# 第15章

# 平衡搜索树

# 本章内容

- **15.1 AVL 树**
- *15.2 红-黑树
- *15.3 分裂树
- **15.4 B-树**

# 15.4 B-树

# 数据访问方法

- 当字典足够小，可以驻留在内存中时，AVL树和红-黑树都能够保证获得很好的性能。
- 对于较大的字典（外部字典或文件），它们必须存储在磁盘上。
- ?

# 15.4.1 索引顺序访问方法ISAM

- ISAM( Indexed Sequential Access Method)方法
  - 可用的磁盘空间被划分为很多块，**块是磁盘空间的最小单位**，被用来作为输入和输出。**字典元素以升序存储在块**中。
  - ISAM方法提供**顺序访问**和**随机访问**.

- 顺序访问:
  - 依次输入各个块，在每个块中按升序搜索元素。
  - 如果每个块中包含*m* 个元素, 搜索每个元素的磁盘访问次数是*1/m.*

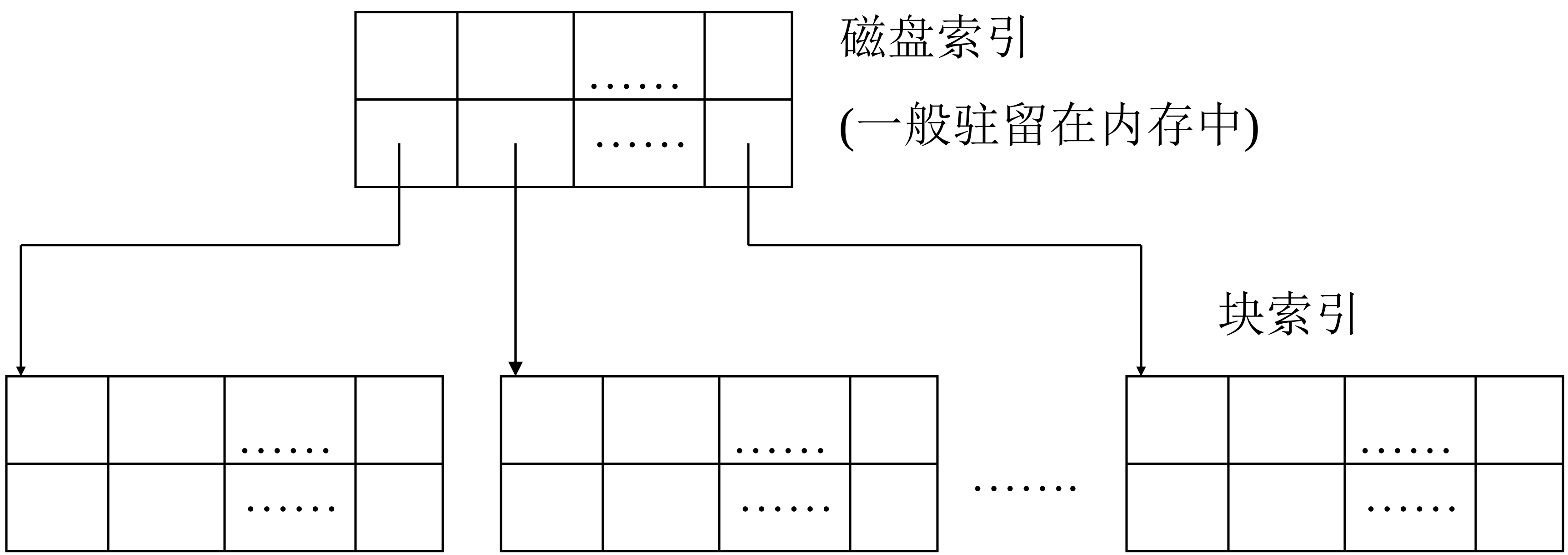
# 15.4.1 索引顺序访问方法ISAM

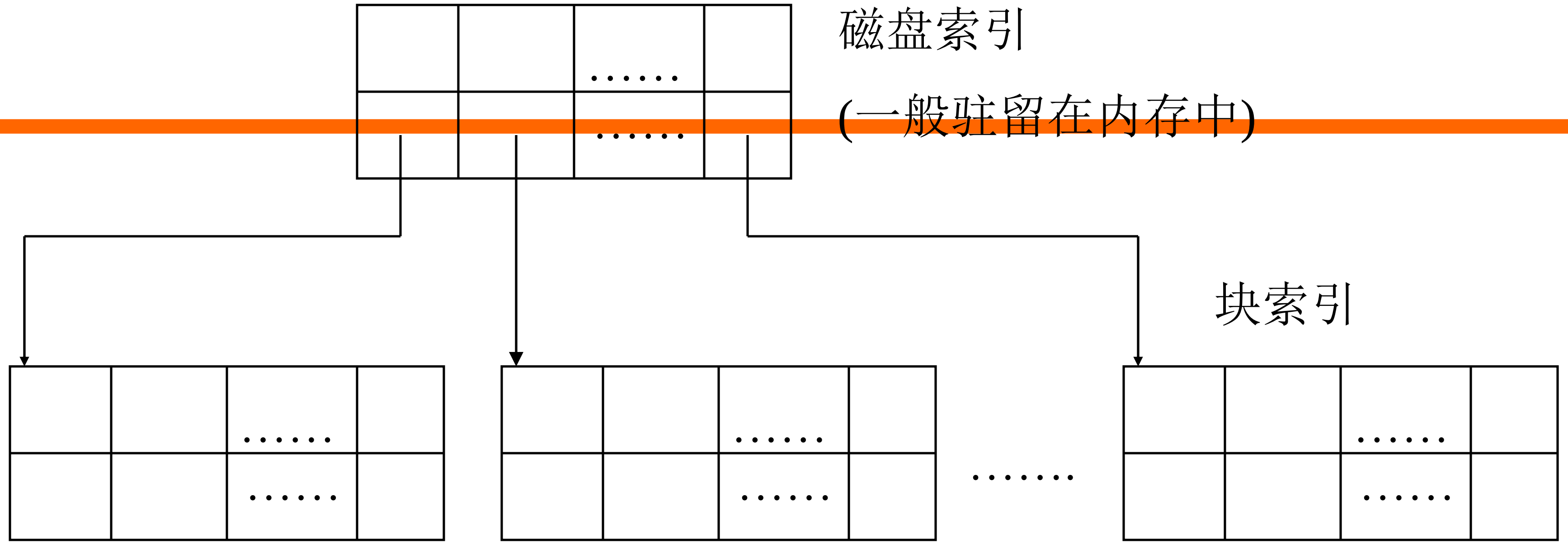
- 随机访问：
  - 必须维护一个**索引表**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 最大关键值 | | | | | …… | | |
| 块的位置 | | | | | …… | | |

- 索引一般足以驻留内存.
- 随机访问关键字为*k*的元素:
  - 搜索索引表
  - 关键字为*k*的元素所属的块从磁盘读入内存
  - 在块中搜索关键字为*k*的元素
- 一次随机访问需要一次磁盘访问

- 当字典跨越几个磁盘时。元素按升序被分配到各个磁盘以及每个磁盘的不同块中。
- 每个磁盘都有一个块索引





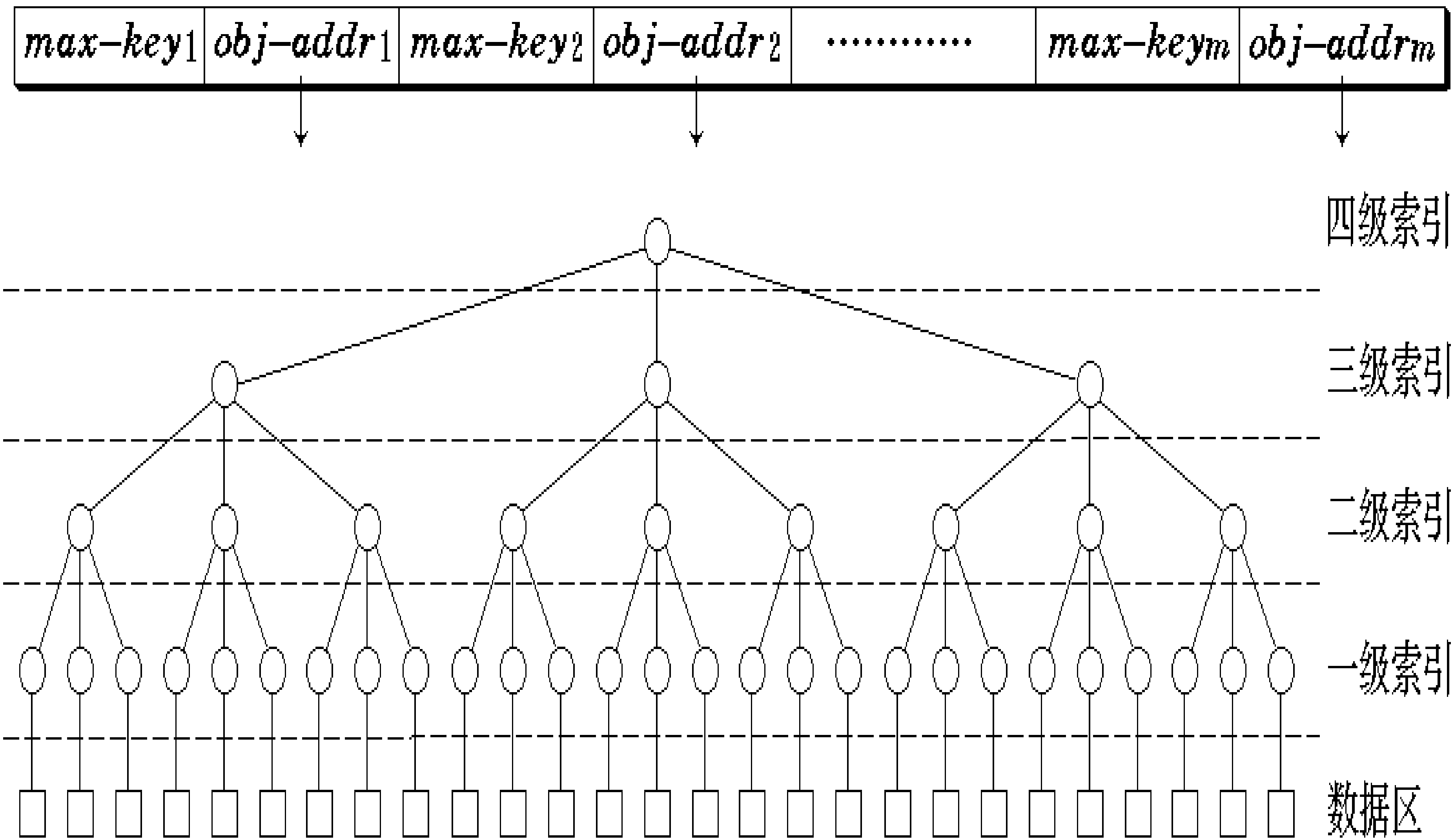

- 随机访问一个元素:
  - 搜索驻留内存的磁盘索引
  - 在相应磁盘中读入块索引并搜索元素所在的块
  - 从磁盘中读入块，并搜索元素
- 一次随机访问需要两次磁盘访问

# ISAM问题

- 当执行插入和删除操作时，会面临很大的问题——块间元素的移动
- 解决办法:在每个块中预留一些空间
  - 插入少量元素时，不需要块和块之间移动元素
  - 删除后，空间保留

# 15.4.2 *m*叉搜索树

- 当数据对象数目特别大，索引表本身也很大，在内存中放不下，需要分批多次读取外存才能把索引表搜索一遍。
- 在此情况下，可以建立索引的索引，称为二级索引。二级索引可以常驻内存，二级索引中一个索引项对应一个索引块，登记该索引块的最大关键码及该索引块的存储地址。
- 如果二级索引在内存中也放不下，需要分为许多块多次从外存读入。可以建立二级索引的索引，叫做三级索引。
- 必要的话，还可以有4级索引，5极索引，…。




# 多级索引结构形成*m*路搜索树

- 这种多级索引结构形成一种 $m$ 叉树。
  - 树中每一个分支结点表示一个索引块，它最多存放 $m$ 个索引项，每个索引项分别给出各子树结点 (低一级索引块) 的最大关键码和结点地址。
  - 树的叶结点中各索引项给出在数据表中存放的对象的关键码和存放地址。
  - 这种$m$叉树用来作为多级索引，就是$m$路搜索树。
  - 这时，访问外存次数等于读入索引次数再加上1次读取对象。
  - $m$路搜索树可能是动态索引结构，即在整个系统运行期间，树的结构随数据的增删及时调整，以保持最佳的搜索效率。

# 15.4.2 m叉搜索树

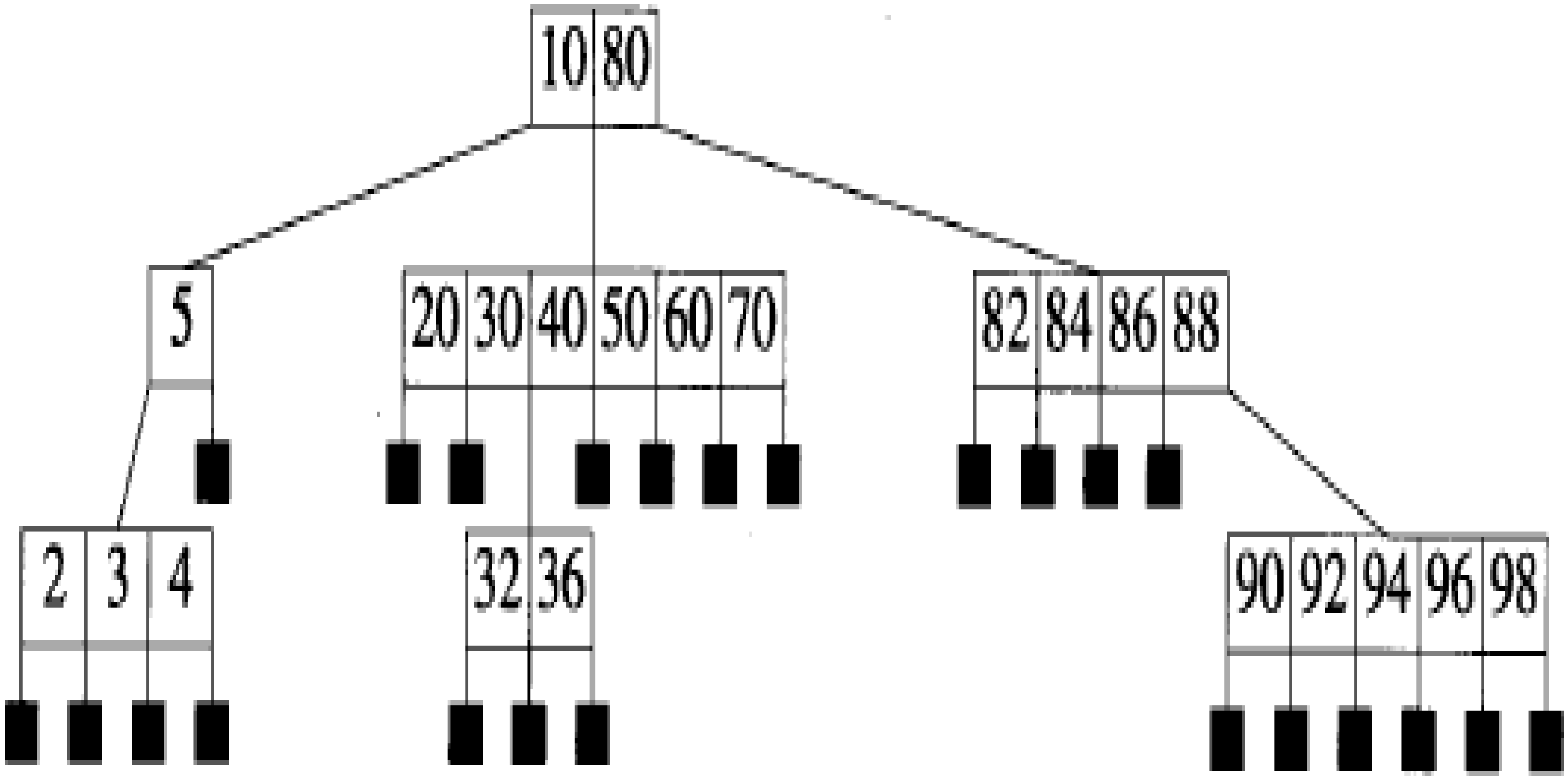
- m叉搜索树(m-way Search Trees )定义：
- m 叉搜索树可以是一棵空树，如果非空，它必须满足以下特征：
  - 1) 在相应的扩充搜索树中(用外部节点替换空指针)，每个内部节点**最多**可以有**m** 个子女及**1～m-1**个元素(外部节点不含元素和子女)。
  - 2) 每个含**p 个元素**的节点，有**p+ 1**个子女。
  - 3) (接下页)

3).含p 个元素的任意节点，设$k_1$，$k_2$，$k_3$，……...＜$k_p$是这些元素的关键值，$C_0$，$C_1$......$C_p$是该节点的p+1个孩子，节点内的排列:

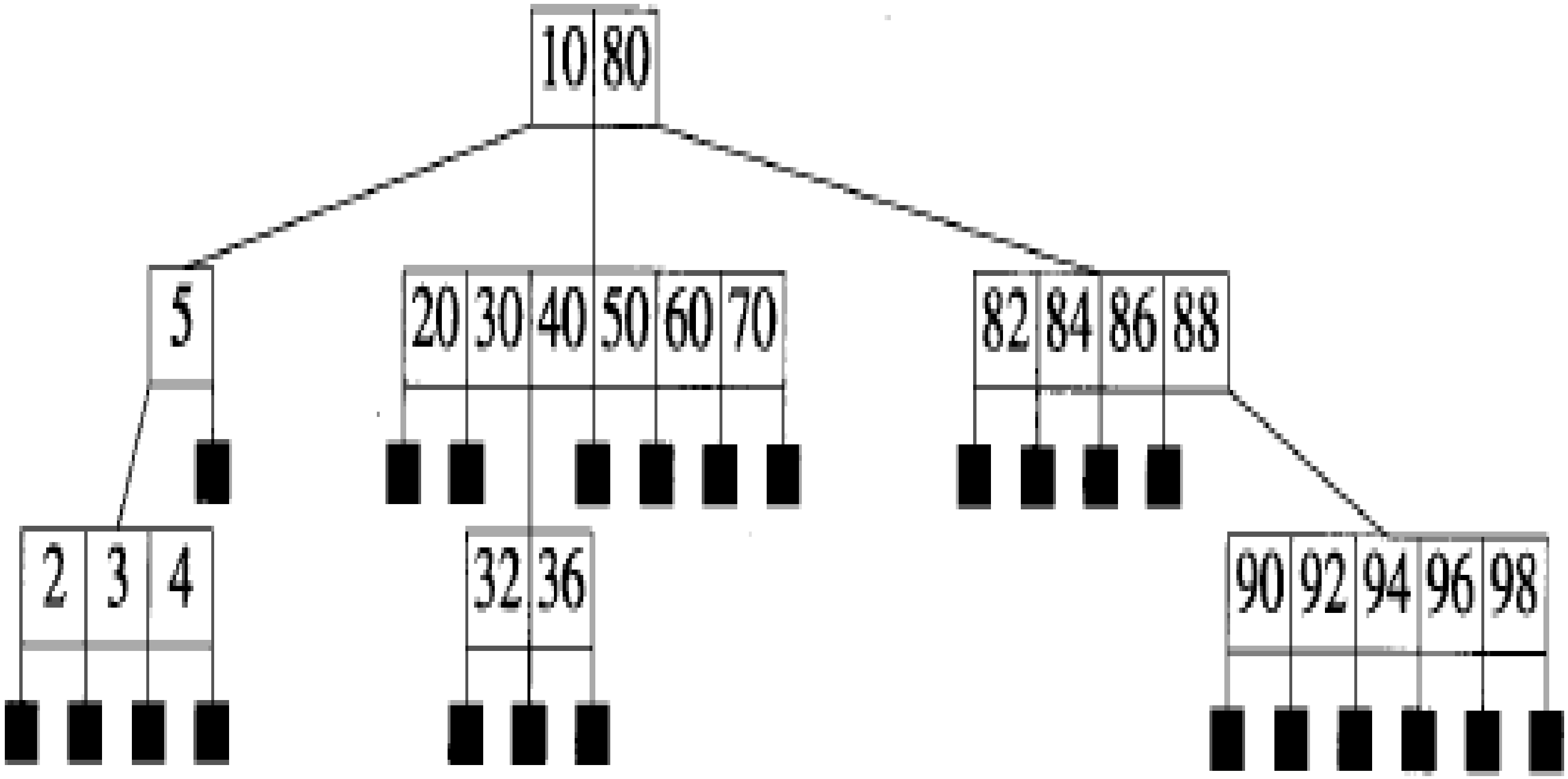
$C_0$ $e_1$ $C_1$ $e_2$ $C_2$ $e_3$ $C_3$ ………………….. $e_p$ $C_p$

- $k_1 < k_2 < k_3 < \ldots\ldots < k_p$
- $C_0$为根的子树： $< k_1$
- $C_1$为根的子树： $> k_1,$ $< k_2$
- …….
- $C_i$为根的子树： $> k_i,$ $< k_{i+1}$
- …
- $C_p$为根的子树： $> k_p$

# 例：7叉搜索树

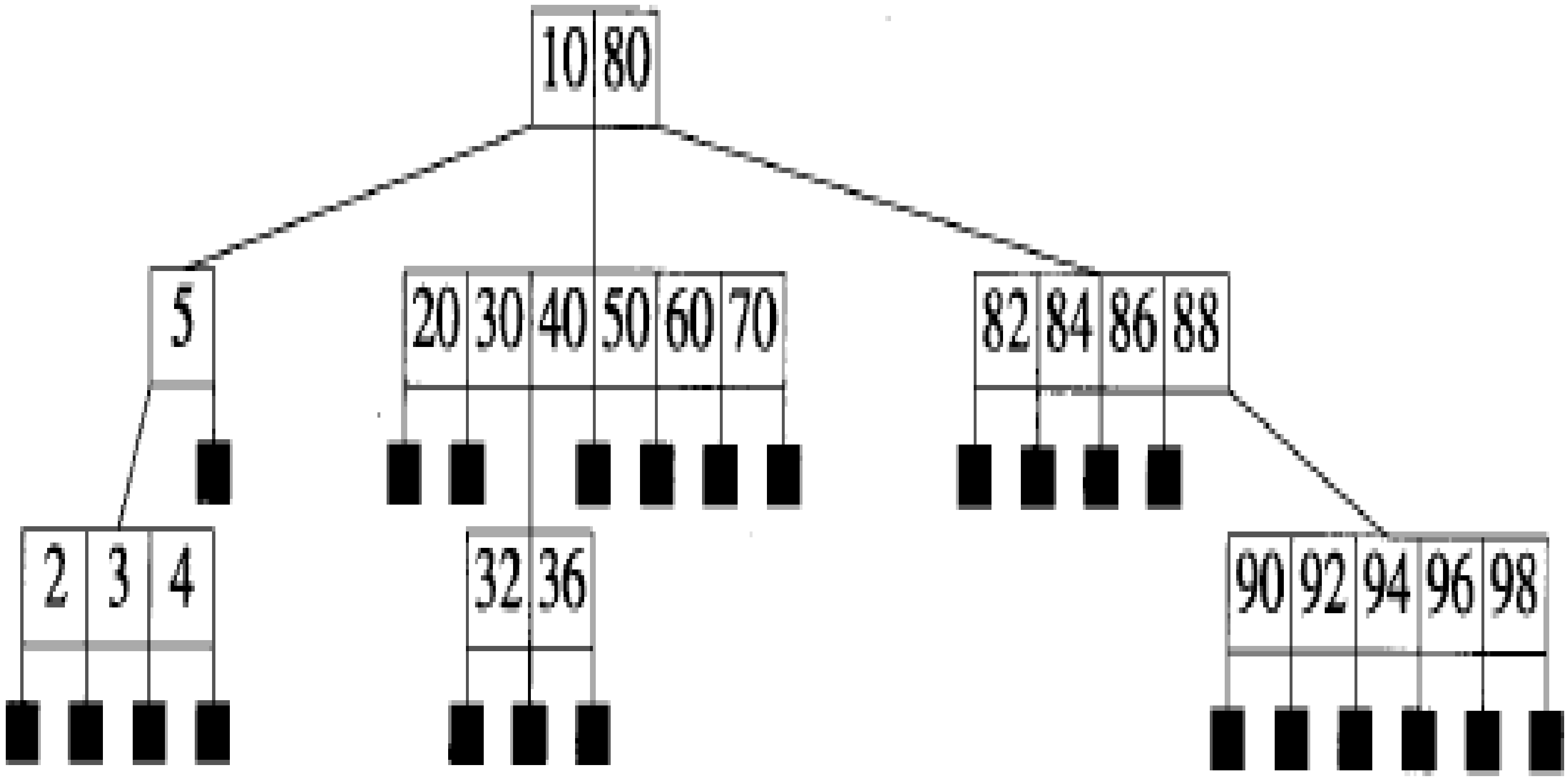


# m叉搜索树的搜索



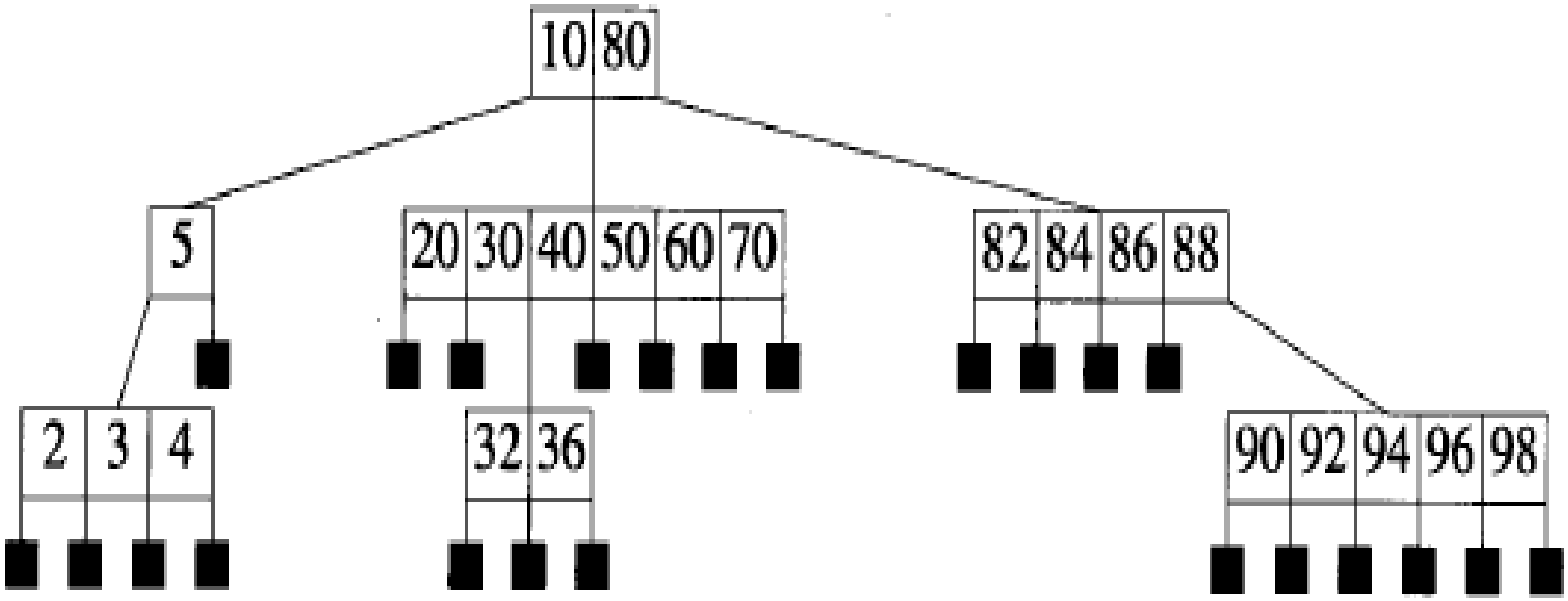

- 搜索: 32 ,31

# m叉搜索树的插入




- 插入：31,65
  1. 搜索31,65
  2. 能容纳？不能容纳？

# m叉搜索树的删除




- 删除: 20， 5， 10
  - 20： 相邻子树都为空, 简单地从节点中删除。
  - 5： 相邻子树一个不空,从不空的相邻子树中找一个元素替换被删除元素
  - 10： 相邻子树都不空， 从不空的相邻子树中找一个元素替换被删除元素

# m叉搜索树的高度

- 高度：h（不包括外部节点）
- 元素数目： n
- **$h \le n \le m^h-1$**
  - 证明：
  - 最少元素数：每层一个节点，每个节点一个元素。
  - 最多元素数：1~h-1层的每个节点m个孩子(h层节点没有孩子)，每个节点m-1个元素。
    - 最多节点数：$\sum_{i=0}^{h-1} m^i = (m^h - 1)/(m - 1)$

$\to \log_m(n+1) \le h \le n$

- 搜索、插入和删除操作需要的磁盘访问次数：O (h) 。

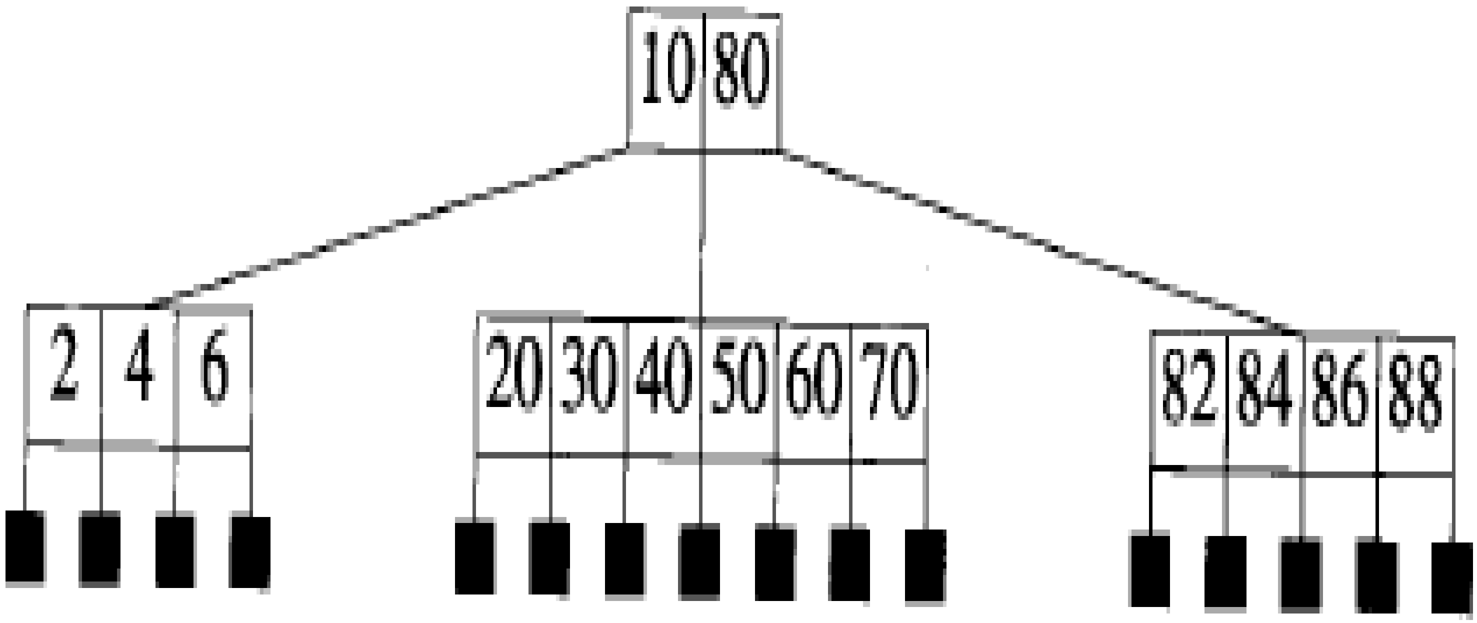
# m叉搜索树的高度

- 例，一棵高度为5的200叉搜索树
  - 最多能够容纳2005-1=32*1010-1个元素
  - 最少只容纳五个元素
- 例，一棵含有32*1010-1个元素的200叉搜索树
  - 高度最小是5
  - 高度最大为32*1010-1
- 为保证操作性能，必须确保高度值接近于logm(n+1)。

# 15.4.3 m阶B-树

- 定义[ m阶B-树(m序B-树)]：
- m阶B-树(B-Trees of Order m) 是一棵m叉搜索树，如果B-树非空，那么相应的扩充树满足下列特征：
  - 1) 根节点至少有**2**个孩子。
  - 2) 除了根节点以外，所有内部节点至少有$\lceil m/2 \rceil$个孩子。
  - 3 )所有外部节点位于同一层上。

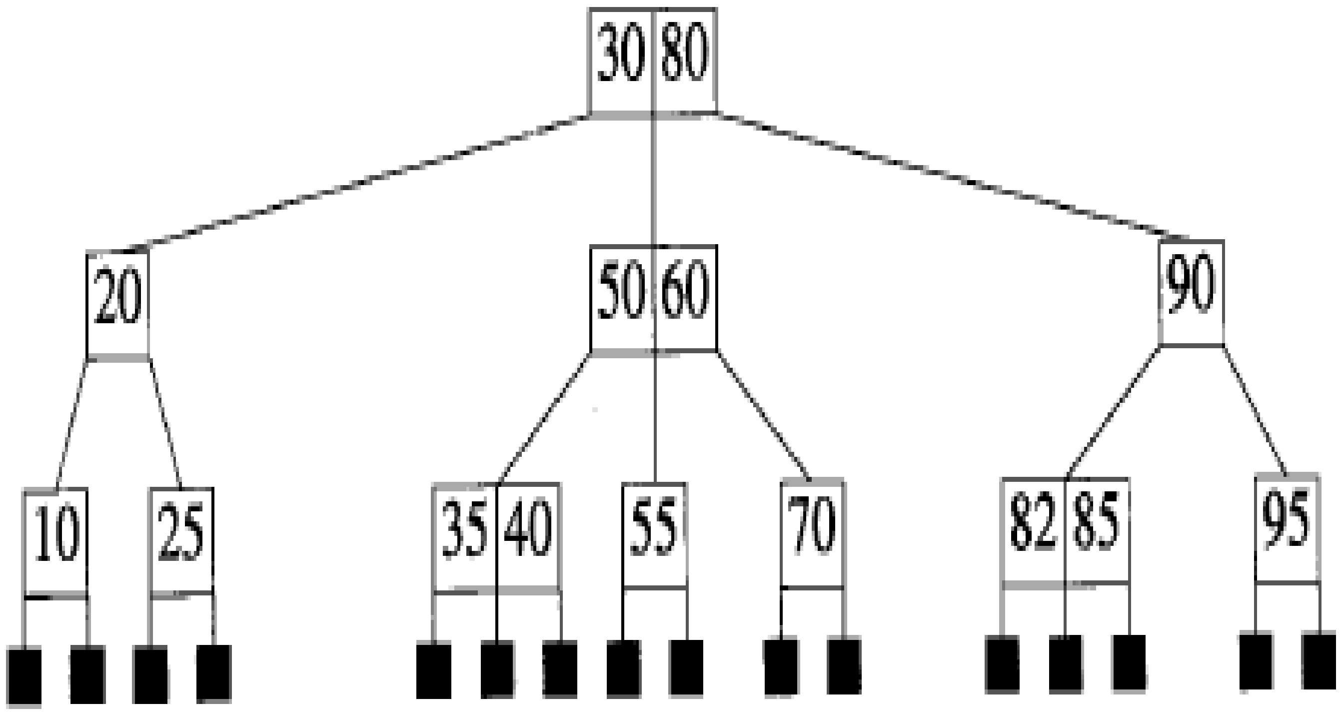
# 例：7阶B-树



# m阶B-树

- 2阶B-树⇒二阶B-树的所有内部节点都恰好有2个孩子。
  - 2阶B-树是一棵满二叉树(所有外部节点必须在同一层上).
- 3阶B-树
  - 3阶B-树的内部节点既可以有2个也可以有3个孩子，因此也把三阶B-树称作 2-3 树.
- 4阶B-树
  - 4阶B-树的内部节点必须有2个、3个或4个孩子，这种树也叫作2-3-4 树(或简称2 , 4树)。

# 例：  3阶B-树(2-3树)



# 15.4.4 B-树的高度

- 定理15-3：
  - 设T是一棵高度为h的m阶B-树
  - $d=\lceil m/2 \rceil$，且n是T中的元素个数，则：
  - (a) $2d^{h-1}-1 \le n \le m^h-1$
  - (b) $\log_m(n+1) \le h \le \log_d((n+1)/2)+1$
- 证明： $n \le m^h-1$（ m叉搜索树）;

  1层最少1个， 2层最少2个， 3层最少2d个节点，

  4层—h层依次最少有$2d^2$， $2d^3$ …$2d^{h-2}$个节点，

  h+1层最少有$2d^{h-1}$，

  最少外部节点数= $2d^{h-1}$

  $n \ge 2d^{h-1}-1$

- 实际上， B-树的序取决于磁盘块的大小和单个元素的大小。

# So

- 一棵高度为3的200序B-树中至少有19999个元素，而高度为5的200序B-树中至少有$2*10^8-1$个元素。
- 因此，如果使用200序或更高序B-树，即使元素数量再多，树的高度也可以很小。
- 实际上，B-树的序取决于磁盘块的大小和单个元素的大小。
  - 节点小于磁盘块的大小并无好处，这是因为每次磁盘访问只读或写一个块。
  - 节点大于磁盘块的大小会带来多重磁盘访问，每次磁盘访问都伴随一次搜索和时间延迟，因此节点大于磁盘块的大小也是不可取的。

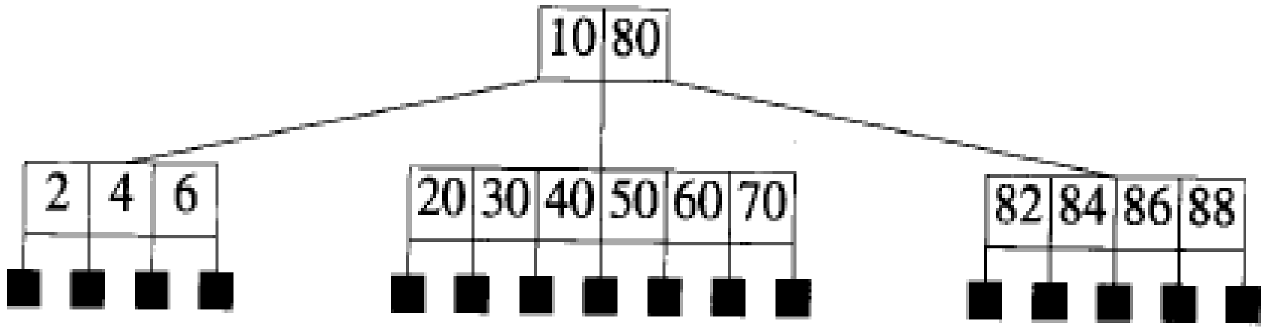
# 15.4.5 B-树的搜索

- B-树的搜索算法与m叉搜索树的搜索算法相同。
- 磁盘访问次数最多是h(h是B-树的高度)。

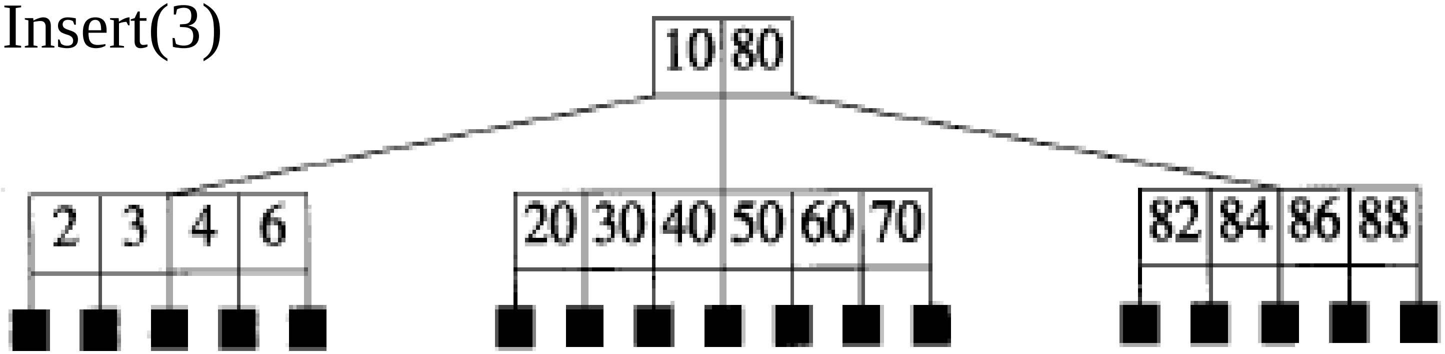
# 15.4.6 B-树的插入

- 将一个元素插入B-树中时
  - 首先要检查具有相同关键值的元素是否存在，如果找到了这个元素，那么插入失败，因为不允许重复值存在。
  - 当搜索不成功时，便可以将元素插入到搜索路径中所遇到的最后一个内部节点处。
  - 饱和？当新元素需要插入到饱和节点中时，饱和节点需要被分开。

# 15.4.6 B-树的插入




Insert(3)




a)

- 对根节点及左孩子有两次磁盘读操作
- 对左孩子另有一次磁盘写操作

# B-树的插入-饱和

- 设P是饱和节点，现将带有空指针的新元素e插入到P中，得到一个有m个元素和m+1个孩子的溢出节点。
- 用下面的序列表示溢出节点（ei是元素，ci是孩子指针）：

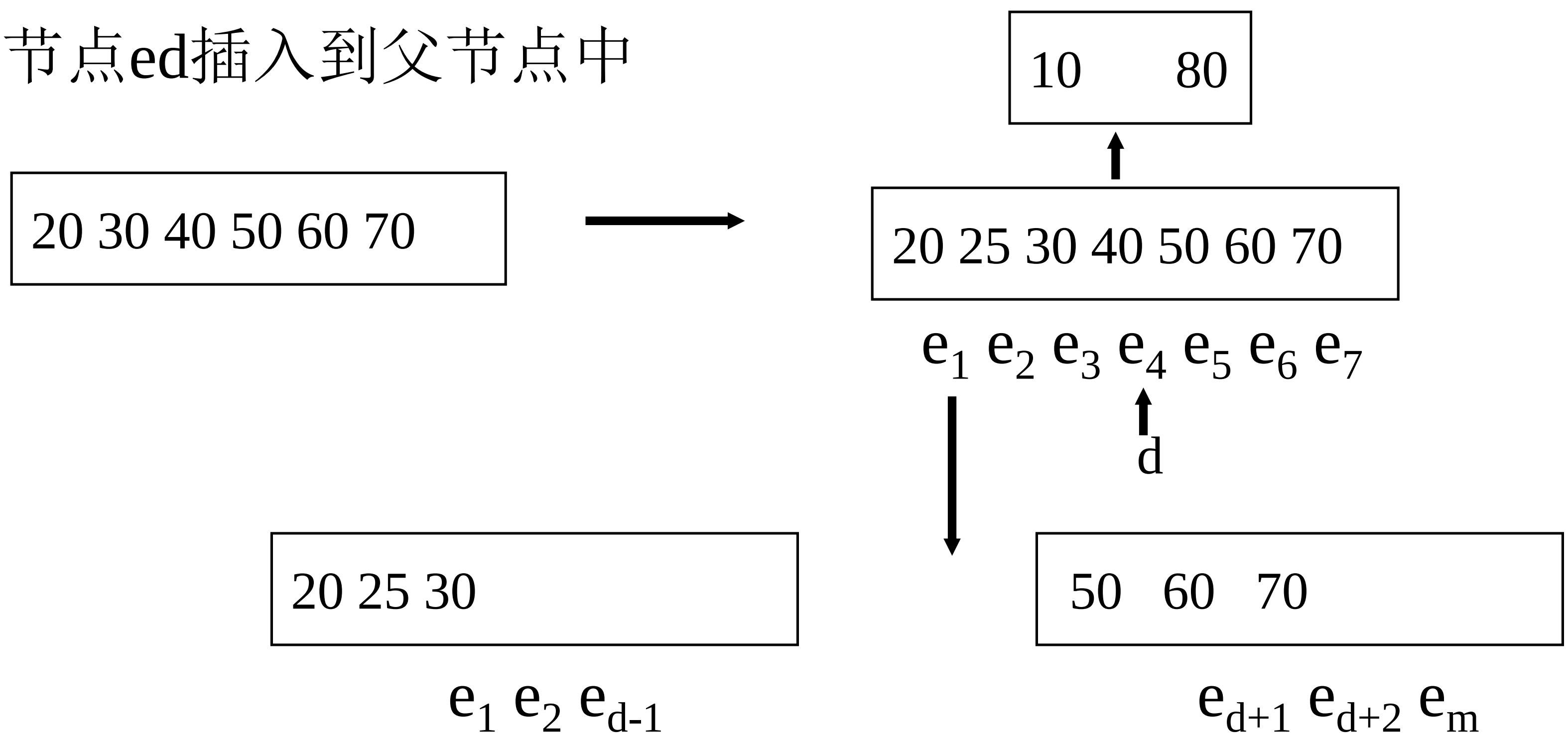
  m，c0，（e1, c1），. . . ，（em, cm）
- 从ed处分开此节点，其中d=⌈m/2⌉。
- 左边的元素保留在P中，右边的元素移到新节点Q中，（ed, Q）被插入到P的父节点中。
- 新的P和Q的格式为：
- P：d-1，c0，（e1, c1），. . . ，（ed-1, cd-1）
- Q：m-d，cd，（ed+1, cd+1），. . . ，（em, cm）
- 注意P和Q的孩子数量至少是d。

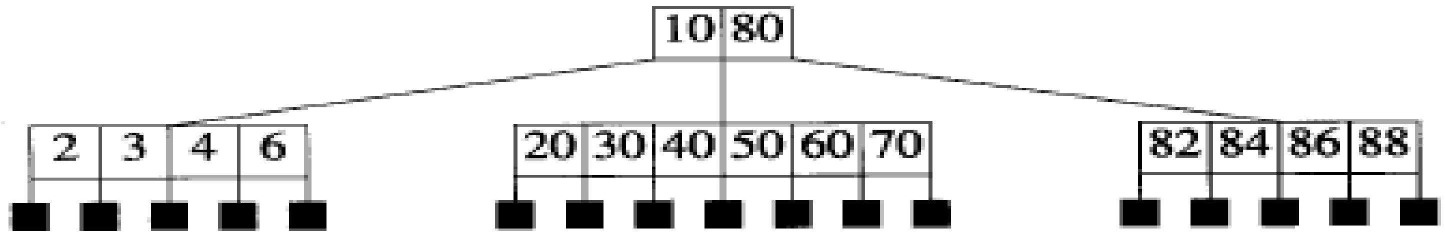
- Insert(25)

- 当新元素需要插入到饱和节点中时，饱和节点需要被分开。
  - 7，0，(20,0)，(25,0)，(30,0)，(40,0)，(50,0)，(60,0)，(70,0)
- 节点从$e_d$处分开此节点， $d = \lceil m/2 \rceil = 4$

  P：3，0，(20,0)，(25,0)，(30,0)

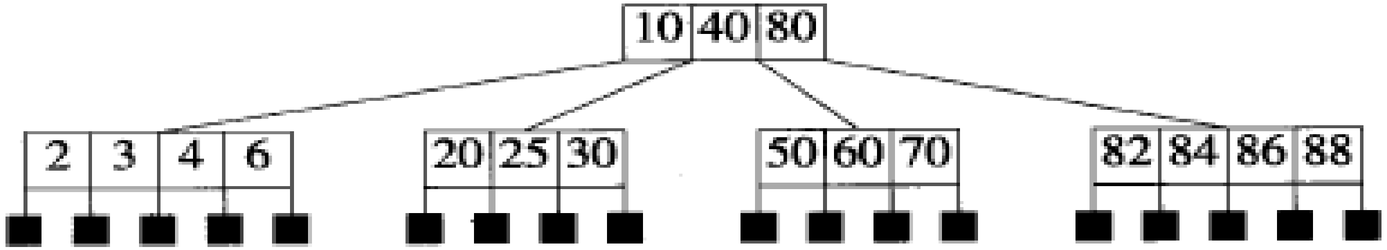
  Q：3，0，(50,0)，(60,0)，(70,0)
- 节点ed插入到父节点中






a)

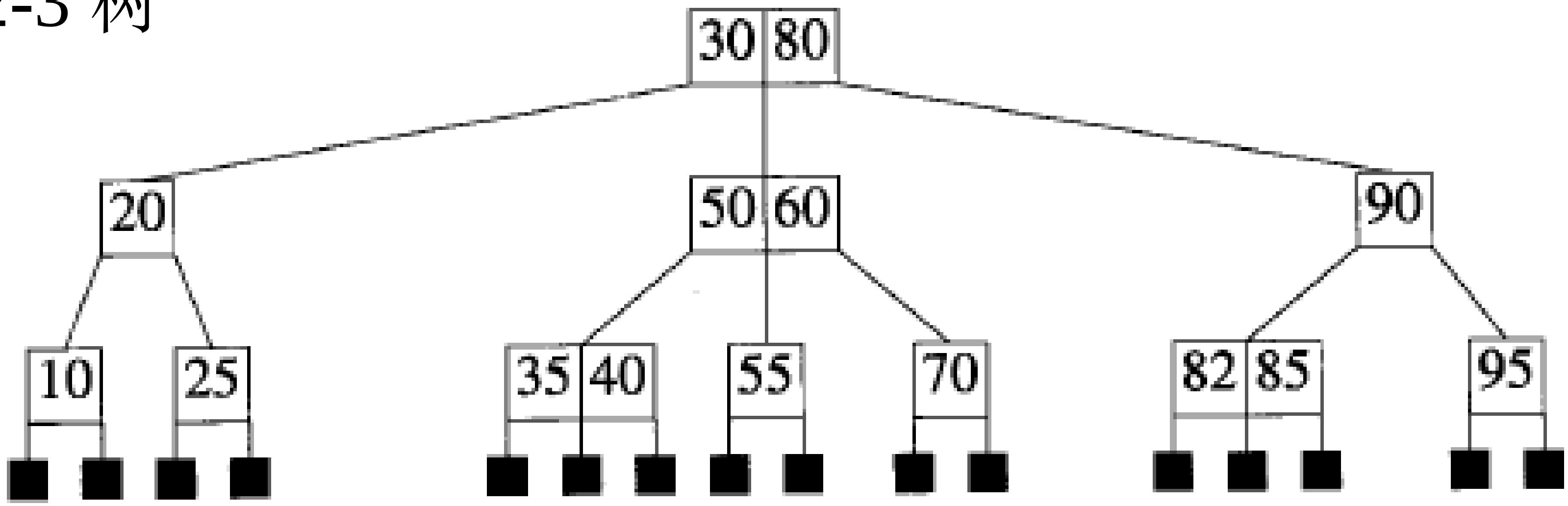
•Insert(25)：当新元素需要插入到饱和节点中时，饱和节点需要被分开。节点从$e_d$处分开此节点， $d = \lceil m/2 \rceil$



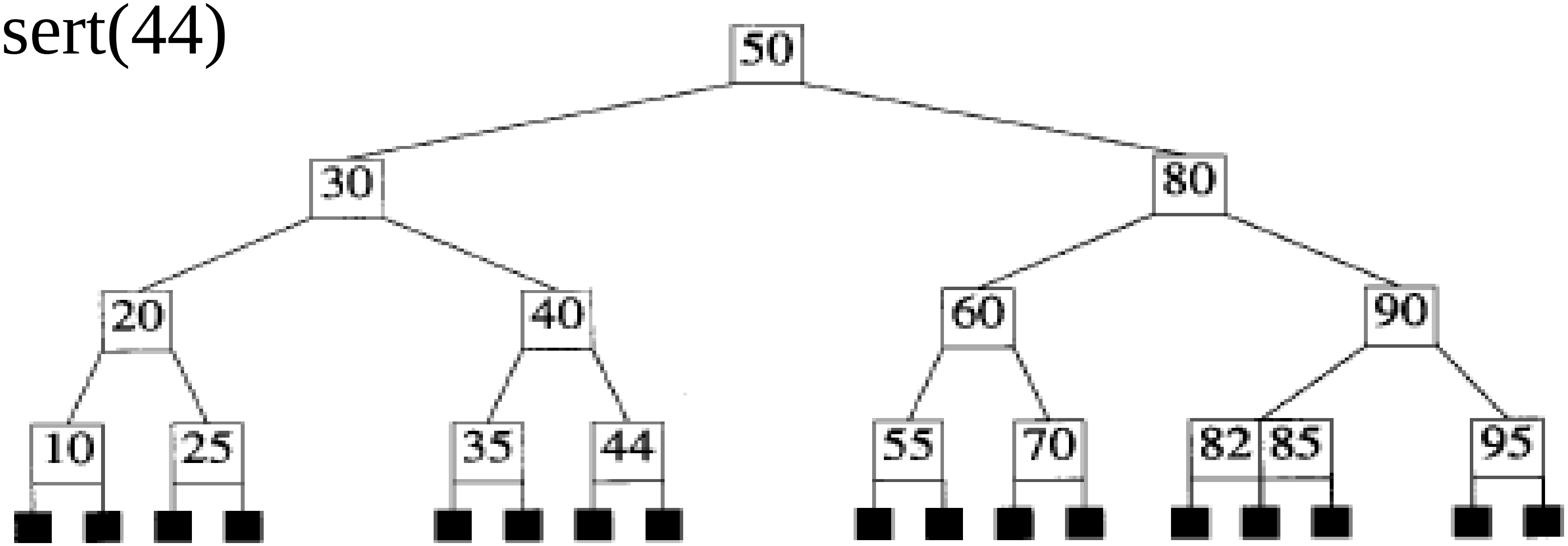

b)

- 需要从磁盘中得到根节点及其中间孩子
- 写回分开的两个节点
- 写回修改后的根节点
- 磁盘访问次数一共是5次。

# 2-3 树




## Insert(44)




c)

# 磁盘访问的总次数

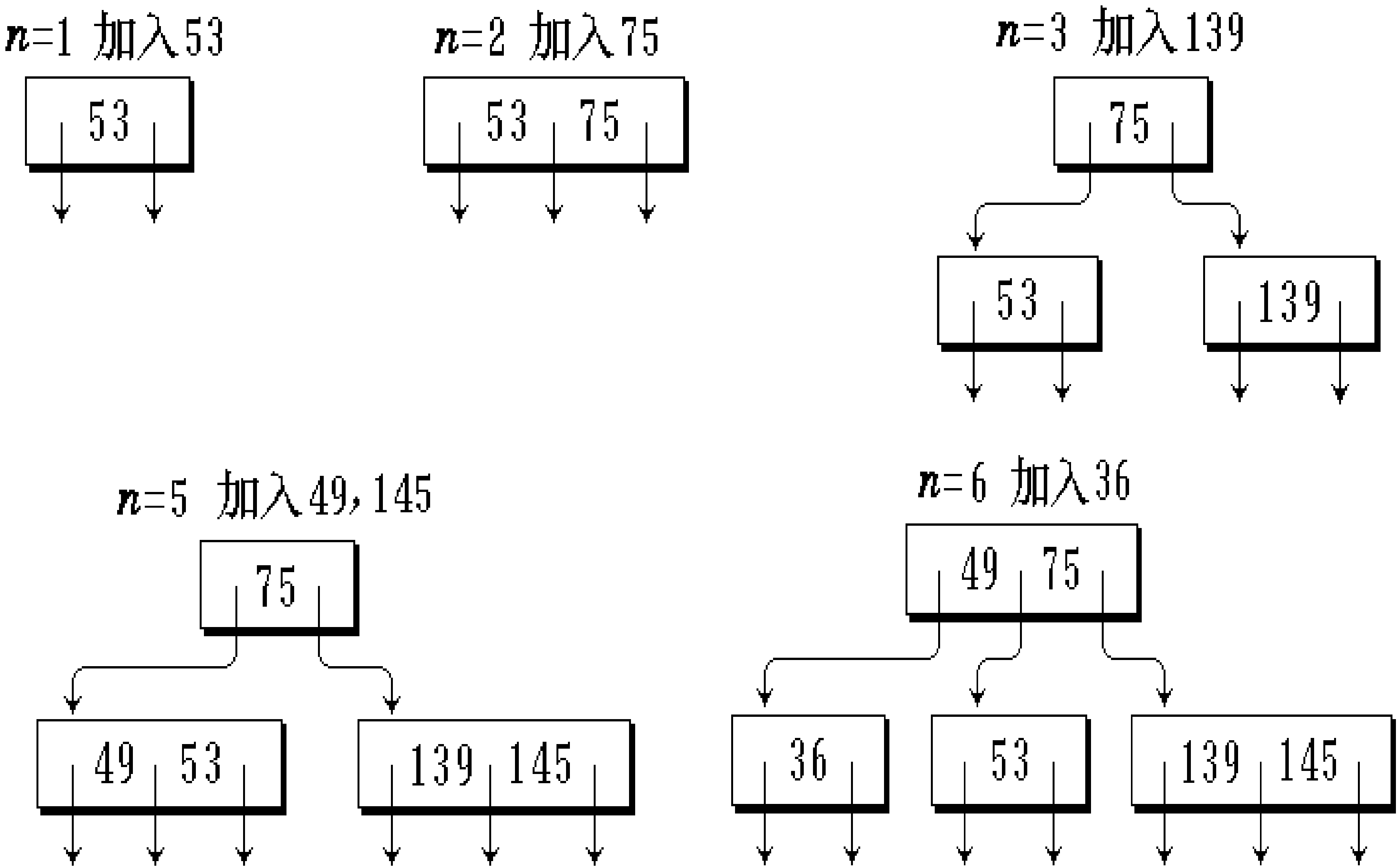
- Insert(44):
  - 搜索 44： 3
  - 3个节点被分开( split): 6 (每个节点被分开：2次写操作 )
  - 产生一个新的根节点并写回磁盘： 1
  - 磁盘访问的总次数： 10
- 假设:
  - B-树的高度: h
  - s个节点分裂

⇒ 磁盘访问次数

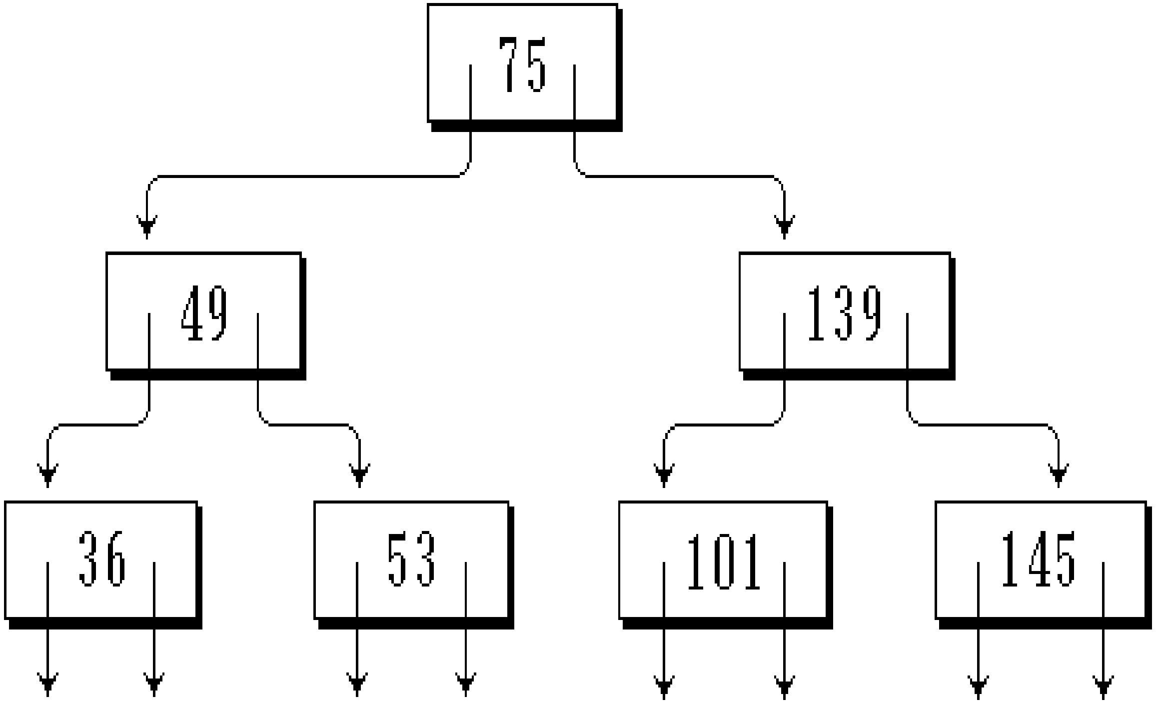
=h+2s+1 //1:回写新的根节点或插入后没有导致分裂的节点

最多: 3h+1

# 示例：从空树开始逐个加入关键码建立3阶B_树



$n$=7 加入101



# 15.4.7 B-树的删除

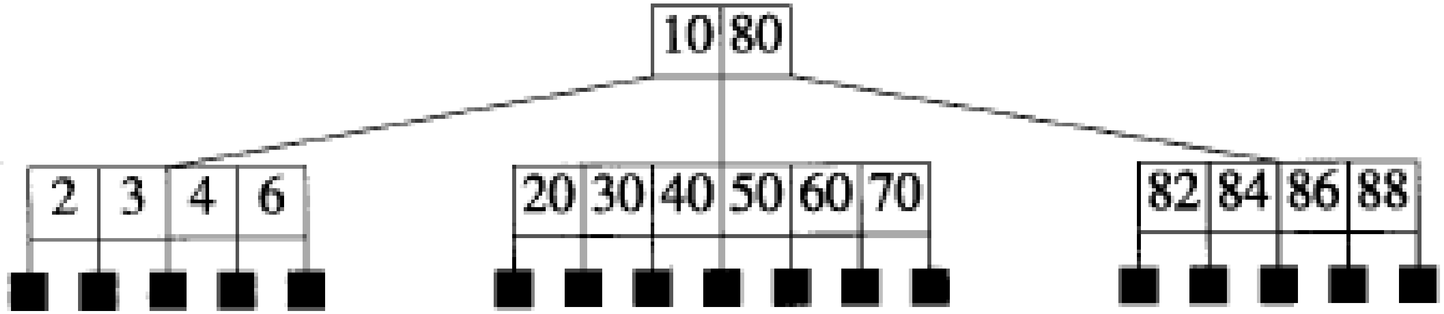
- 删除分为两种情况：

1. 被删除元素位于其孩子均为外部节点的节点中(即元素在树叶中)。
2. 被删除元素在非树叶节点中。既可以用左相邻子树中的最大元素，也可以用右相邻子树中的最小元素来替换被删除元素，这样2就转化为1 。

# B-树的删除

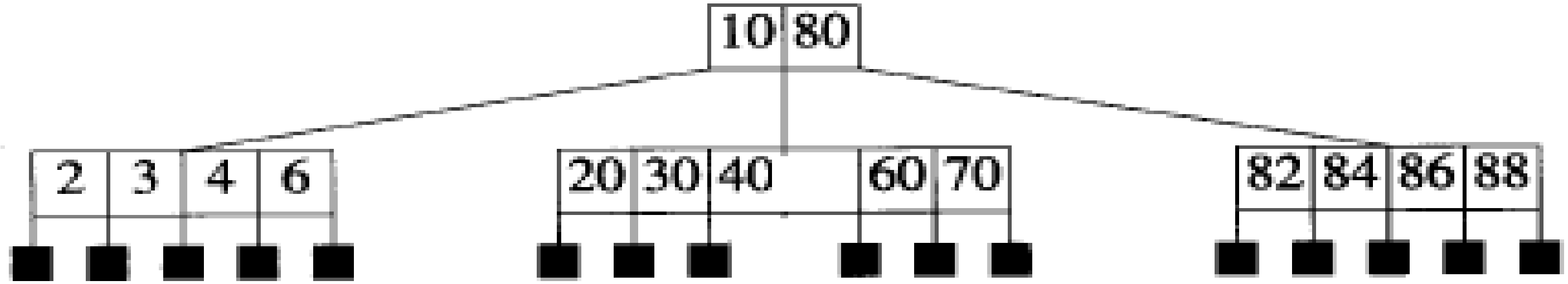
- 第1种情况:

1. 从一个包含多于最少数目元素*(如果树叶同时是根节点，那么最少元素数目是1，如果不是根节点，则为⌈m/2⌉-1 )*的树叶中删除一个元素，只需要将修改后的节点写回。




a)

Delete(50)



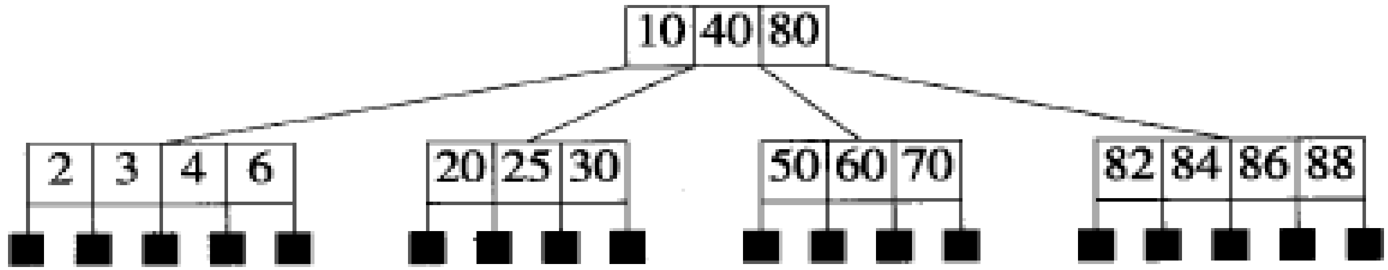

- 磁盘访问次数是2(从根到包含50的树叶)+1(写回修改后的树叶)=3。

# B-树的删除

- 第1种情况:

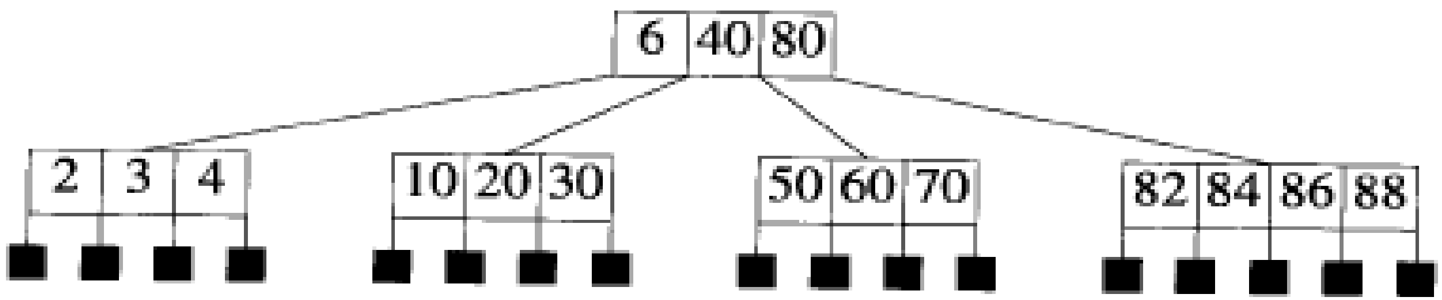
2. 当被删除元素在一个非根节点中且该节点中的元素数量为最小值时，

    (1) 它的最相邻的兄弟(最相邻的左或右兄弟)有多于最少数目元素，用其最相邻的左或右兄弟中的元素来替换它。




b)

Delete(25)：从左相邻兄弟借6



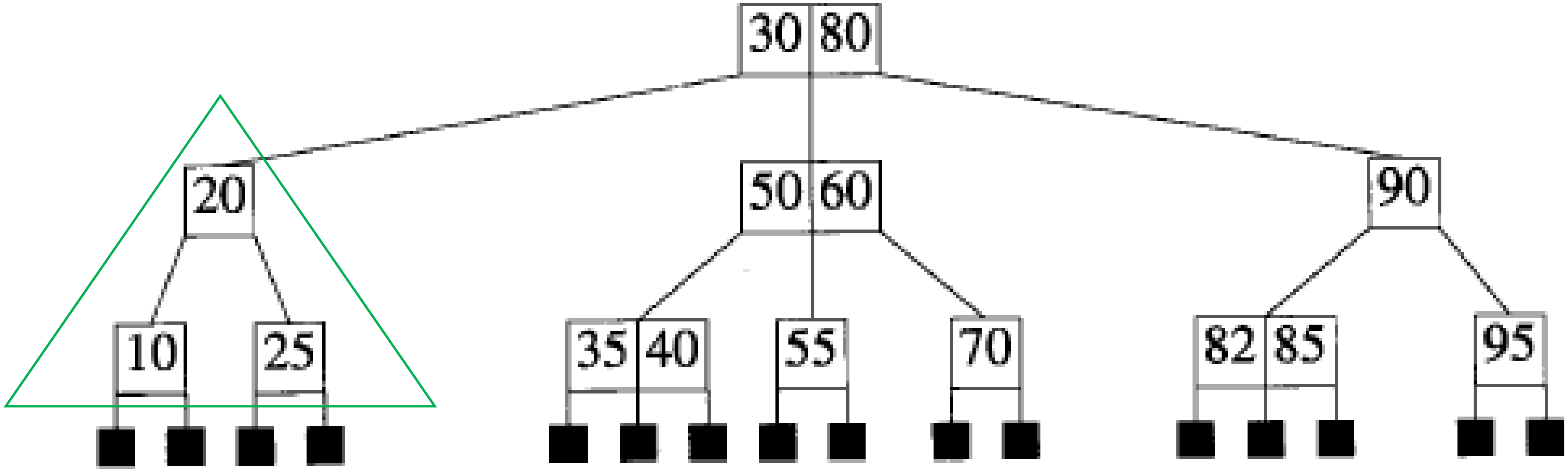

- 磁盘访问次数是2（从根到包含25的树叶）+1（读取该树叶的最相邻左兄弟）+3（写回修改后的树叶、兄弟和父节点）=6。

# B-树的删除

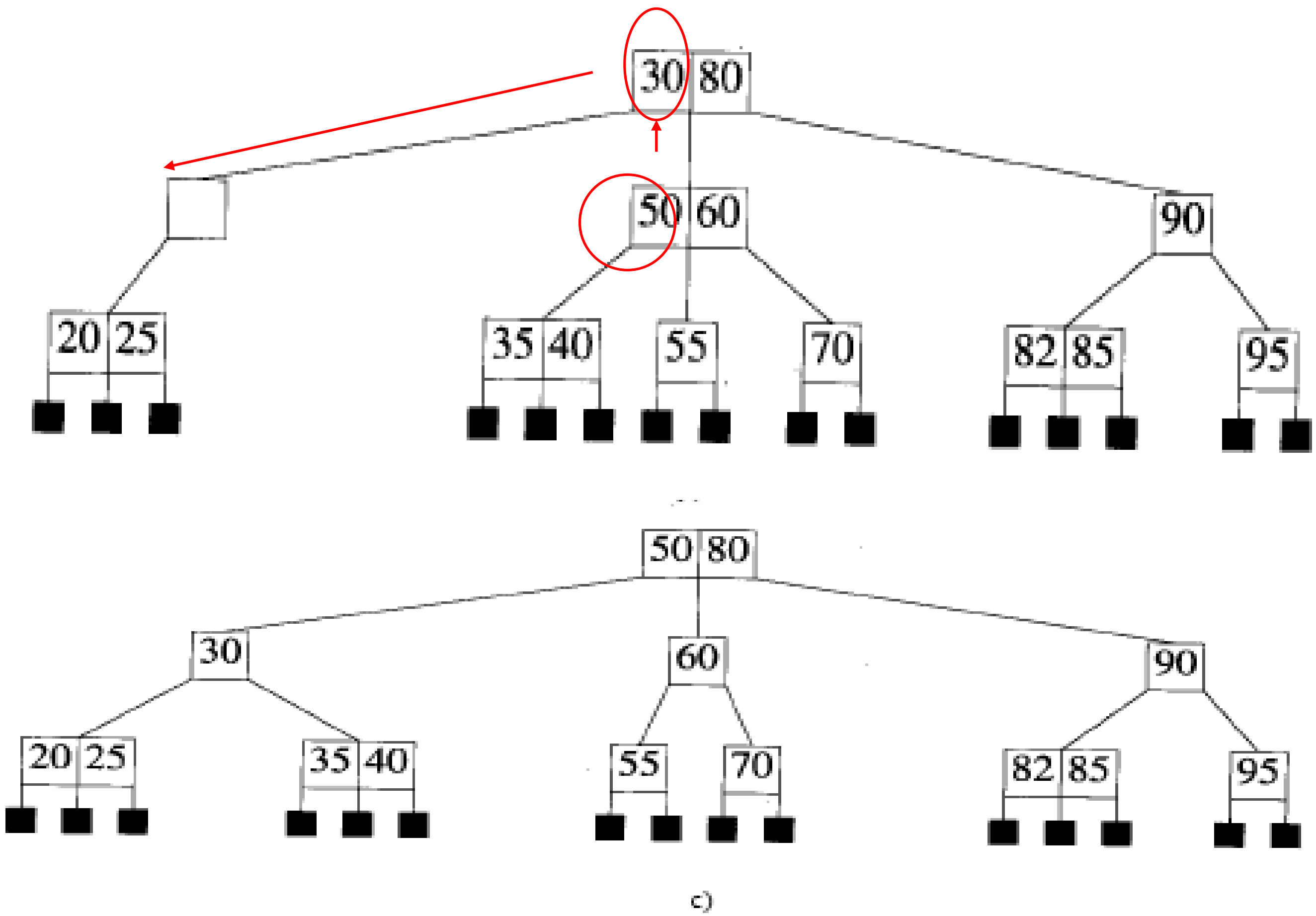
- 第1种情况:

2. 当被删除元素在一个非根节点中且该节点中的元素数量为最小值时，

   (2) 它的最相邻的兄弟(最相邻的左或右兄弟)有最少数目元素，将两个兄弟与父节点中介于两个兄弟之间的元素合并成一个节点。删除父节点中的相应元素。




Delete(10) （10），（25）和20合并

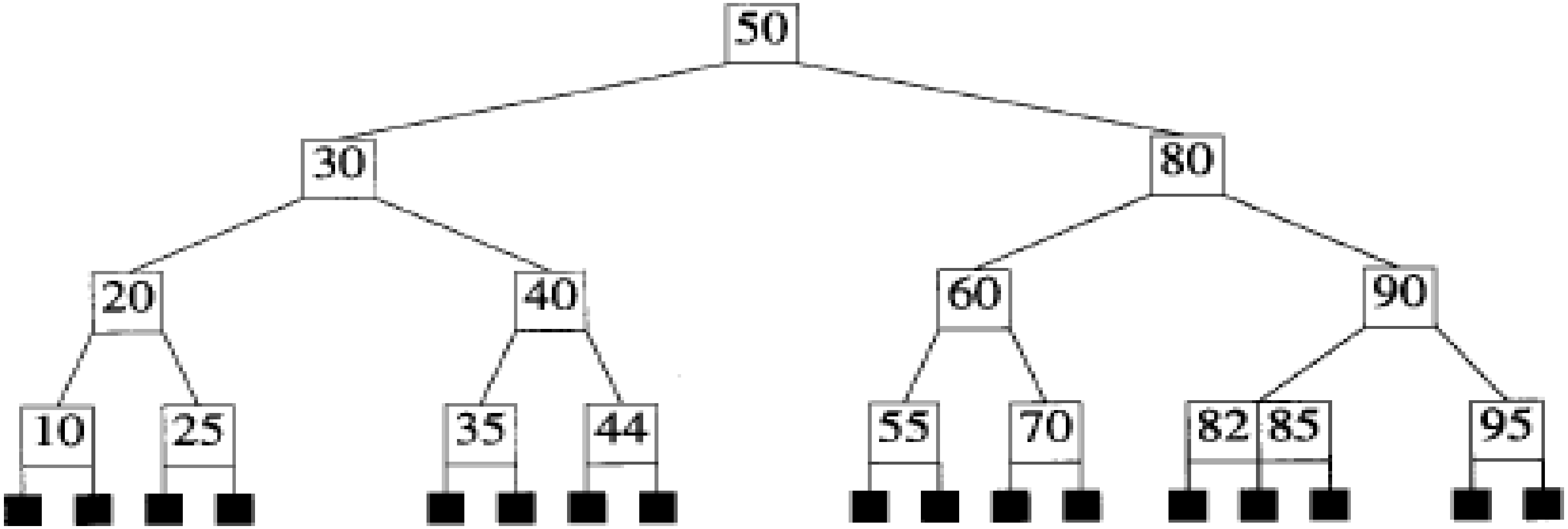



磁盘访问次数是3（到达包含被删除元素的树叶）+2（读第二和三层的最相邻右兄弟节点）+4（将第一，二和三层的4个修改后的节点写回磁盘）

因此总的磁盘访问次数是9次。

# B-树的删除

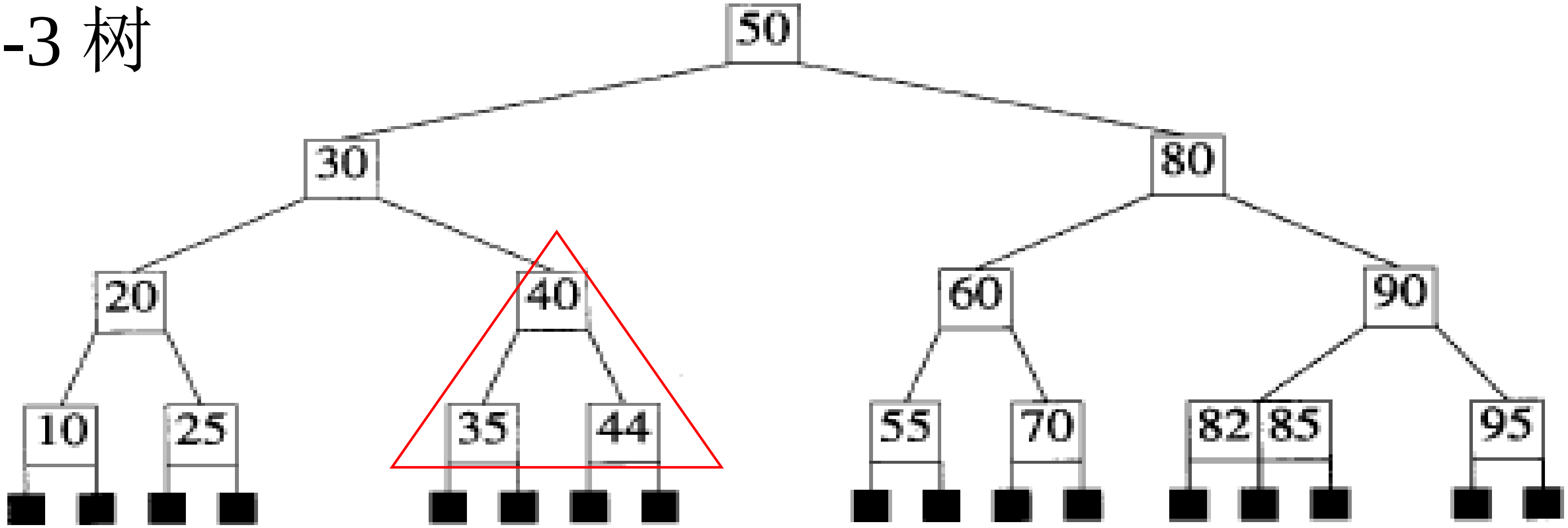
- 最坏情况下，这种过程会一直回溯到根节点。当根节点缺少一个元素时，它变成空节点，将被抛弃，树的高度减1。




c)

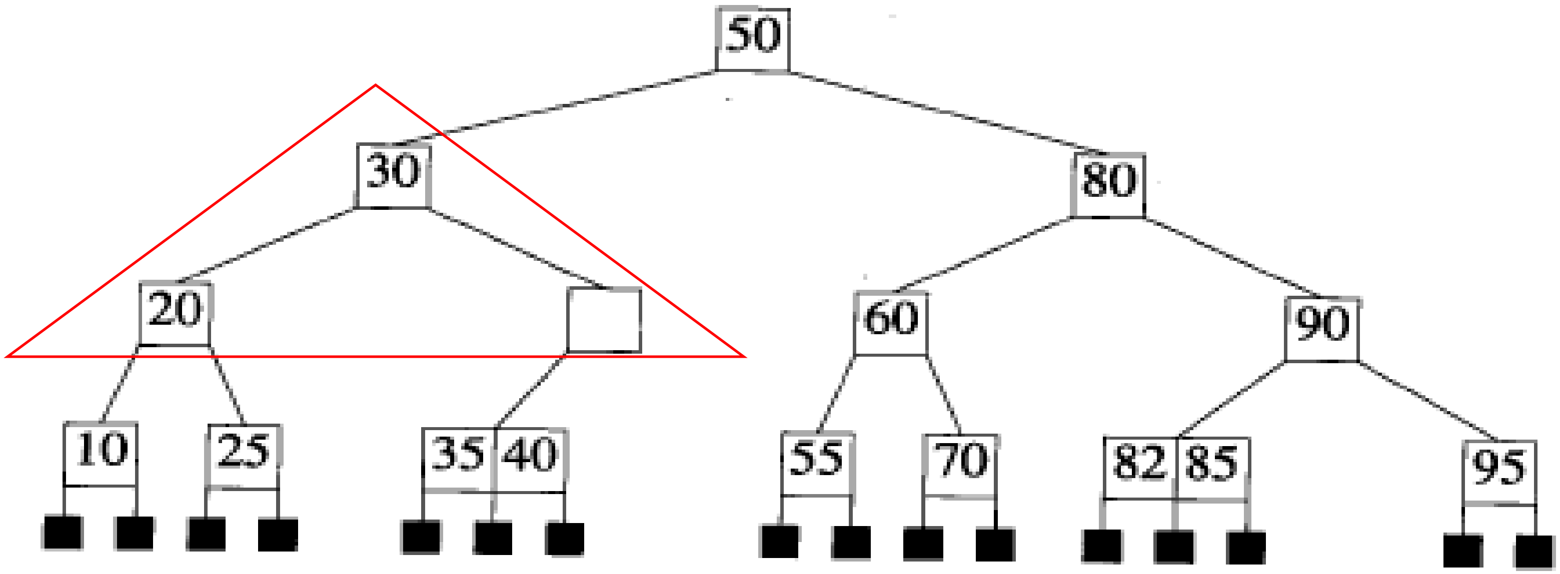
Delete(44)

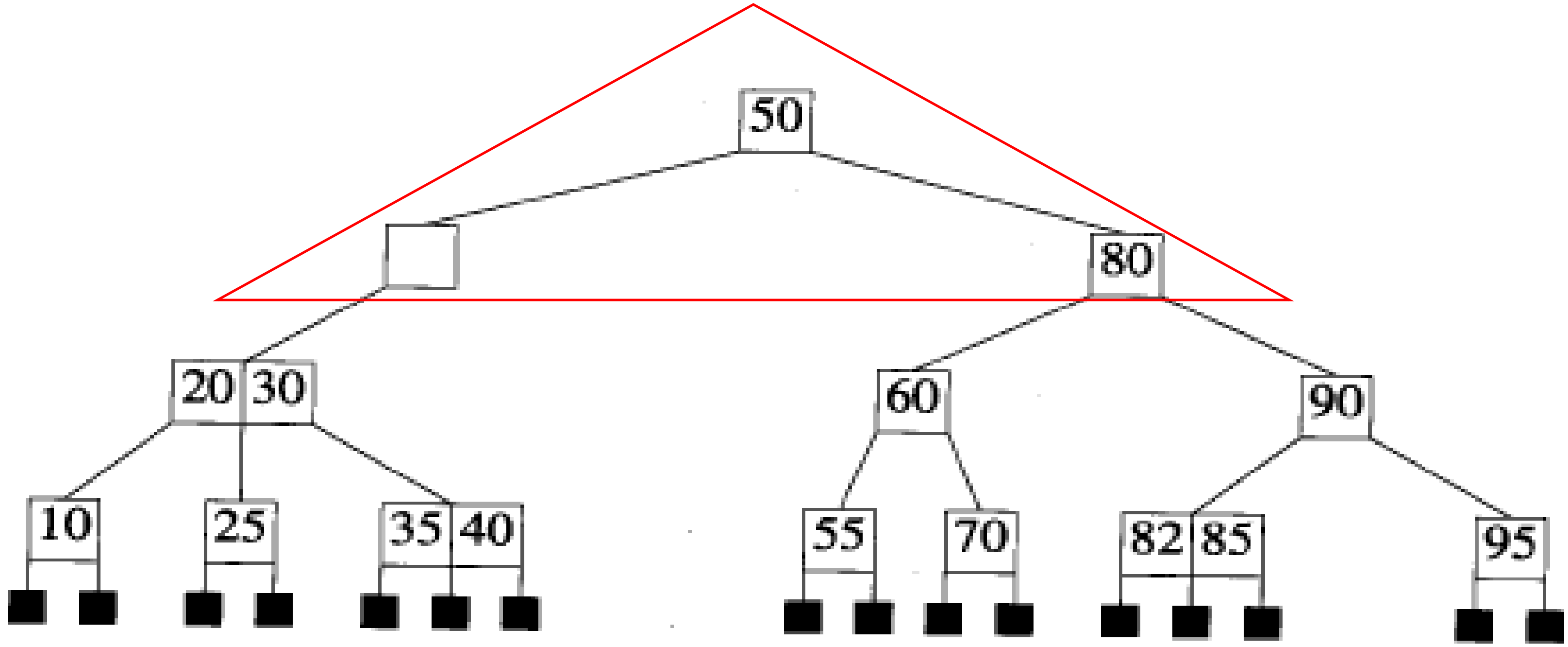
# 2-3 树

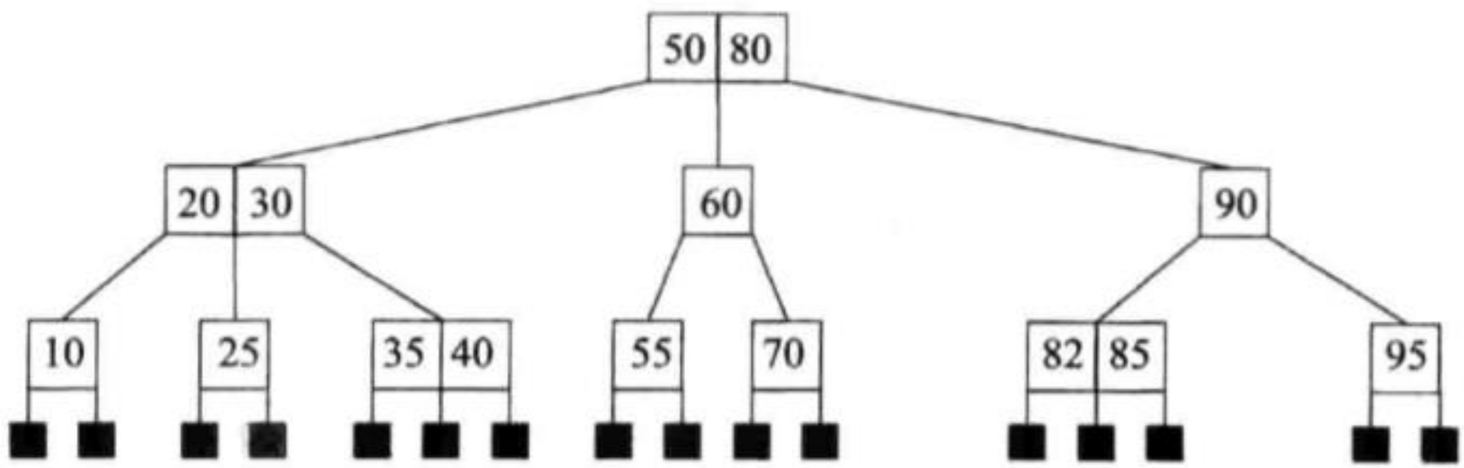



c)

Delete(44)




50
80
20 30
60
90
10
25
35 40
55
70
82 85
95


50 80
20 30
60
90
10
25
35 40
55
70
82 85
95

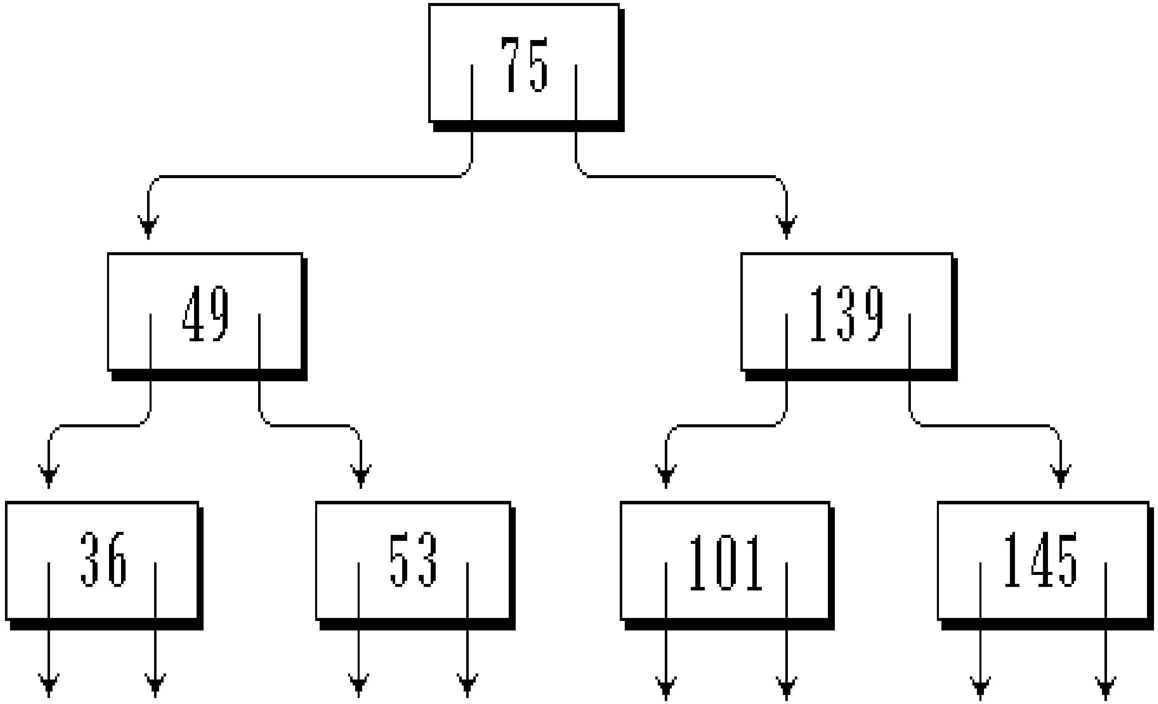
# 磁盘访问的总次数

- Delete (44):
  - 搜索 44： 4
  - 读取最相邻的兄弟： 3
  - 修改后的节点写回： 3((35,40),(20,30,),(50,80))
  - 磁盘访问的总次数: 10

# 磁盘访问的总次数

- 对于高度为h的B-树的删除操作的最坏情况：
  - 当合并发生在h,h- 1 , . . . , 3 层
  - 在2层时 ,需要从最相邻兄弟中获取一个元素。
- 最坏情况下磁盘访问次数是3h：
  - 找到包含被删除元素的节点:　h次读访问
  - 获取第2至h 层的最相邻兄弟:　h-1次读访问
  - 在第3至h层的合并：h-2次写访问
  - 对修改过的根节点和第2层的两个节点：3次写访问。

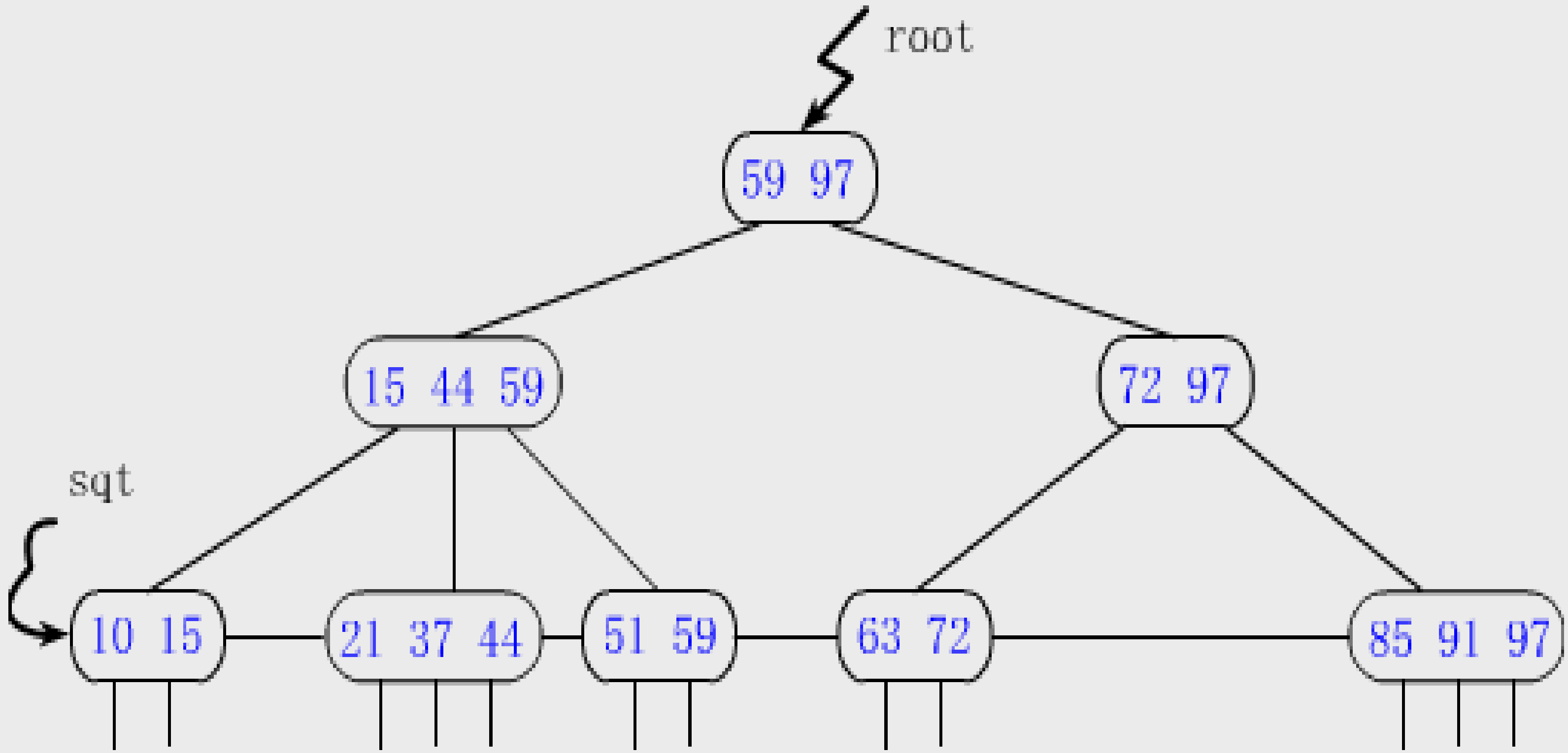
# 依次删除53，75，139，49，145，36，101



# B+树

- B+树是一种常用于文件组织的B-树的变形树。一棵m阶的B＋树和B-树的差异在于：
  1. 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点的本身依关键字的大小从小到大顺序链接。
  2. 所有的非终端结点可以看成是索引部分，结点中仅含有其子树（根结点）中最大（或最小）关键字。
- 通常在B＋树上有两个头指针，一个指向根结点，另一个指向关键字最小的叶子结点。

# B+树




一棵3阶的B+树

# B+树的搜索

- 对可进行两种查找运算：一种是从最小关键字起进行顺序查找；另一种是从根结点开始进行随机查找。
- 在查找时，若非叶结点上的关键字等于给定值，并不终止，而是继续向下直到叶子结点。因此，在B+树中，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径.

# B+树的插入

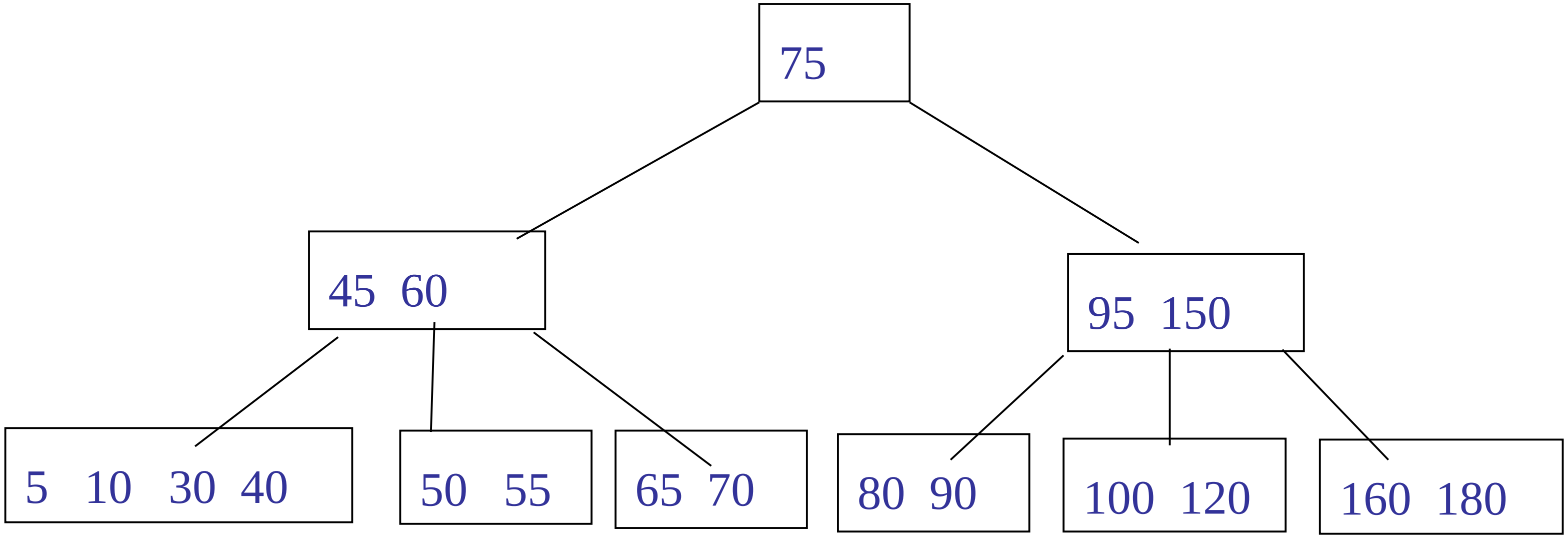
- B+树的插入仅在叶子结点上进行，当结点中的关键字个数大于m时要分裂成两个结点，它们所含关键字的个数分别为：
- $\lceil (m+1)/2 \rceil$和$\lceil (m+1)/2 \rceil$
- 并且它们的双亲结点中应同时包含这两个结点的最大关键字。

# B+树的删除

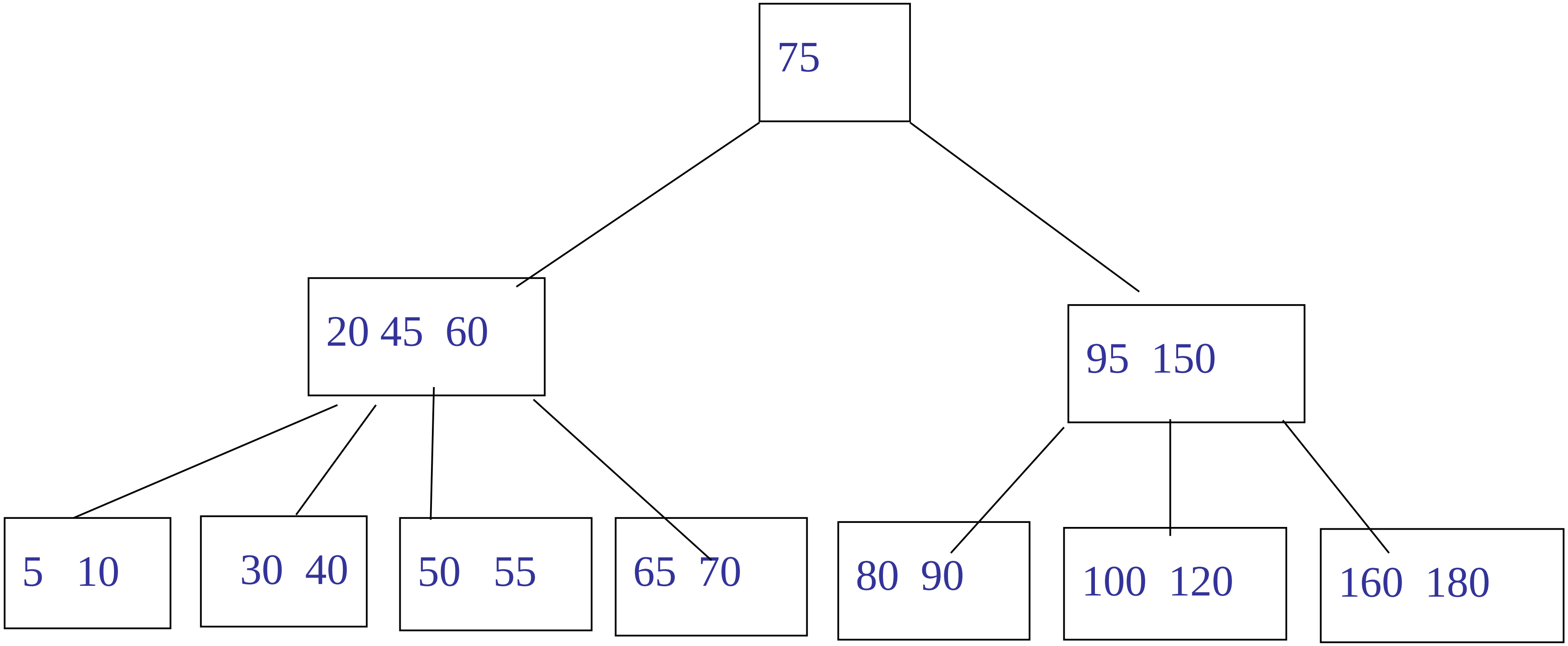
- B+树的删除仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个"分界关键字"存在。若因删除而使结点中关键字的个数少于$\lceil m/2 \rceil$时，则可能要和该结点的兄弟结点合并，合并过程和B-树类似。

# 作业：

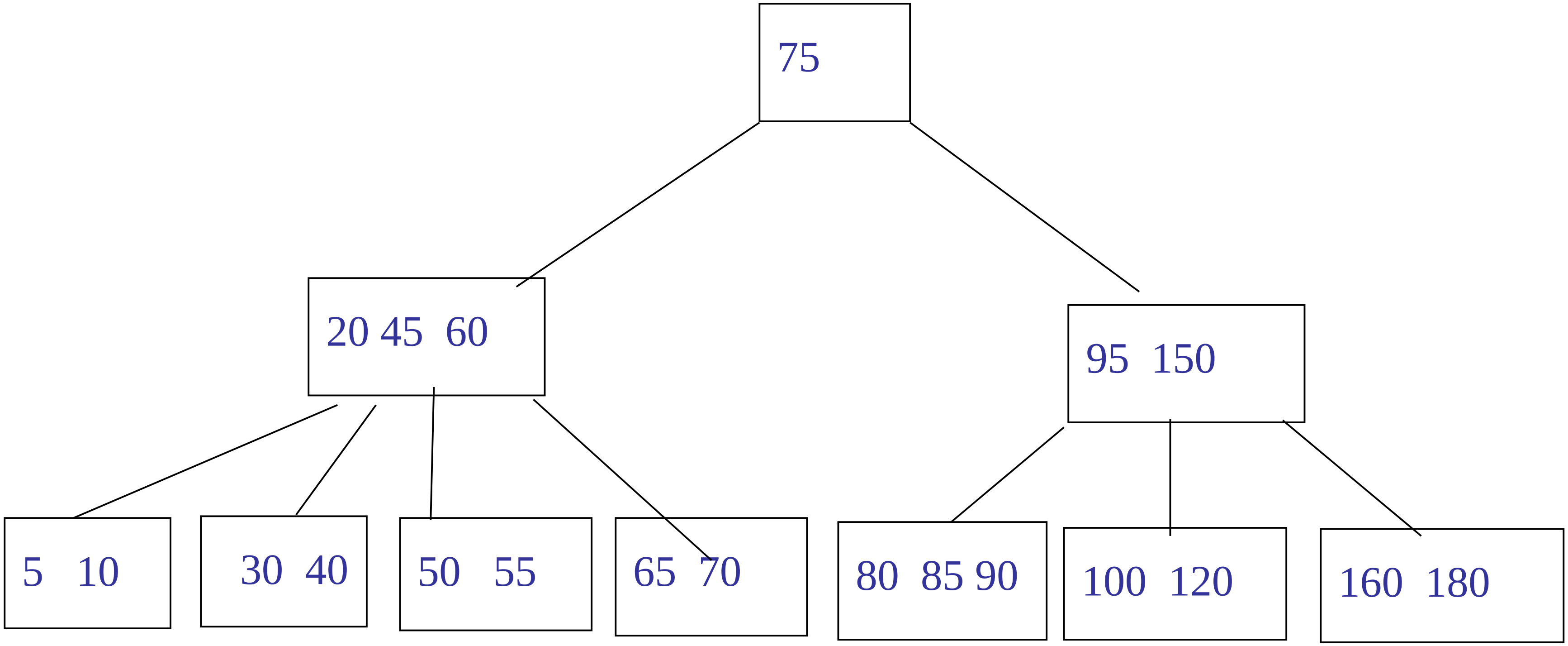
- 62
- 在下列5阶B-树中首先依次插入关键字20，85，然后依次删除关键字120，70，30，画出每次插入或删除元素后的B-树。



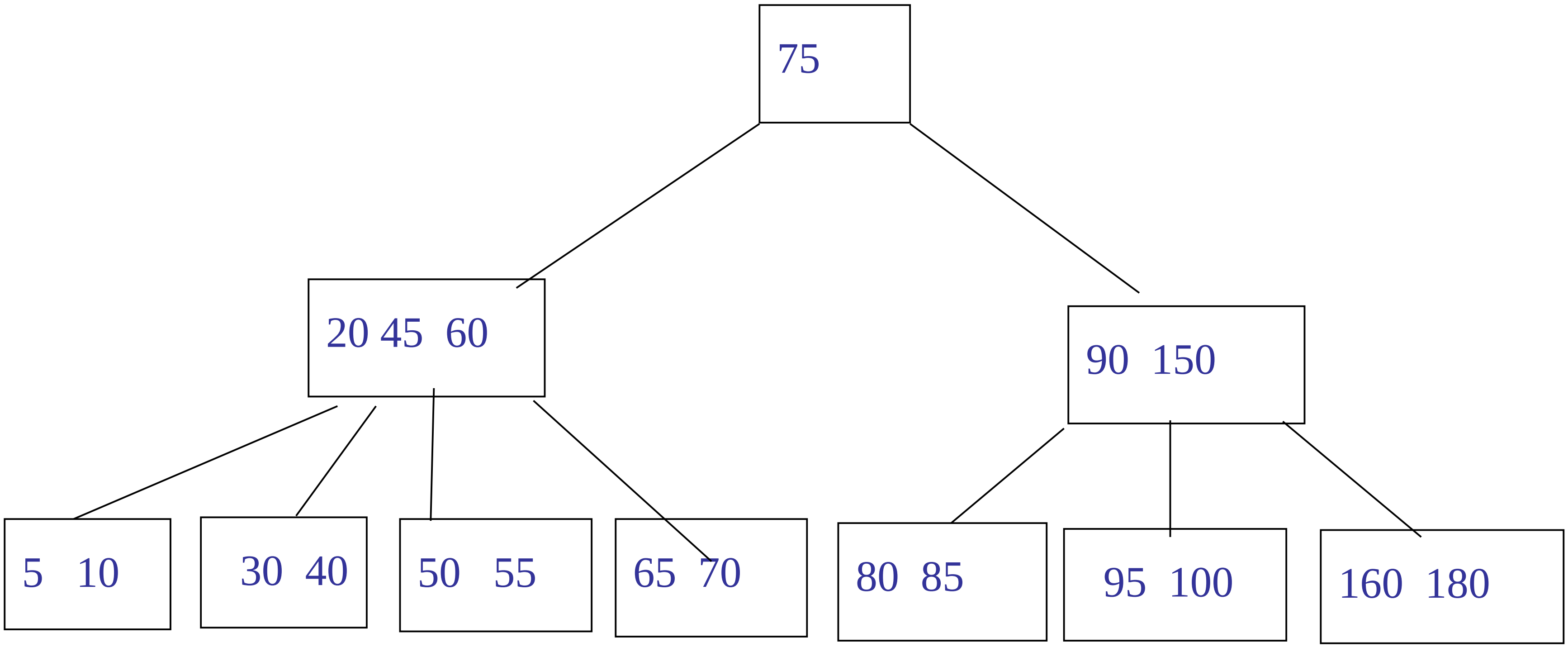
- 插入20



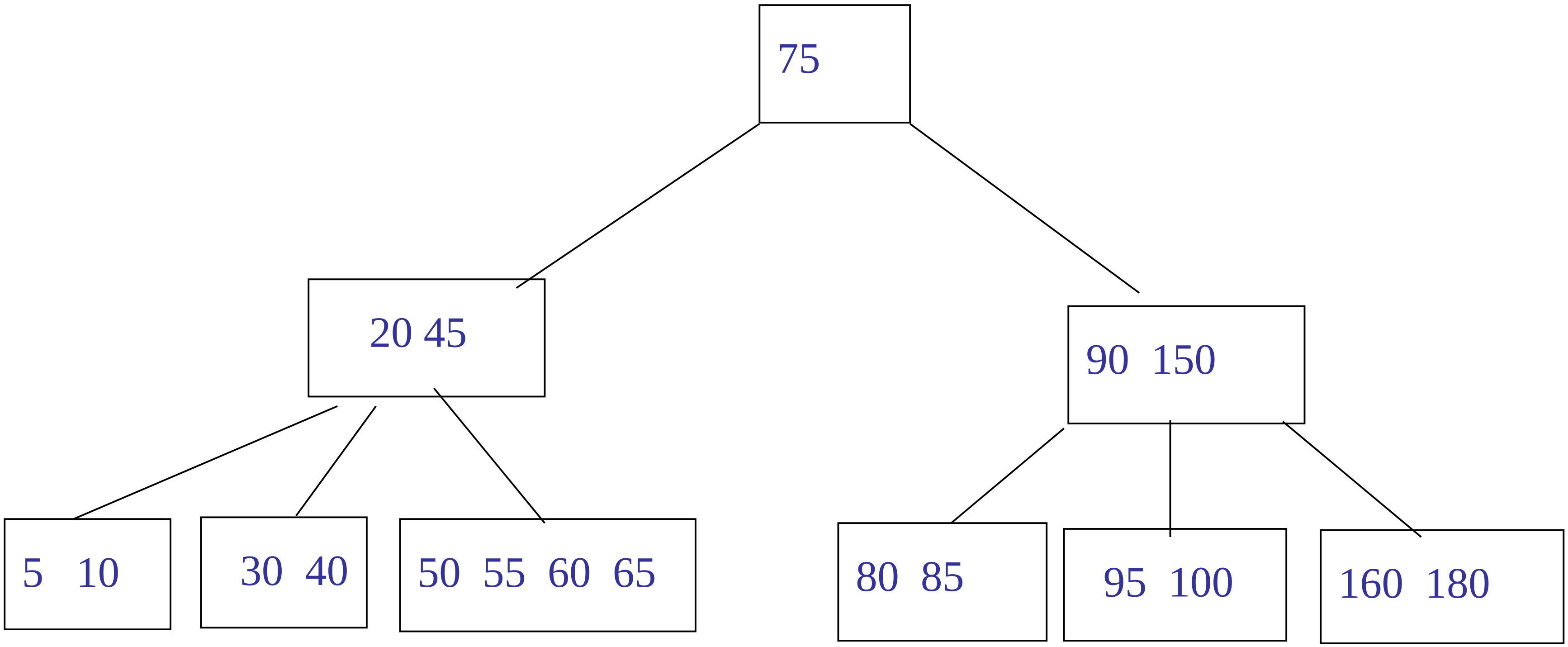
- 插入85



- 删除120



- 删除70



- 删除30