第9章 查找

回顾

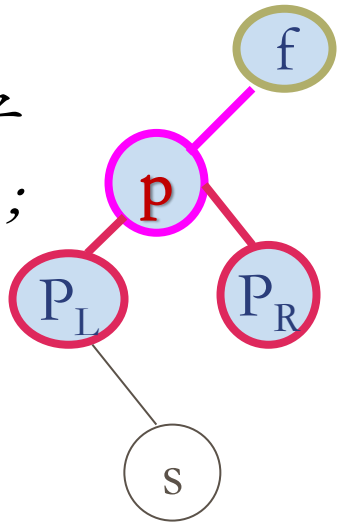
二叉树排序树

- **插入** 查找不成功，生成一个新结点s，插入到二叉排序树中；查找成功则返回。

- **删除** *p为叶子： 删除此结点时，直接修改*f指针域即可；
*p只有一棵子树（或左或右）：令P_L或P_R为*f的左孩子即可；
*p有两棵子树：

  法1：令p的左子树为f的左子树，p的右子树接为s的右子树； //即 $f_L=P_L$； $S_R=P_R$；

  法2：直接令s代替p，s的左子树接为P_L
  // s为p左子树最右下方的结点



回顾

平衡二叉树

- 平衡二叉树上所有结点的平衡因子只可能是-1,0,和1

1、首先确定属于何种类型的旋转；

- LL平衡旋转（单右旋）
- RR平衡旋转（单左旋）
- LR平衡旋转（先左后右）
- RL平衡旋转（先右后左）

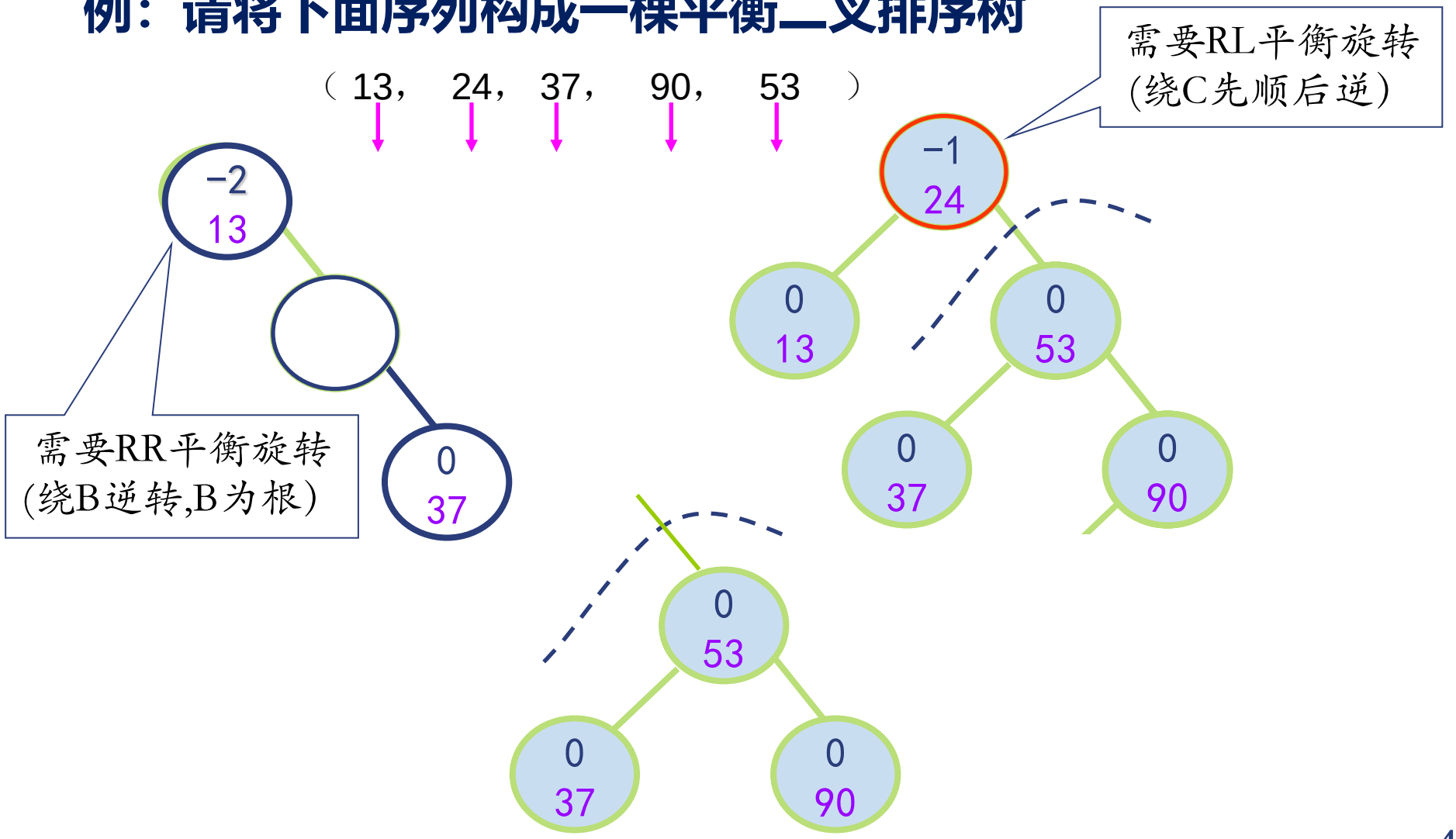
2、确定旋转轴

LL和RR型的旋转轴在沿着失衡路径上，以失去平衡点的直接后继结点作为旋转轴；LR和RL的旋转轴是沿着失衡路径上，失去平衡点的后两层结点作为旋转轴。

回顾

例：请将下面序列构成一棵平衡二叉排序树

（ 13， 24， 37， 90， 53 ）



回顾

平衡二叉树建立

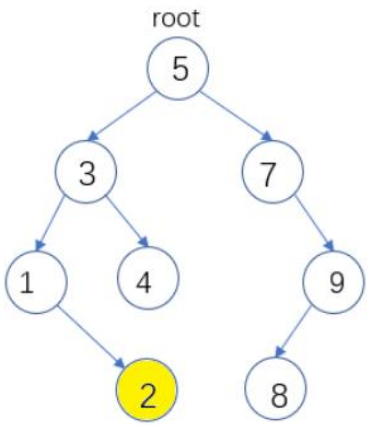
- 若新结点存储分配成功，返回1，否则返回0。
- 通过递归方式将新结点作为叶结点插入并逐层修改各结点的平衡因子。
- 在发现不平衡时立即执行相应的平衡化旋转操作，使得树中各结点重新平衡化。

```
Status InsertAVL(BSTree &T,ElemType e,bool &taller){
    if(T==NULL){//若为空树，插入一个数据元素为e的新节点作为BBST的根节点数的深度增1
        T=(BSTree)malloc(sizeof(BSTNode)); T->data=e;T->lchild=T->rchild=NULL; T->bf=EH;
        taller=true;
    }
    else{ //不为空树
        if(T->data==e){//待插关键字和BBST根节点关键字相同不进行插入
            taller=false; return 0;
        }
        else if(e<T->data){//小于且左子树中不存在关键字和待插关键字相同的的节点就插入
            if(InsertAVL(T->lchild,e,taller)==FAILED) return 0;//如插入不成功返回FAILED
            if(taller){//如果插入成功并且左子树长高了
                switch(T->bf){//BBST根节点的现状
                case LH://左子树高
                    LeftBalance(T);//左平衡处理
                    taller=false;//标记为未长高
                    break;
                case EH://平衡的
                    T->bf=LH;//标记为左高
                    taller=true;//标记为长高
                    break;
                case RH://右子树高
                    T->bf=EH;//标记为平衡
                    taller=false;//标记为未长高
                    break;
                }
            }
        }
```

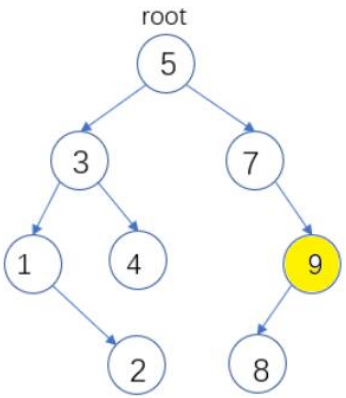
```
        else{//大于且右子树中不存在关键字和待插关键字相同的的节点就插入
            if(InsertAVL(T->rchild,e,taller)==FAILED) return FAILED;
                //如果再右子树中插入不成功返回不成功信息
            if(taller){//插入成功就判断右子树是否长高
                switch(T->bf){//BBST根的现状
                case LH://左子树高
                    T->bf=EH;
                    taller=false;
                    break;
                case EH://平衡
                    T->bf=RH;
                    taller=true;
                    break;
                case RH://右子树高
                    RightBalance(T);
                    taller=false;
                    break;
                }
            }
        }
    return 1;
}
```

9.2.2 平衡二叉树

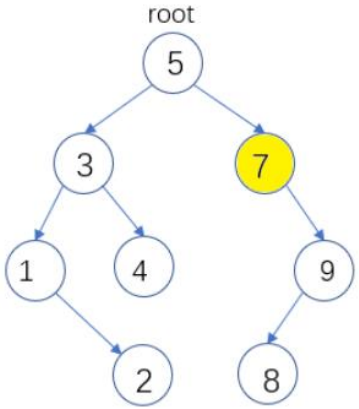
平衡二叉树删除



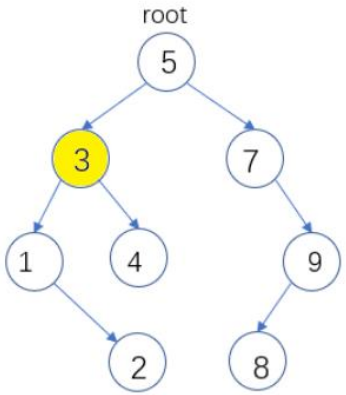
情况1：删除的点为叶节点
即没有左右子树
例如删除2这个节点



情况2：删除节点有左子树
没有右子树
例如删除9这个节点



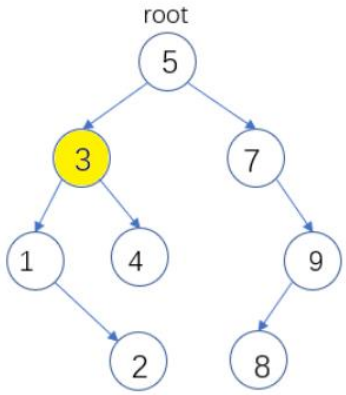
情况3：删除节点没有左子树有右子树
例如删除7这个节点



情况4：删除节点既有左子树，又有右子树
例如删除3这个节点

1. **删除节点为叶节点。**这种情况最简单，直接将其置空，释放，然后返回给父节点即可。
2. **删除节点有左子树没有右子树。** 先保存该节点的地址(方便释放)，将该节点直接等于左子树地址， 相当于该节点存的就是左子树的地址，将原来节点地址覆盖了。然后释放。
3. **删除节点有右子树没有左子树 。**与2处理相同，只是将左子树换为右子树。

9.2.4 平衡二叉树



情况4：删除节点既有左子树，又有右子树
例如删除3这个节点

4. 既有左子树又有右子树

① 找到左子树中的最大值，将值赋给给节点，然后将左子树最大值这个节点删除(删除可以用递归实现)

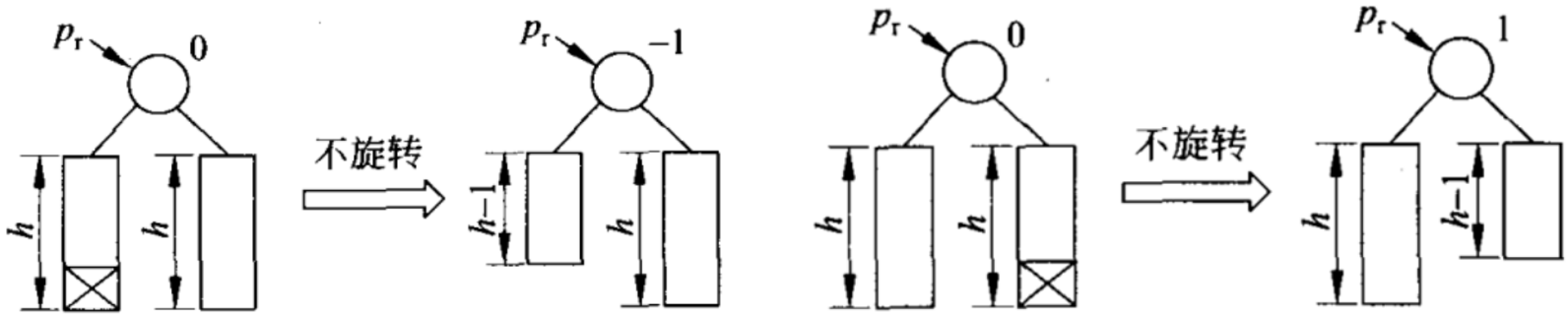
② 找到左子树中的最小值，将值赋给给节点，然后将右子树最小值这个节点删除

当然这样会有个弊端：当一直删除时，会导致树高度失衡，导致一边高，一边低，解决这样的办法可以删除左右子树最大最小节点交替实行。或者记录一高度，主要删除，左子树或者右子树高的那一边。

9.2.2 平衡二叉树

平衡二叉树删除调整

1. **祖先节点的平衡因子原来为 0，在它的左子树或者右子树被缩短后，它的平衡因子变为 -1 或者 1。**由于以该节点为根的子树的高度没有改变，则不需要往上更新了，删除结束。

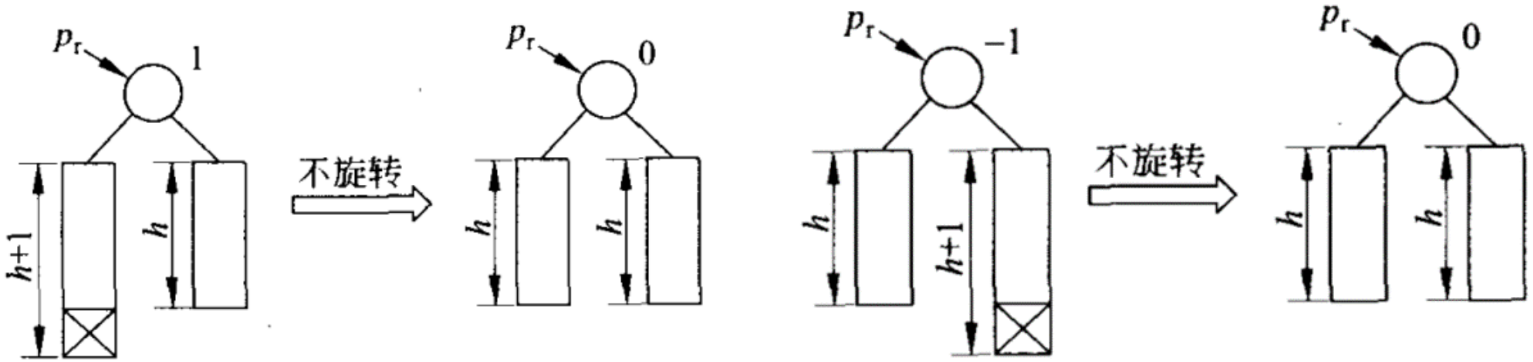


(a) p_r的左子树上删除　　(b) p_r 的右子树上删除

9.2.2 平衡二叉树

平衡二叉树删除调整

2. **祖先节点的平衡因子原来为 1 或者 -1，在它的较高的子树被缩短后，它的平衡因子改为 0。**此时以该节点为根的子树平衡，但其高度减 1，所以需要继续往上更新。



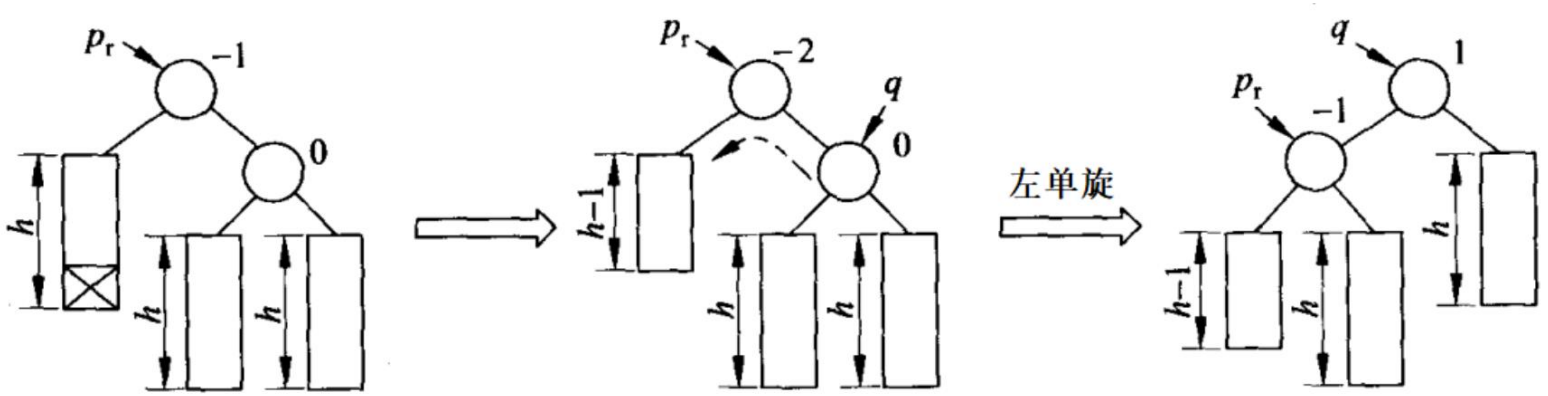
(a) p_r的左子树上删除　　(b) p_r的右子树上删除

9.2.2 平衡二叉树

平衡二叉树删除调整

3. **祖先节点的平衡因子原来为 -1 或者 1，在它的较矮的子树被缩短后，它的平衡因子变为 -2 或者 2。**此时以该节点为根的子树不平衡，需要进行平衡化旋转来恢复平衡。根据祖先节点较高的子树的根（该子树未被缩短）的平衡因子，有3 种平衡化操作：

   (1) 如果祖先节点较高的子树的根的平衡因子为 0，则执行一个单旋转来恢复子树的平衡。



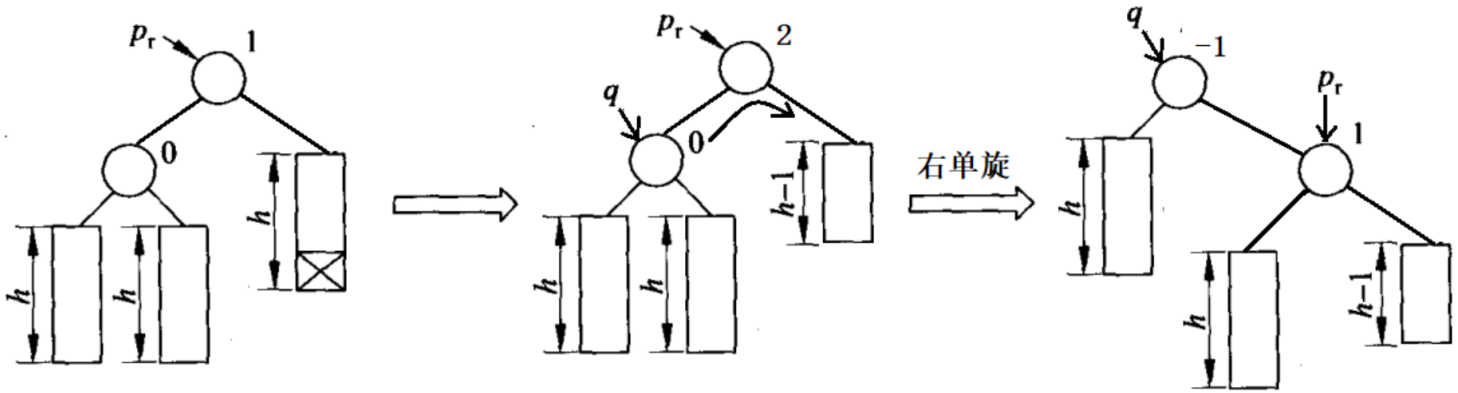
(a) p_r的左子树上删除

9.2.2 平衡二叉树

平衡二叉树删除调整

3. 祖先节点的平衡因子原来为 -1 或者 1，在它的较矮的子树被缩短后，它的平衡因子变为 -2 或者 2。此时以该节点为根的子树不平衡，需要进行平衡化旋转来恢复平衡。根据祖先节点较高的子树的根（该子树未被缩短）的平衡因子，有3 种平衡化操作：

(1) 如果祖先节点较高的子树的根的平衡因子为 0，则执行一个单旋转来恢复子树的平衡。



(b) p_r的右子树上删除

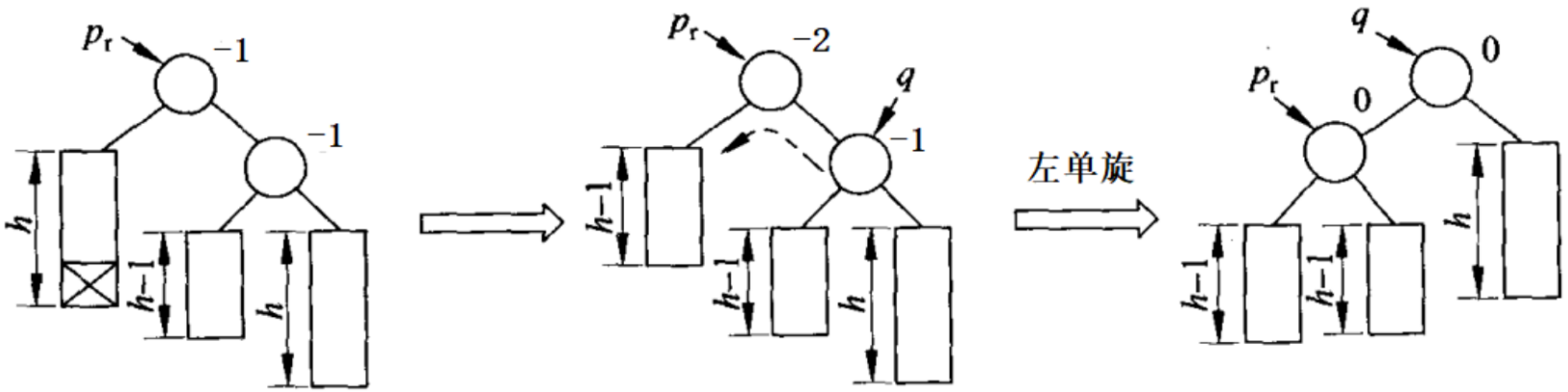
由于旋转平衡后以 *q 为根节点的子树的高度没有发生改变，所以不需要再往上更新了，删除结束。

9.2.2 平衡二叉树

平衡二叉树删除调整

3. 祖先节点的平衡因子原来为 -1 或者 1，在它的较矮的子树被缩短后，它的平衡因子变为 -2 或者 2。此时以该节点为根的子树不平衡，需要进行平衡化旋转来恢复平衡。根据祖先节点较高的子树的根（该子树未被缩短）的平衡因子，有3 种平衡化操作：

(2) 如果祖先节点较高的子树的根的平衡因子和祖先节点的平衡因子的正负号相同，则执行一个单旋转来恢复平衡。



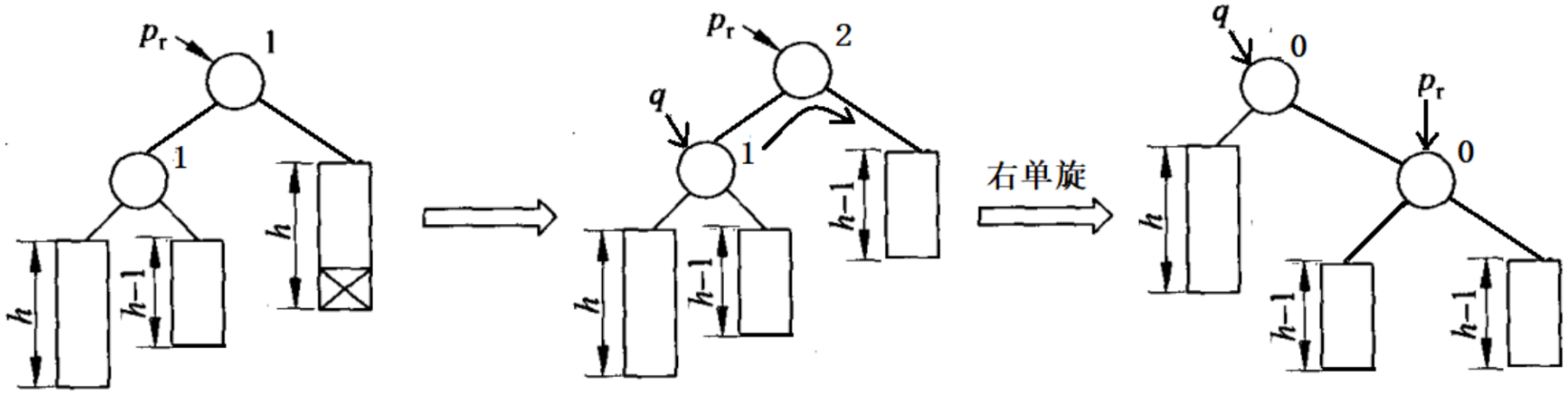
(a) p_r的左子树上删除

9.2.2 平衡二叉树

平衡二叉树删除调整

3. 祖先节点的平衡因子原来为 -1 或者 1，在它的较矮的子树被缩短后，它的平衡因子变为 -2 或者 2。此时以该节点为根的子树不平衡，需要进行平衡化旋转来恢复平衡。根据祖先节点较高的子树的根（该子树未被缩短）的平衡因子，有3 种平衡化操作：

(2) 如果祖先节点较高的子树的根的平衡因子和祖先节点的平衡因子的正负号相同，则执行一个单旋转来恢复平衡。



(b) p_r的右子树上删除

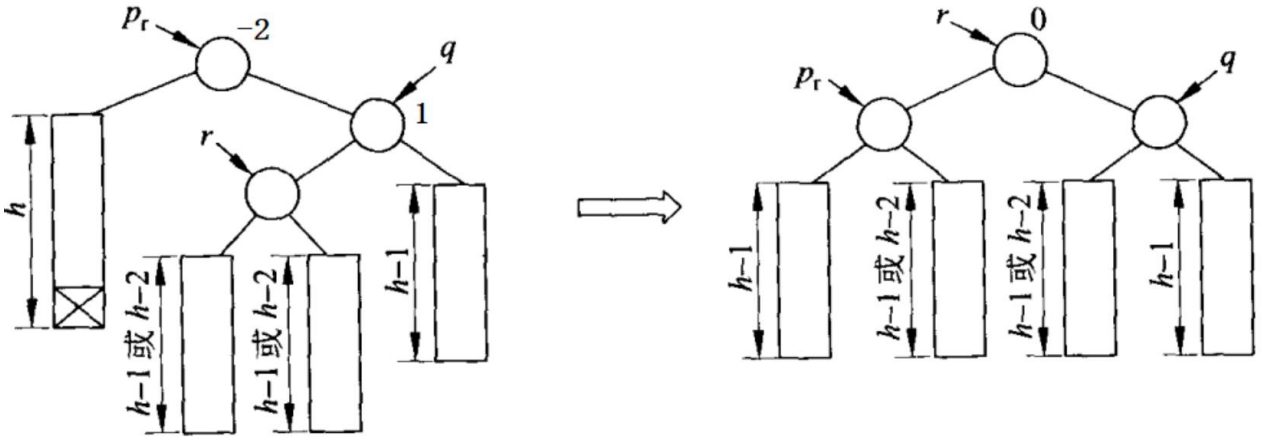
由于经过旋转平衡旋转后以 *q 为根节点的子树的高度降低了 1，所以需要继续往上更新。

9.2.2 平衡二叉树

平衡二叉树删除调整

3. **祖先节点的平衡因子原来为 -1 或者 1，在它的较矮的子树被缩短后，它的平衡因子变为 -2 或者 2。**此时以该节点为根的子树不平衡，需要进行平衡化旋转来恢复平衡。根据祖先节点较高的子树的根（该子树未被缩短）的平衡因子，有3 种平衡化操作：

   (3) 如果祖先节点较高的子树的根的平衡因子和祖先节点的平衡因子的正负号相反，则执行一个双旋转来恢复平衡。



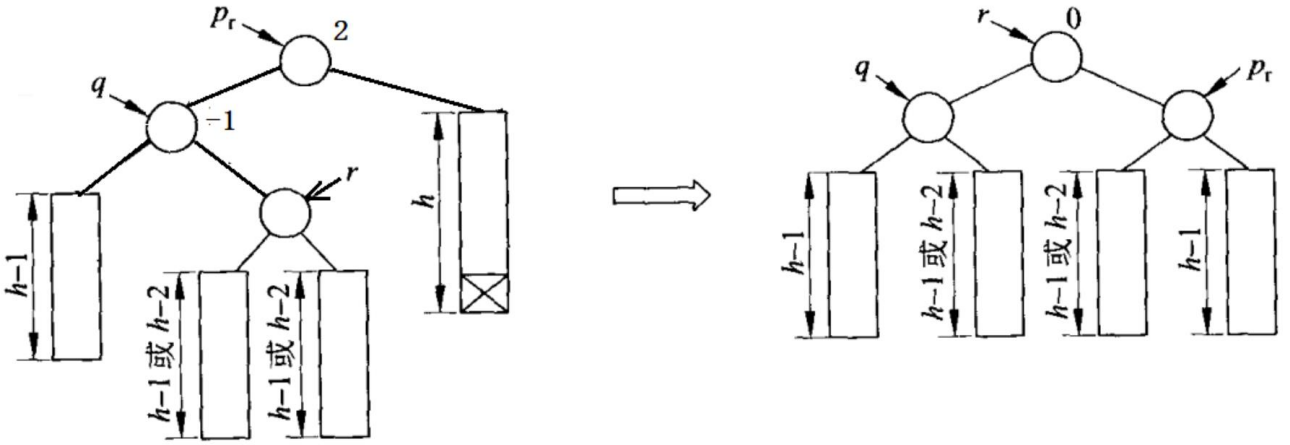
(a) p_r的左子树上删除

9.2.2 平衡二叉树

平衡二叉树删除调整

3. 祖先节点的平衡因子原来为 -1 或者 1，在它的较矮的子树被缩短后，它的平衡因子变为 -2 或者 2。此时以该节点为根的子树不平衡，需要进行平衡化旋转来恢复平衡。根据祖先节点较高的子树的根（该子树未被缩短）的平衡因子，有3 种平衡化操作：

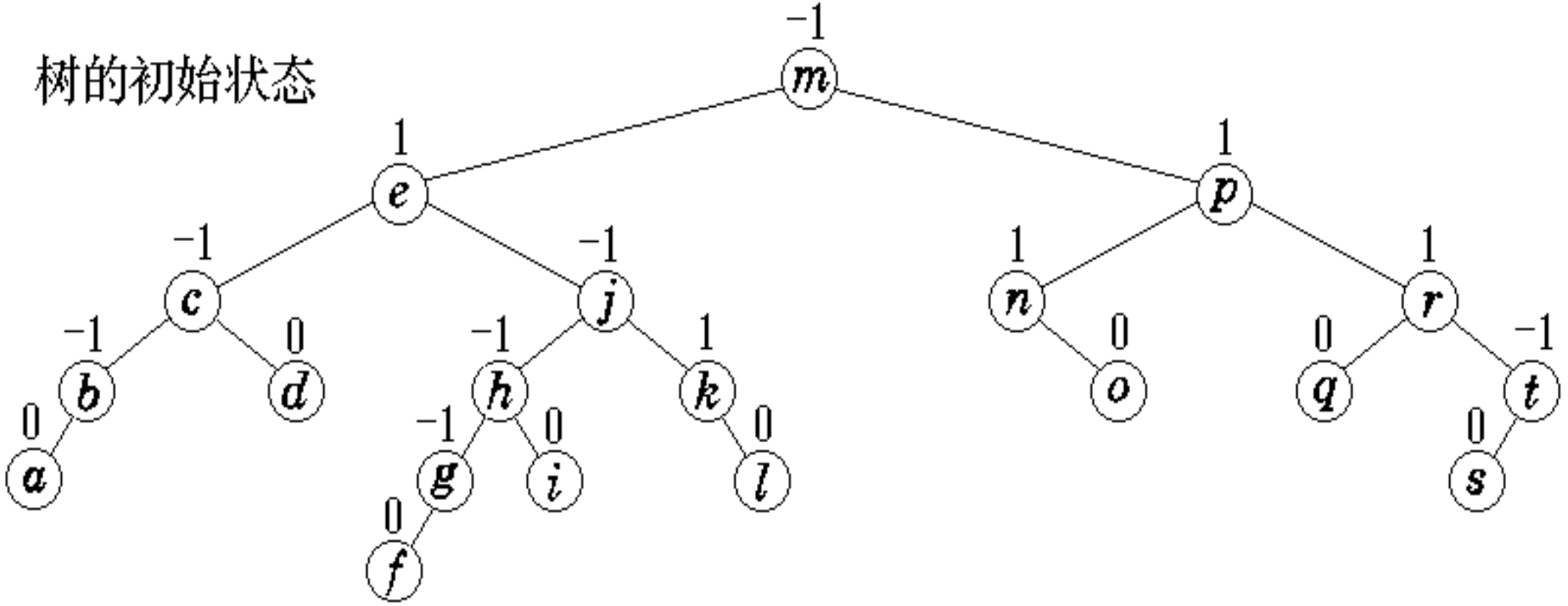
(3) 如果祖先节点较高的子树的根的平衡因子和祖先节点的平衡因子的正负号相反，则执行一个双旋转来恢复平衡。

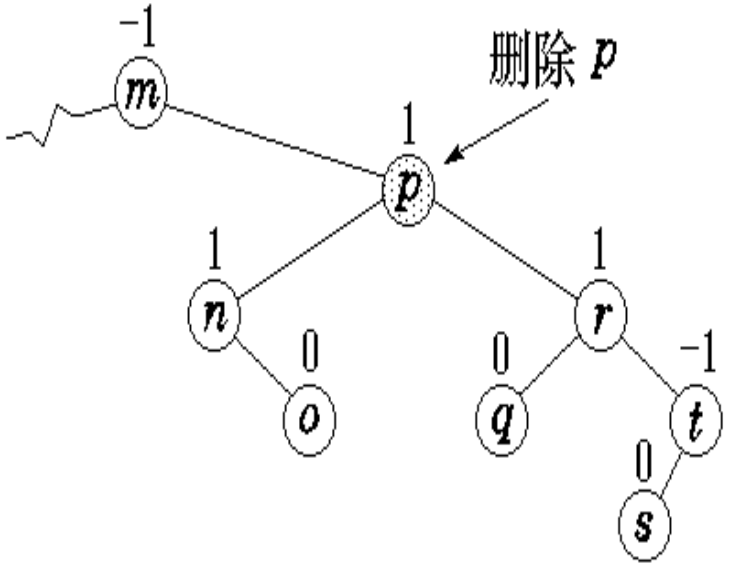


(b) p_r的右子树上删除

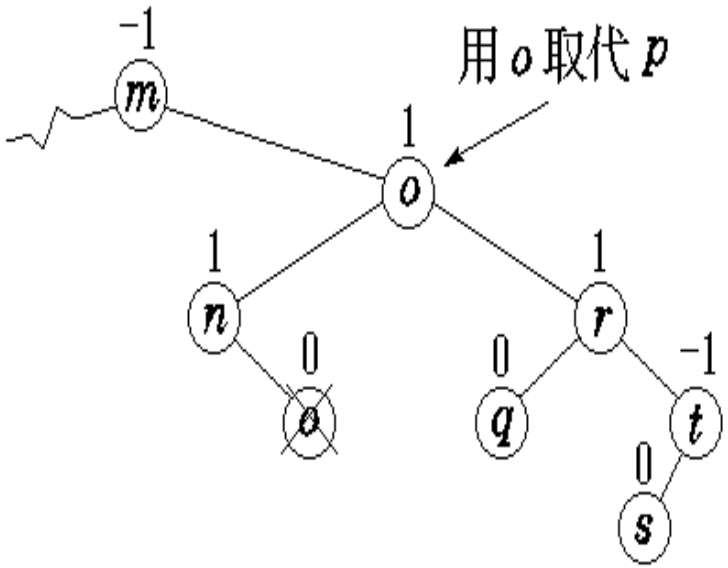
由于经过平衡化处理后以 *r 为根的子树的高度降低了 1，所以还需要继续往上更新。

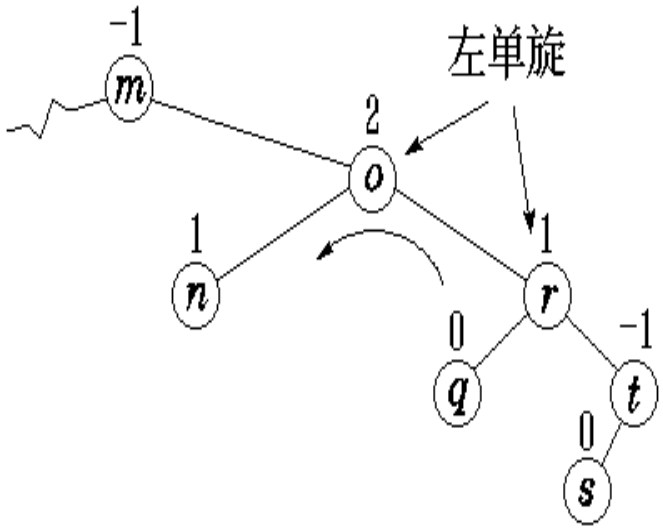
树的初始状态



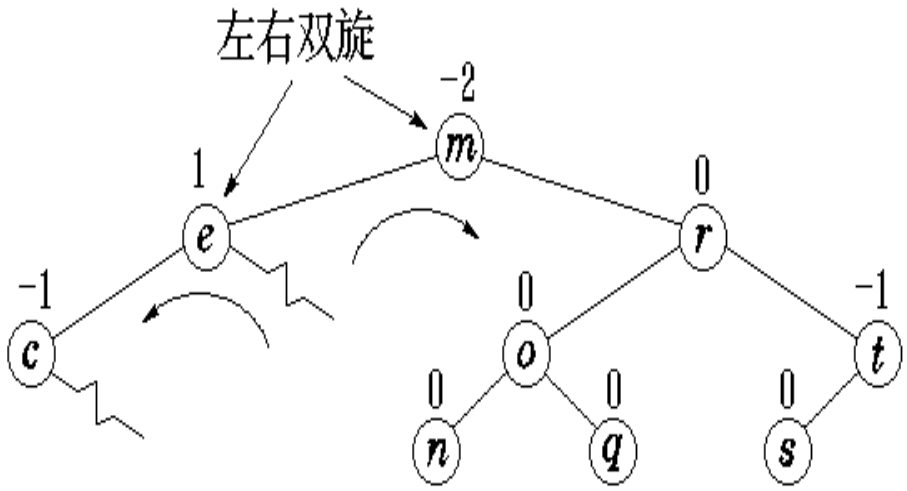


寻找 p 的中序下的直接前驱 o，用 o 取代 p，删除 o，平衡旋转

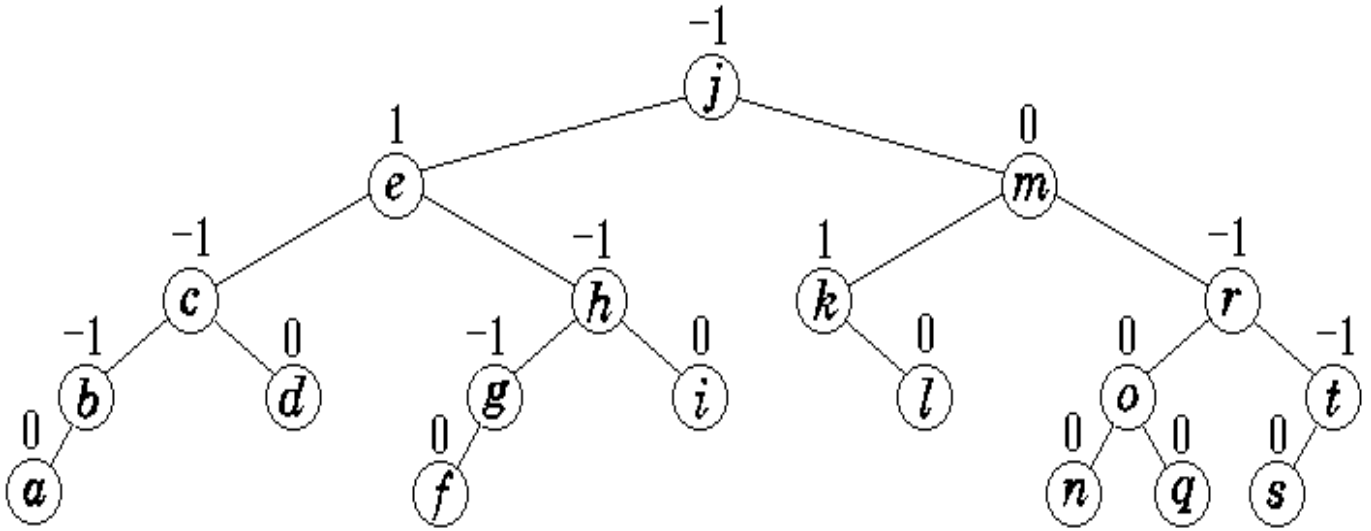




以 r 为旋转轴作左单旋，子树的高度减 1 m 发生不平衡



首先以 j 为旋转轴作左单旋，再以 j 为旋转轴作右单旋，让 e 成为 j 的左子女，m 成为 j 的右子女。树的高度减 1



9.2 动态查找表

典型的动态表————二叉排序树

一、二叉排序树的定义

二、二叉排序树的插入与删除

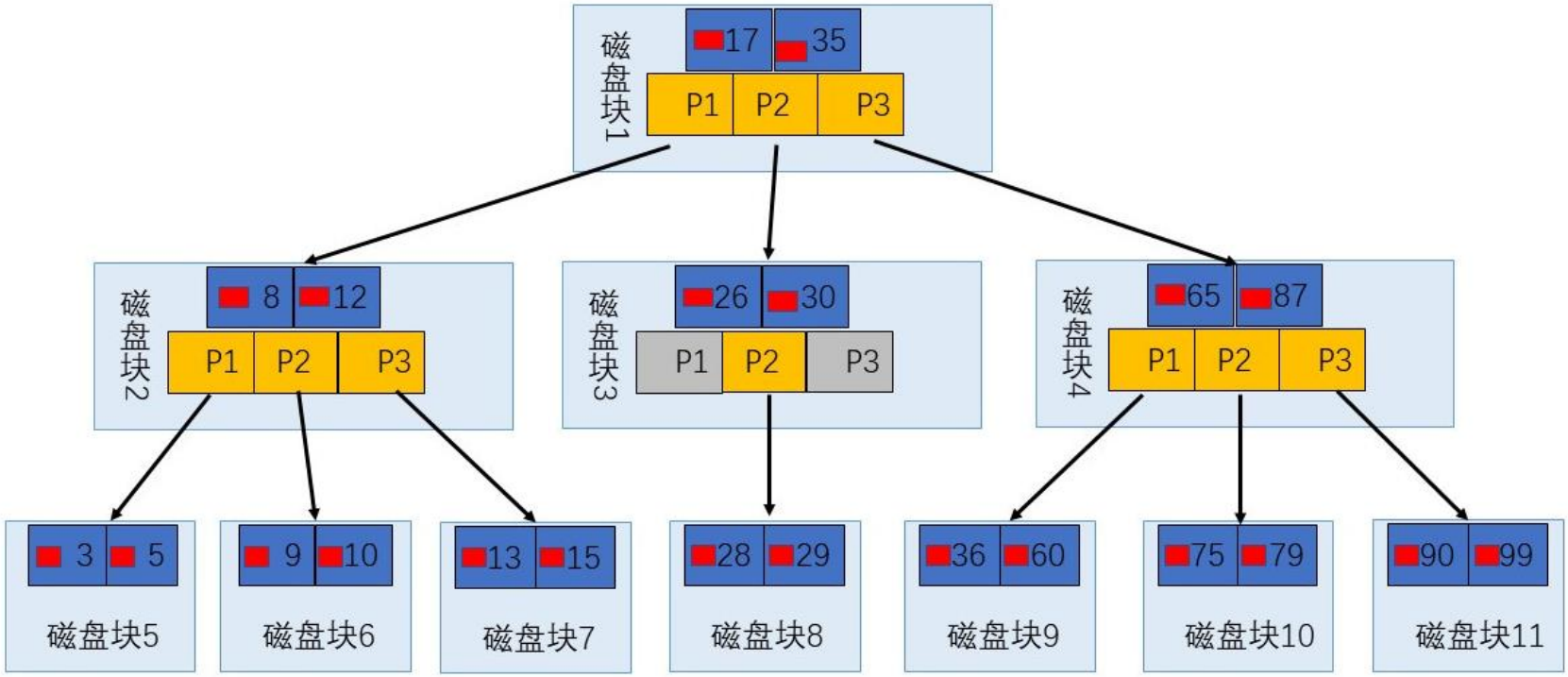
三、二叉排序树的查找分析

四、平衡二叉树(难点)

五、B-树和B+树

9.2.3 B-树和B+树

多路查找树 (Muit-Way Search Tree)，其每一个**结点的孩子数**可以**多于两**个且**每一个结点处可以存储多个元素**。



9.2.3 B-树和B+树

1、B-树的定义

B-树是一种平衡的**多路查找树**。一棵m阶的B-树，或为空树，或为满足下列特性的m叉树：

①树中每个结点至多有m棵子树，m-1个关键字；

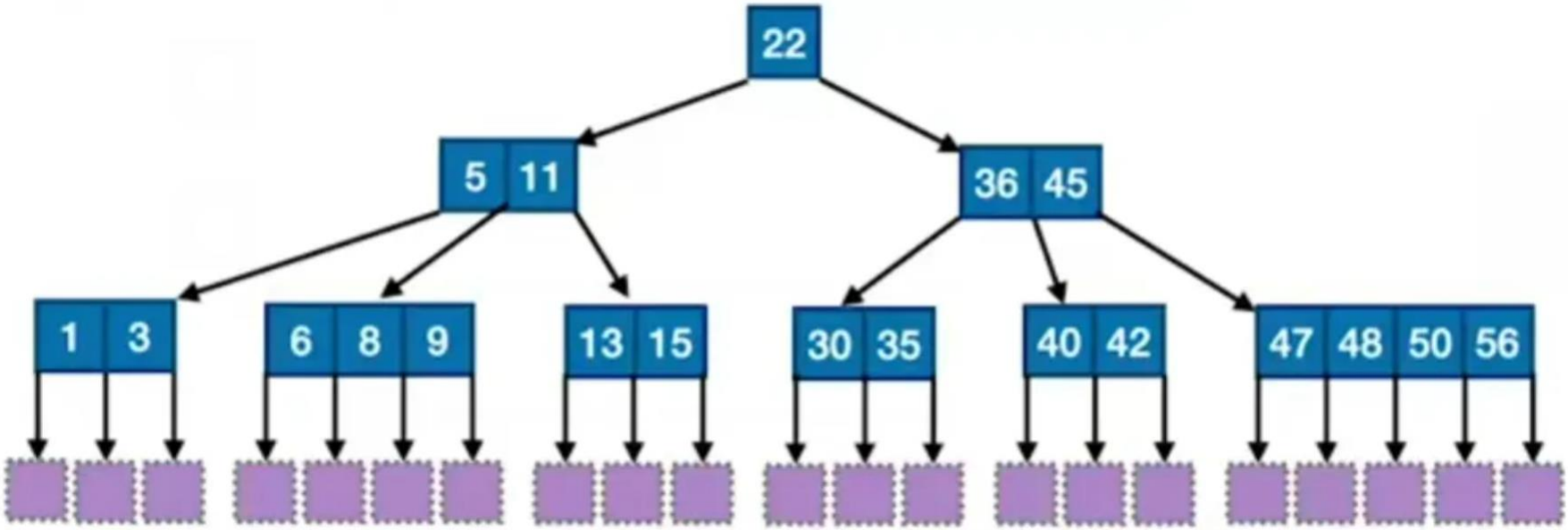
②若根结点不是叶子结点，则**至少有两棵子树**；

③除根之外的所有非终端结点至少有$\lceil m/2 \rceil$ 棵子树，至少有$\lceil m/2 \rceil$-1 个关键字；

9.2.3 B-树和B+树

1、B-树的定义

5阶B树



9.2.3 B-树和B+树

1、B-树的定义

④所有的非终端结点中包含下列信息数据

$$(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$$

其中：$K_i(i = 1, \ldots, n)$为关键字，且$K_i < K_{i+1}(i = 1, \ldots, n-1)$

，$\left(\left\lceil \frac{m}{2} \right\rceil - 1 \le n \le m - 1\right)$;

$A_i(i = 1, \ldots, n)$为指向子树根结点的指针，且指针A_{i-1}所指子树中所有结点的关键字均小于$K_i(i = 1, \ldots, n)$，A_i所指子树中所有结点的关键字均大于K_i，n为关键字的个数（或n＋1为子树个数）。

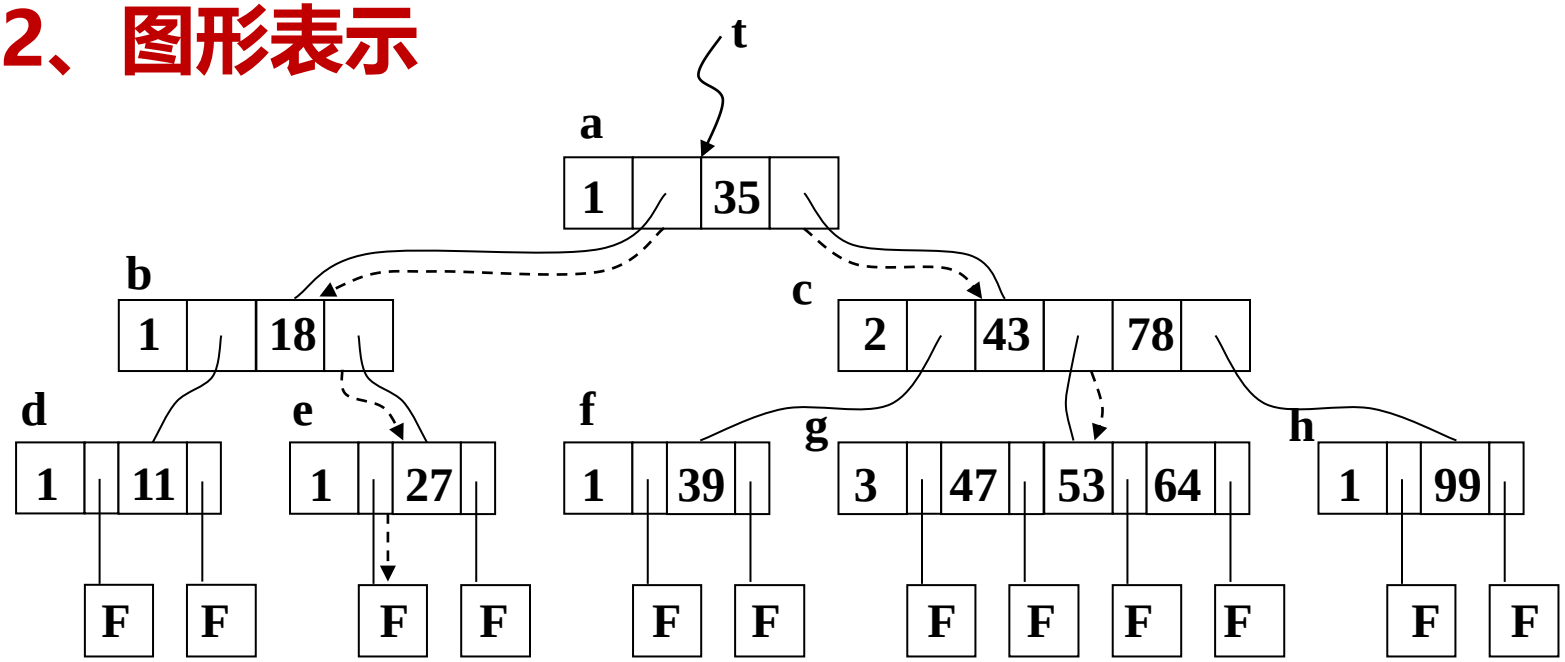
9.2.3 B-树和B+树

1、B-树的定义

⑤所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

9.2.3 B-树和B+树

2、图形表示



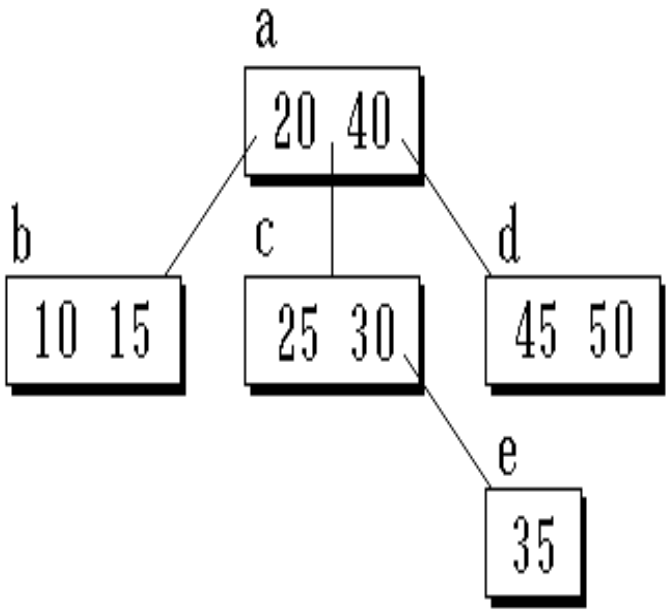
一棵4阶的B-树

结点中关键字个数为1<=n<=3

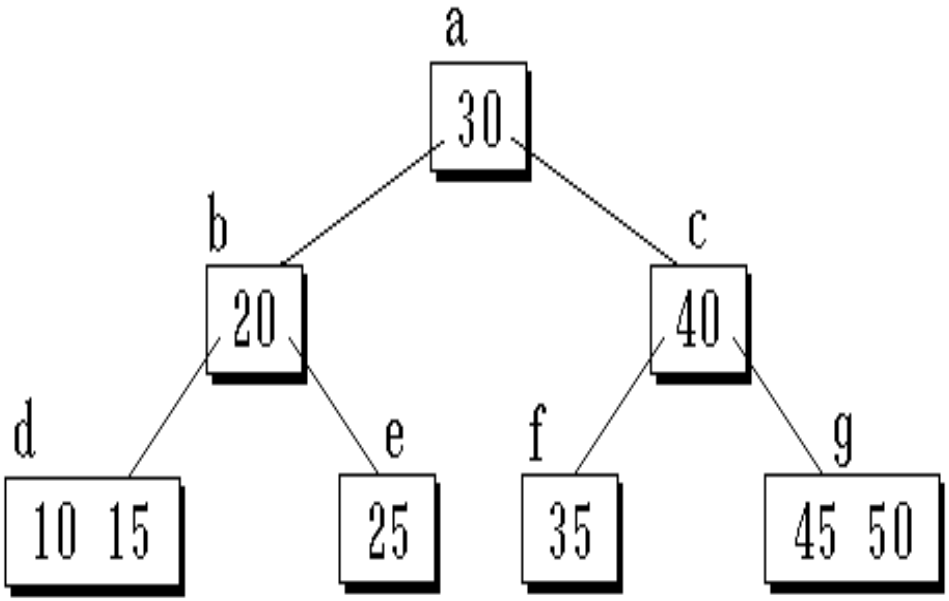
子树数量范围为2<=k<=4

9.2.3 B-树和B+树

2、图形表示



非B-树



B-树

9.2.3 B-树和B+树

3、B-树的类型定义

```
#define M 3  //B-树的阶，根据实际情况需要设定
typedef struct BTNode{
        int keynum;  //结点中的关键字个数
        struct BTNode *parent;  //指向双亲结点
        KeyType key[M+1];//关键字向量，0号单元未用
        struct BTNode *ptr[M+1]  //子树指针向量
        Record  *recptr[M+1]; /*记录指针向量，0号单元
      未用*/
 }BTNode, *Btree;
```

9.2.3 B-树和B+树

3、B-树的类型定义

B-树查询结构体

```
typedef struct {
  BTNode          *pt;     //指向找到的结点
  int             i;  //1..m-1，在结点中的关键字序号
  int             tag;     //1:查找成功，0:查找失败
} Result;                  //B-树的查找结果类型
```

9.2.3 B-树和B+树

4、B-树的查找

从根结点出发，沿指针**搜索结点**和**在结点内进行**顺序（或折半）**查找**两个过程交叉进行。

若**查找成功**，则**返回指向**被查关键字所在**结点的指针**和**关键字在结点中的位置**；

若**查找不成功**，则**返回插入位置**。

9.2.3 B-树和B+树

- 查找过程如下：

  将根节点作为当前节点 //常驻RAM

  只要当前节点非外部节点

    在当前节点中顺序或折半查找 //RAM内部

    若找到目标关键码，则**返回查找成功**

    否则 //止于某一对下层引用

      沿引用，转至对应子树

      将其根节点读入内存

      //I/O操作，最为耗时

      更新当前节点

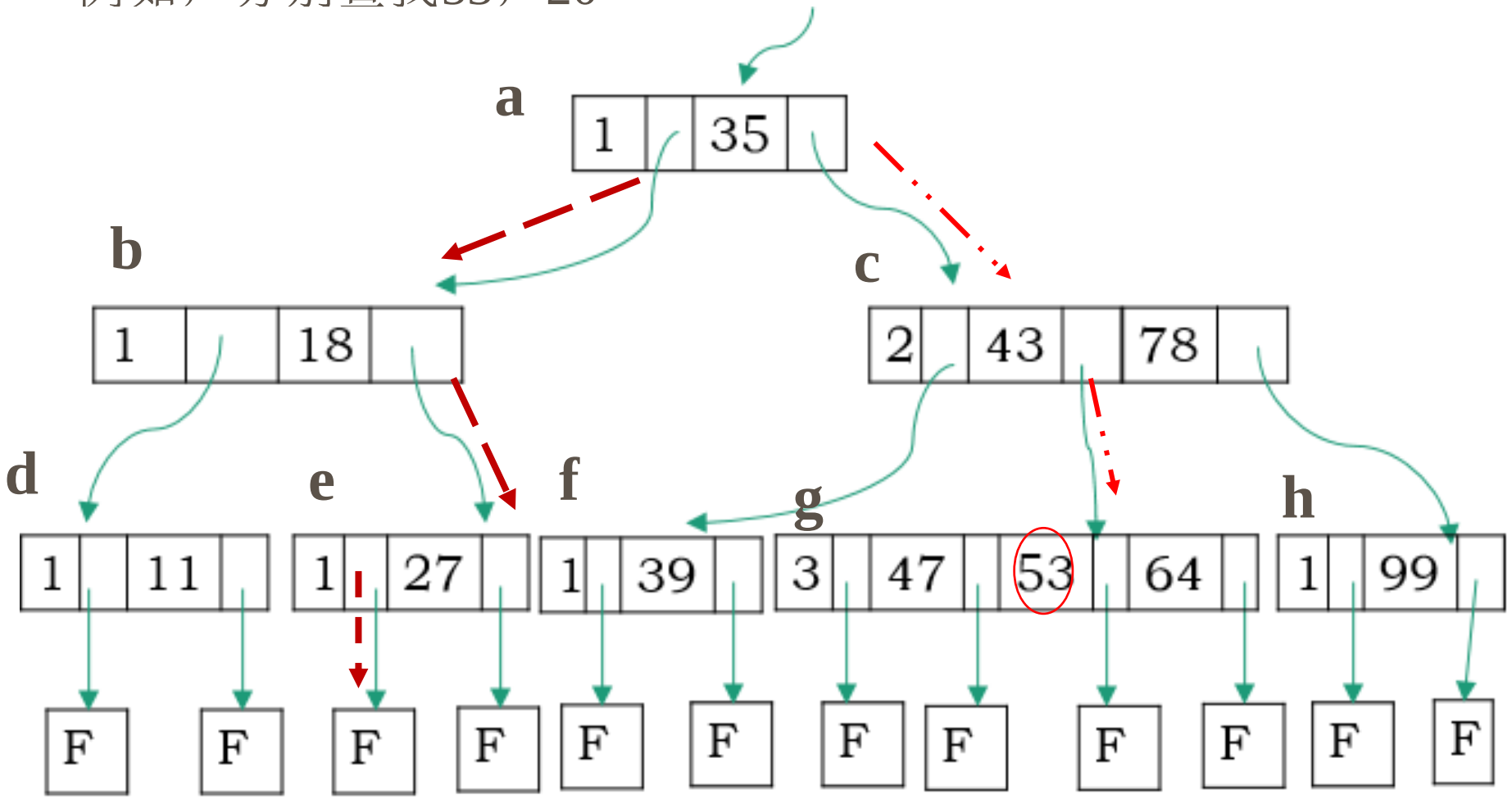
  返回查找失败

9.2.3 B-树和B+树

```
Result SearchBTree (BTree T, KeyType K) {
    //在m阶B-树T上查找关键字K，返回结果(pt, i , tag)。若查找成功，
    //则特征值tag＝1，指针pt所指结点中第i个关键字等于K；否则
    //特征值tag＝0，等于K的关键字应插入在指针pt所指结点中第i和第i＋1个关键字之间
    p = T;    q = NULL;  //初始化，p指向待查结点，q指向p的双亲
    found = FALSE; i = 0;
    while (p && !found) {
        i = Search (p, K);
        /*在p->key[1..keynum]中查找，i使得: p->key[i] < K < p->key[i + 1]*/
        if (i > 0 && p->key[i] = = K)
            found = TRUE;   //找到待查关键字
        else {
            q = p;
            p = p->ptr[i];
        }
    }
    if (found)
        return  (p, i, 1);       //查找成功
    else
        return  (q, i, 0);       //查找不成功，返回K的插入位置信息
} // SearchBTree
```

9.2.3 B-树和B+树

例如，分别查找53，20



9.2.3 B-树和B+树

4、B-树的查找

查找分析

在B-树上进行查找包含两种基本操作：

① 在B-树中找结点（操作在磁盘上进行）；

② 在结点中找关键字（操作在内存中进行）。

在**磁盘上进行查找的次数**、即待查关键字所在结点在B-树上的层次数，是决定B-树查找效率的首要因素。

9.2.3 B-树和B+树

4、B-树的查找

查找分析

- 若定义在 m 阶B-树中，失败结点位于第 h +1层。则在这棵B-树中关键字个数 N 最小能达到多少?

  从B-树的定义知:

  第一层 至少 1 个结点

  第二层 至少 2 个结点

  第三层 至少 2 ⌈m / 2⌉ 个结点

  第四层 至少 2 $\lceil m / 2 \rceil^2$ 个结点

  ……

  第 h 层 至少有2 $\lceil m / 2 \rceil^{h-2}$ 个结点。所有这些结点都不是失败结点。

  第 h+1 层 至少有2 $\lceil m / 2 \rceil^{h-1}$个结点都是失败结点

9.2.3 B-树和B+树

4、B-树的查找

查找分析

- 而l＋l层的结点为叶子结点。若m阶B-树中具有N个关键字，则叶子结点即查找不成功的结点为N＋l，因此：

$$N+1 \geq 2*(\lceil m/2 \rceil)^{l-1}$$

$$l \leqslant \log_{\lceil \frac{m}{2} \rceil}\left(\frac{n+1}{2}\right)+1$$

含有n个关键字的B-树上查找时，从根结点到关键字结点的路径上涉及的结点数不超过上式。

9.2.3 B-树和B+树

5、B-树的插入

①节点插入

由于B-树结点中的关键字个数必须 $\geq \lceil m/2 \rceil - 1$ 。

因此，每次插入一个关键字不是在树中添加一个叶子结点，而是首先在最低层的某个非终端结点中添加一个关键字，若该结点的关键字个数不超过**m-1**，则插入完成，否则要产生结点的“分裂”。

9.2.3 B-树和B+树

5、B-树的插入

②结点“分裂”：

设结点 A 中已经有 m-1 个关键字，当再插入一个关键字后，结点中的状态为$(m, A_0, K_1, A_1, K_2, A_2, \cdots\cdots, K_m, A_m)$，其中 $K_i < K_{i+1}, 1 \le i < m$

这时必须把结点 p 分裂成两个结点 p 和q，它们包含的信息分别为：

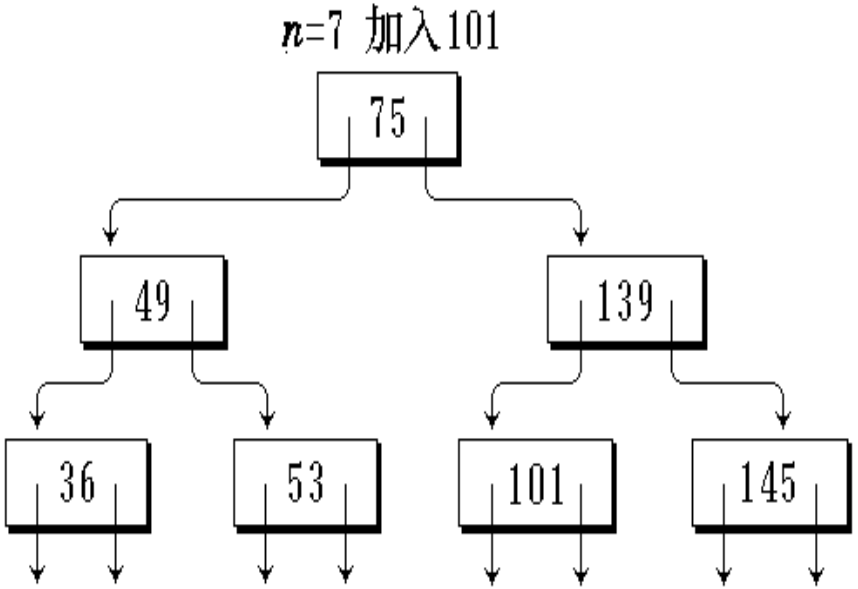
p：$(\lceil m/2 \rceil - 1,\ A_0,\ K_1,\ A_1,\ \cdots\cdots,\ K_{\lceil m/2 \rceil - 1},\ A_{\lceil m/2 \rceil - 1})$

q：$(m - \lceil m/2 \rceil,\ A_{\lceil m/2 \rceil},\ K_{\lceil m/2 \rceil + 1},\ A_{\lceil m/2 \rceil + 1},\ \cdots\cdots,\ K_m,\ A_m)$

位于中间的关键字 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组$(K_{\lceil m/2 \rceil},\ q)$，**插入到这两个结点的双亲结点中去**。

9.2.3 B-树和B+树

5、B-树的插入

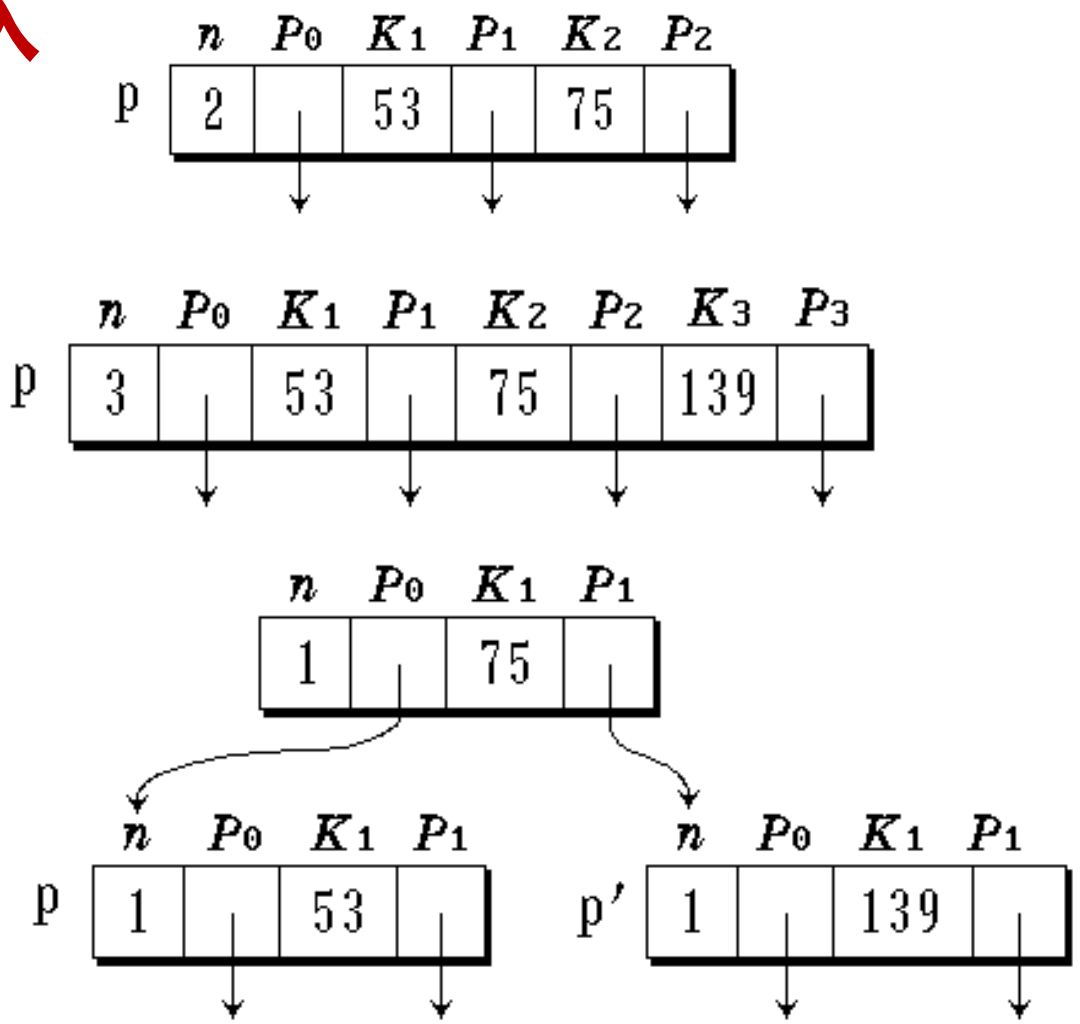


若设B-树的高度为h，那么在自顶向下查找到叶结点的过程中需要进行h次读盘。

- 在插入新关键字时，需要自底向上分裂结点，最坏情况下从被插关键字所在叶结点到根的路径上的所有结点都要分裂。

9.2.3 B-树和B+树

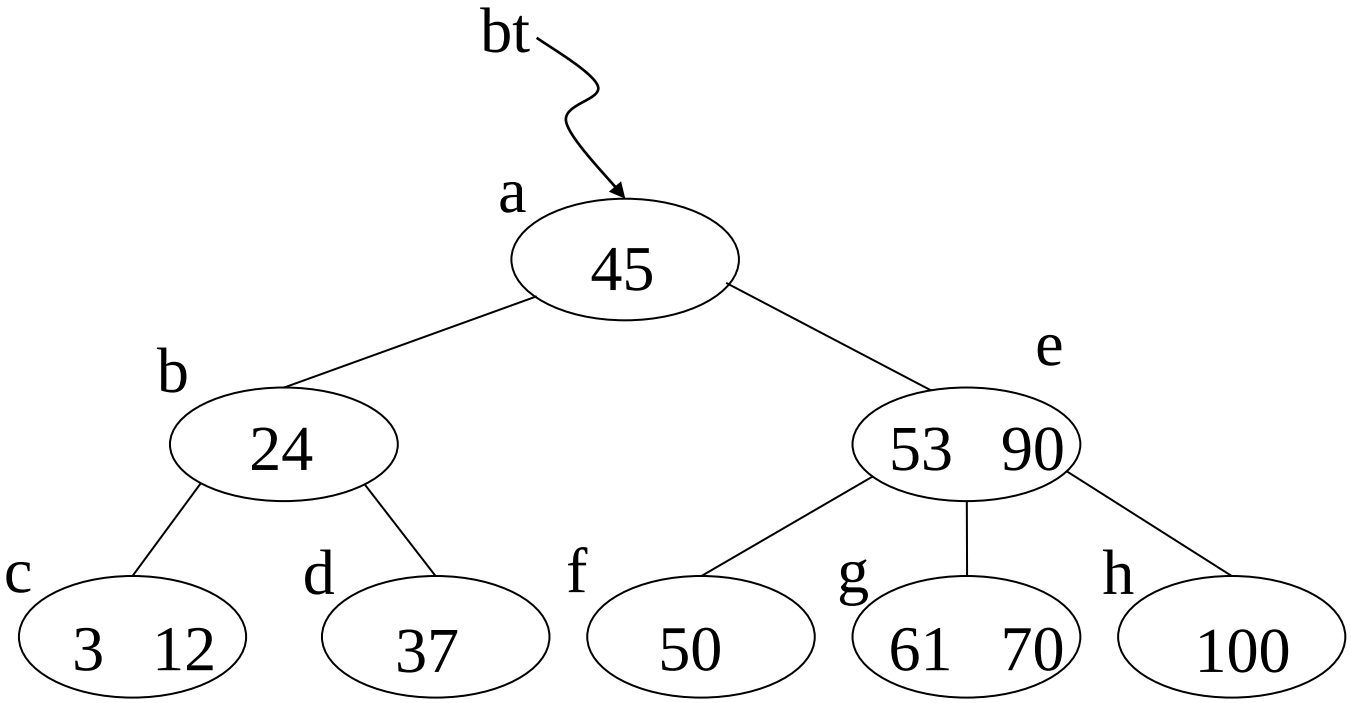
5、B-树的插入



9.2.3 B-树和B+树

5、B-树的插入

例如，图2所示为3阶B-树（图中略去F结点，即叶子结点），假设需一次插入关键字30，26，85和7。

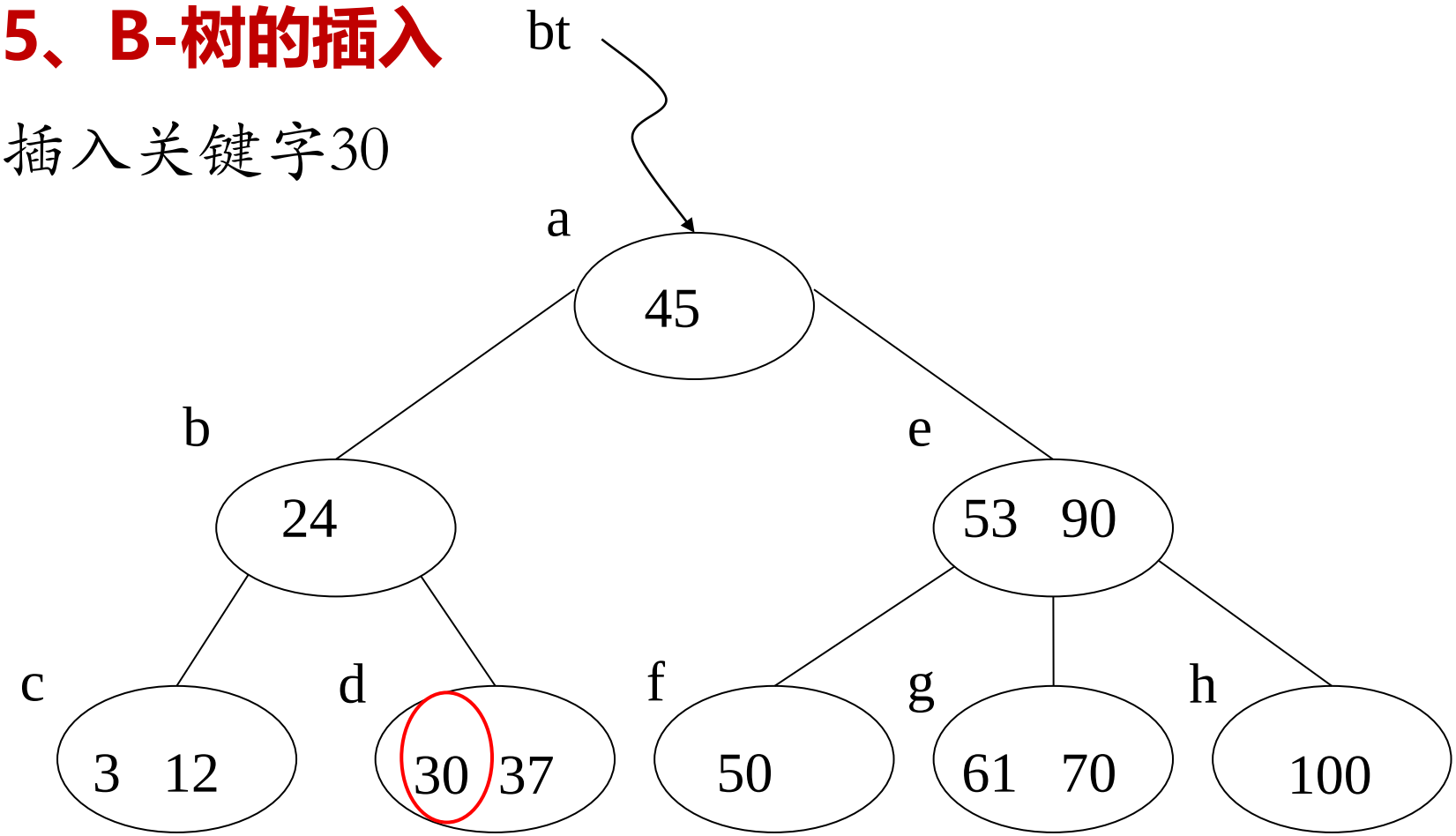


一棵2-3树

9.2.3 B-树和B+树

5、B-树的插入

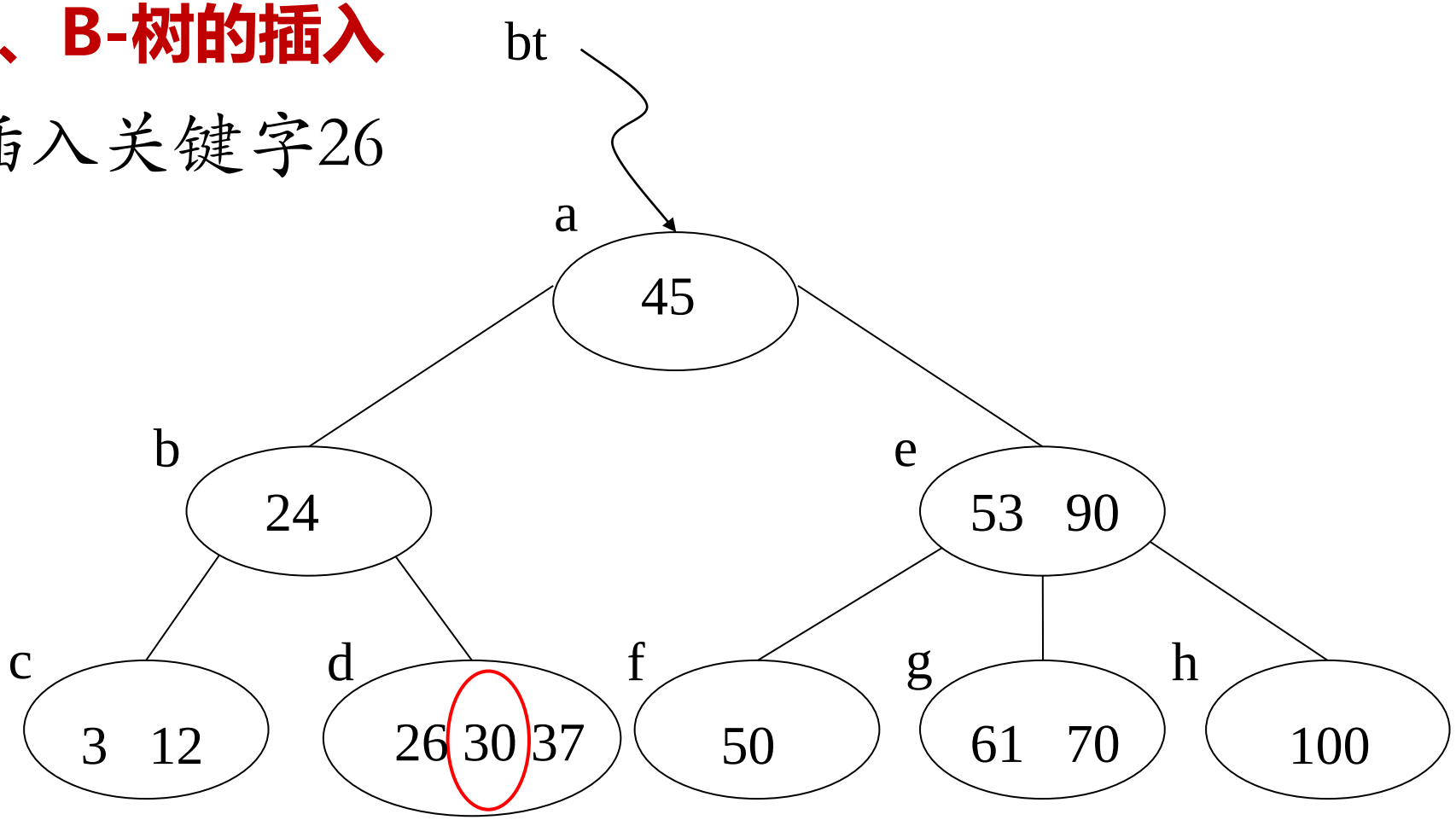
插入关键字30



9.2.3 B-树和B+树

5、B-树的插入

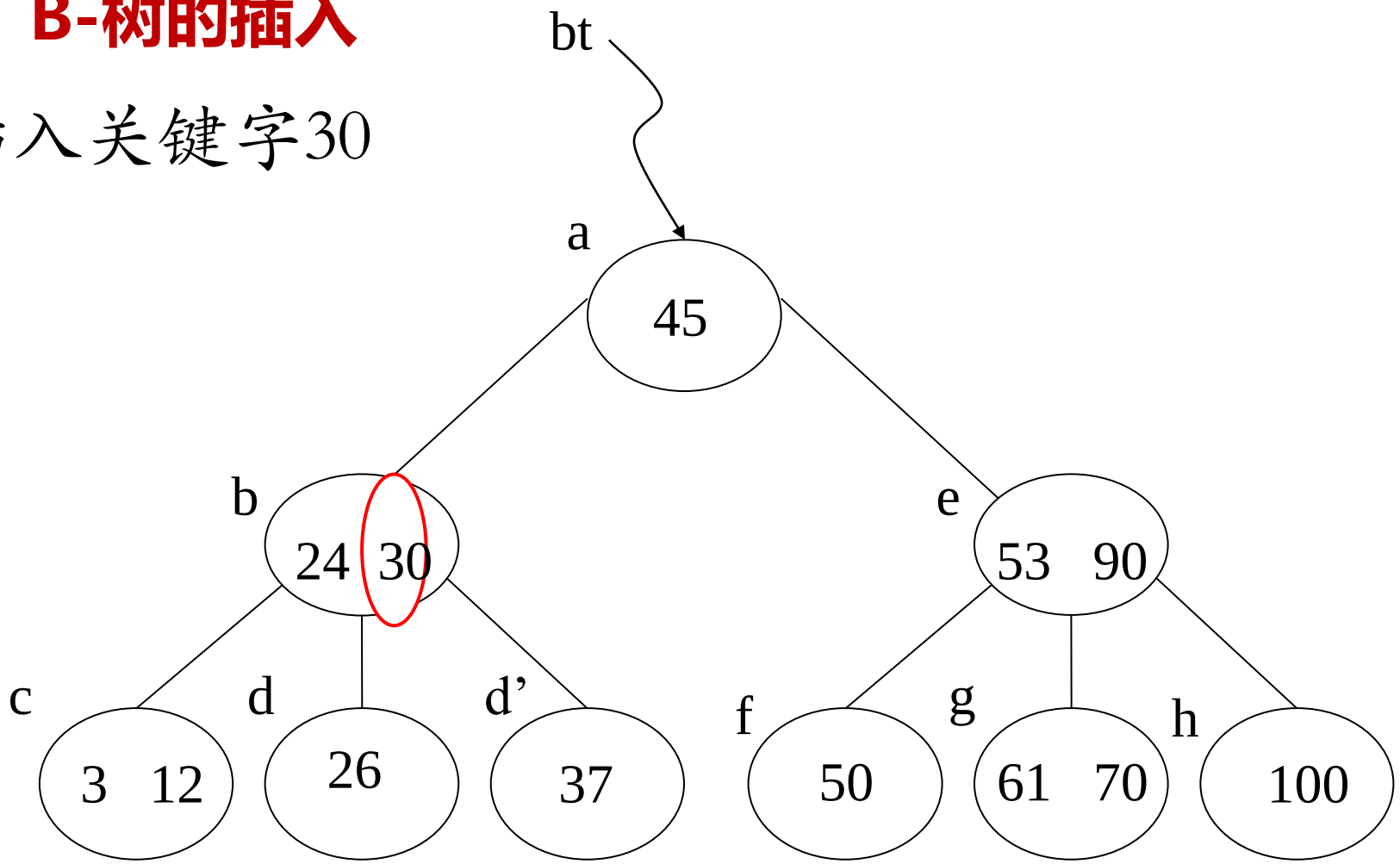
插入关键字26



9.2.3 B-树和B+树

5、B-树的插入

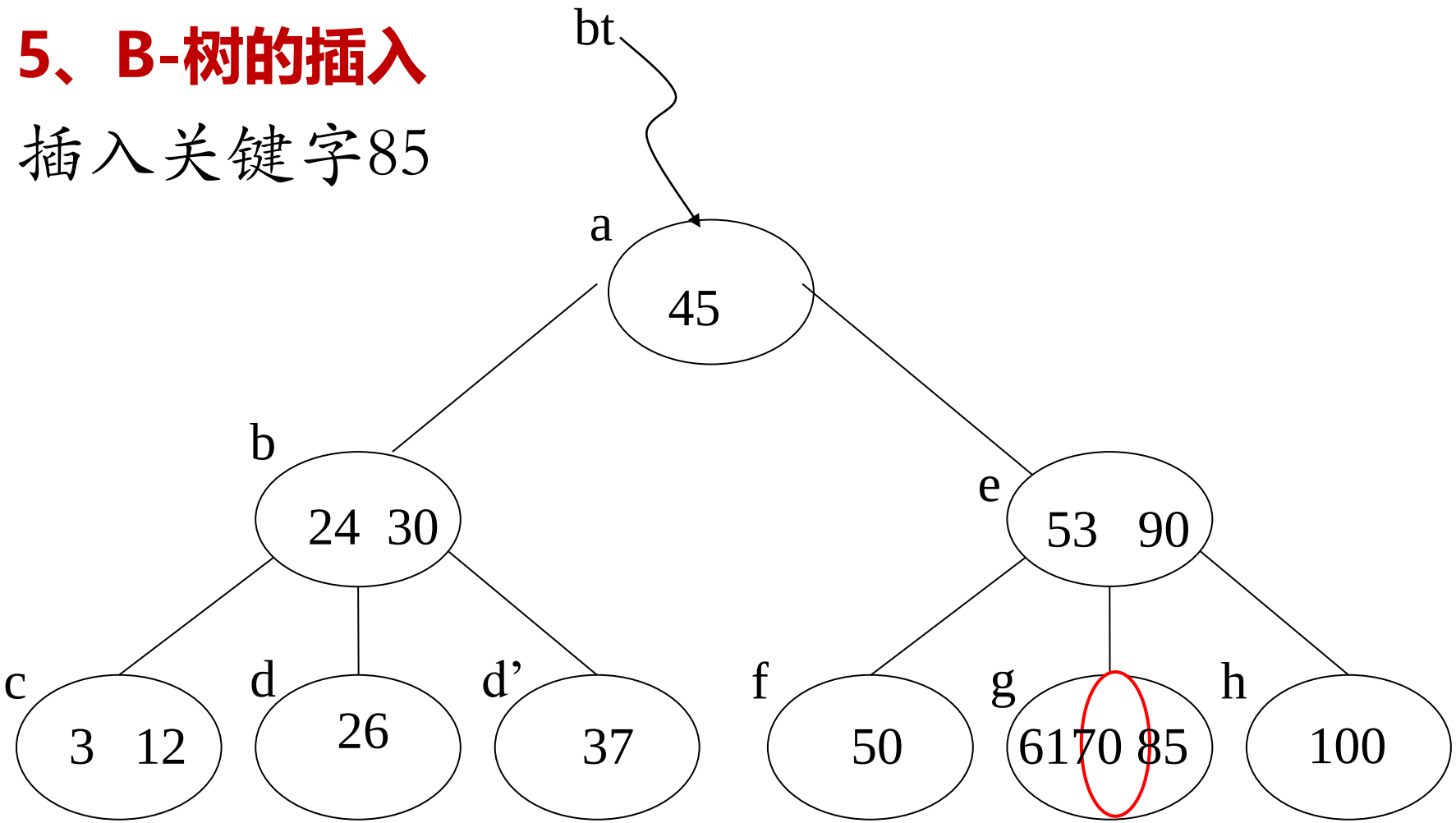
插入关键字30



9.2.3 B-树和B+树

5、B-树的插入

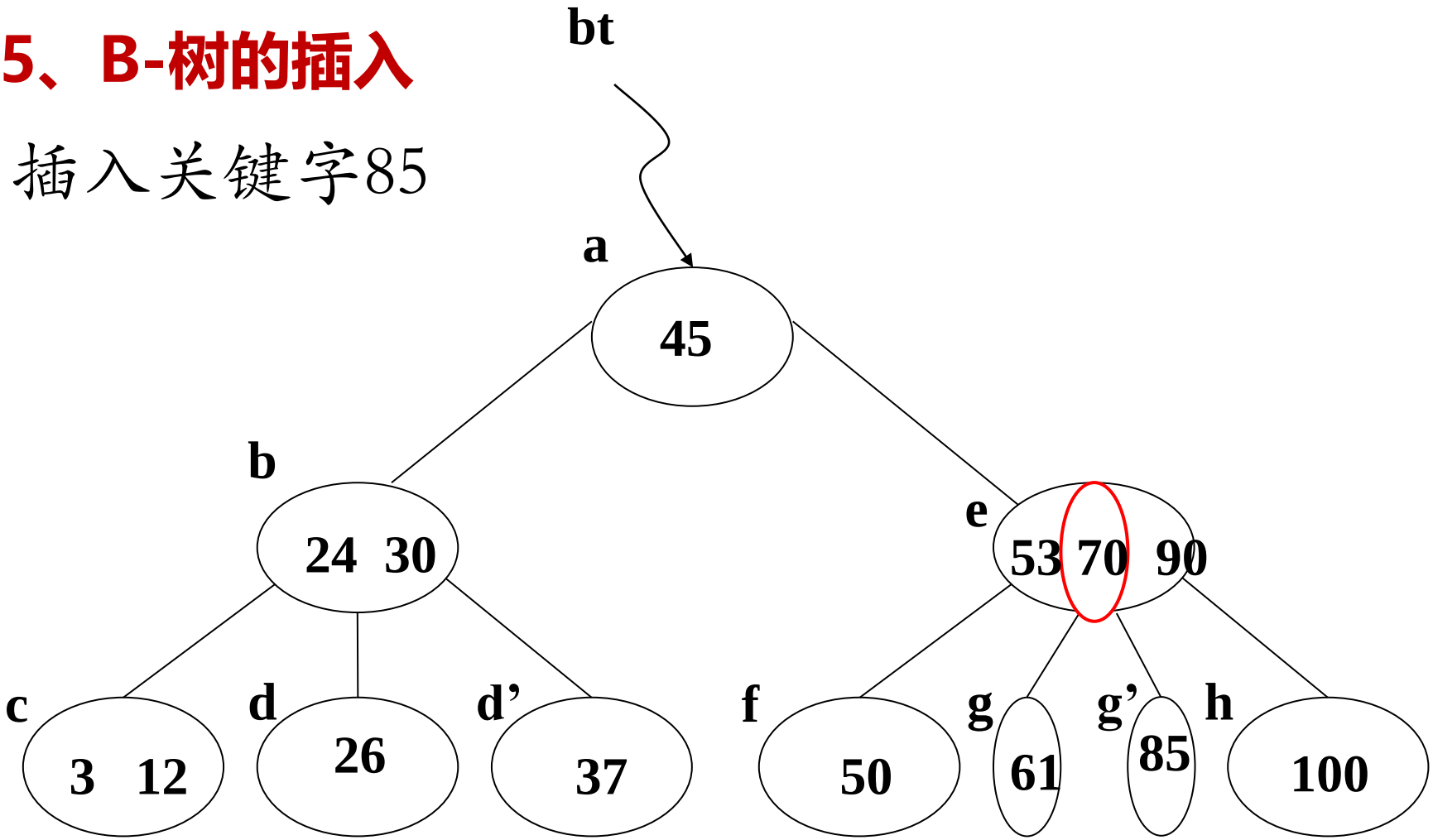
插入关键字85



9.2.3 B-树和B+树

5、B-树的插入

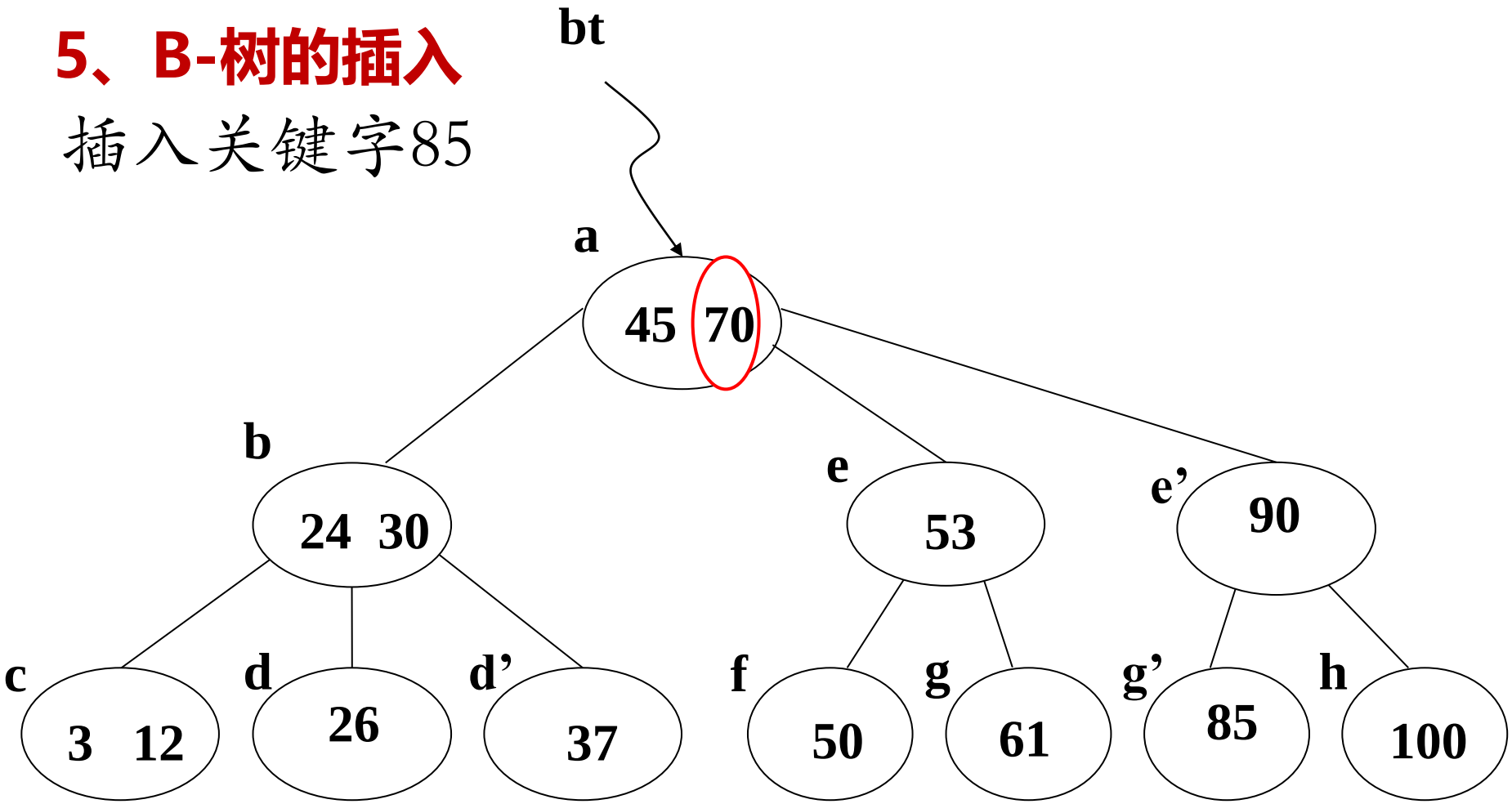
插入关键字85



9.2.3 B-树和B+树

5、B-树的插入

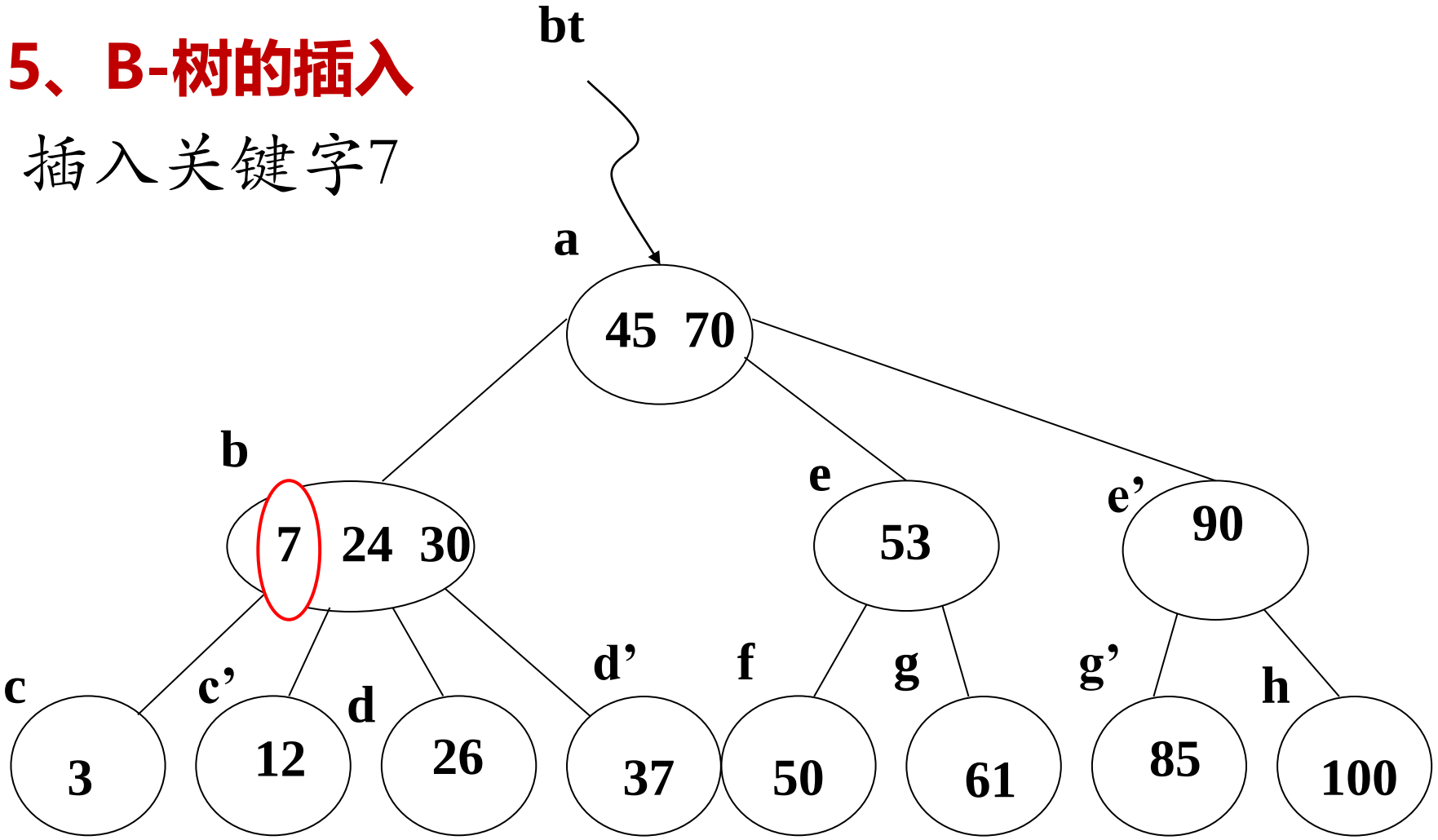
插入关键字85



9.2.3 B-树和B+树

5、B-树的插入

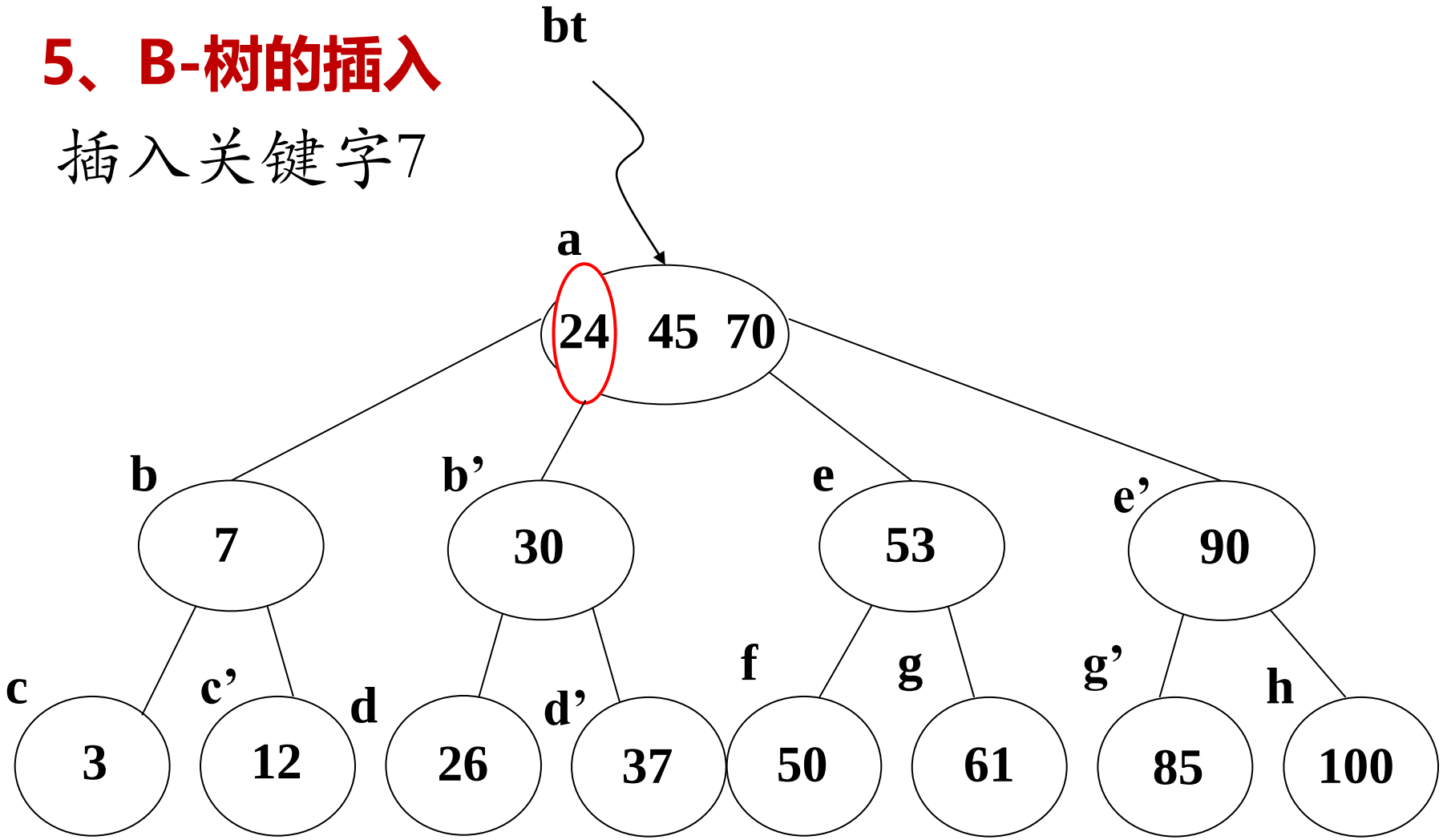
插入关键字7



9.2.3 B-树和B+树

5、B-树的插入

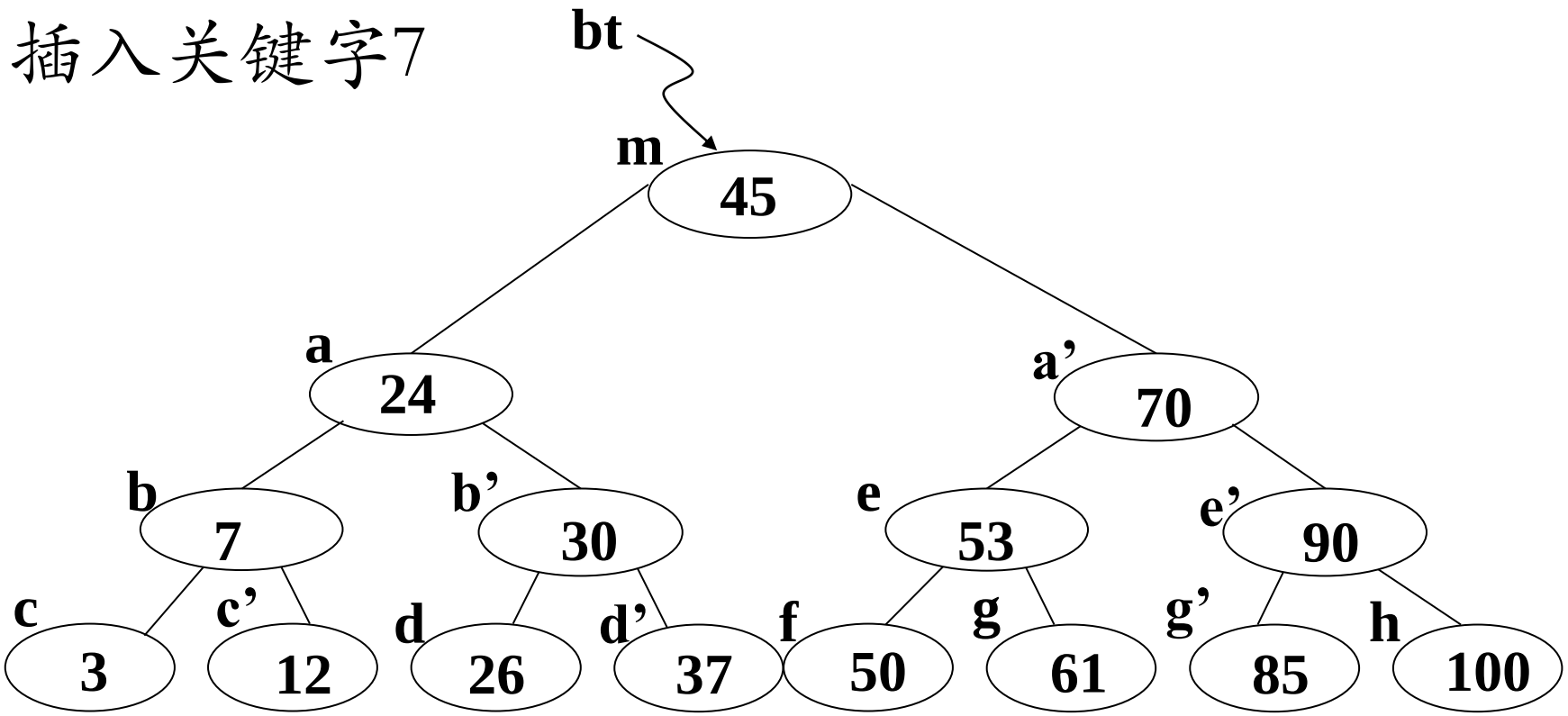
插入关键字7



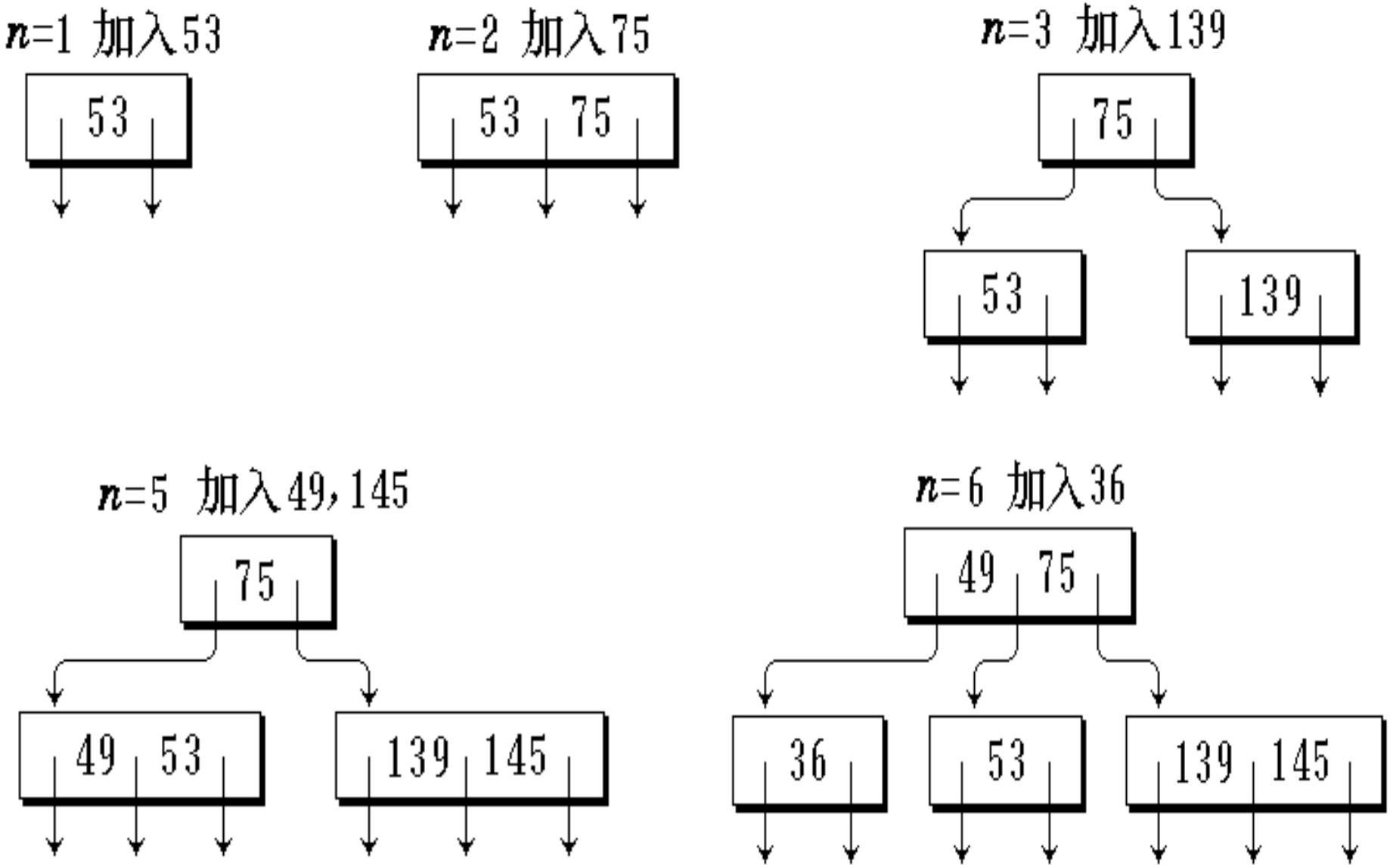
9.2.3 B-树和B+树

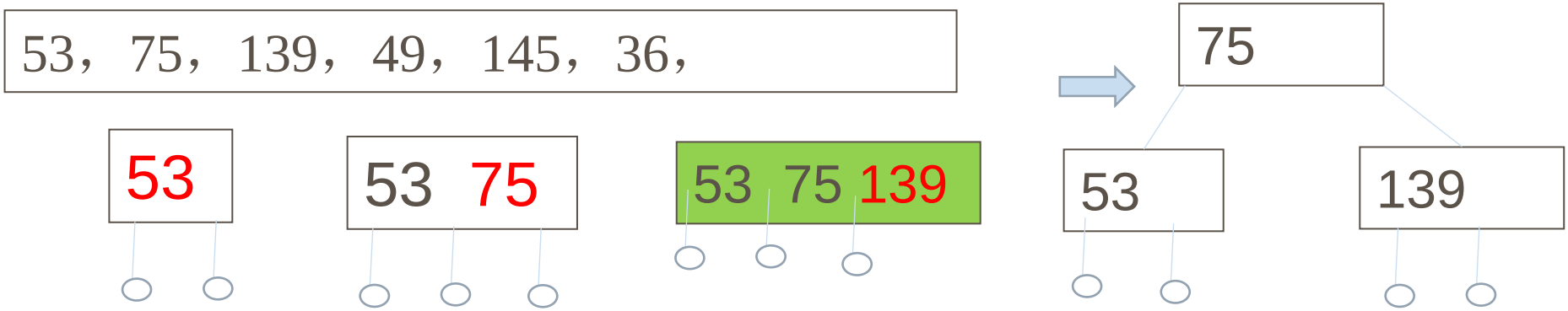
5、B-树的插入

插入关键字7



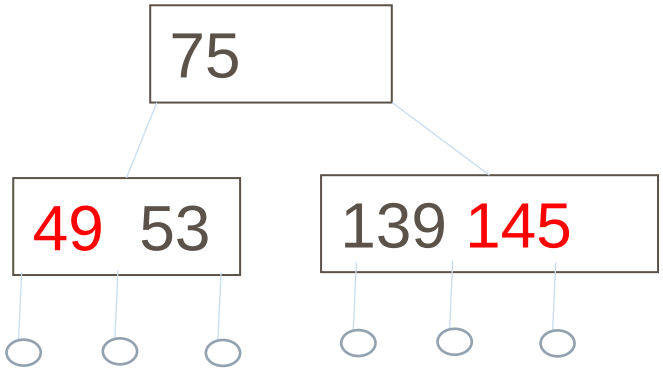
示例：从空树开始逐个加入关键字建立3阶B-树

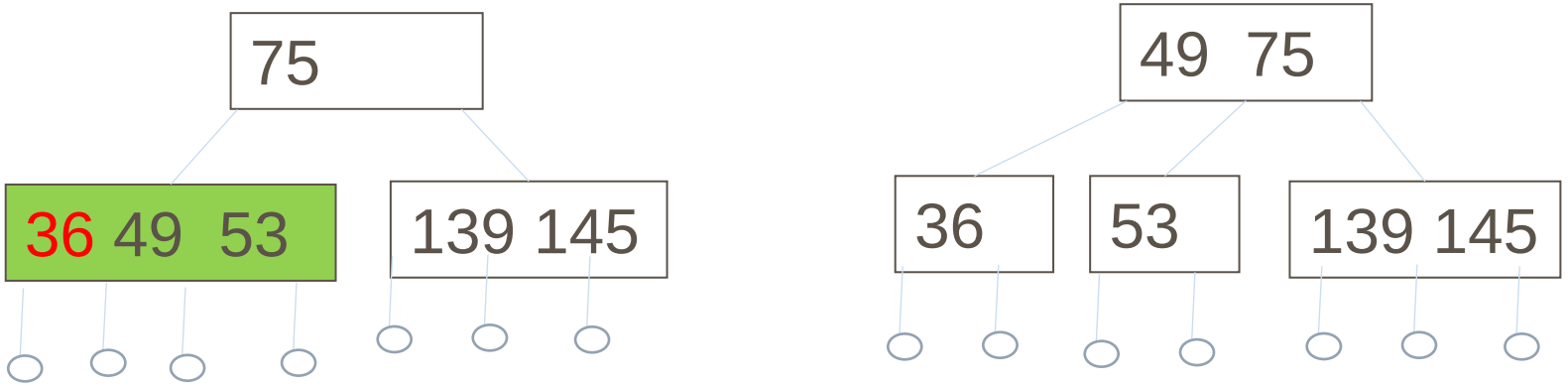




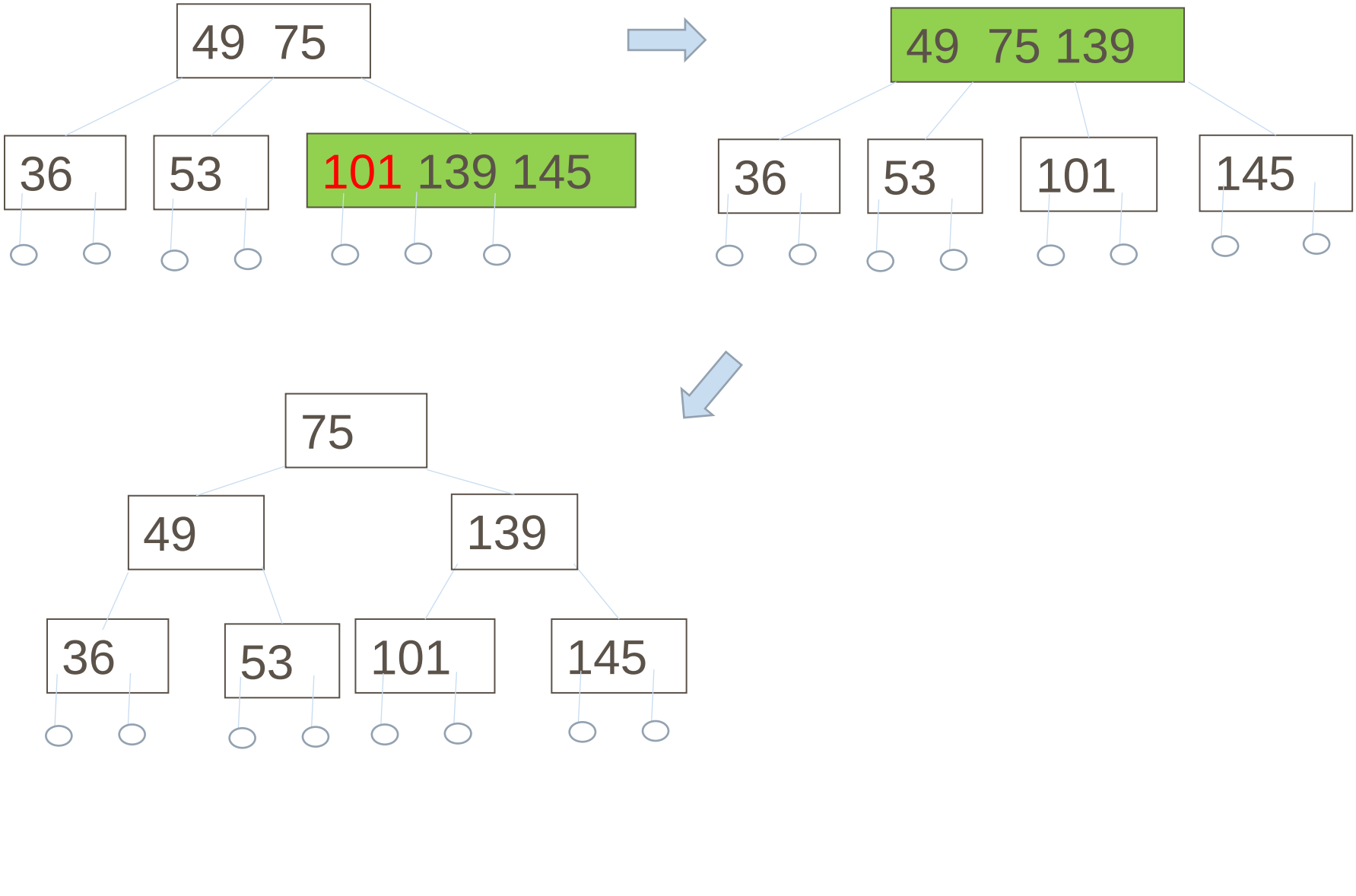
上溢时，分裂

取中间关键字，以及指向分裂后结点的指针，一起上移到双亲结点中





53，75，139，49，145，36，101，



9.2.3 B-树和B+树

```
int InsertBTree(BTree &T,KeyType K,BTree q, int i){
//在m阶B树T上结点*q的 key[i]与 key[i+1]之间插人关键字K。
//若引起结点过大,则沿双亲链进行必要的结点分裂调整,使T仍是m阶B树。
    BTree ap=NULL;  int finished=0;
    KeyType x;      x=K;
    while (q && !finished){
        Insert(q,i,x,ap);//将x和ap分别插人到q->key[i+1]和q>ptr[i+1]
        if (q->keynum<M) finished=1;//插入完成
        else{
            int s=(M+1)/2;
            split(q,s,ap); // 分裂结点 *q
            x=q->key[s];
            //将q->key[s+1..m],q->ptr[s..m]和q->recptr[s +1..m]移人新结点 *ap
            q=q->parent;
            if (q) i=Search(q,x);//在双亲结点 *q中查找x的插入位置
        }//else
    }//while
    if (!finished)//T是空树(参数q初值为 NULL)或者根结点已分裂为结点 *q和*ap
        NewRoot(T,q,x,ap);//生成含信息(T,x,ap)的新的根结点*T,原T和ap为子树指针
    return 1;
}//InsertBTree
```

9.2.3 B-树和B+树

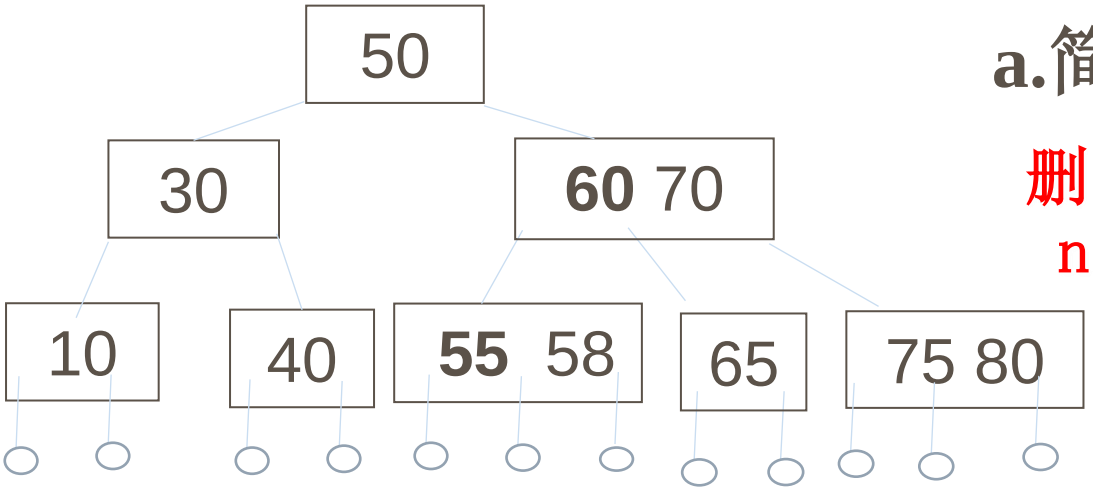
6、B-树的删除

在B-树中删除一个关键字，则首先应找到该关键字所在结点，并从中删除之。

①若所删关键字为非终端结点中的K_i，则以指针A_i所指子树中的最小关键字Y替代K_i，然后在相应的结点中删去Y。

②若所删关键字在最下层结点中。有3种情况：

a．被删关键字所在非终端结点不是根结点且删除前该结点中关键字个数 n≥[m/2]，则直接删去该关键字Ki和相应的指针Ai，并将修改后的结点写回磁盘，删除结束

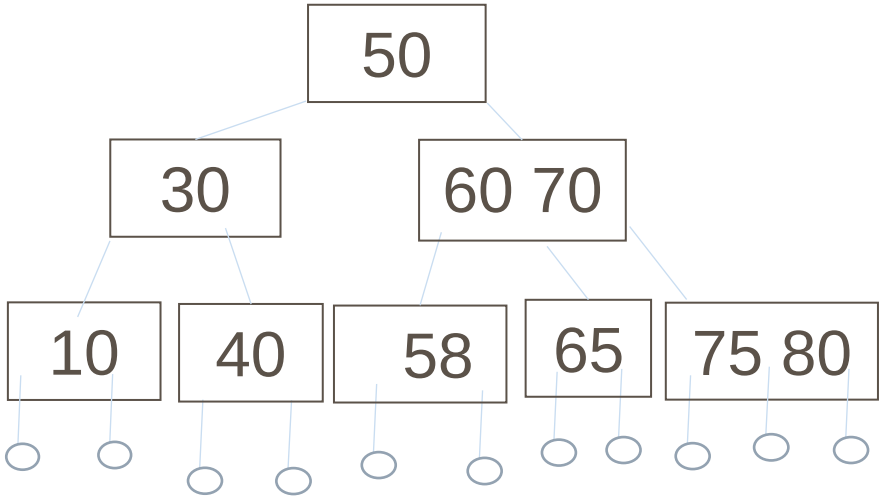


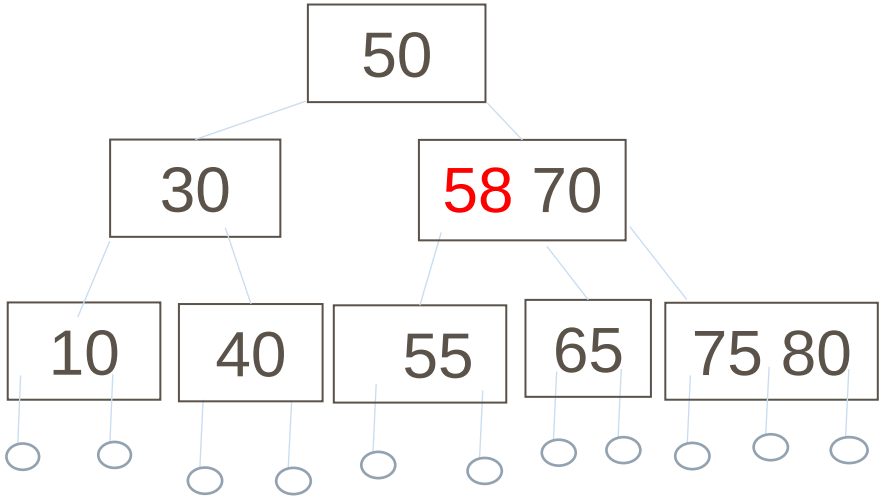
a.简单删除

删除前该结点中关键字个数 $n \geq \lceil m/2 \rceil$

例如：

1. 删除55
2. 删除60





9.2.3 B-树和B+树

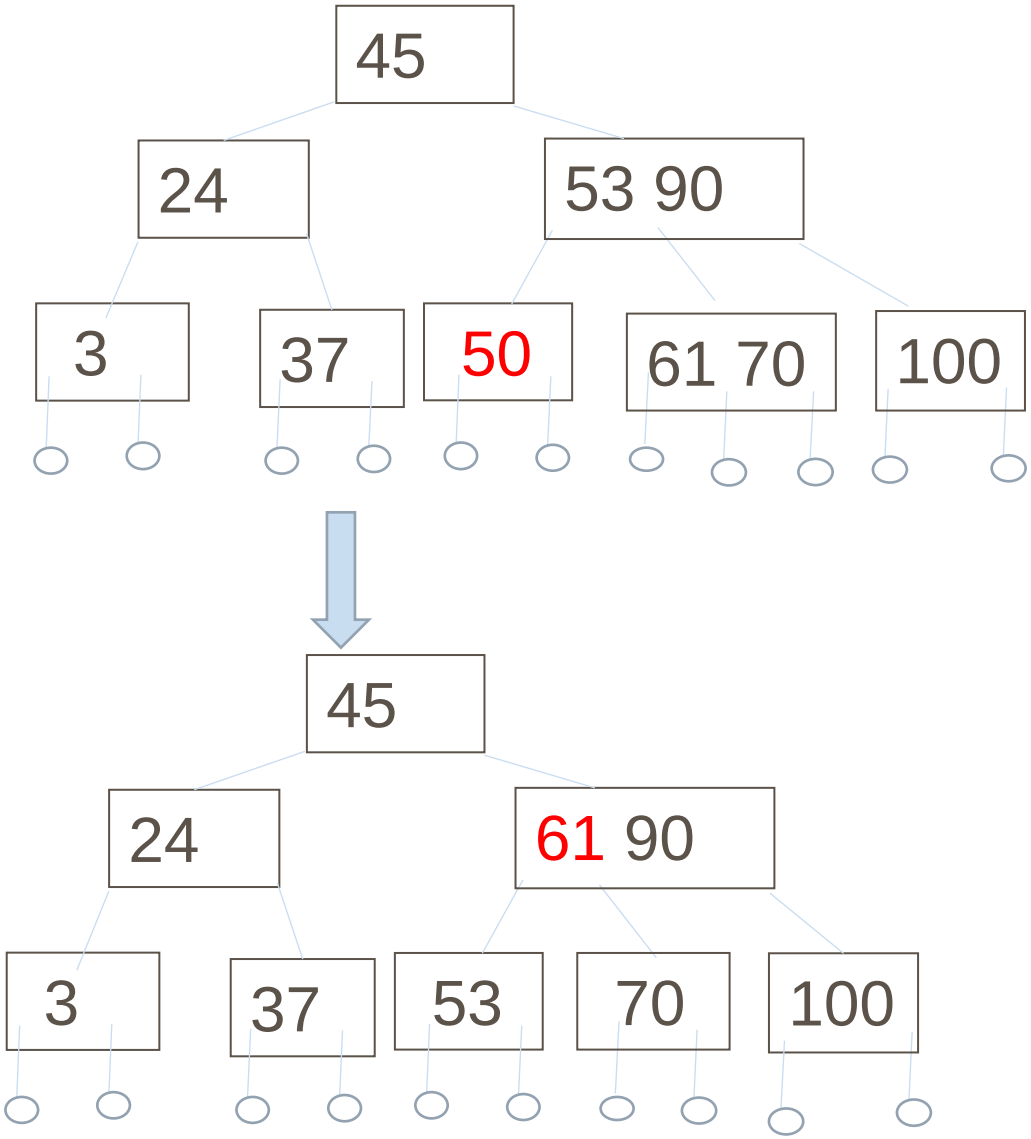
6、B-树的删除

②若所删关键字在最下层结点中。有3种情况：

b．被删关键字所在非终端结点删除前关键字个数n=⌈m/2⌉-1，若这时与该结点相邻的右兄弟（或左兄弟）结点的关键字个数n≥⌈m/2⌉：

则可按以下步骤调整该结点、右兄弟（或左兄弟）结点以及其双亲结点，以达到新的平衡。

- 将右兄弟（或左兄弟）结点中的最小（或最大)关键字上移到双亲结点的 Ki 位置；
- 将双亲结点中刚刚大于（或小于）该被删关键字的关键字 Ki （1≤i≤n）下移至被删关键字所在的结点中。



b.结点联合调整

若被删关键字所在结点删除前关键字个数：

$n = \lceil m/2 \rceil - 1$，（下溢）

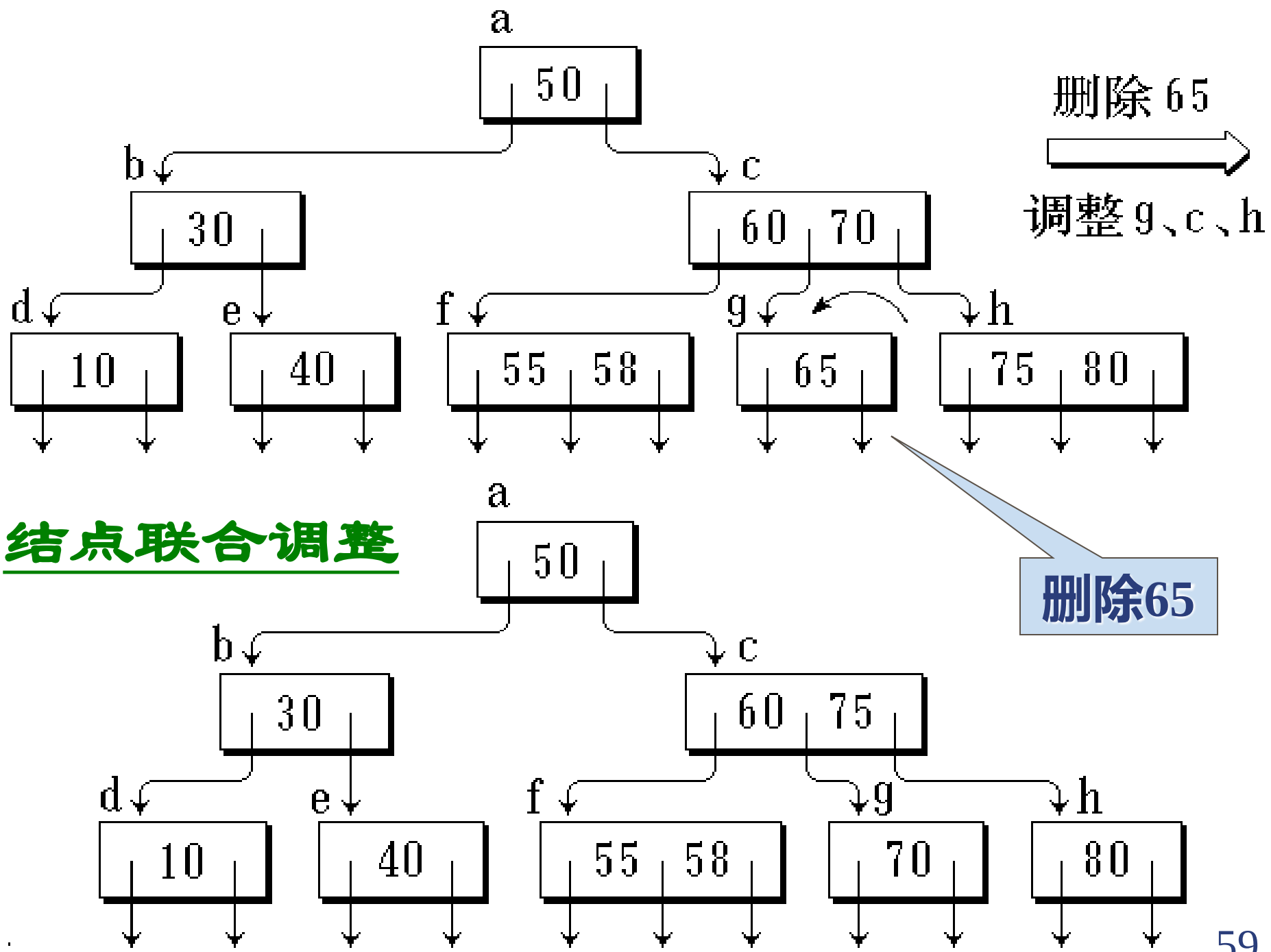
该结点相邻的右兄弟（或左兄弟）结点的关键字个数：

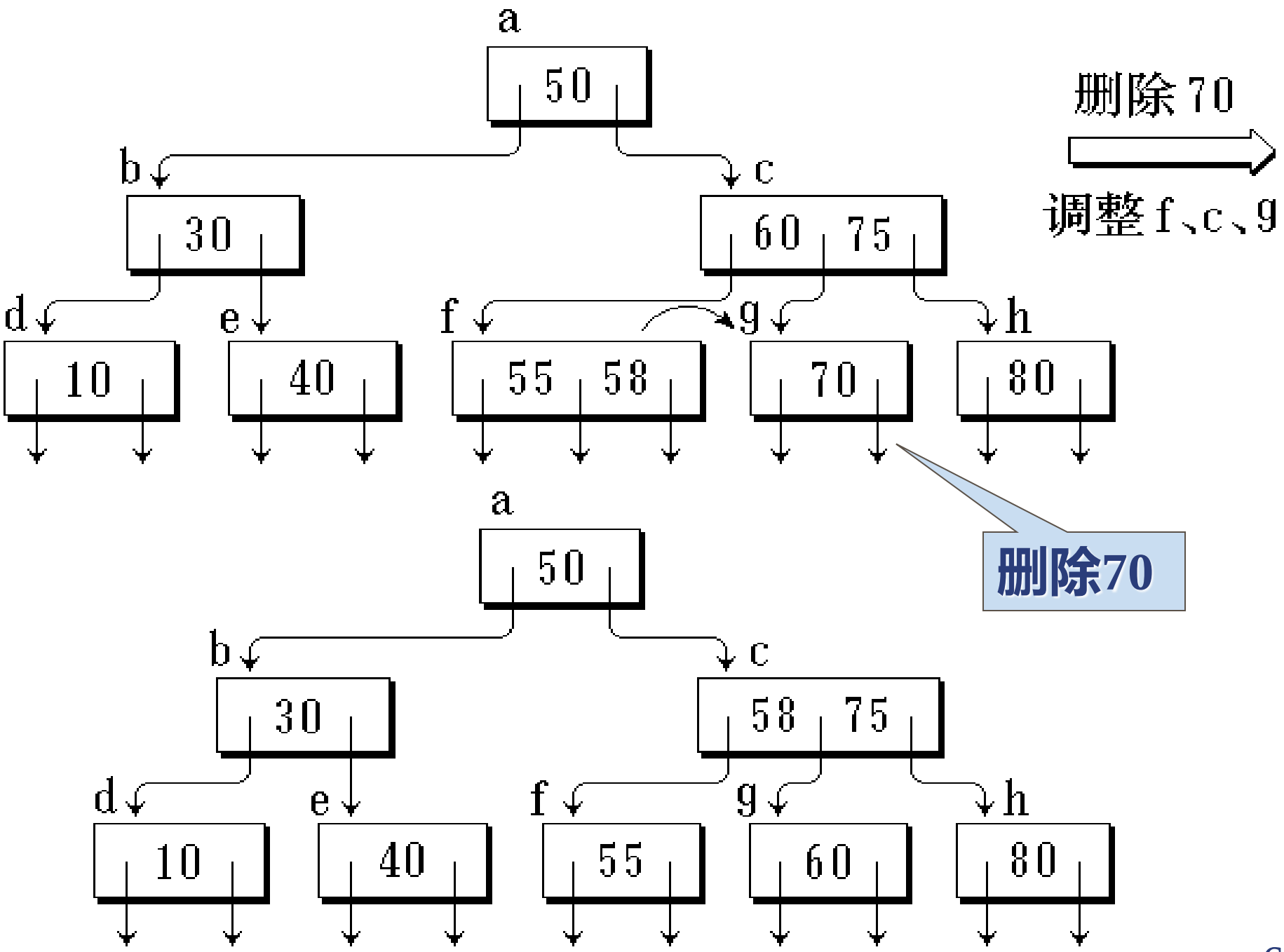
$n \geq \lceil m/2 \rceil$

办法：兄弟结点上移一个关键码给双亲，双亲下移一个相邻关键码到溢出结点

向兄弟“借”

例如： 删除50


a
50
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
删除 65
调整 g、c、h
删除65
结点联合调整
a
50
b
30
c
60 75
d
10
e
40
f
55 58
g
70
h
80


a
50
b
30
c
60 75
d
10
e
40
f
55 58
g
70
h
80
删除70
调整f、c、g
删除70
a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80

9.2.3 B-树和B+树

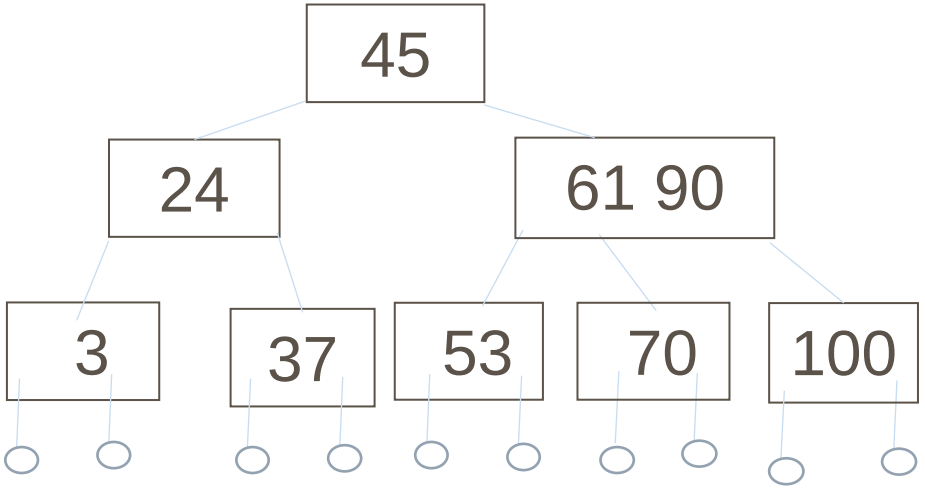
6、B-树的删除

②若所删关键字在最下层结点中。有3种情况：

c．被删除关键字所在的结点和其相邻的兄弟结点中的关键字个数均等于⌈m/2⌉-1。

若该结点有右兄弟，且其右兄弟结点地址由双亲结点中的指针Ai 所指，则在删除关键字之后，它所在结点中剩余的关键字和指针，加上双亲结点中的关键字Ki一起，合并到Ai 所指的兄弟结点中（若没有右兄弟，则合并到左兄弟结点中）

如果因此导致双亲结点中的关键字数目小于 ⌈m/2⌉-1 ，则以此类推，做相应的处理。

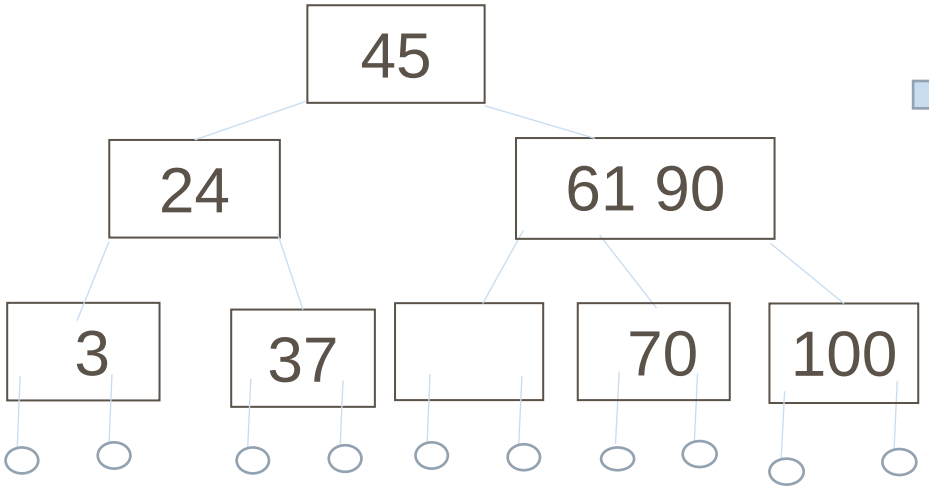


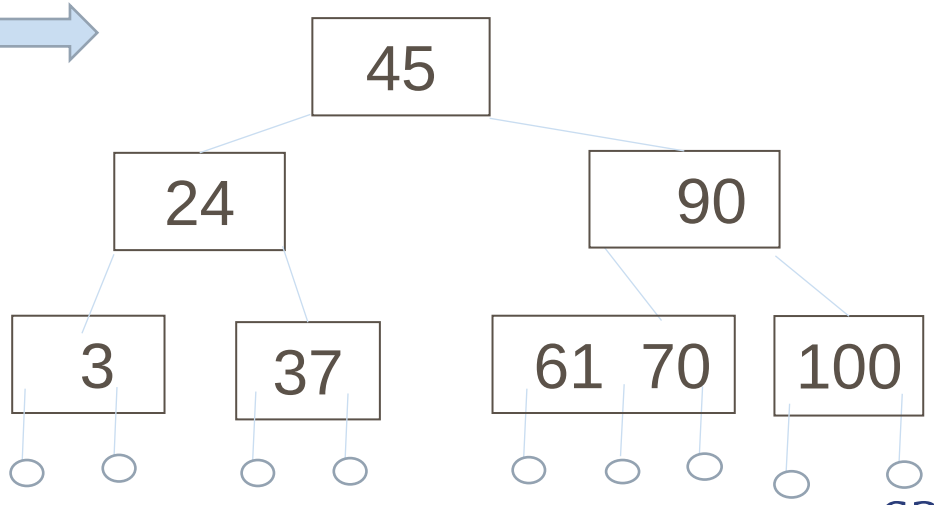
c.结点合并

被删除关键字所在结点和其相邻的兄弟结点中的关键字个数**均等于⌈m/2⌉-1**

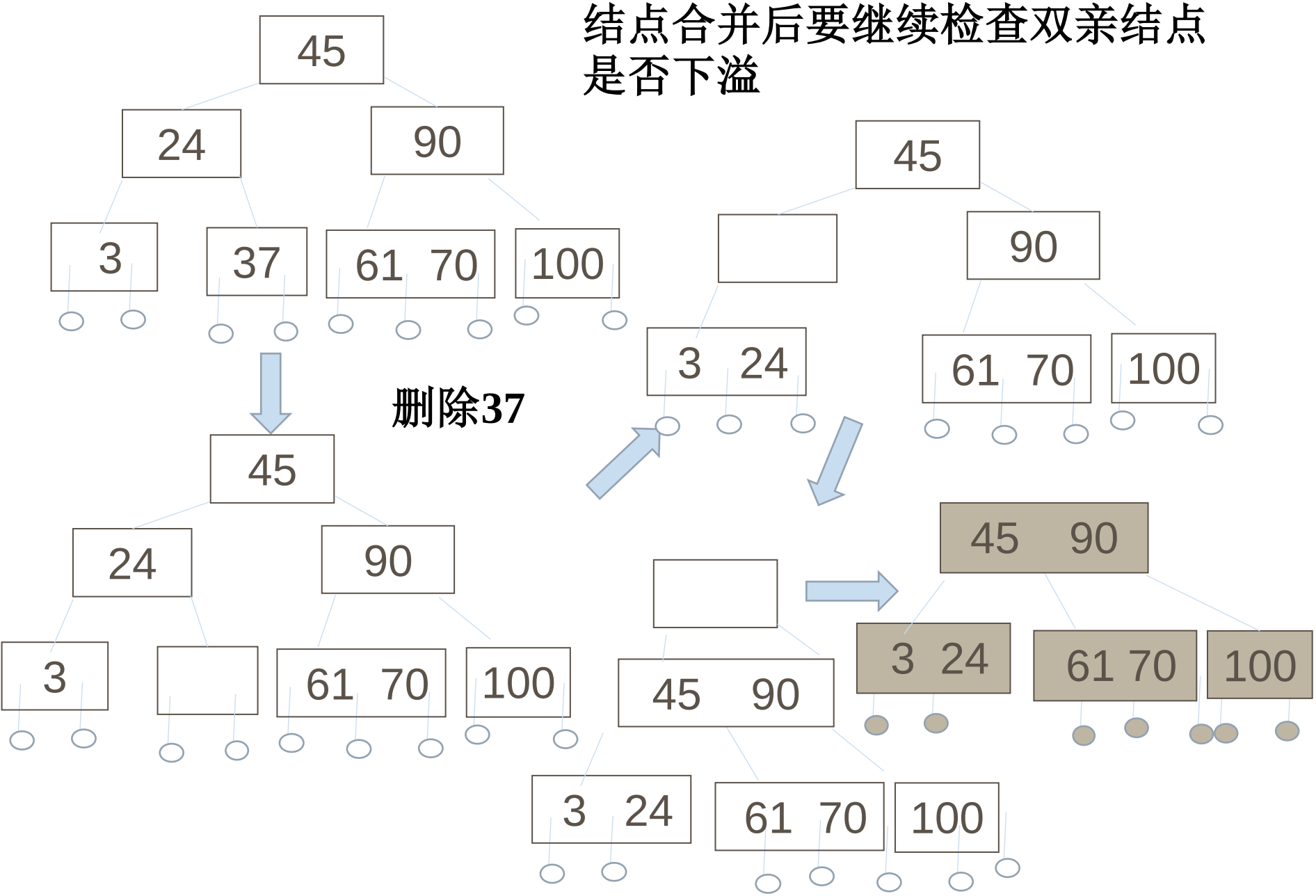
办法：从父结点中取出介于被删除结点和兄弟结点之间的关键字，连同被删结点剩余部分**一起合并到兄弟结点中**。

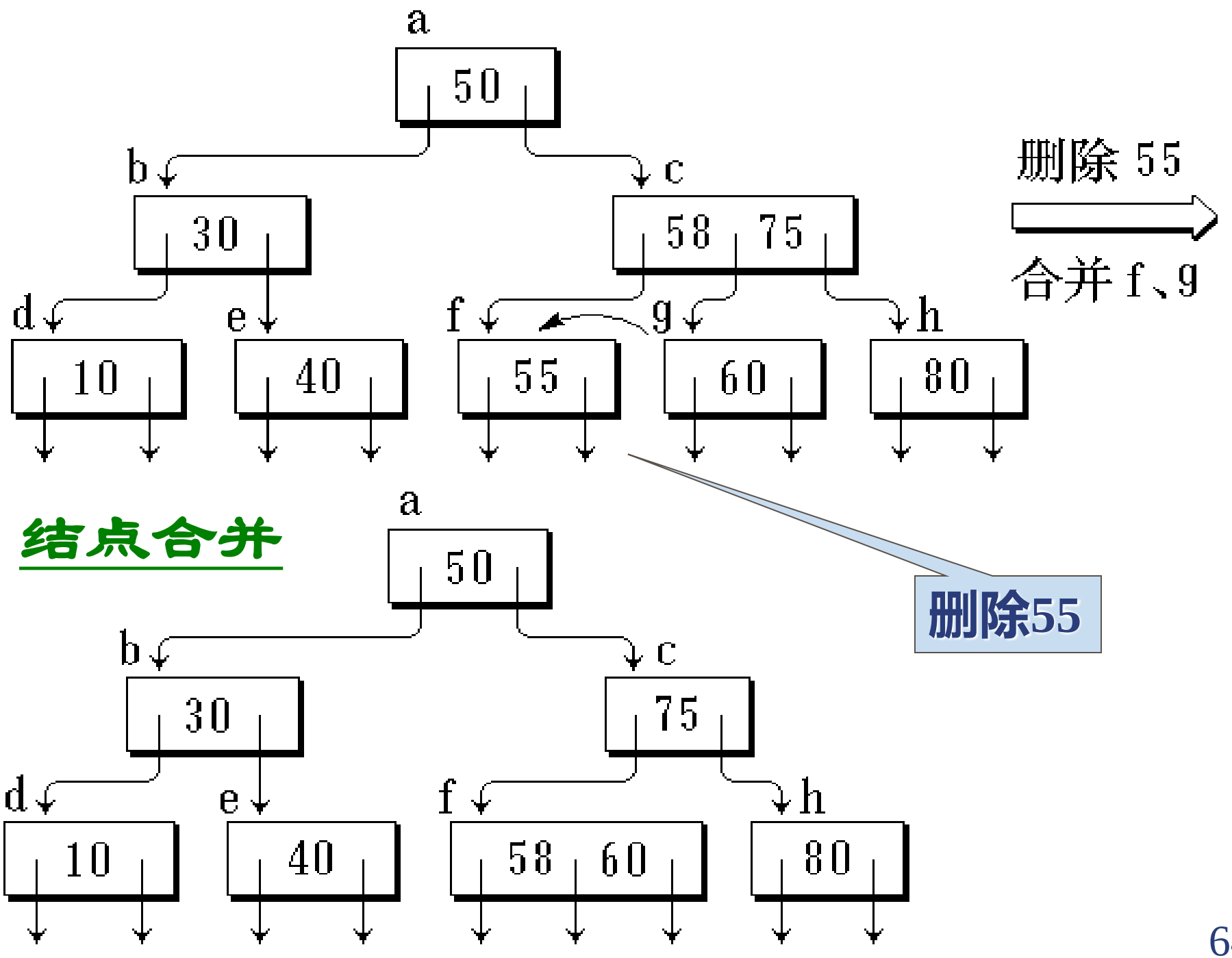
删除53



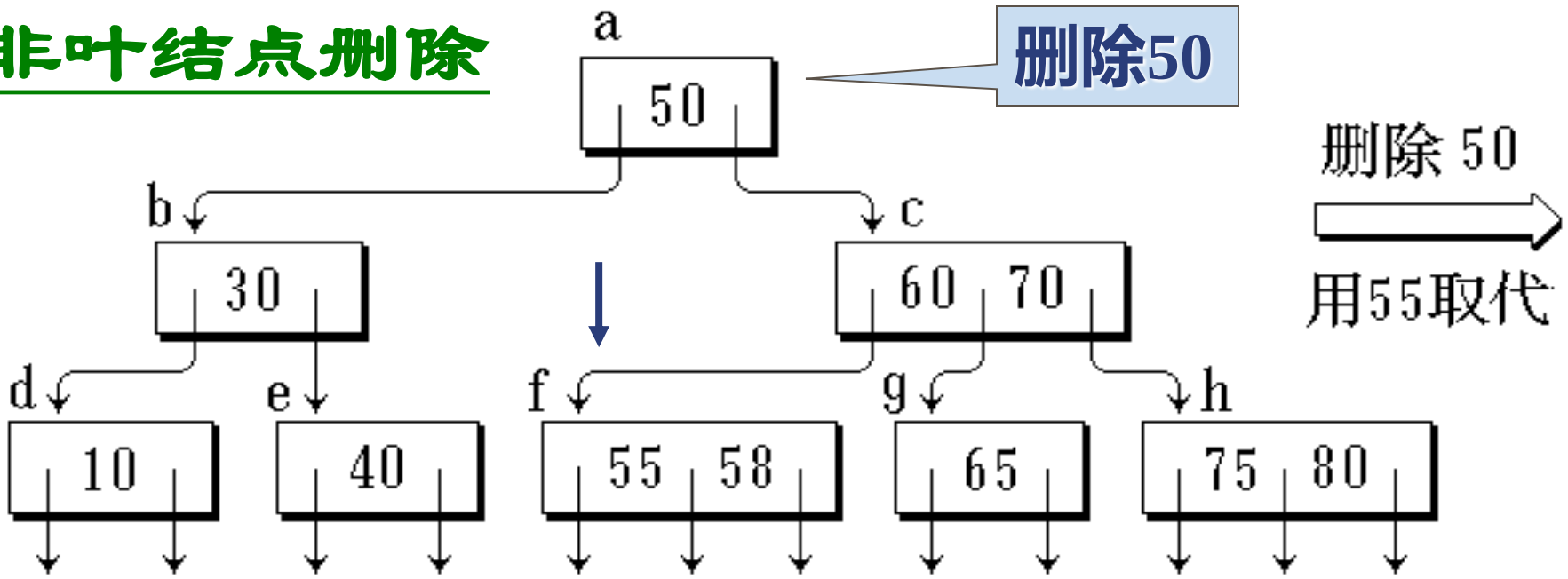


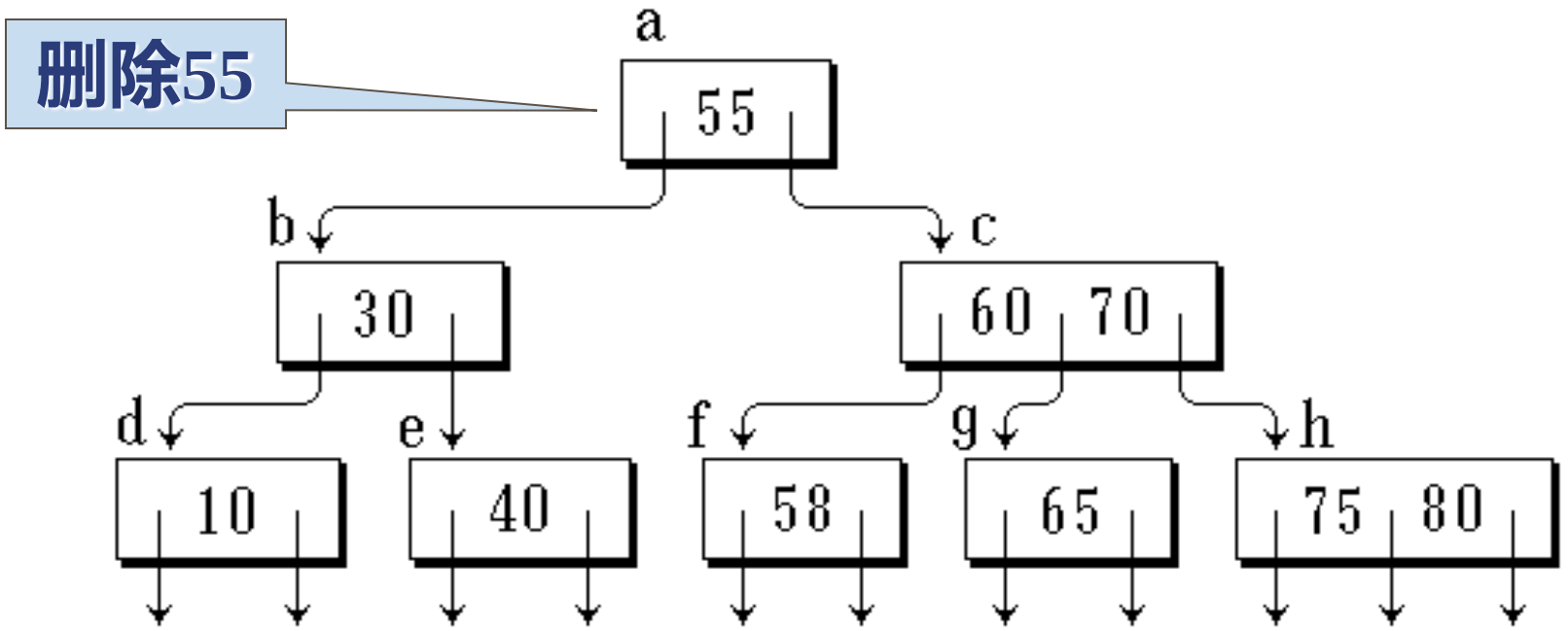
结点合并后要继续检查双亲结点是否下溢

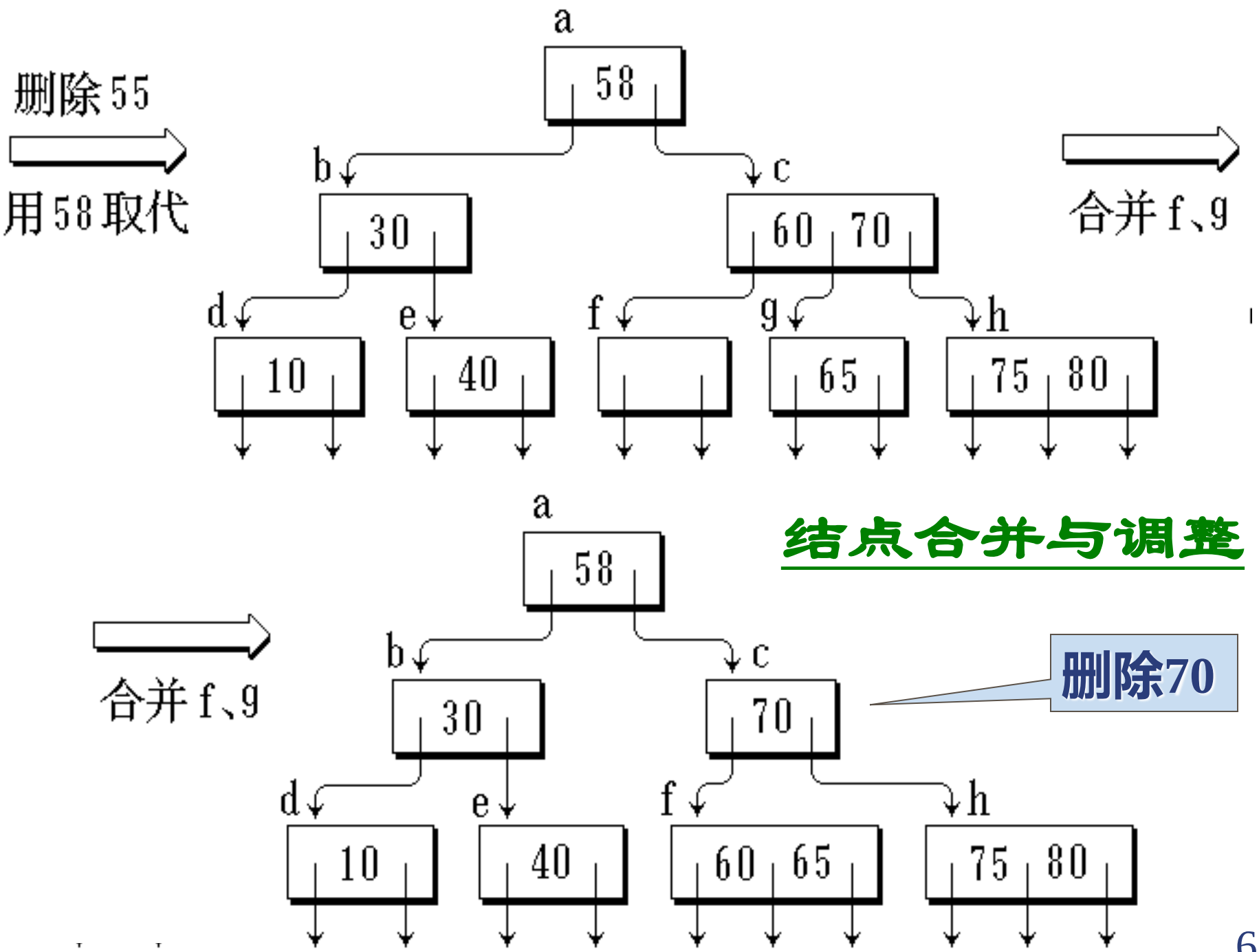


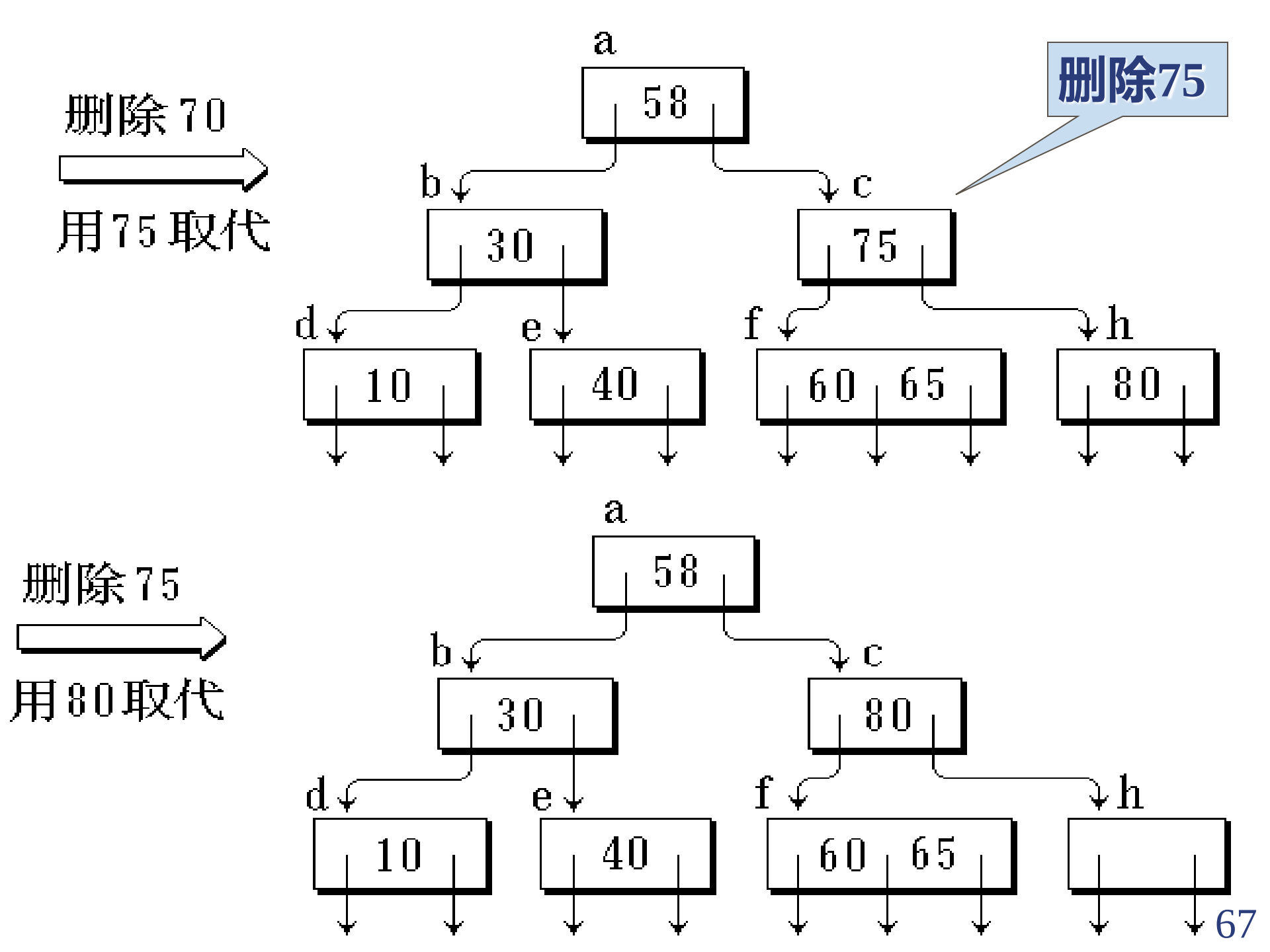

a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80
删除 55
合并 f、g
结点合并
删除55
a
50
b
30
c
75
d
10
e
40
f
58 60
h
80

非叶结点删除

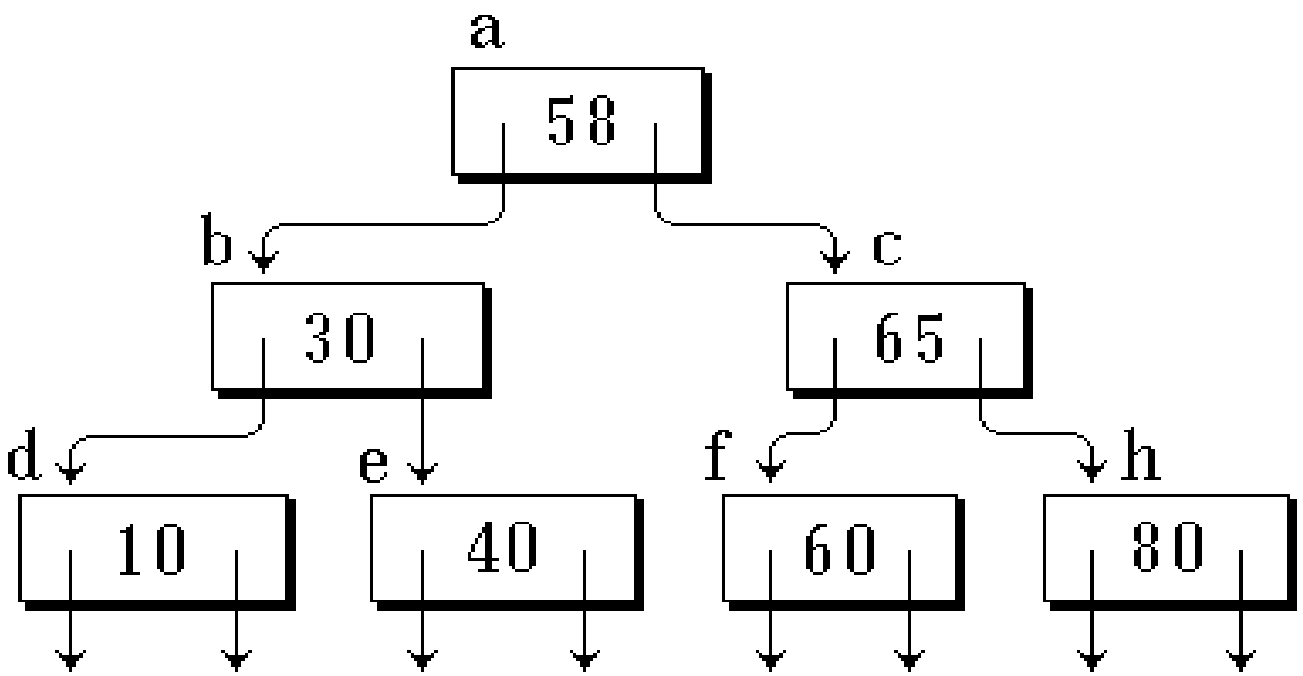

a
50
删除50
删除 50
用55取代
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80


删除55
a
55
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80


删除55
用58取代
a
58
b
30
c
60 70
d
10
e
40
f
g
65
h
75 80
合并f、g
结点合并与调整
合并f、g
a
58
b
30
c
70
删除70
d
10
e
40
f
60 65
h
75 80


删除70
用75取代
a
58
b
30
c
75
d
10
e
40
f
60
65
h
80
删除75
删除75
用80取代
a
58
b
30
c
80
d
10
e
40
f
60
65
h

调整 f、c、h



9.2.3 B-树和B+树

1、B+树定义

B^+树是应文件系统所需而出的一种B-树的变型树。一棵m阶的B^+树和m阶的B-树的差异在于：

（1）每个结点最多m棵子树，每个结点中最多含有m个关键字；

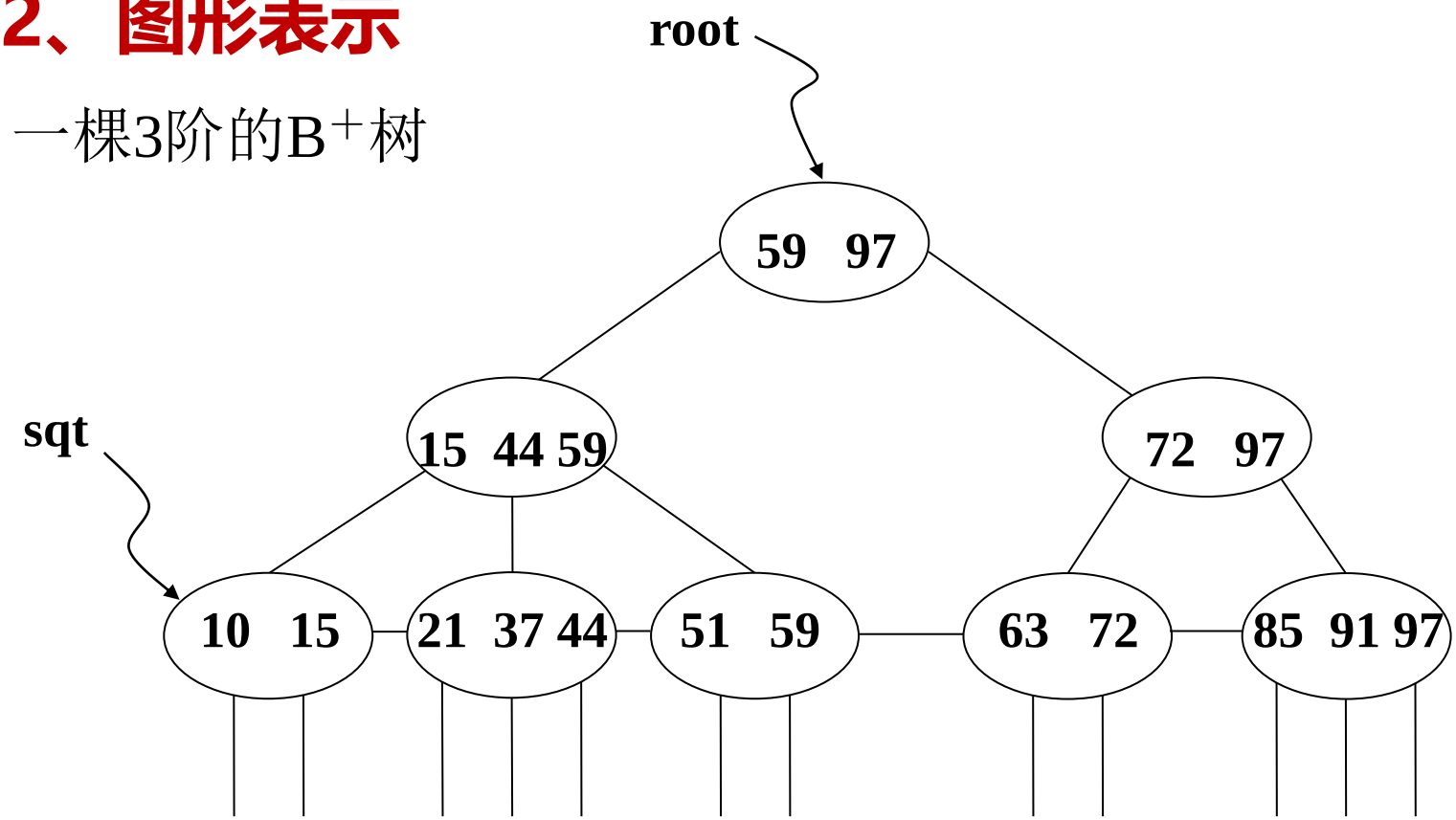
（2）所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

（3）所有的非终端结点可以看成是索引部分，结点仅含其子树中的最大（或最小）关键字。

9.2.3 B-树和B+树

2、图形表示

一棵3阶的B+树



通常在B+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。

9.2.3 B-树和B+树

3、B^+树的查找

对B^+树可以进行两种查找运算:

①从最小关键字起顺序查找;

②从根结点开始，进行随机查找。

在查找时，若非终端结点上的索引值等于给定值，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。其余同B-树的查找类似。

9.2.3 B-树和B+树

4、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 $\left\lceil \frac{m+1}{2} \right\rceil$ 和 $\left\lceil \frac{m}{2} \right\rceil$。并且，它们的双亲结点中应同时包含这两个结点中的最大关键字。其余同B-树的插入类似。

9.2.3 B-树和B+树

4、B⁺树的删除

B⁺树的删除也仅在叶子结点进行，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$\lceil \frac{m}{2} \rceil$ 时，其和兄弟结点的合并过程亦和B-树类似。

正在答疑