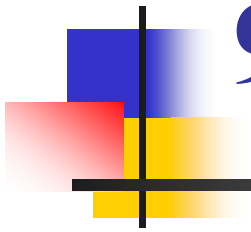


9.2.2 B-树和B+树

B-树

- B-树是一种平衡的多路查找树
- m阶B-树--空树或满足下列特性的m叉树：

(1) 树中每个结点最多m棵子树；

(2) 若根结点不是叶结点，则至少有2棵子树；

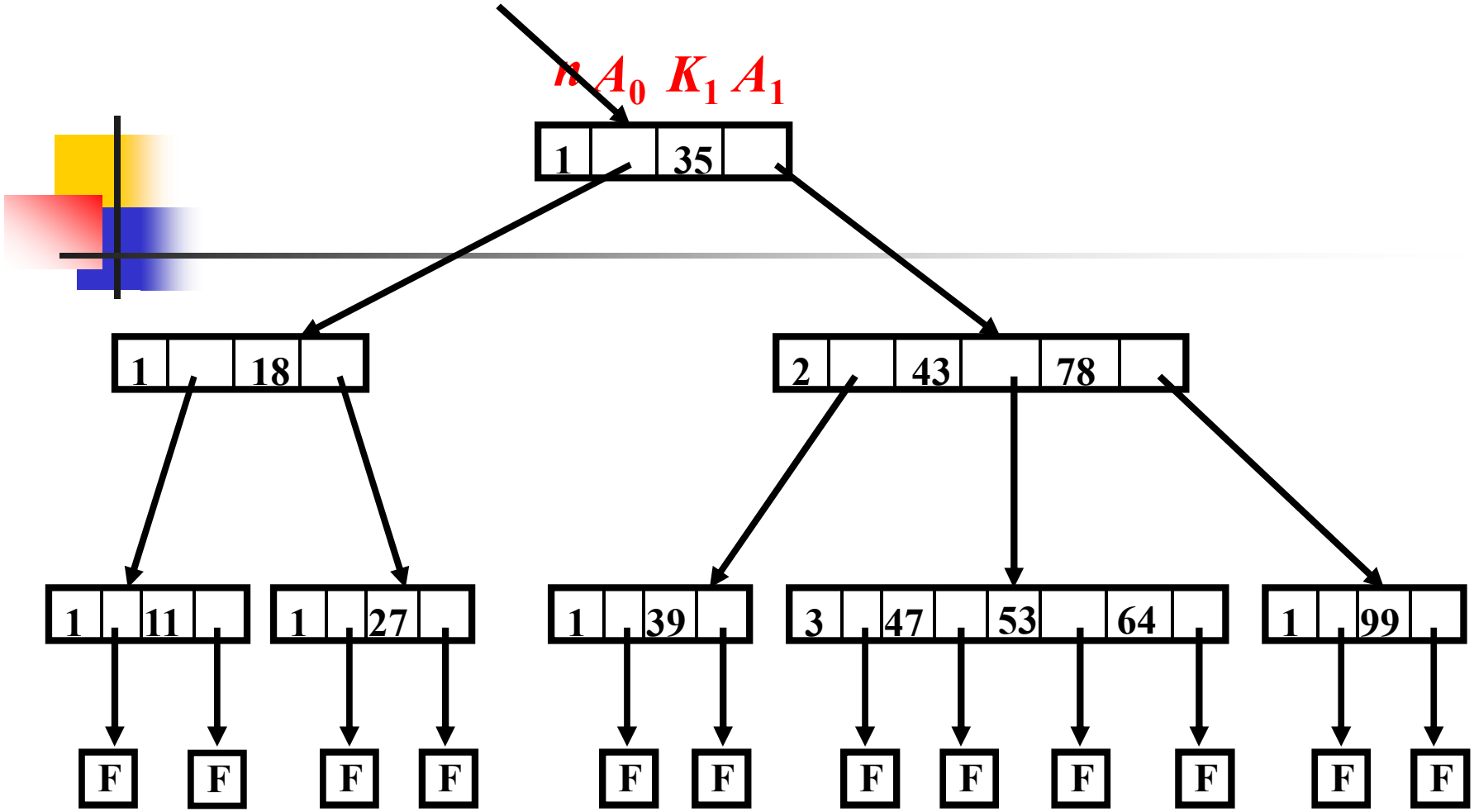
(3) 除根之外所有非叶结点至少有$\lceil m/2 \rceil$棵子树；

(4) 所有的非叶结点包含下列信息数据

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

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

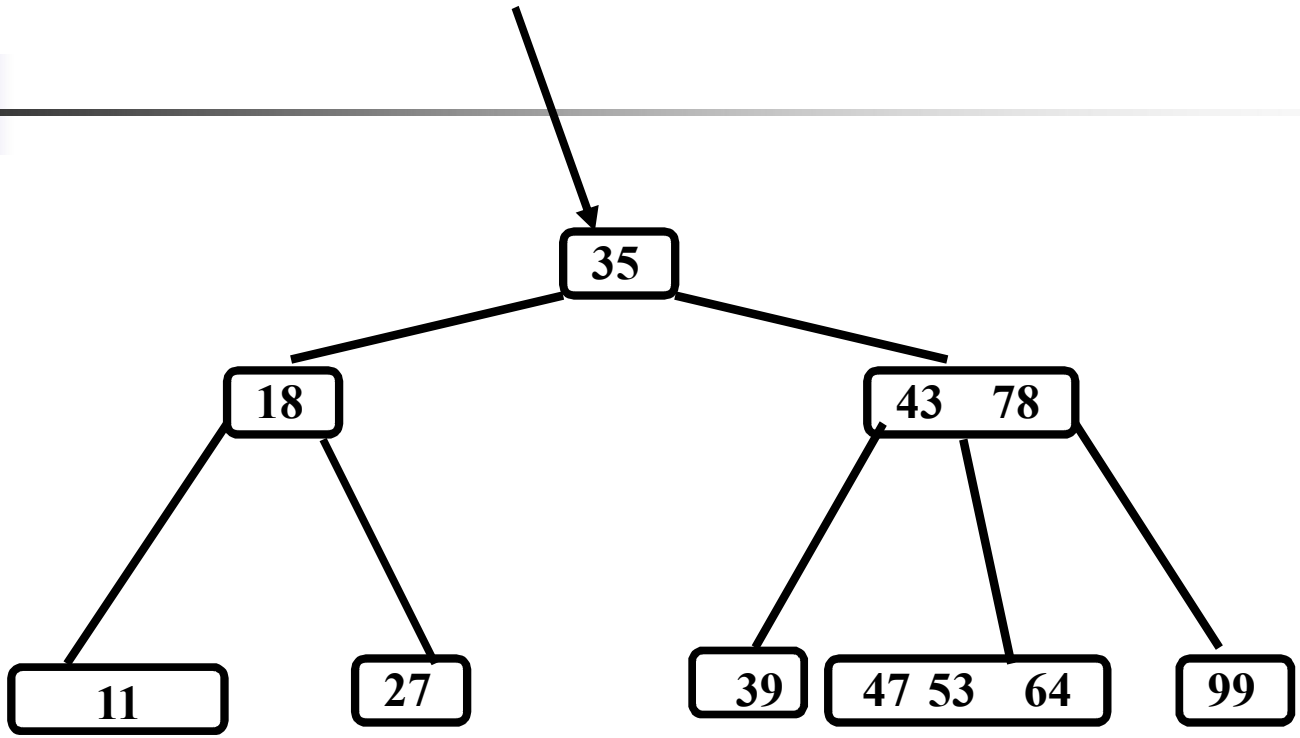
- n个关键字递增有序：$K_1 < K_2 <, \ldots, < K_n$
- $n+1$棵子树：$A_0, A_1, \ldots, A_n$
- A_{i-1}子树上所有结点的关键字均小于K_i
- A_i子树上所有结点的关键字均大于K_i
- n要求满足：$\lceil m/2 \rceil - 1 \le n \le m-1$
- 所有叶子结点位于同一层上，不带信息，代表查找失败！



一棵4阶的B-树

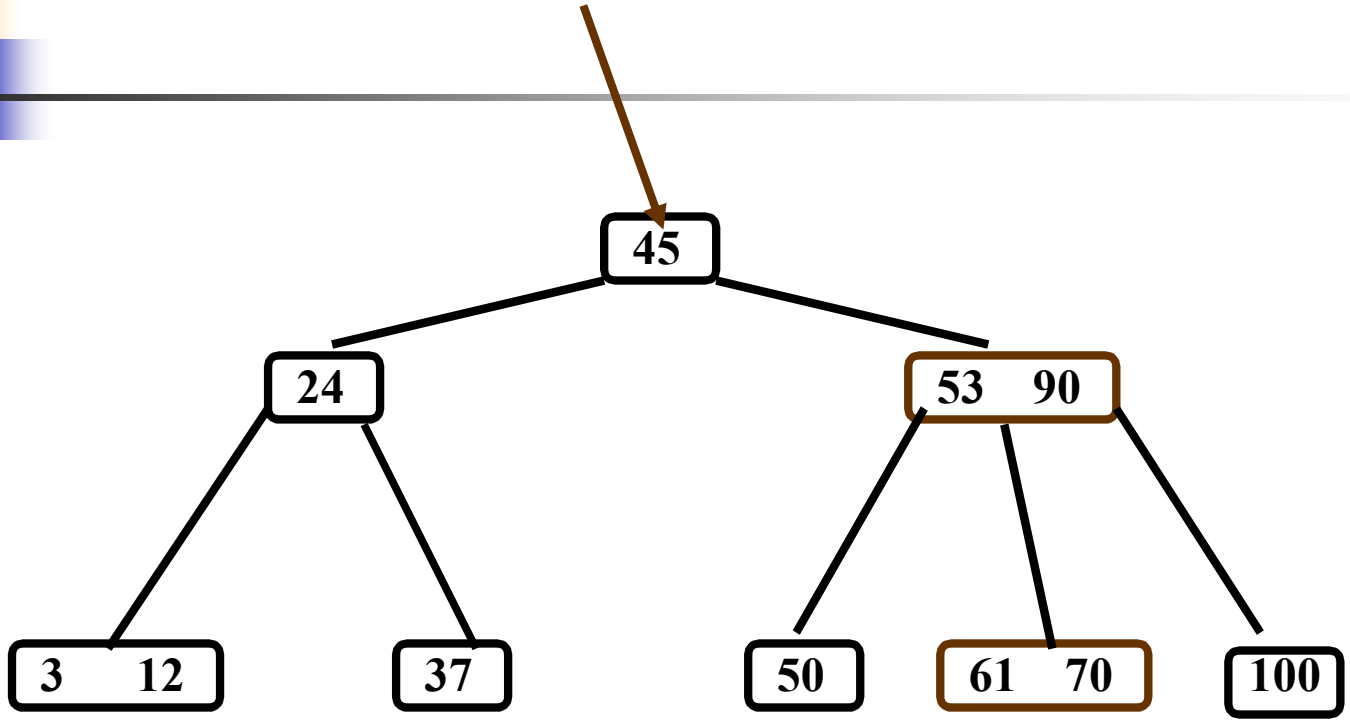
每个非叶结点最多有4棵子树，最少有2棵子树

4阶B-树的简化画法



一棵4阶的B-树

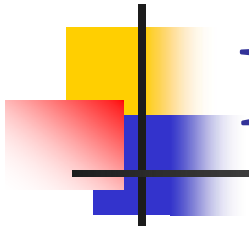
每个非叶结点最多有4棵子树，最少有2棵子树



一棵3阶的B-树

3阶的B-树又称为2-3树

每个非叶结点最多有3棵子树，最少有2棵子树



B-树的应用背景

- 当查找的文件较大，且存放在磁盘等直接存取设备中时,为提高查找效率,建索引----->B-树
- 结点中关键字还跟有一个指针指向相应记录的存放位置

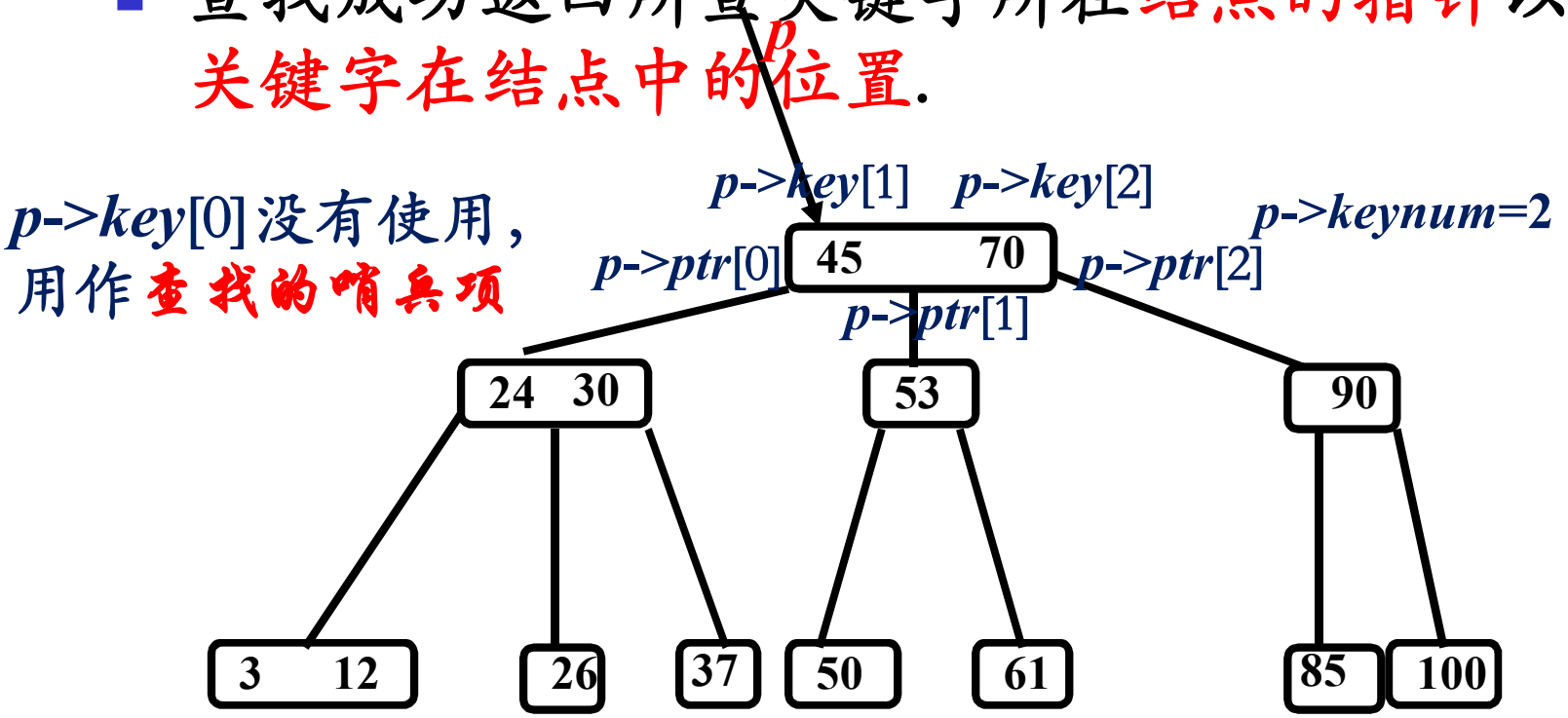
$(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$

B-树的存储结构

```
#define m 10
typedef struct BTNode {
        int keynum;   // n
        KeyType key[m] ;  //K1,  K2, ,…, Kn
        struct BTNode * parent ;
        struct BTNode *ptr[m];  //A0, A1, A2,…,  An
        Record*recptr[m] ;
        }BTNode,*BTree;
```

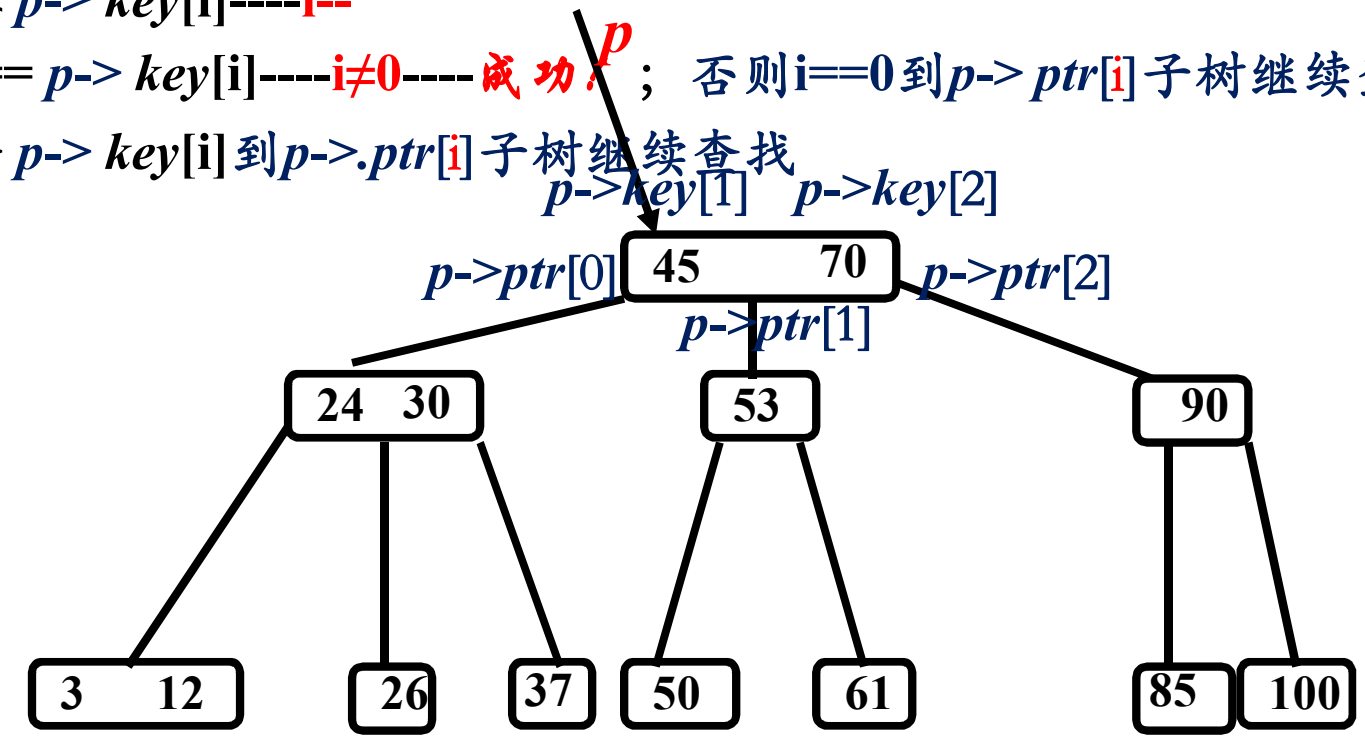
B-树查找

- 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
- 查找成功返回所查关键字所在结点的指针以及关键字在结点中的位置.

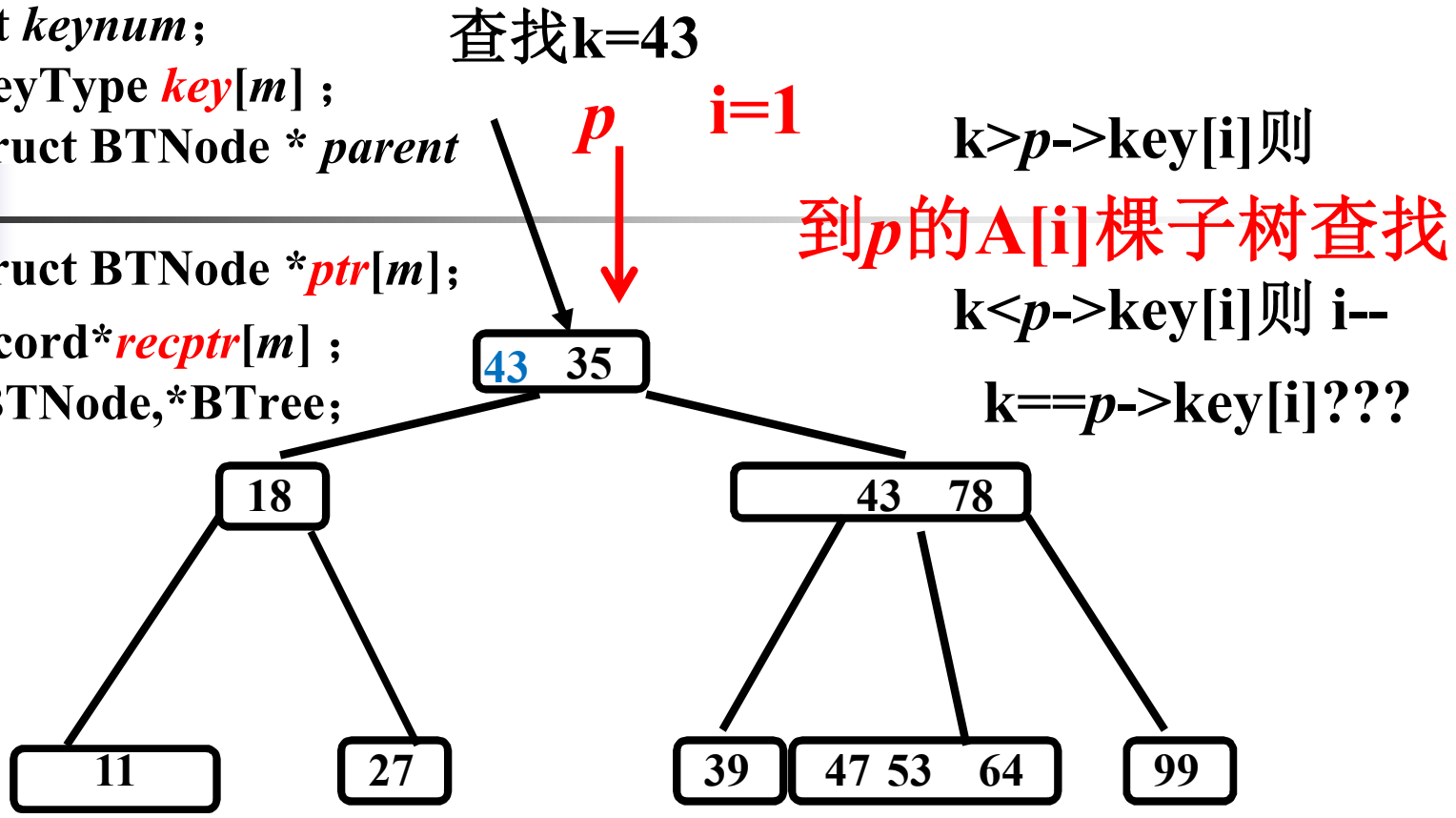


B-树查找——在*p*结点查找k

- 在*p*结点内进行顺序—从*p*结点的最后一个关键字查看到第一个关键字
- *p->key*[0]=k; //哨兵项
- i= *p->* keynum---- *p*结点的最后一个关键字的位置
- k< *p-> key*[i]----i--
- k== *p-> key*[i]----i≠0----成功！; 否则i==0到*p-> ptr*[i]子树继续查找
- k> *p-> key*[i]到*p->.ptr*[i]子树继续查找

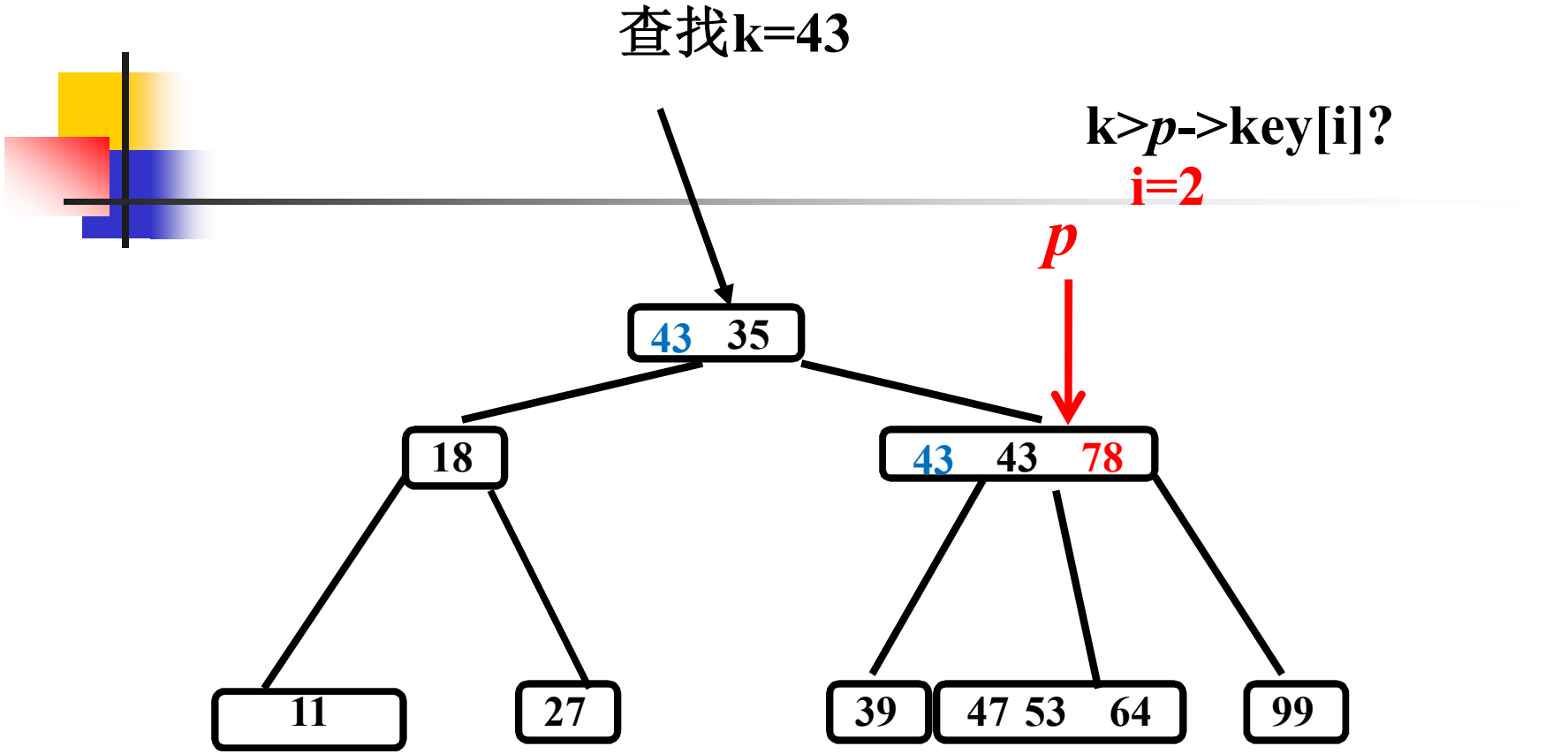


- #define *m* 10
  typedef struct BTNode {
  int *keynum*;
  KeyType *key*[*m*] ;
  struct BTNode * *parent*
  ;
  struct BTNode **ptr*[*m*];
  Record**recptr*[*m*] ;
  }BTNode,*BTree;



蓝色字体内容为哨兵项

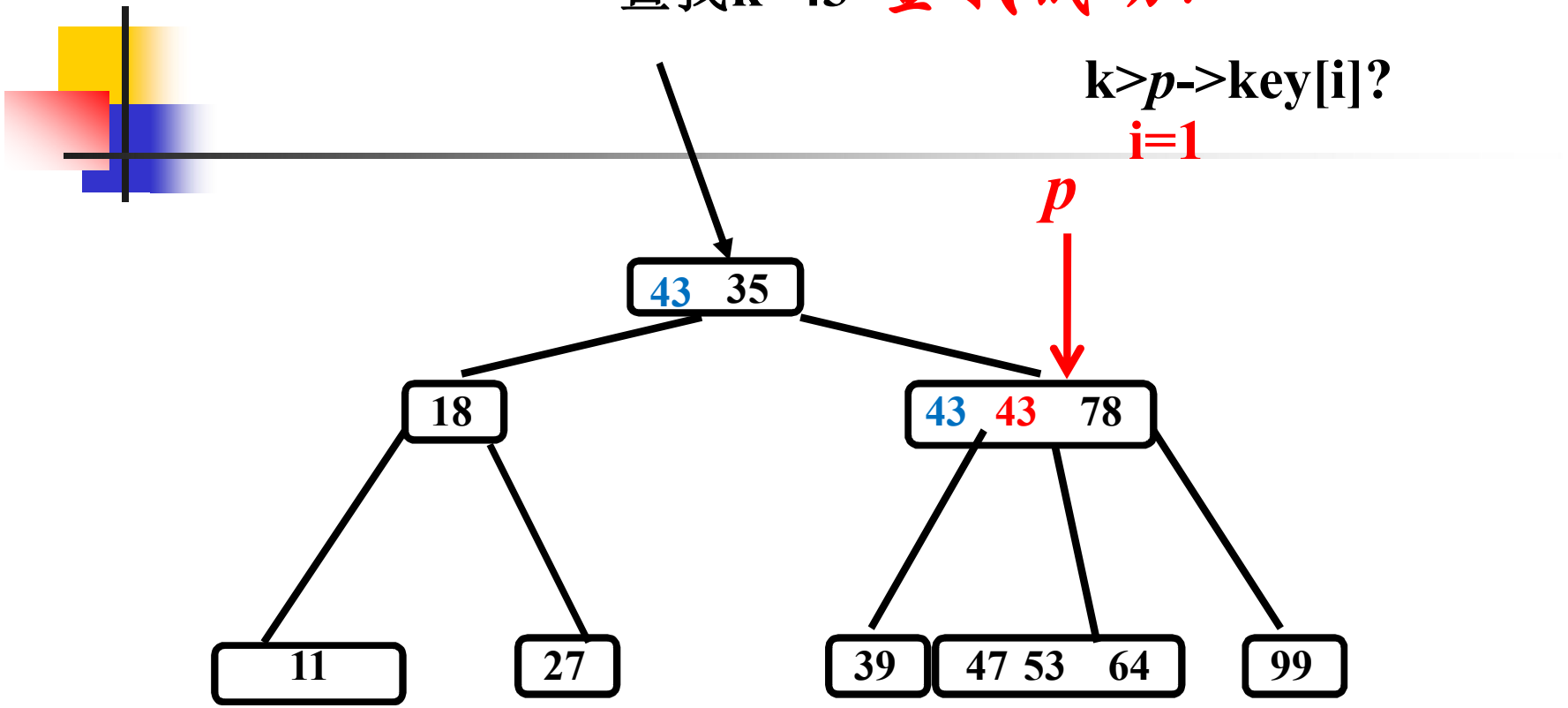
一棵4阶的B-树



蓝色字体内容为哨兵项

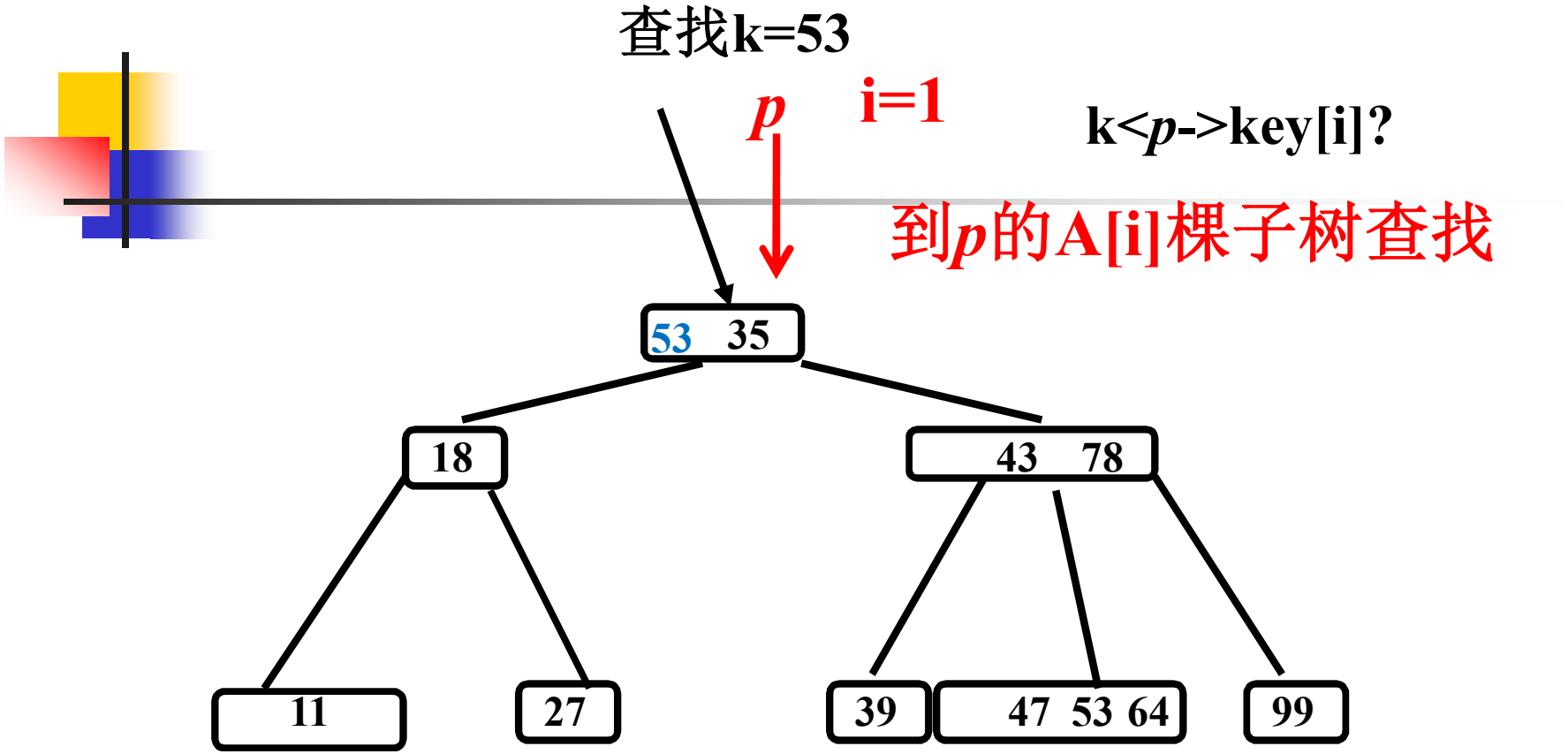
一棵4阶的B-树

查找k=43 查找成功！



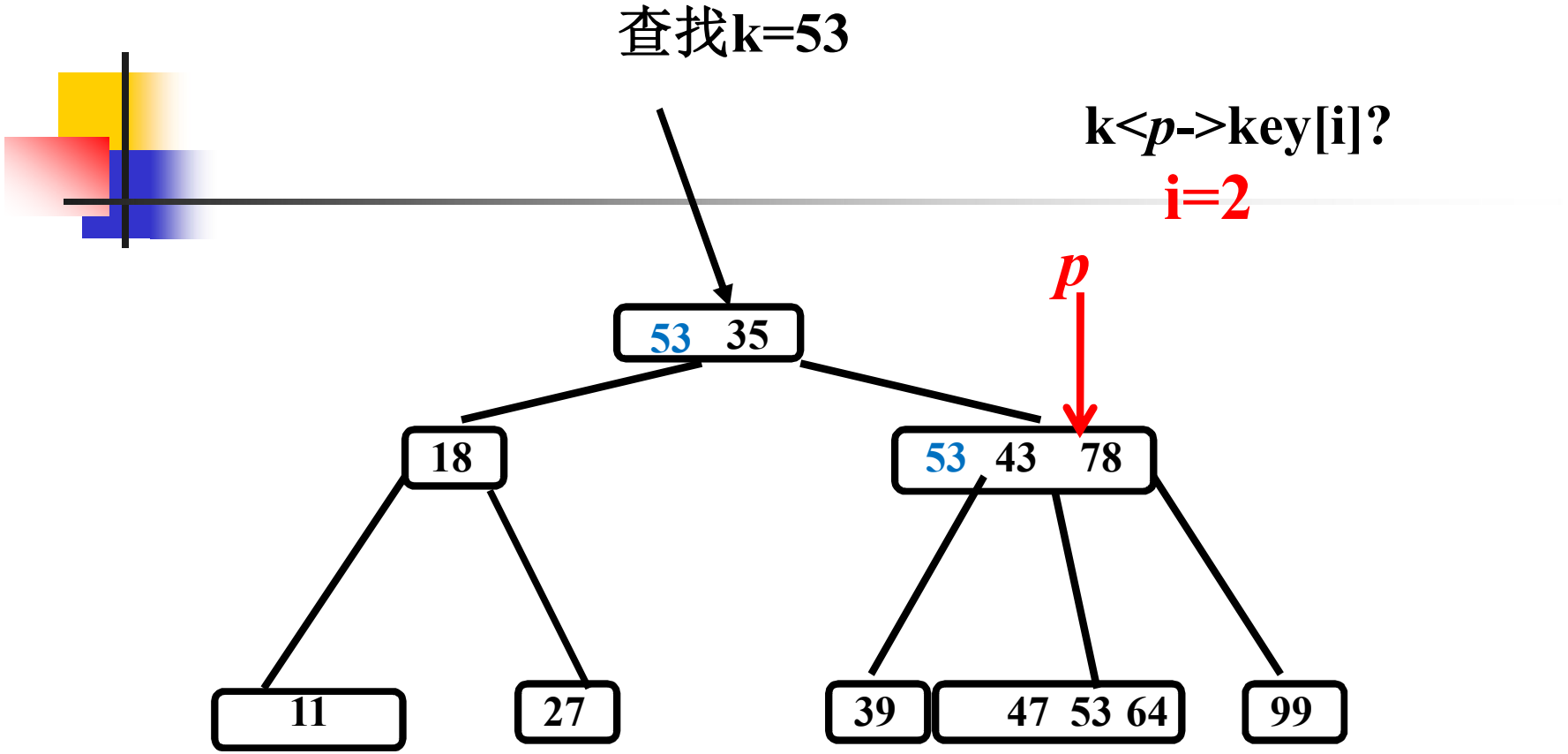
蓝色字体内容为哨兵项

一棵4阶的B-树



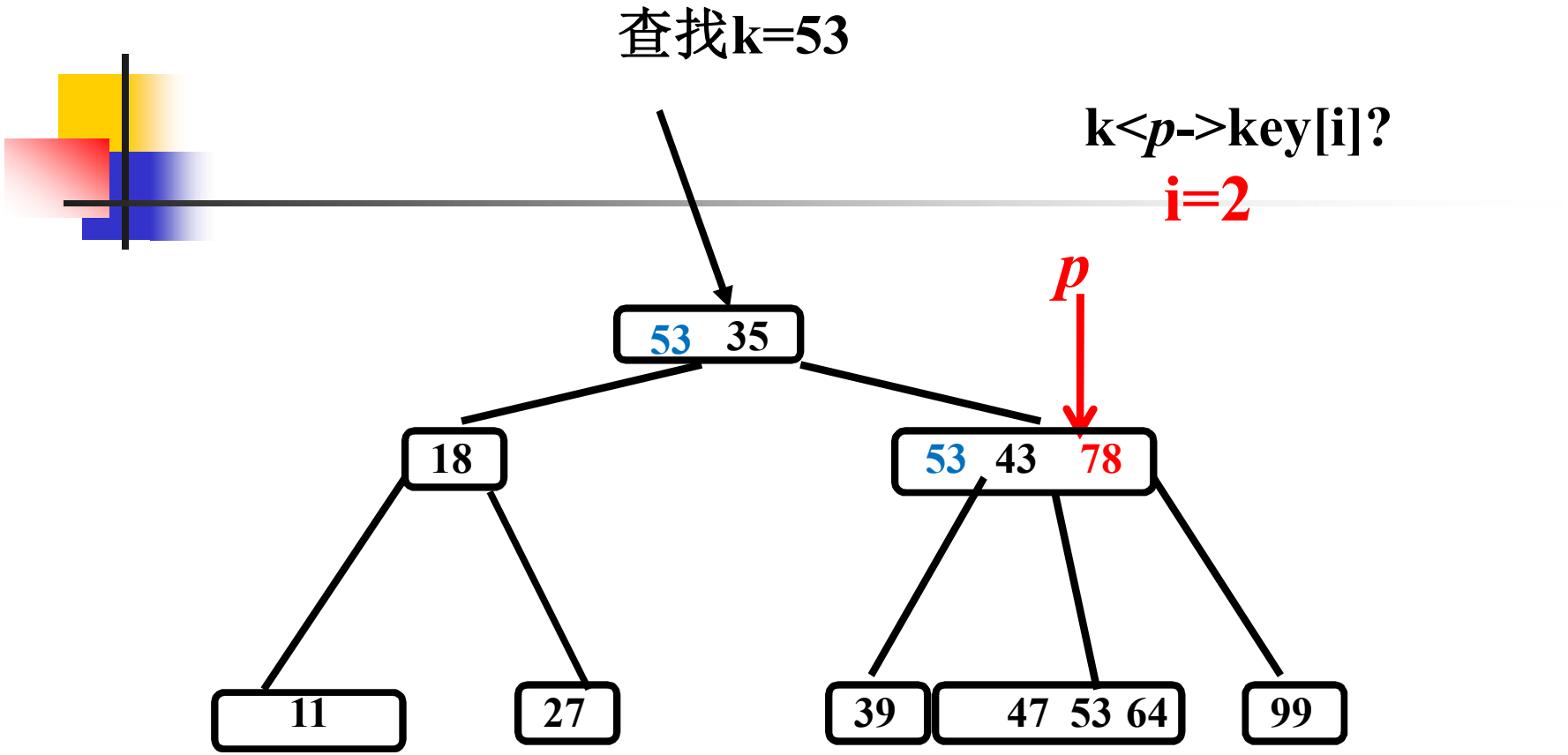
蓝色字体内容为哨兵项

一棵4阶的B-树

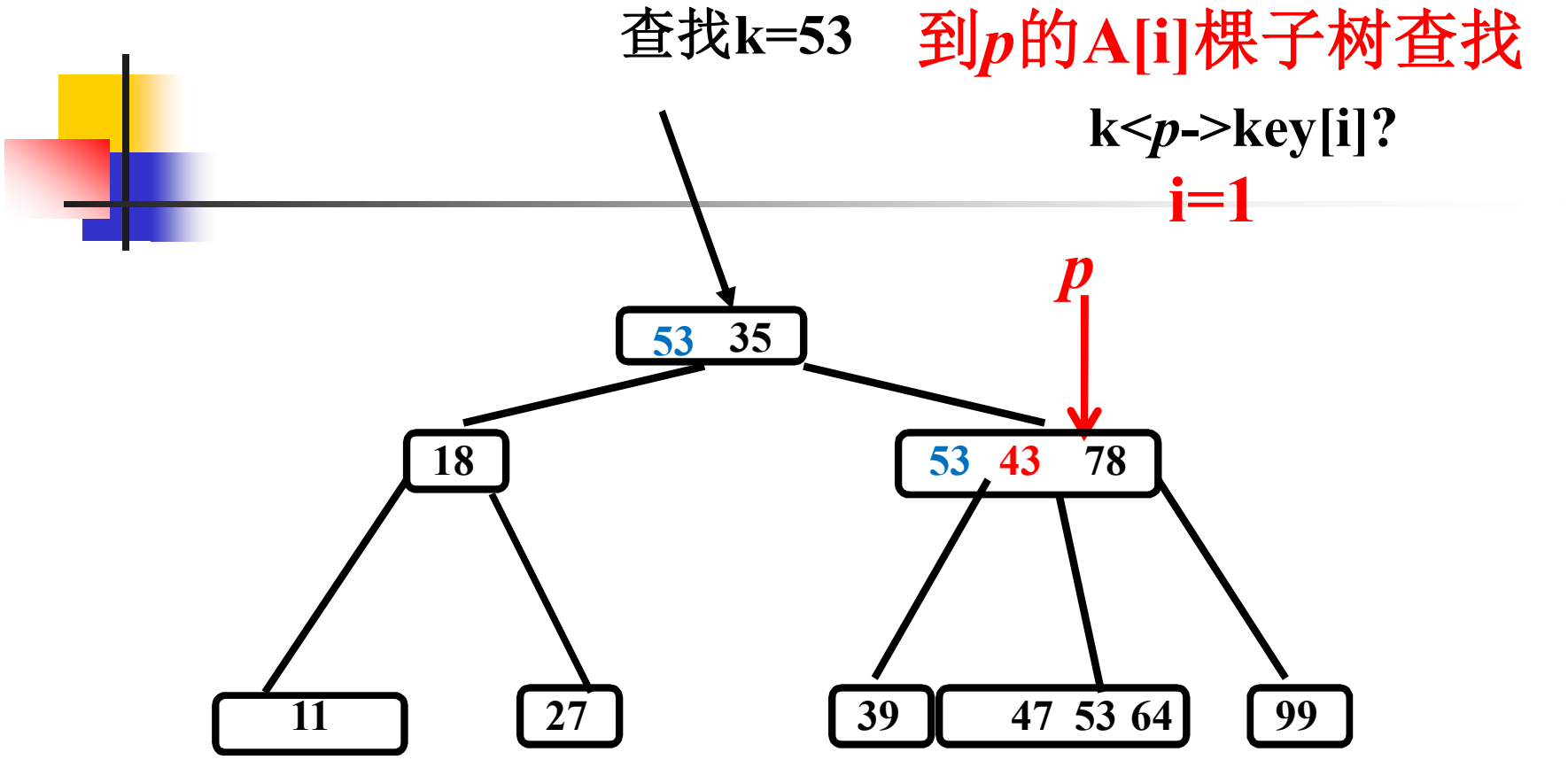


蓝色字体内容为哨兵项

一棵4阶的B-树

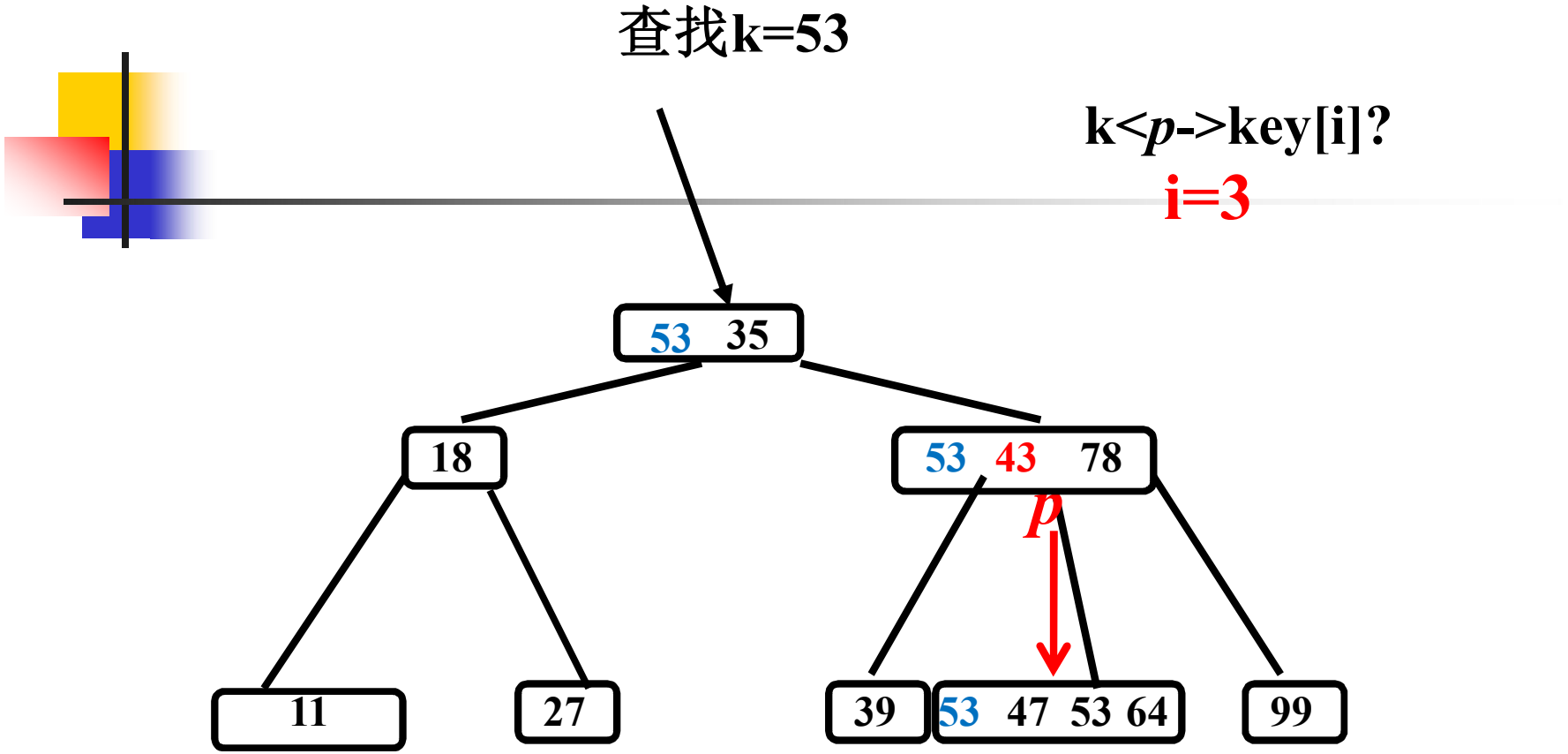


一棵4阶的B-树



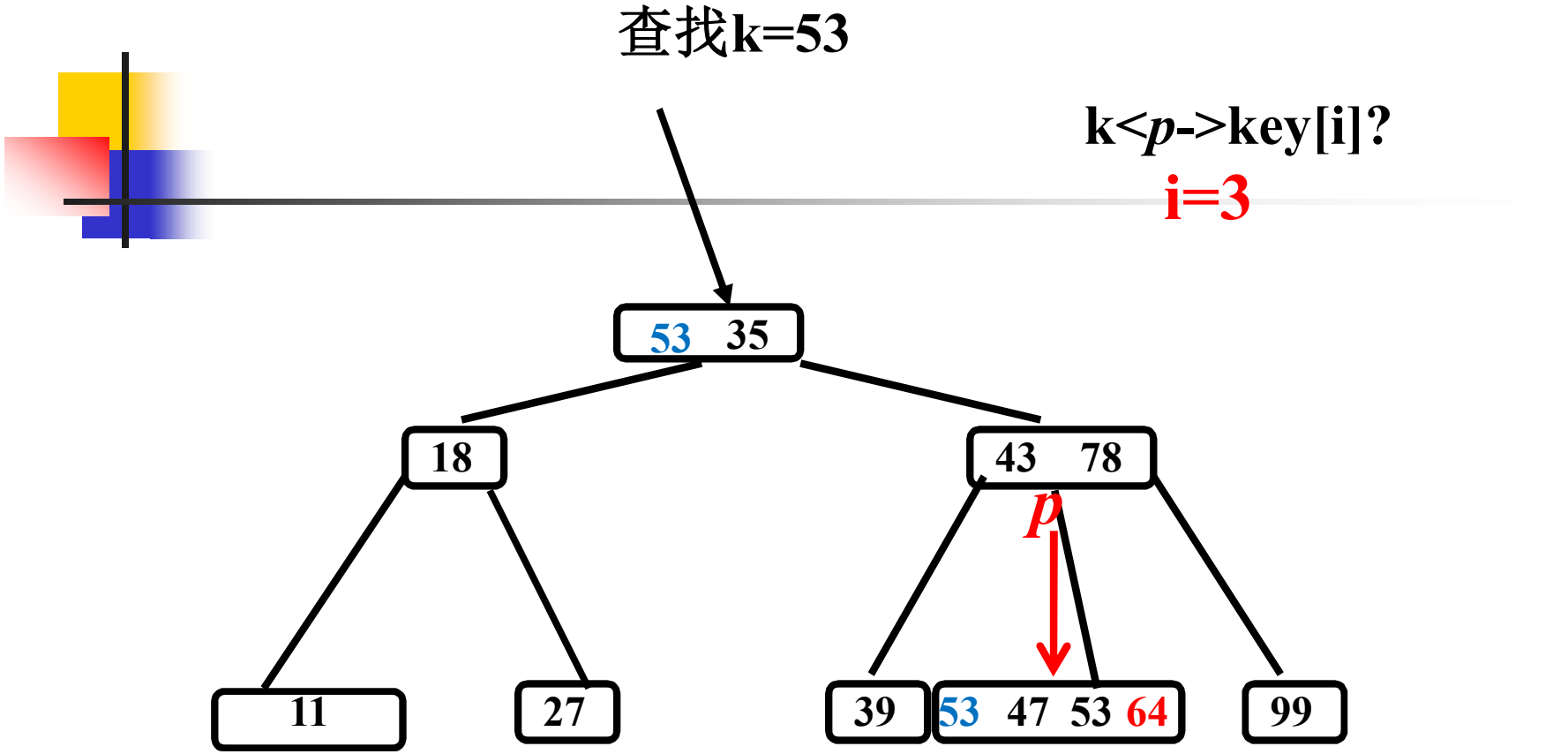
蓝色字体内容为哨兵项

一棵4阶的B-树



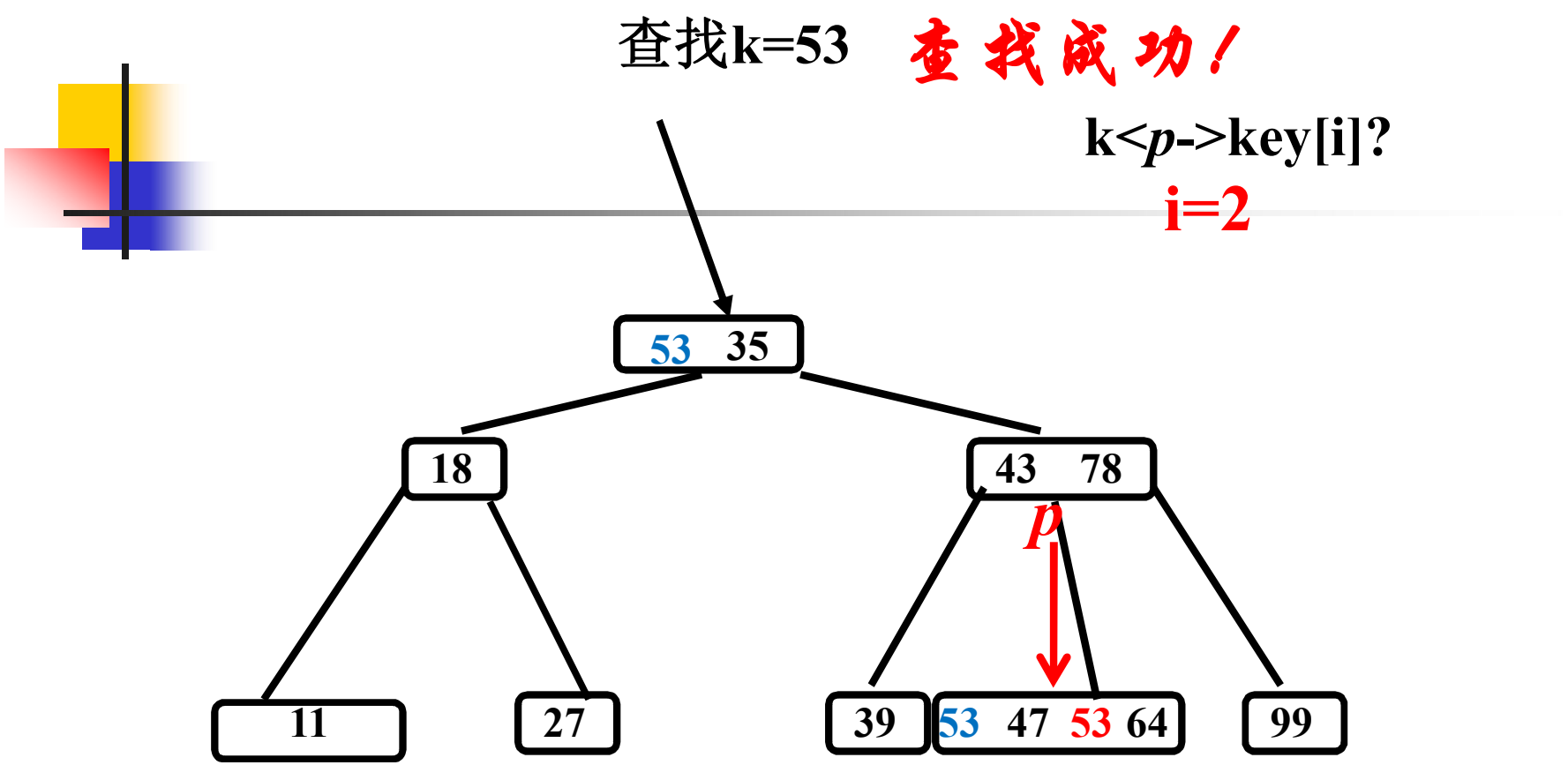
蓝色字体内容为哨兵项

一棵4阶的B-树



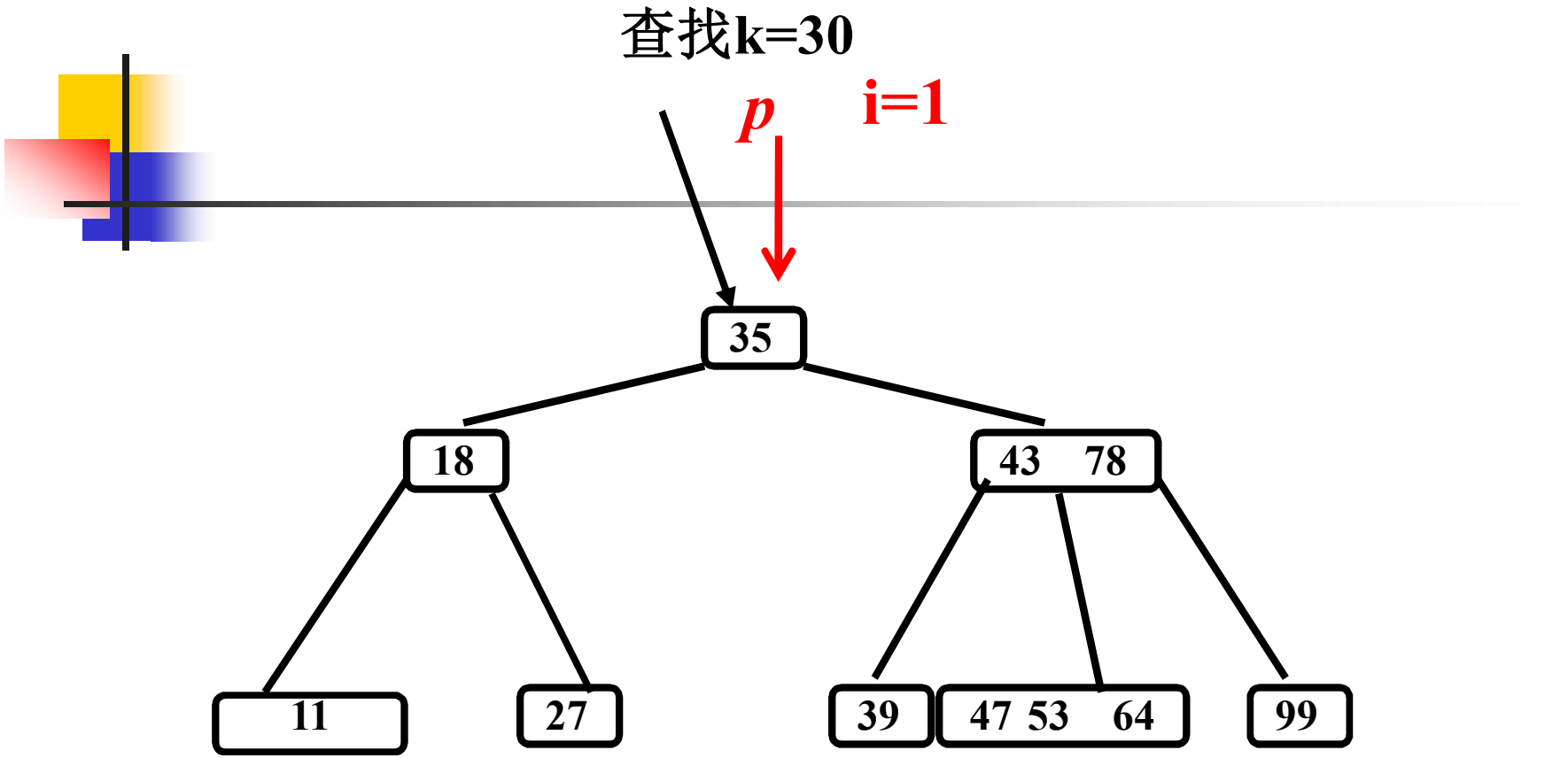
蓝色字体内容为哨兵项

一棵4阶的B-树



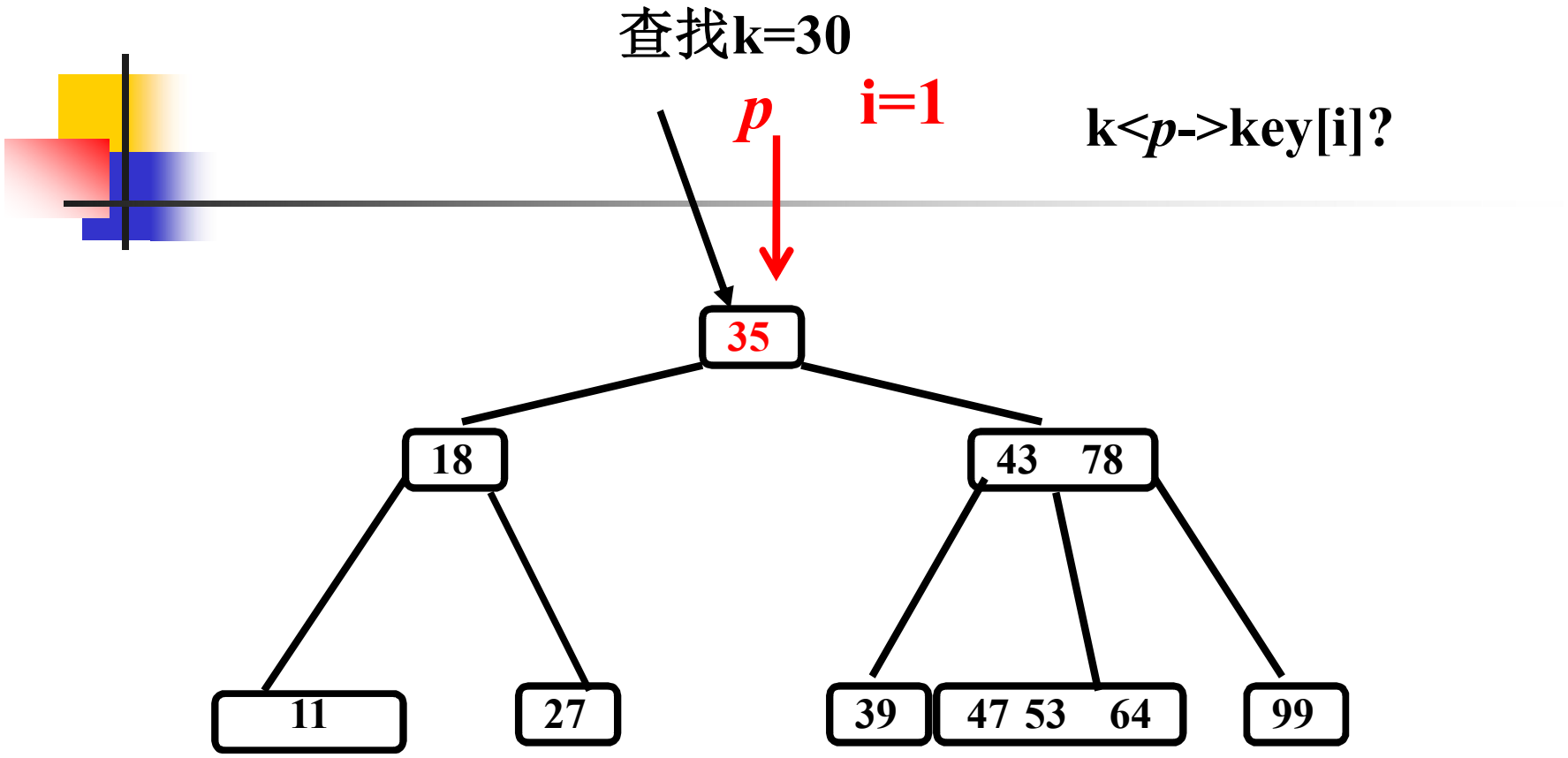
蓝色字体内容为哨兵项

一棵4阶的B-树



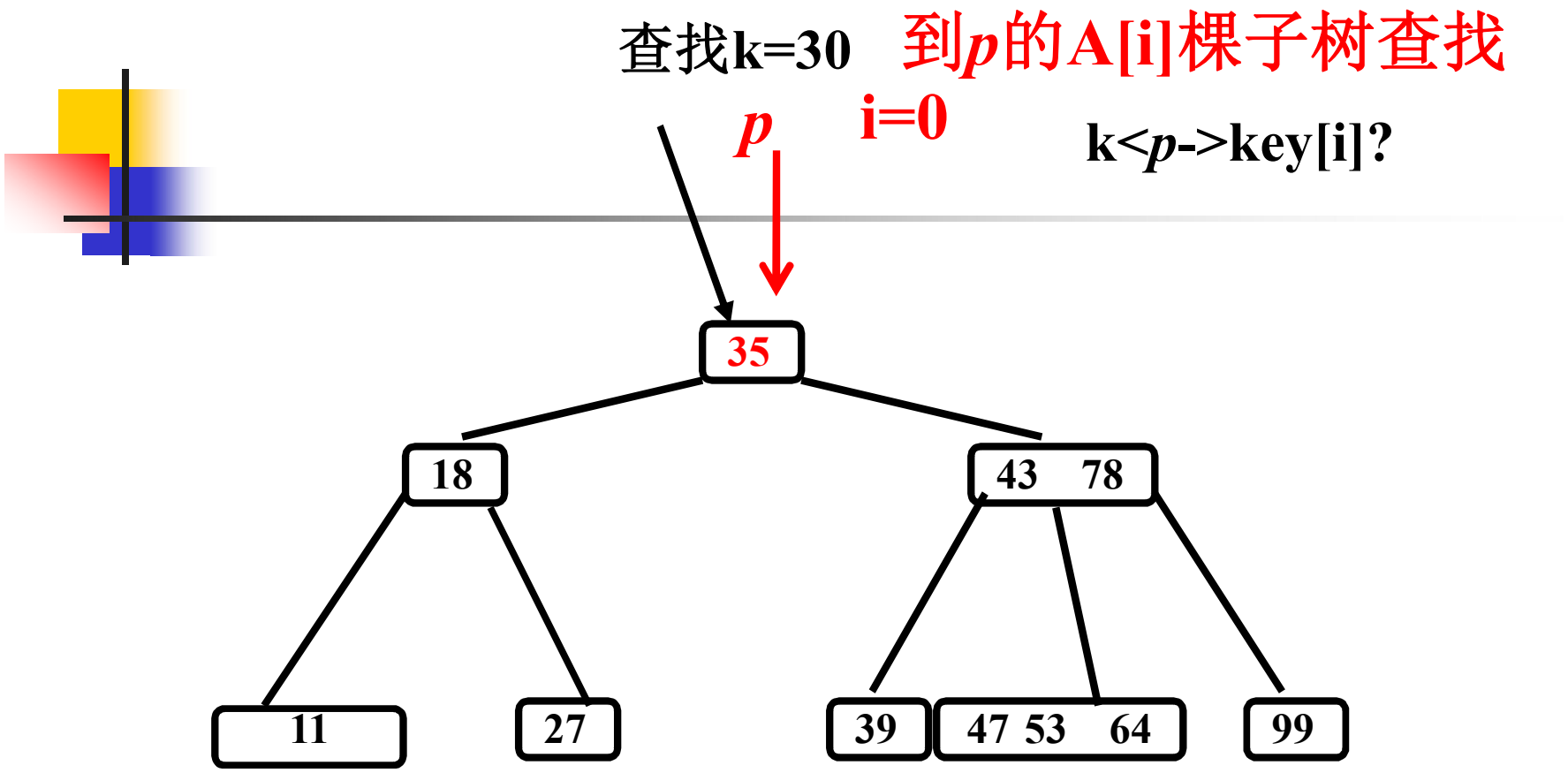
蓝色字体内容为哨兵项

一棵4阶的B-树



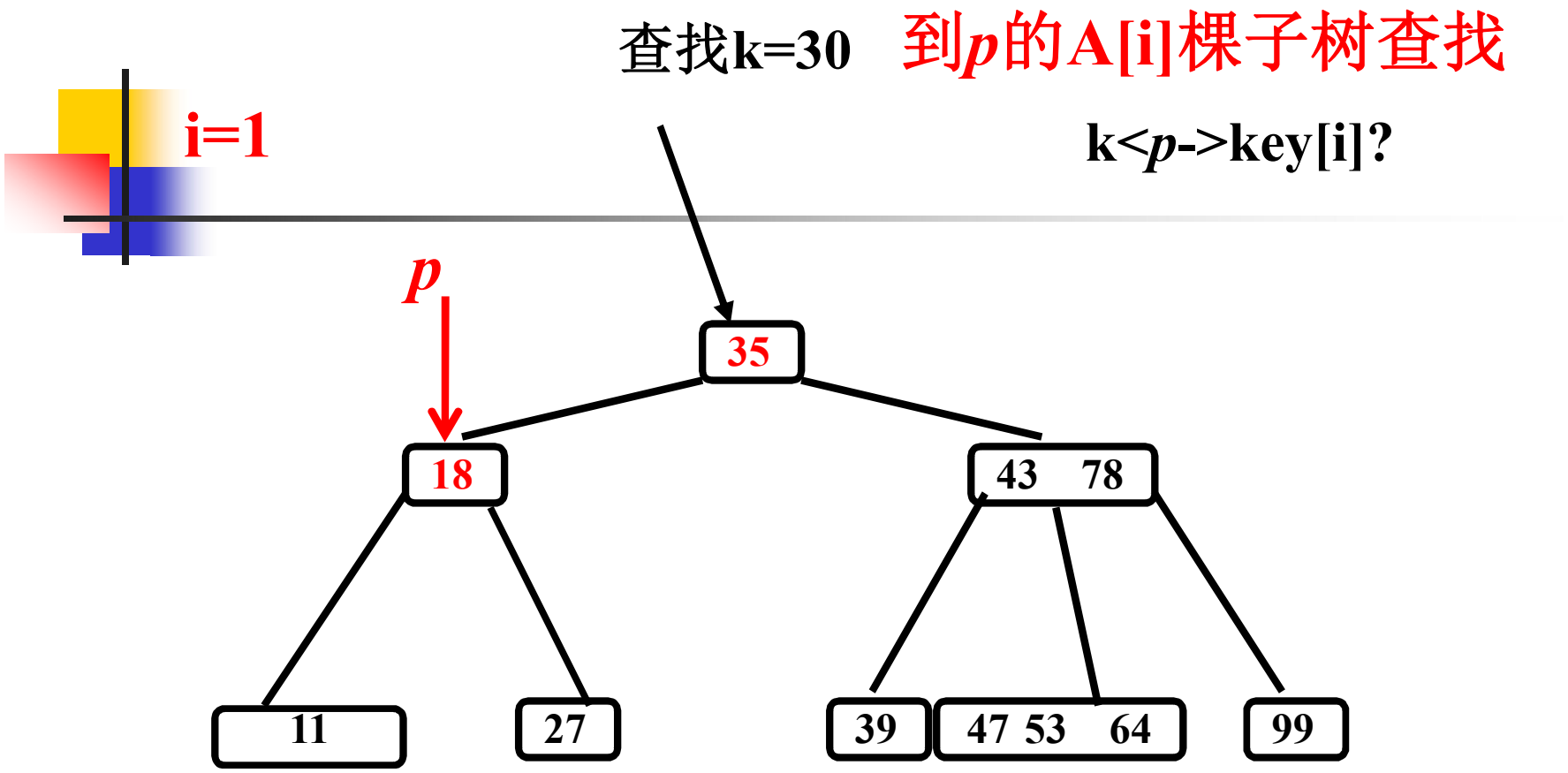
蓝色字体内容为哨兵项

一棵4阶的B-树



蓝色字体内容为哨兵项

一棵4阶的B-树



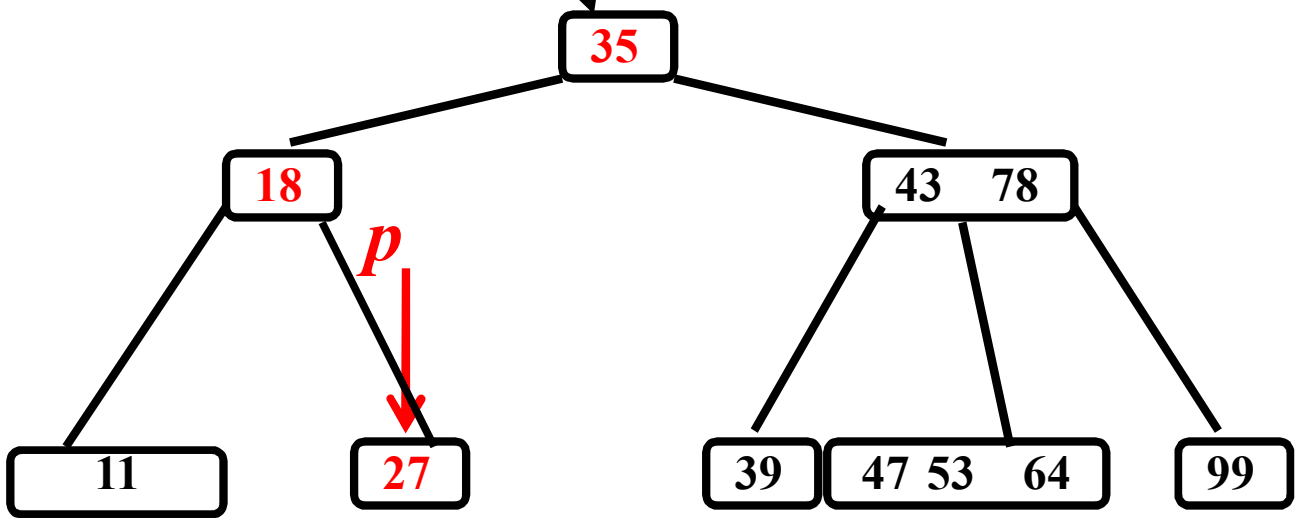
蓝色字体内容为哨兵项

一棵4阶的B-树

查找k=30　到*p*的A[i]棵子树查找

i=1

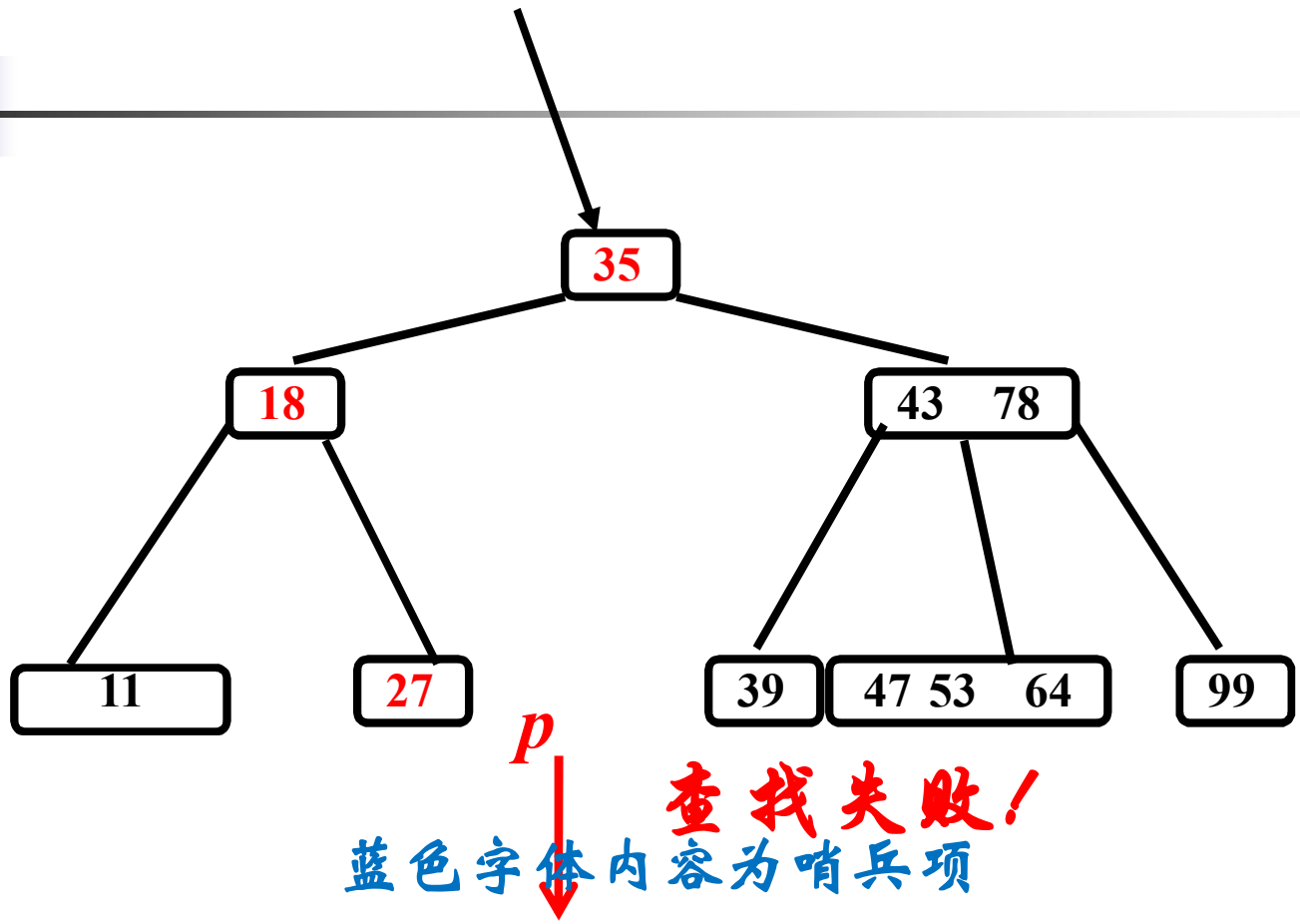
k<*p*->key[i]?



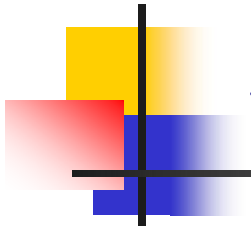
蓝色字体内容为哨兵项

一棵4阶的B-树

查找30



一棵4阶的B-树



查找结果

- typedef struct{
  BTNode * pt;
  int i;
  int tag;
  }Result;

```
Result  SearchBTree(BTree T, KeyType k)
{
    BTNode *p=T, q=NULL;
    while(p!=NULL)
    {   i=p->keynum;
        p->key[0]=k;
        while(k<p->key[i])   i--;
        if(k==p->key[i] && i>0) return (p,i,1);
        else {   q=p; p=p->ptr[i];  }
    }
   return (q,i,0);
}
```

在结点中找关键字

在B-树中找结点

B-树查找分析

- 两种基本操作：

(1) 在B-树中找结点　　----磁盘上进行

(2) 在结点中找关键字

B-树查找分析

- 通常B-树是存储在外存上的，操作(1)就是通过指针在磁盘相对定位，将结点信息读入内存，之后，再对结点中的关键码有序表进行顺序查找或折半查找。因为，在磁盘上读取结点信息比在内存中进行关键码查找耗时多，所以，在磁盘上读取结点信息的次数，即B-树的层次树是决定B-树查找效率的首要因素。

B-树查找分析

- *n*个关键字的*m*阶B-树，最坏情况下达到多深呢?
- 根结点到关键字所在结点的路径上涉及的结点数不超过

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

B-树查找分析

- 深度为$h+1$的m阶B-树**至少**含有多少个结点?
- 第1层　　1
- 第2层　　2
- 第3层　　$2*\lceil m/2 \rceil$
- 第4层　　$2*\lceil m/2 \rceil^2$
- ……
- 第$h+1$层　$2*\lceil m/2 \rceil^{h-1}$

B-树查找分析

- *m*阶B-树的深度为*h*+1,第*h*+1层为叶子结点，含有*n*个关键字,则叶子结点必为*n*+1
- $n+1 \geq 2 * \lceil m/2 \rceil^{h-1}$
- $h-1 \leq \log_{\lceil m/2 \rceil}((n+1)/2)$
- $h \leq \log_{\lceil m/2 \rceil}((n+1)/2)+1$

B-树的插入

- 在B-树上插入*x*，首先在B-树上进行查找，确定插入位置，然后进行插入。
- 插入不是在叶结点上进行的，而是在最底层的某个非叶（终端结点）中添加一个关键字
- 若插入后该结点上关键字个数不超过*m*-1个，则直接插入即可；
- 否则，该结点上关键字个数达到*m*个，因而使该结点的子树超过了*m*棵→进行调整(“分裂”)

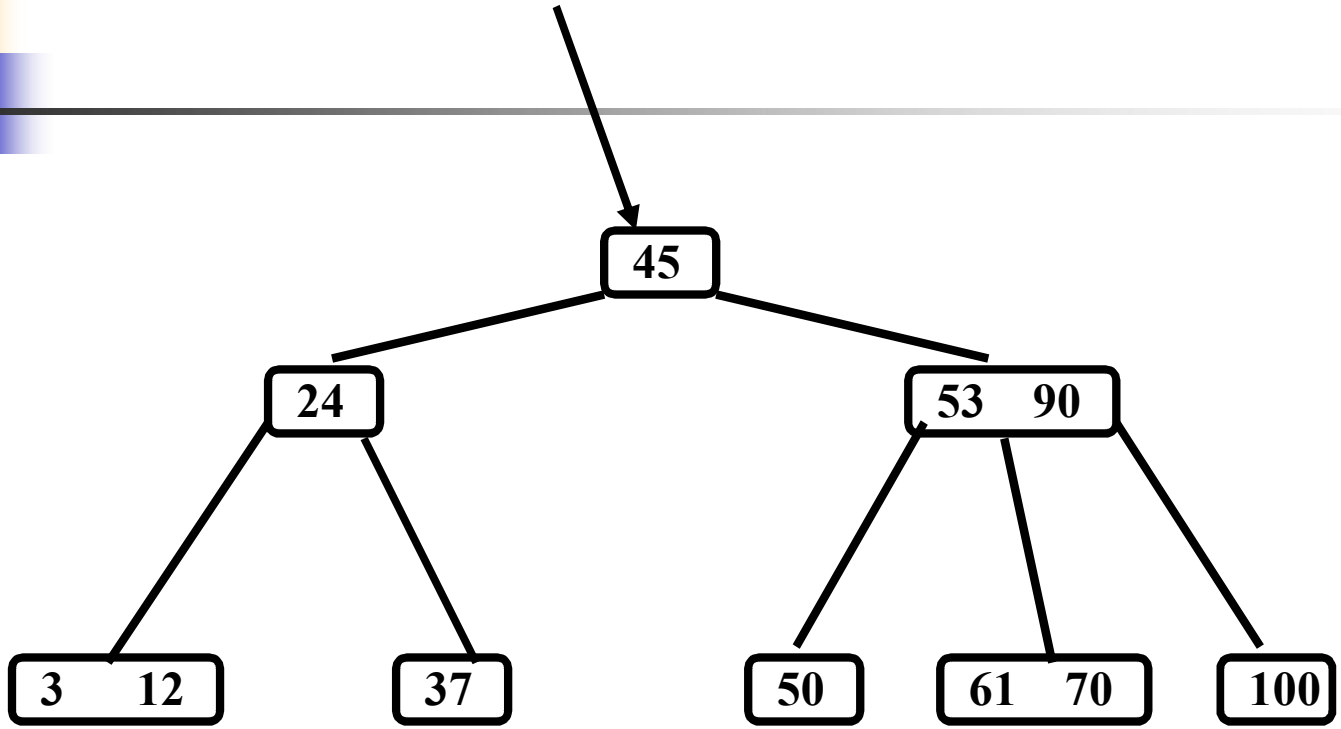
B-树的插入

- **分裂**方法为：一个结点插入一个新的关键字后为:

$$(m, A_0, K_1, A_1, \ldots, K_m, A_m),$$

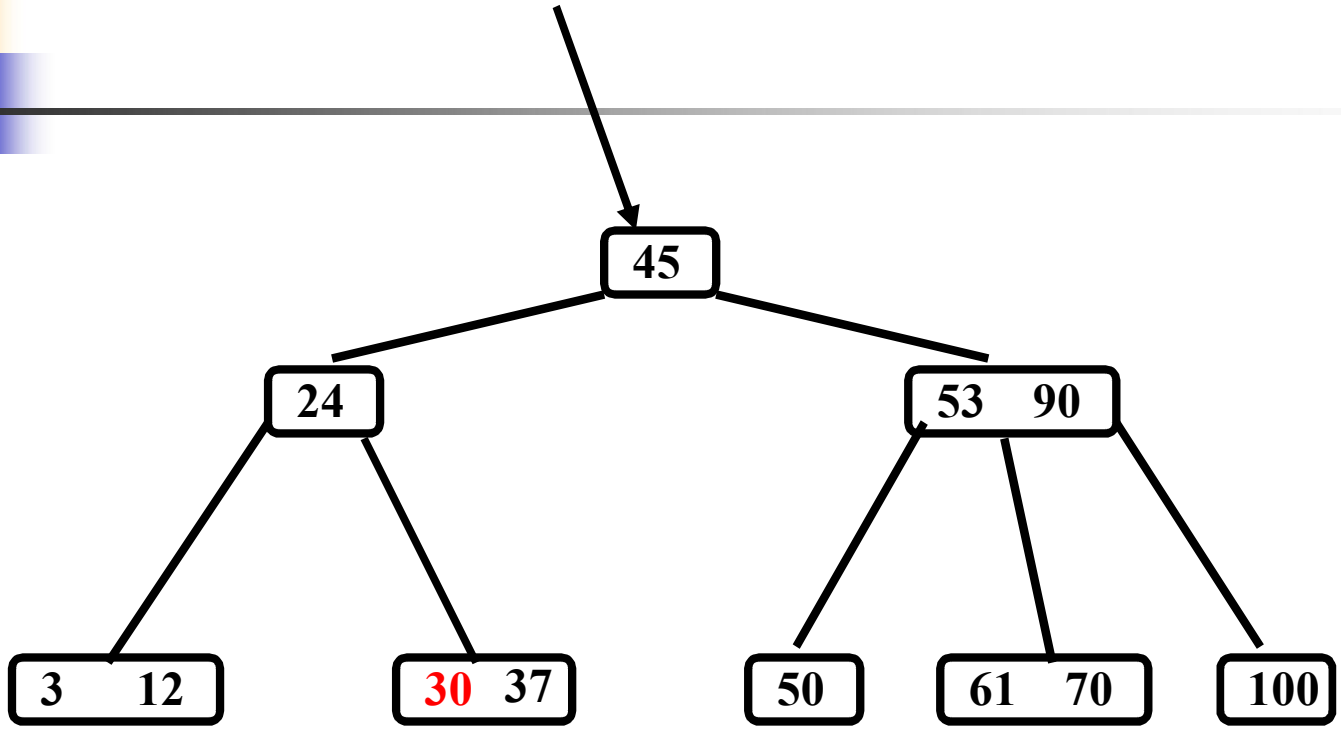
将其分裂为两个结点:

- $(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \ldots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$
- $(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \ldots, K_m, A_m)$
- 并把中间的一个关键字$K_{\lceil m/2 \rceil}$拿出来插入到该结点的双亲结点中去
- **若双亲结点不满足m阶B-树的定义**，就需要再分裂、再往上插，从而可能导致B-树可能朝着根的方向生长。



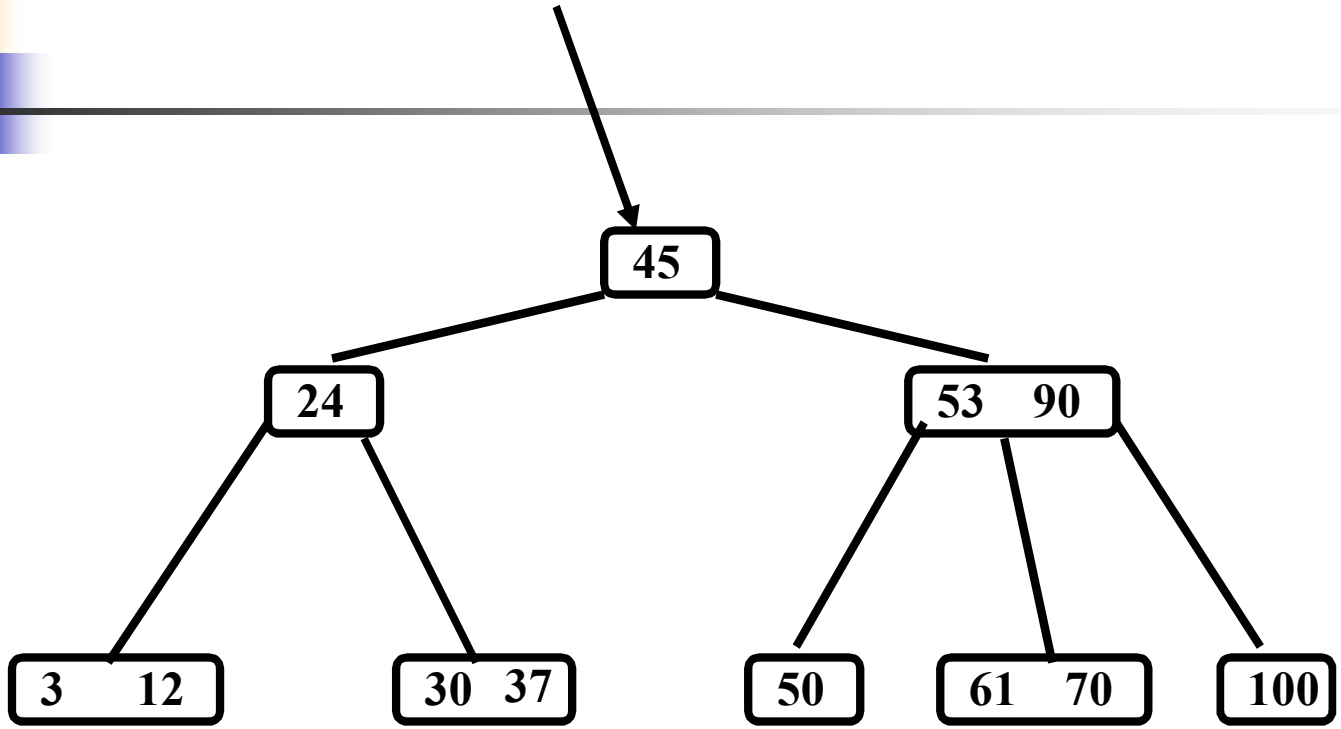
插入30

一棵3阶的B-树



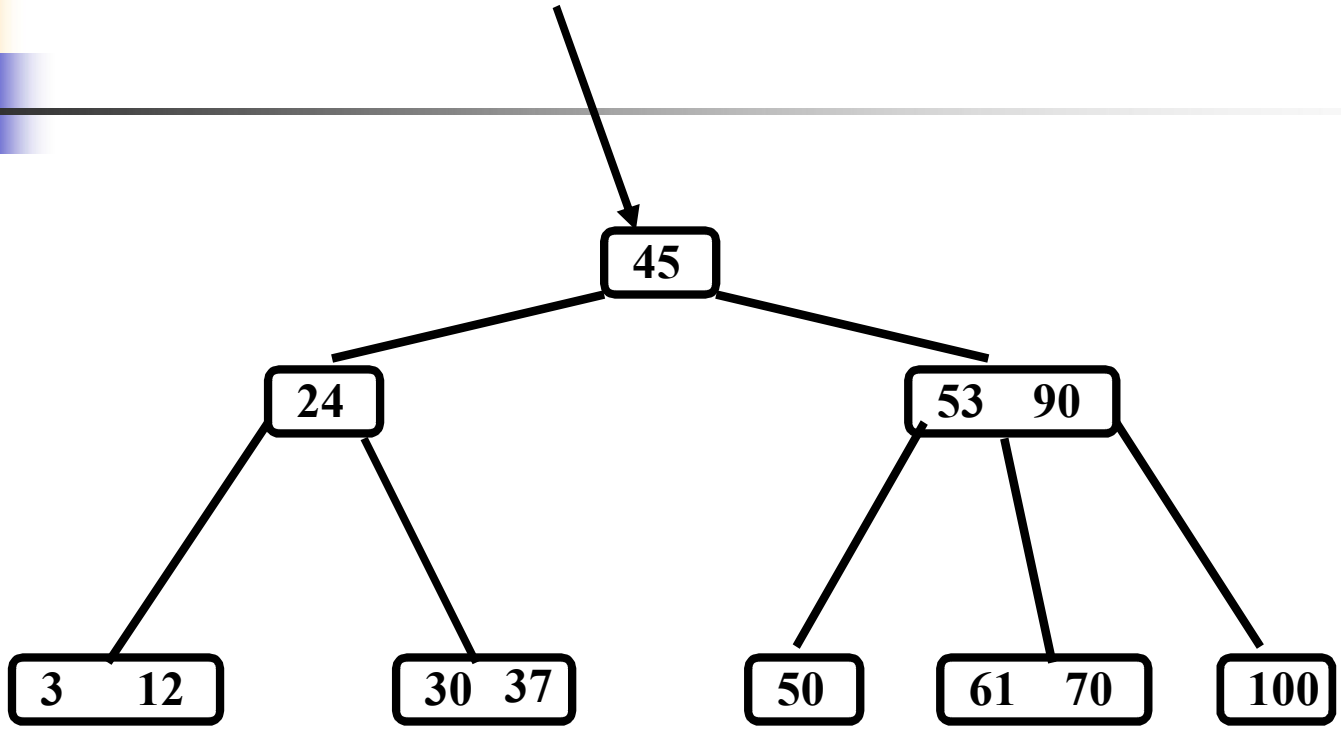
插入30

一棵3阶的B-树



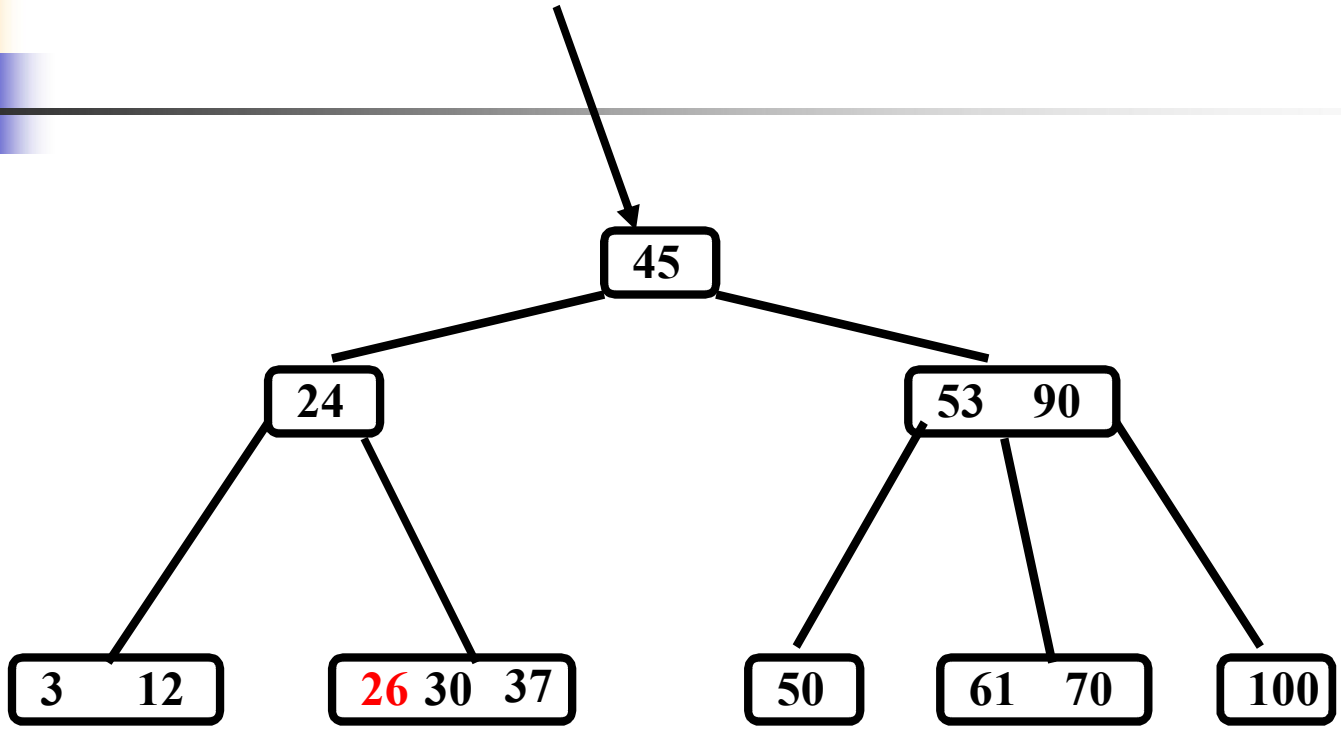
插入30

一棵3阶的B-树



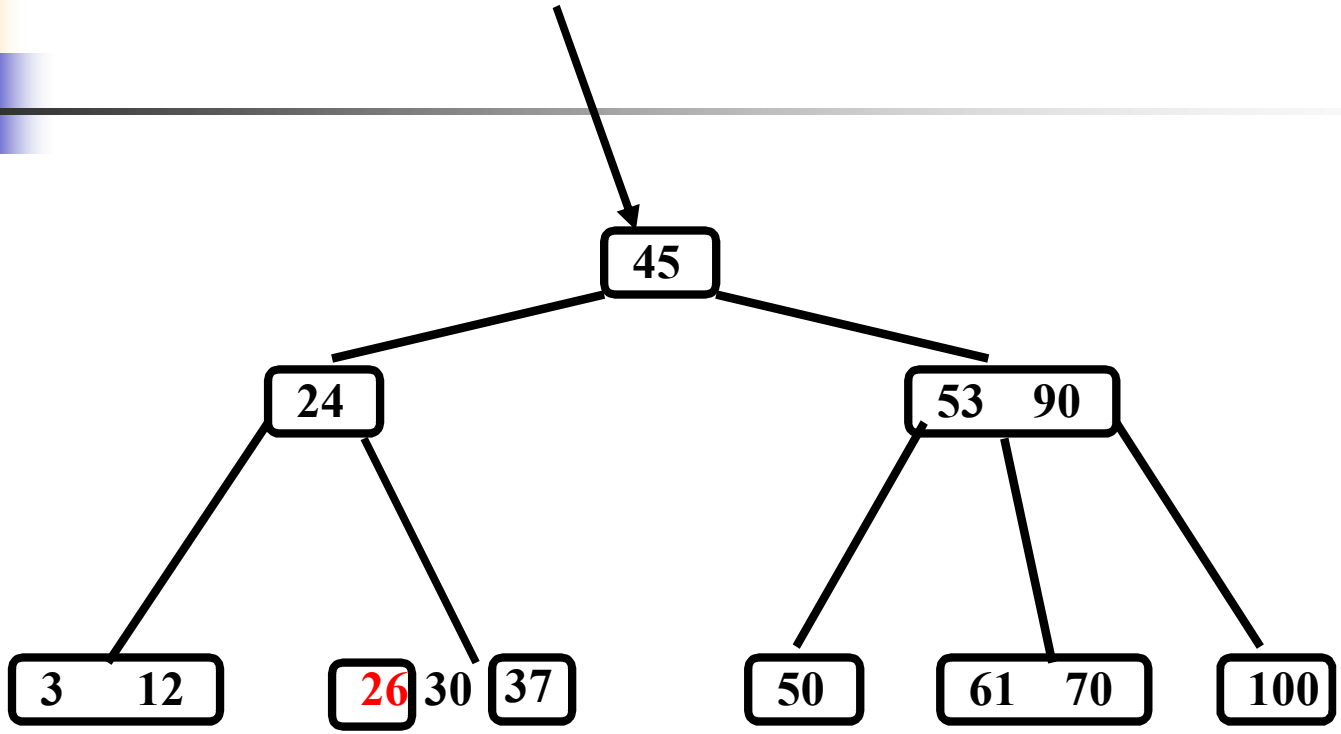
插入26

一棵3阶的B-树



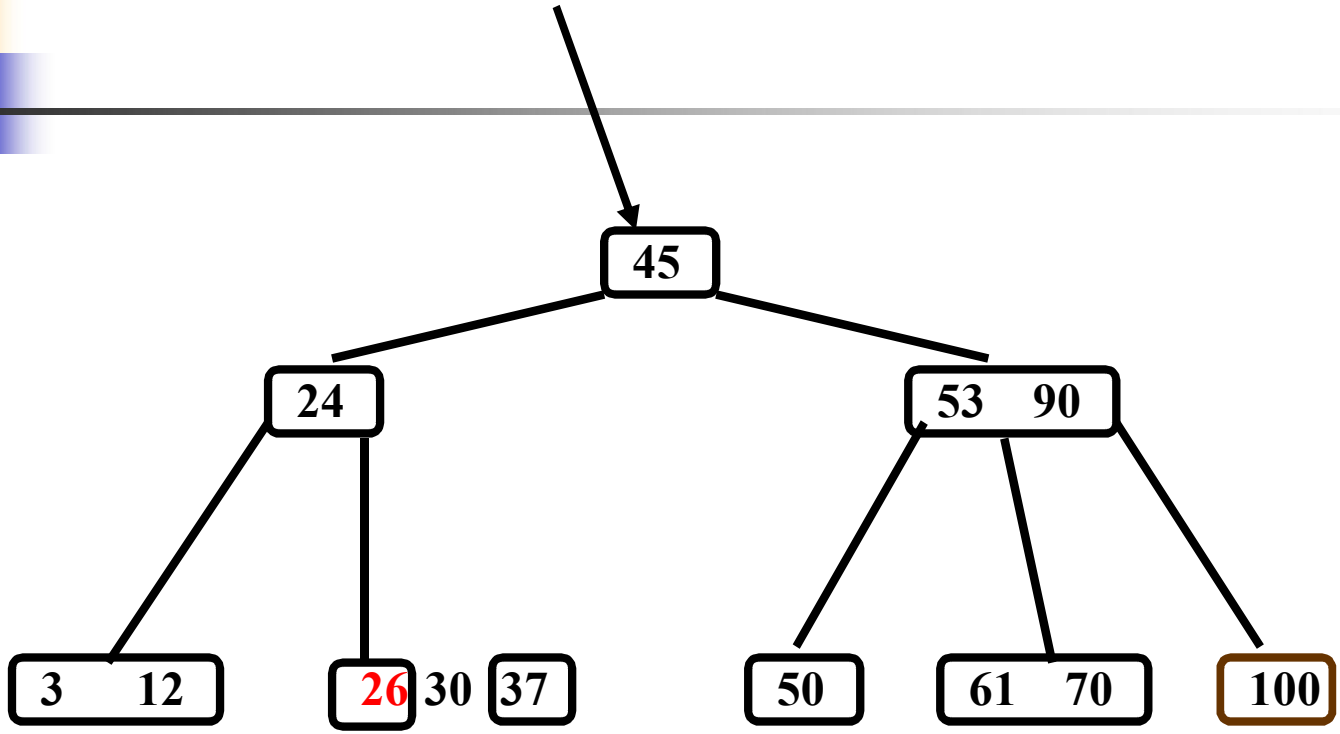
插入26

一棵3阶的B-树



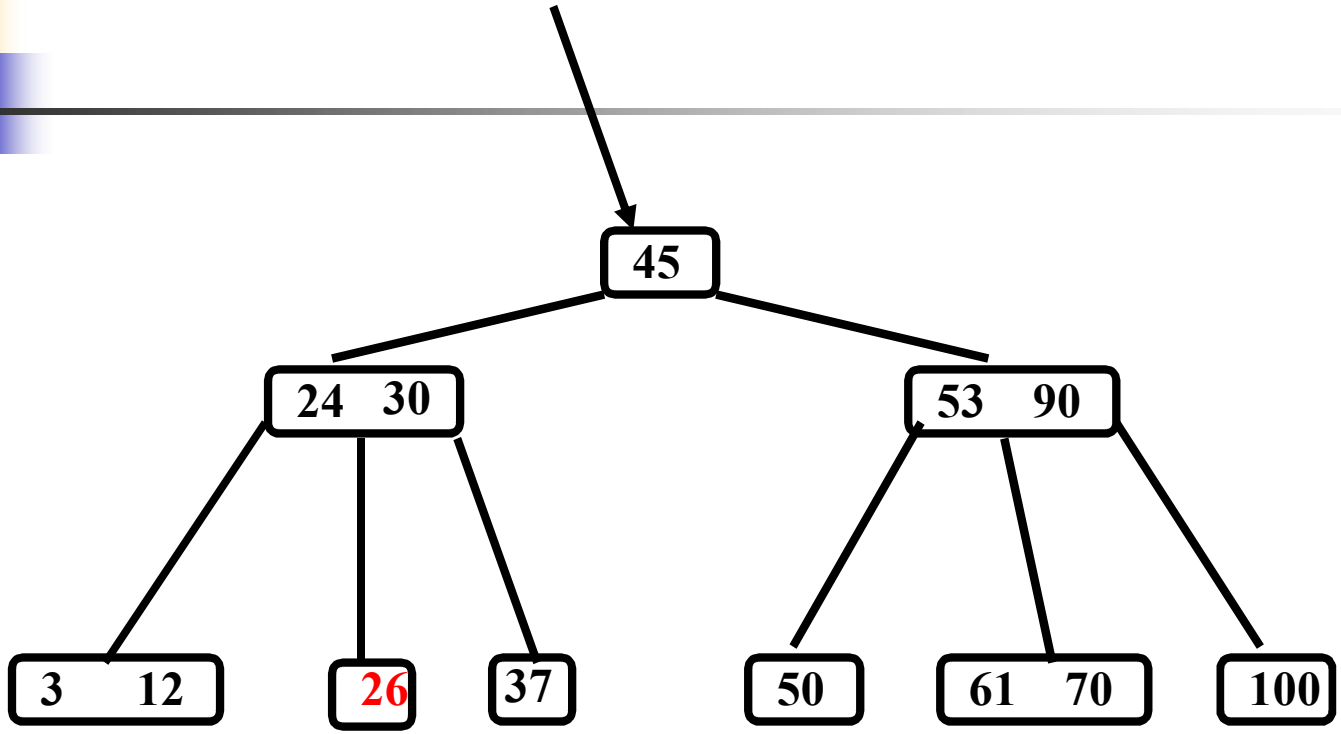
插入26

一棵3阶的B-树



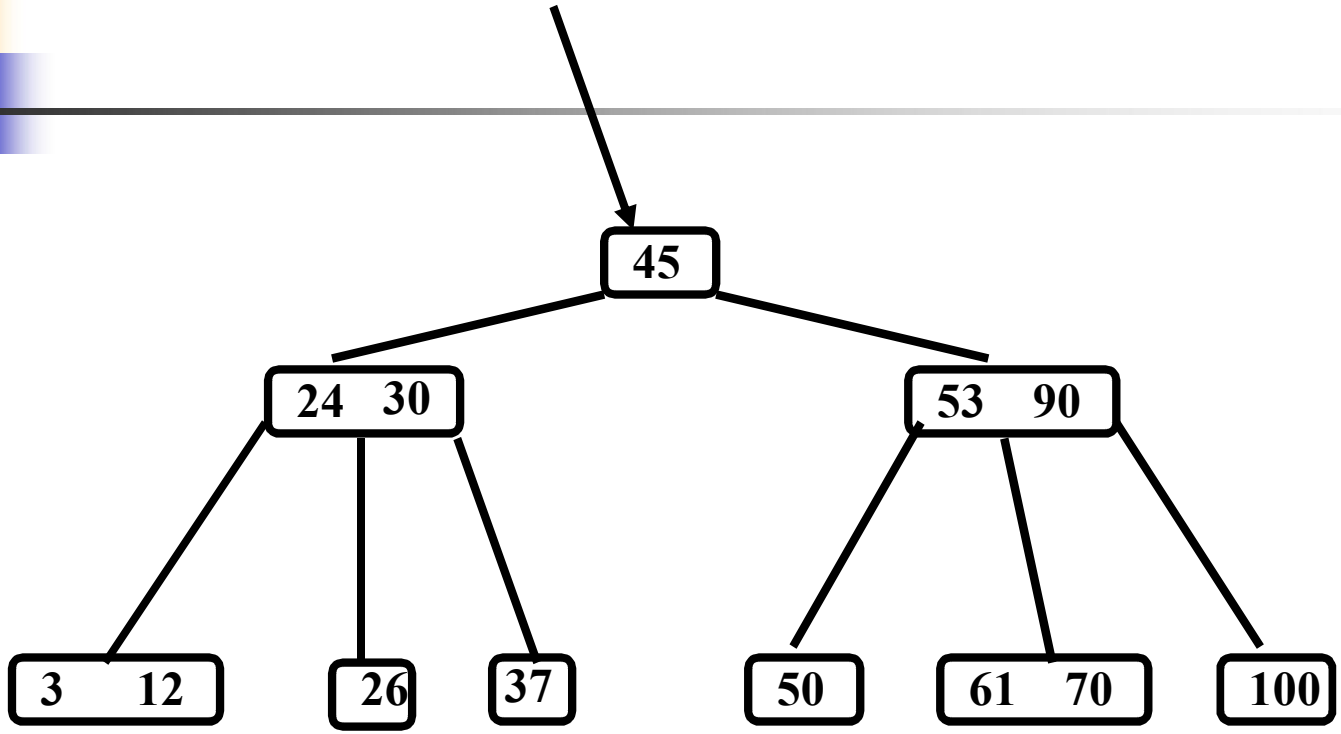
插入26

一棵3阶的B-树



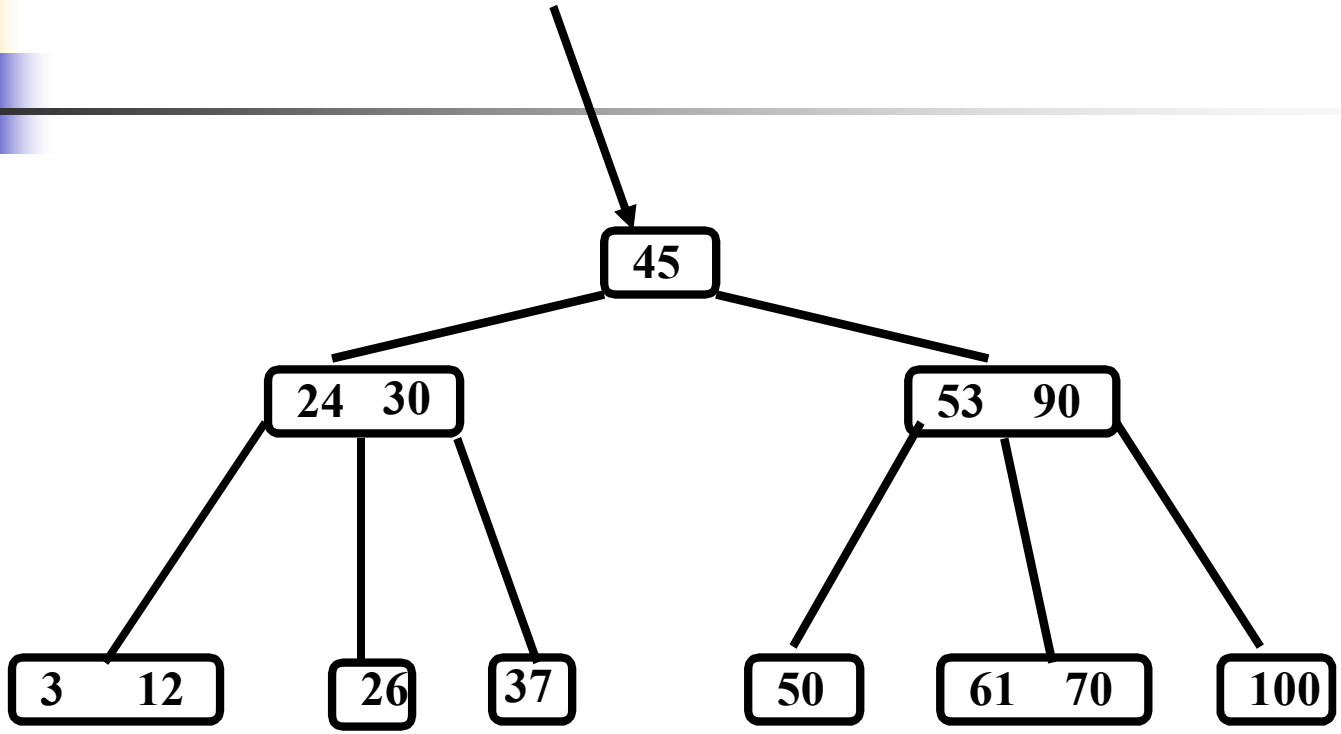
插入26

一棵3阶的B-树



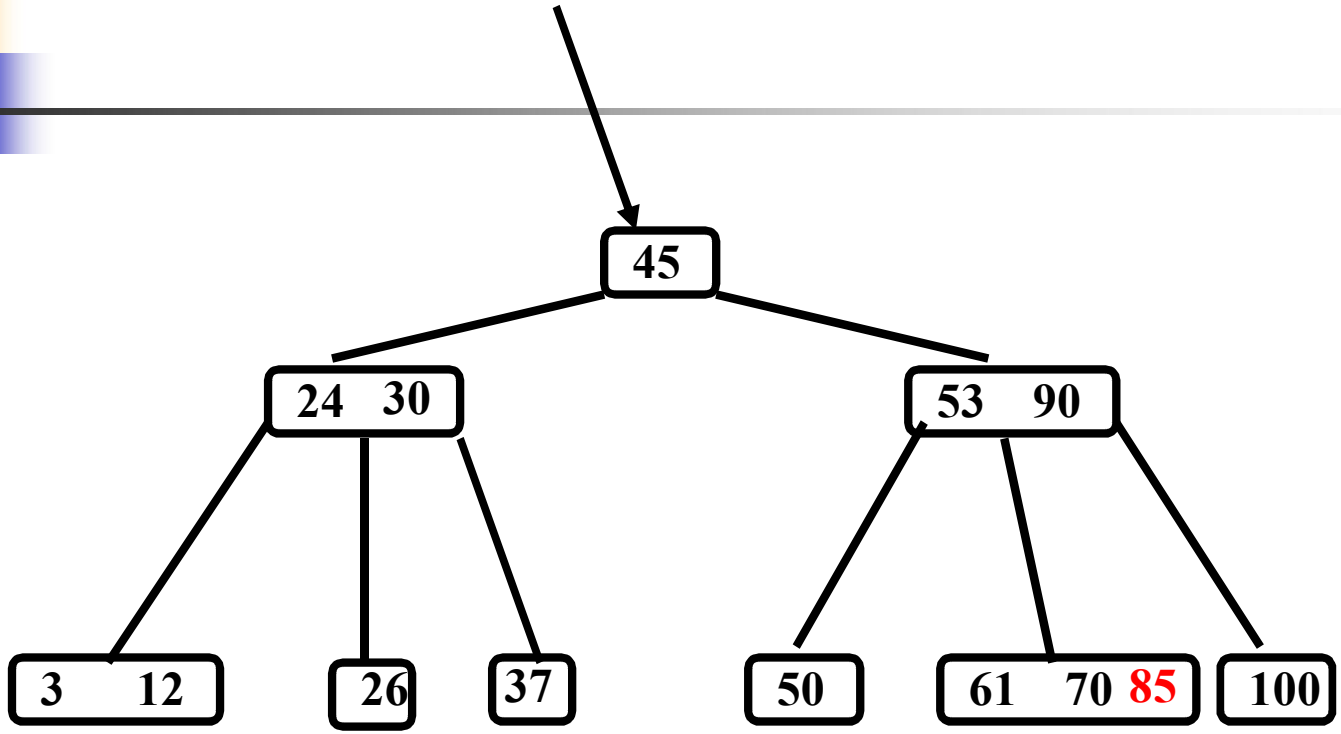
插入26

一棵3阶的B-树



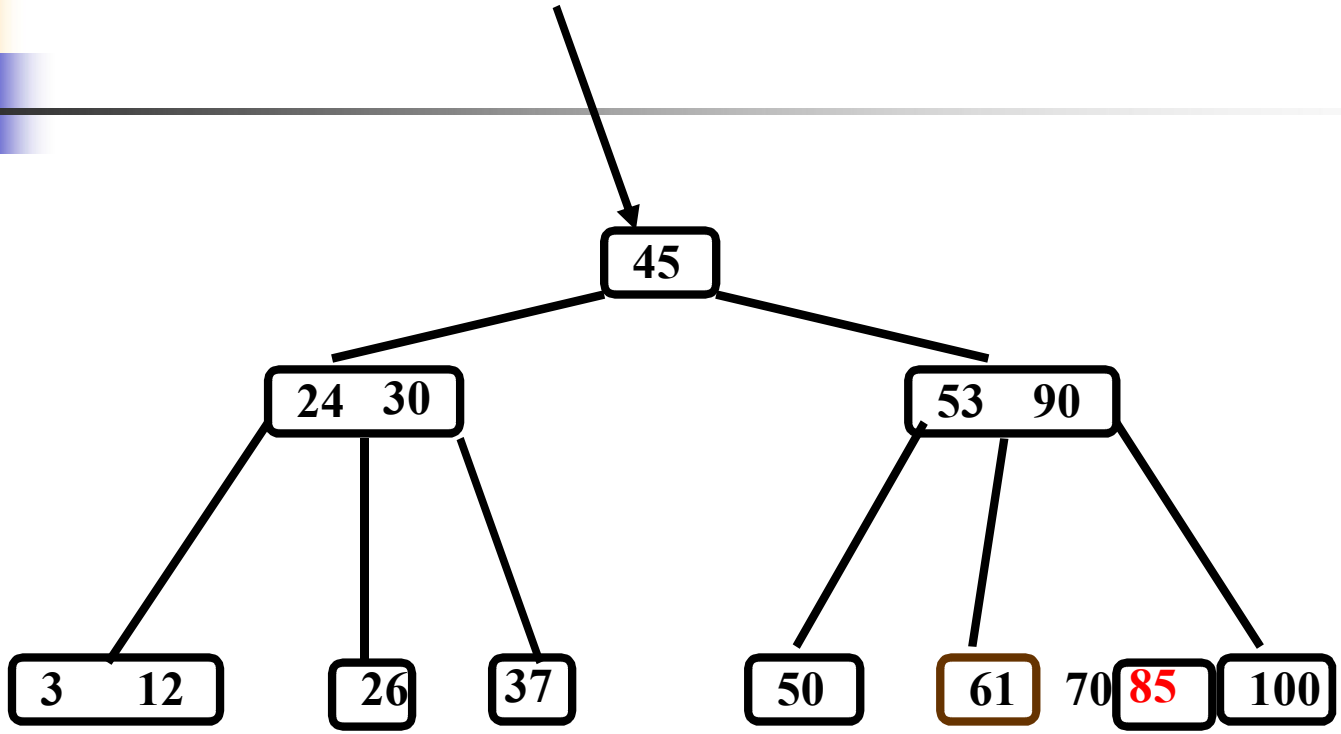
插入85

一棵3阶的B-树



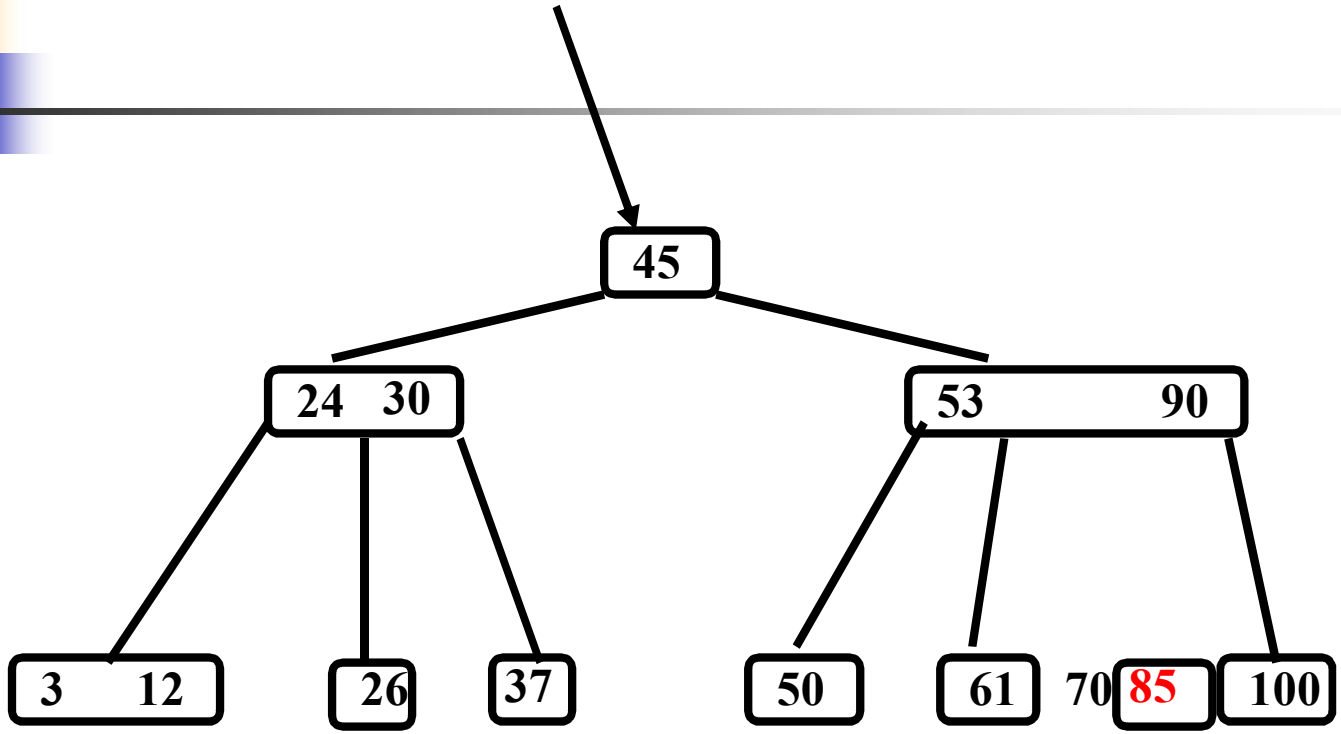
插入85

一棵3阶的B-树



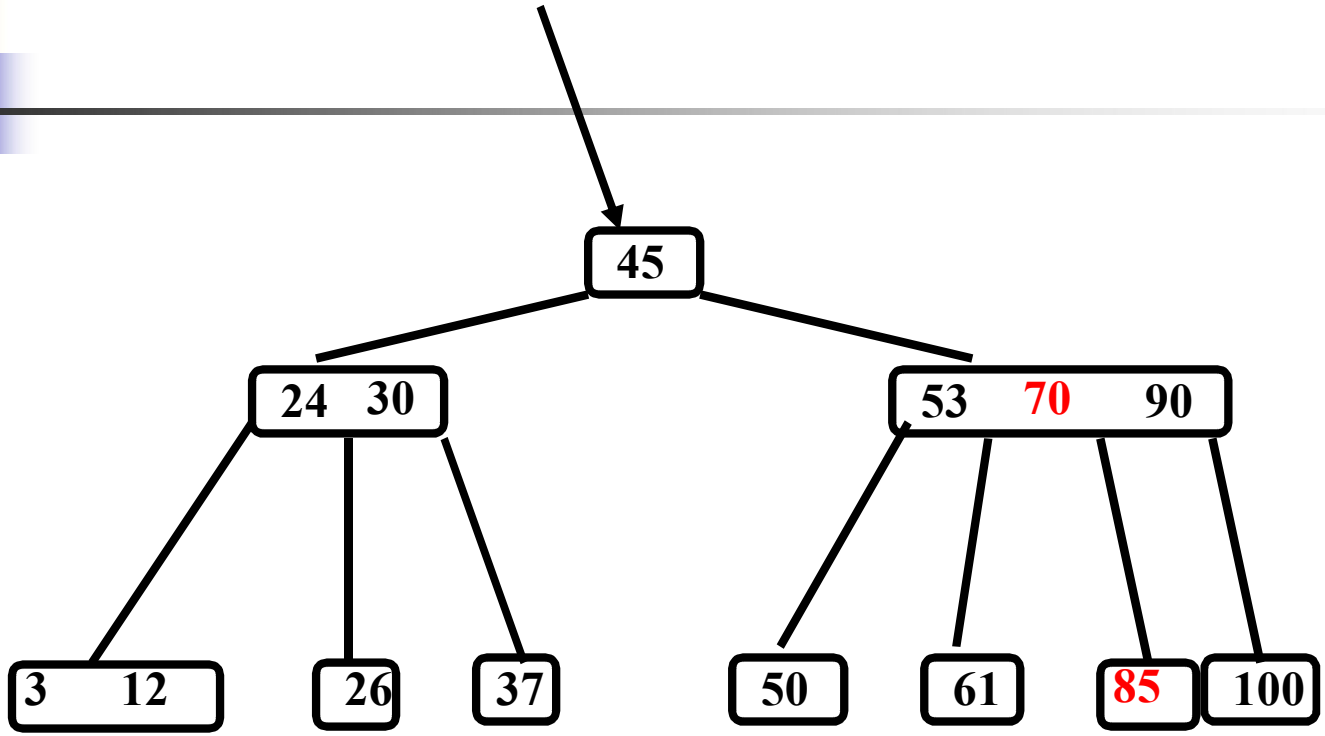
插入85

一棵3阶的B-树



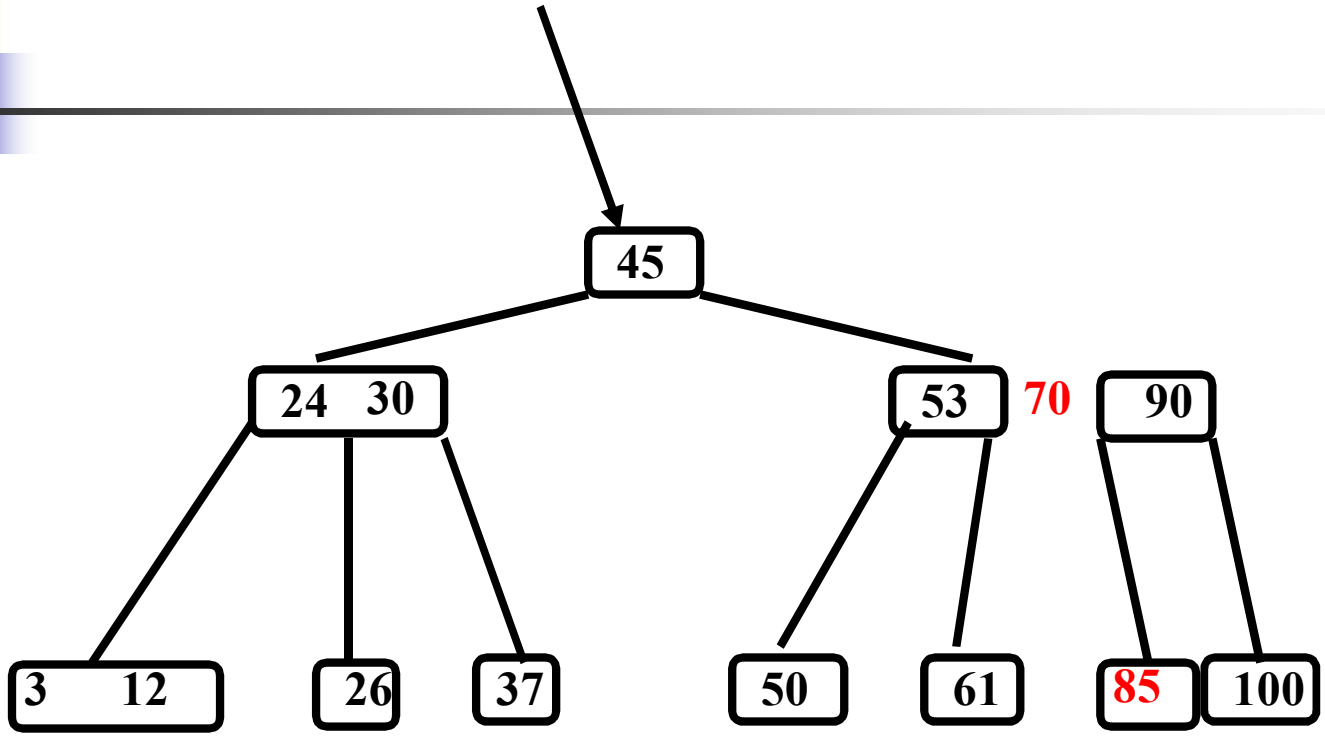
插入85

一棵3阶的B-树



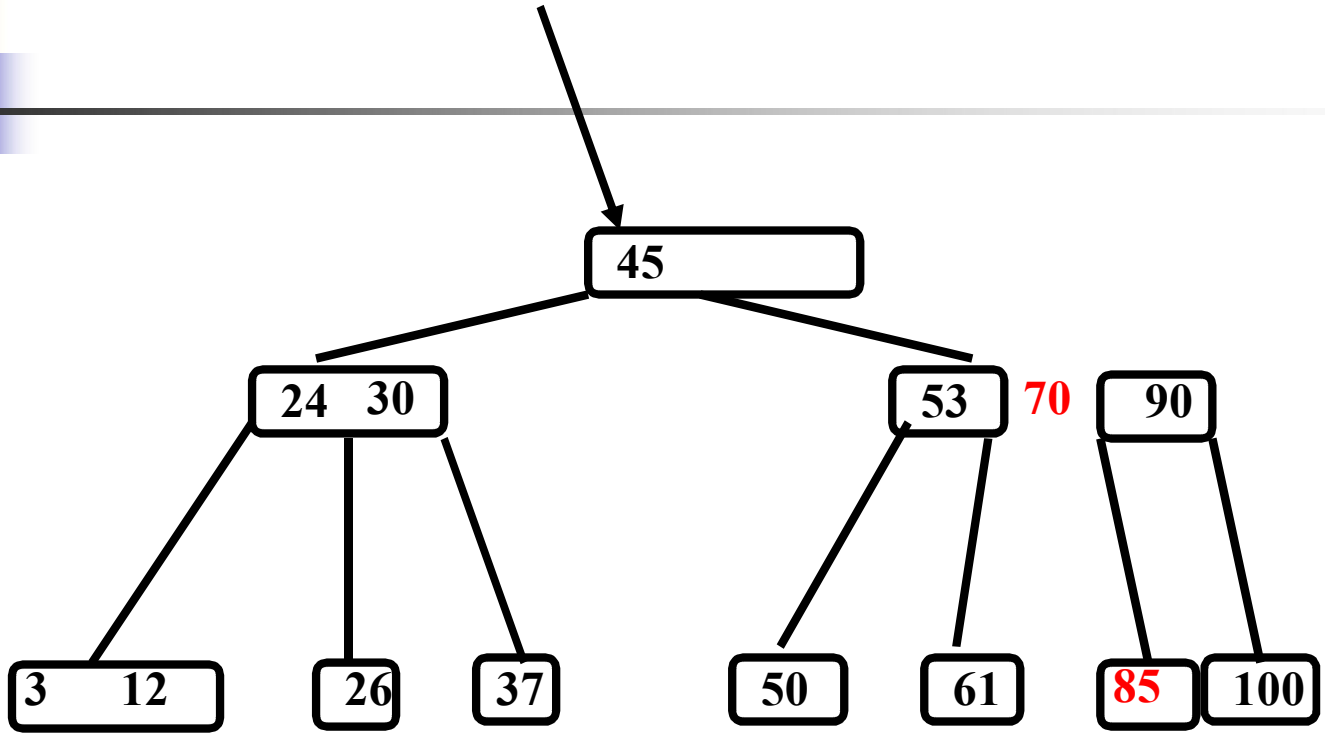
插入85

一棵3阶的B-树



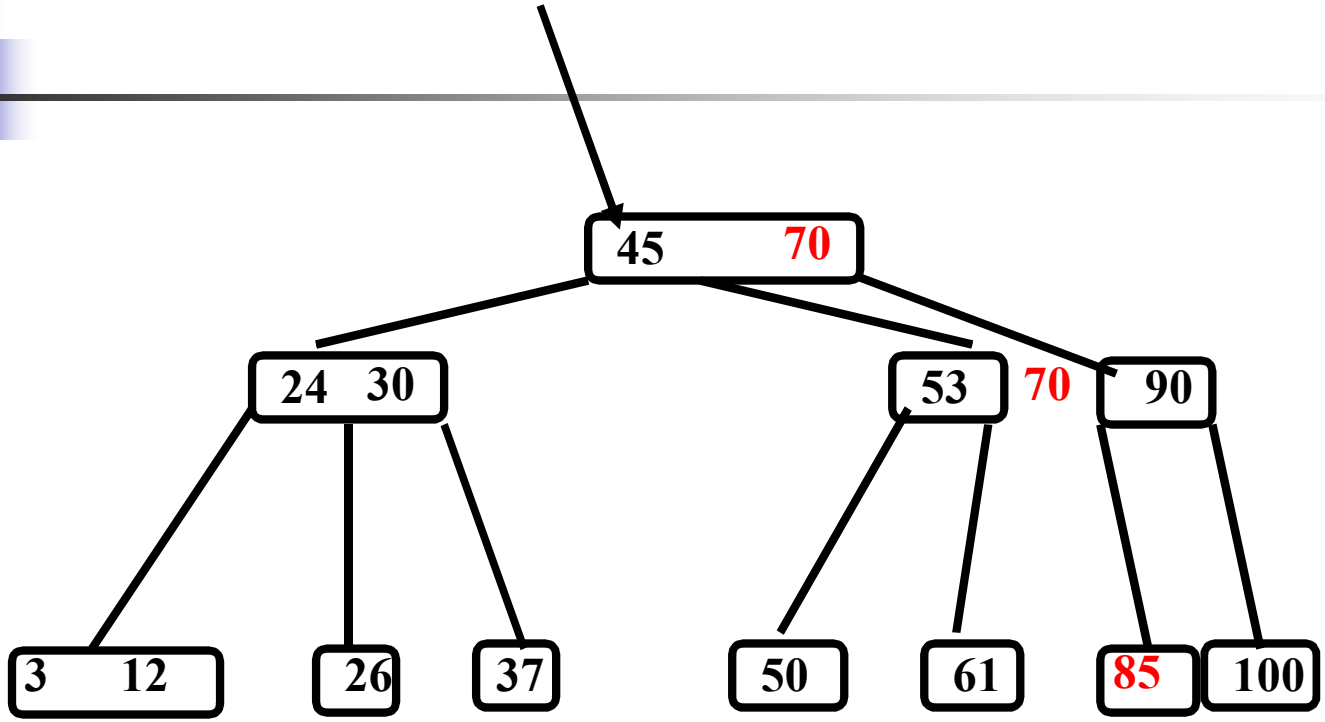
插入85

一棵3阶的B-树



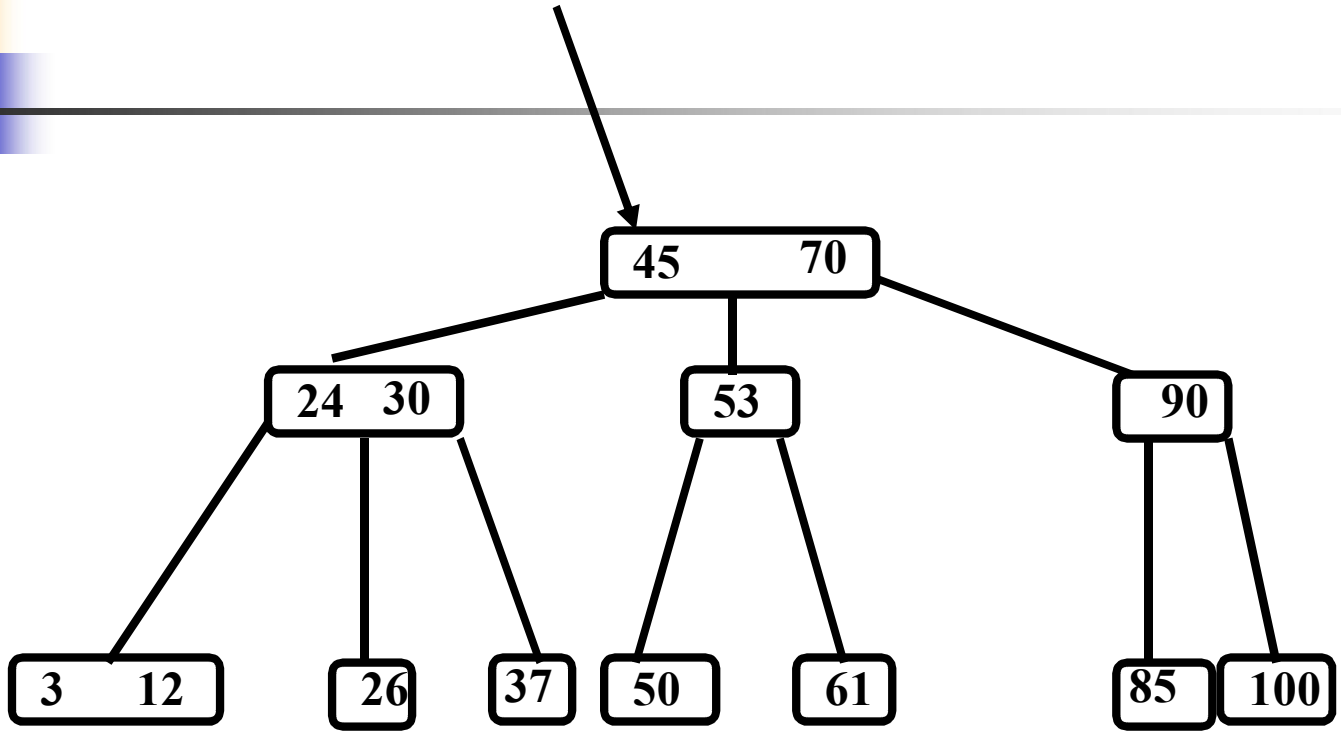
插入85

一棵3阶的B-树



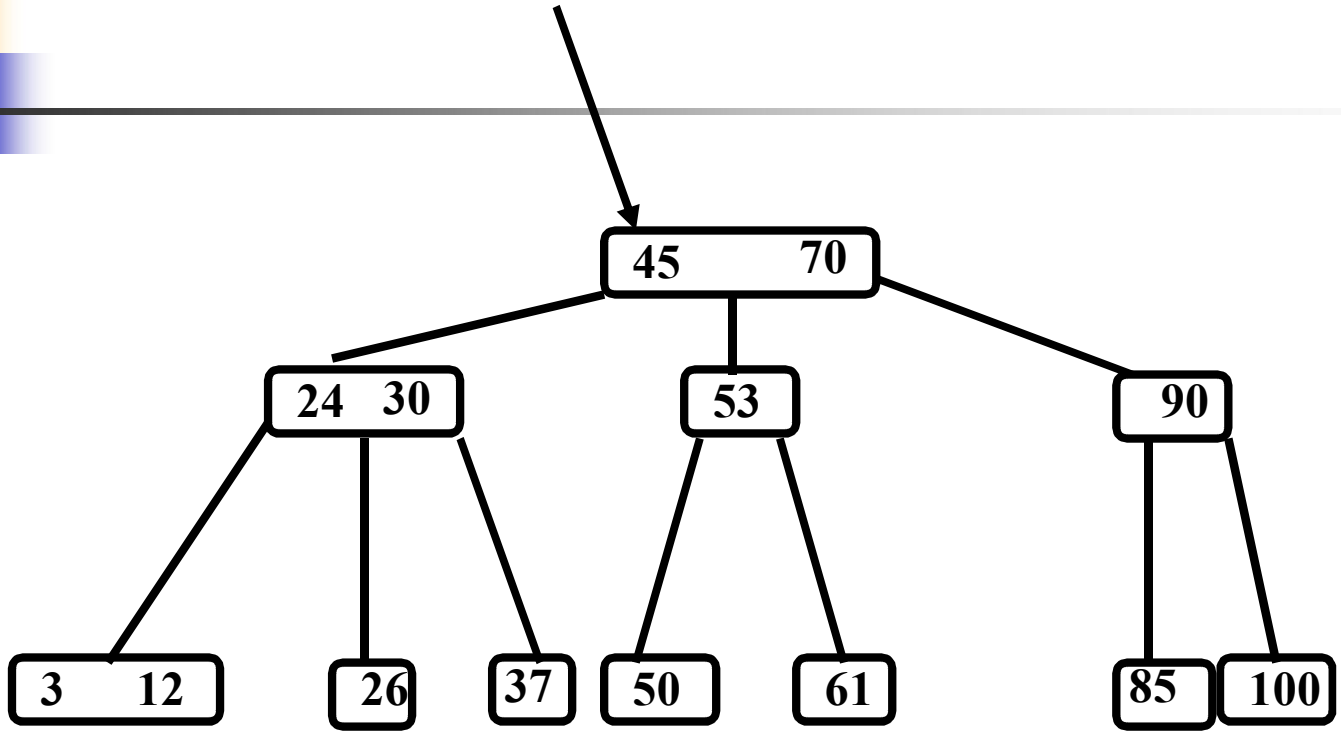
插入85

一棵3阶的B-树



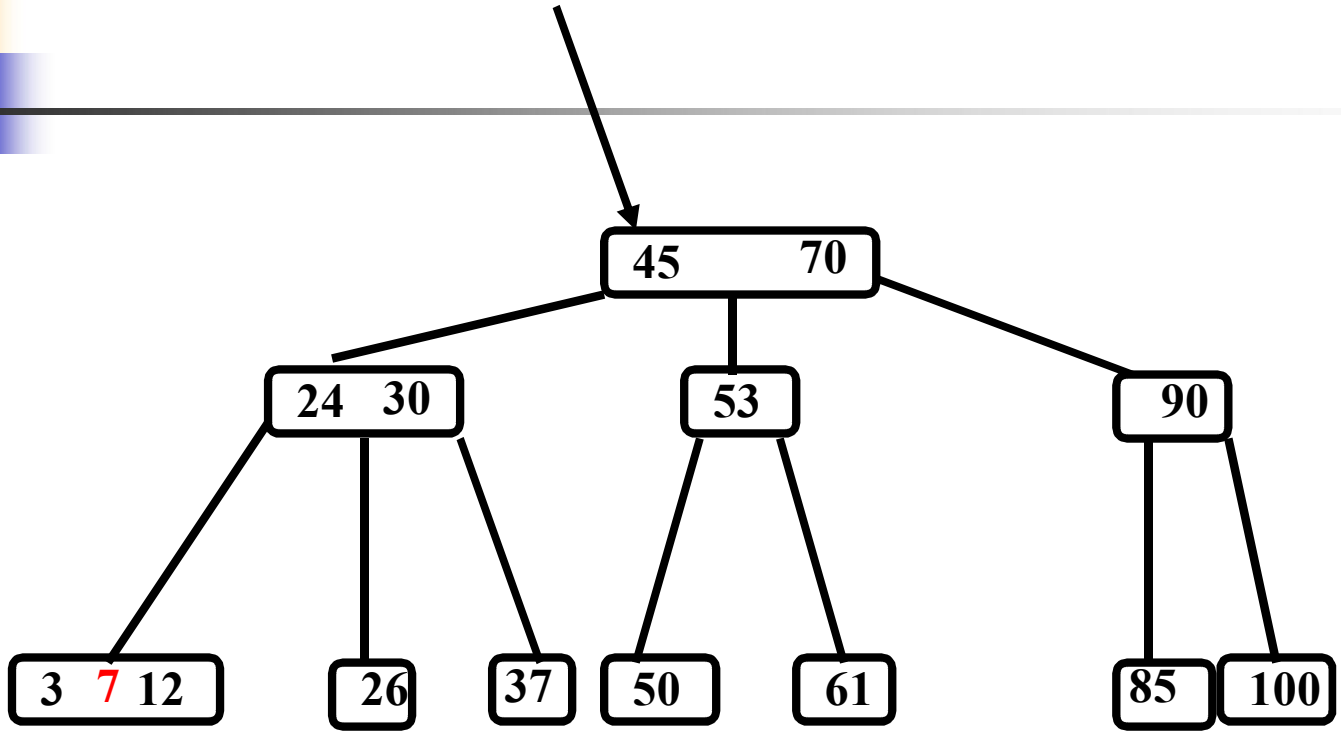
插入85

一棵3阶的B-树



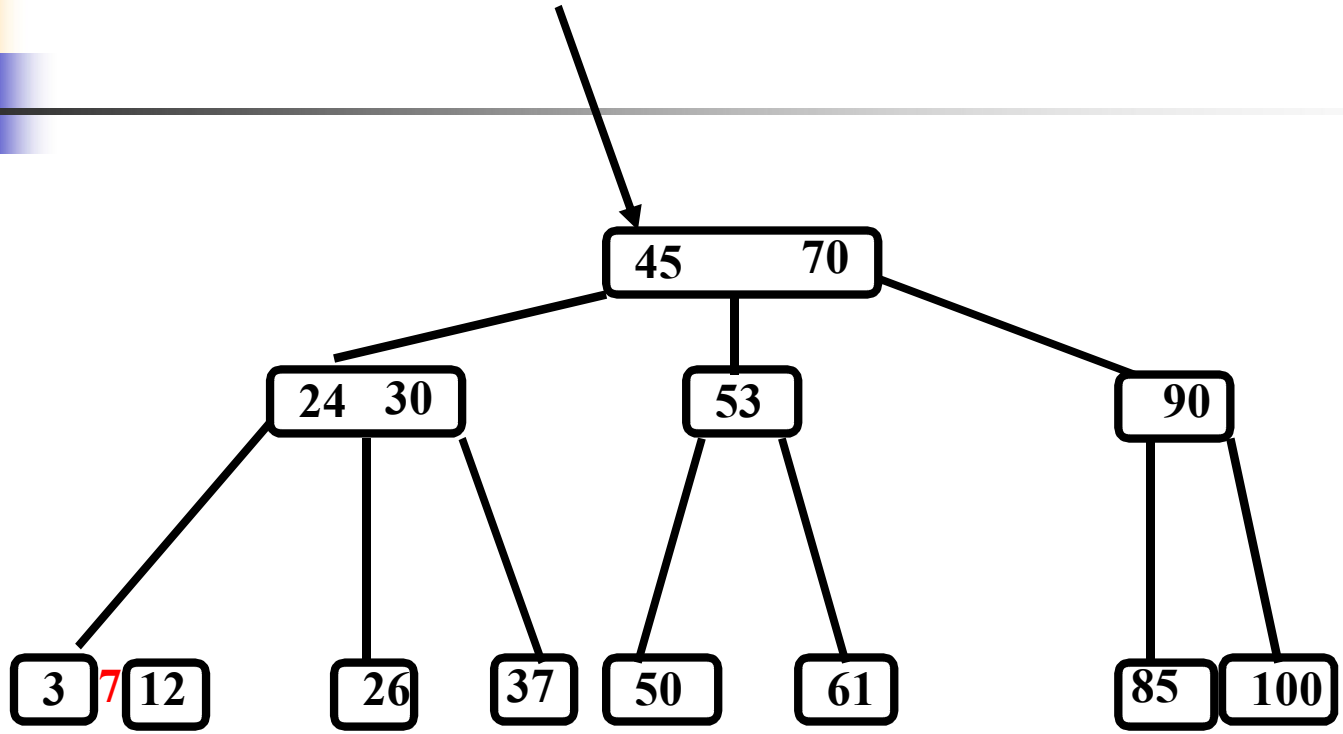
插入7

一棵3阶的B-树



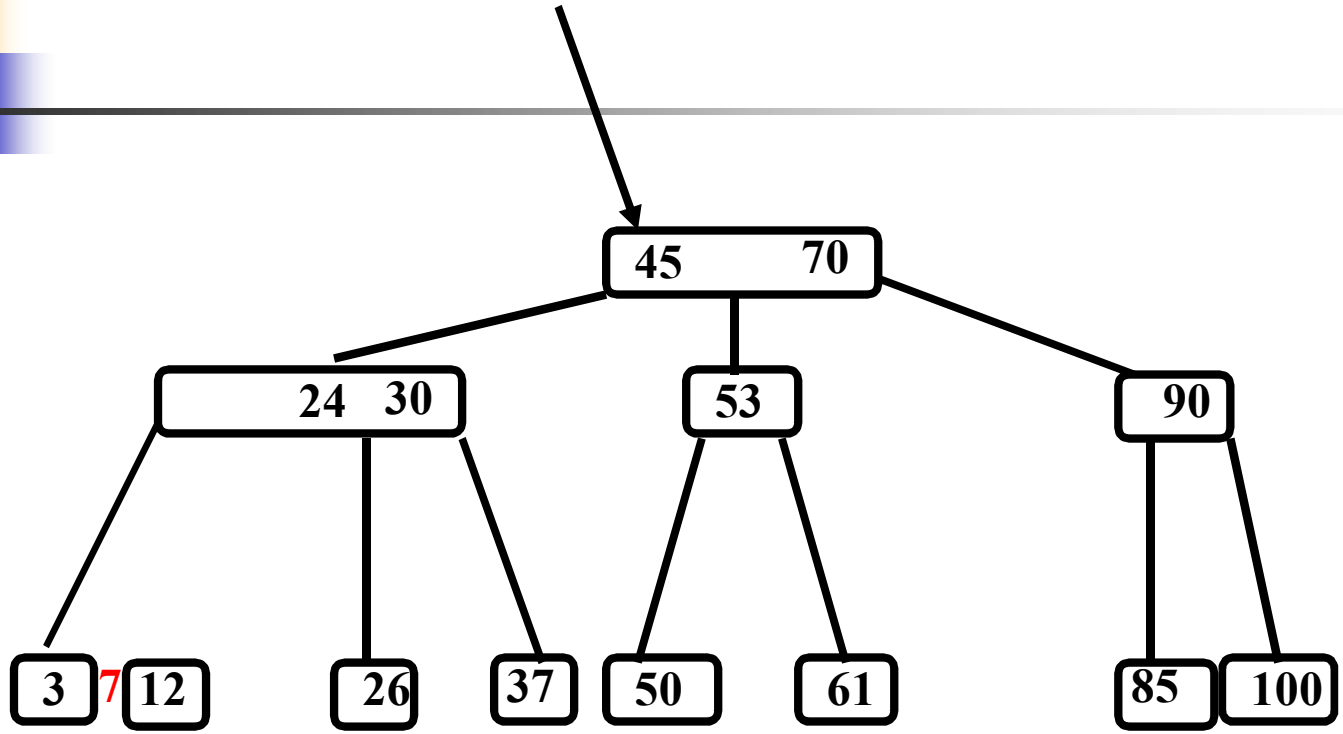
插入7

一棵3阶的B-树



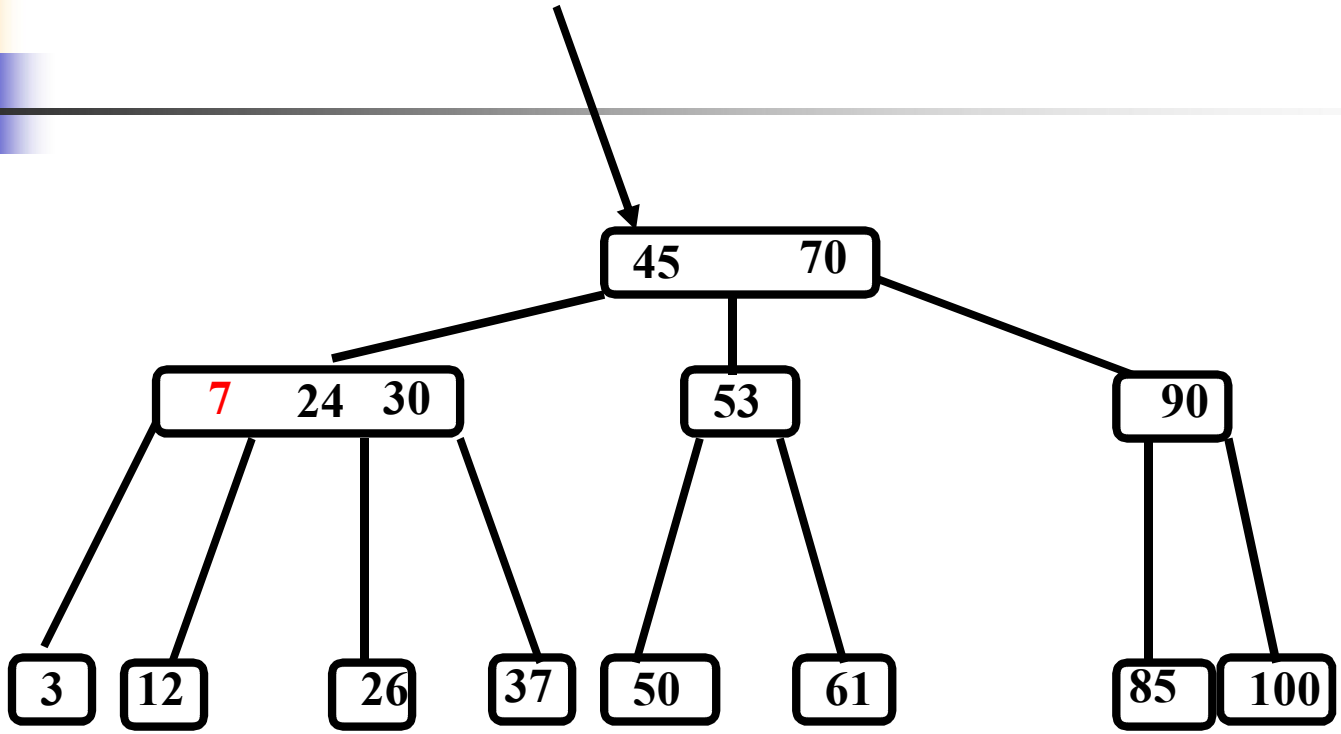
插入7

一棵3阶的B-树



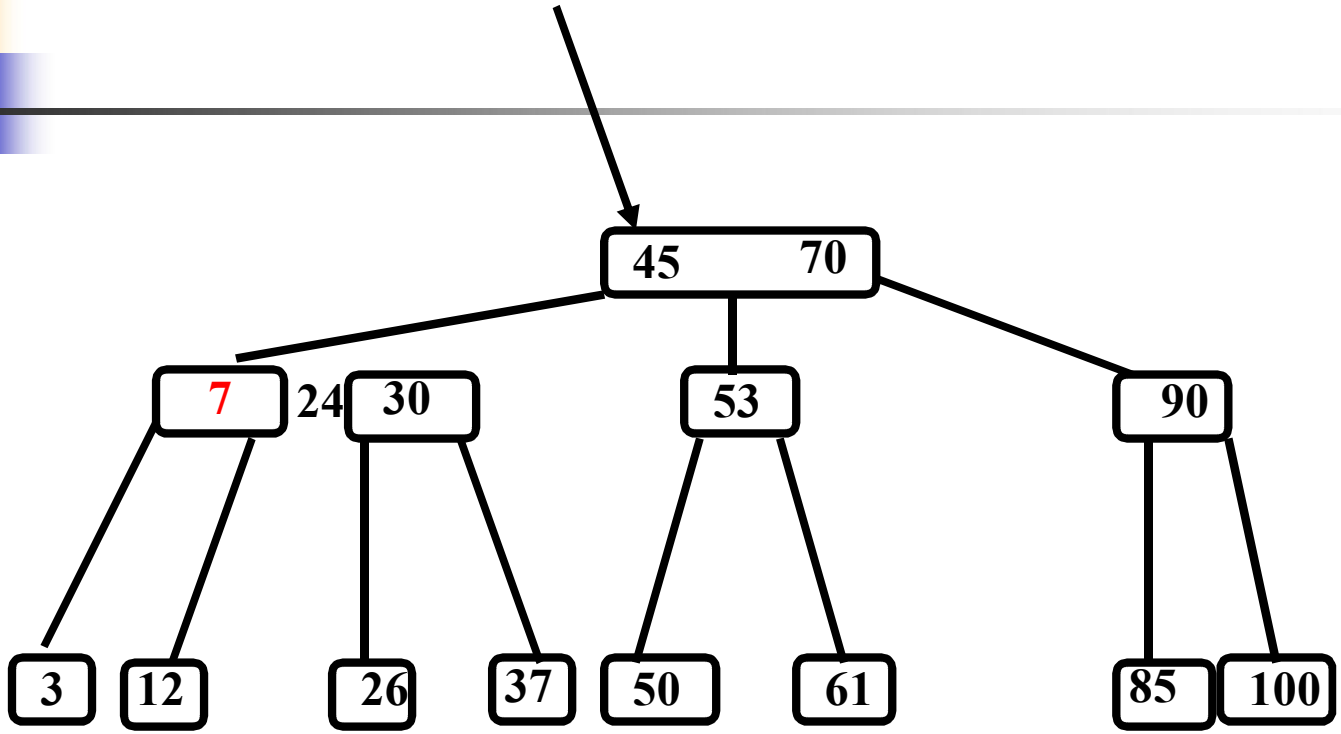
插入7

一棵3阶的B-树



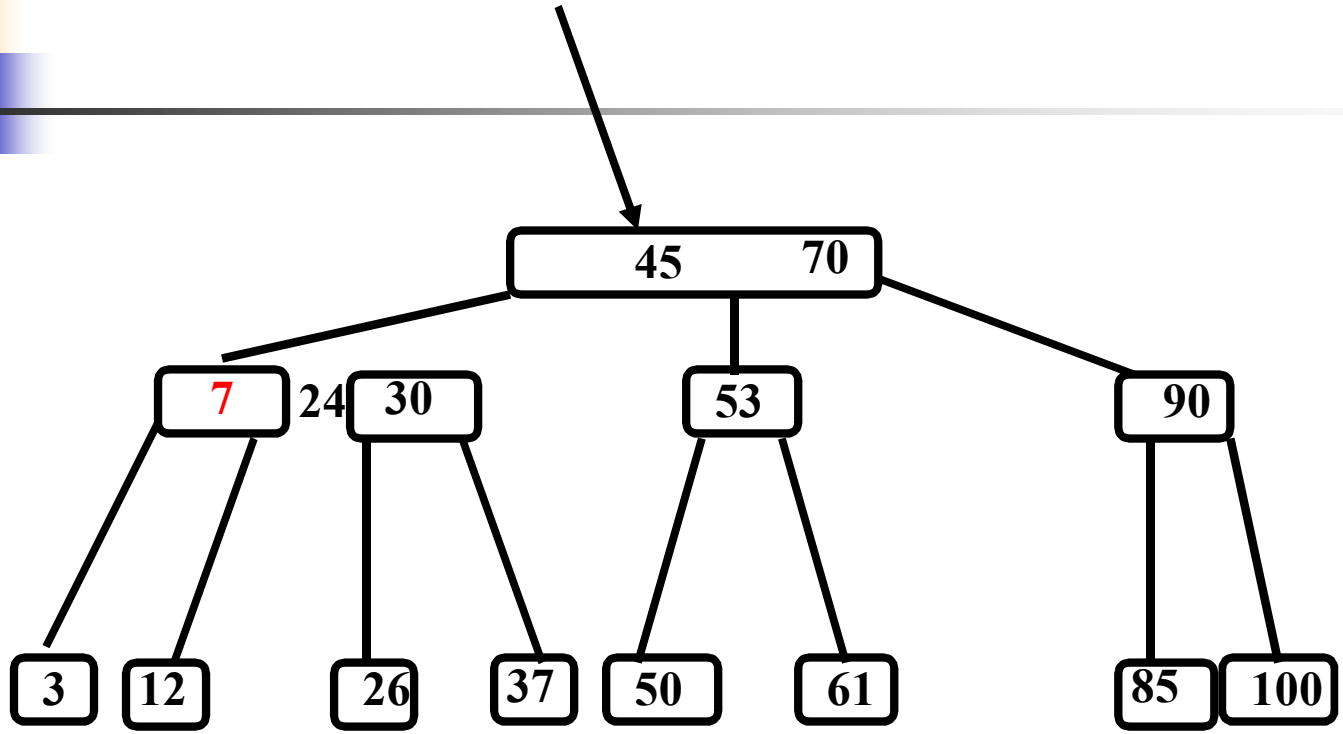
插入7

一棵3阶的B-树



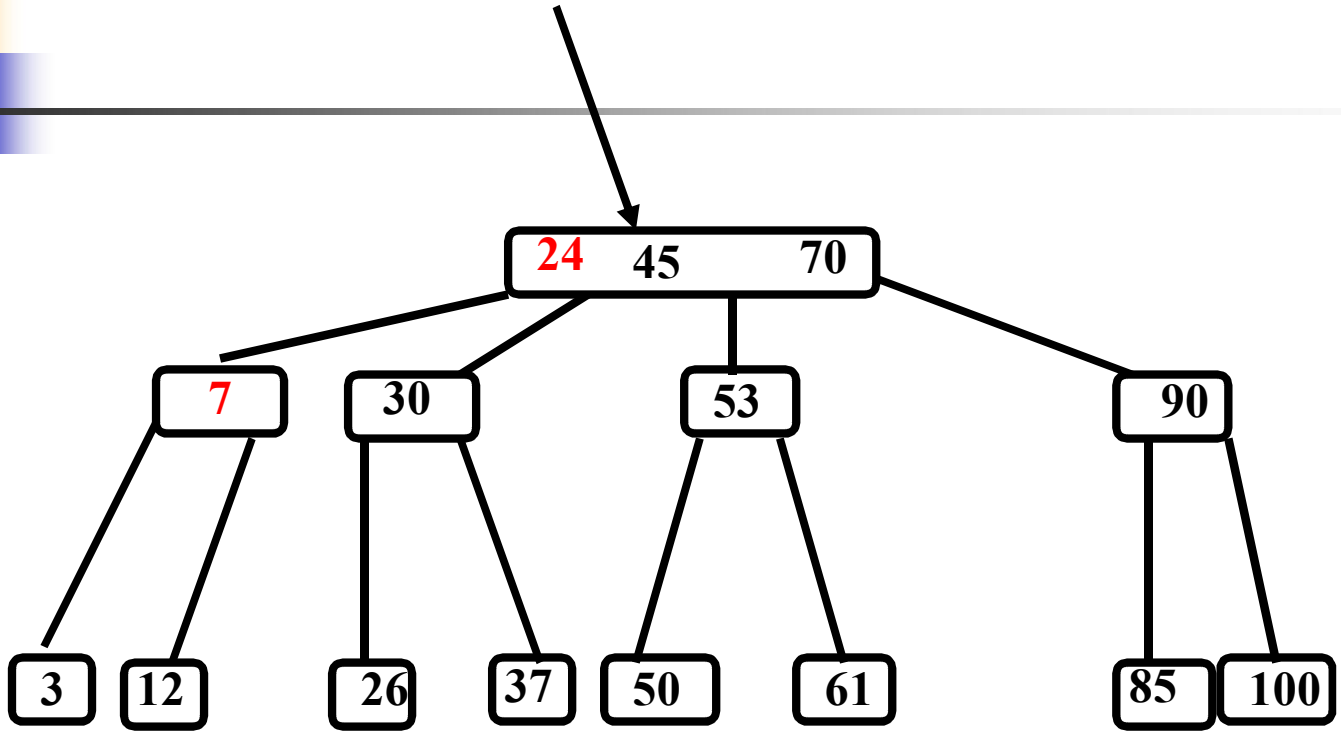
插入7

一棵3阶的B-树



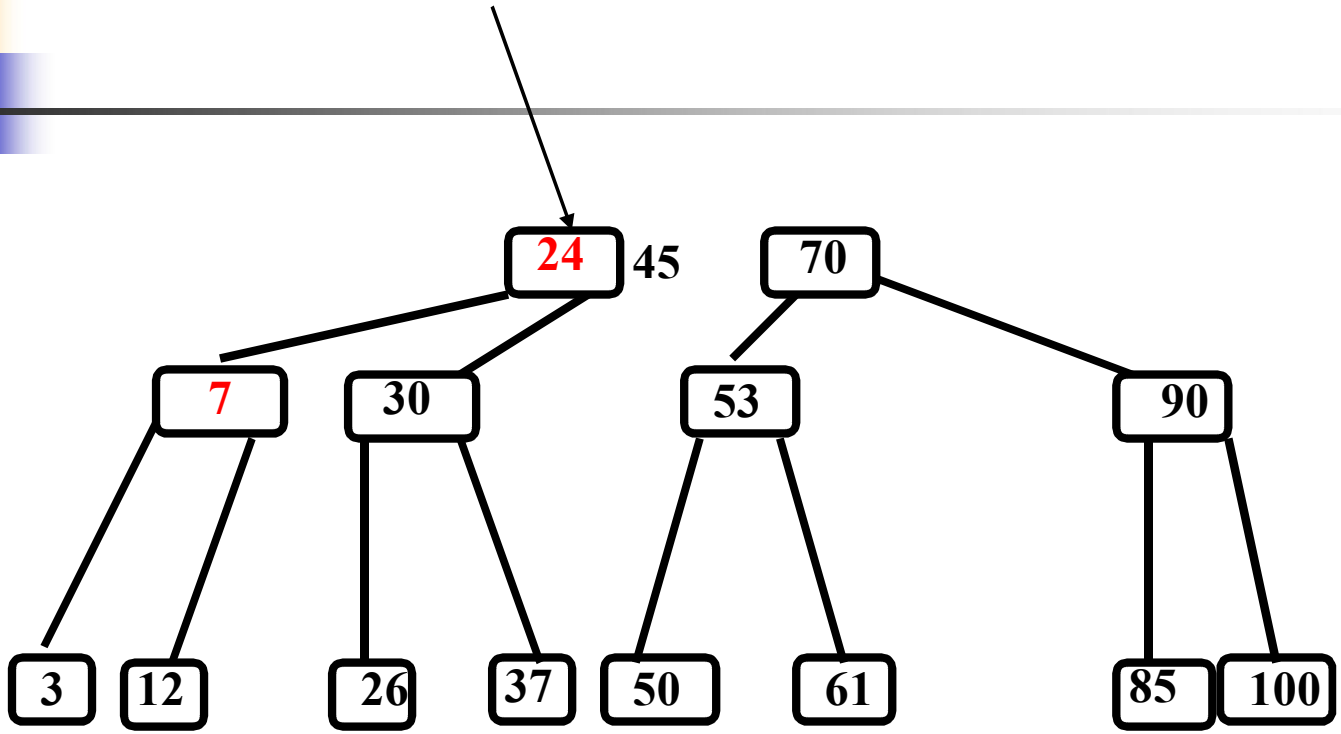
插入7

一棵3阶的B-树



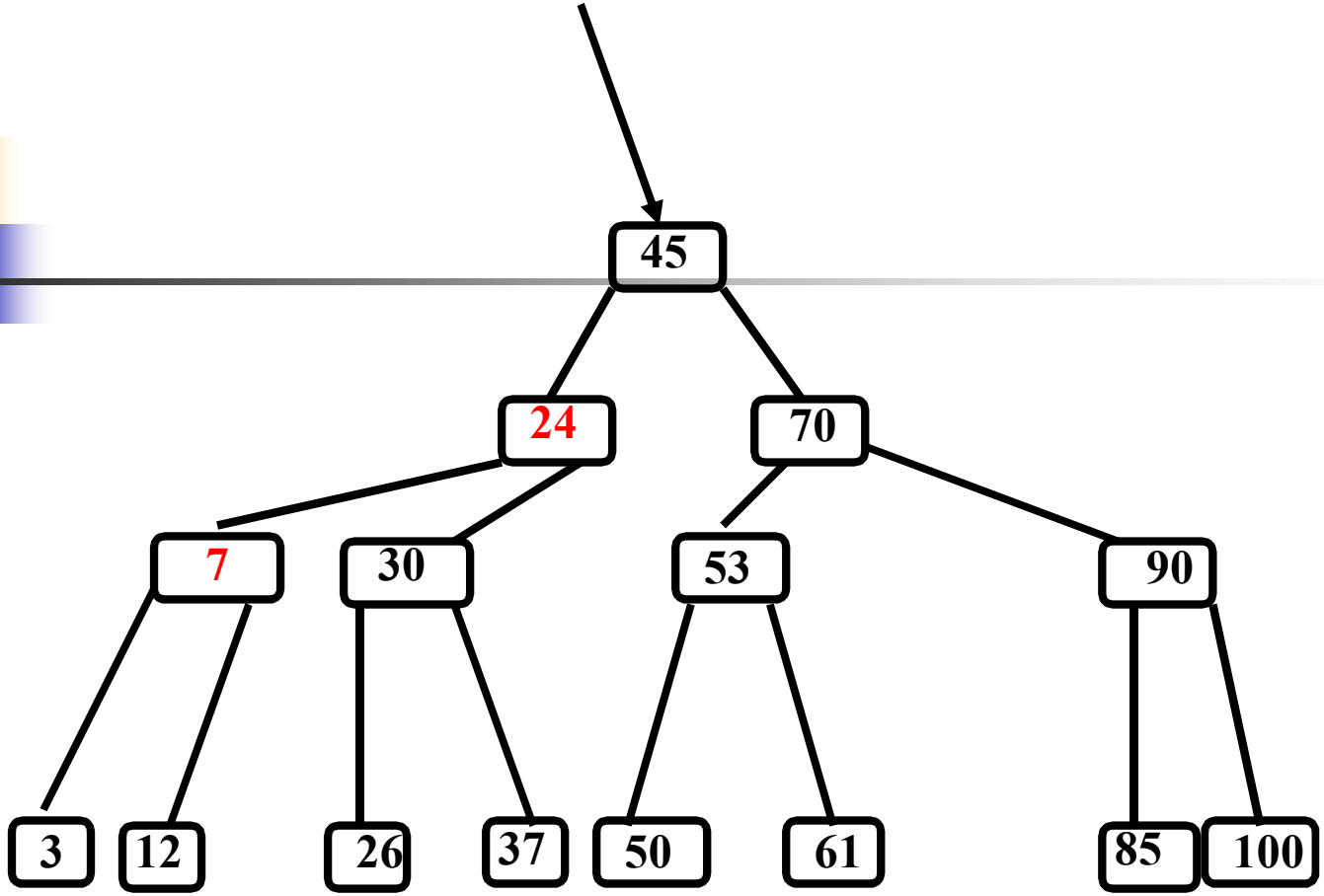
插入7

一棵3阶的B-树



插入7

一棵3阶的B-树



插入7

一棵3阶的B-树

B-树的删除

- 删除操作的两个步骤：
  - ✓ 第一步骤：在树中查找被删关键字K所在的位置
  - ✓ 第二步骤：进行删去K的操作

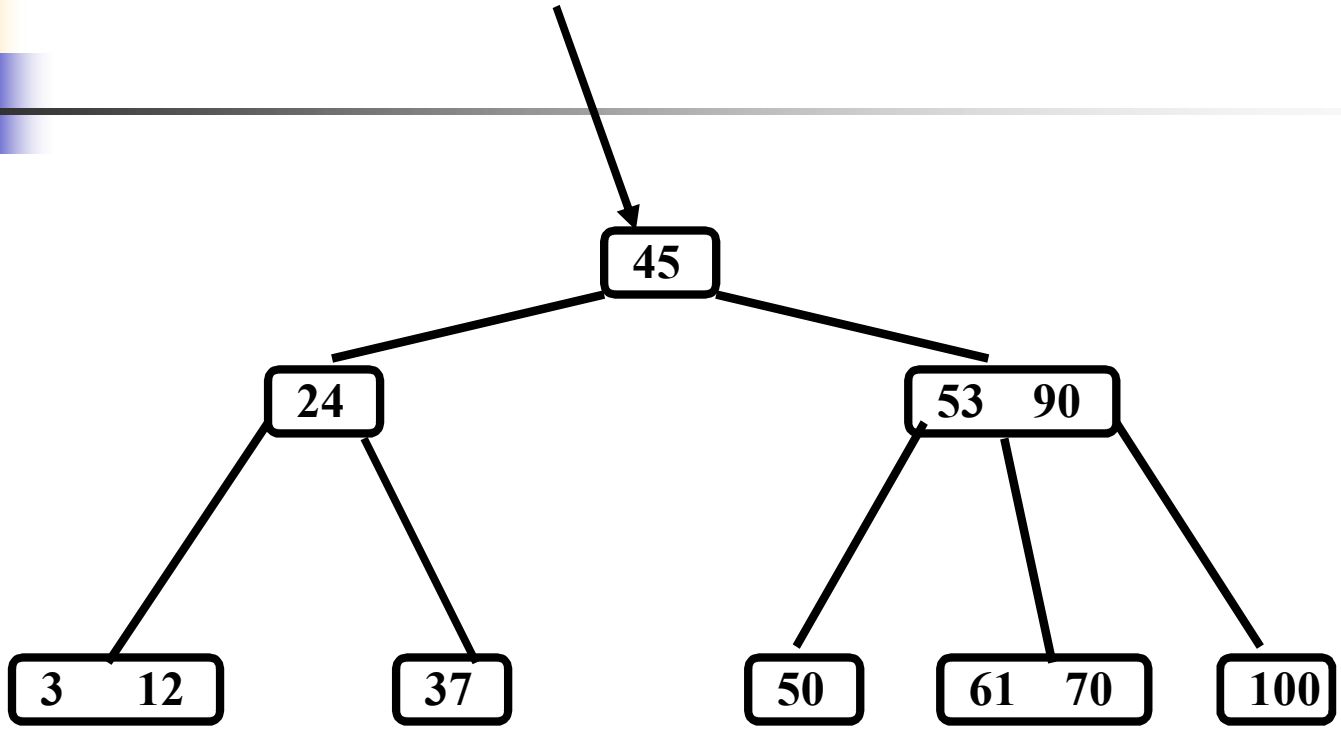
B-树的删除

- 根据关键字K在B-树中的位置，删除操作有2种情况：
  - 删除最底层非叶结点中关键字

    若删除操作前，结点中关键字个数不小于$\lceil m/2 \rceil$，直接删去；否则合并调整。
  - 所删关键字为非最底层非叶结点中的关键字K_i，则以指针A_i所指子树上的最小关键字Y代替K_i，然后删Y。
- 所有删除操作最终都是要：删除最底层非叶结点中关键字

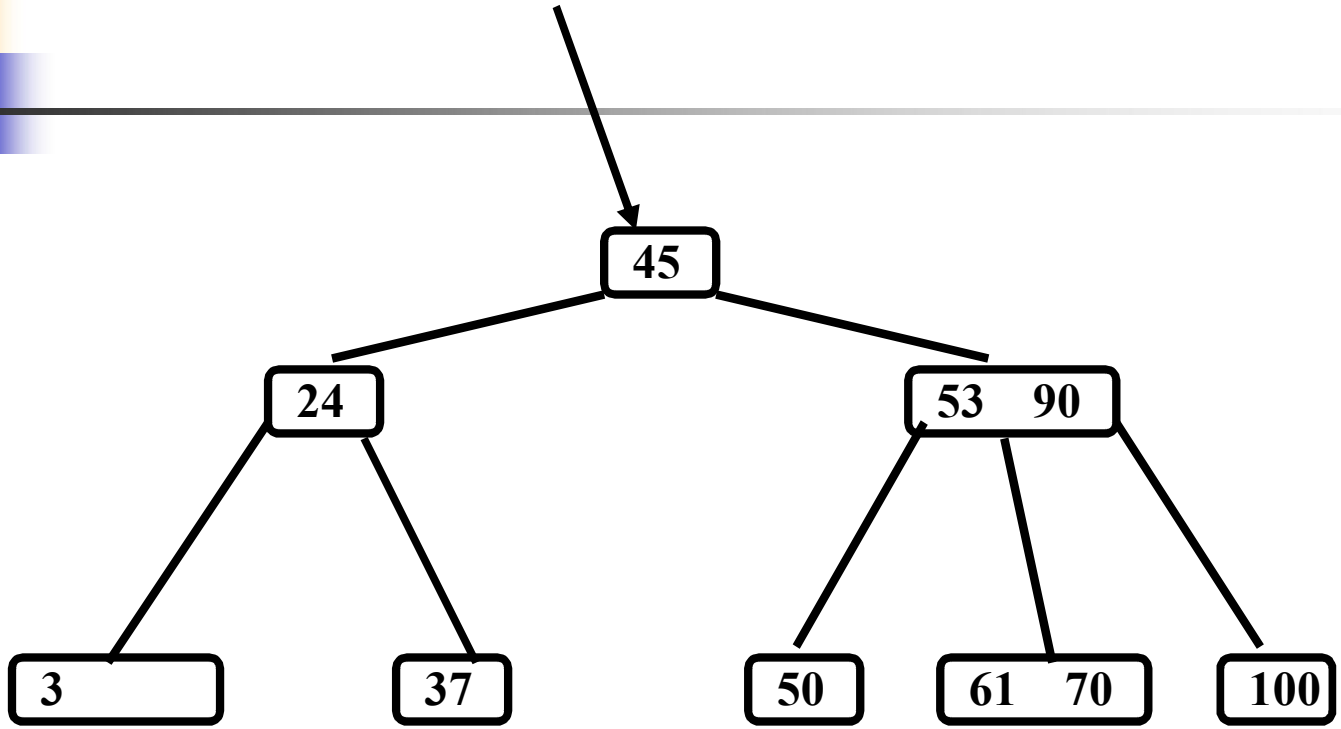
B-树的删除----合并调整操作

- 若被删关键字K所在结点中关键字个数等于$\lceil m/2 \rceil - 1$，删除后结点中关键字个数等于$\lceil m/2 \rceil - 2$，不符合m阶B-树的定义：
  - “与兄弟借”：而与该结点相邻右（或左）兄弟结点中的关键字个数大于$\lceil m/2 \rceil - 1$，则将其兄弟结点中的最小（或最大）关键字上移至双亲结点中，而将双亲结点中小于（或大于）且紧靠上移关键字的关键字下移至被删关键字所在结点。
  - “兄弟没有能力借给它”：与该结点相邻的（左右）兄弟结点中关键字个数均等于$\lceil m/2 \rceil - 1$，设其有右兄弟，且右兄弟的地址是双亲中的A_i，则删除关键字后，所在结点剩余的关键字和指针加上双亲中的K_i一起合并到A_i所指的结点。若无右兄弟，有左兄弟，类似合并操作，左兄弟的地址是双亲中的A_{i-1}，则删除关键字后，所在结点剩余的关键字和指针加上双亲中的K_i一起合并到A_{i-1}所指的结点。合并后检查双亲结点是否满足定义，不满足则继续调整。



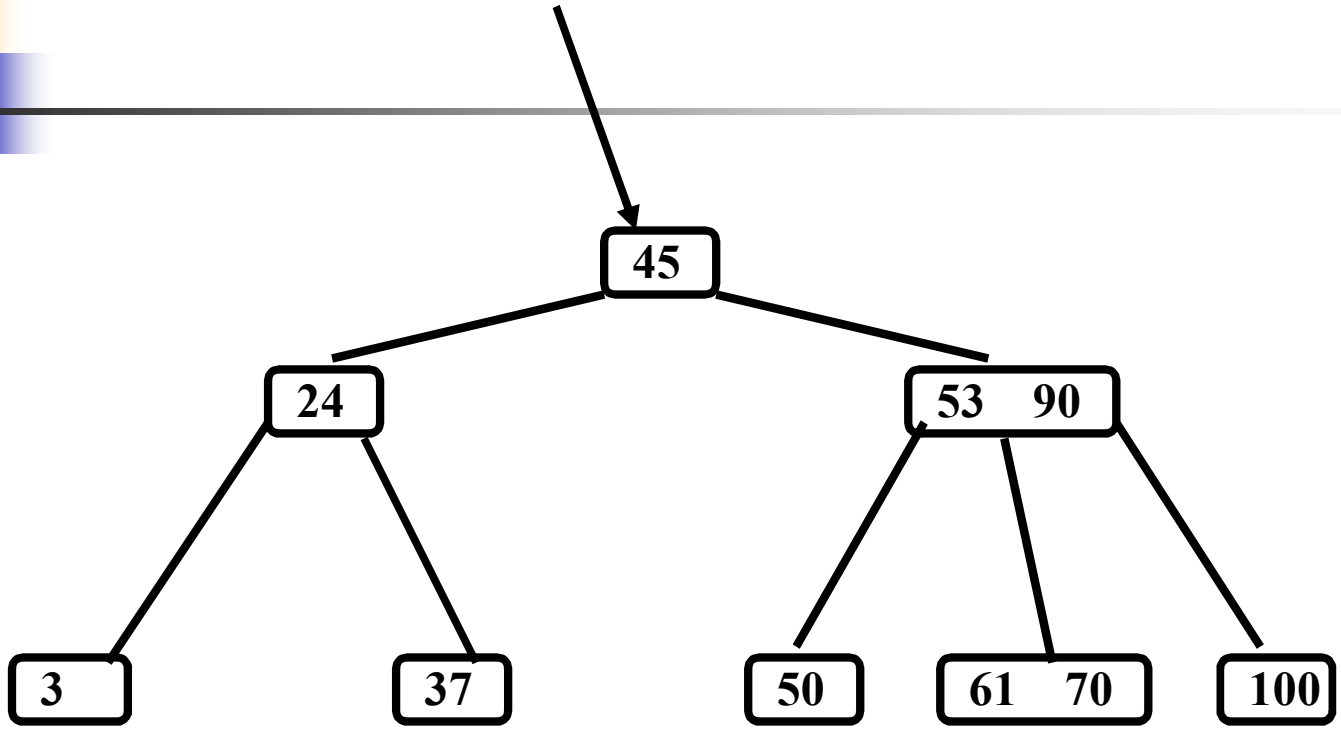
删除12

一棵3阶的B-树



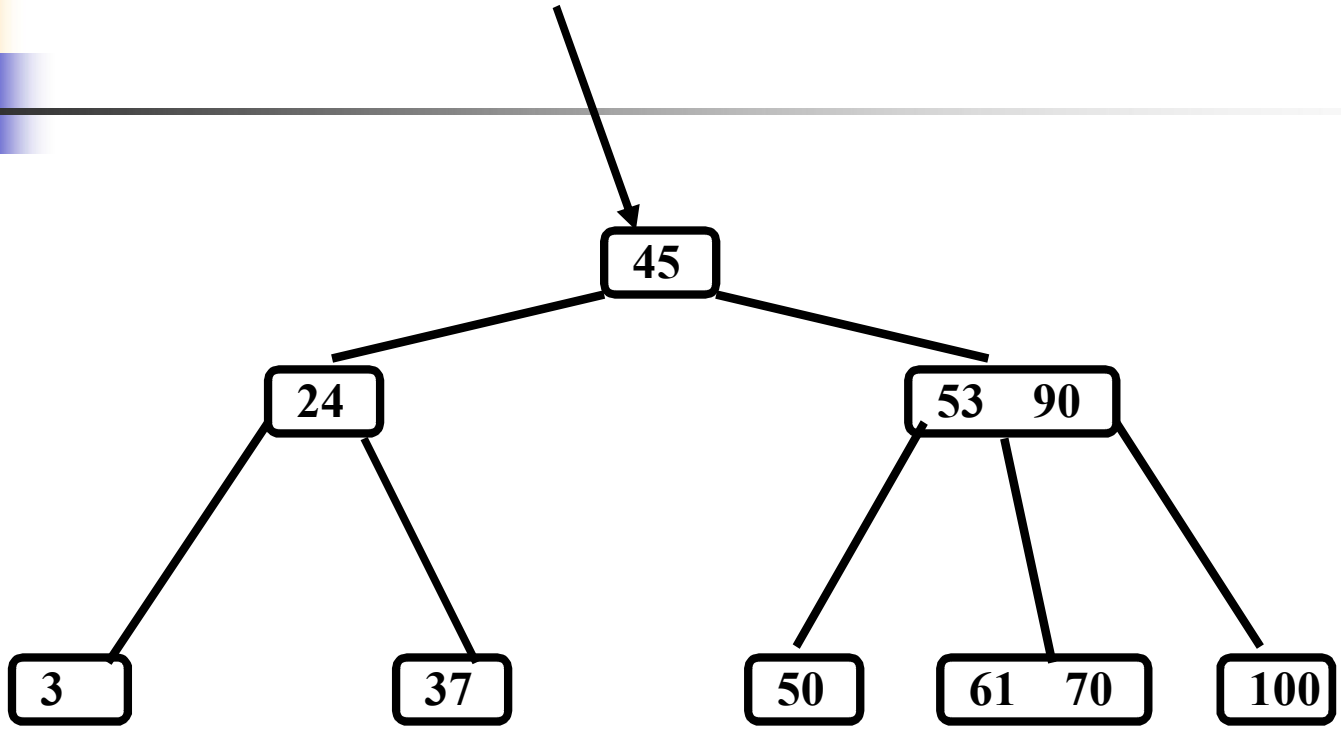
删除12

一棵3阶的B-树



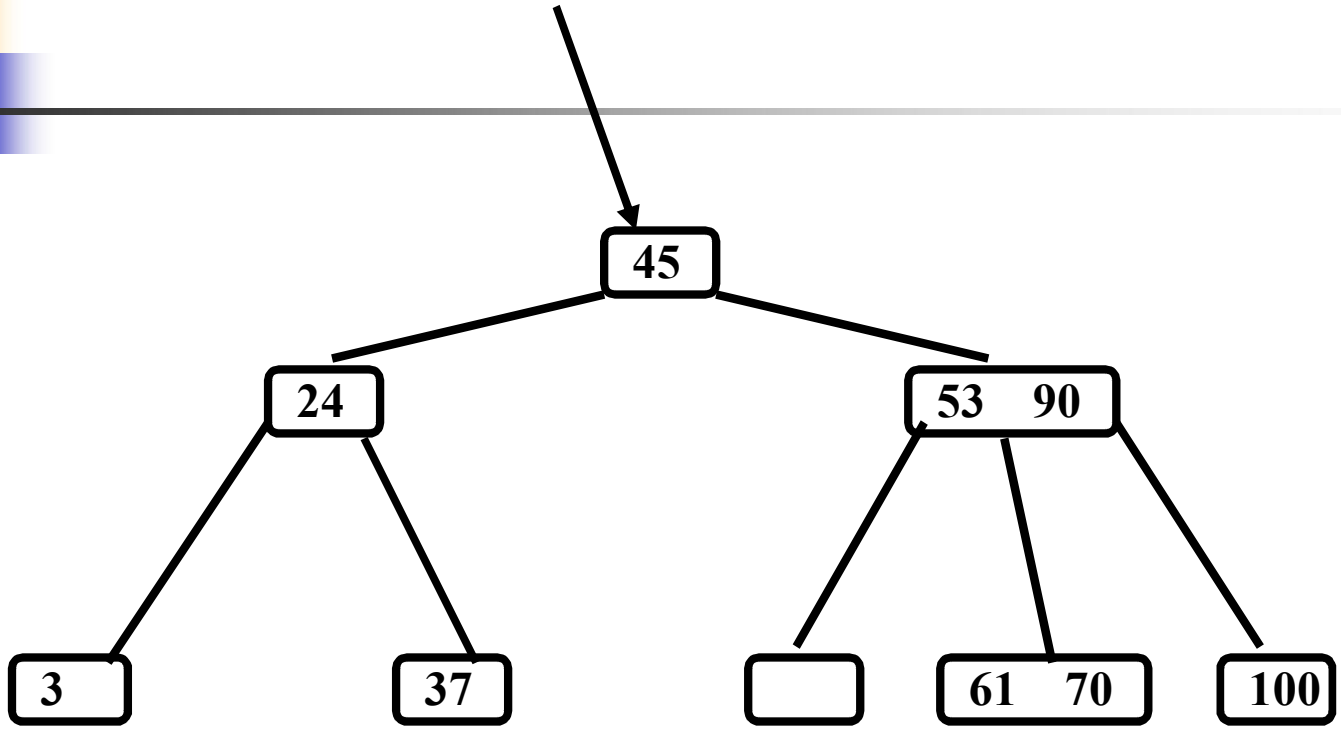
删除12

一棵3阶的B-树



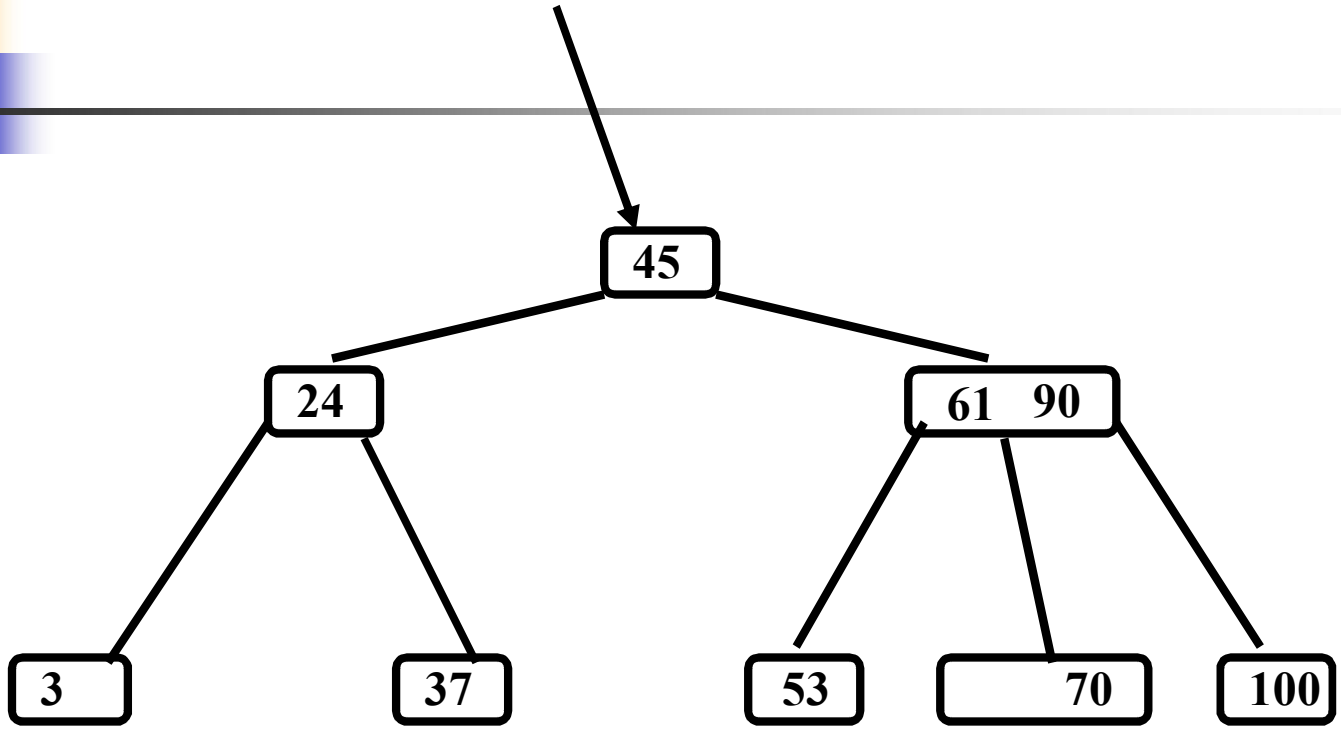
删除50

一棵3阶的B-树



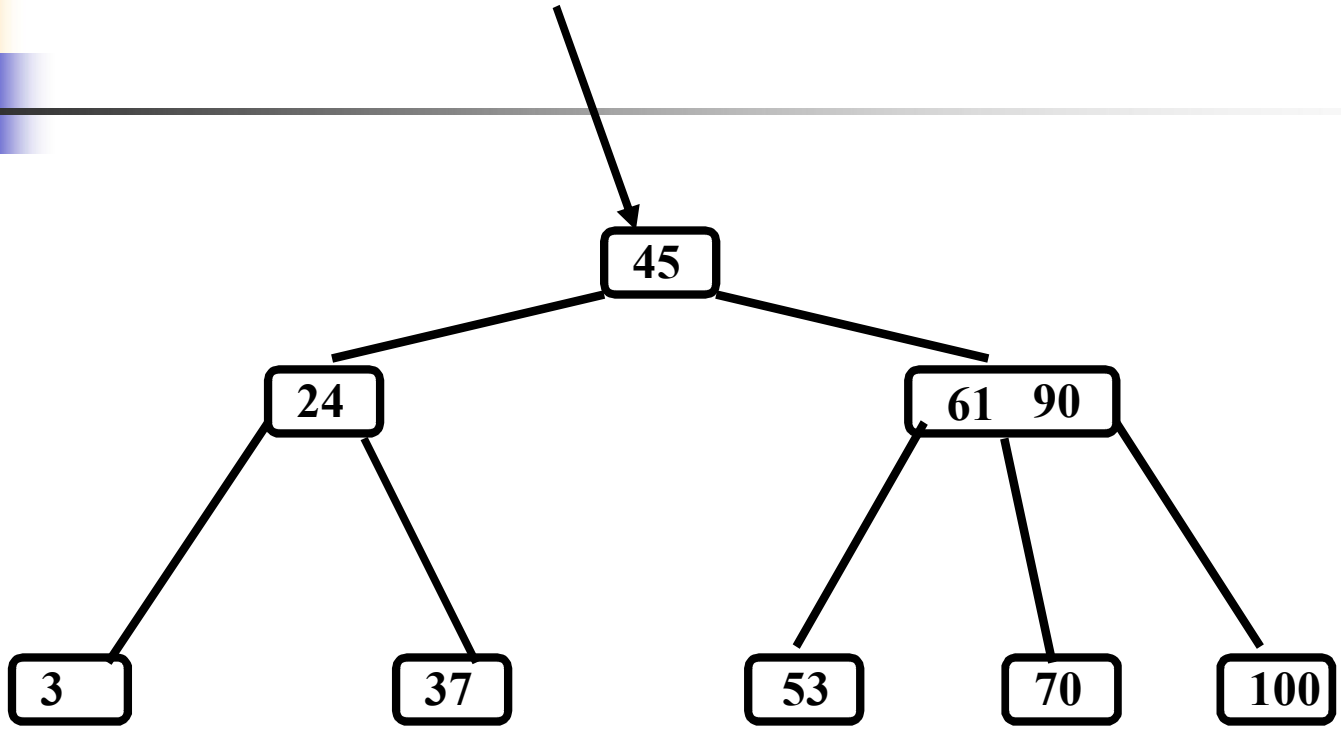
删除50

一棵3阶的B-树



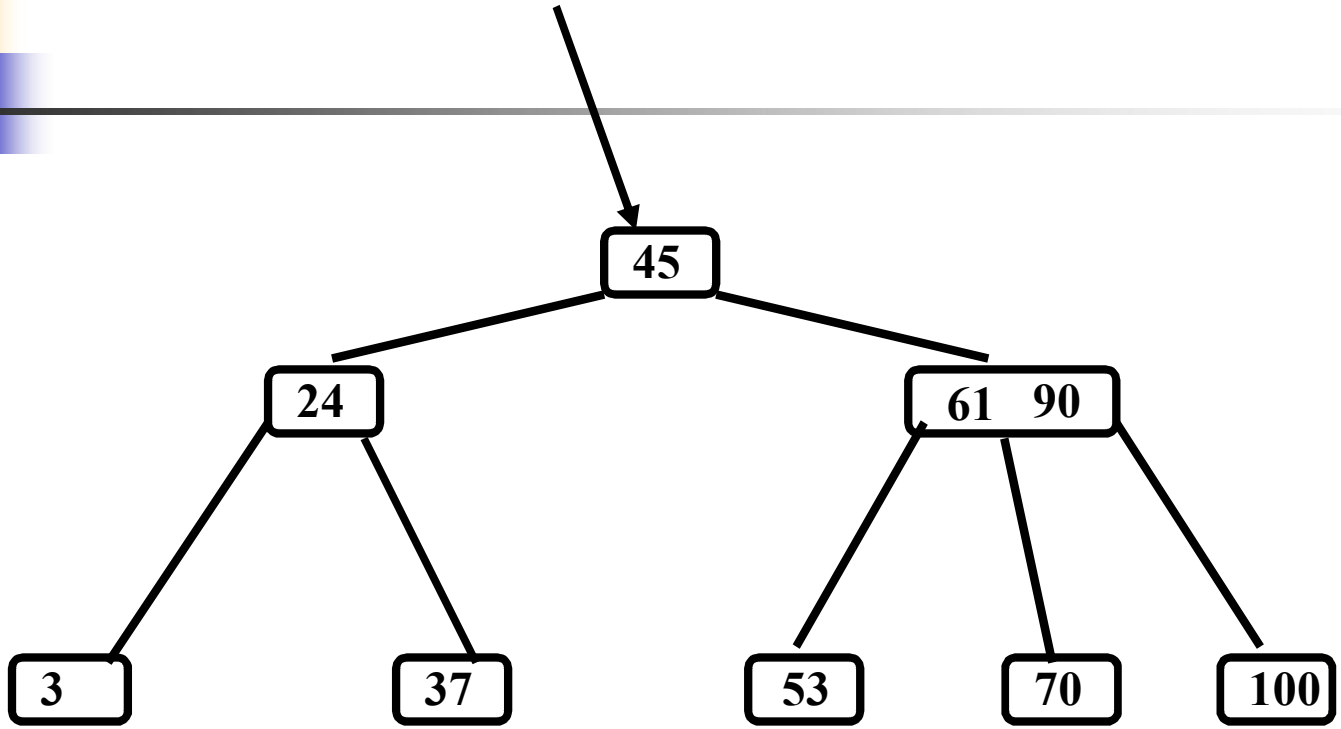
删除50

一棵3阶的B-树



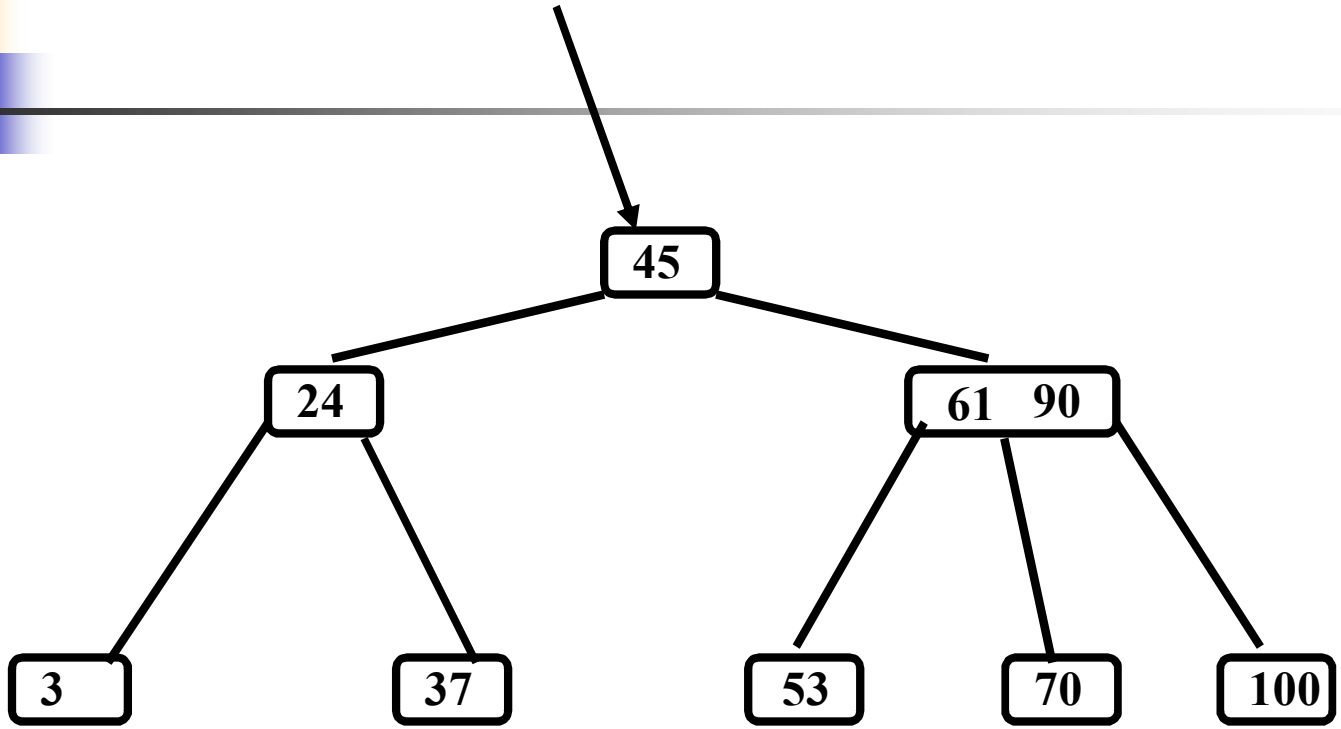
删除50

一棵3阶的B-树



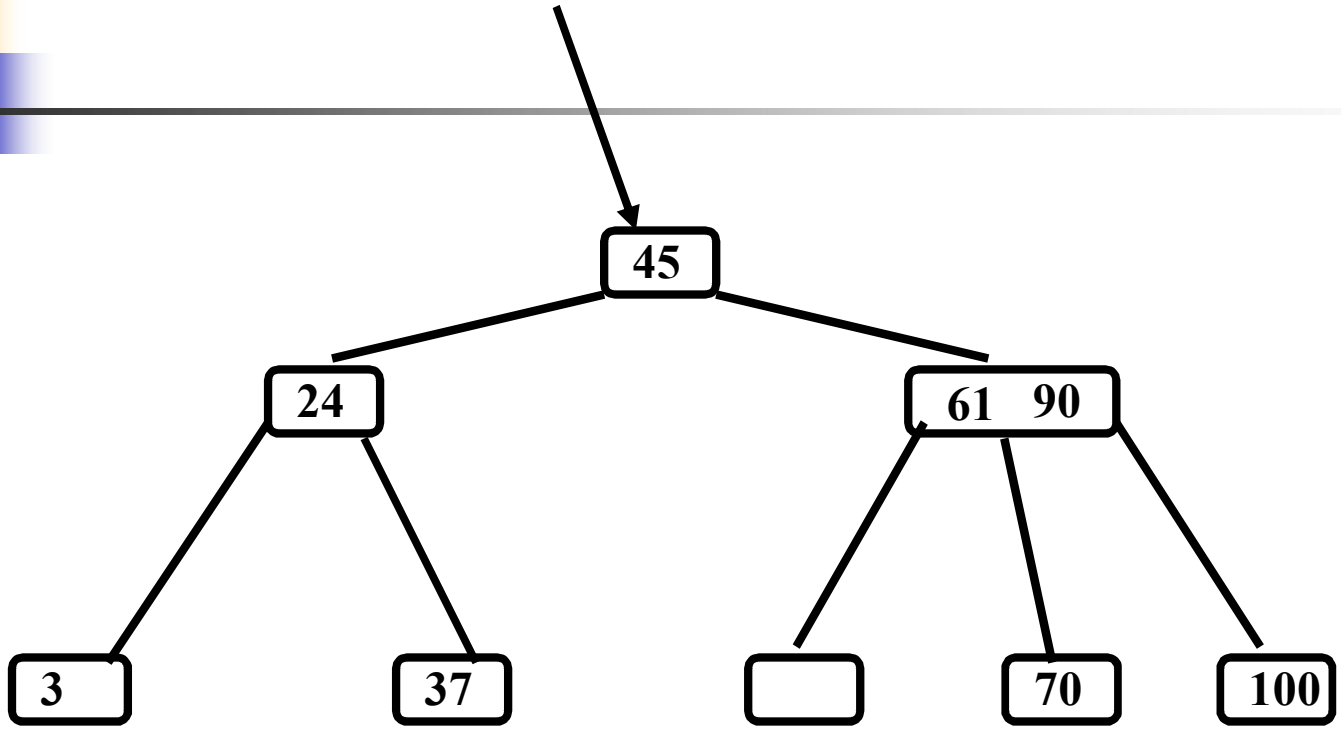
删除50

一棵3阶的B-树



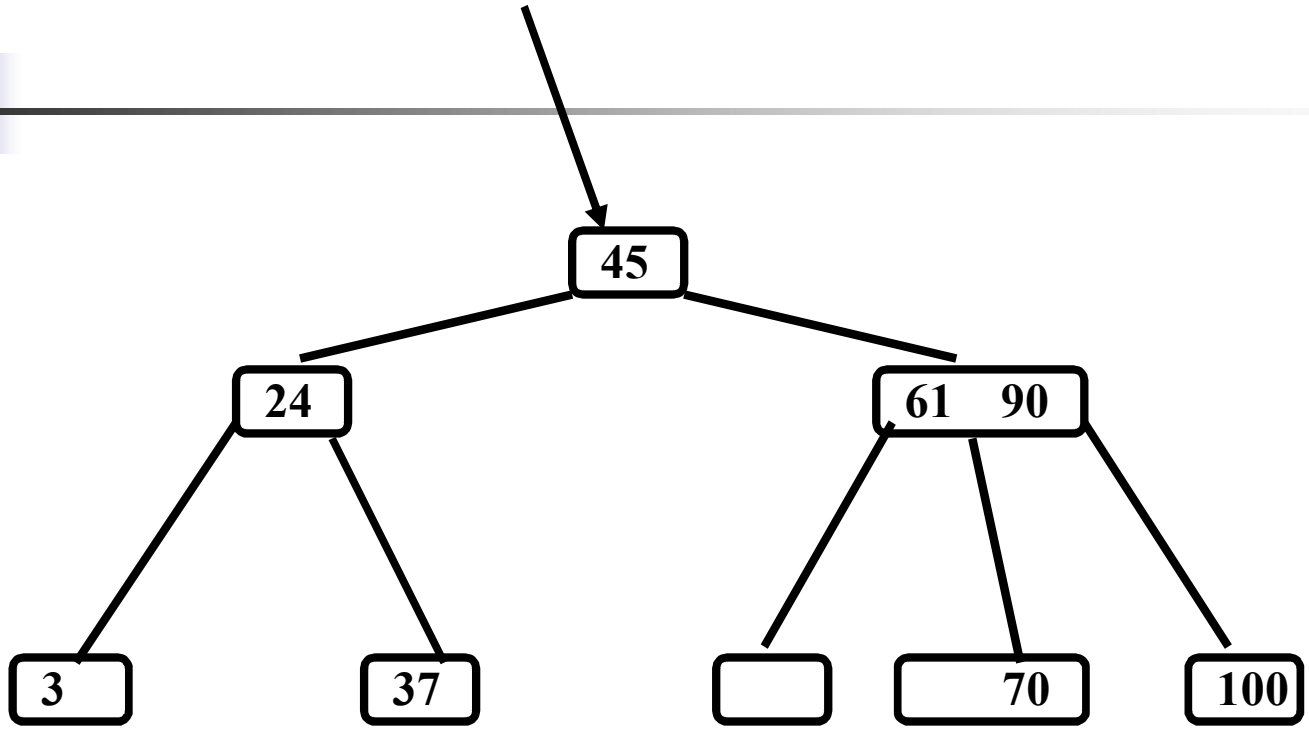
删除53

一棵3阶的B-树



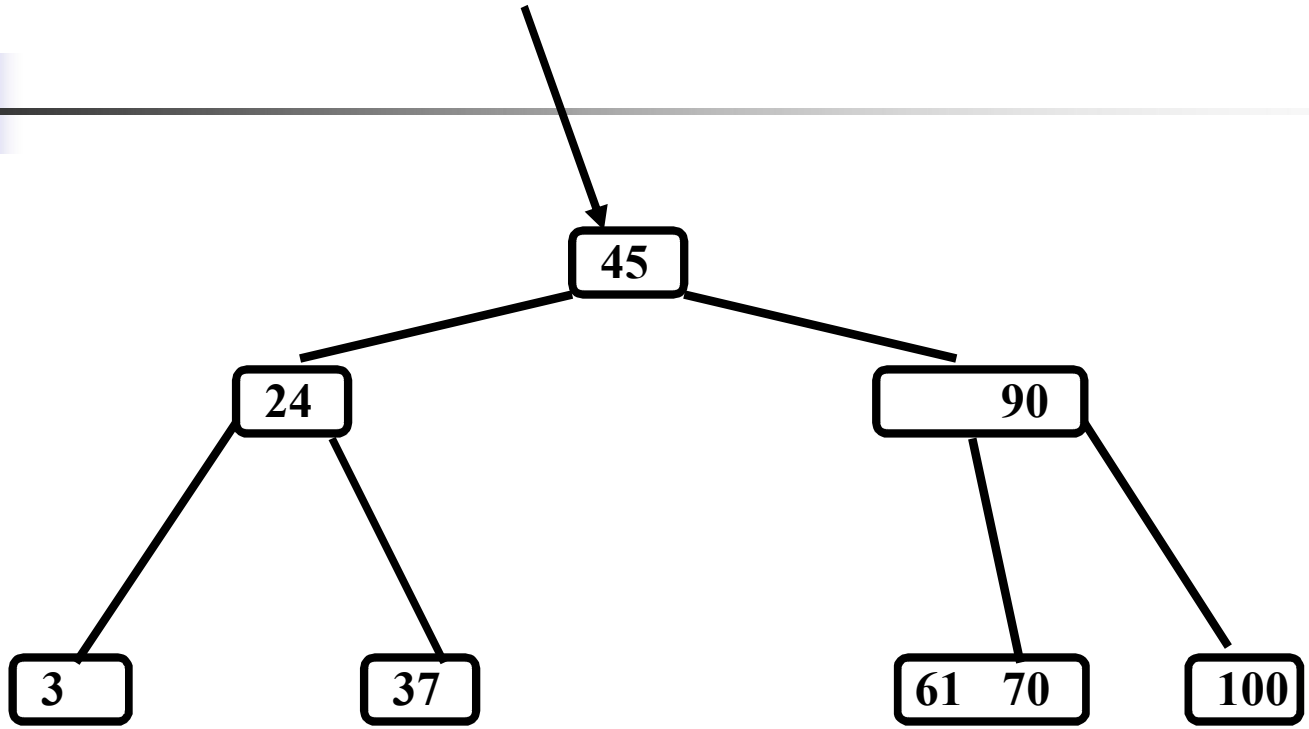
删除53

一棵3阶的B-树



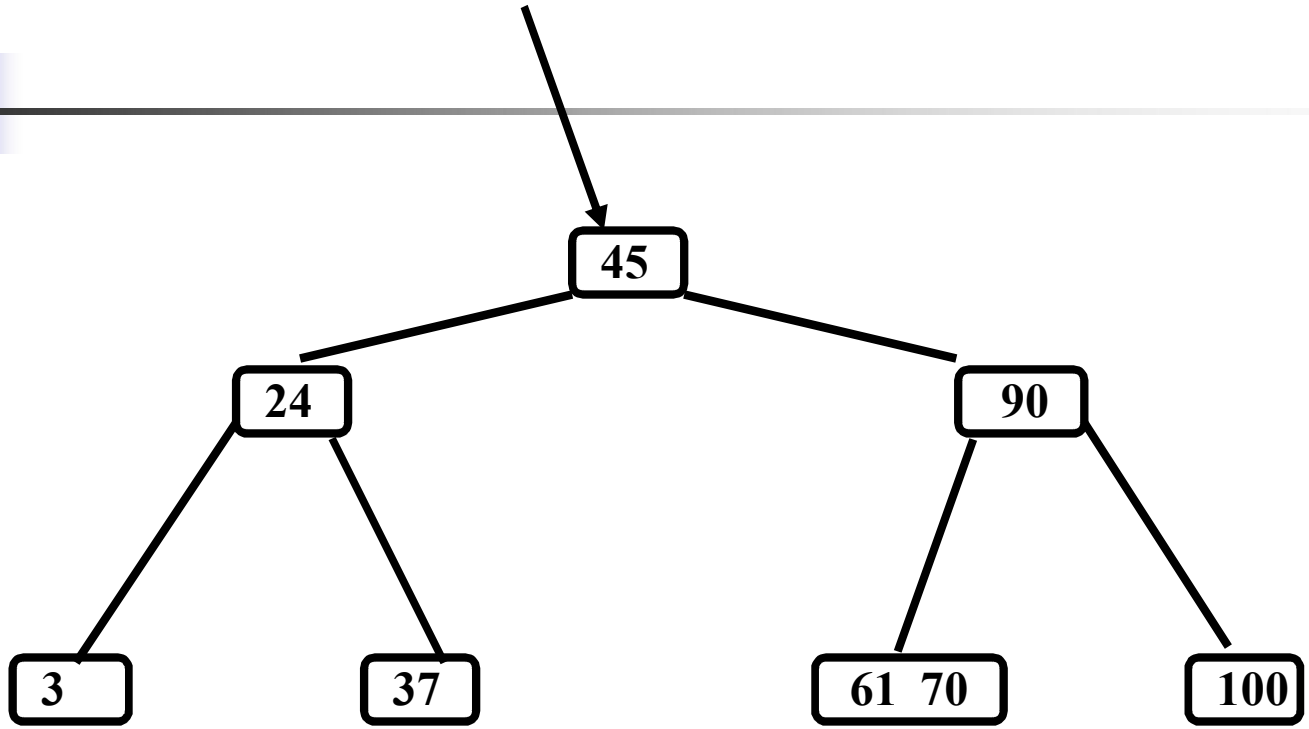
删除53

一棵3阶的B-树



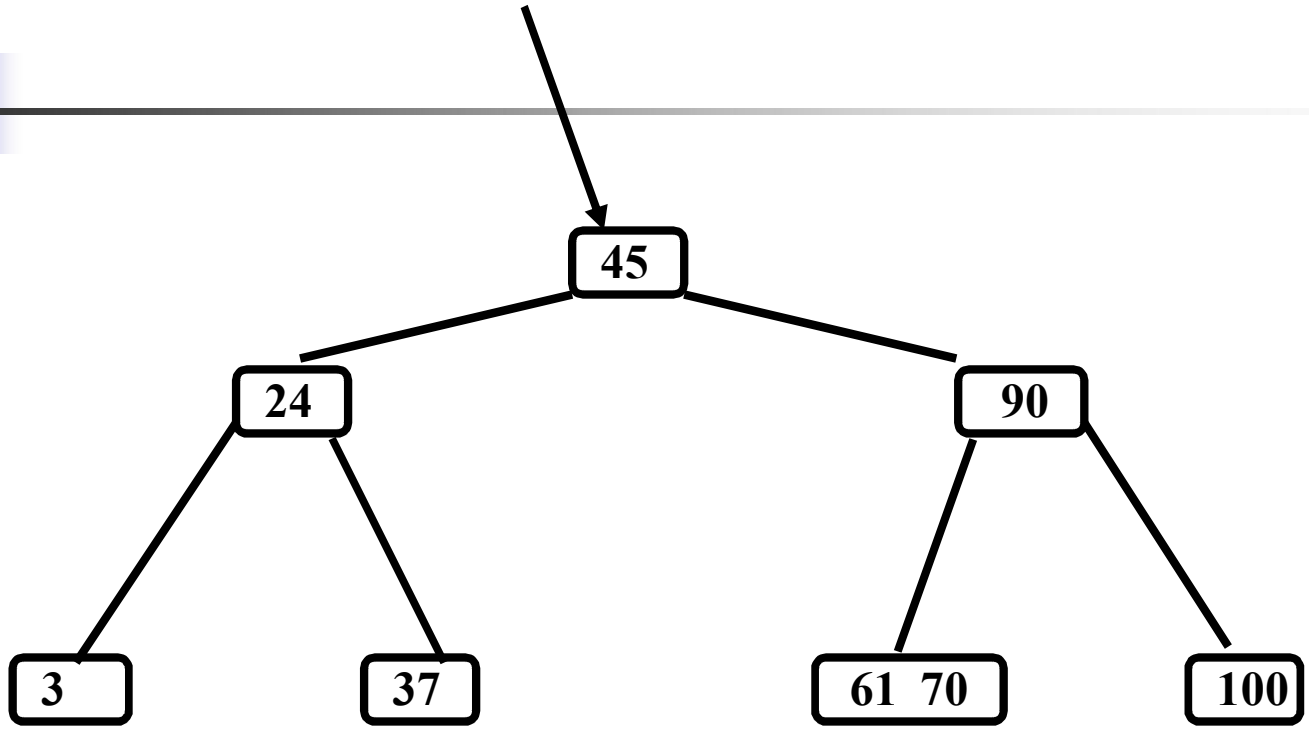
删除53

一棵3阶的B-树



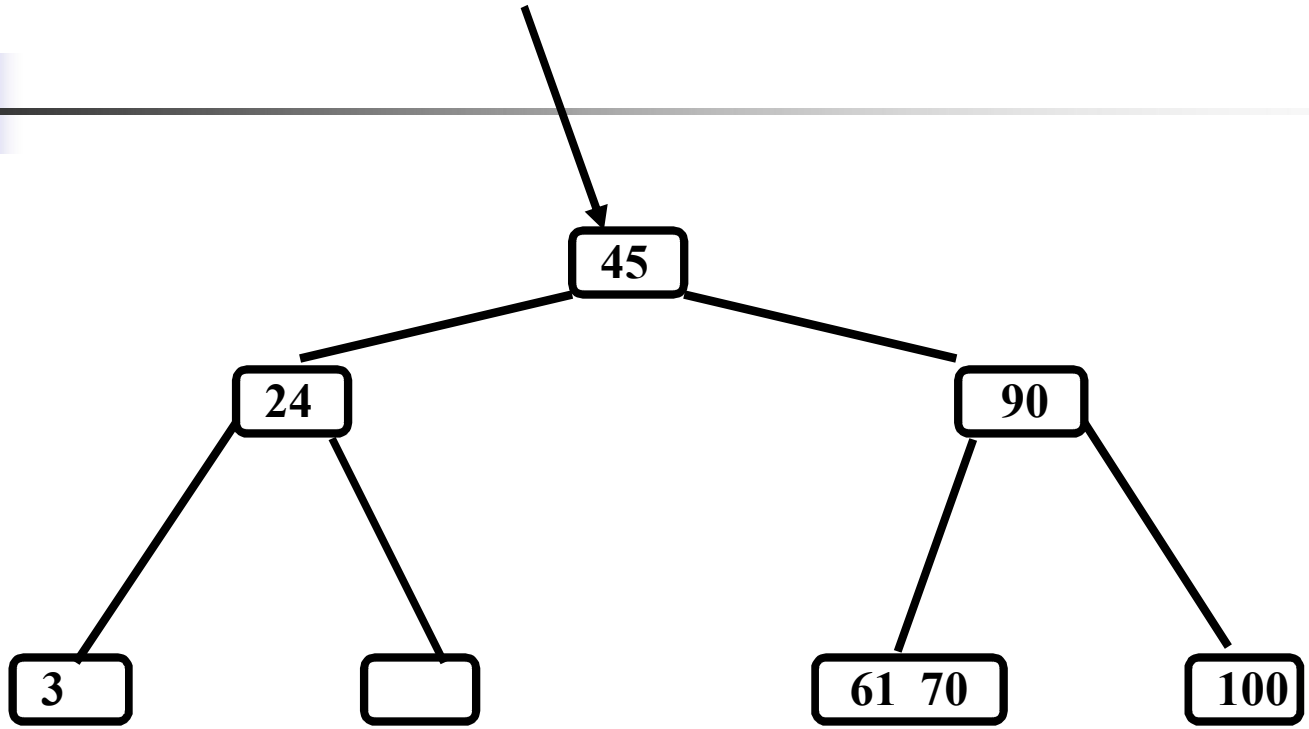
删除53

一棵3阶的B-树



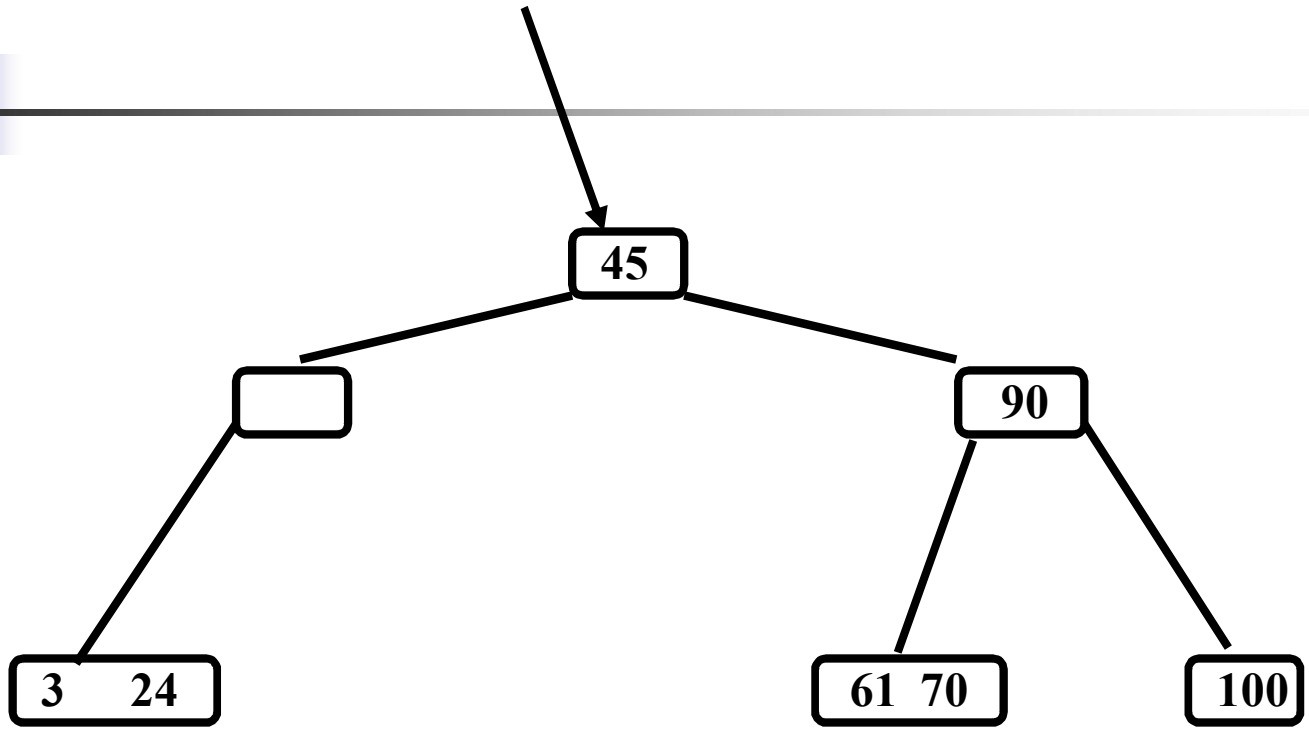
删除37

一棵3阶的B-树



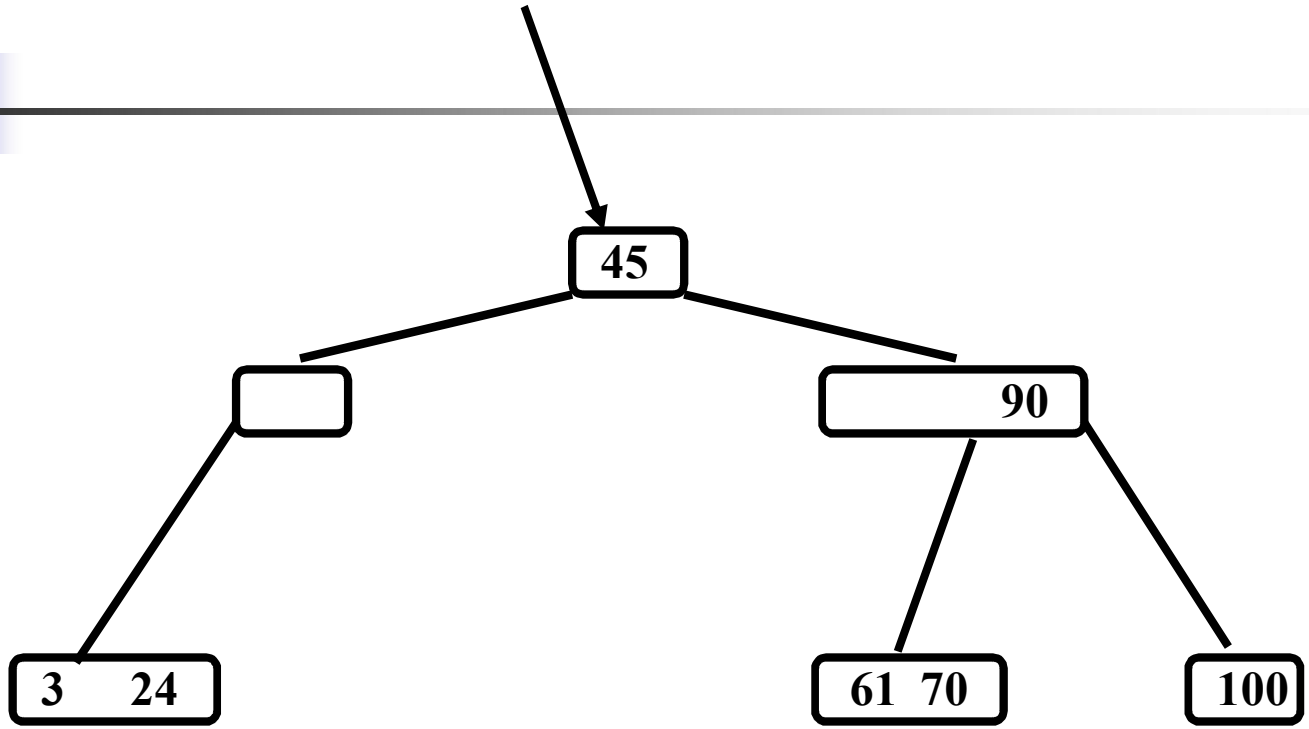
删除37

一棵3阶的B-树



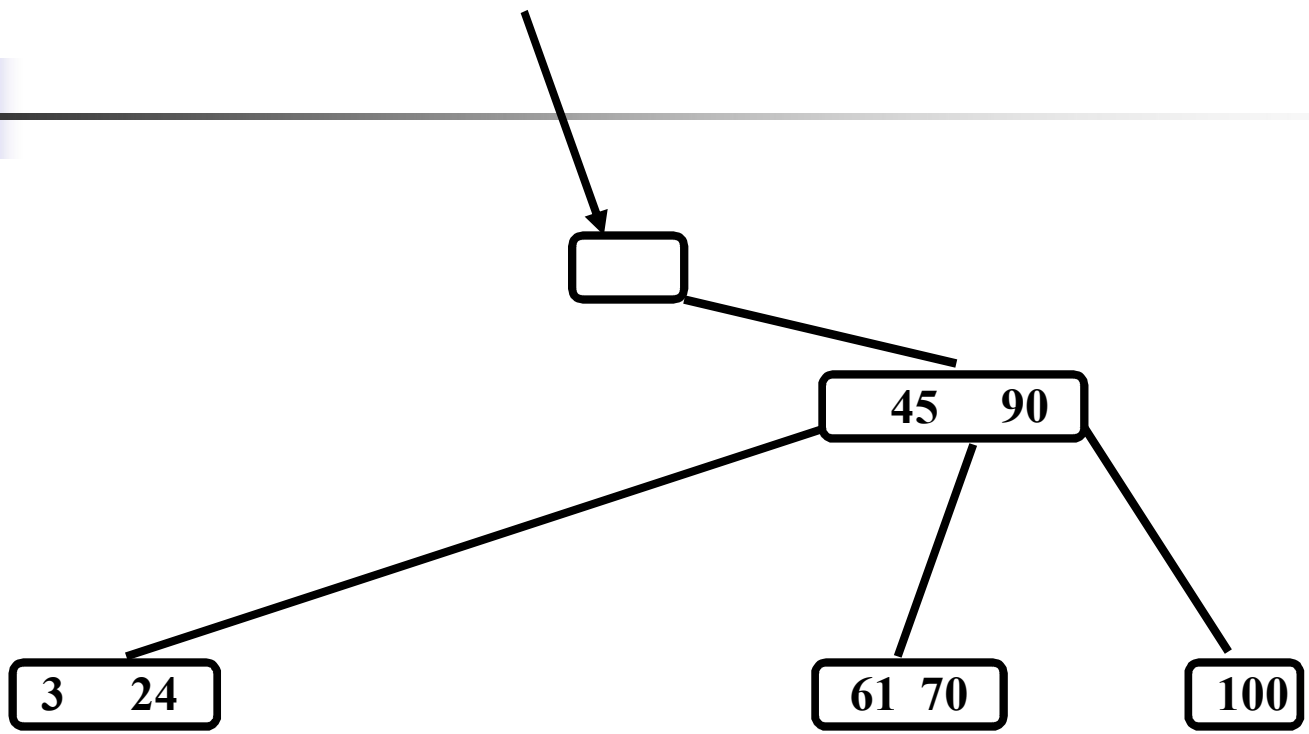
删除37

一棵3阶的B-树



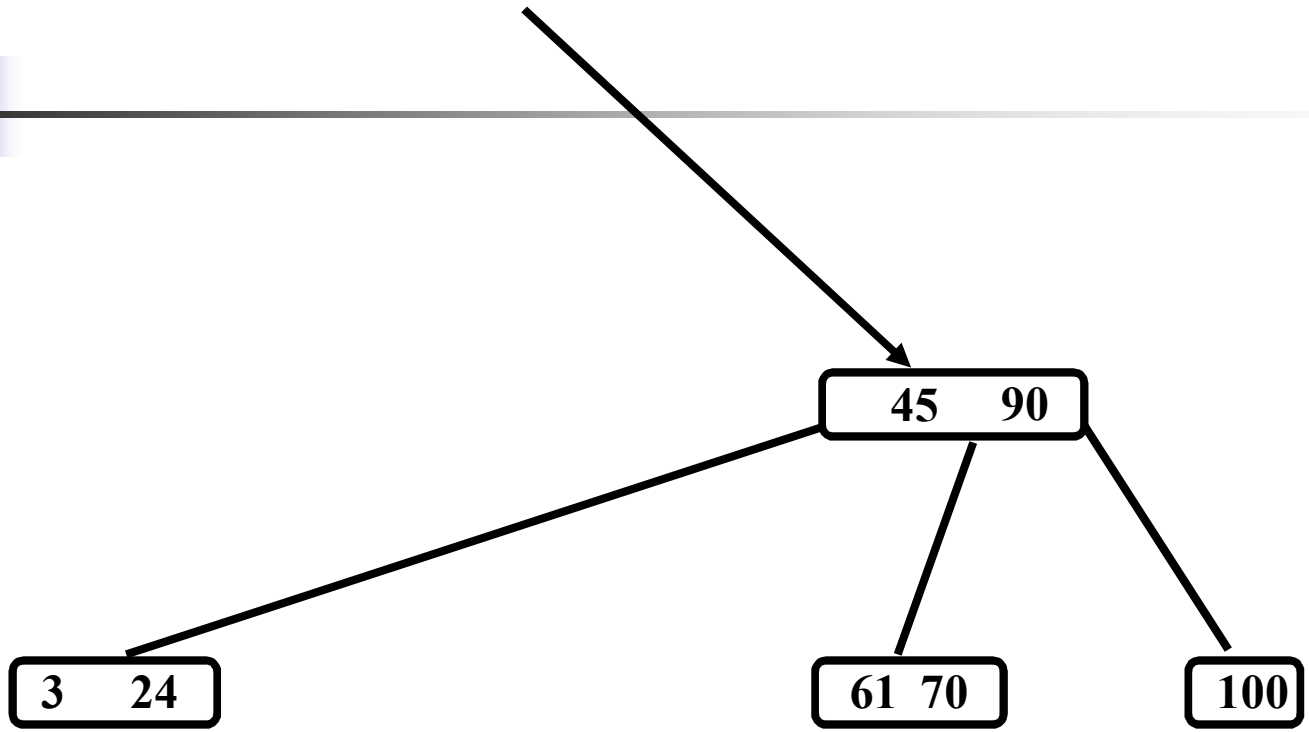
删除37

一棵3阶的B-树



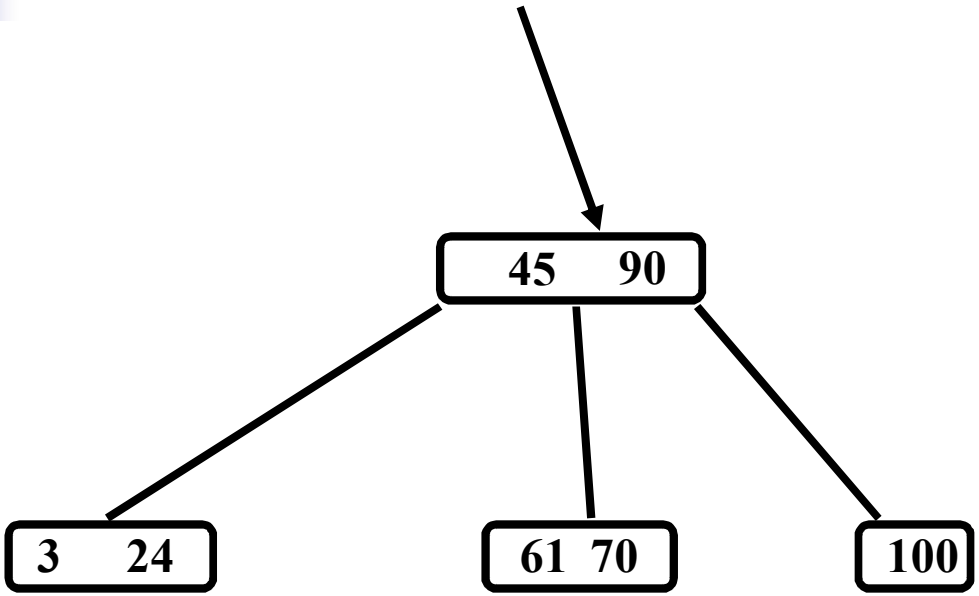
删除37

一棵3阶的B-树



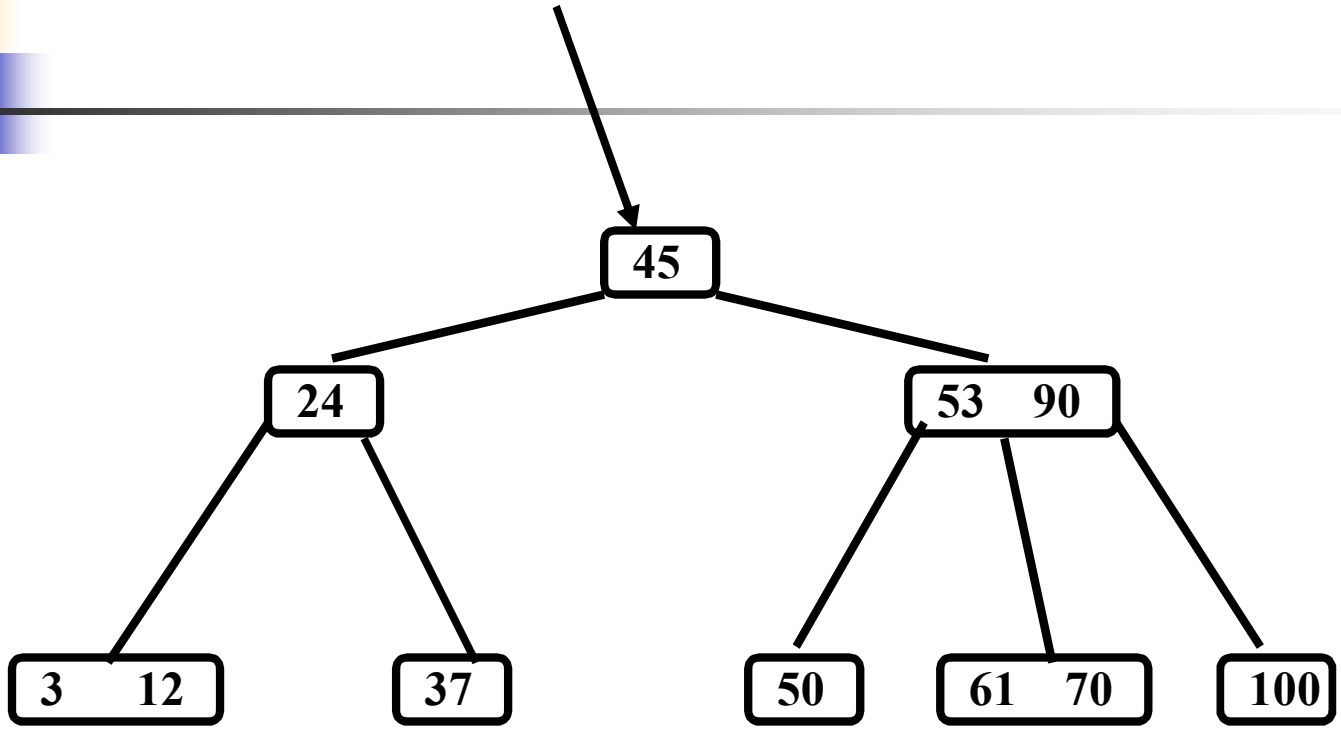
删除37

一棵3阶的B-树



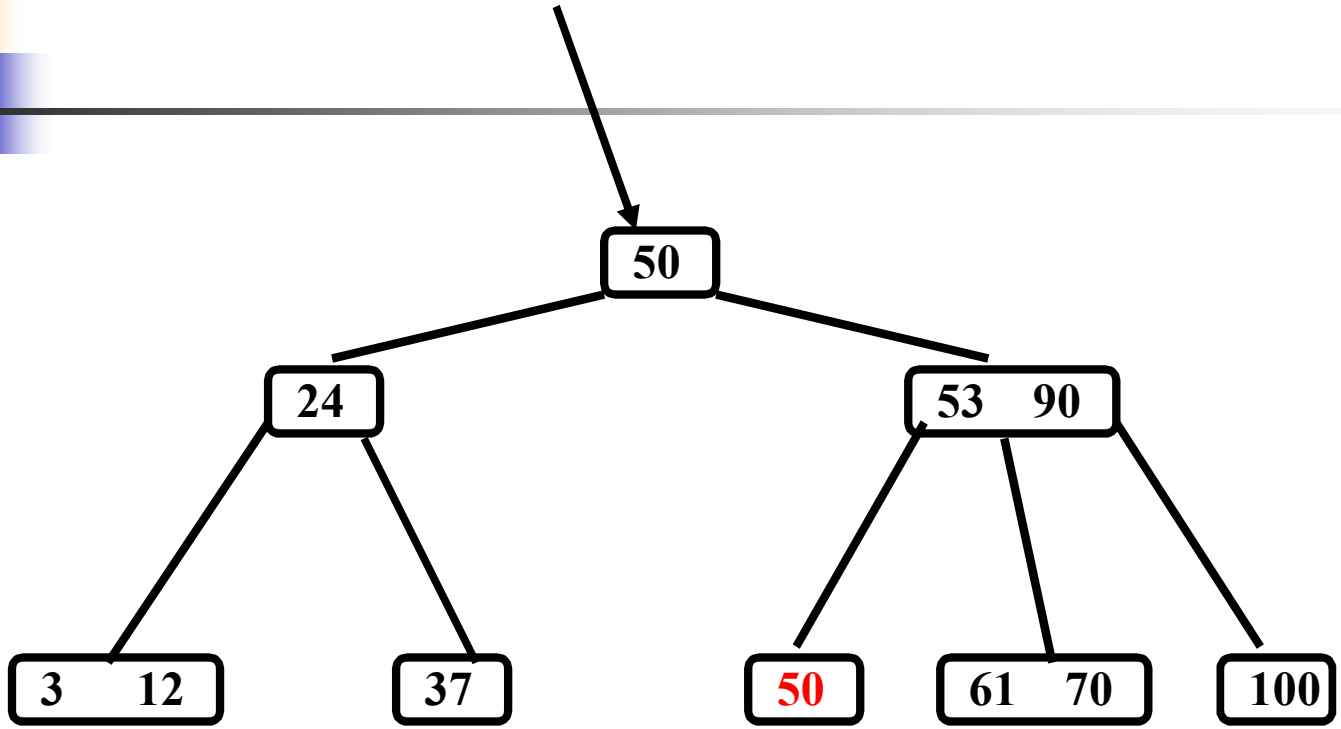
删除37

一棵3阶的B-树



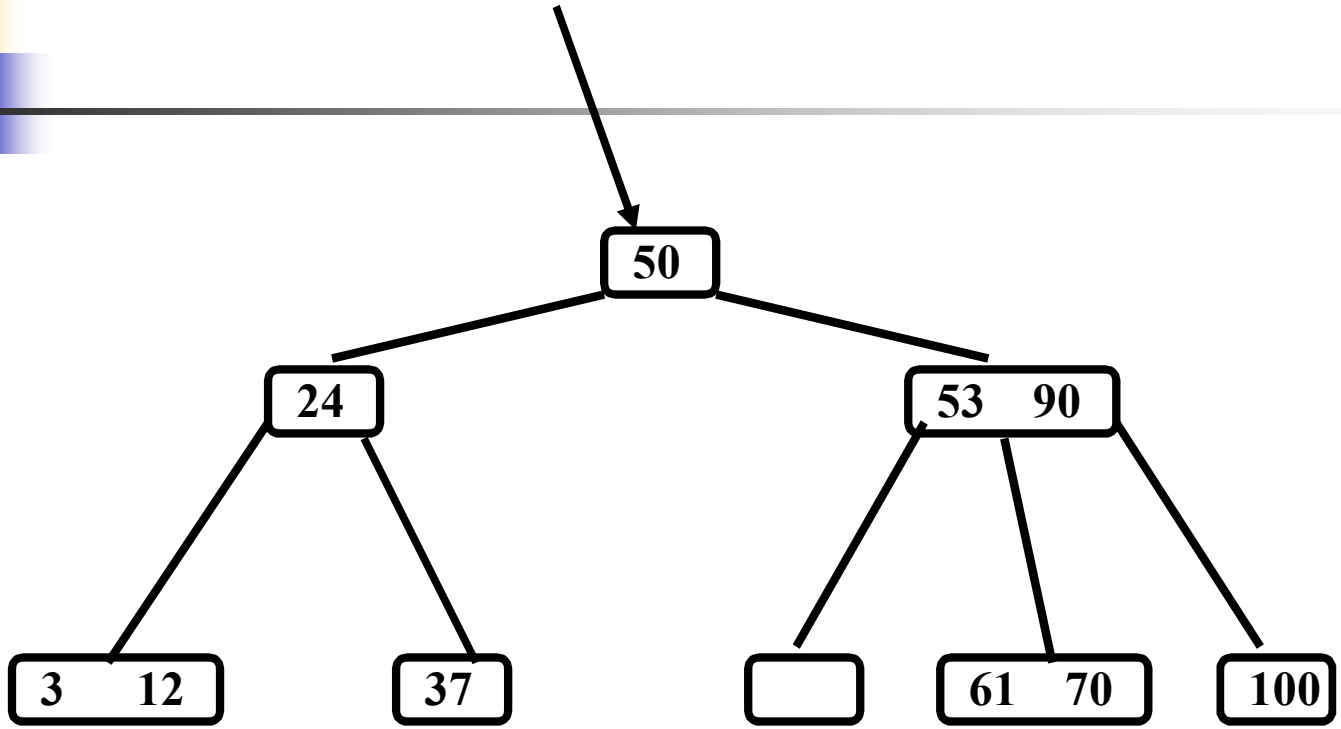
删除45

一棵3阶的B-树



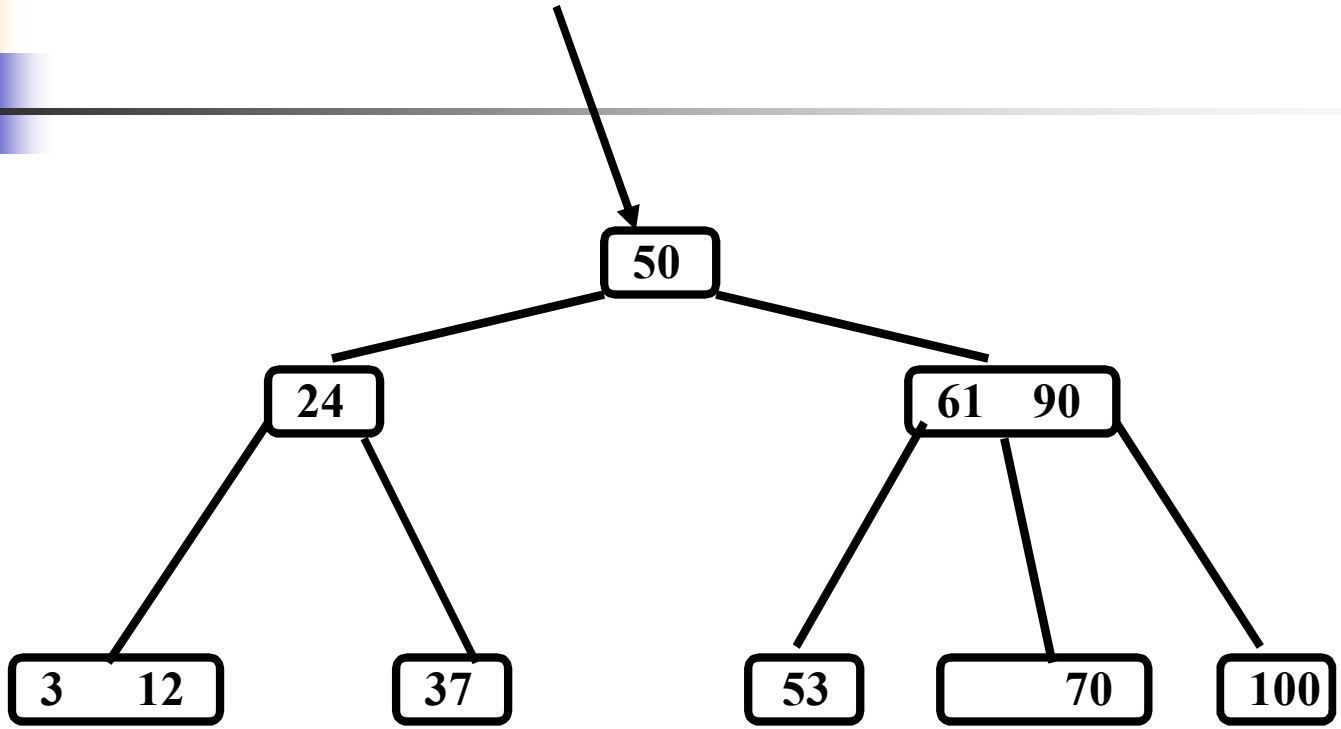
删除45

一棵3阶的B-树



删除45

一棵3阶的B-树

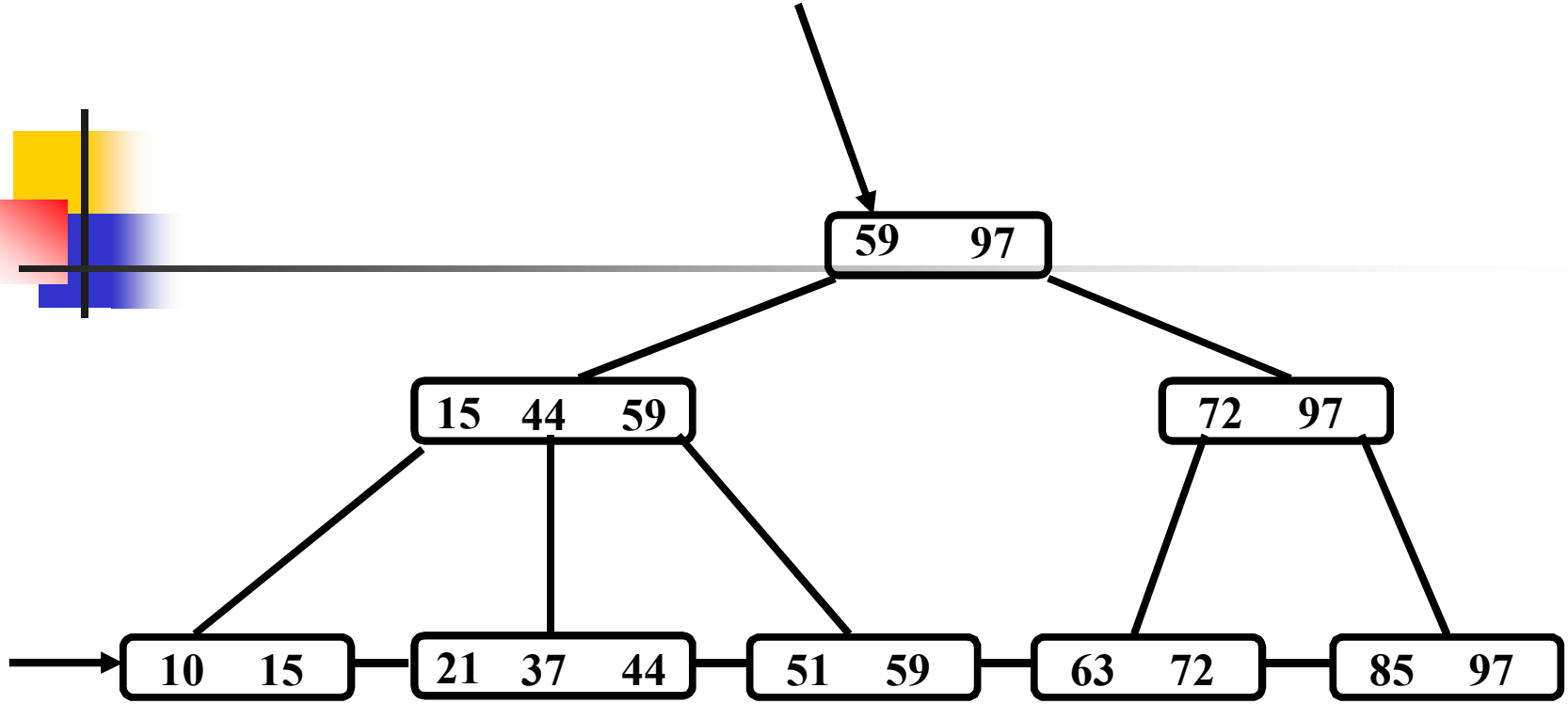


删除45

一棵3阶的B-树

B+树

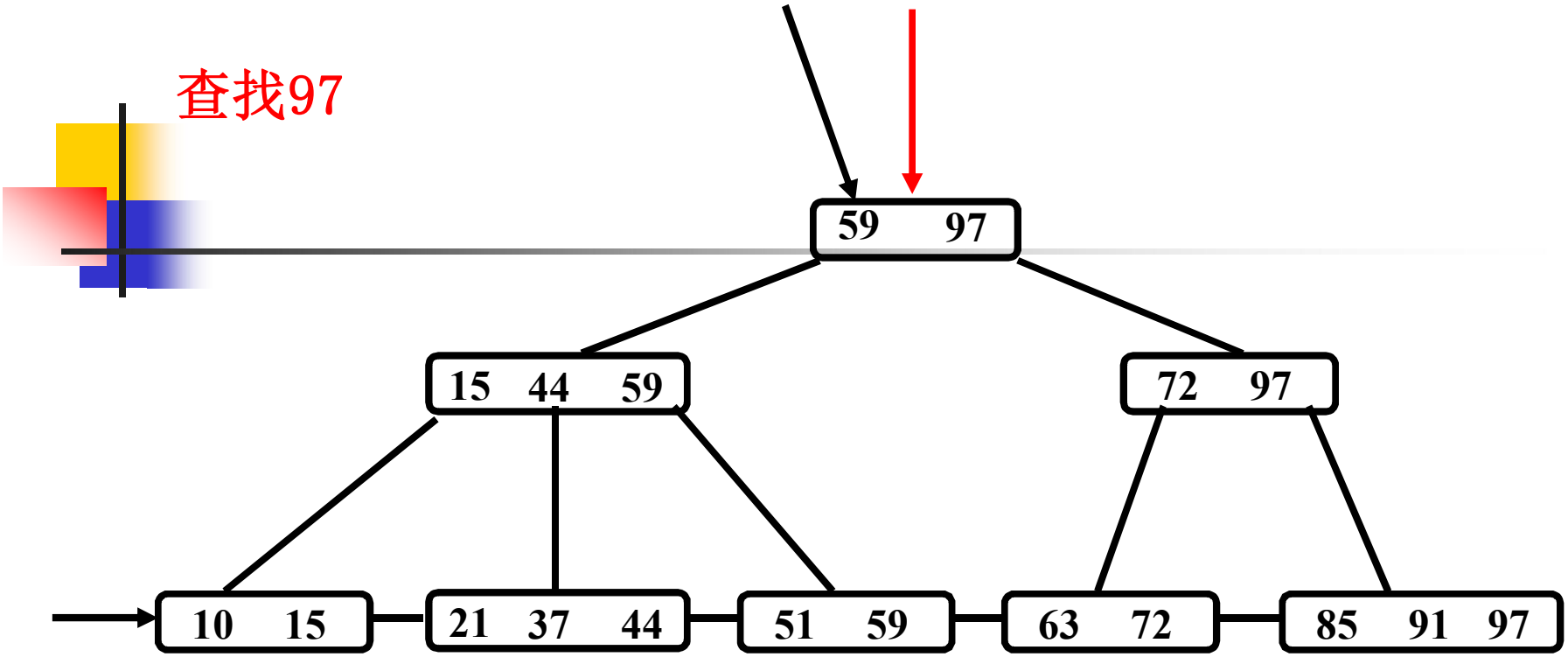
- B+树是应文件系统所需而产生的一种B-树的变型树。一棵*m*阶的B+树和*m*阶的B-树的差异在于：
  - 有*n*棵子树的结点中含有*n*个关键字；
  - 所有的叶子结点中包含了全部关键字 的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。
  - 所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。



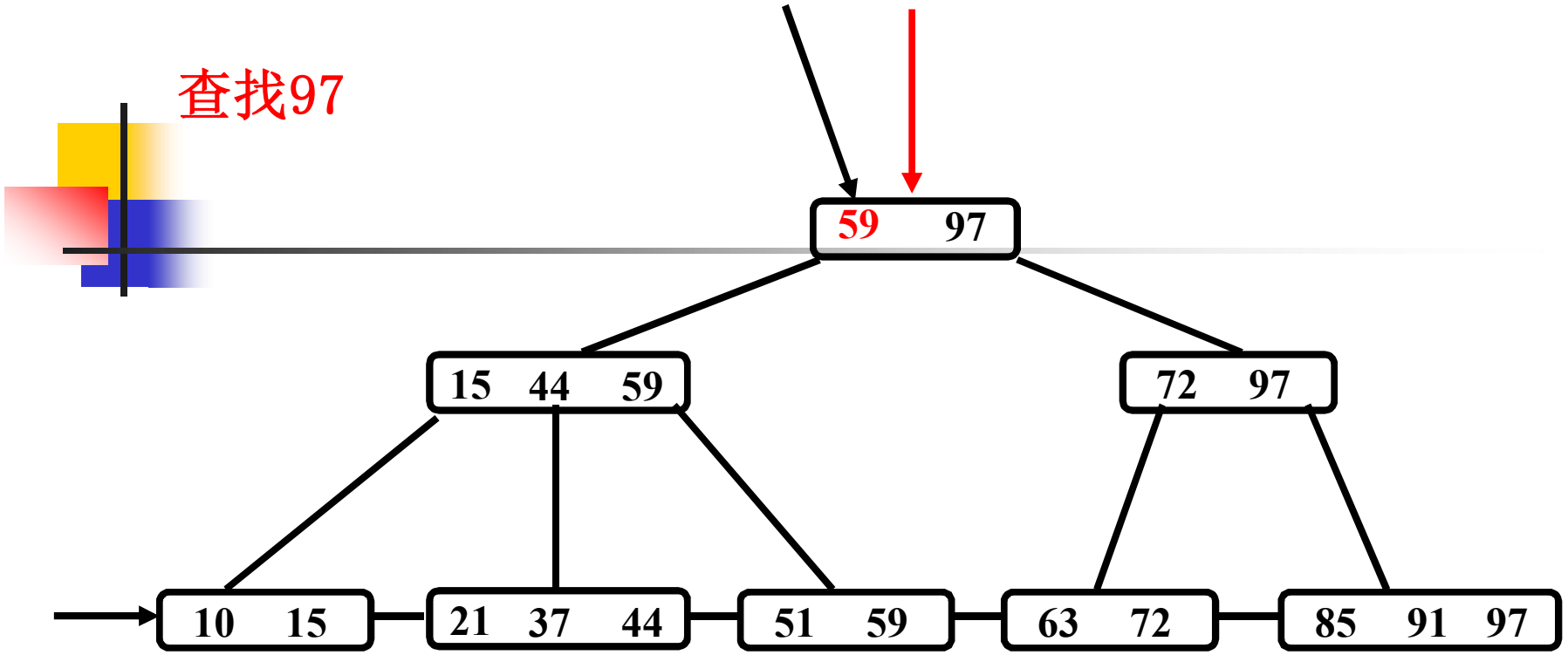
一棵3阶的B+树

B+树的查找

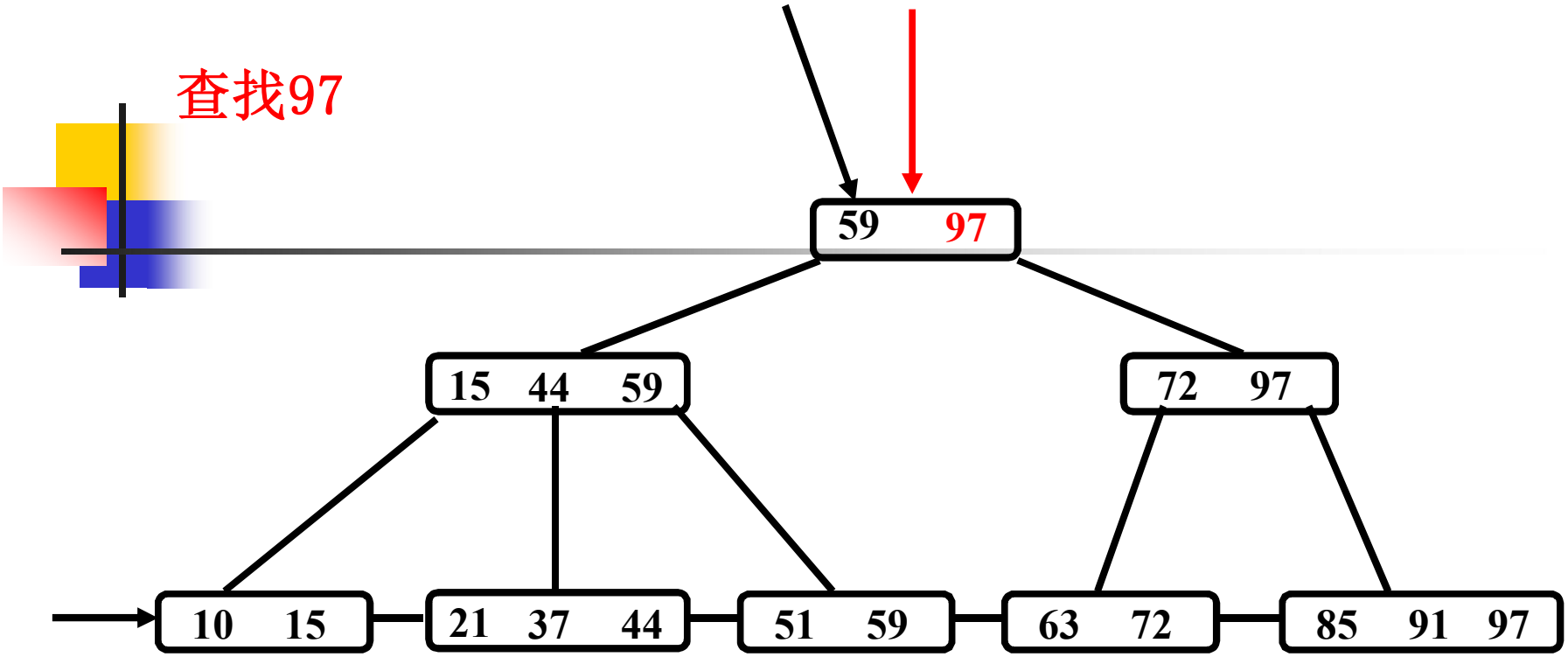
- 和B-树的查找稍有不同：
- 在B+树上，既可以进行**缩小范围**的查找，也可以进行**顺序查找**；
- 在进行缩小范围的查找时，不管成功与否，都必须**查到叶子结点才能结束**；
- 若在结点内查找时，给定值≤K_i，则应继续在A_i所指子树中进行查找。



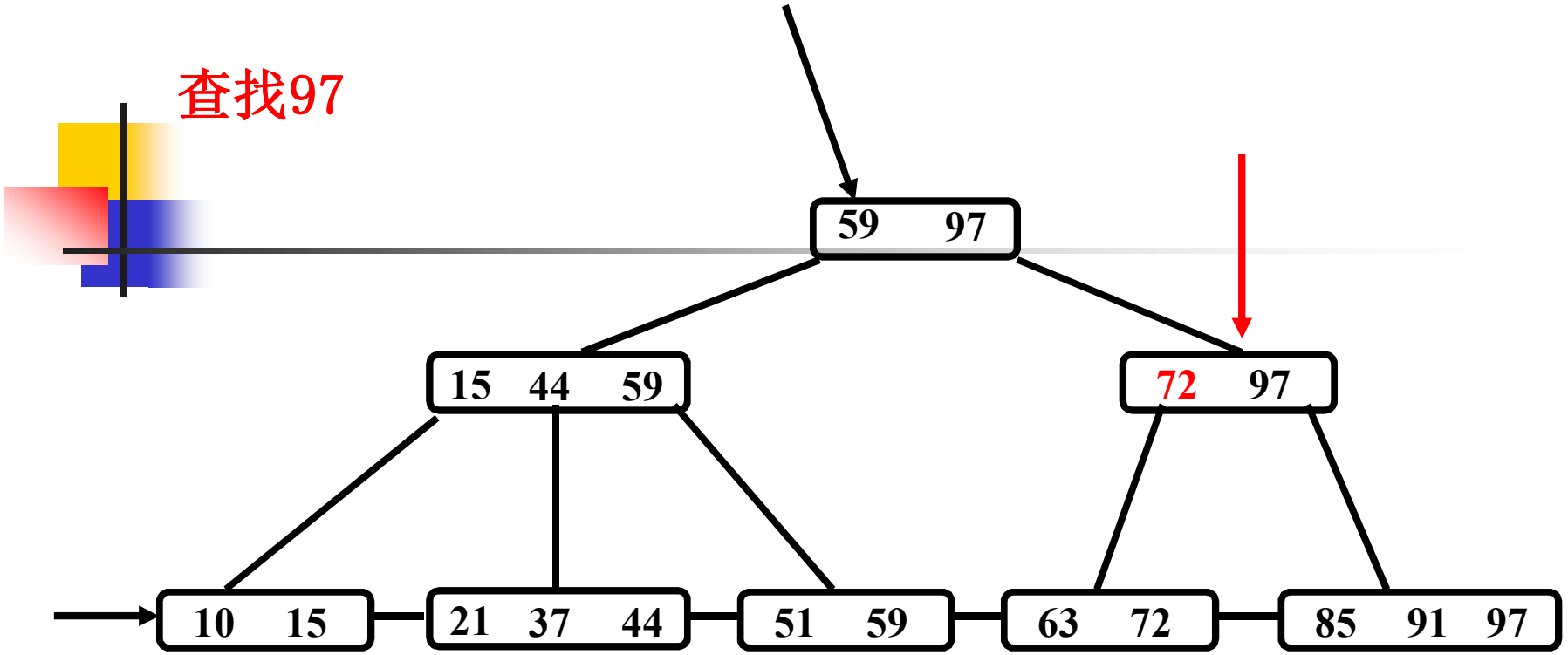
一棵3阶的B+树



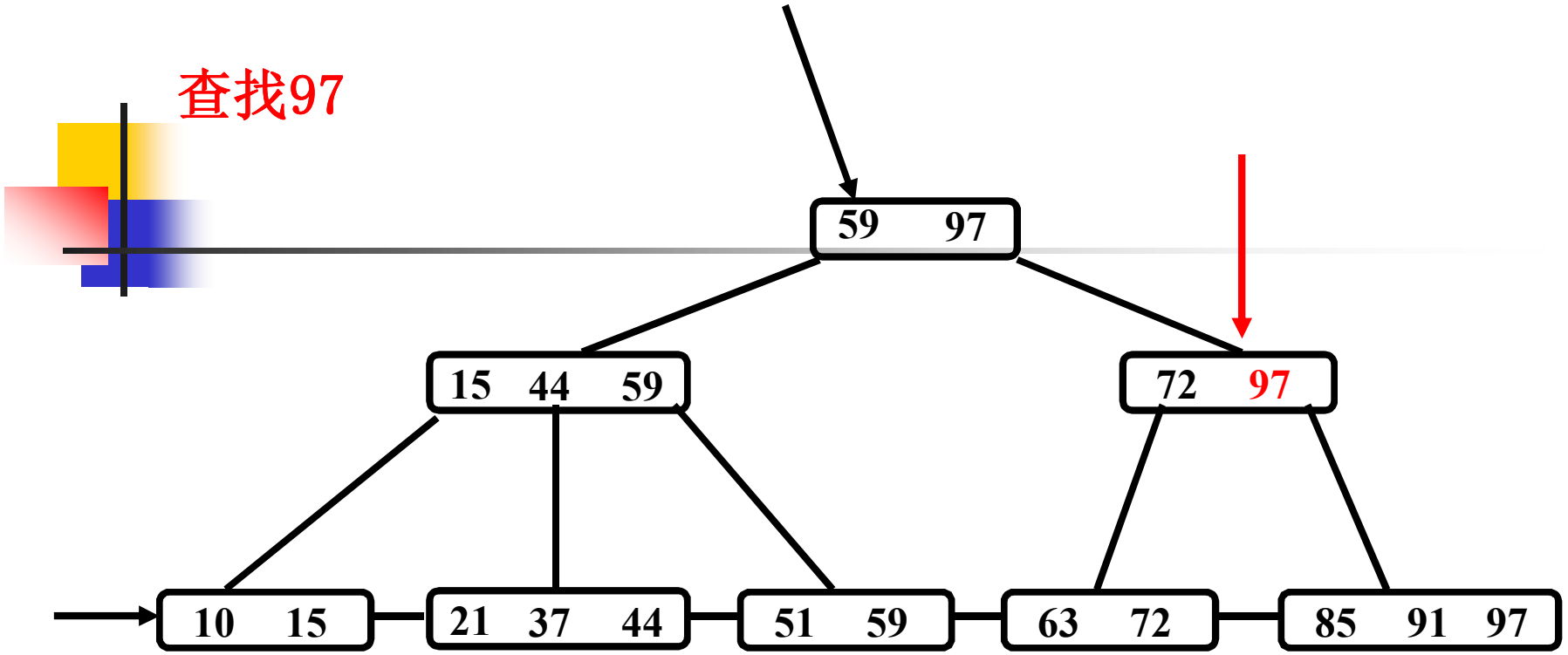
一棵3阶的B+树



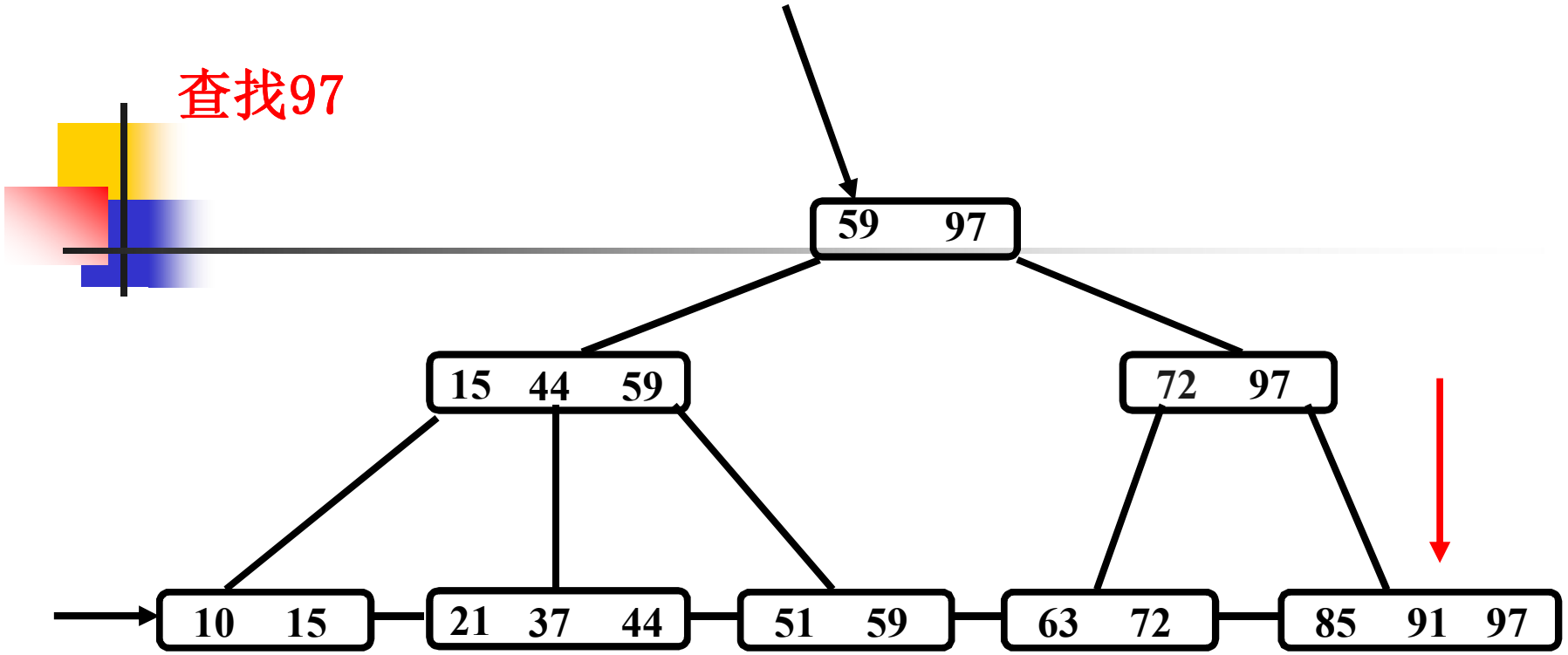
一棵3阶的B+树



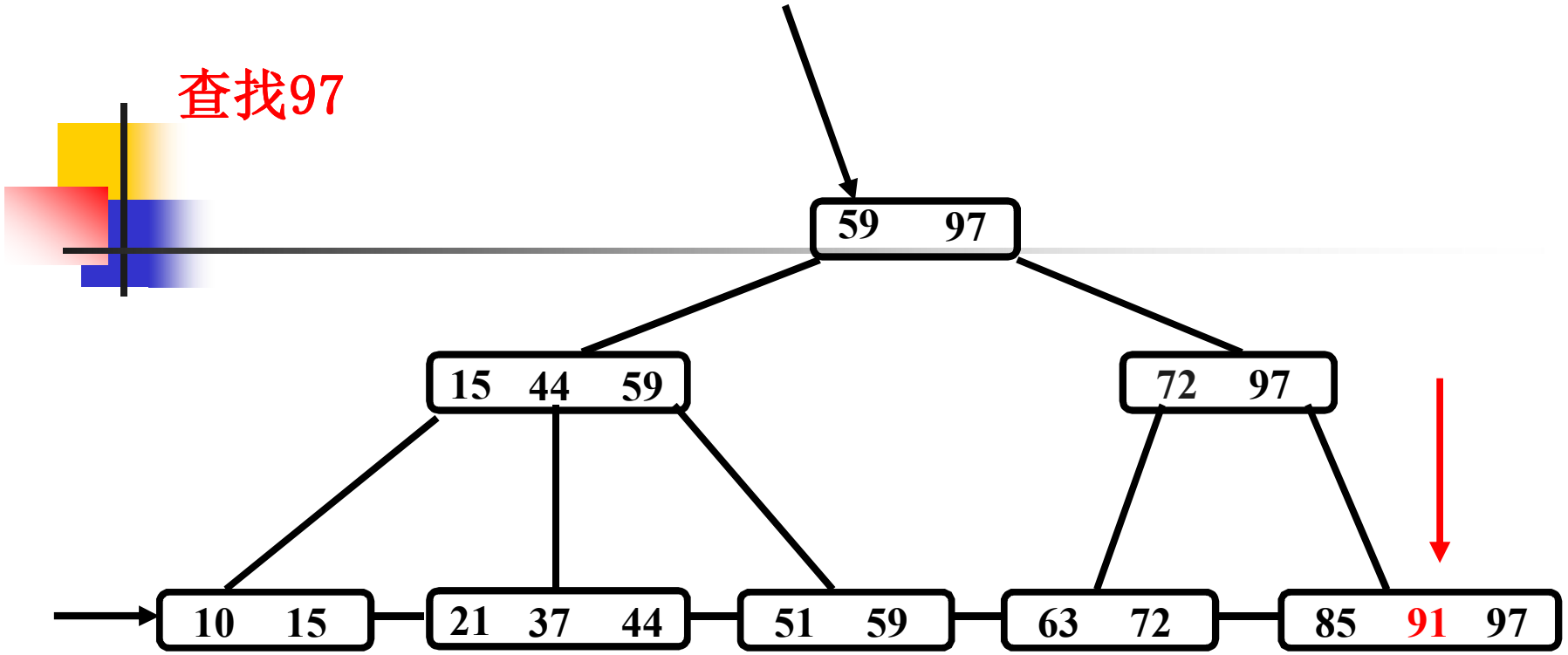
一棵3阶的B+树



一棵3阶的B+树

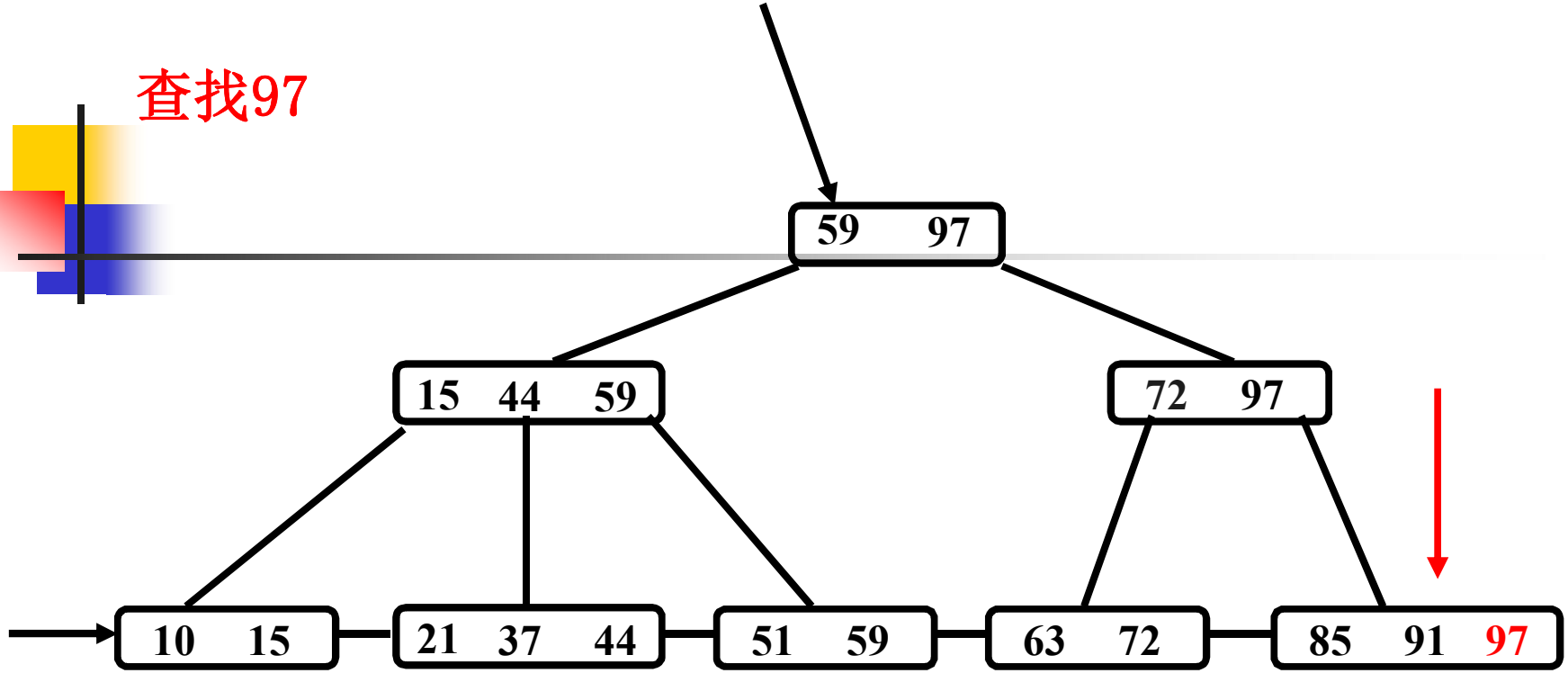


一棵3阶的B+树



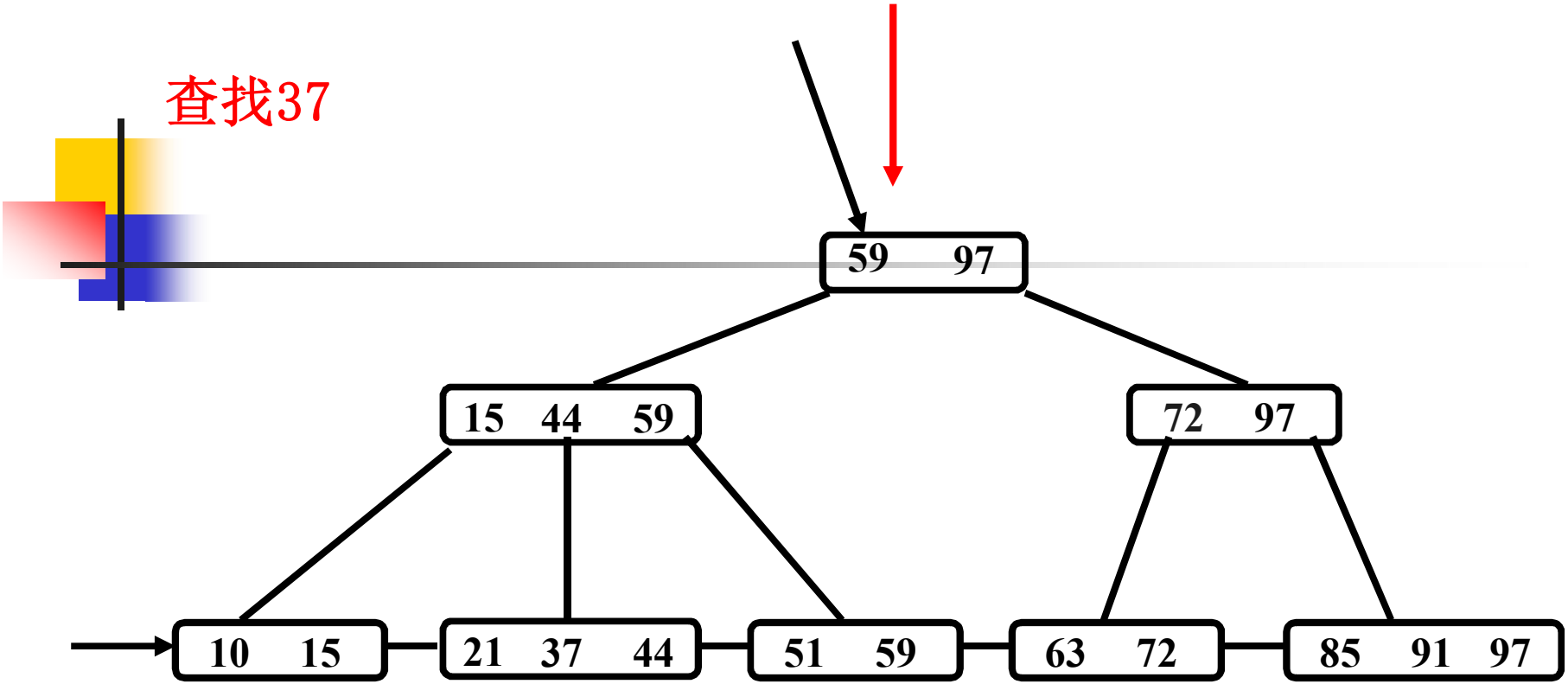
一棵3阶的B+树

查找97

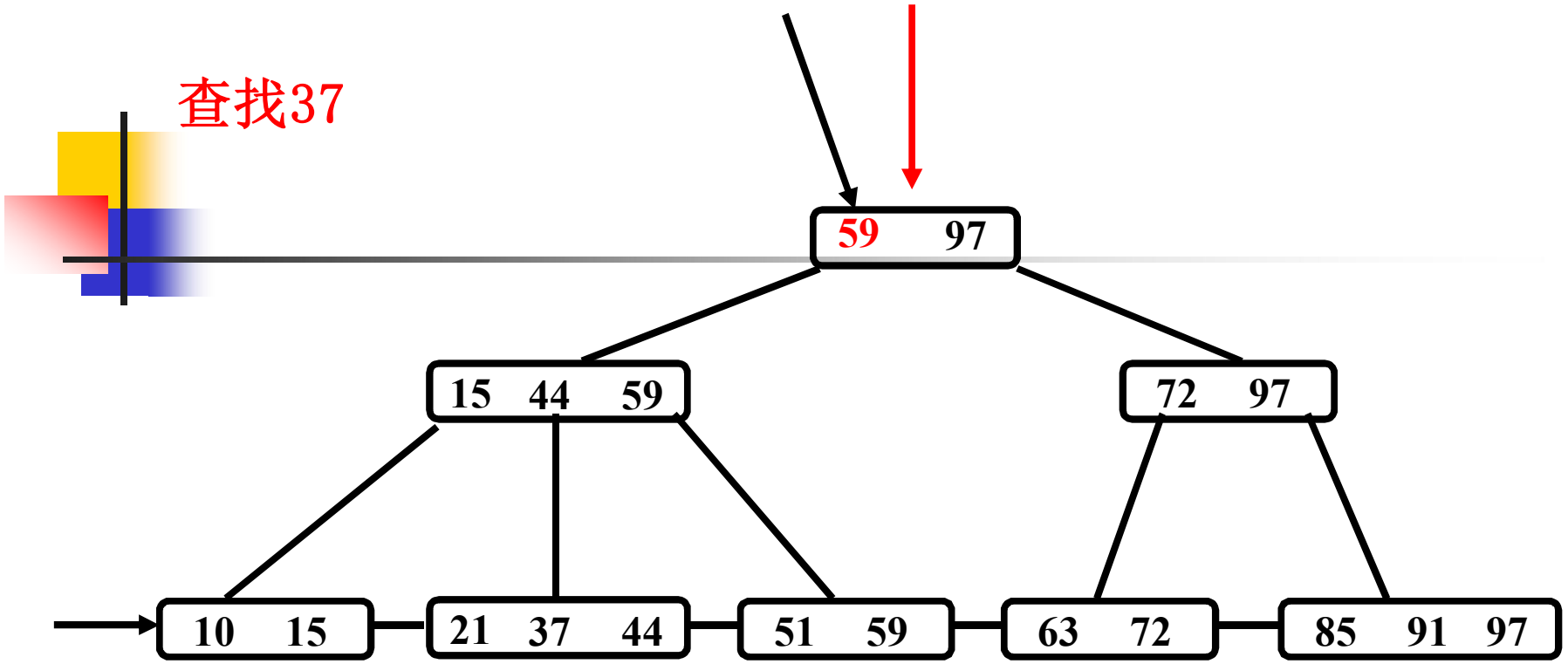


查找成功！

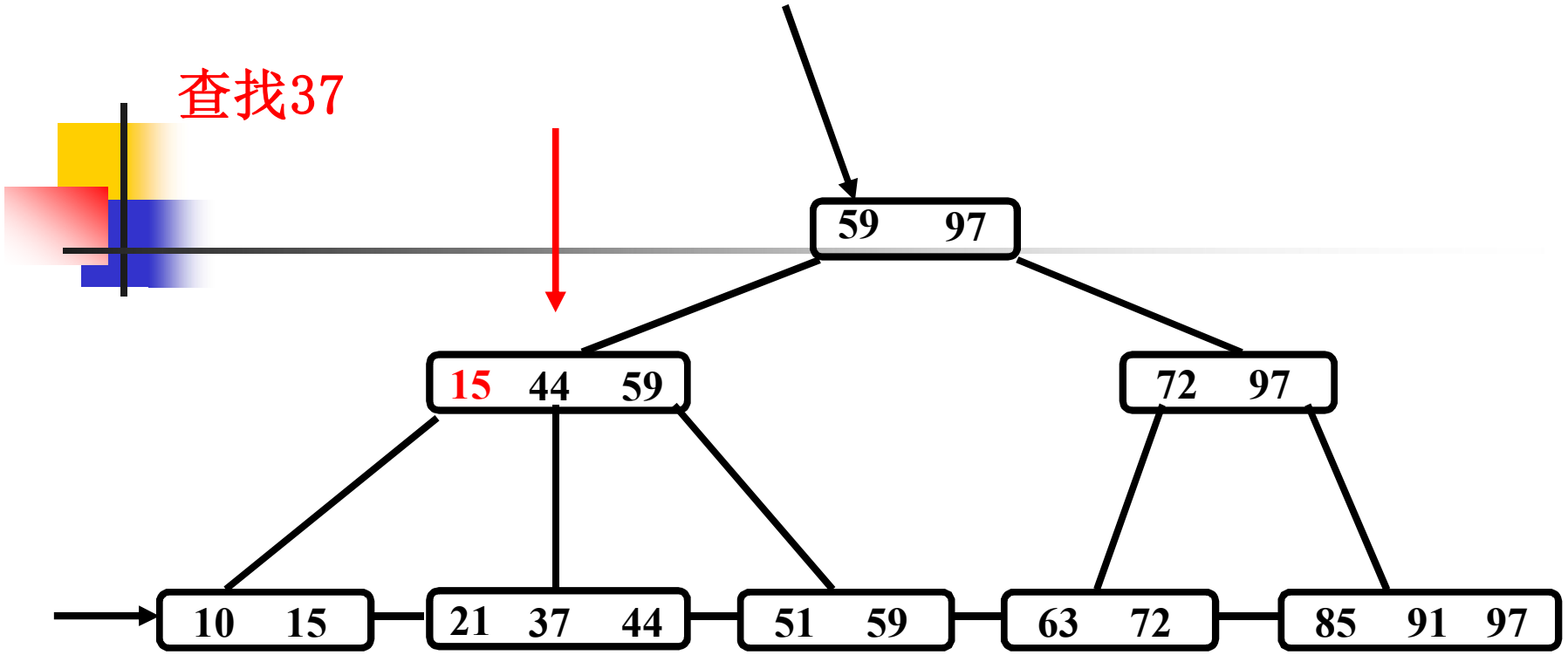
一棵3阶的B+树



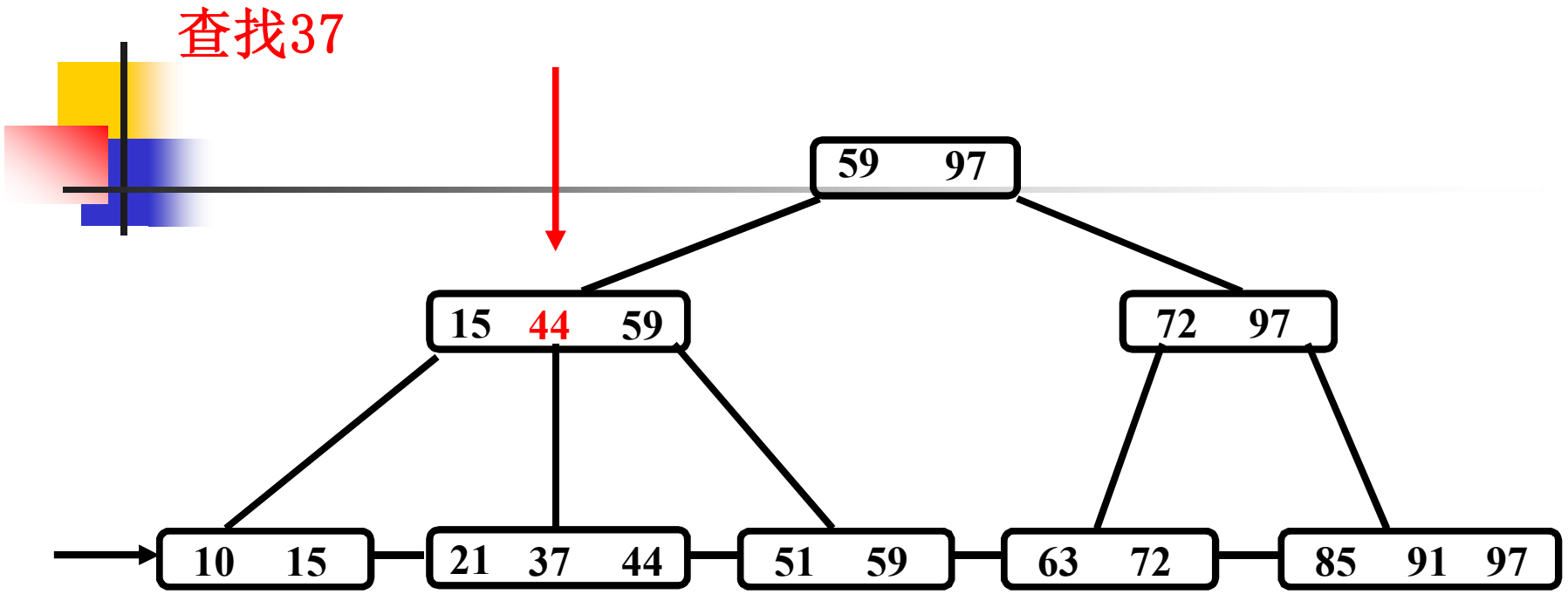
一棵3阶的B+树



一棵3阶的B+树

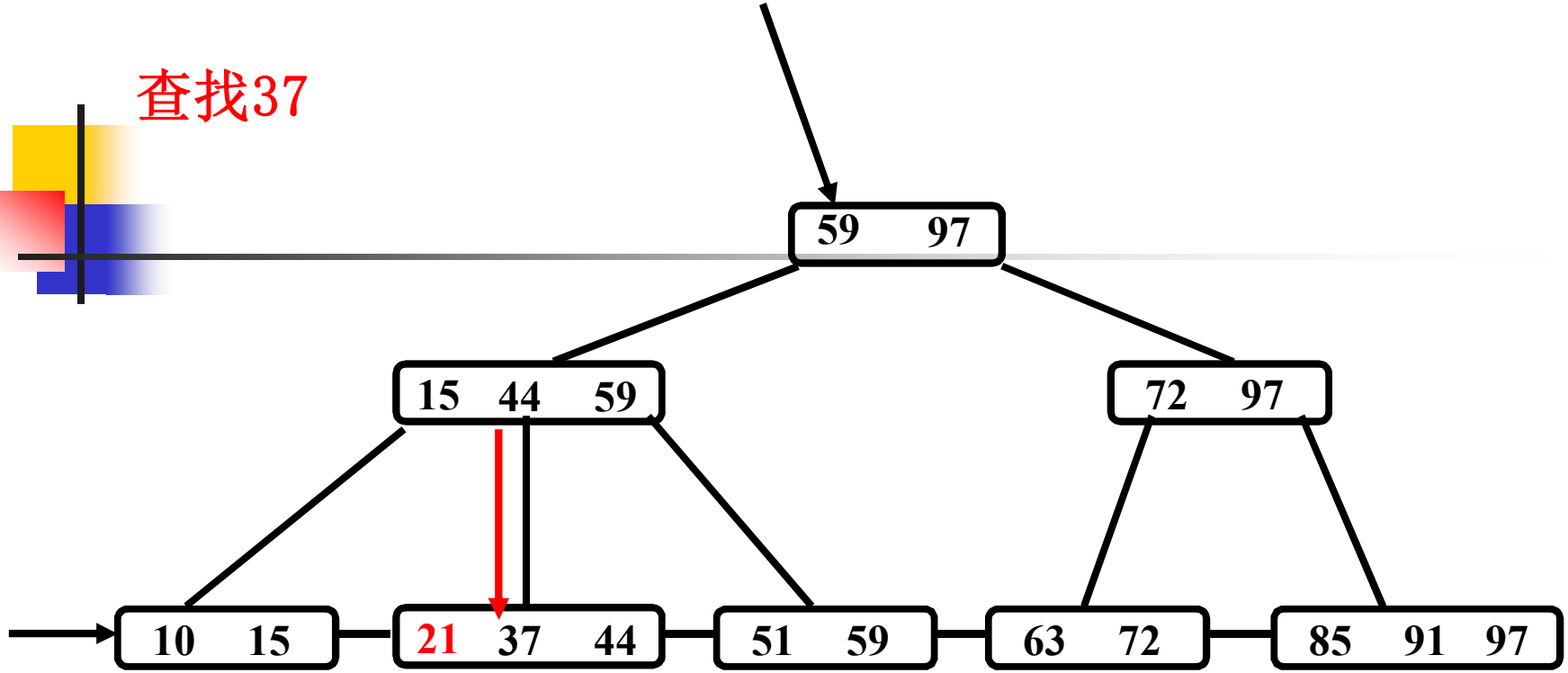


一棵3阶的B+树



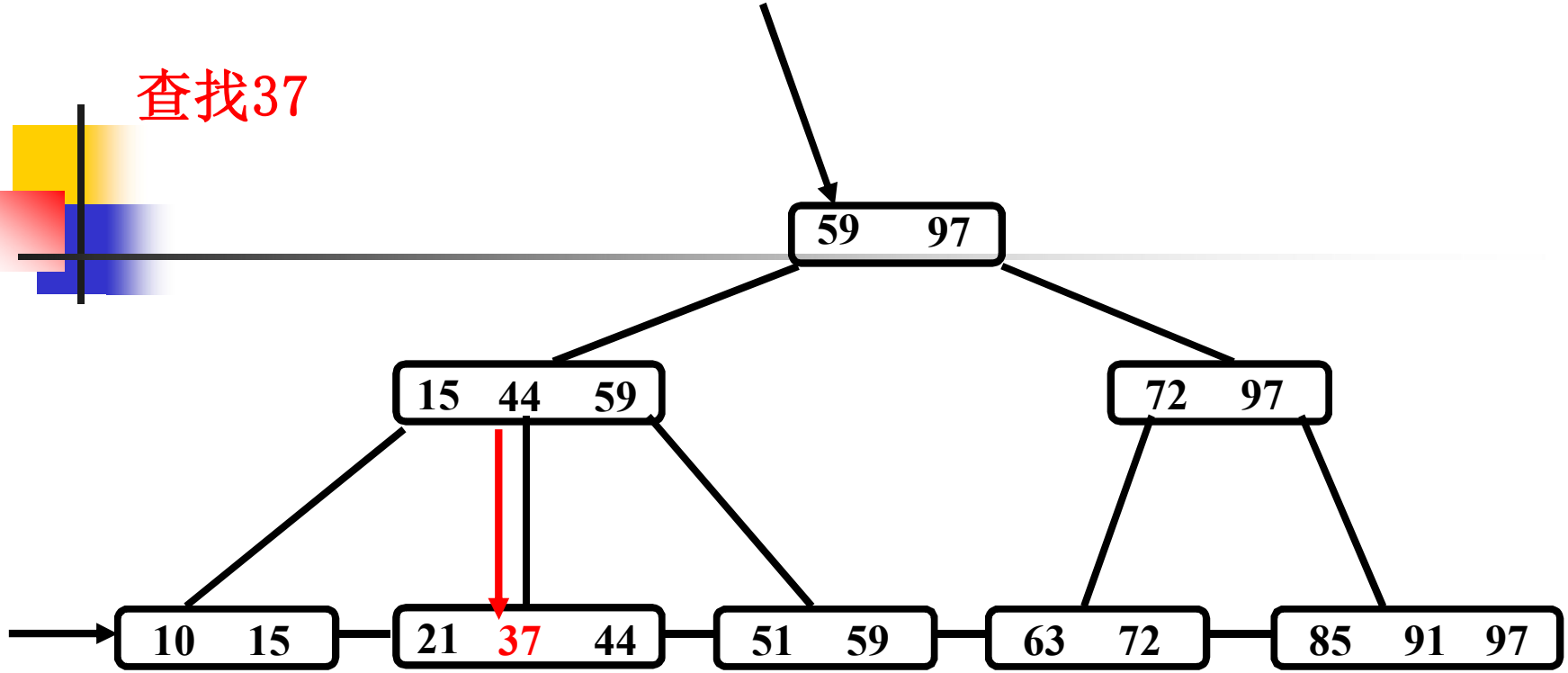
一棵3阶的B+树

查找37



一棵3阶的B+树

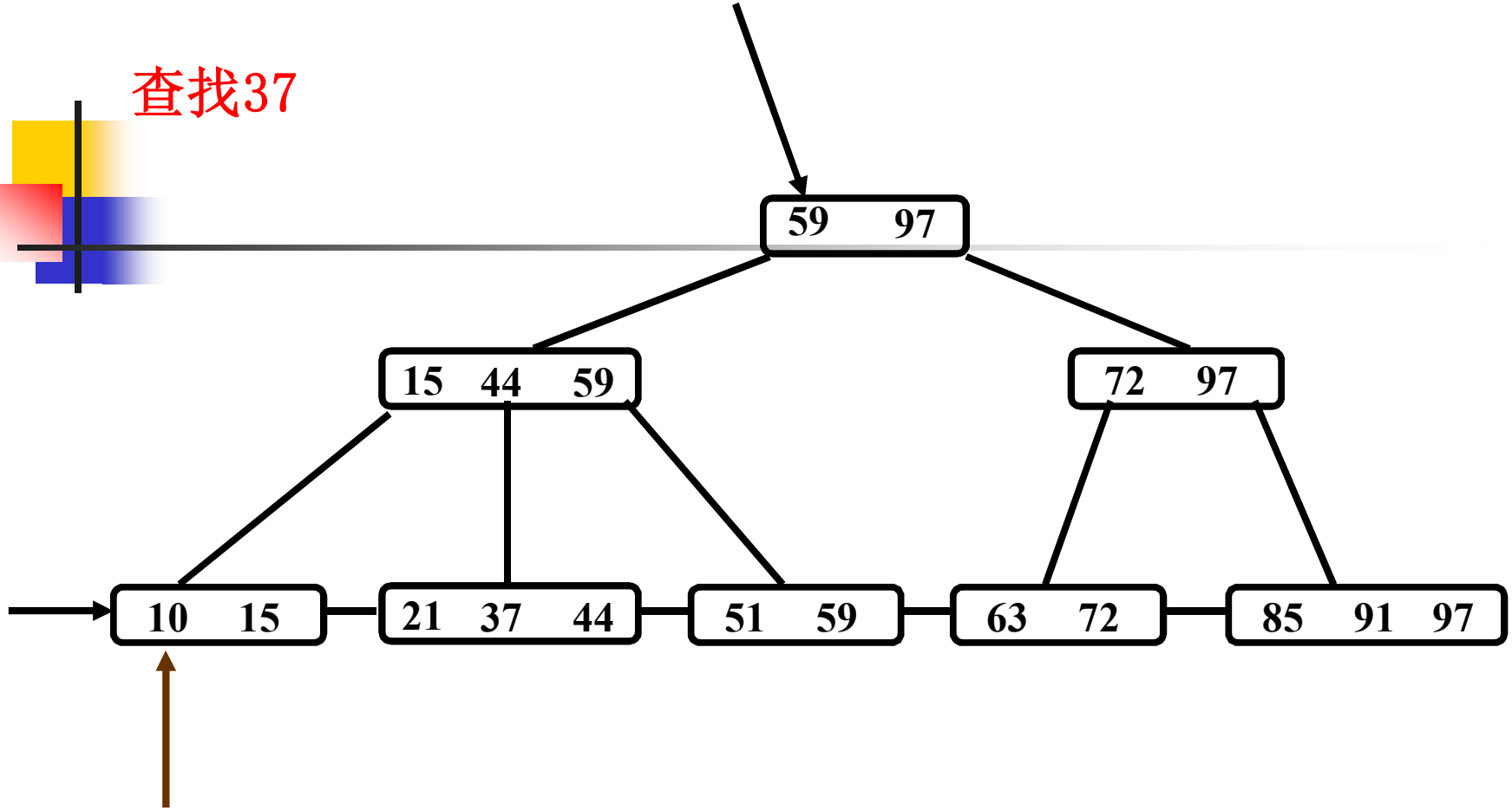
查找37



查找成功！

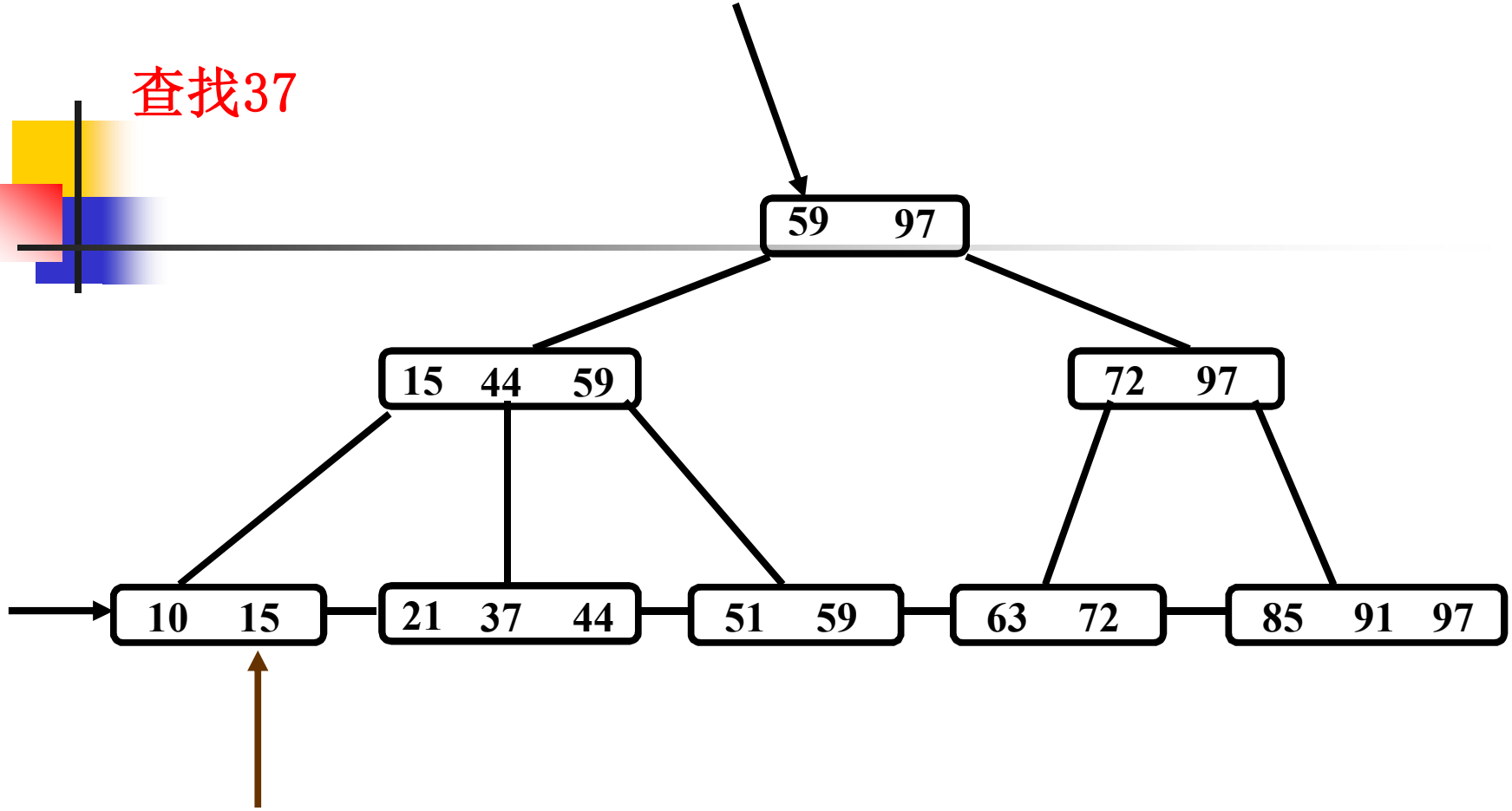
一棵3阶的B+树

查找37



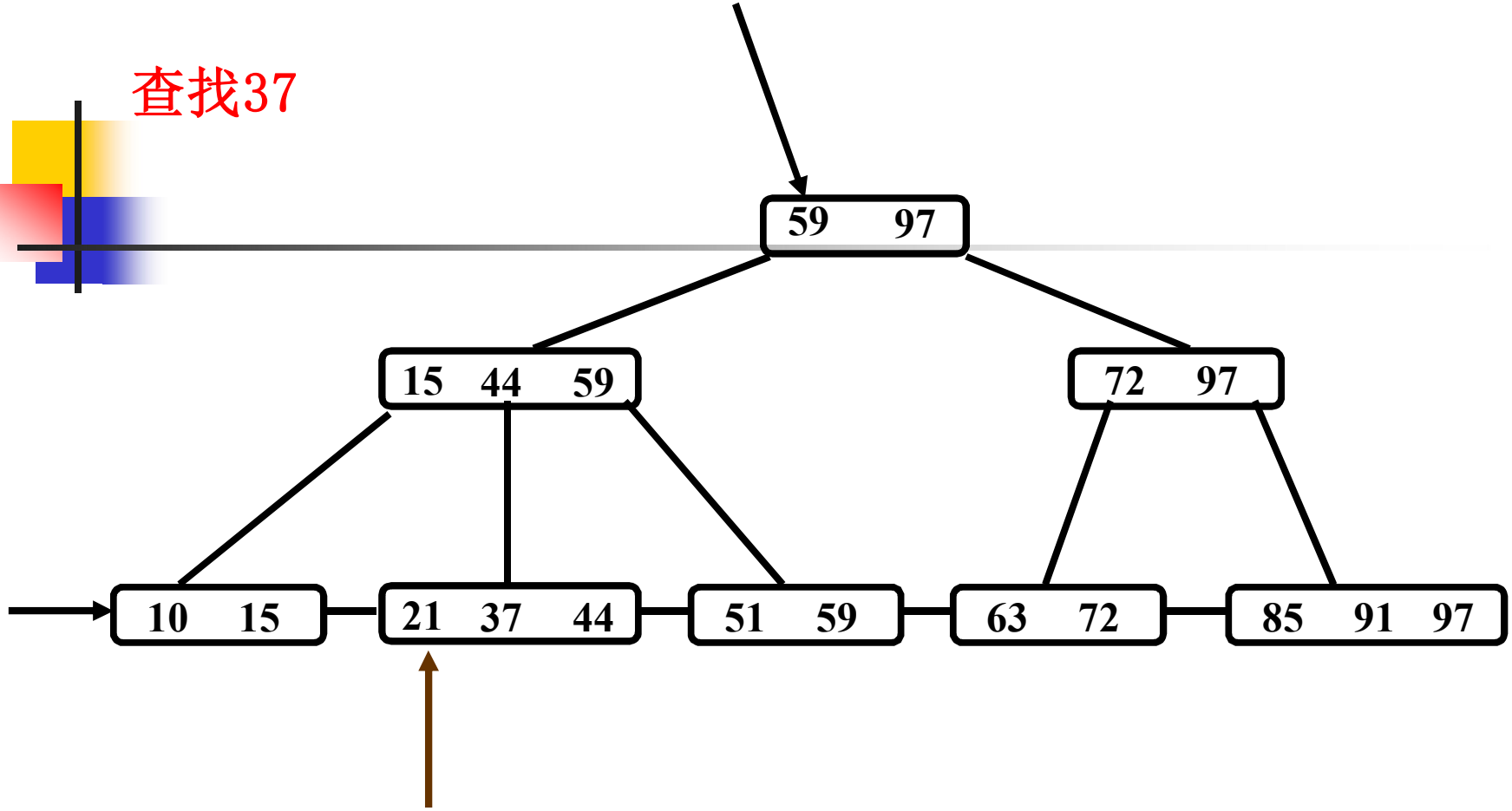
一棵3阶的B+树

查找37



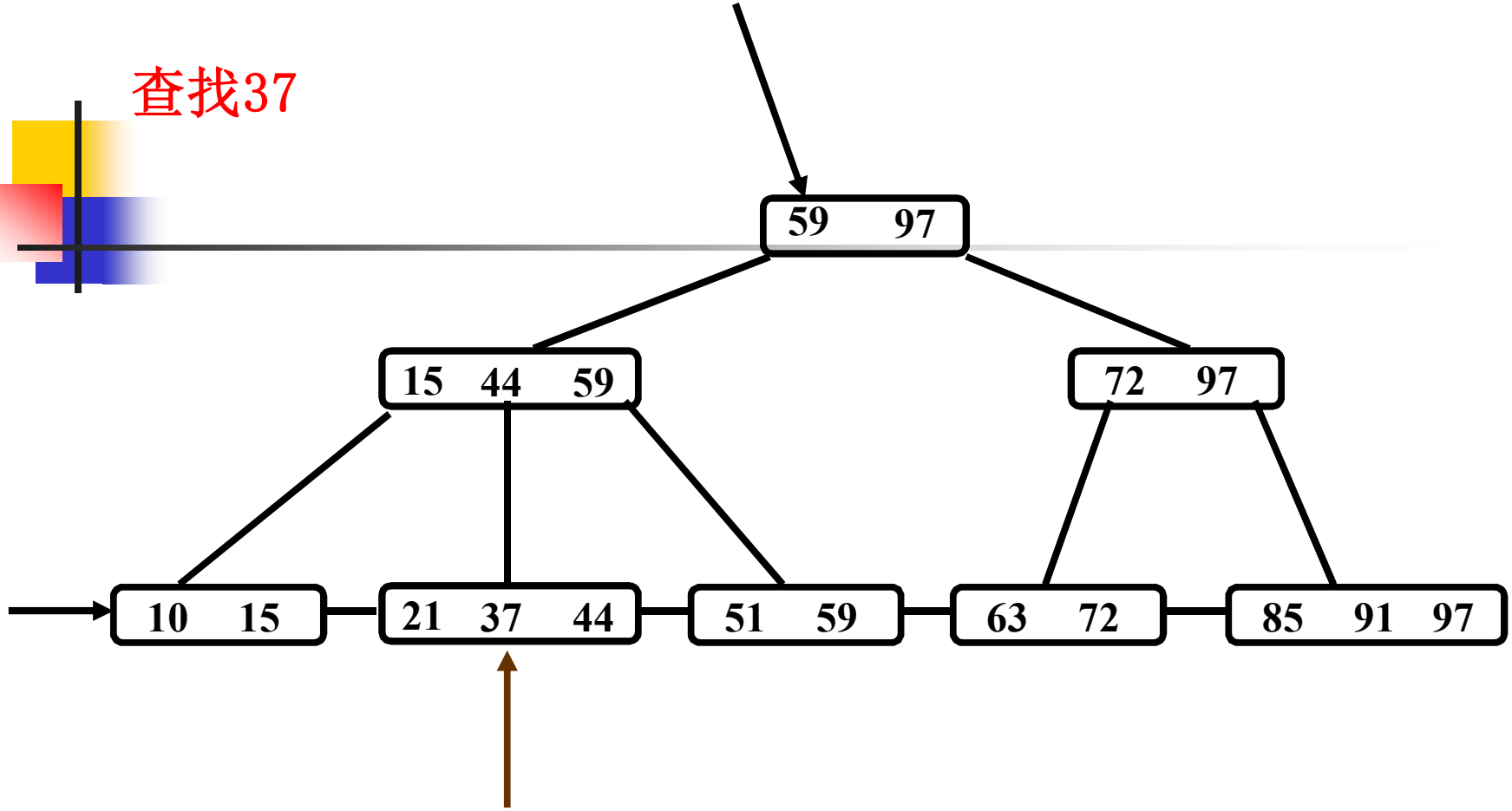
一棵3阶的B+树

查找37



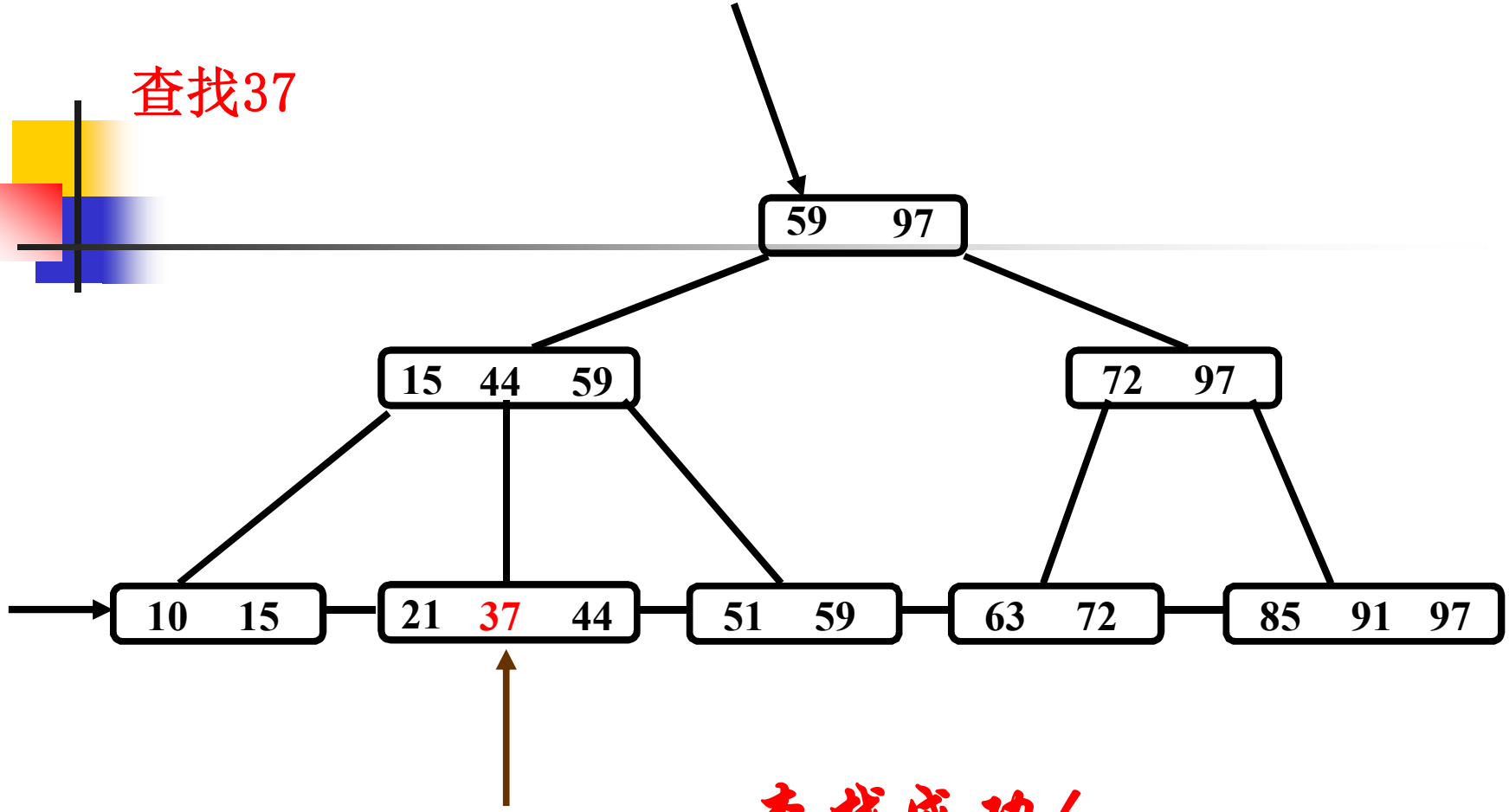
一棵3阶的B+树

查找37



一棵3阶的B+树

查找37



查找成功！

一棵3阶的B+树