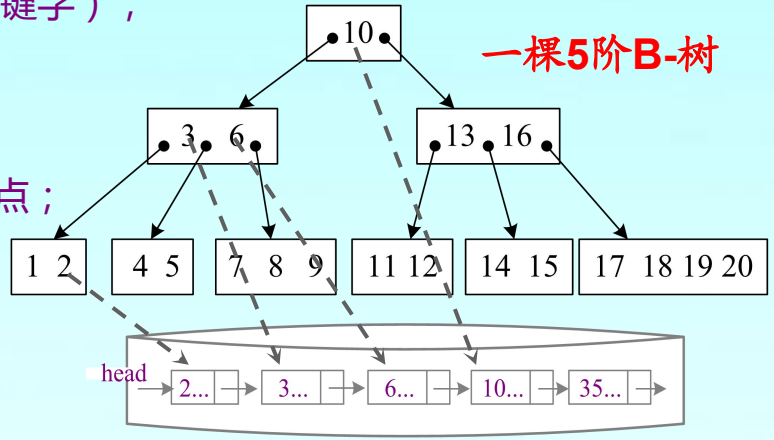
（2）除根节点外，其他节点至少有$\lceil m/2 \rceil$个孩子节点
（即至少有$\lceil m/2 \rceil - 1$个关键字）；

（3）若根节点不是叶子节点，则根节点至少有两个孩子节点；

（4）每个节点的结构为 

（5）所有叶子节点都在同一层上，

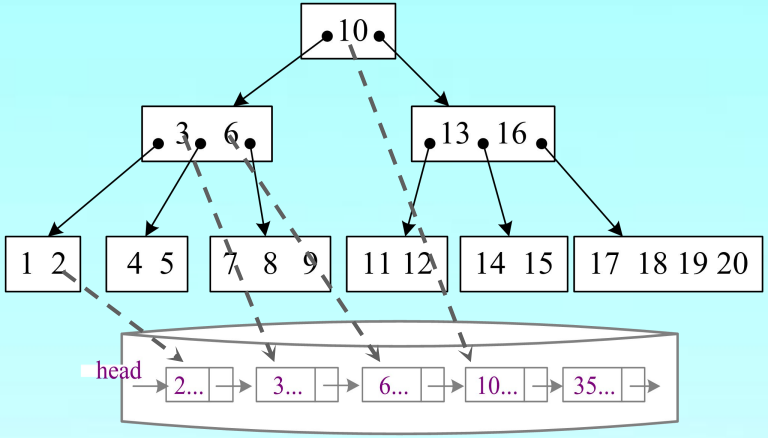
即B-树是所有节点的平衡因子均等于0的多路查找树
——自平衡多叉查找树

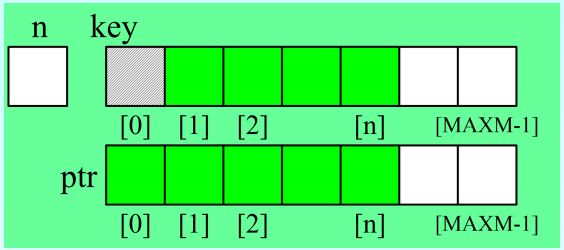


B-树的存储结构

n	p_0	k_1	p_1	k_2	p_2	…	k_n	p_n

```
#define MAXM 10                          //B-树的最大的阶数
typedef int KeyType;       //关键字类型
typedef struct node        //B-树节点类型
{
  int keynum;                    //节点当前的关键字的个数
  KeyType key[MAXM];      //[1..keynum]存放关键字,[0]不用
  FileBlock *filePoint[MAXM]; //指向外存主文件中数据块的地址
  struct node *parent;     //双亲节点指针
  struct node *ptr[MAXM];  //孩子节点指针数组[0..keynum]
} BTNode;
```

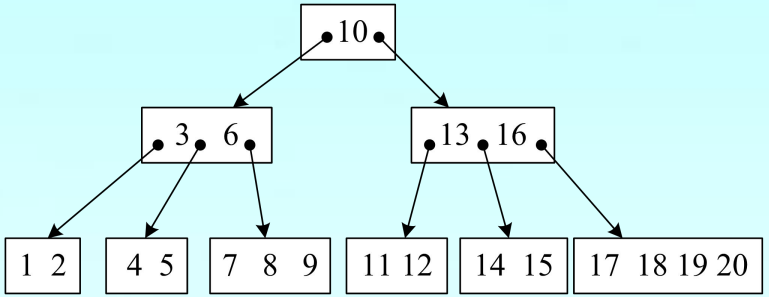




B-树的查找

- 在一棵B-树上顺序查找关键字 k
  - 将k与根节点中的key[i]进行比较：
    （1）若k=key[i]，则查找成功；
    （2）若k < key[1]，则沿着指针ptr[0]所指的子树继续查找；
    （3）若key[i] < k < key[i+1]，则沿着指针ptr[i]所指的子树继续查找；
    （4）若k > key[n]，则沿着指针ptr[n]所指的子树继续查找。
- 特点
  - n+1路的查找
  - 可以用折半的方法，
    对有序的数据key[1..n]查找

n	p_0	k_1	p_1	k_2	p_2	...	k_n	p_n

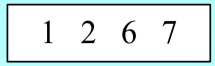


B-树的插入

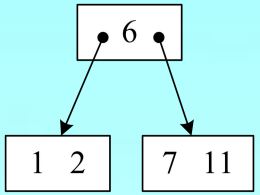
- 问题
  - 将关键字k插入到B-树
- 步骤：分两步完成

  （1）利用B-树的查找算法找出该关键字的插入节点（插入节点一定要是索引部分的叶子节点）

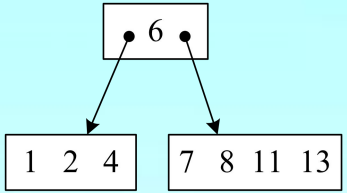
  （2）判断该节点是否还有空位置，分情况处理
  - 若该节点满足n < m-1，说明该节点还有空位置
    - 把关键字k插入到该节点的合适位置上；
  - 若该节点有n=m-1,说明该节点已没有空位置
    - 把节点分裂成两个
- 例
  - 创建一棵5阶B-树（如图）



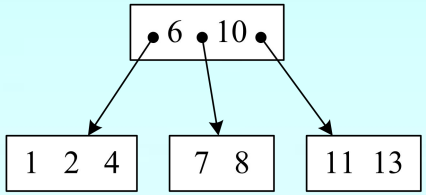
(a)插入1,2,6,7



(b)插入11



(c)插入4,8,13

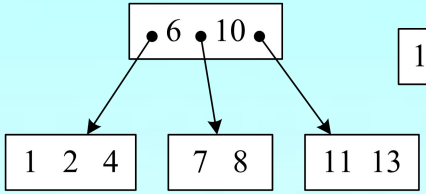


(d)插入10

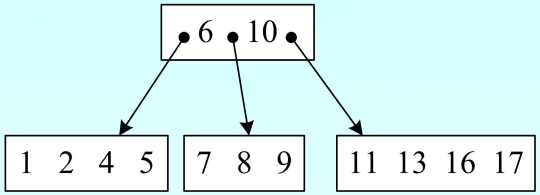
例　B-树的创建

问题

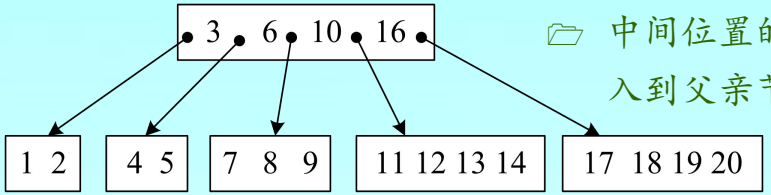
- 关键字序列为：
  {1,2,6,7,11,4,8,13,10,5,17,9,16,3,20,12,14,18,19,15}
- 创建一棵5阶B-树。



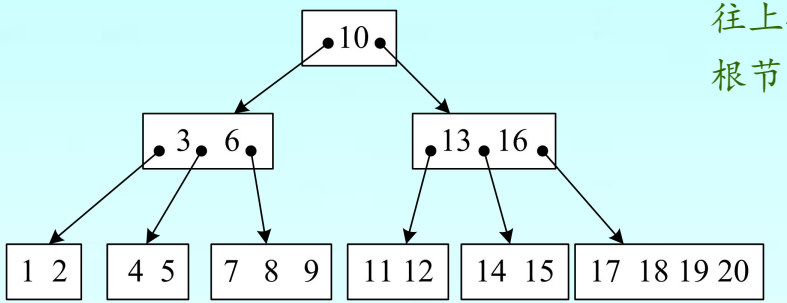
(d)插入1,2,6,7,11,4,8,13,10



(e)插入5,17,9,16



(f)插入3,20,12,14,18,19



(h)插入15

- 分裂的做法：取一新节点，把原节点上的关键字和k按升序排序后，从中间位置（即⌈m/2⌉处）把关键字（不包括中间位置的关键字）分开
  - 左部分所含关键字放在旧节点中
  - 右部分所含关键字放在新节点中
  - 中间位置的关键字连同新节点的存储位置插入到父亲节点中
- 如果父节点的关键字个数也超过Max，则要再分裂，再往上插，直至这个过程传到根节点为止。

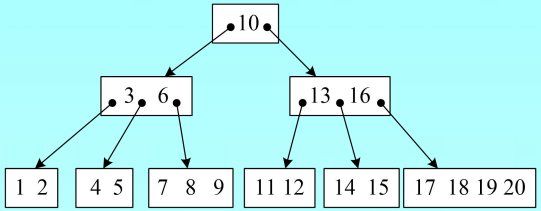
B-树的删除

- 问题
  - 在B-树上删除关键字k
- 要求
  - 删除后的节点中的关键字个数≥⌈m/2⌉-1
  - 节点中关键字个数太少，要“合并”
- 过程（两步）

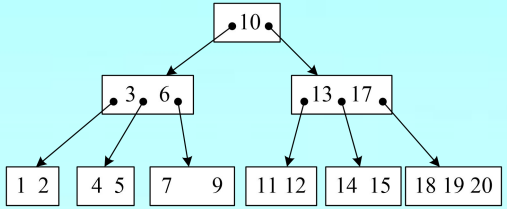
（1）利用查找算法找出该关键字所在的节点

（2）在节点上删除关键字k分两种情况

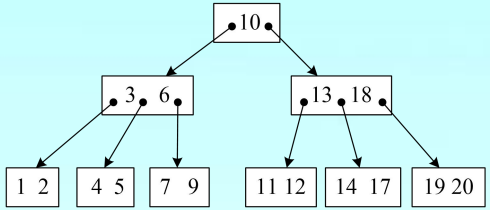
- 在叶子节点上删除关键字
- 在非叶子节点上删除关键字



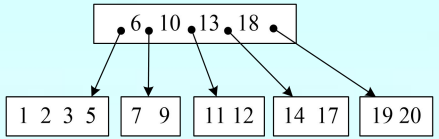
(a) 初始5阶B-树



(b) 删除8,16后的结果



(c) 删除15后的结果



(d) 删除4后的结果

在非叶子节点上删除关键字关键字key[i](1≤i≤n)的过程

- 在删去该关键字后，以该节点ptr[i]所指子树中的最小关键字key[min]来代替被删关键字key[i]所在的位置（ptr[i]所指子树中的最小关键字key[min]一定是在叶子节点上）。
- 然后再以指针ptr[i]所指节点为根节点查找并删除key[min]（即再以ptr[i]所指节点为B-树的根节点，以key[min]为要删除的关键字，然后再次调用B-树上的删除算法），把在非叶子节点上删除关键字k的问题转化成了在叶子节点上删除关键字key[min]的问题。具体有如下三种情况

① 假如被删节点的关键字个数大于Min，说明删去该关键字后该节点仍满足B-树的定义，则可直接删去该关键字。

② 假如被删节点的关键字个数等于Min，说明删去关键字后该节点将不满足B-树的定义。（**合并方法**：若该节点的左（或右）兄弟节点中关键字个数大于Min，则把该节点的左（或右）兄弟节点中最大（或最小）的关键字上移到双亲节点中，同时把双亲节点中大于（或小于）上移关键字的关键字下移到要删除关键字的节点中，这样删去关键字k后该节点以及它的左（或右）兄弟节点都仍旧满足B-树的定义。）

③ 假如被删节点的关键字个数等于Min，并且该节点的左和右兄弟节点（如果存在的话）中关键字个数均等于Min。（合并方法：把要删除关键字的节点与其左（或右）兄弟节点以及双亲节点中分割二者的关键字合并成一个节点。如果因此使双亲节点中关键字个数小于Min，则对此双亲节点做同样处理，以致于可能直到对根节点做这样的处理而使整个树减少一层。）

```
typedef struct node  //B-树节点类型
{
  int keynum;              //关键字个数
  KeyType key[MAXM];       //关键字
  FileBlock *filePoint[MAXM];
  struct node *parent;     //双亲
  struct node *ptr[MAXM];  //孩子
} BTNode;
```