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**

AVL 树

- 平衡树：最坏情况下的高度为O(logn)的树。如果当搜索树的高度总是O(logn)时，能够保证每个搜索树操作所占用的时间为O(logn)。
- AVL(Adelson-Velsky和Landis1962年提出)树——一种平衡树。

AVL树

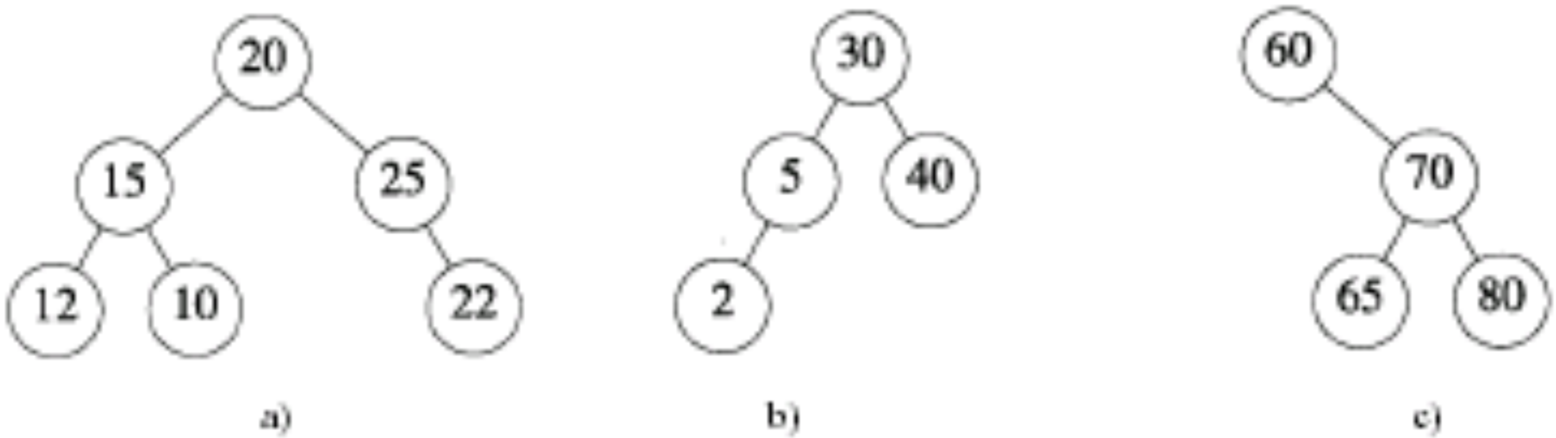
AVL树定义：

- 空二叉树是AVL树。
- 如果T是一棵非空的二叉树， T_L 和T_R分别是其左子树右子树，当T满足以下条件时， T是一棵AVL树。

1. **T_L 和 T_R是AVL树,**
2. **$|h_L - h_R| \le 1$ ， h_L和h_R**分别是左子树和右子树的高度。

AVL搜索树

- **AVL搜索树**（平衡二叉搜索树/平衡二叉排序树）：既是二叉搜索树，也是**AVL树** 。

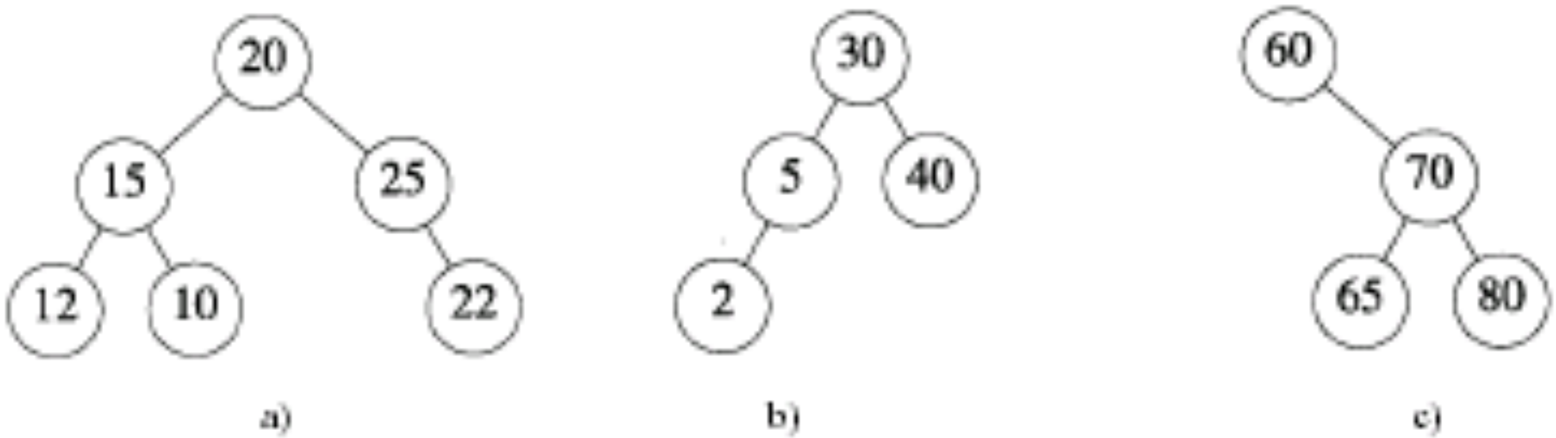


- AVL 树?

- AVL搜索树?

AVL搜索树

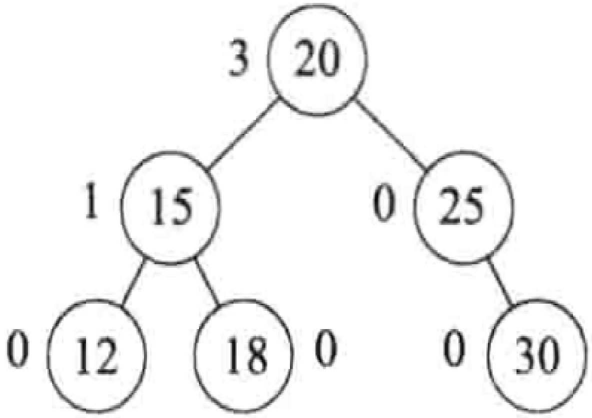
- **AVL搜索树**（平衡二叉搜索树/平衡二叉排序树）：既是二叉搜索树，也是**AVL树** 。



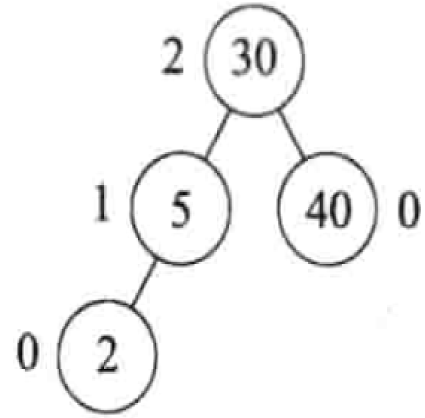
- AVL 树?
  - (a) 、(b)
- AVL搜索树?
  - (b)

带索引的AVL搜索树

- 带索引的**AVL**搜索树既是带索引的二叉搜索树，也是**AVL**树。



a)

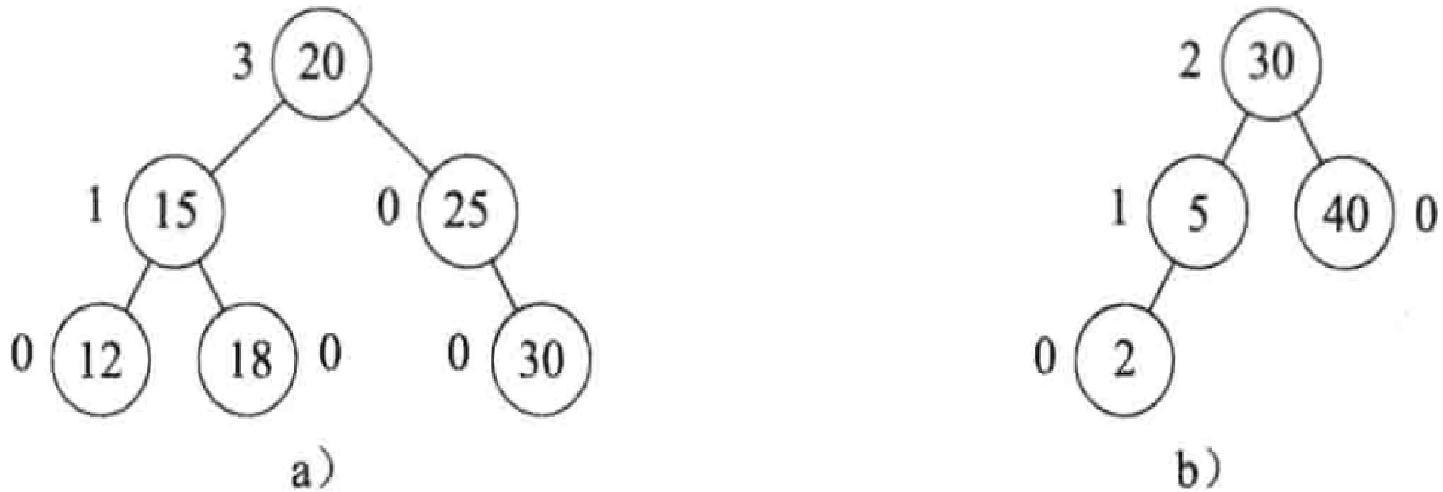


b)

•带索引的AVL搜索树?

带索引的AVL搜索树

- 带索引的**AVL**搜索树既是带索引的二叉搜索树，也是**AVL**树。



•带索引的AVL搜索树?

–(a) 、 (b)

AVL 树的特征

1. n 个元素(节点)的AVL树的高度是O(logn)。
2. 对于每一个n(n≥0)值，都存在一棵AVL树。(否则，在插入完成后，一棵AVL树将不是AVL树，因为对当前元素数来说不存在对应的AVL树)。
3. 一棵n元素的AVL搜索树能在O(高度)= O(logn)的时间内完成搜索。
4. 将一个新元素插入到一棵n元素的AVL搜索树中，可得到一棵n+1元素的AVL树，这种插入过程可以在O(logn)时间内完成。
5. 从一棵n元素的AVL搜索树中删除一个元素，可得到一棵n-1元素的AVL树，这种删除过程可以在O(logn)时间内完成。

- 特征2可以从特征4推出。

AVL 树的高度

- 一棵有n个节点的AVL树的高度至多：
  - $1.44 \log_2 (n+2)$.
- 一棵有n个节点的AVL树的高度至少：
  - $\log_2 (n+1)$.
- N_h：高度为h的AVL树中的最小节点数（画图，从高度为1开始画）。
- $N_h = N_{h-1} + N_{h-2} + 1$，$N_0 = 0$, $N_1 = 1$
- Fibonacci number(斐波那契数列):
  - $F_h = F_{h-1} + F_{h-2}$，$F_0 = 0$, $F_1 = 1$

F_h 和 N_h之间的关系

- $N_h = F_{h+2} - 1 \quad h>=0$
- 归纳证明:
- 因为 $F_3 = F_2 + F_1 = 2F_1+F_0=2$
- 所以$N_1= 1$, $h=1$ 成立;
- 假设 $h< m$ 成立，证明$h=m$时，成立;
- $N_m = N_{m-1} + N_{m-2}+1$
- $= (F_{m+1} -1)+ (F_m-1) +1$ (归纳假设）
- $= (F_{m+1} +F_m) -1$
- $= F_{m+2} -1$

F_h 和 N_h之间的关系

- 按照斐波那契定理可知

$$F_h = \phi^h / \sqrt{5}, h \ge 0, \text{其中}, \phi = (1+\sqrt{5})/2$$

$$N_h = F_{h+2} - 1, h \ge 0$$

$$N_h = \phi^{h+2} / \sqrt{5} - 1$$

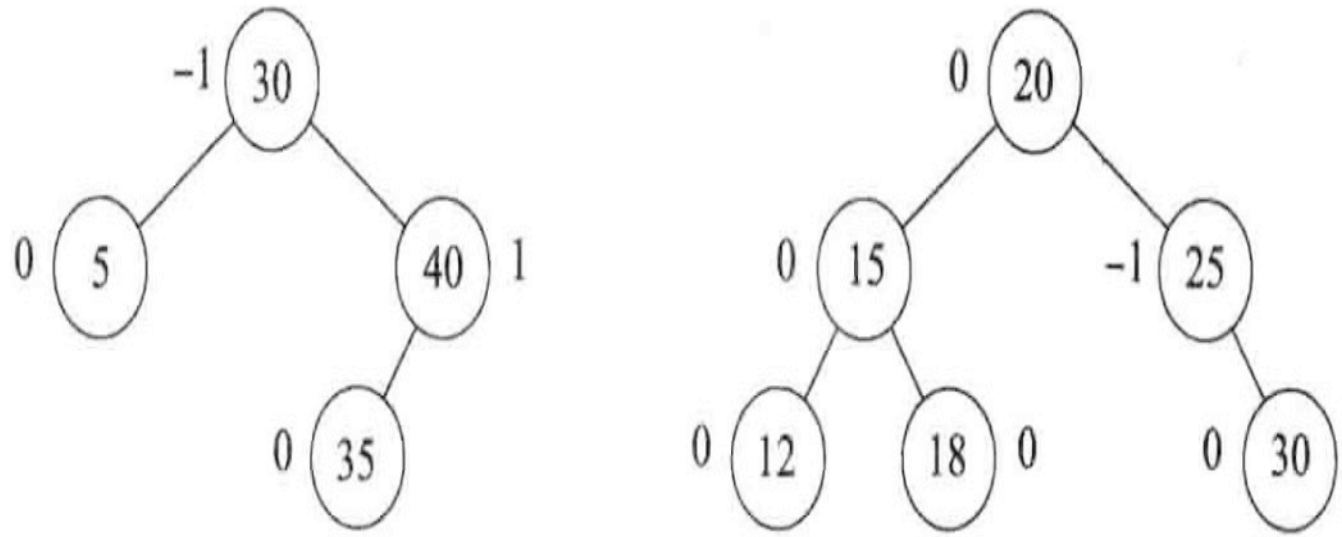
$$n \ge \phi^{h+2} / \sqrt{5} - 1$$

$$h = \log_\phi(\sqrt{5}(n+1)) - 2 \approx 1.44 \log_2(n+2) = O(\log n)$$

AVL树的描述

- 一般用链表方式来描述
  - 要为每个节点增加一个平衡因子
- **平衡因子(Balance Factor)**：
  - 节点x的平衡因子**bf(x)** 定义为:
    x的左子树的高度 - x的右子树的高度
  - AVL树平衡因子的可能取值为：–1, 0, 和 1.

具有平衡因子的AVL 树

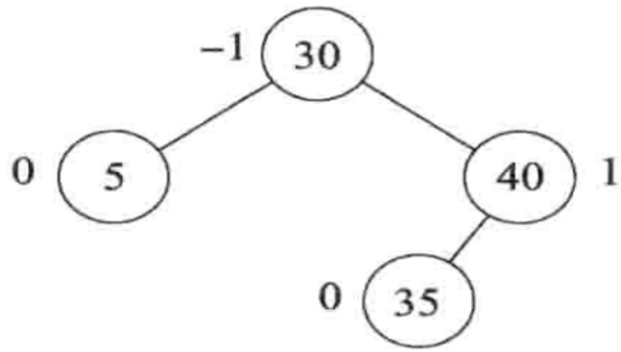


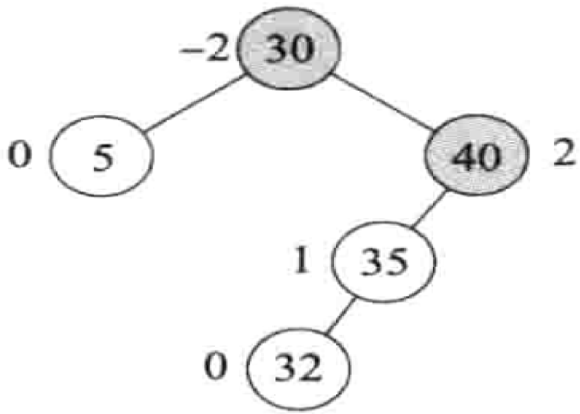
AVL搜索树的搜索

- 使用二叉搜索树的搜索
- 搜索所需时间：
  - O(logn)

AVL搜索树的插入

- 用二叉搜索树的的插入方法将元素插入到AVL搜索树中

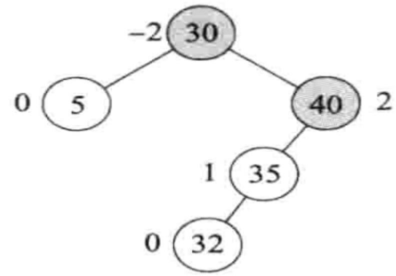




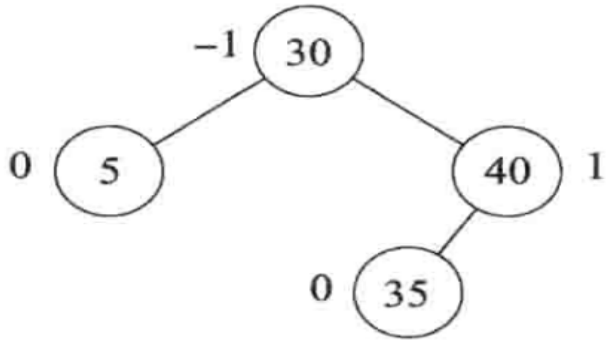
- 得到的树可能不再是AVL树(不平衡)
- 如果树成为不平衡的，我们需要进行调整来恢复树的平衡
- 如何调整？

AVL搜索树的插入

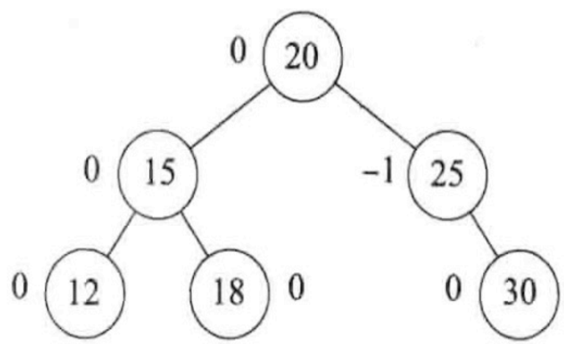
- 由插入操作导致产生不平衡树的几种现象：
  1. 不平衡树中的平衡因子的值限于-2,-1,0,1,2.
  2. 平衡因子为2的节点在插入前平衡因子为1，与此类似，平衡因子为-2的，插入前为-1。
  3. 在插入后，只有从根到新插入节点的路径上的节点才可能改变平衡因子。
  4. 假设A是平衡因子是-2或2的节点且是距离新插入节点最近的祖先结点。在插入前从A到新插入节点的路径上的所有节点的平衡因子都是0。
- A是AVL中的哪个节点？



- 节点X：在插入前，我们从根节点往下移动寻找新元素的插入位置时，从根节点到新元素的路径上，**最后一个**遇到的平衡因子是**-1或1**的节点.

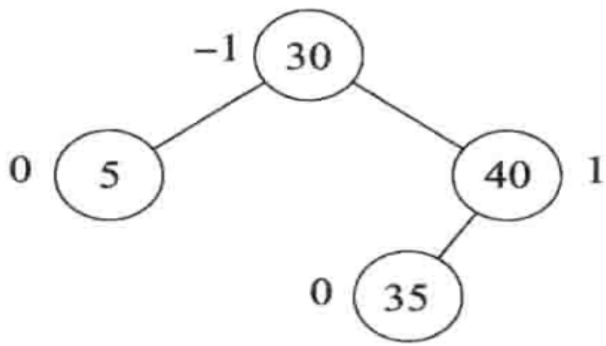


- 插入32， 节点X：？

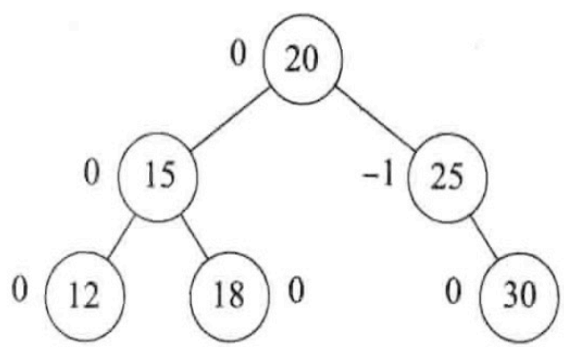


- 插入22， 节点X：？
- 插入50， 节点X：？
- 插入10， 节点X：？
- 插入14， 节点X：？

- 节点X：在插入前，我们从根节点往下移动寻找新元素的插入位置时，从根节点到新元素的路径上，**最后一个**遇到的平衡因子是**-1或1**的节点.



- 插入32， 节点X：40



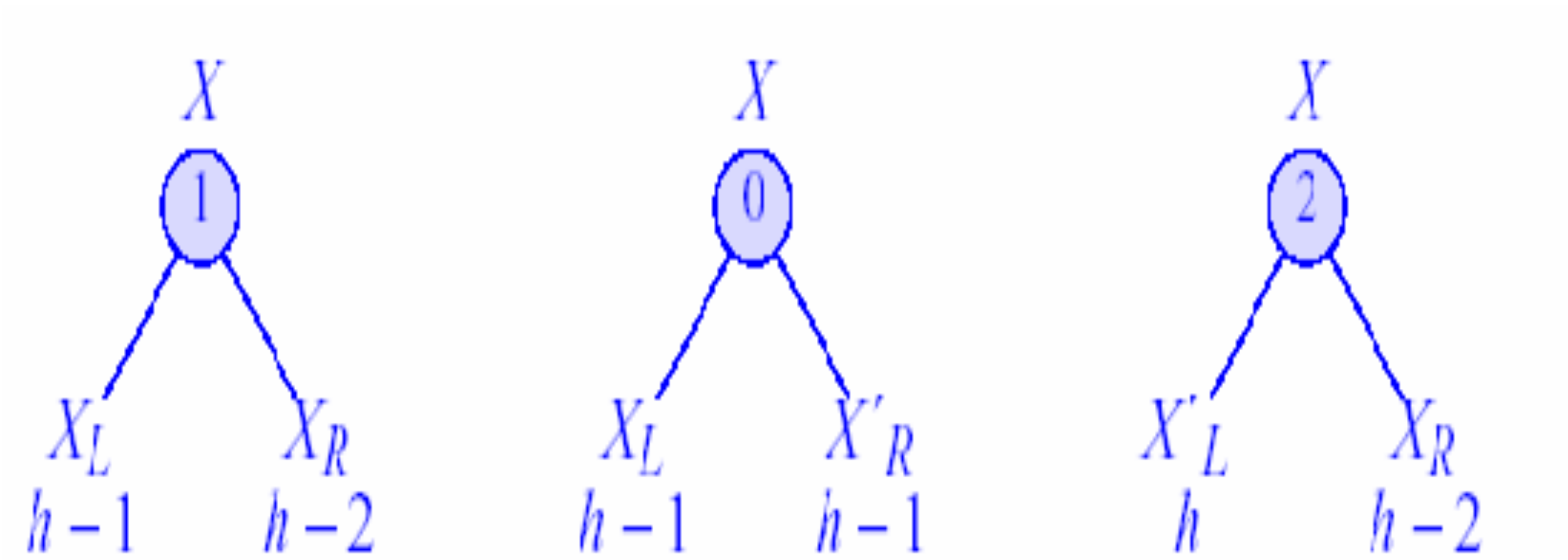
- 插入22， 节点X：25
- 插入50， 节点X：25
- 插入10， 节点X：不存在
- 插入14， 节点X：不存在

AVL搜索树的插入

- 如果节点X不存在，从根节点到新插入节点的途径中，所有节点在插入前的平衡因子都是0.
- 插入后，树的平衡不会被破坏
- 因此，只有插入后的树是不平衡的，X才存在

AVL搜索树的插入

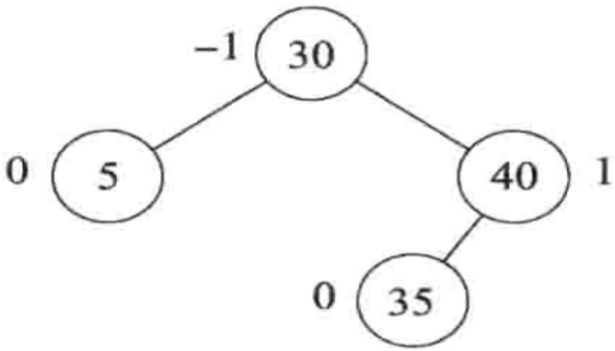
- 插入前bf(X)=1，插入后bf(X)=0或bf(X)=2

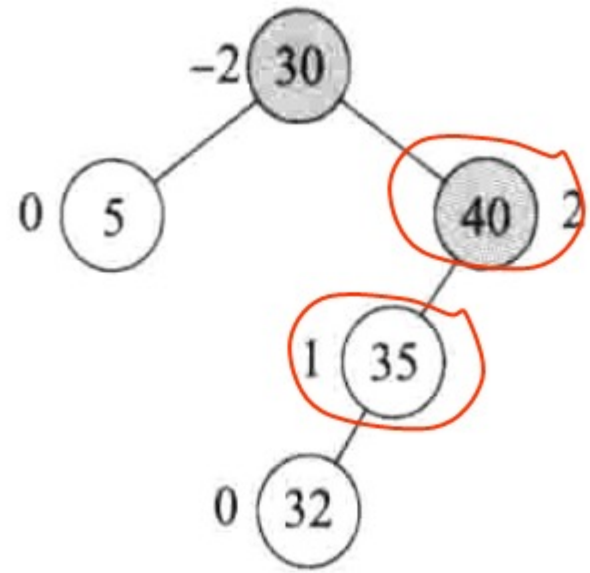


(a) 插入之前 (b)插入到X_R中之后 (c)插入到X_L中之后

AVL搜索树的插入

Insert(32)





AVL搜索树的插入

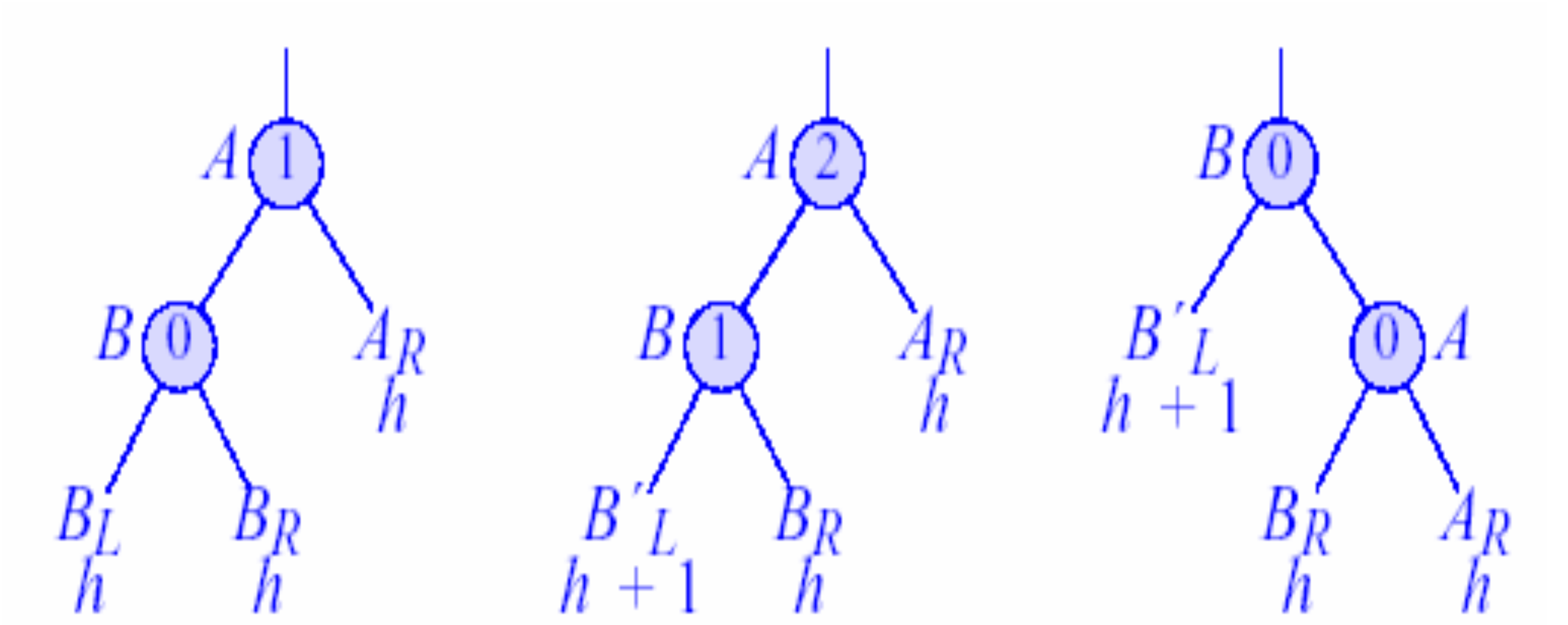
- 一棵树从平衡变为不平衡的唯一过程是：在插入操作之后，平衡因子bf(X)的值由-1变为-2，或者由1变为2
- 节点X就是之前提到的A节点

不平衡类型

- 在插入后, A的平衡因子是-2或2，当节点A已经被确定时，A的不平衡性：

1. **LL**: 新插入节点在A节点的左子树(子节点)的左子树（孙节点）中
2. **LR**: 新插入节点在A节点的左子树的右子树中
3. **RR**: 新插入节点在A节点的右子树的右子树中
4. **RL**: 新插入节点在A节点的右子树的左子树中

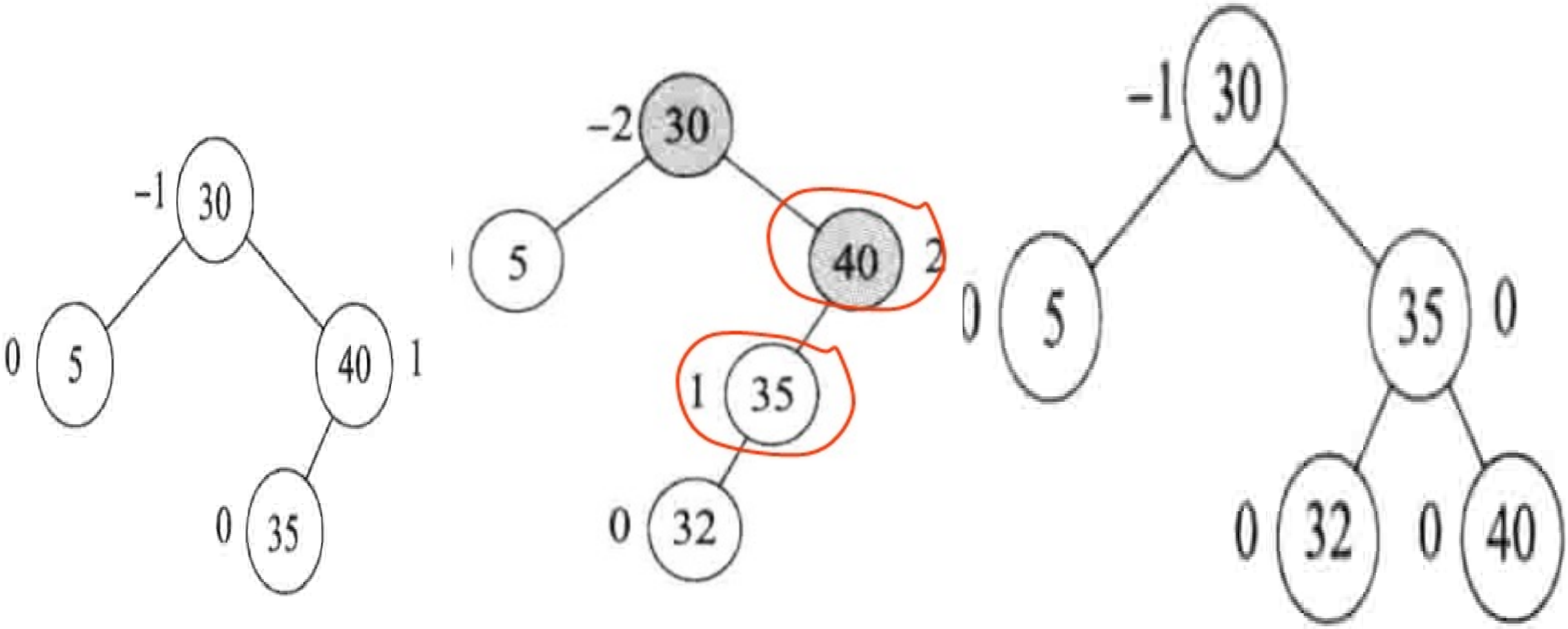
LL旋转



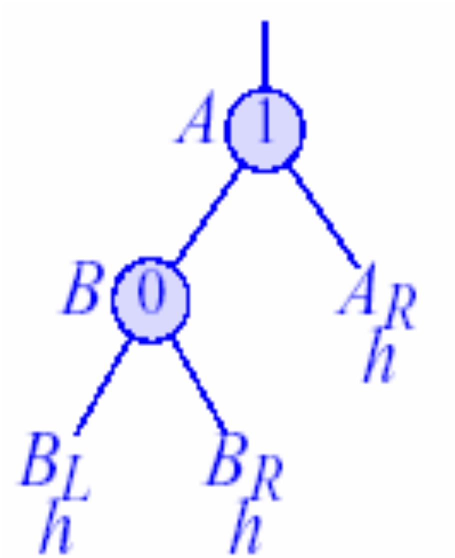
(a) 插入之前　　(b)插入到B_L中之后　　(c) LL旋转后

- 可以证明吗？

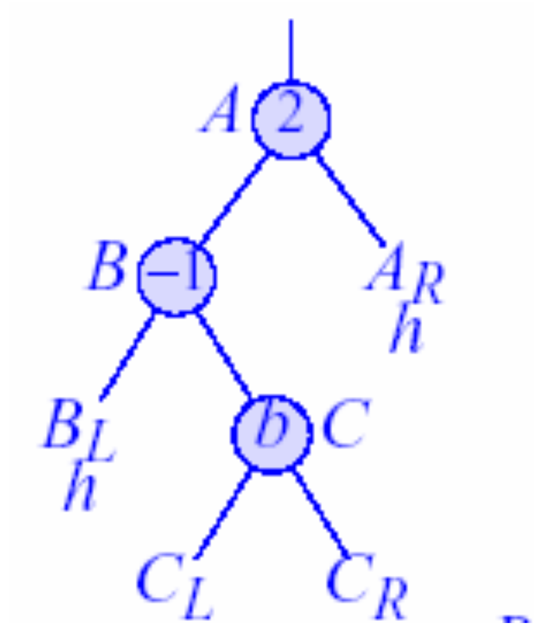
LL旋转示例



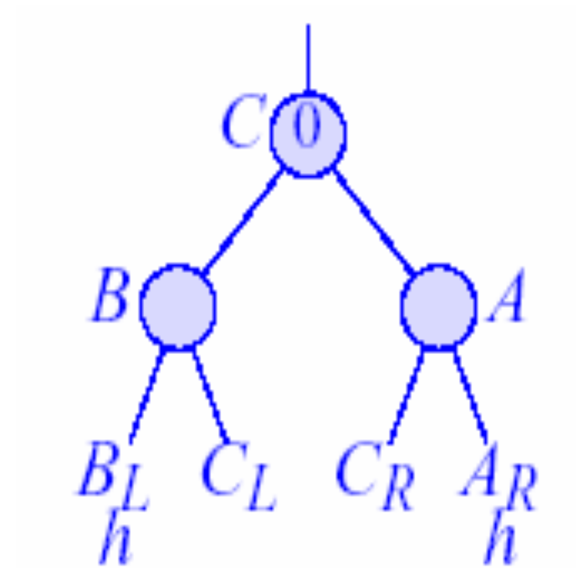
LR 旋转



(a)插入之前

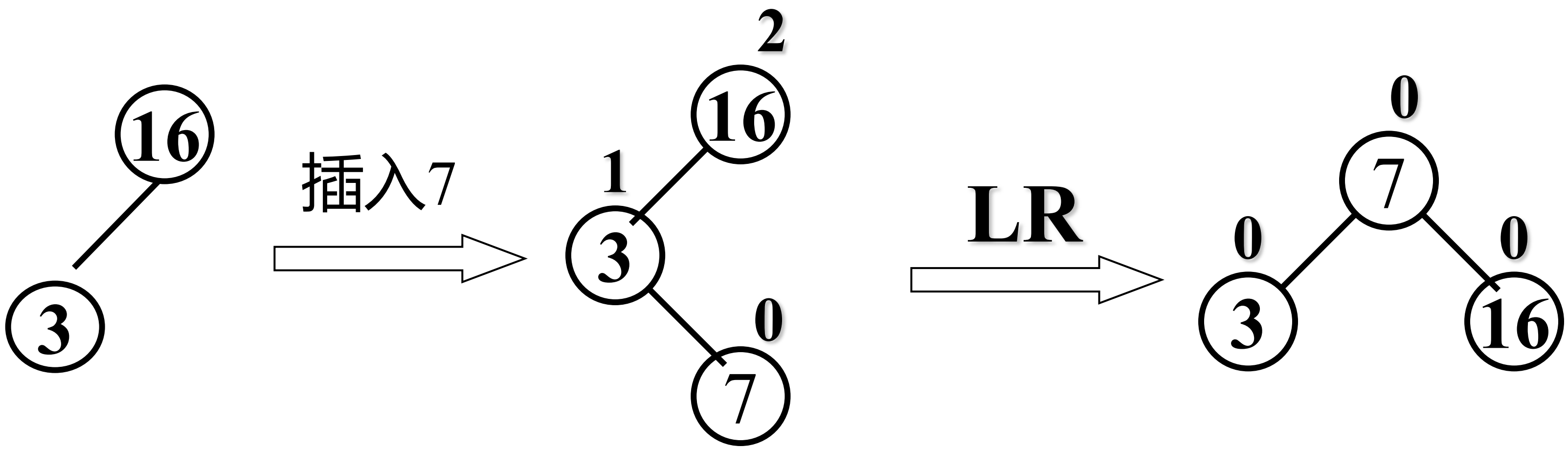


(b)插入到B_R之后



(c)LR旋转之后

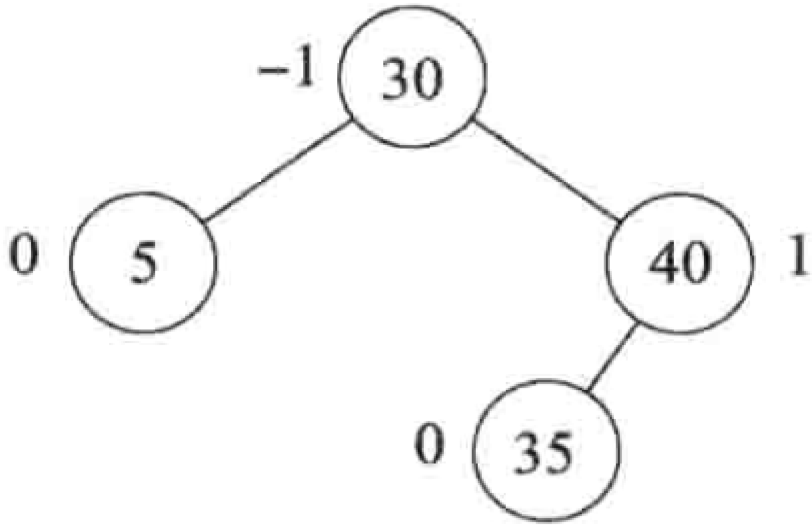
例：LR旋转



- 插入复杂性：O(高度)，然后一次旋转就可恢复平衡

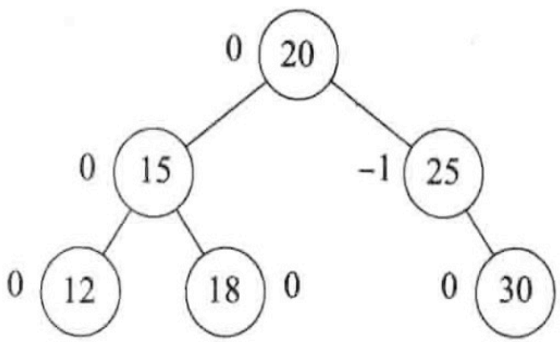
习题-请提交

- 插入37

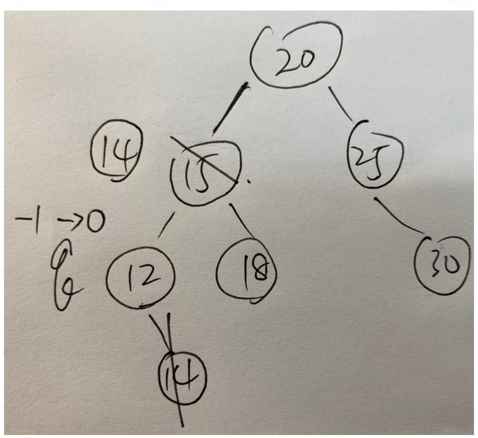


AVL搜索树的删除

- 通过执行二叉搜索树的删除，可从AVL搜索树中删除一个元素。但也会导致产生不平衡树。
- 设q:删除节点的父节点。



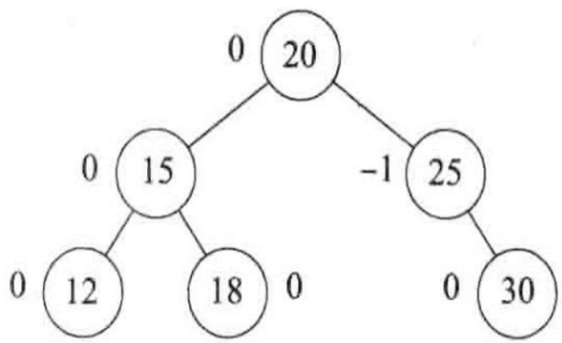
- 删除 25， q:?
- 删除 15， q:?



- 删除15，q:?

AVL搜索树的删除

- 通过执行二叉搜索树的删除，可从AVL搜索树中删除一个元素。但也会导致产生不平衡树。
- 设q:删除节点的父节点。



- 删除 25， q:20
- 删除 15， q:原15的节点（根的左孩子）

AVL搜索树的删除

- 设q:删除节点的父节点。
- 如果删除节点在左子树: bf(q)-1;

  删除节点在右子树： bf(q)+1
- 由删除操作导致产生不平衡树的几种现象：
  - 1)如果q的新平衡因子是0，那么它的高度减少了1，需要改变它的父节点(如果有的话)和其他某些祖先节点的平衡因子。
  - 2)如果q的新平衡因子是－1或1，那么它的高度与删除前相同，无需改变其祖先的平衡因子值。
  - 3)如果q的新平衡因子是－2或2，那么树在q节点是不平衡的。
- 由删除操作产生的不平衡分为六种类型：R0, R1, R-1, L0, L1, L-1。

AVL搜索树的删除

- 请注意！

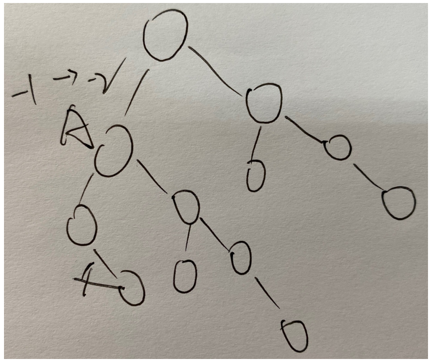
- 删除叶子节点或者度为1的节点，平衡因子从被删除节点的父节点开始变化。

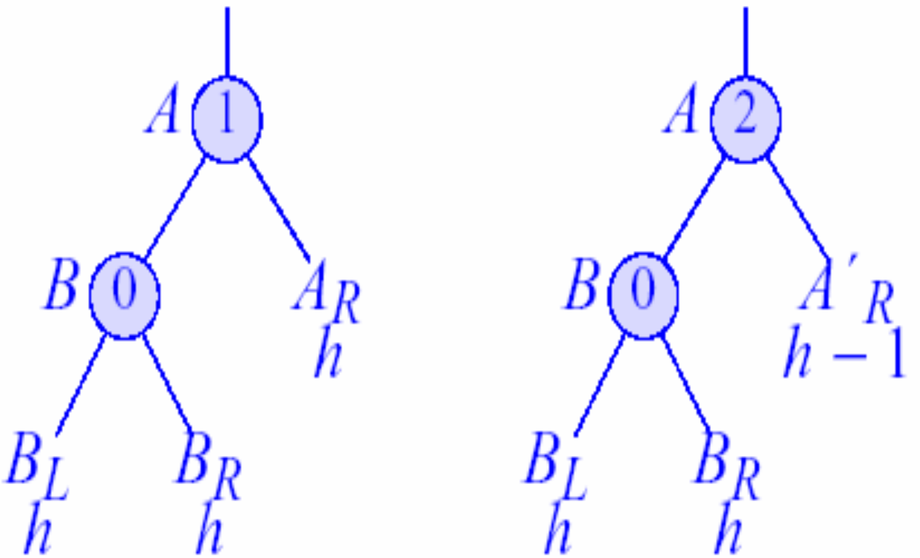
- 删除度为2的节点需要使用复制删除的技巧，使用被删除节点的**直接前驱（左子树的最大值）**来替换被删除节点，然后删除直接前驱，那么平衡因子就是从直接前驱的父节点开始变化的。

R0 旋转

设A是从q到根节点的路径第一个平衡因子变为2或－2的节点。

删除发生在A的右子树：R0, R1, R-1,
删除发生在A的左子树： L0, L1, L-1





(a) 删除之前　　　　(b) 从A_R中删除之后

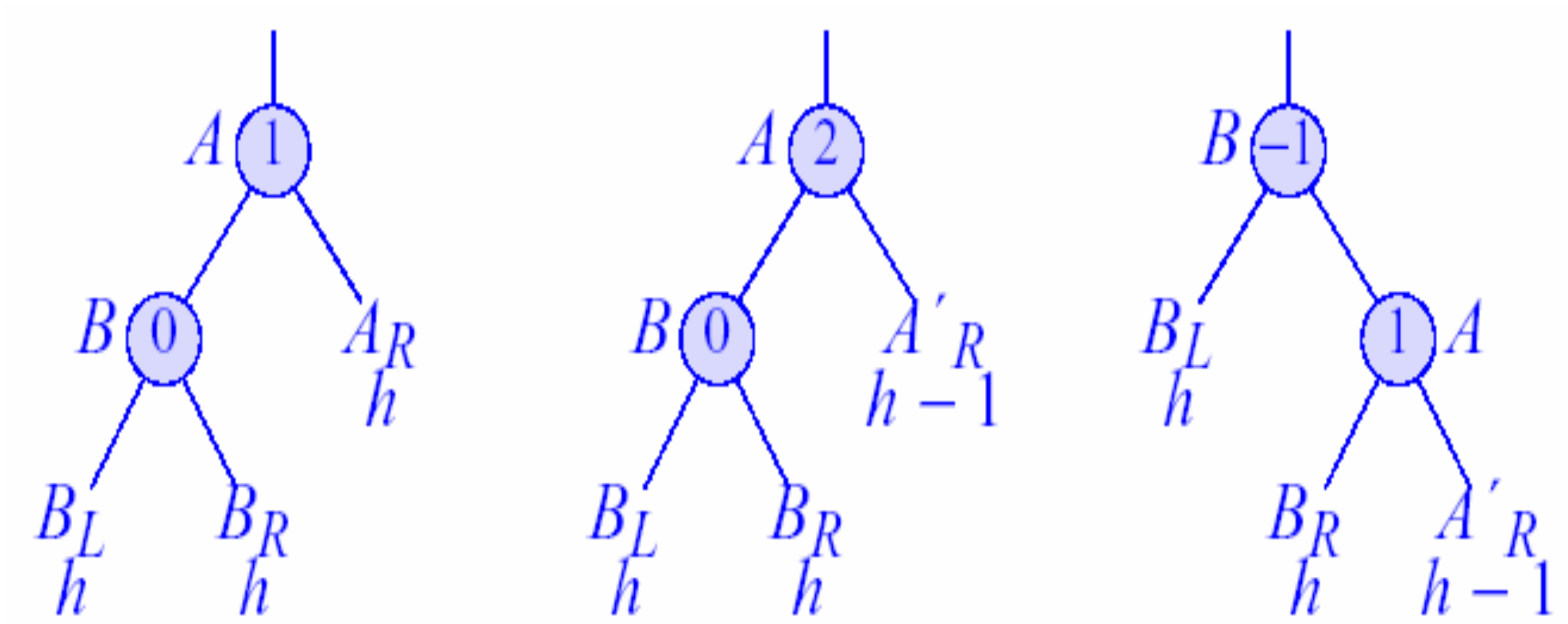
若删除后bf(A)=2，之前bf(A)=1，则A一定有一颗以B为根节点的左子树

根据bf(B)的值，R0表示删除发生在A的右子树且bf(B)=0

- R0 旋转:单旋转

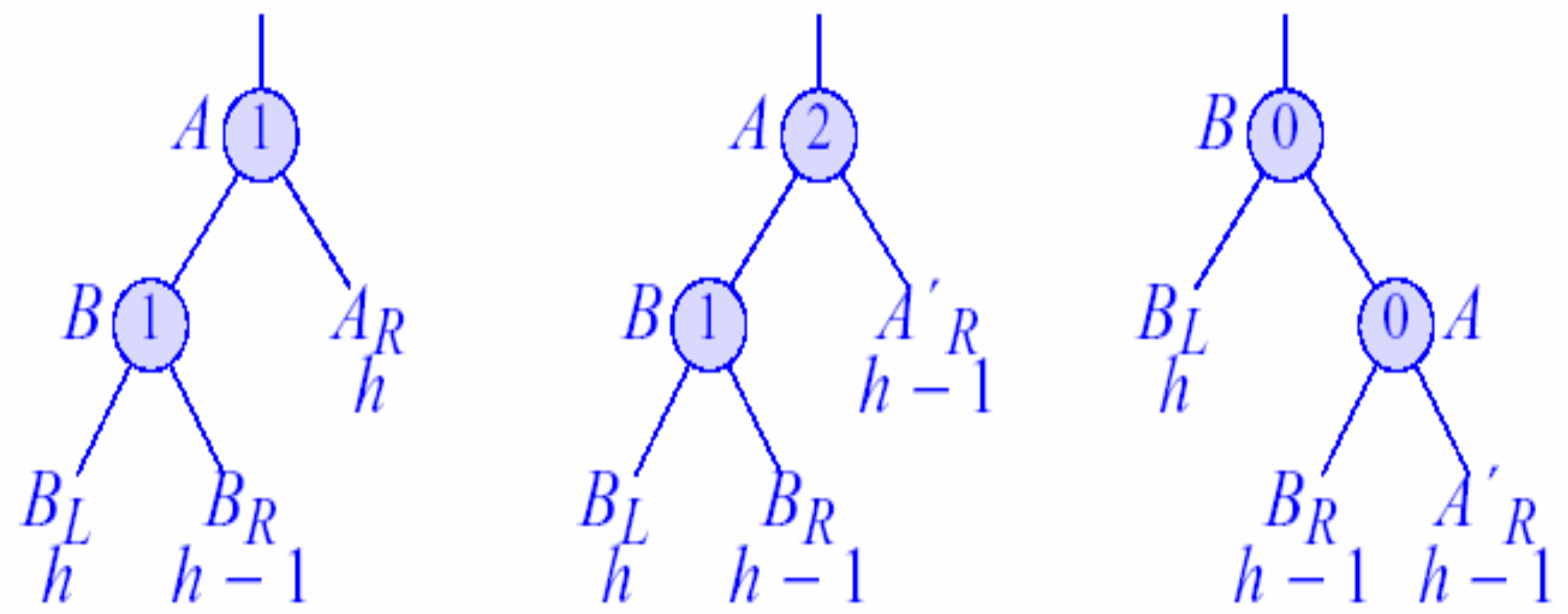
R0 旋转

- R0 旋转:单旋转,整棵树在一次旋转后获得平衡



(a) 删除之前　　(b) 从A_R中删除之后　　(c) R0 旋转之后

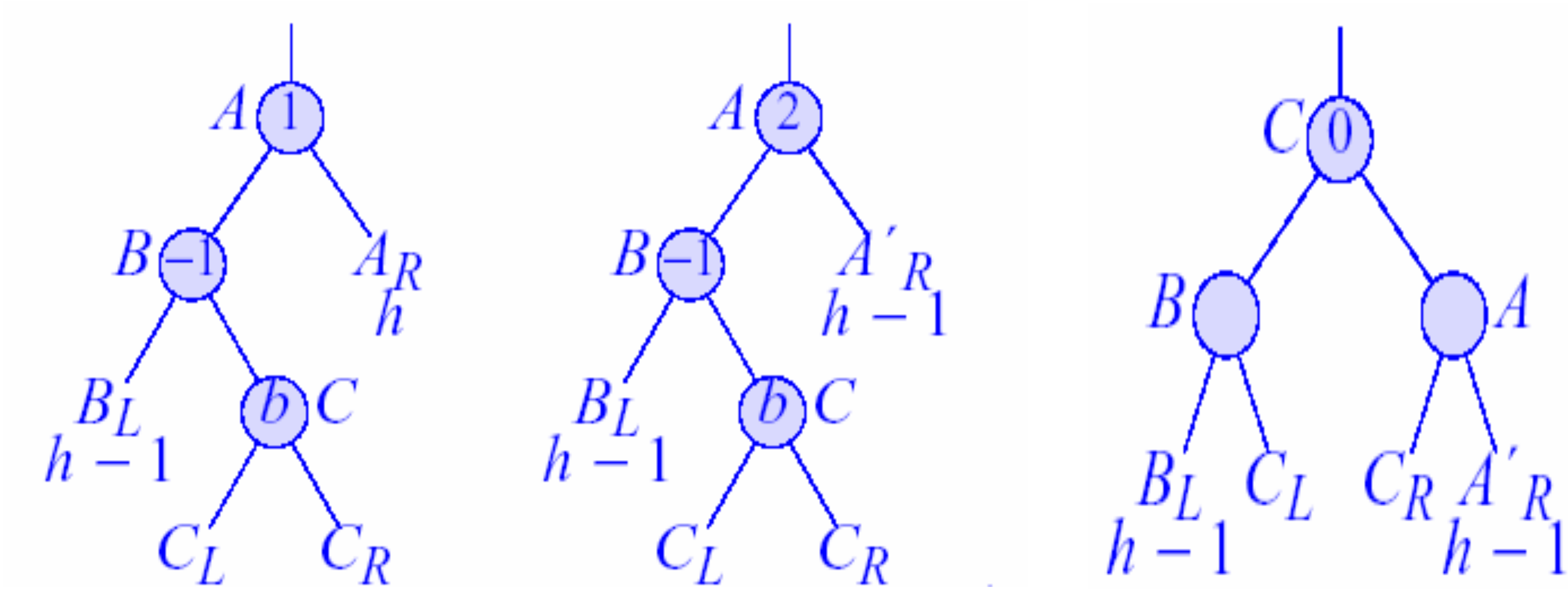
R1 旋转



(a) 删除之前 (b) 从A_R中删除之后 (c) R1 旋转之后

- R1 旋转:单旋转，必须检查到根节点路径上的节点，旋转次数O(logn)

R-1 旋转

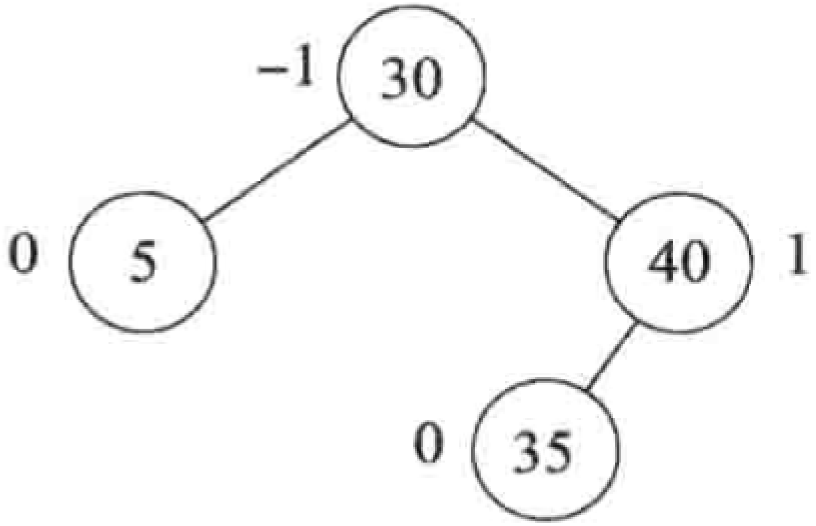


(a) 删除之前 (b) 从A_R中删除之后 (c) R-1 旋转之后

- R-1 旋转:双旋转,必须检查到根节点路径上的节点(C_L 高度跟C_R高度均为h-1)

例：

- 删除5



15.4 B-树

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。

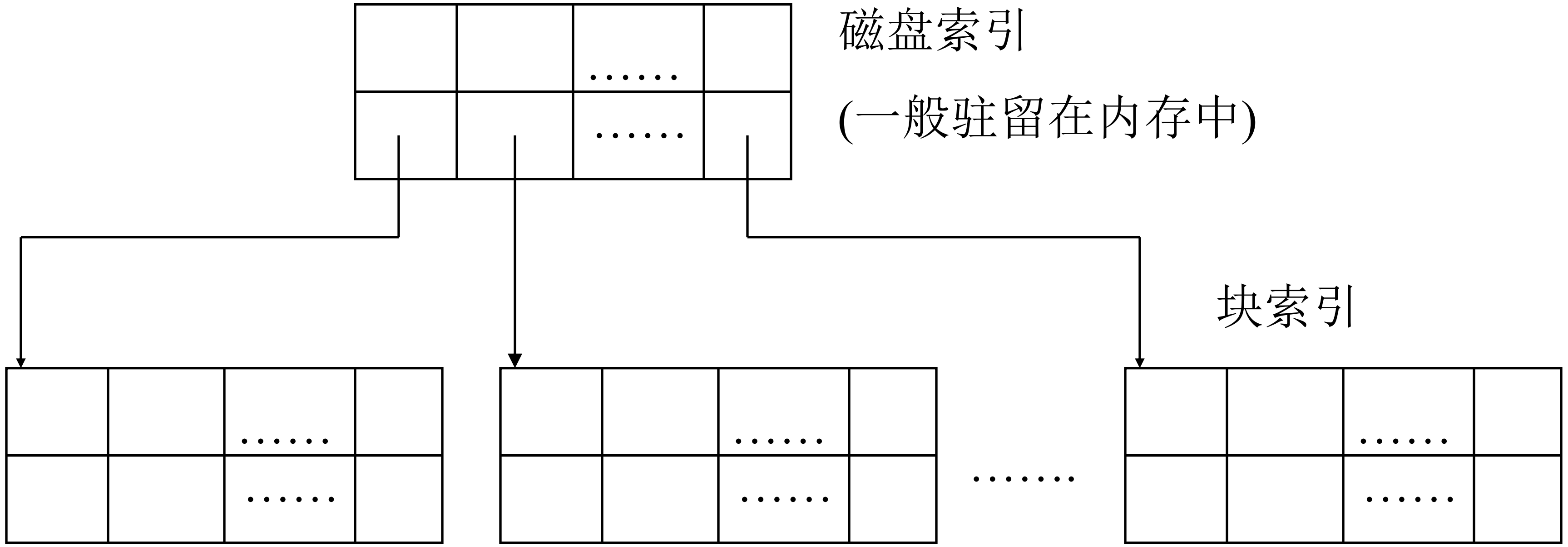
15.4.1 索引顺序访问方法ISAM

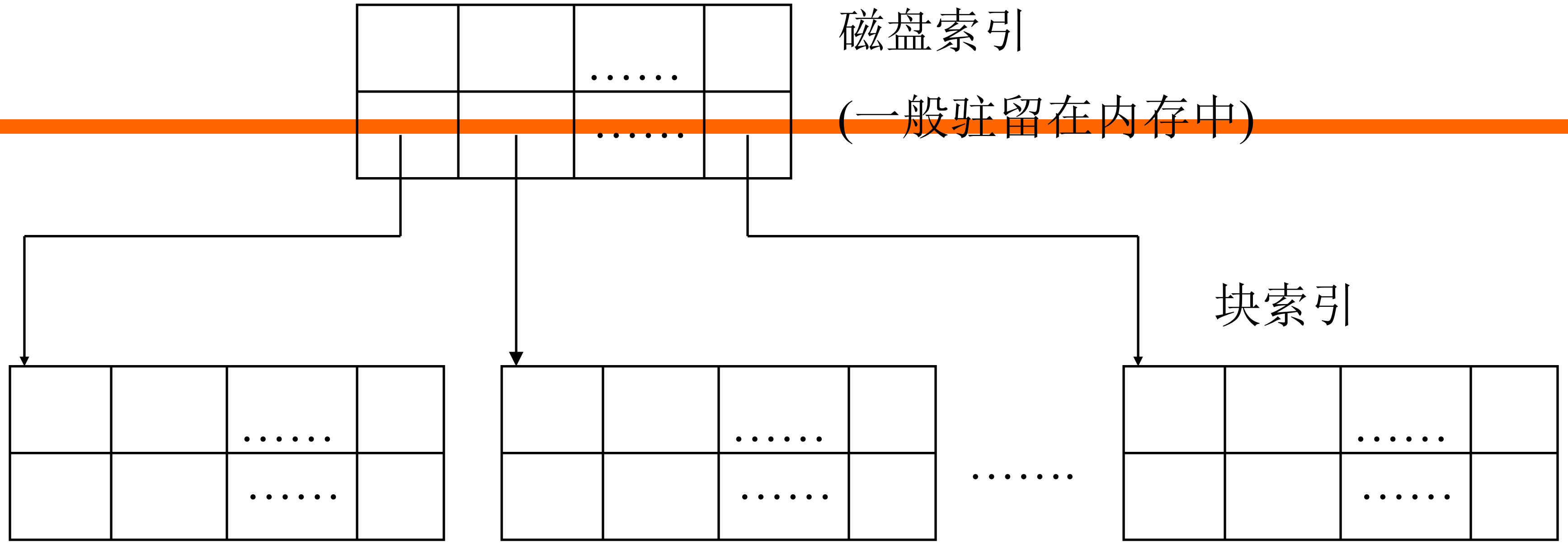
- 随机访问：
  - 必须维护一个**索引表**

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
- 对存储在磁盘上的数据，B-树是一种适合于索引方法的数据结构

15.4.2 m叉搜索树

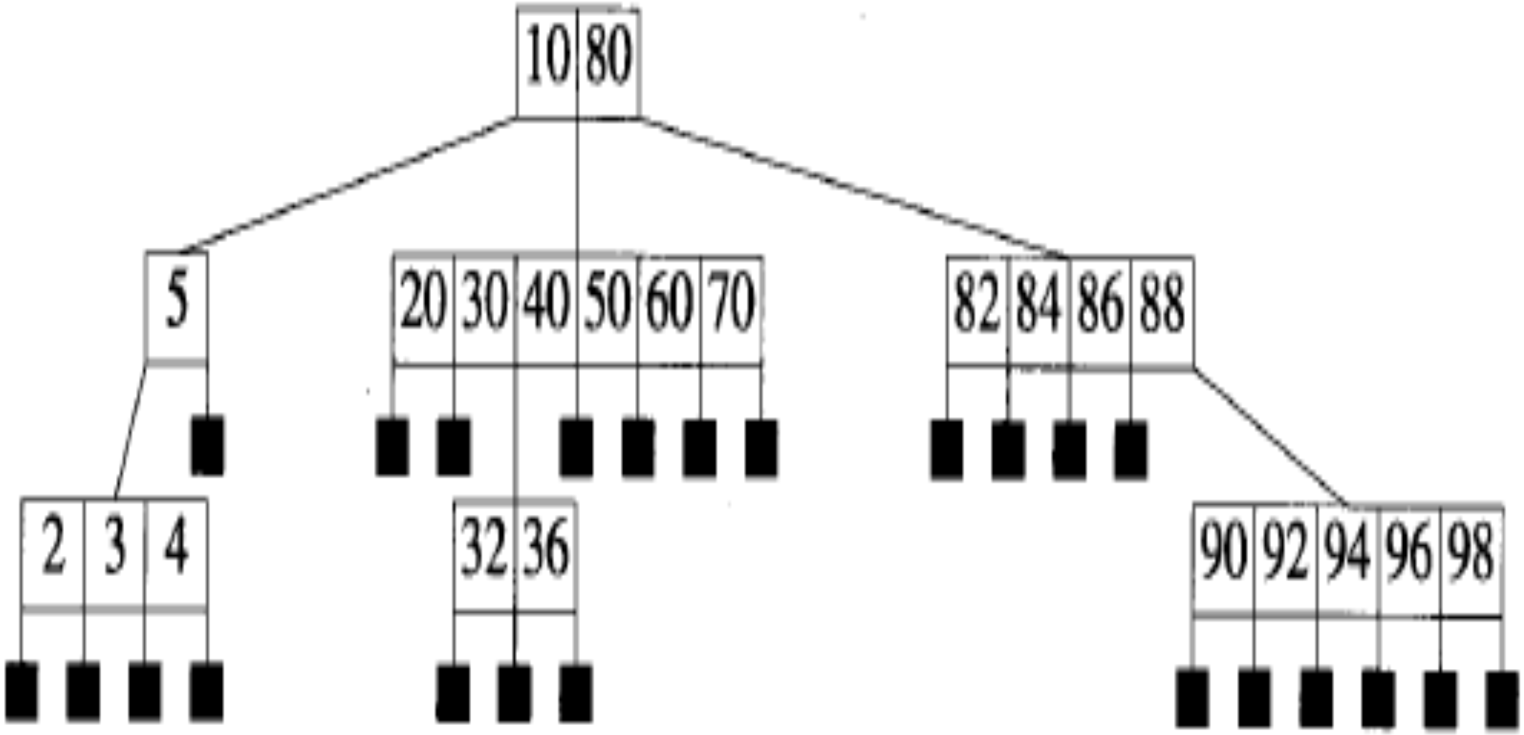
- m叉搜索树(m-way Search Trees)定义：
- m 叉搜索树可以是一棵空树，如果非空，它必须满足以下特征：
  - 1) 在相应的扩充搜索树中(用外部节点替换空指针)，每个内部节点**最多**可以有**m** 个子女及**1～m-1**个元素(外部节点不含元素和子女)。
  - 2) 每个含**p** 个元素的节点，有**p+ 1**个子女。
  - 3) (接下页)

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 k_1 C_1 k_2 C_2 k_3 C_3 ………………….. k_p C_p

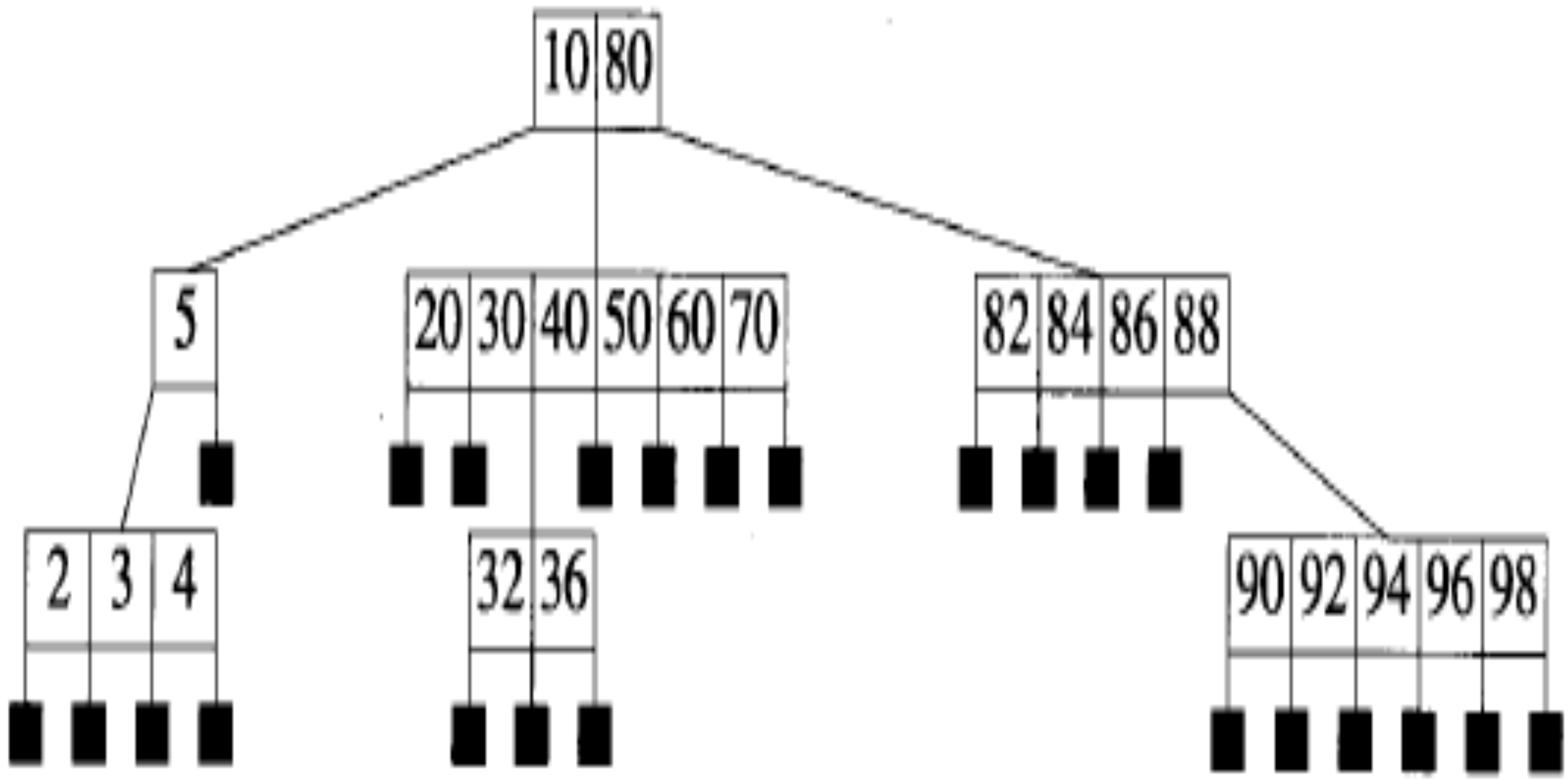
- $k_1<k_2<k_3<$ ……..$<k_p$
- C_0为根的子树： $<k_1$
- C_1为根的子树： $>k_1$, $<k_2$
- …….
- C_i为根的子树： $>k_i$, $<k_{i+1}$
- …
- C_p为根的子树： $> k_p$

例：7叉搜索树（每个节点最多容纳6个元素）



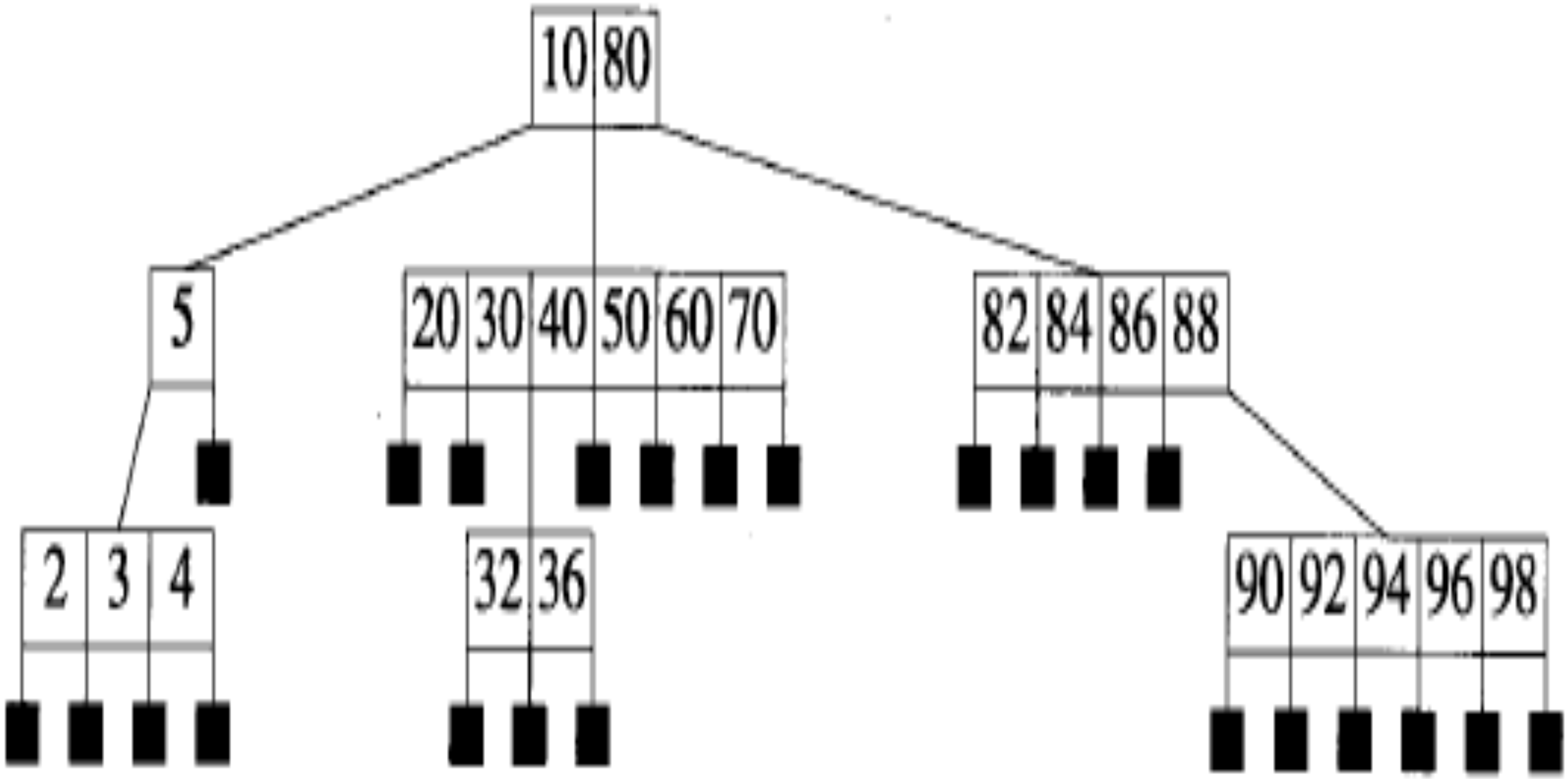
定义中把外部节点包含进来是有用的，实际应用中，用空指针

m叉搜索树的搜索



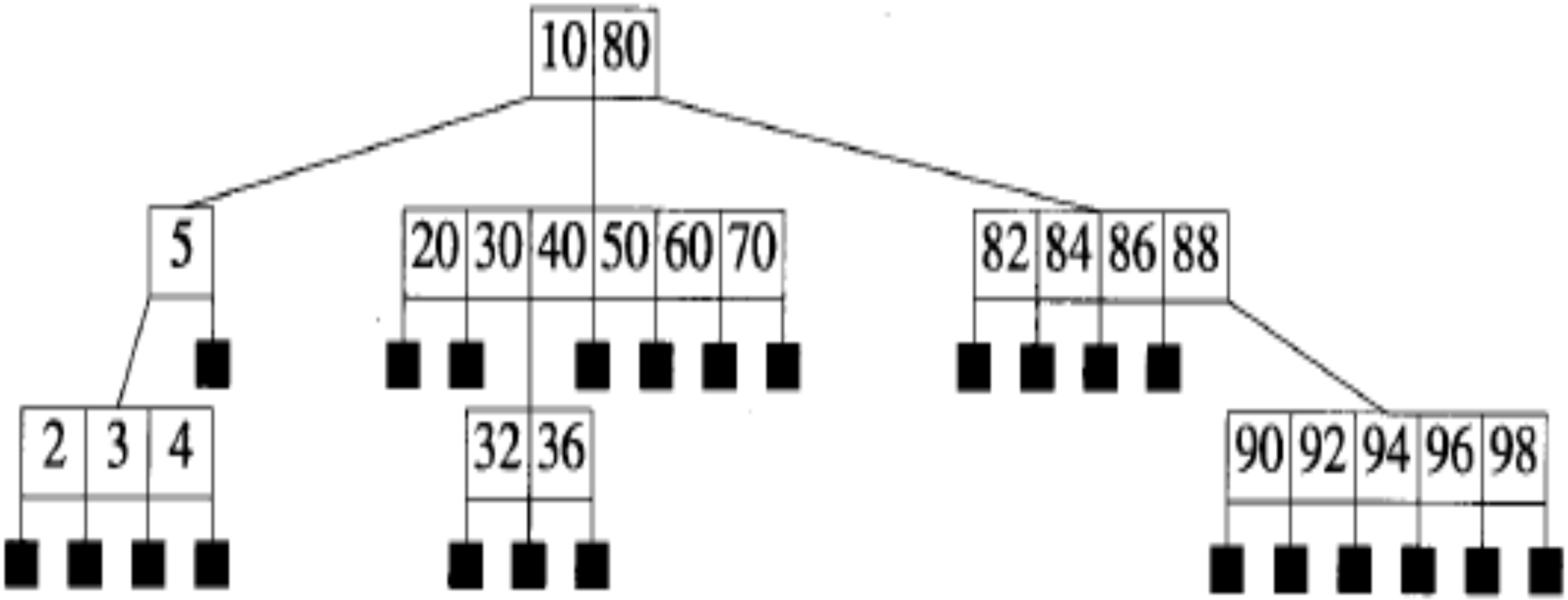
- 搜索: 32 ,31

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？65不能容纳，作为第6个孩子

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 （c_0最大值，或者c_1最小值）。
  - 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，5替换，然后4替换

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： $\sum_{i=0}^{h-1} m^i = (m^h-1)/(m-1)$

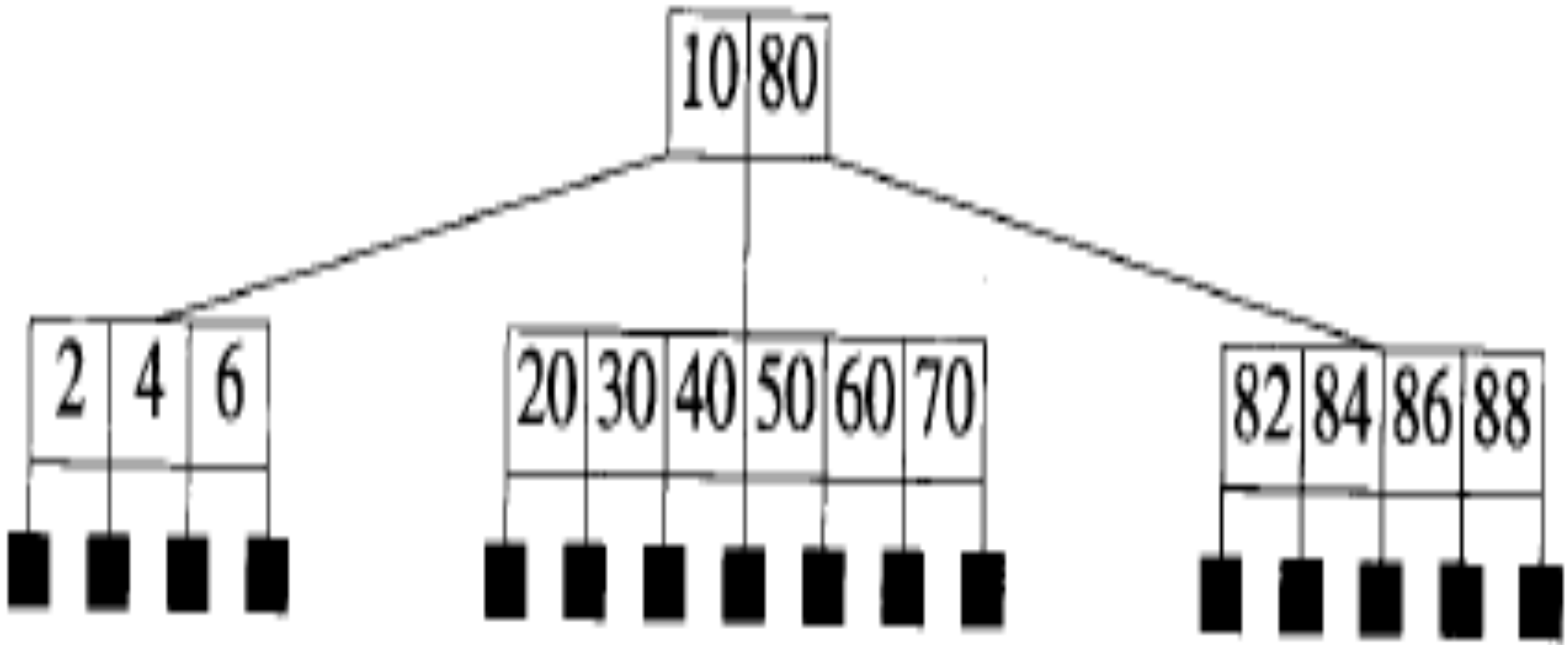
$\rightarrow \log_m(n+1) \le h \le n$

- 搜索、插入和删除操作需要的磁盘访问次数：O (h) 。

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。

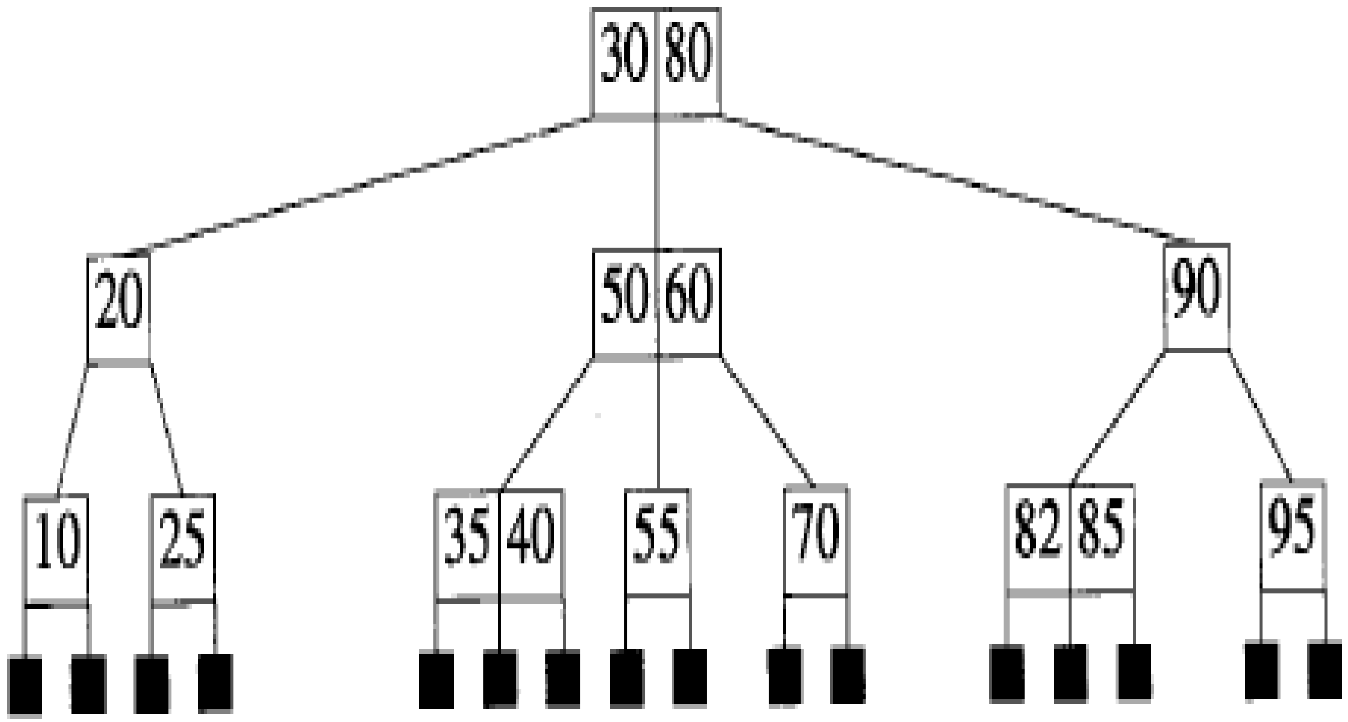
例：7阶B-树



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:

  $1+(d-1)(2+2d+\ldots+2d^{h-2})$

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

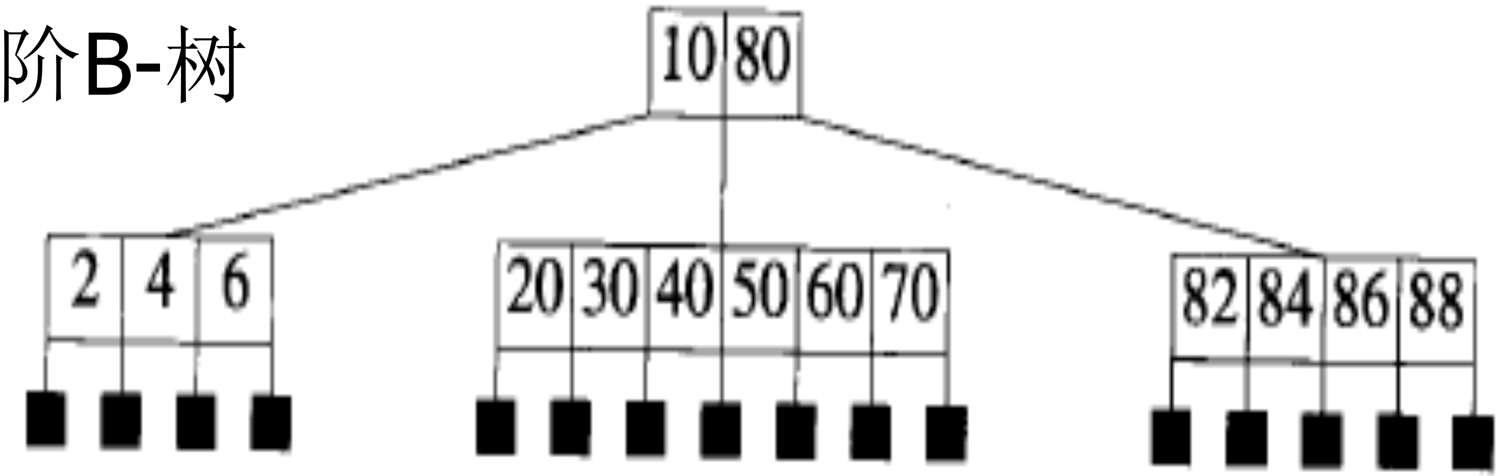
- 实际上， B-树的序（阶数）取决于磁盘块的大小和单个元素的大小。

15.4.5 B-树的搜索

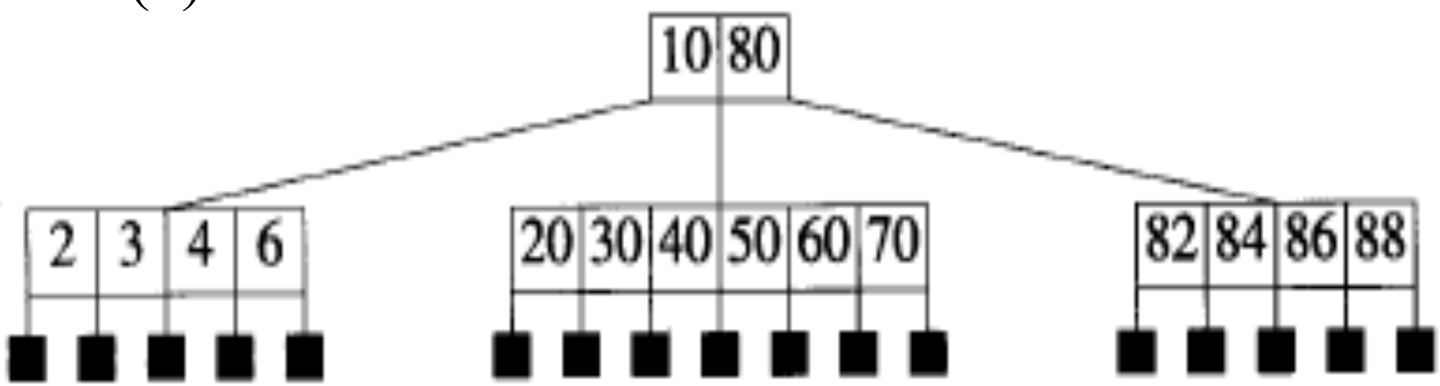
- B-树的搜索算法与m叉搜索树的搜索算法相同。
- 磁盘访问次数最多是h(h是B-树的高度)。

15.4.6 B-树的插入

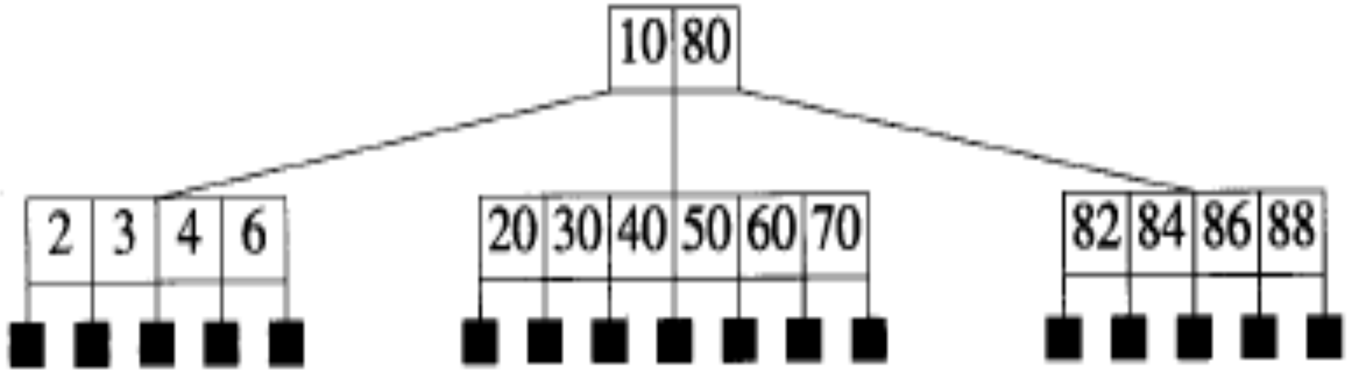
- 7阶B-树



Insert(3)



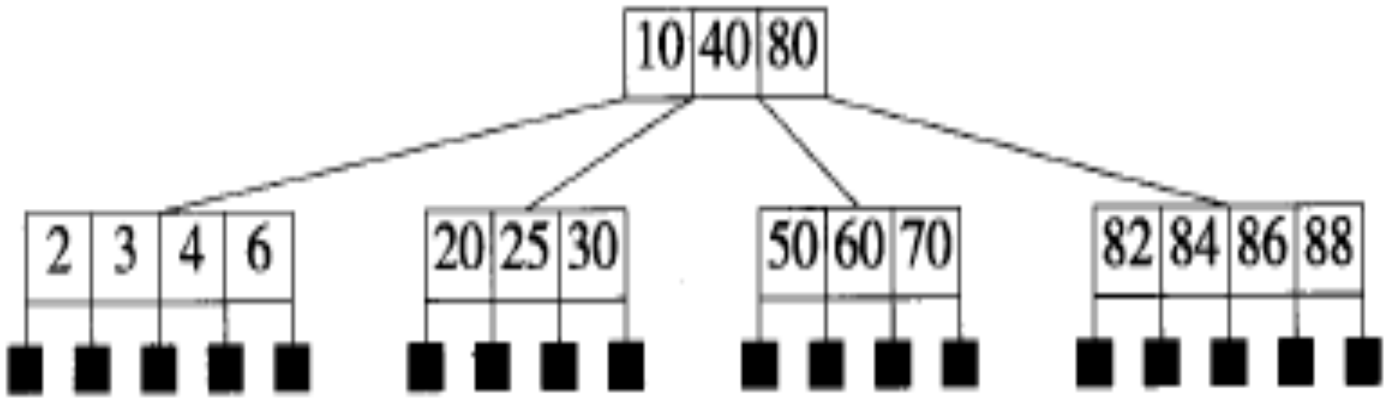
a)



•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， $d = \lceil m/2 \rceil$

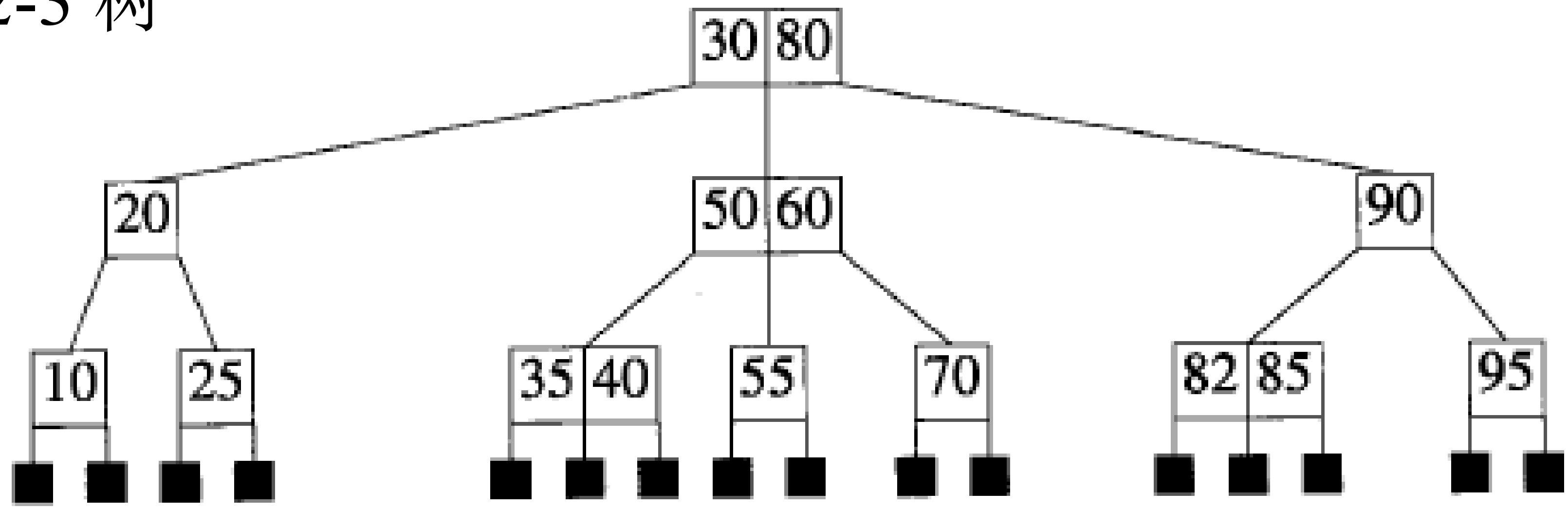
•P: d-1, c_0, (e_1, c_1),…,(e_d-1, c_d-1)

•Q: m-d, c_d, (e_d+1, c_d+1),…,(e_m, c_m)

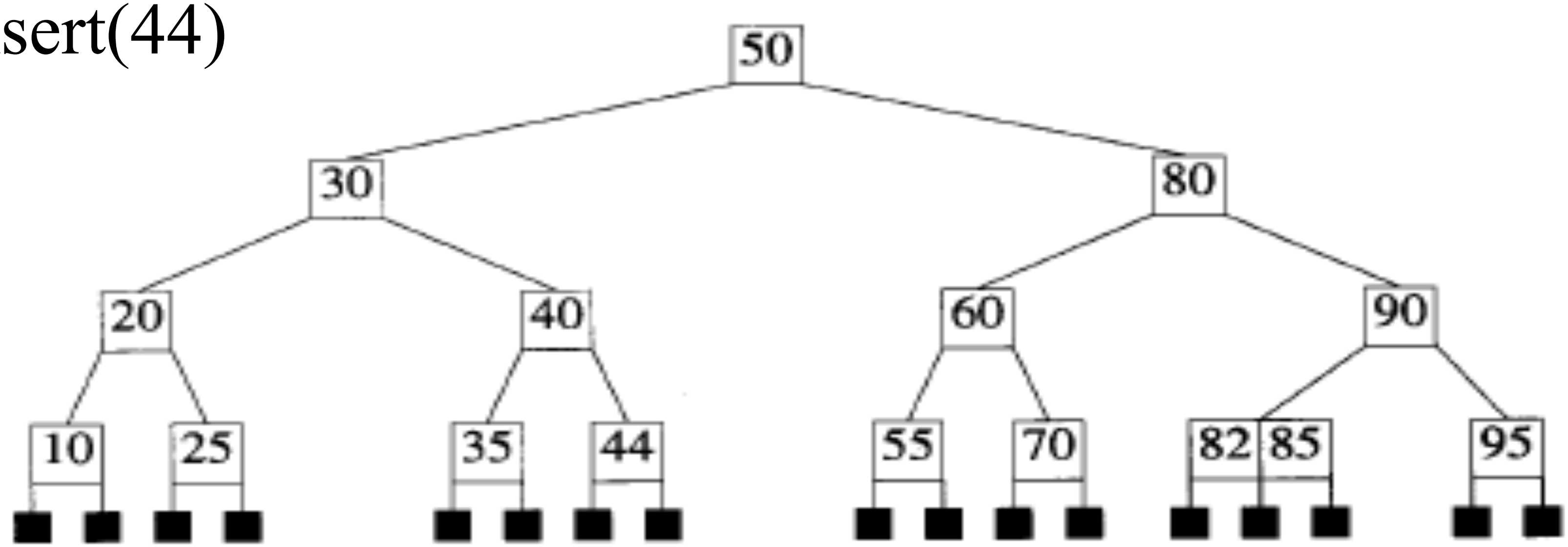


b)

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

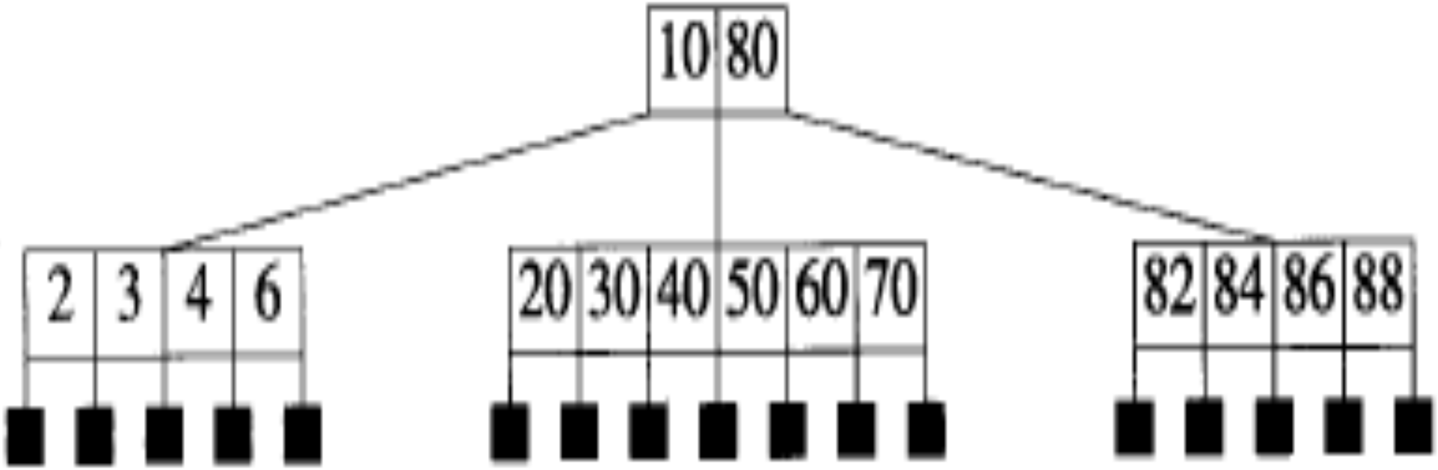
15.4.7 B-树的删除

- 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 。

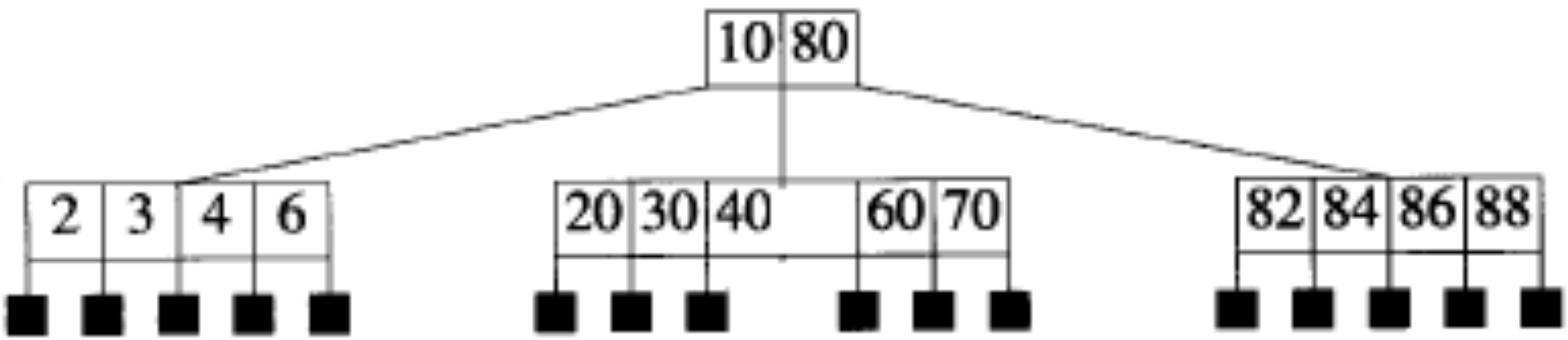
B-树的删除

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$\lceil m/2 \rceil-1$)*的树叶中删除一个元素，只需要将修改后的节点写回。


10 80
2 3 4 6
20 30 40 50 60 70
82 84 86 88

Delete(50)

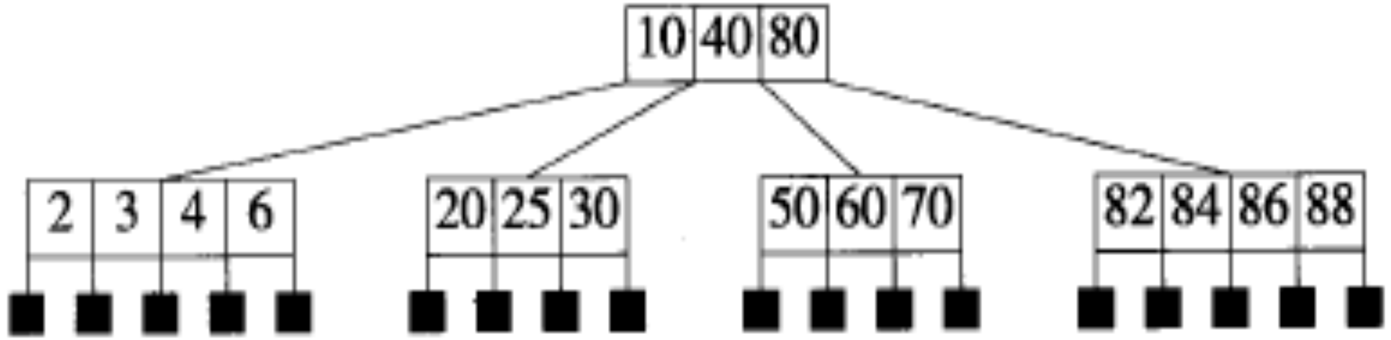
a)


10 80
2 3 4 6
20 30 40 60 70
82 84 86 88

B-树的删除

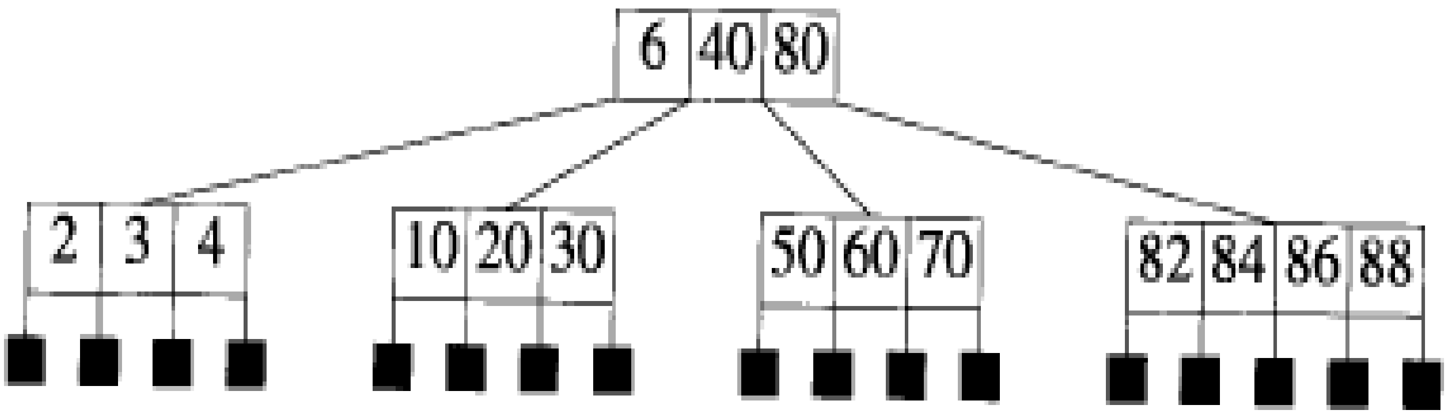
2. 当被删除元素在一个非根节点中且该节点中的元素数量为最小值时，

(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

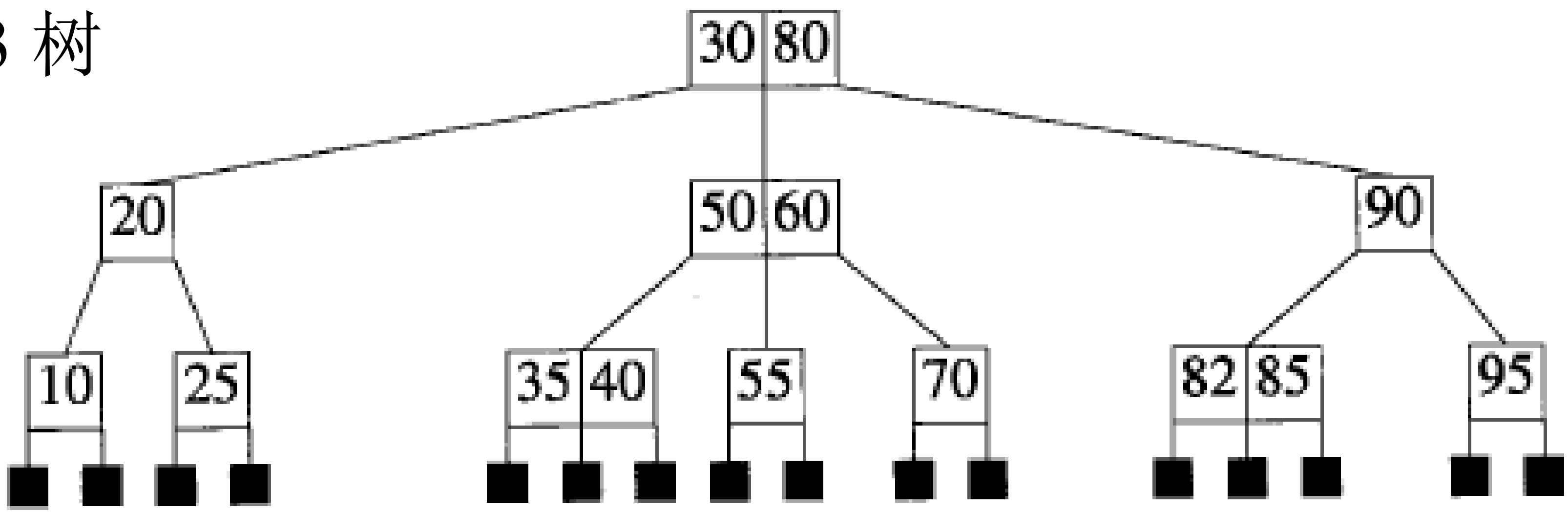


B-树的删除

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，

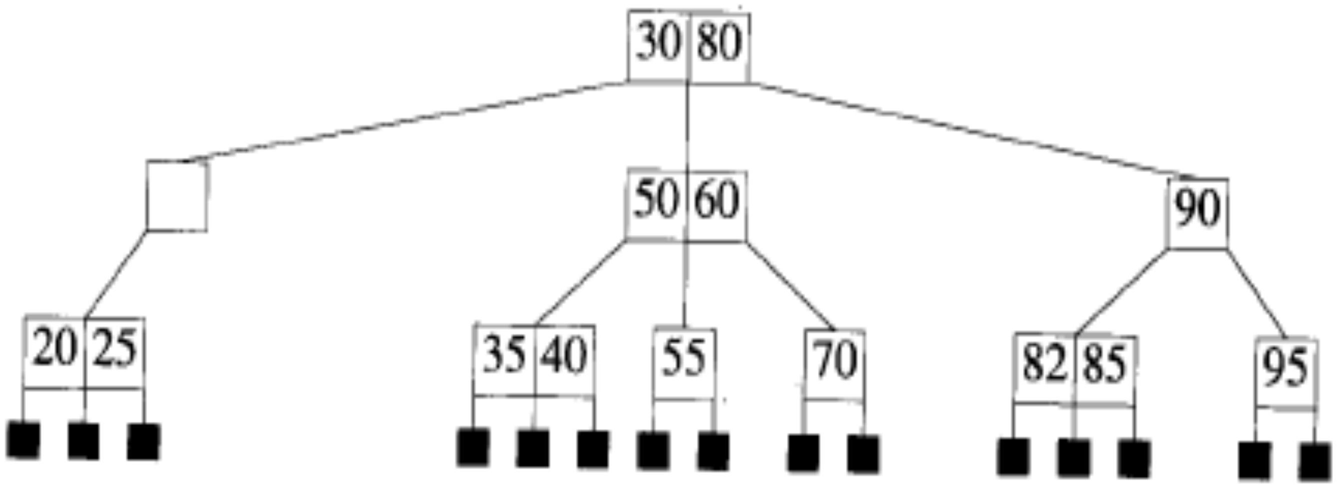
   (2) 它的最相邻的兄弟(最相邻的左或右兄弟)有最少数目元素，将**两个兄弟与父节点中介于两个兄弟之间的元素合并成一个节点**。删除父节点中的相应元素。

2-3 树



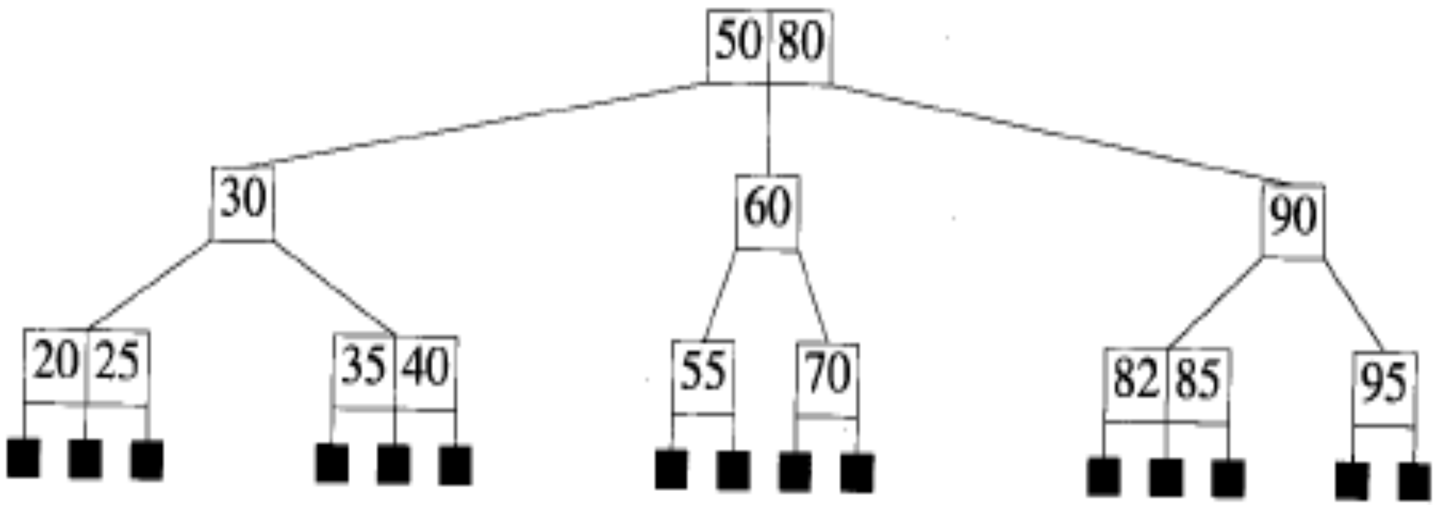
Delete(10)：（25）和20合并

2-3 树



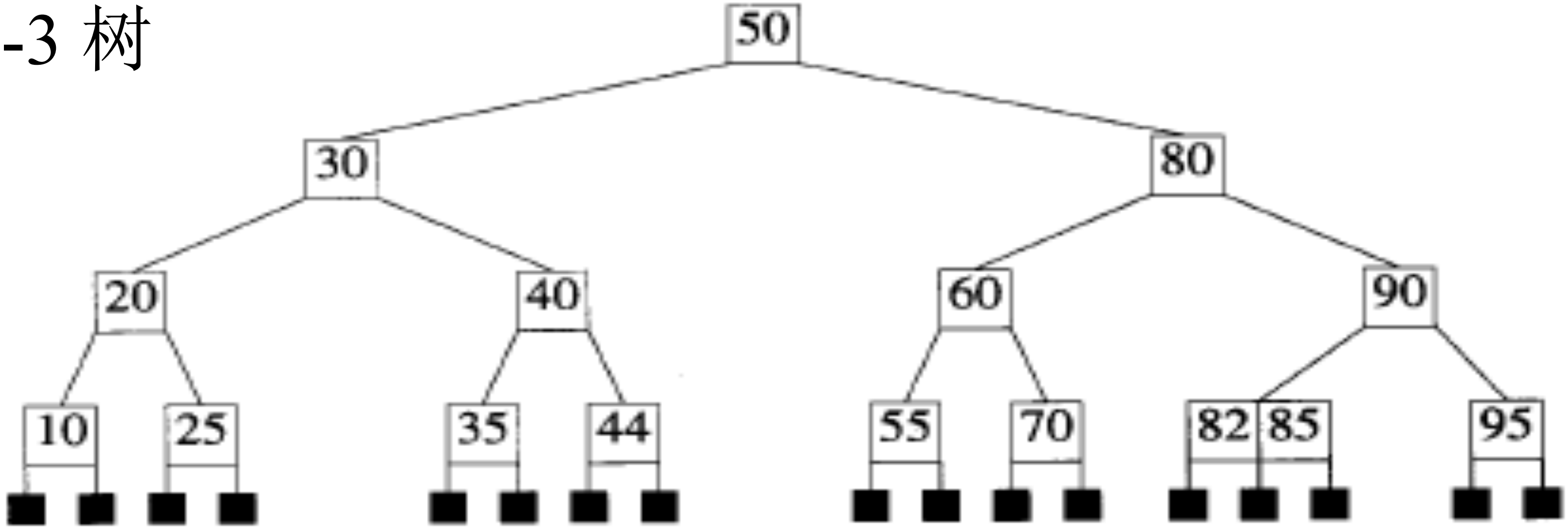
b)

Delete(10)： 借50



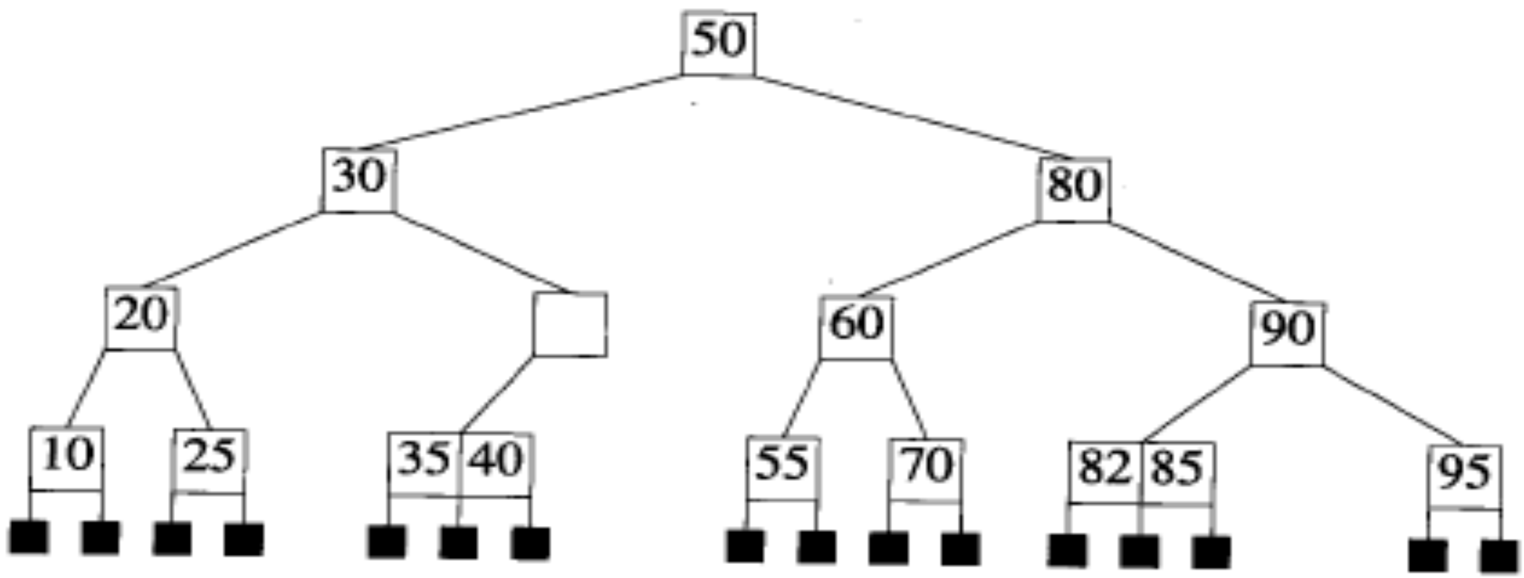
c)

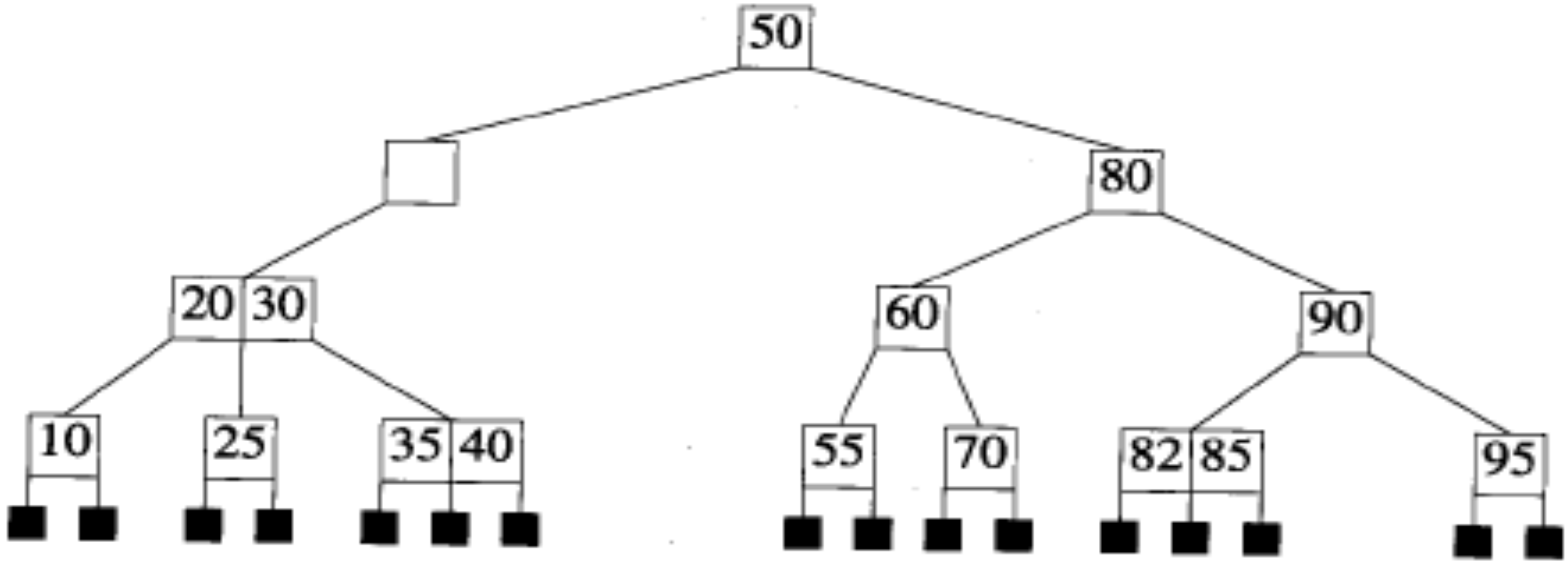
2-3 树

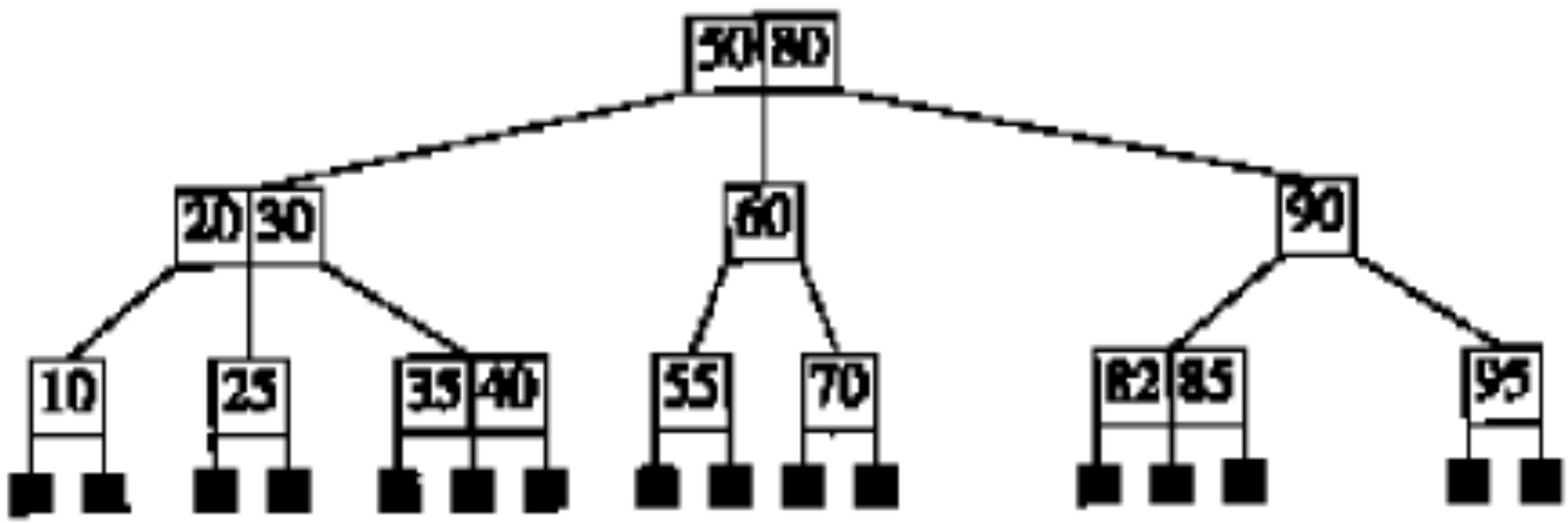


c)

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

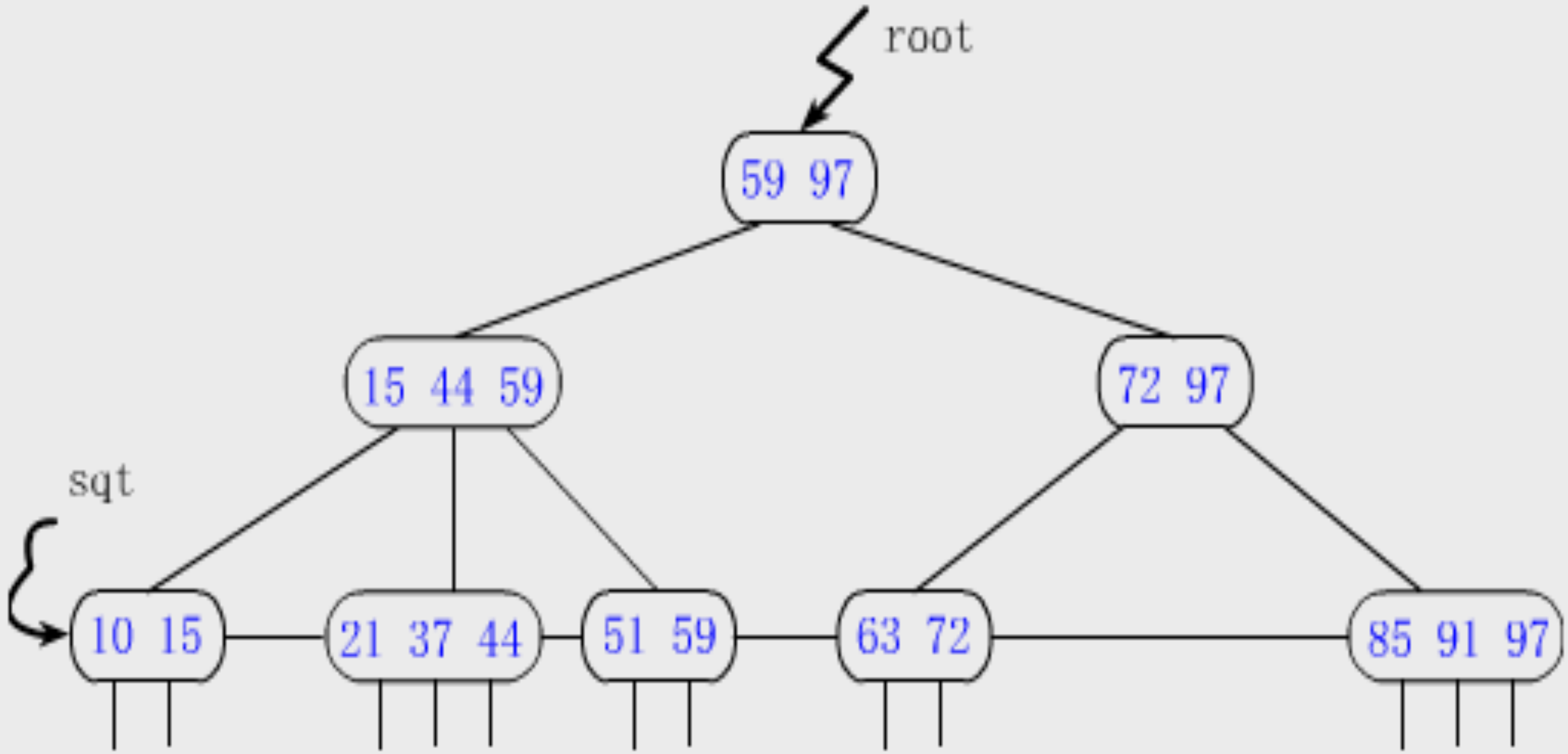
磁盘访问的总次数

- 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。

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

- P366：1， *关键字：**15，14，13，12，11，10，9***
- P388: 57