



教学要求

- 了解不同数据结构上的查找方法；
- 掌握各种查找结构的性质、在静态和动态环境下的查找算法的设计思想和实现方法；
- 掌握各种查找方法的时间性能(平均查找长度)的分析方法；
- 能够根据具体情况选择适合的方法解决实际问题。

主要内容

- | | |
|-----|-------------|
| 5.1 | 线性查找 |
| 5.2 | 折半查找 |
| 5.3 | 分块查找 |
| 5.4 | 二叉查找树 |
| 5.5 | 平衡二叉树 |
| 5.6 | B-树 |
| 5.7 | 地址散列法（哈希查找） |



【定义】 查找是根据给定的值，在查找表中确定一个其关键字等于给定值的纪录或数据元素的过程。

线性查找

折半查找

分快查找

二叉查找树

传统的查找方法

查找分类

较先进的查找方法 (散列法, 哈希表)

几个概念:

- **查找表**: 由同一类型的数据元素 (或纪录) 构成的集合;
- **关键字**: 数据元素中某一数据项的值, 用以表示一个数据元素;
- **静态查找**: 查找+提取数据元素属性信息;
- **动态查找**: 查找 + (插入或删除元素)。

5.1 线性查找

【算法一】顺序表上的线性查找

```
Int Search ( k, last, F )
```

```
Keytype k, int last; LIST F ;
```

```
/* 在F中查找关键字为k的纪录，若找到，则返回该纪录  
   所在的下标，否则返回0 */
```

```
{ int i ;
```

```
  F[0].key = k ;
```

```
  i = last ;
```

```
  while ( F[i].key != k )
```

```
    i = i - 1 ;
```

```
  return i ;
```

```
} /*Search */
```

```
Struct records {  
    keytype key ;  
    fields    other ; }  
Typedef records LIST[maxsize] ;  
LIST F ;
```

【算法二】 链表上的线性查找

```
LIST Search( k, F )
```

```
Keytype k; LIST F ;
```

```
{ LIST p ;
```

```
  p=F;
```

```
  while ( p! = NULL )
```

```
    if ( p->data.key == k )
```

```
      return p ;
```

```
    else
```

```
      p = p->next ;
```

```
  return p ;
```

```
}
```

```
Struct CellType {
    records data ;
    CellType * next ; }
Typedef CellType * LIST ;
```

性能分析

【定义】 为确定纪录在查找表中的位置，需和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的**平均查找长度**(Average Search Length, ASL)。

设： P_i 为查找表中第*i*个记录的概率， $\sum P_i=1$

C_i 为比较次数。

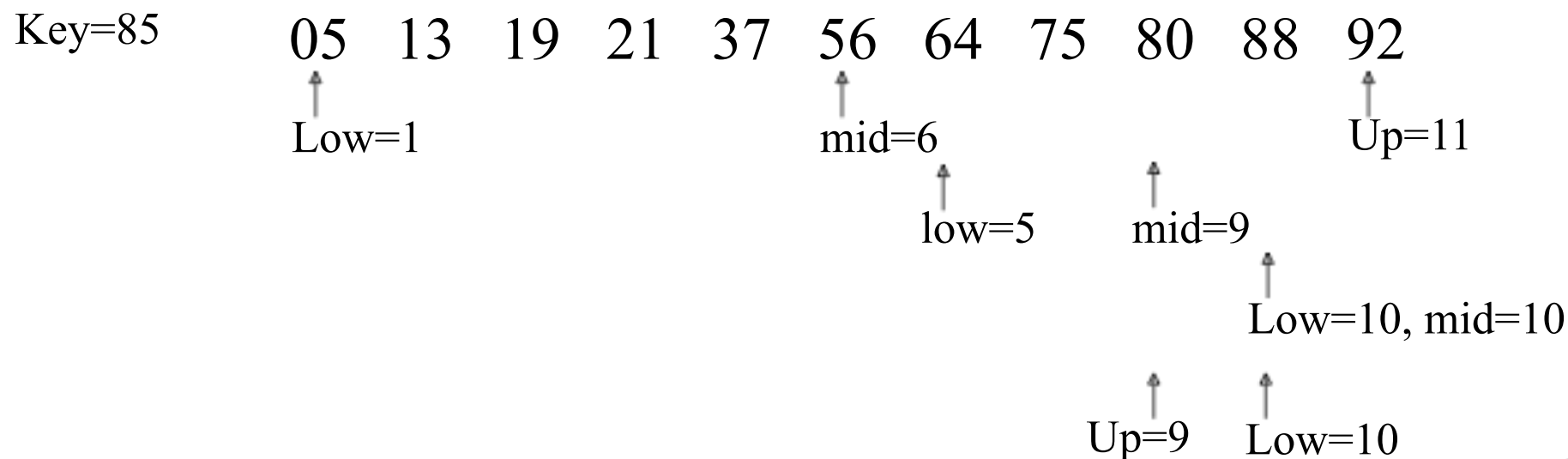
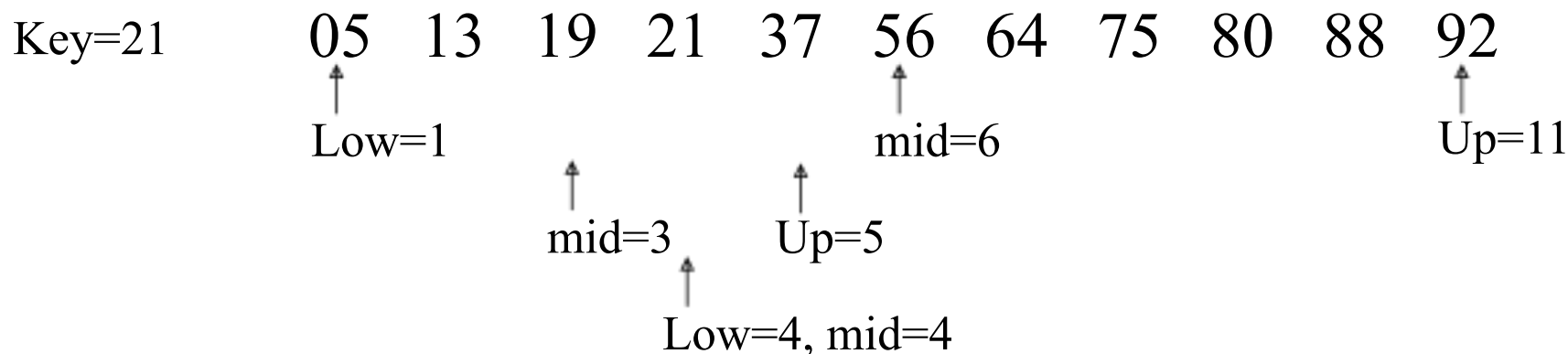
$$ASL = \sum_{i=1}^n P_i \cdot C_i$$

在等概率情况下 $P_i=1/n$ ， 且一般情况下 $C_i = n - i + 1$

$$\begin{aligned} ASL &= nP_1 + (n-1)P_2 + \dots + 2P_{n-1} + P_n \\ &= 1/n \cdot \sum_{i=1}^n (n - i + 1) \\ &= (n+1)/2 \end{aligned}$$

5.2 折半查找

查找表=[05,13,19,21,37,56,64,75,80,88,92]



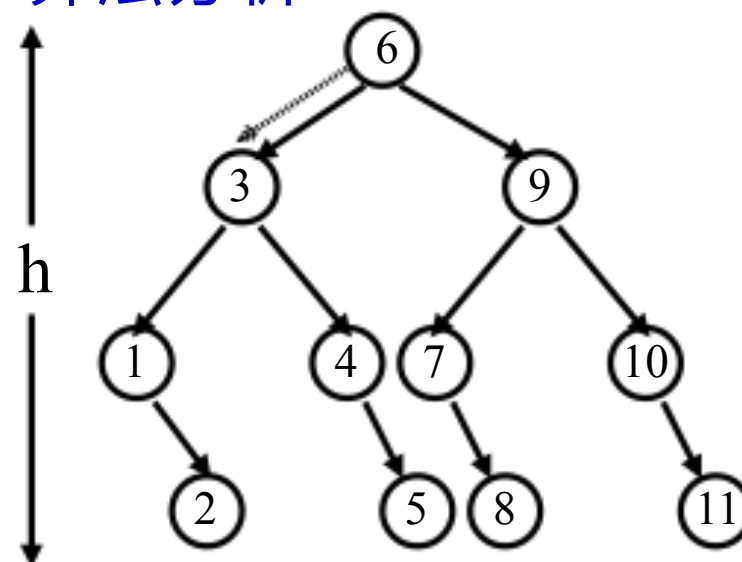
折半查找算法

```

Int Binary-Search(keytype k, LIST F )
{
    int low , up , mid ;
    low = 1 ; up = last ;
    while ( low <= up )
        {
            mid = ( low + up ) / 2 ;
            if ( F[mid].key == k )
                return mid ;
            else if ( F[mid].key > k )
                up = mid - 1 ;
            else
                low = mid + 1 ;
        }
    return -1
}
    
```

$O(\log_2 n)$

算法分析



$$n = 2^h - 1 \quad (h = \log_2(n+1))$$

$$ASL_{bs} = \sum_{i=1}^n P_i \cdot C_i$$

$$P_i = 1/n$$

$$= \frac{1}{n} \cdot \sum_{j=1}^h j \cdot 2^{j-1}$$

$$= \frac{(n+1)/n \cdot \log_2(n+1) - 1}{1}$$

平均查找长度


```
int Bsearch( F , i , j , k )  
{  int m;  
   if(i>j) return( -1 );  
   else {  
       m=( i + j ) / 2;  
       if( F[m].key == k )  
           return m;  
       if( F[m].key < k )  
           return( Bsearch( F , i , m-1 , k ) );  
       else  
           return( Bsearch( F , m+1 , j , k ) );  
   }  
}
```

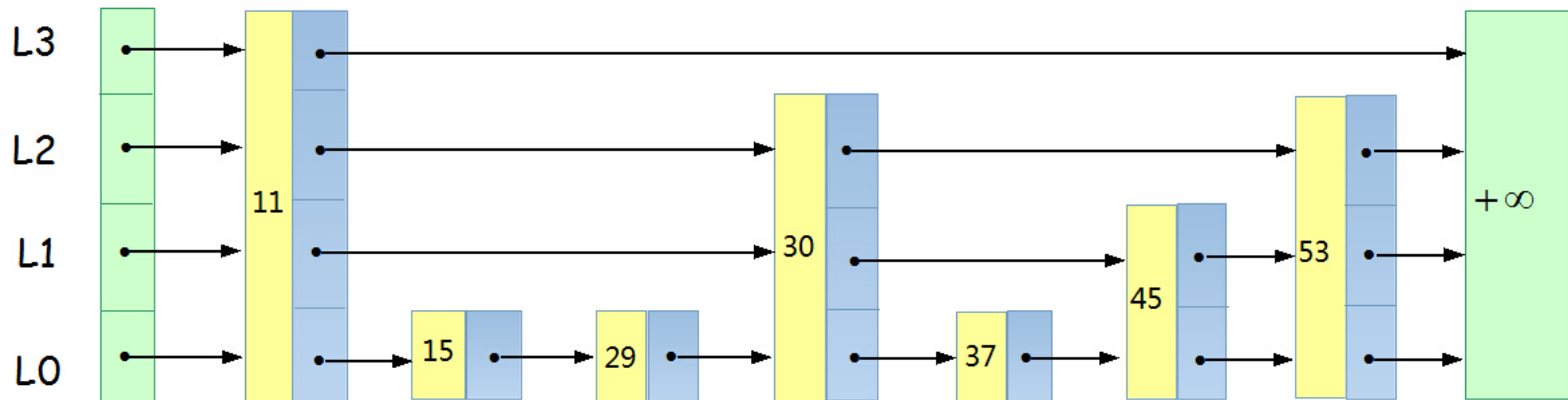
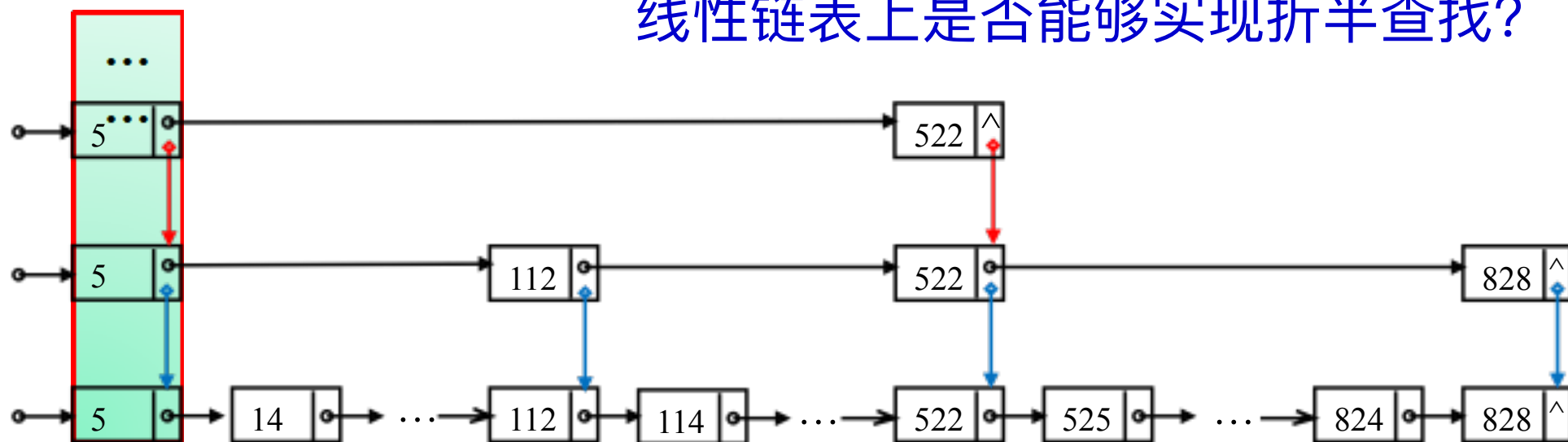
//折半查找的递归算法

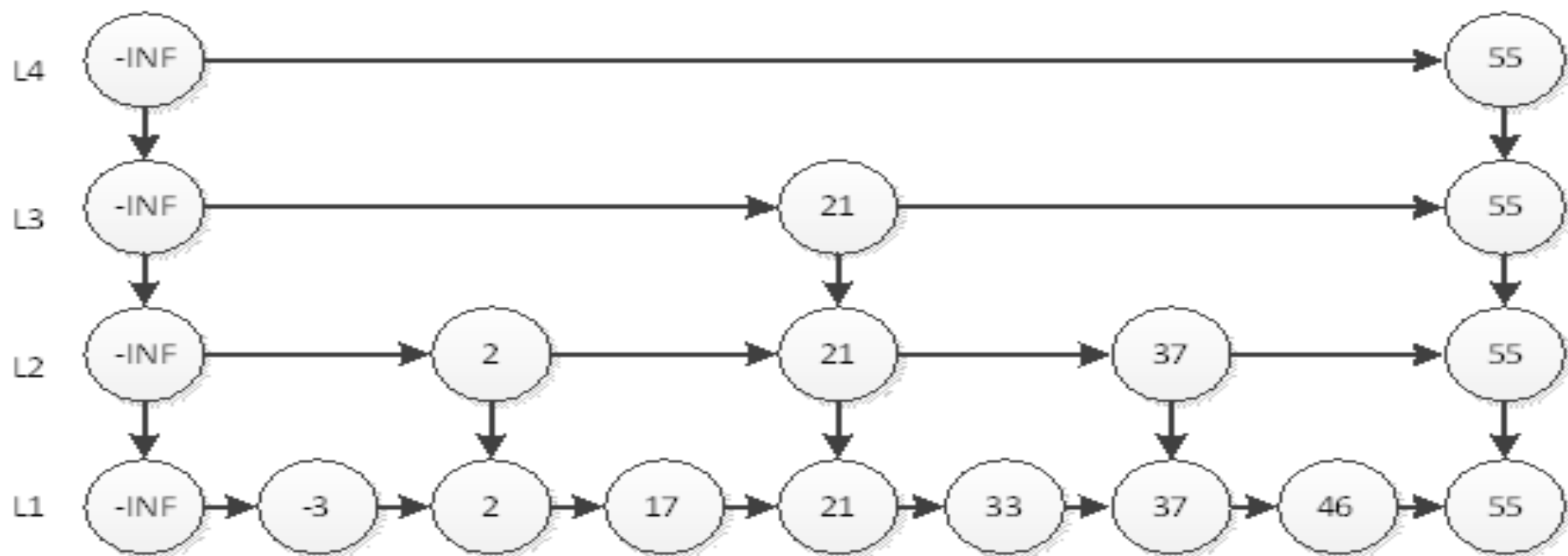
调用： Bsearch(F,1,n,k)

思考题：

1. 已知由 n 个整数构成序列的数据分布为先下降/上升再上升/下降，即一开始数据是严格递减/递增的，后来数据是严格递增/递减的，设计尽可能高效算法，找到序列中的最小/大值。
2. 在给定的一个已经排好序的数组中，找出指定数字出现的次数；例如数组 $[1, 2, 3, 3, 3, 4, 5]$ 中3出现的次数为3次。
3. 已知按升序排列的数组，求与给定值 $target$ 相同的最后一个/第一个元素位置。
4. 在一个有序数组中只有一个元素出现一次，其余元素均出现2次，找出这个元素。
5. 求一个数 num 的算术平方根 \sqrt{num} ，一个数 num 的算术平方根 \sqrt{num} 一定在 $0 \sim \sqrt{num}/2$ 之间，并且满足 $\sqrt{num} = num / \sqrt{num}$ ，可以利

线性链表上是否能够实现折半查找？





Skip List
 又称跳跃表
 简称跳表

- redis和levelDB都是用了它；
- 他在有序链表的基础上进行扩展；
- 解决了有序链表结构查找特定值困难的问题；
- 一种可以与二分查找相比的链式数据结构；
- 查找特定值的时间复杂度为 $O(\log n)$ ；

5.3 分块查找 (线性查找+折半查找)

	0	1	2												
I	22	44	74	索引表											
X	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	22	12	13	9	8	33	42	44	38	24	48	60	58	74	47

Int index_search(k, last, blocks, ix, F,L)

Keytype k ; int last,blocks ; index ix LIST F ;int L ;

{ int i, j ;

i = 0;

while ((k > ix[i])&&(i < blocks)) i++ ;

if(i<blocks)

{ j = i*L;

while((k != F[j].key)&&(j <= (i+1)*L-1)&&(j < last))

j = j + 1 ;

if (k == F[j].key) return j ;

}

return -1 ;

}

Typedef keytype index[maxblock]

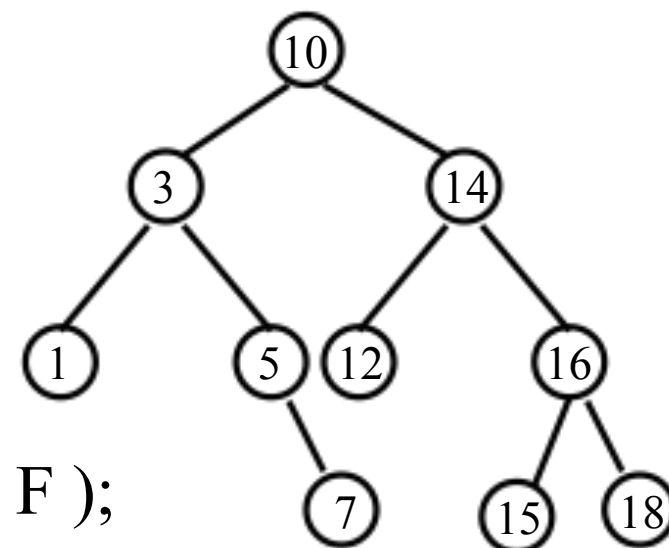
5.4 二叉查找树(二叉排序树)

```
Struct CellType {
    records data ;
    CellType *lchild,*rchild ;}
Typedef CellType * BST ;
```

```
BST search( keytype k, BST F );
```

1、查找

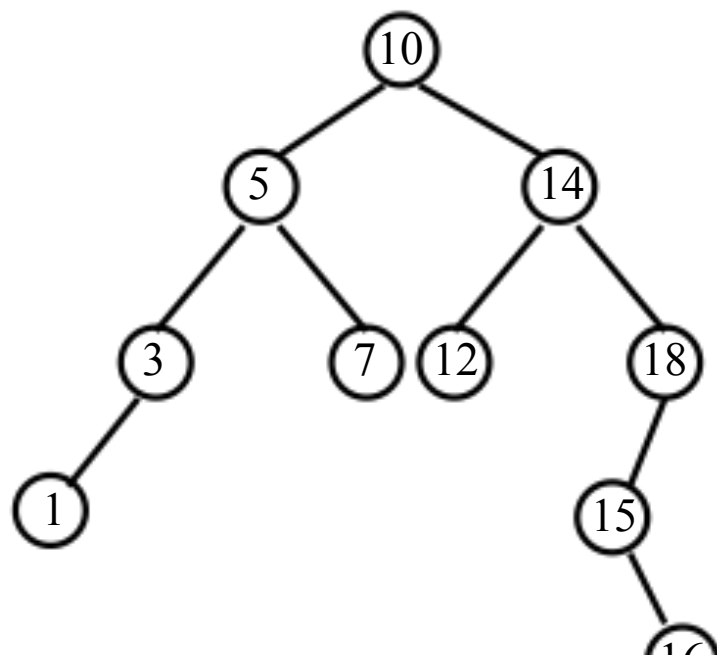
```
{ p = F ;
  if ( p == NULL )
    return Null ;
  else if ( k == p->data.key )
    return p ;
  else if ( K < p->data.key )
    return ( search ( k, p->lchild ) ) ;
  else if ( K > p->data.key )
    return ( search ( k, p->rchild ) ) ;
}
```



2、在二叉查找树中插入新结点

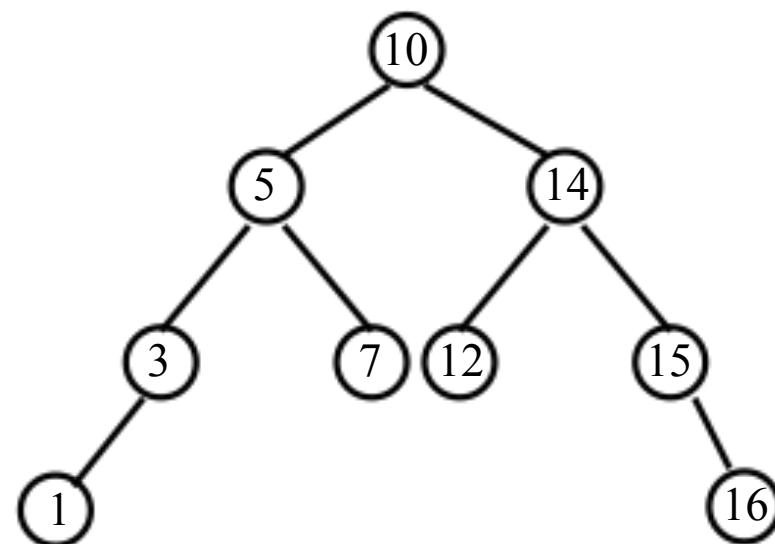
```
Void Insert ( Records R , BST &F )  
{  
    if ( F ==NULL )  
        { F = new CellType ;  
          F->data = R ;  
          F->lchild = NULL ;  
          F->rchild = NULL ; }  
    else if ( R.key < F->data.key )  
        Insert ( R , F->lchild )  
    else if ( R.key >= F->data.key )  
        Insert ( R , F->rchild )  
}
```

3、在二叉查找树中删除结点

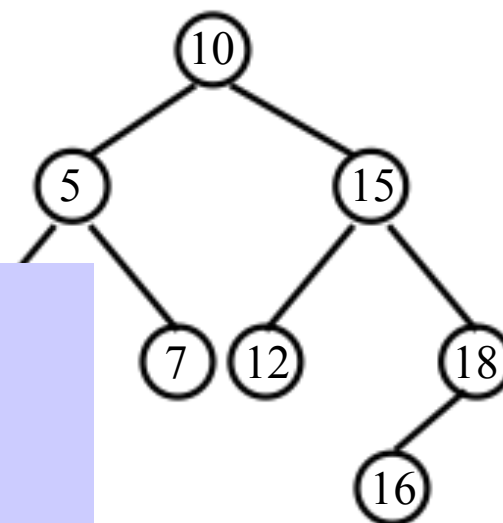


删除三类结点：

- 1、被删除的结点是叶子结点
- 2、被删除的结点只有一颗非空的左子树或右子树
- 3、被删除的结点有两棵非空的子树



删除18之后



删除14之后

在二叉查找树中删除结点
在F所指的非空二叉树中
删除关键字最小的结点，
返回该结点中的数据。

```
Records DeleteMin( F )
{
    records tmp ; BST p ;
    if ( F->lchild == NULL )
    {
        p = F ;
        tmp = F->data ;
        F = F->rchild ;
        Delete p ;
        return tmp ;
    }
    else
    return ( DeleteMin( F->lchild ) ;
}
```

```
Void Delete ( keytype k ,BST &F )
{ 用右子树中最左边的结点作为继承结点
    if ( F != NULL )
        if ( k < F->data.key )
            Delete( k, f->lchild ) ;
        else if ( k > F->data.key )
            Delete( k, f->rchild ) ;
        else
            if ( F->rchild == NULL )
                F = F->lchild ;
            else if( F->lchild == NULL )
                F = F->rchild ;
            else
                F->data =DeleteMin(F->rchild)
```

【例5-1】 二叉树用二叉链表表示，且每个结点的键值互不相同，编写算法判别该二叉树是否为二叉排序树。

```
int Judge(BTree *bt)
{   BTree *s[100],*p=bt;
    int top=0,preval=min;
    do{   while(p){   s[top++]=p;
                    p=p->lchild;   }
        if(top>0){
            p=s[--top];
            if(p->data<preval) return 0; //不是二叉排序树
            preval=p->data;
            p=p->rchild;   }
        }while (p||top>0);
    return(1) ;   //是二叉排序树
}
```

5.5 AVL树, 又称平衡二叉树 (Balanced Binary Tree)

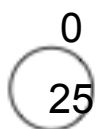
G.M. Adelson-Velsky 和 E.M. Landis, 首次出现在1962年论文:
《An algorithm for the organization of information》

格奥尔吉·阿杰尔松-韦利斯基, 前苏联数学家, 计算机科学家。1922年出生于俄罗斯萨马拉。1962年与叶夫吉尼·兰迪斯发明了AVL树。后移居以色列, 现居住于阿什杜德。

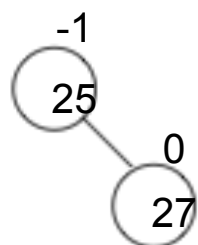
【定义】 AVL树或者是一棵空二叉树, 或者具有如下性质的二叉查找树:
其左子树和右子树都是高度平衡的二叉树, 且左子树和右子树高度之差的绝对值不超过1。
结点的平衡因子 BF (Balanced Factor) 定义为: 结点左子树与右子树的高度之差。

AVL中的任意结点的 BF 只可能是 -1, 0, +1

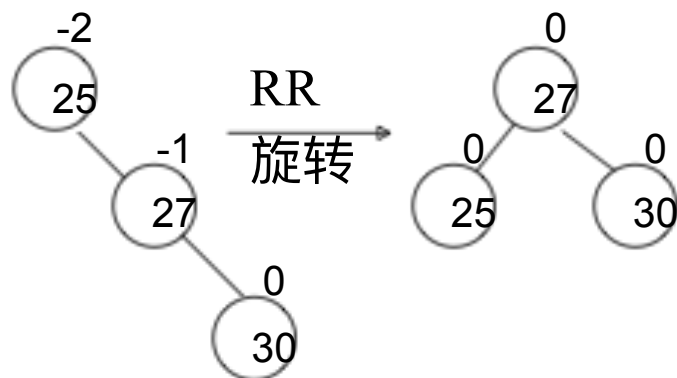
输入={25, 27, 30, 12, 11, 18, 14, 20, 15, 22}



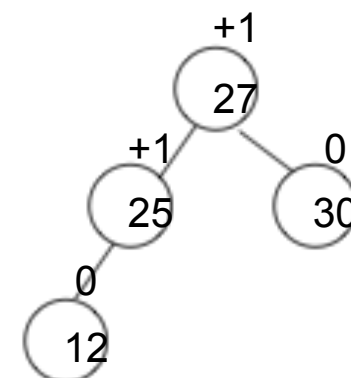
(a)插入25



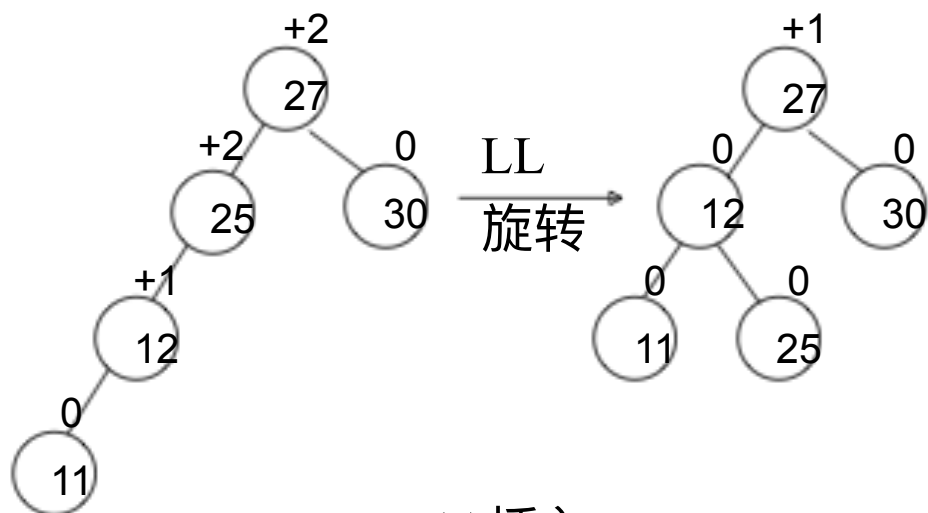
(b)插入27



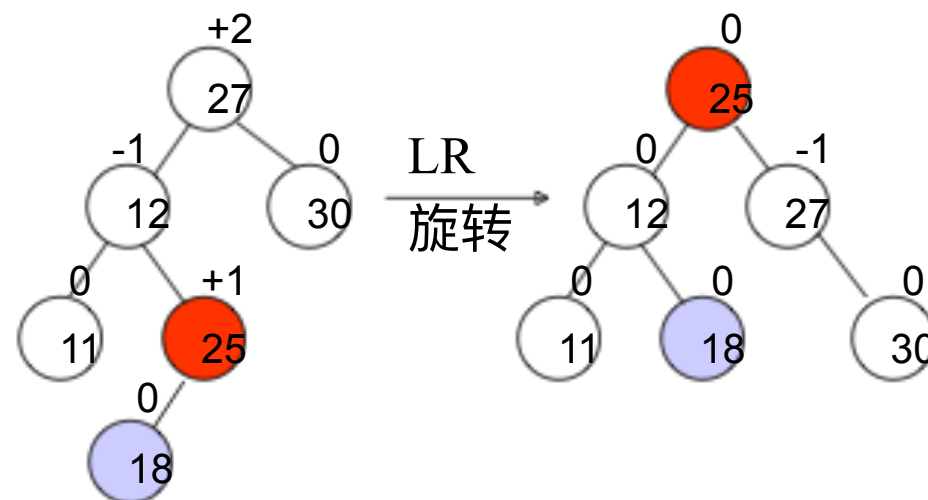
(c)插入30



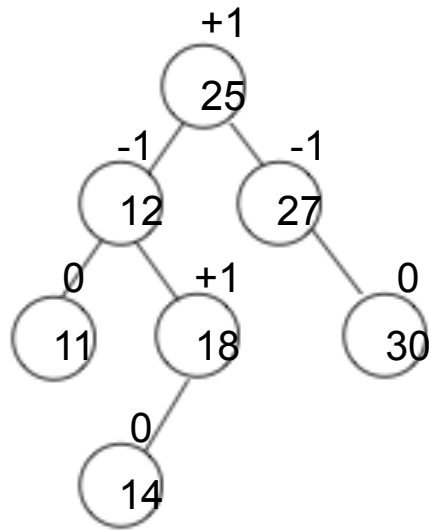
(d)插入12



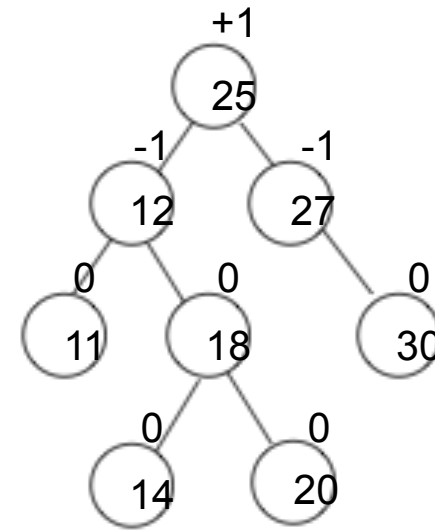
(e)插入11



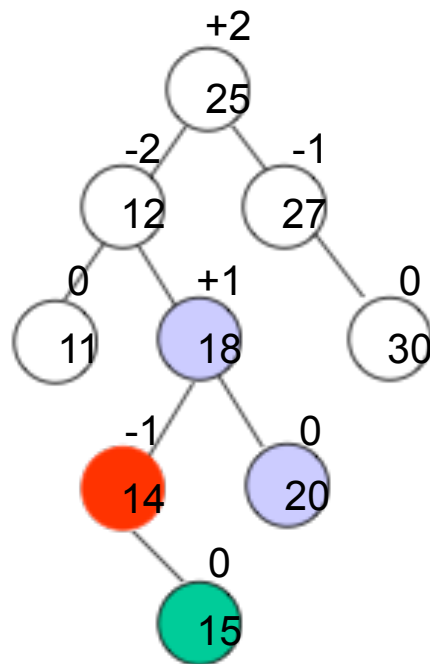
(f)插入18



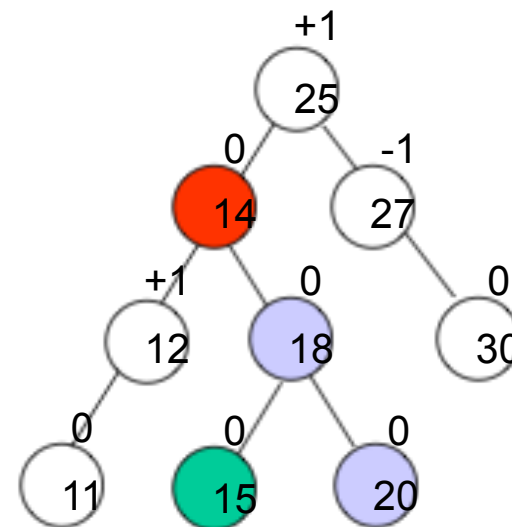
(g)插入14



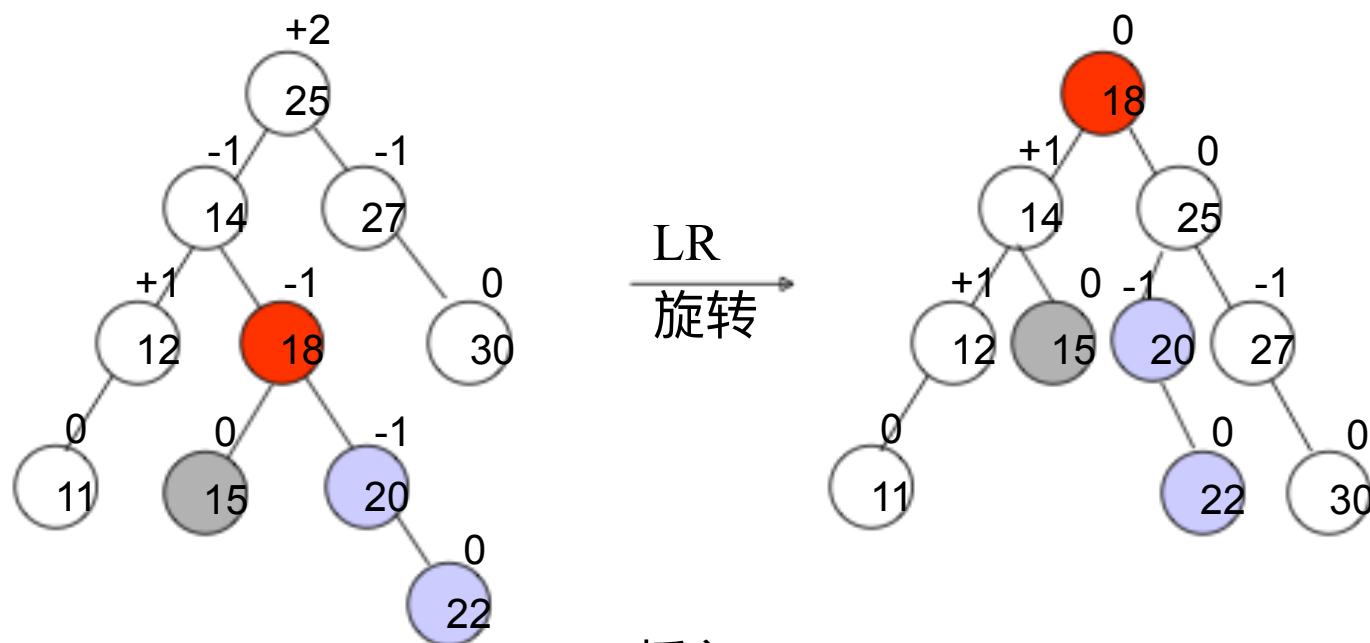
(h)插入20



RL
旋转



(i)插入15

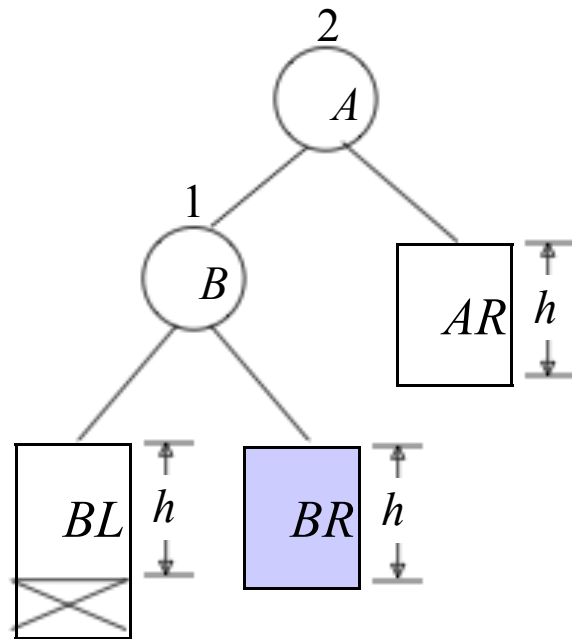


(i)插入22

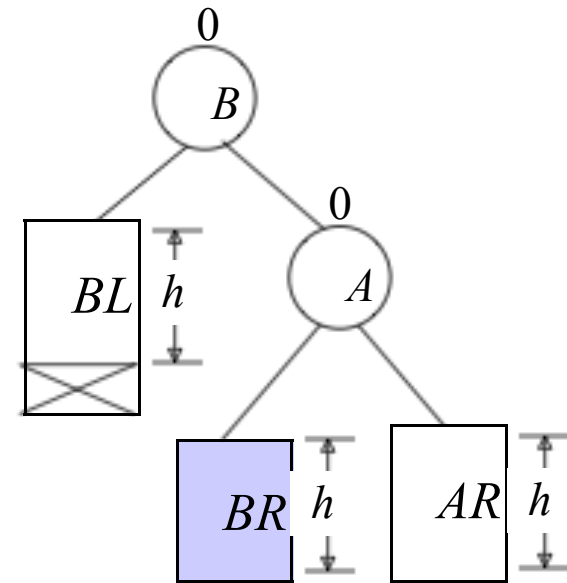
设：Y表示新插入的结点，

A表示离新插入结点Y最近的且平衡因子变为 ± 2 的祖先结点

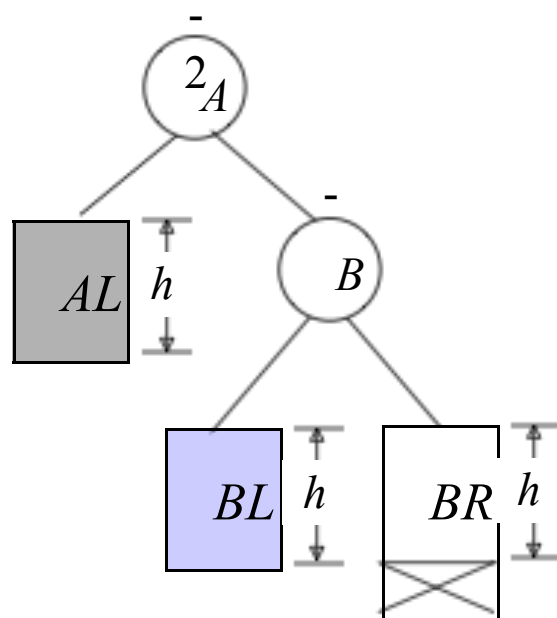
- 对称
- LL：新结点Y被插入到A的左子树的左子树上；
 - LR：新结点Y被插入到A的左子树的右子树上；
 - RR：新结点Y被插入到A的右子树的右子树上；
 - RL：新结点Y被插入到A的右子树的左子树上；



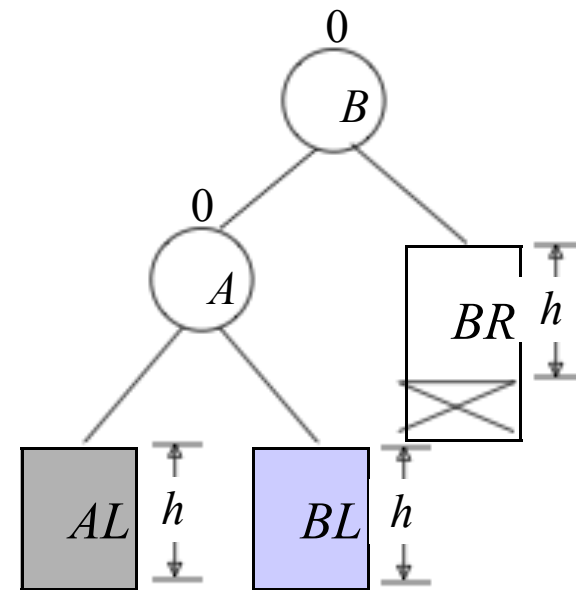
LL型



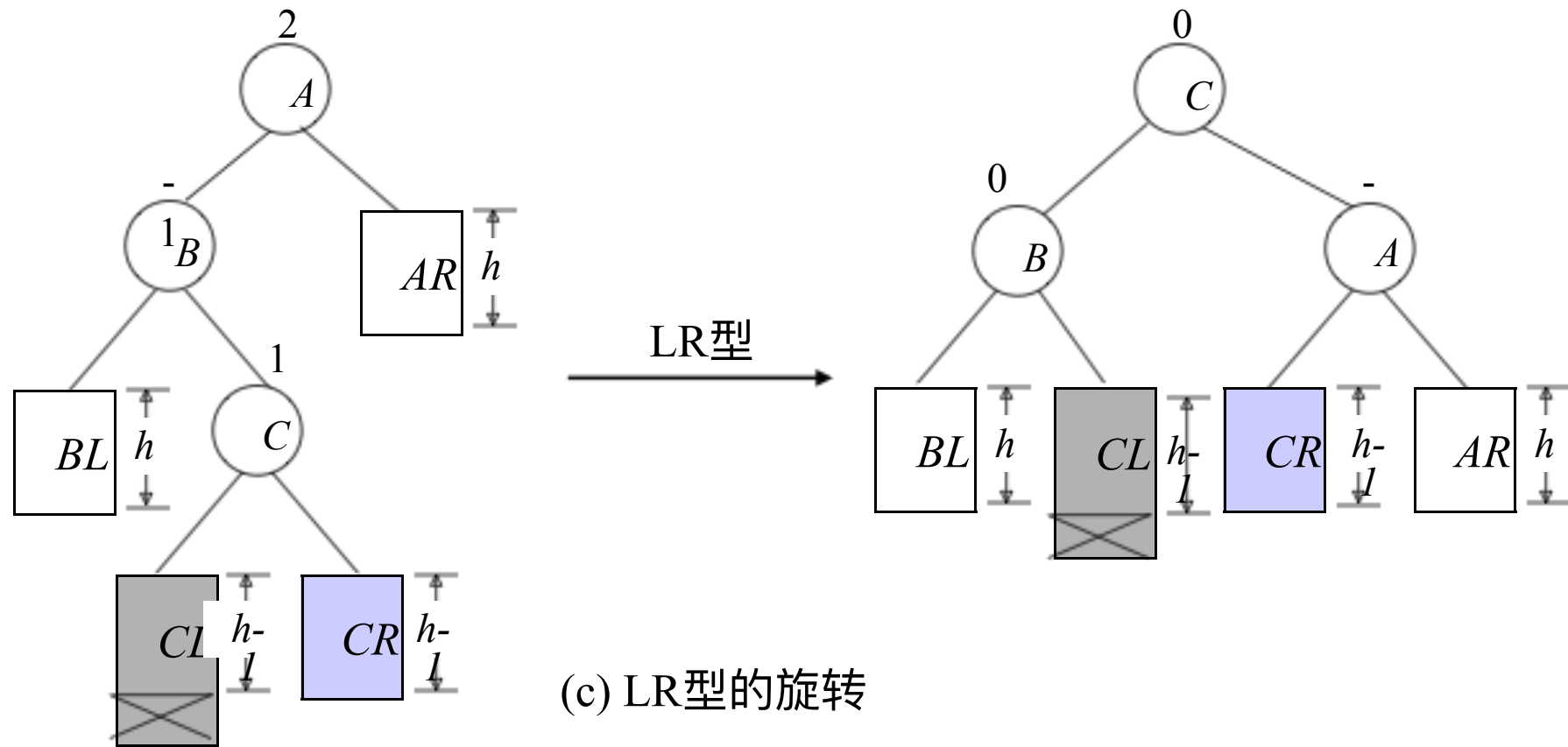
(a) LL型的旋转

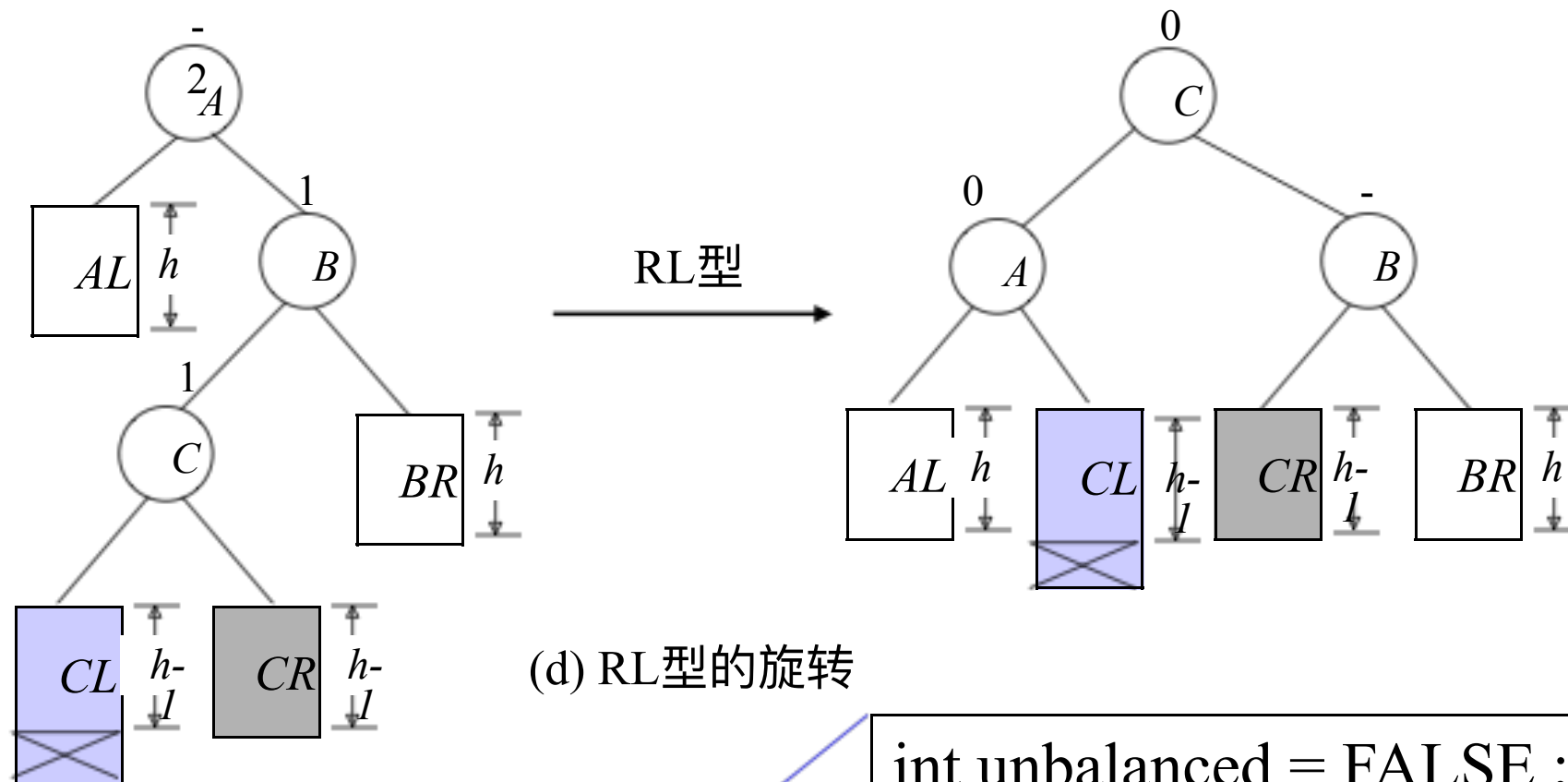


RR型



(b) RR型的旋转





(d) RL型的旋转

AVL 树的结构类型

```
int unbalanced = FALSE ;
struct Node {
    ElementType data ;
    int bf ;
    struct Node *lchild, *rchild ;
}
typedef Node *AVLT ;
```

unbalanced = TRUE;

```
void AVLInsert ( AVLTree &T, ElementType R, int &unbalanced )
{ if( !T ) //向空二叉树中插入元素
  { unbalanced = TRUE ;
    T = new Node ;
    T->data = R ;
    T->lchild = T->rchild = NULL ;
    T->bf = 0;
  }
  else if ( R.key < T->data.key ) //在左子树上插入
  { AVLInser( T->lchild, R, unbalanced );
    if ( unbalanced )
      switch ( T->bf ) {
        case -1: T->bf = 0 ;
                    unbalanced = FALSE ;
                    break ;
        case 0:  T->bf = 1 ;
                    break ;
        case 1:  LeftRotation ( T, unbalanced ); }
  }
}
```

```

else if ( R.key >=T->data.key )    //在右子树上插入
{  AVLInsert ( T->rchild , R, unbalanced ) ;
    if ( unbalanced )
        switch ( T->bf ) {
            case 1: T->bf = 0 ;
                    unbalanced = FALSE ;
                    break ;
            case 0: T->bf = -1 ;
                    break ;
            case -1: RightRotation ( T, unbalanced ) ;
        }
    }
else
    unbalanced = FALSE ;
} //AVLInsert
    
```

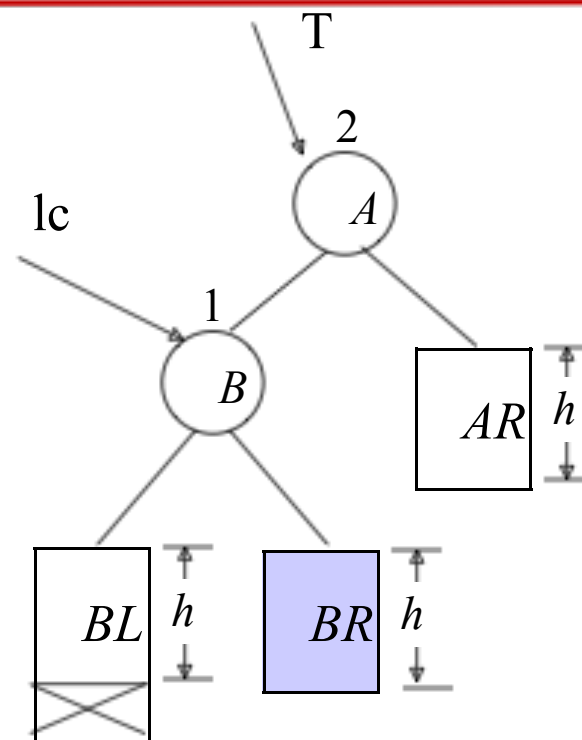
```
void LeftRotation ( AVL &T , int &unbalanced )
```

```
{
    AVLT gc , lc ;
    lc = T->lchild ;
    if ( lc->bf == 1 ) //LL旋转
    {
        T->lchild = lc->rchild ;
        lc->rchild = T ;
        T->bf = 0 ;
        T = lc
    }
    else //LR旋转
    {
        gc = lc->rchild ;
        lc->rchild = gc->lchild ;
        gc->lchild = lc ;
        T->lchild = gc->rchild ;
        gc->rchild = T ;
```

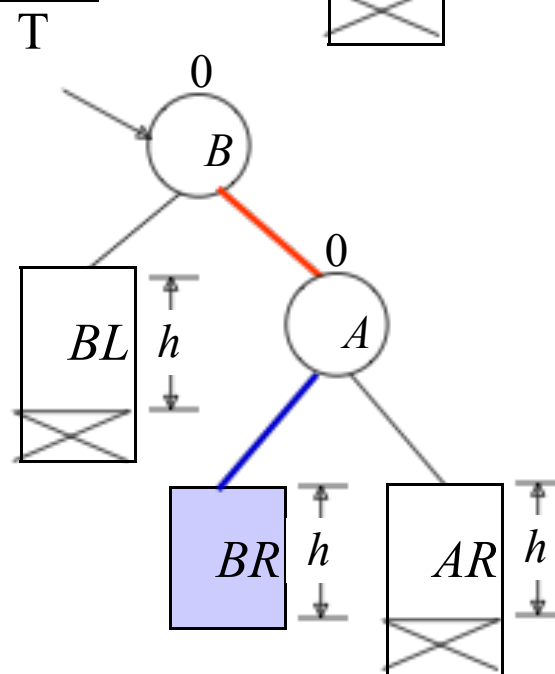
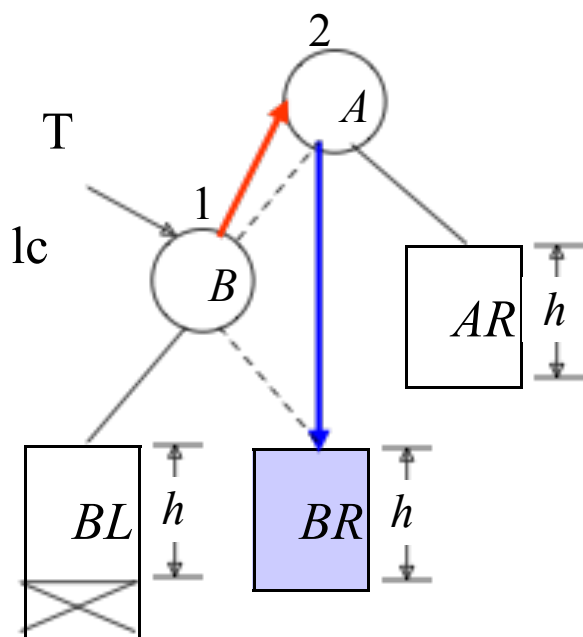
```
        switch ( gc->bf ) {
            case 1: T->bf = -1 ;
                    lc->bf = 0 ;
                    break ;
            case 0: T->bf = lc->bf = 0 ;
                    break ;
            case -1: T->bf = 0 ;
                    lc->bf = 1 ;

        }
        T = gc ;
    }
    T->bf = 0 ;
    unbalanced = FALSE ;
}
```

```
lc = T->lchild ;
if ( lc->bf == 1 ) //LL旋转
{
    T->lchild = lc->rchild ;
    lc->rchild = T ;
    T->bf = 0 ;
    T = lc
}
...
```



(a) LL型的旋转



```
lc = T->lchild;
```

.....

```
else //LR旋转
```

```
{ gc = lc->rchild;
```

```
  lc->rchild = gc->lchild;
```

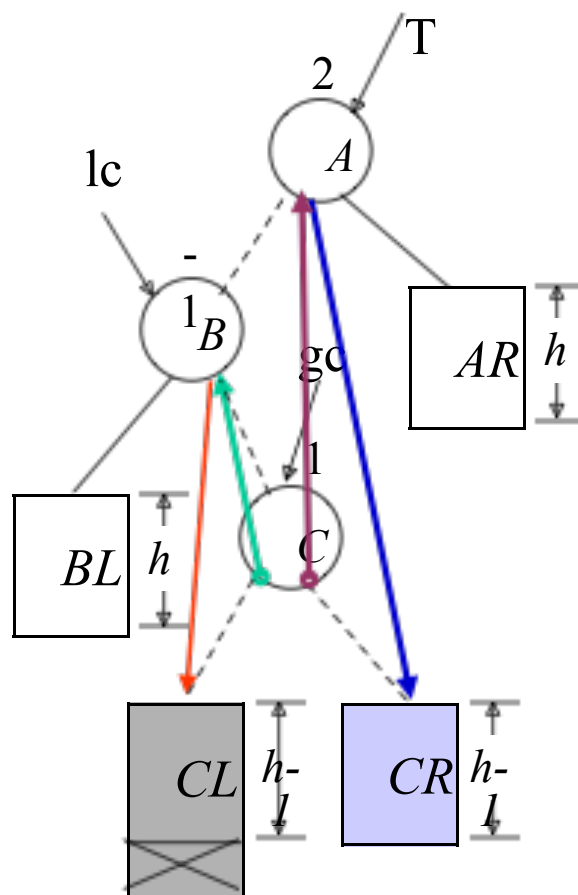
```
  gc->lchild = lc;
```

```
  T->lchild = gc->rchild;
```

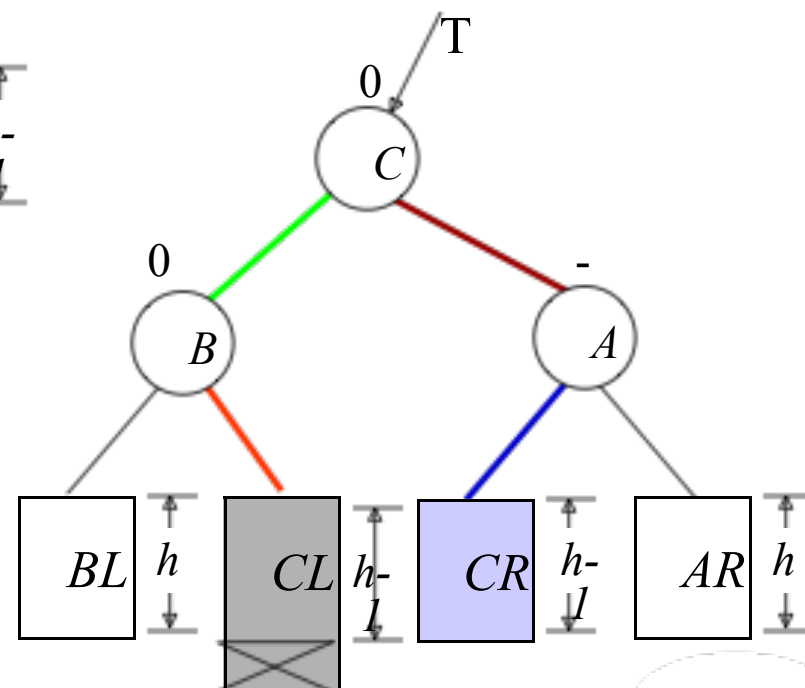
```
  gc->rchild = T;
```

.....

```
T = gc;
```



(c) LR型的旋转



```
void RightRotation(AVLTree *&T)
{
    AVLTree *gc,*rc ;
    rc=T->rchild ;
    if(rc->bf==-1)           //RR旋转
    {
        T->rchild=rc->lchild ;
        rc->lchild=T ;
        T->bf=0;
        T=rc;
    }
    else                     //RL旋转
    {
        gc=rc->lchild ;
        rc->lchild=gc->rchild ;
        gc->rchild=rc ;
        T->rchild=gc->lchild ;
        gc->lchild=T;
```

```
switch(gc->bf){    //调整平衡因子
    case -1: T->bf=-1;
               rc->bf=0 ;
               break ;
    case 0: T->bf=rc->bf=0 ;
             break ;
    case 1: T->bf=0;
             rc->bf=-1;
}
T=gc ;
}
T->bf=0 ;
unbalanced=FALSE ;
}
```

分析:

- 高度为h的AVL树最多具有 $N_{hmax}=2^h-1$ 个结点(二叉树性质一)
- 高度为h的AVL树最少具有多少个结点?

$$N_0=1$$

$$N_1=1$$

$$N_2=2$$

.....

$$N_h=N_{h-1}+N_{h-2}+1$$

$$F_i=F_{i-1}+F_{i-2} \quad (\text{Fibonacci polynomial})$$

$$N_{hmi} \quad \log_{\Phi}(\sqrt{5} * (N + 1)) - 2$$

$$n=$$

$$? \quad h \leq \log_{\Phi}(\sqrt{5}(n+1)) - 2$$

$$T(n)=O(\log n)$$

【例5-2】 下述二叉树中,哪一种满足性质:从任一结点出发到根的路径上所经过的结点序列按其关键字有序。

(A) 二叉排序树 (B) 赫夫曼树 (C) AVL树 (D) 堆

【分析】

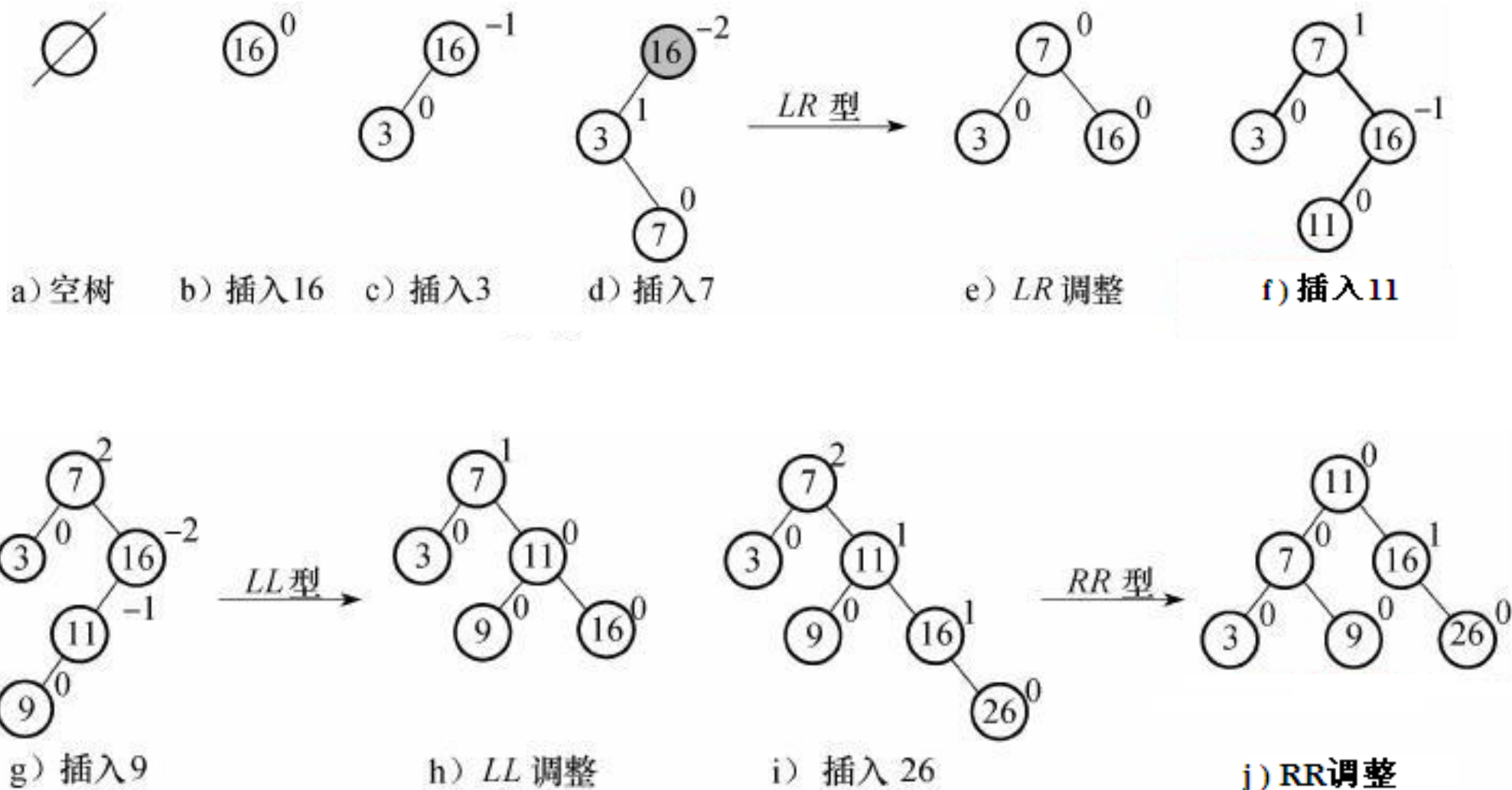
➤对于选项A, 根据二叉排序树的结构特点我们可以知道, 二叉排序树的中序遍历结果是一个有序序列, 而在中序遍历中, 父结点并不总是出现在孩子结点的前面 (或后面), 故该选项不正确。

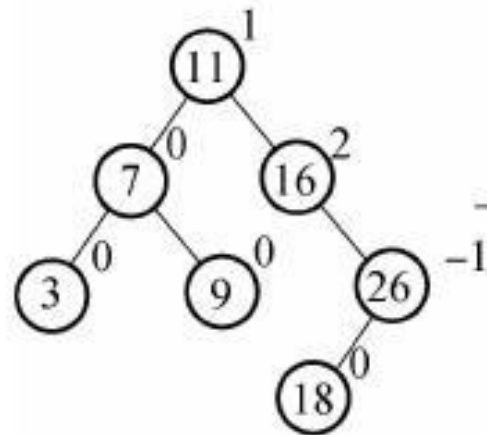
➤对于选项B, 根据赫夫曼树的结构特点我们可以知道, 在赫夫曼树中所有的关键字只出现在叶结点上, 其非叶结点上并没有关键字值, 显然不正确。

➤对于选项C, AVL树其本质上也是一种二叉排序树, 只不过是平衡化之后的二叉排序树, 故该选项也是不正确的。

➤对于选项D, 堆的概念我们会在堆排序中给大家介绍, 根据建堆的过程, 不断地把大者“上浮”, 将小者“筛选”下去, 最终得到的正是一个从任一结点出发到根的路径上所经过的结点序列按其关键字有序的树状结构。

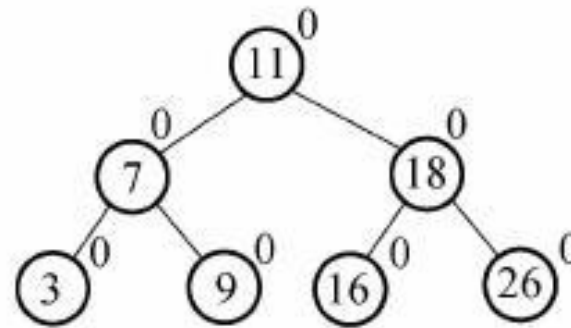
【例5-3】 输入关键码序列为(16, 3, 7, 11, 9, 26, 18, 14, 15),
据此建立平衡二叉树, 给出插入和调整的具体过程。



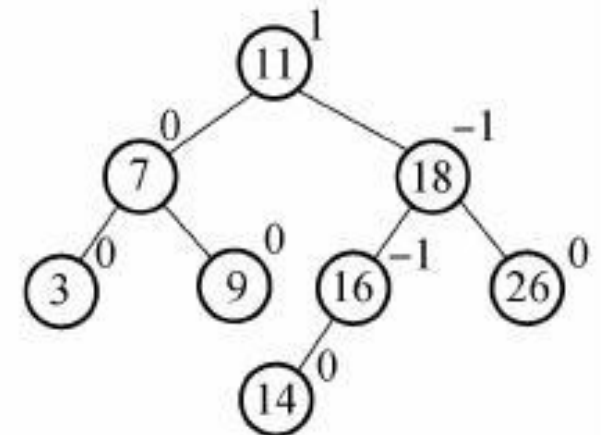


k) 插入18

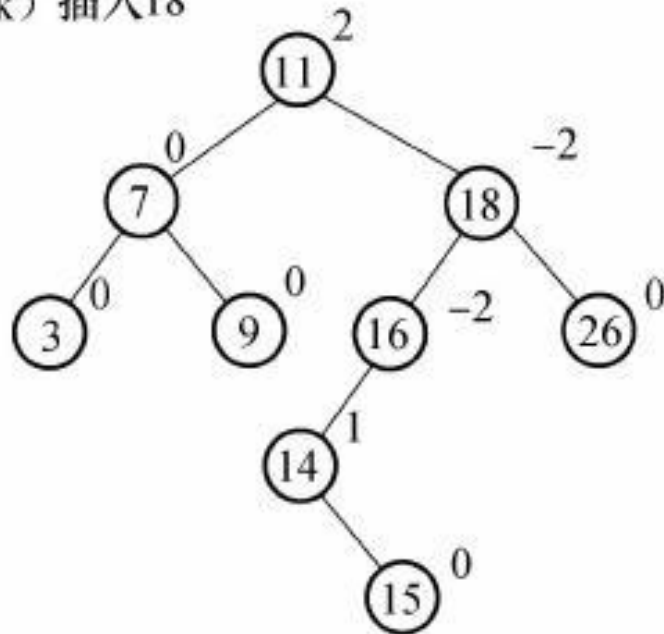
$\xrightarrow{RL \text{ 型}}$



l) RL 调整

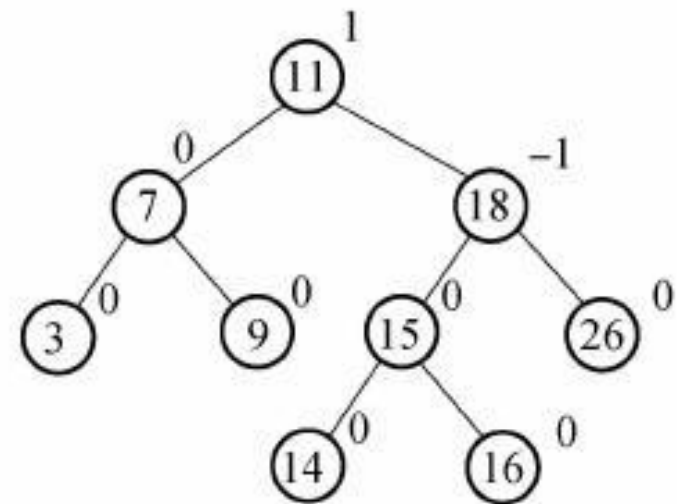


m) 插入14



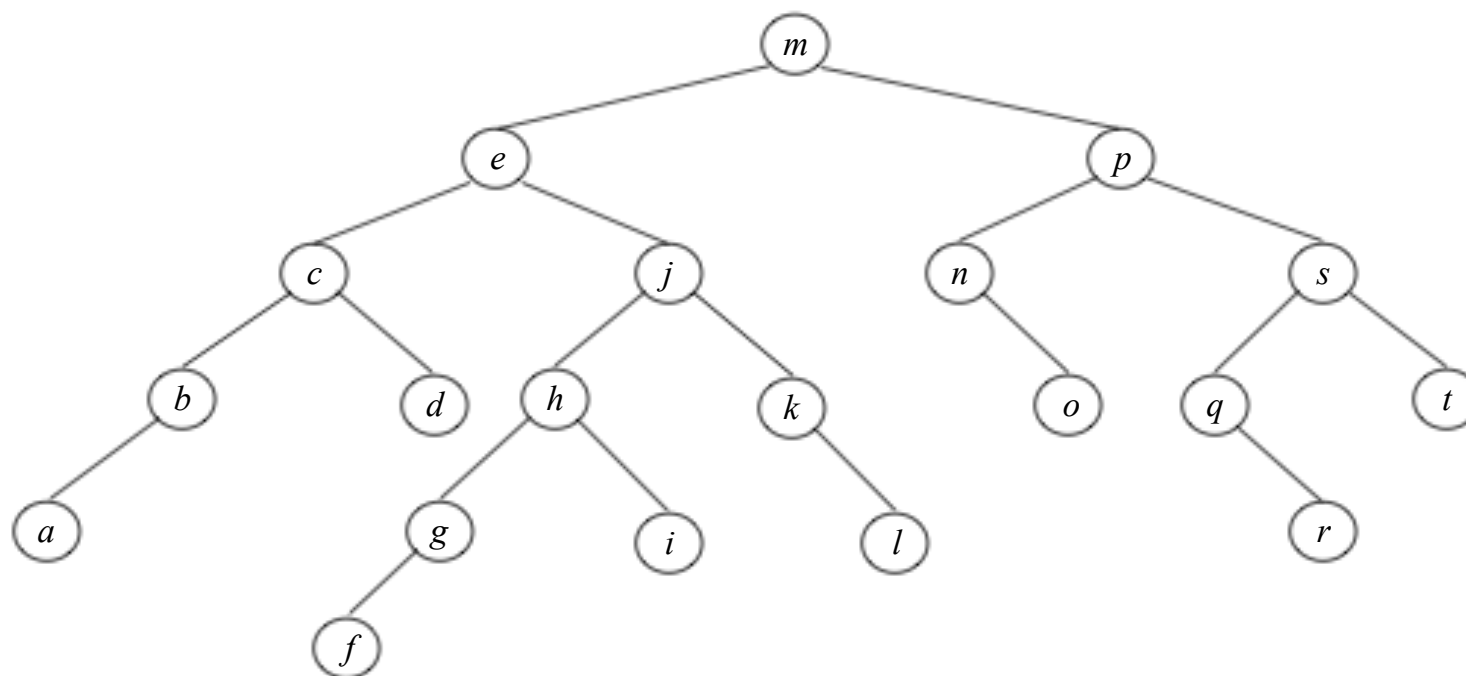
n) 插入 15

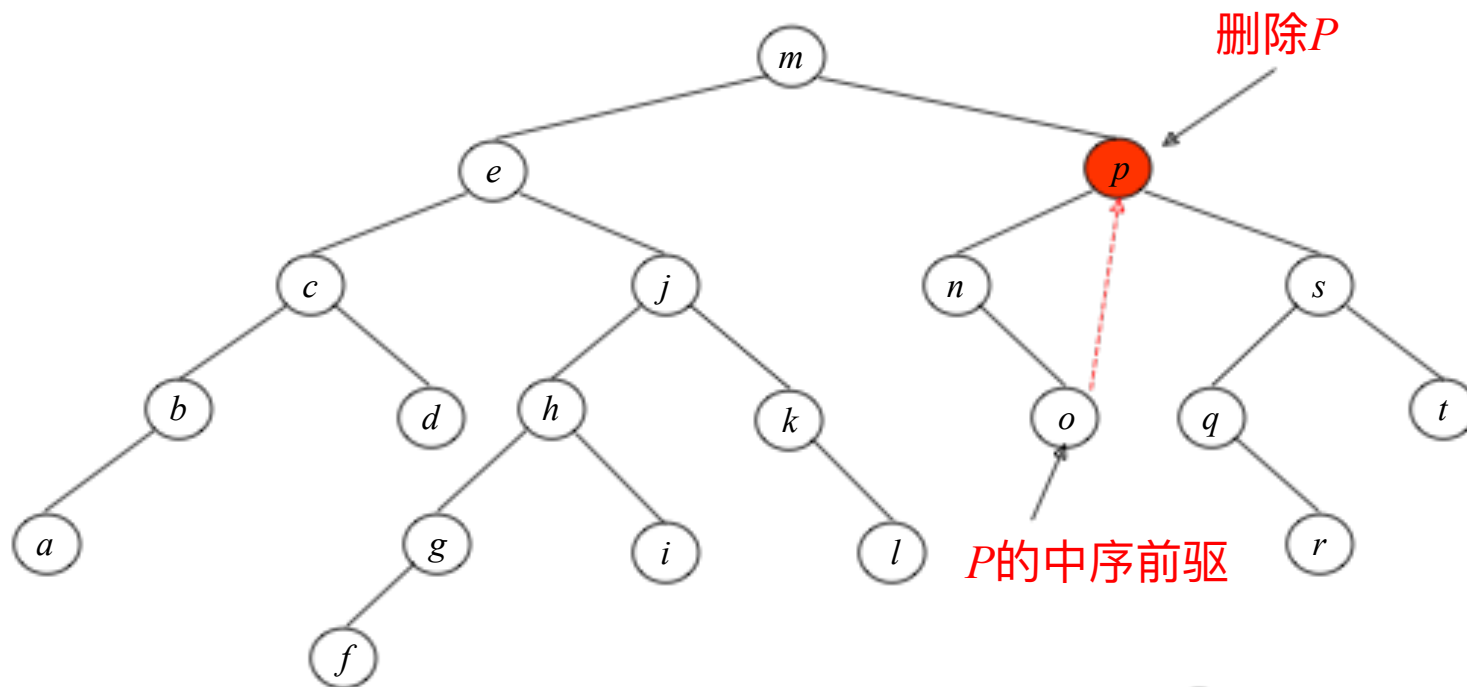
$\xrightarrow{LR \text{ 型}}$



o) LR 调整

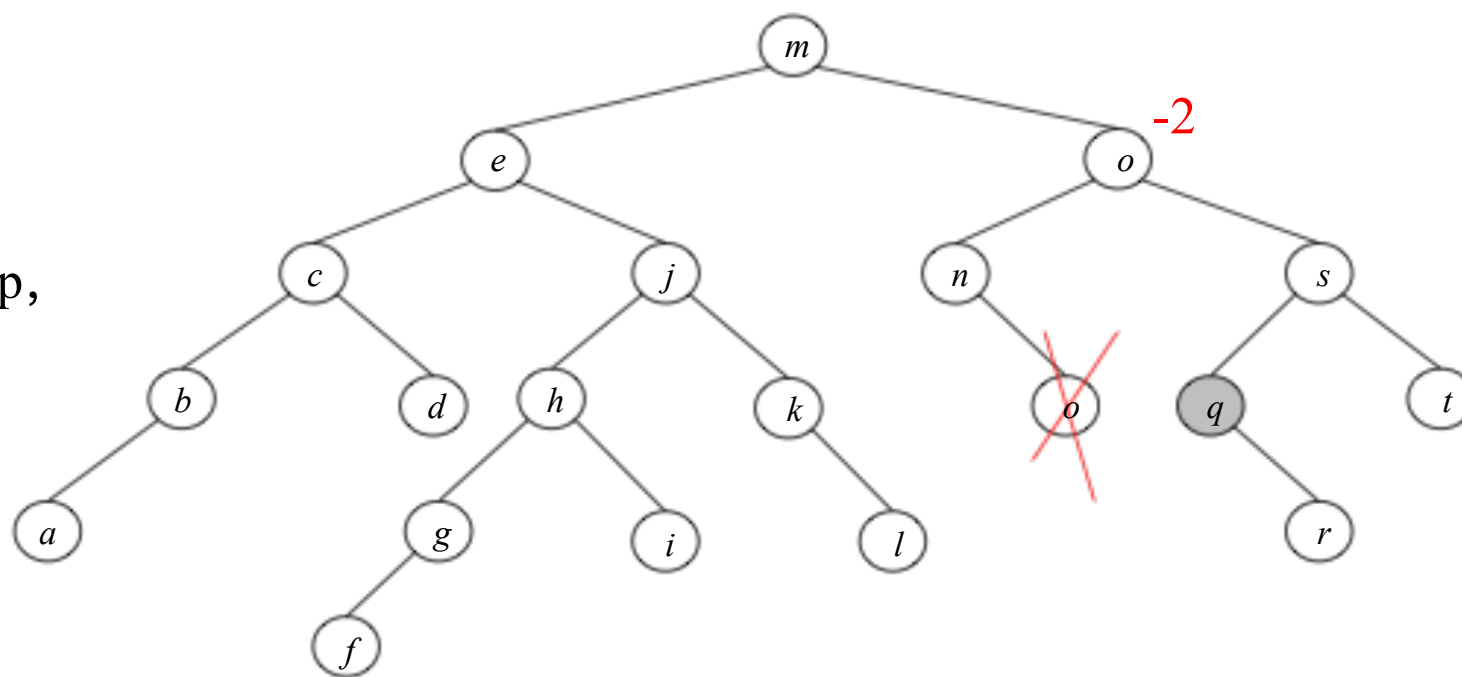
【例5-4】 有一棵平衡二叉树的初始状态如图所示，
请给出删除图中结点p后经调整得到的新的平衡二叉树。

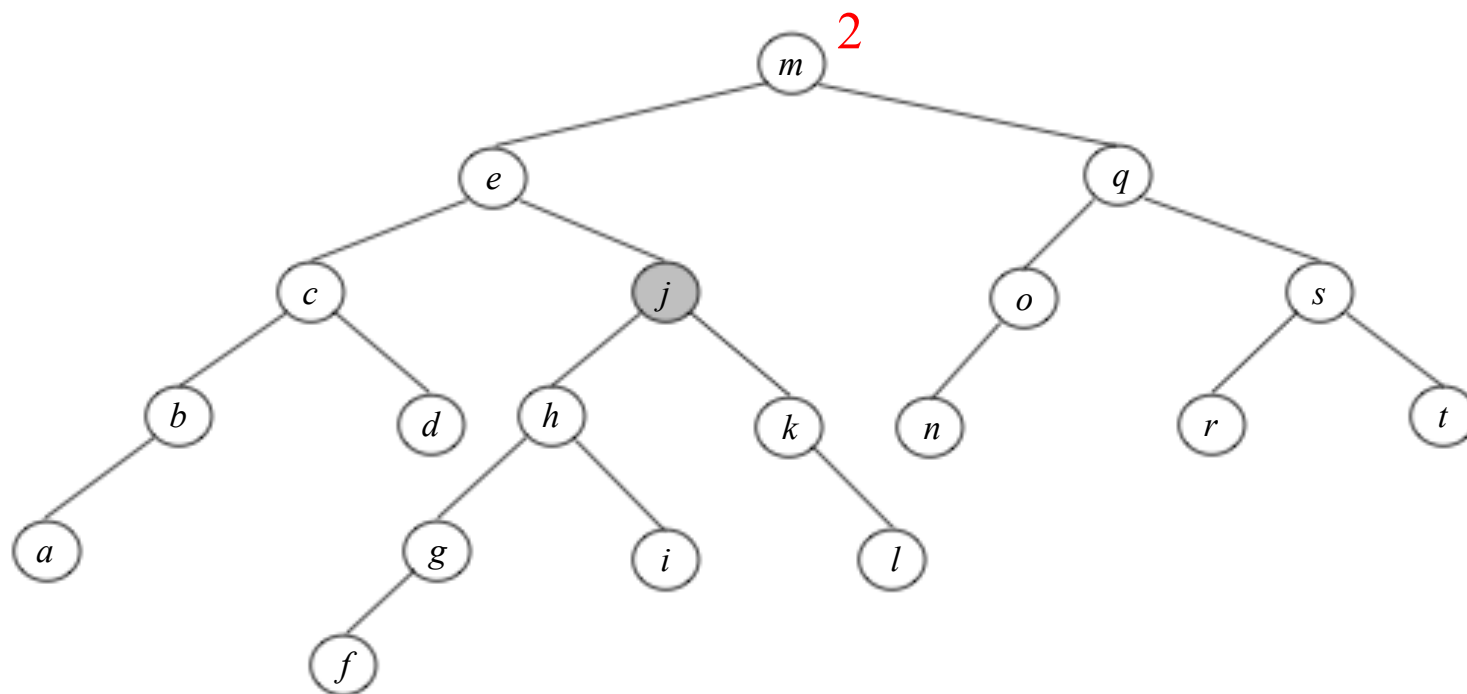




由于结点p既有左孩子，又有右孩子，故在删除结点p的时候应该先找到中序遍历中结点p的直接前驱结点，即结点o。

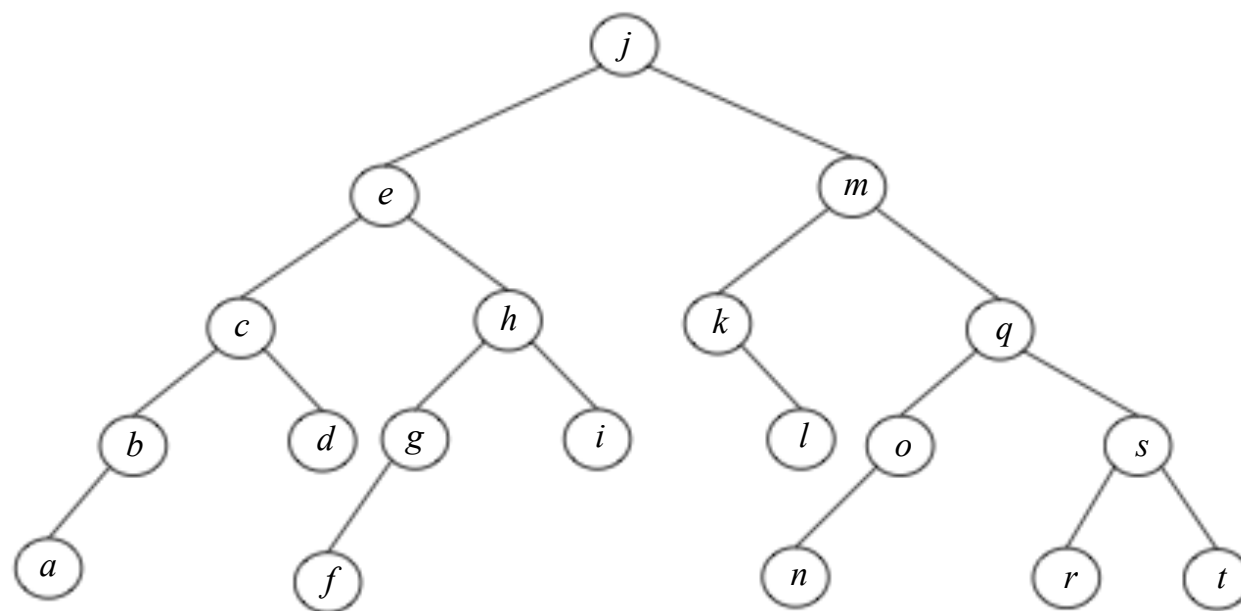
然后用o取代p，
删除o，



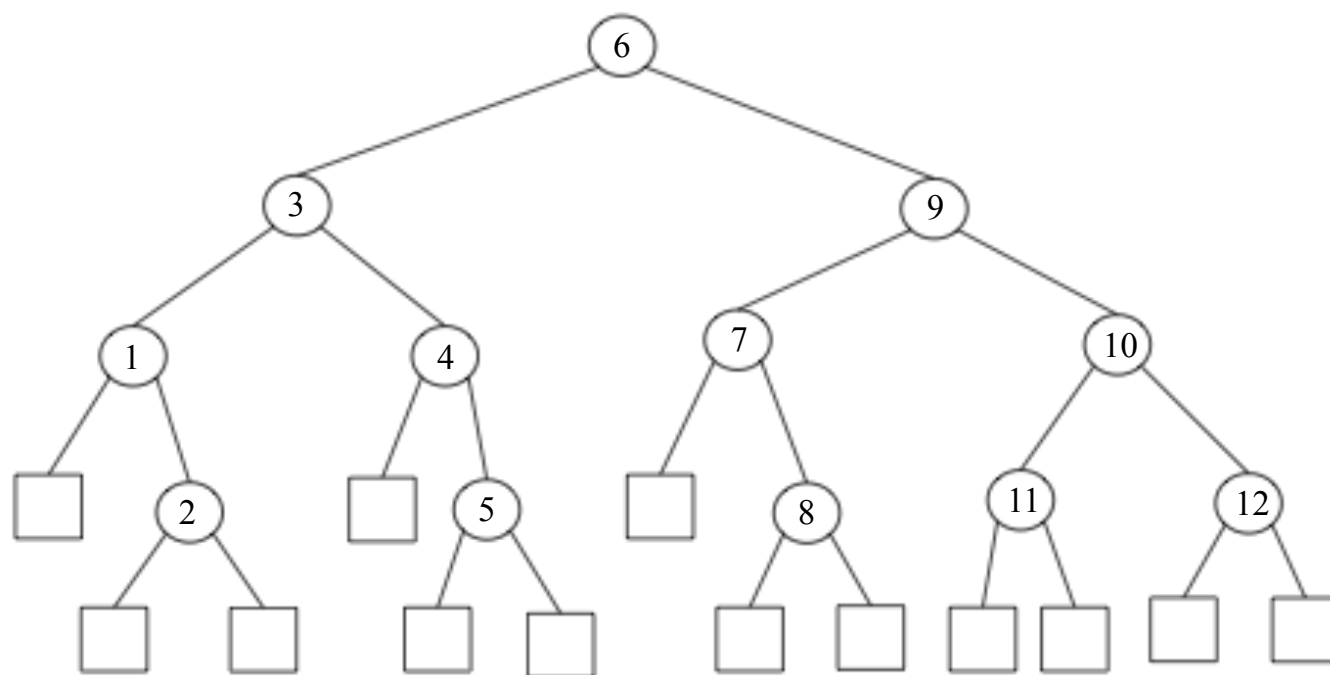


此时结点o的平衡度变为2，发生不平衡，故应对子树o,r,t进行RL型调整。

结点m的平衡度变为-2，发生不平衡，故应对子树m,e,j进行LR调整。

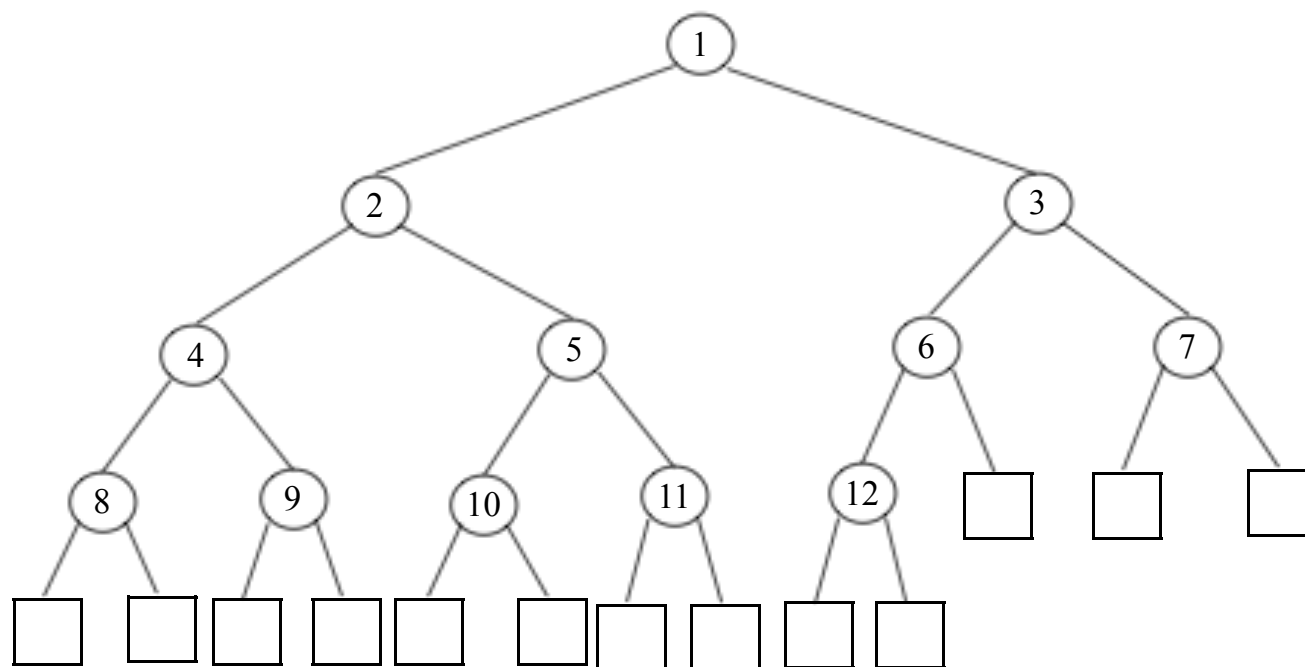


例：长度为12的有序表，按折半查找法对该表进行查找，以等概率查找表内各个元素，则查找成功时所需要的比较次数是多少？



$$ASL_{成功} = (1*1 + 2*2 + 4*3 + 5*4) / 12 = 37/12$$

$$ASL_{失败} = (3*4 + 10*5) / 13 = 62/13$$



$$ASL_{成功} = (1*1 + 2*2 + 4*3 + 5*4) / 12 = 37/12$$

$$ASL_{失败} = (3*4 + 10*5) / 13 = 62/13$$

【例5-5】判断任意二叉树是否是平衡二叉树。

```
int IsAVL(AVL *T) //判断平衡二叉树
```

```
{  int hl,hr;
    if(T==NULL)
        return(1);
    else
    {
        hl=Depth(T->lchild);
        hr=Depth(T->rchild);
        if(abs(hl-hr)<=1)
        {
            if(IsAVL(T->lchild))
                return(IsAVL(T->rchild));
        }
        else
            return(0);
    }
}
```

```
int Depth(AVL *bt) //求二叉树的深度
```

```
{  int ldepth,rdepth;
    if(bt==NULL)
        return(0);
    else
    {  ldepth=Depth(bt->lchild);
        rdepth=Depth(bt->rchild);
        if(ldepth>rdepth)
            return(ldepth+1);
        else
            return(rdepth+1);
    }
}
```

【定义一】一颗m-路查找树或者为空，或者满足以下性质：

(1) 根结点至多包含 m 棵子树，且具有以下结构：

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

其中， A_i 是指向子树的指针， $0 \leq i \leq n < m$ ； K_i 是关键字值，

$$0 \leq i \leq n < m$$

(2) $K_i < K_{i+1}$, $1 \leq i < n$;

(3) 子树 A_i 中的所有关键字值都小于 K_{i+1} ，而大于 K_i , $1 < i < n$;

(4) 子树 A_n 中的所有关键字值都大于 K_n ，子树 A_0 中的所有关键字值都小于 K_1 ;

(5) 每棵子树 A_i 都是 m -路查找树， $0 \leq i \leq n$ 。

设 m -路查找树高度为 h ，其结点数量的最大值是：

$$\sum_{i=0}^{h-1} m^i = \frac{m^h - 1}{m - 1}$$

由于每个结点最多包含 $m-1$ 个关键字，因此在一棵高度为 h 的 m -路查找树中，最多可容纳 $m^h - 1$ 个关键字。

【例5-6】 $h=3$ 的二叉树，最多容纳 7 个关键字；而对于 $h=3$ 的 200-路查找树，最多有 $m^h - 1 = 8 * 10^6 - 1$ 个关键字。

【定义二】 m 分支查找树 T , 是指其所有结点的度都不大于 m 的查找树。空树 T 是一株 m 分支查找树。当 T 非空时, 它有下列性质:

(1) T 是形式如下的一个结点:

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

其中, A_i ($0 \leq i \leq n < m$) 是指向 T 之子树的指针,

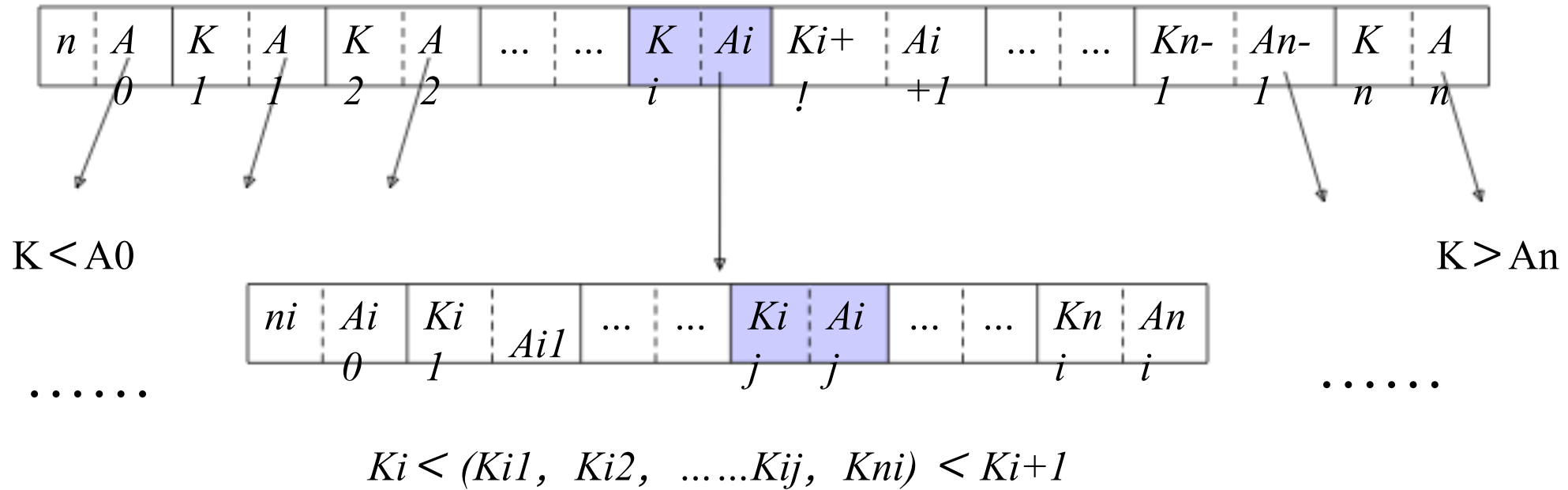
而 K_i ($0 \leq i \leq n$) 是元素, 并且 $0 \leq i \leq n < m$;

(2) $K_i < K_{i+1}$, ($1 \leq i < n$);

(3) 子树 A_i 中的所有元素值小于 K_{i+1} ($0 \leq i < n$);

(4) 子树 A_n 中的所有元素值大于 K_n ;

(5) 子树 A_i ($0 \leq i \leq n$) 也是 m 分支查找树。




5.6 B- 树 B-Tree Balanced-Tree

B-树：B-树是一种非二叉的查找树

除了要满足查找树的特性，还要满足以下结构特性。

一棵 m 阶的 B- 树：

- (1) 树的根或者是一片叶子(一个节点的树),或者其儿子数在 2 和 m 之间;
- (2) 除根外，所有的非叶子结点的孩子数在 $m/2$ 和 m 之间;
- (3) 所有的叶子结点都在相同的深度。



Ceil($m/2$)

B-树的平均深度为 $\log m/2N$ 。执行查找的平均时间为 $O(\log m)$;

B-树应用在数据库系统中的索引，它加快了访问数据的速度；

【B-树定义】

一颗 m 阶 B-树 是一颗 m -路查找树，它或为空，或满足以下性质：

- (1) 根结点至少包含 2 个儿子；
- (2) 除根结点和失败结点以外，所有结点都至少具有 $m/2$ 个儿子；
- (3) 所有失败结点都在同一层。

P188?

- ~~所有 2 阶 B-树 都是满二叉树，只有当关键字个数等于 2^k-1 时，才存在 2 阶 B-树~~
- 对于任意给定的关键字和任意的 $m(>2)$ ，一定存在一棵包含上述所有关键字的 m 阶 B-树。

1、B-树中的关键字值个数

所有失败结点都在 $l+1$ 层的 m 阶 B-树至多包含 $ml-1$ 个关键字。

B-树的第 1 层至少包含 2 个儿子

B-树的第 2 层至少包含 2 个结点

B-树的第 3 层至少包含 $2*(m/2)$ 个结点

B-树的第 4 层至少包含 $2*(m/2)^2$ 个结点

...

B-树的第 $l(>1)$ 层至少包含 $2*(m/2)^{l-2}$ 个结点

每个结点又至少
包含 $m/2$ 个儿子

注：上述所有结点都不是失败结点

若B-树中的关键字分别为 K_1, K_2, \dots, K_n ，且有 $K_i < K_{i+1}$ ，($1 \leq i < N$)
则失败结点的个数为 $N+1$ 。
因为当 x 满足 $K_i < x < K_{i+1}$ ，就会查找失败，其中： $K_0 = -\infty$ ， $K_{N+1} = +\infty$
使得查找不在B-树中的关键字 x 时，就可能到达 $N+1$ 个不同的失败结点。
因此：

$$N+1 = \text{失败结点个数} = \text{第 } l+1 \text{ 层的结点个数} \geq 2^{*(m/2)l-1}$$

$$\text{故有 } N \geq 2^{*(m/2)l-1}, l \geq 1$$

反之，如果一棵 m 阶B-树包含 N 个关键字，则所有非失败结点所在的层号都小于或等于 l ， $l \leq \log_{m/2} \lfloor (N+1)/2 \rfloor + 1$
即每次查找所需进行至多 $l+1$ 次的结点访问。

【例5-7】 $m=200$ 的B-树，当 $N \leq 2 \times 10^6 - 2$ 时，有 $l \leq \log_{100} \lfloor (N+1)/2 \rfloor + 1$
由于 l 为整数，所以 $l \leq 3$ 。而当 $N \leq 2 \times 10^8 - 2$ 时，有 $l \leq 4$ 。
因此，采用高阶的B-树可以用很少的磁盘访问次数来查找大规模的索引。

2、B-树阶的选择

在大多数系统中，B-树上的算法执行时间主要由读、写磁盘的次数来决定，每次读写尽可能多的信息可提高算法得执行速度。

B-树中的结点的规模一般是一个磁盘页，而结点中所包含的关键字及其孩子的数目取决于磁盘页的大小。

高阶B-树会降低在查找索引时所需的磁盘访问次数
—— 人们期望使用高阶B-树。

如果索引中包含 N 个元素，则 $m=N+1$ 阶B-树就只有一层
—— 但 m 的选择受到内存可用空间的限制。

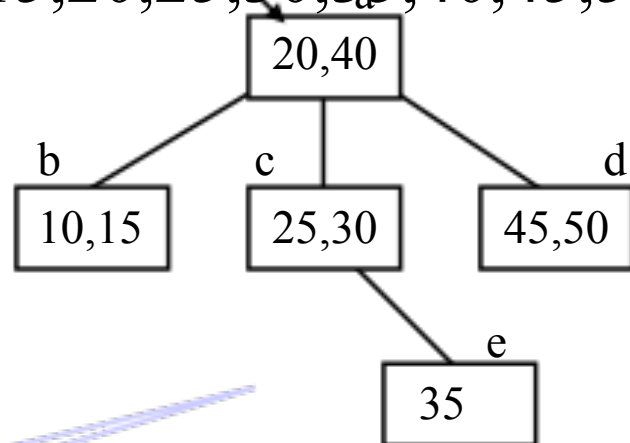
选取较大的结点度数可降低树的高度，以及减少查找任意关键字所需的磁盘访问次数；

如果 m 的值很大，使得索引不能一次性地全部读入内存，就会增加读盘次数，也会增加结点内的查找时间， m 的选择不合理；

合理地选择 m 的目标可使得在B-树中查找关键字 x 所需的总时间最小！

B-树上查找 x 的时间	从磁盘上读入结点的时间
	在结点中查找 x 所需要的时间

【例5-8】 3 分支查找树，元素为
(10,15,20,25,30,35,40,45,50)。

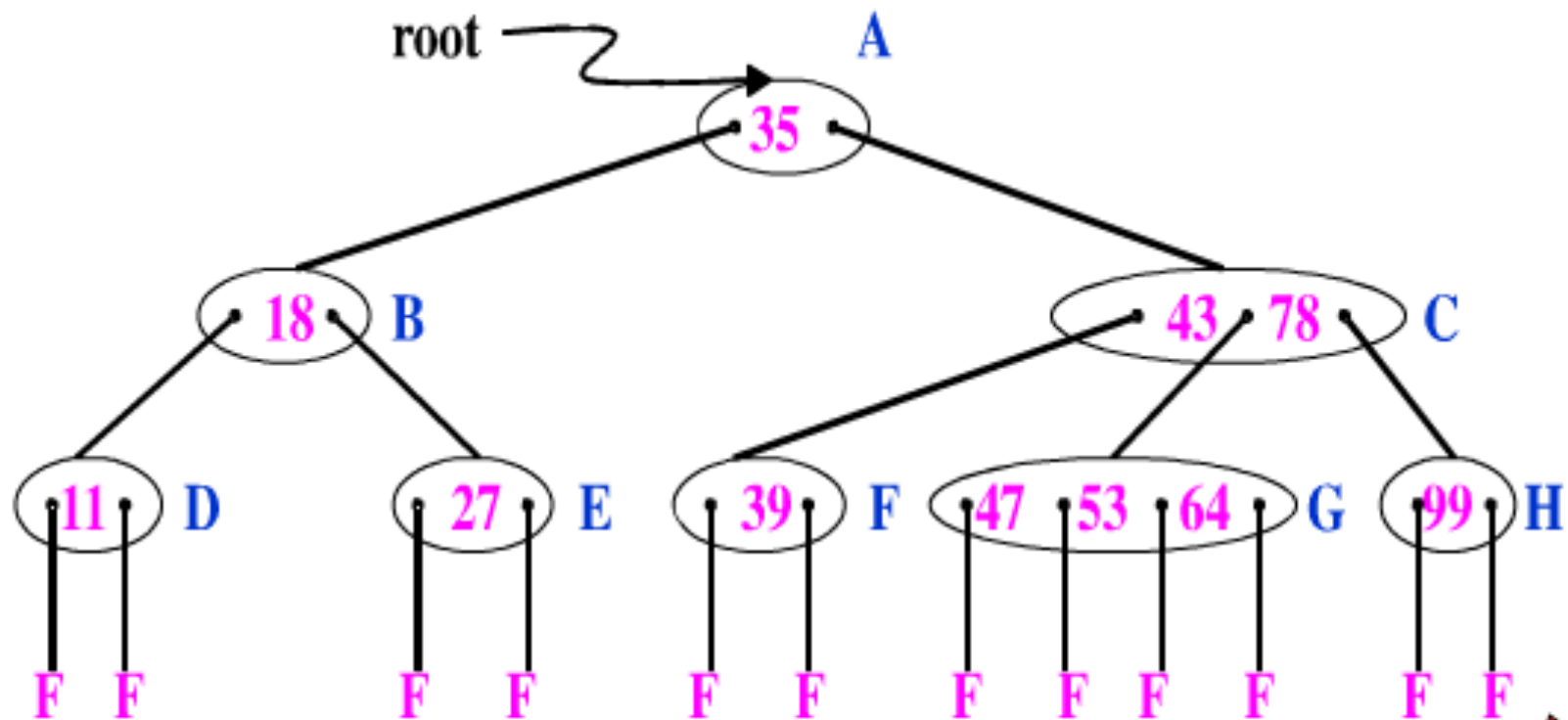


非B-树

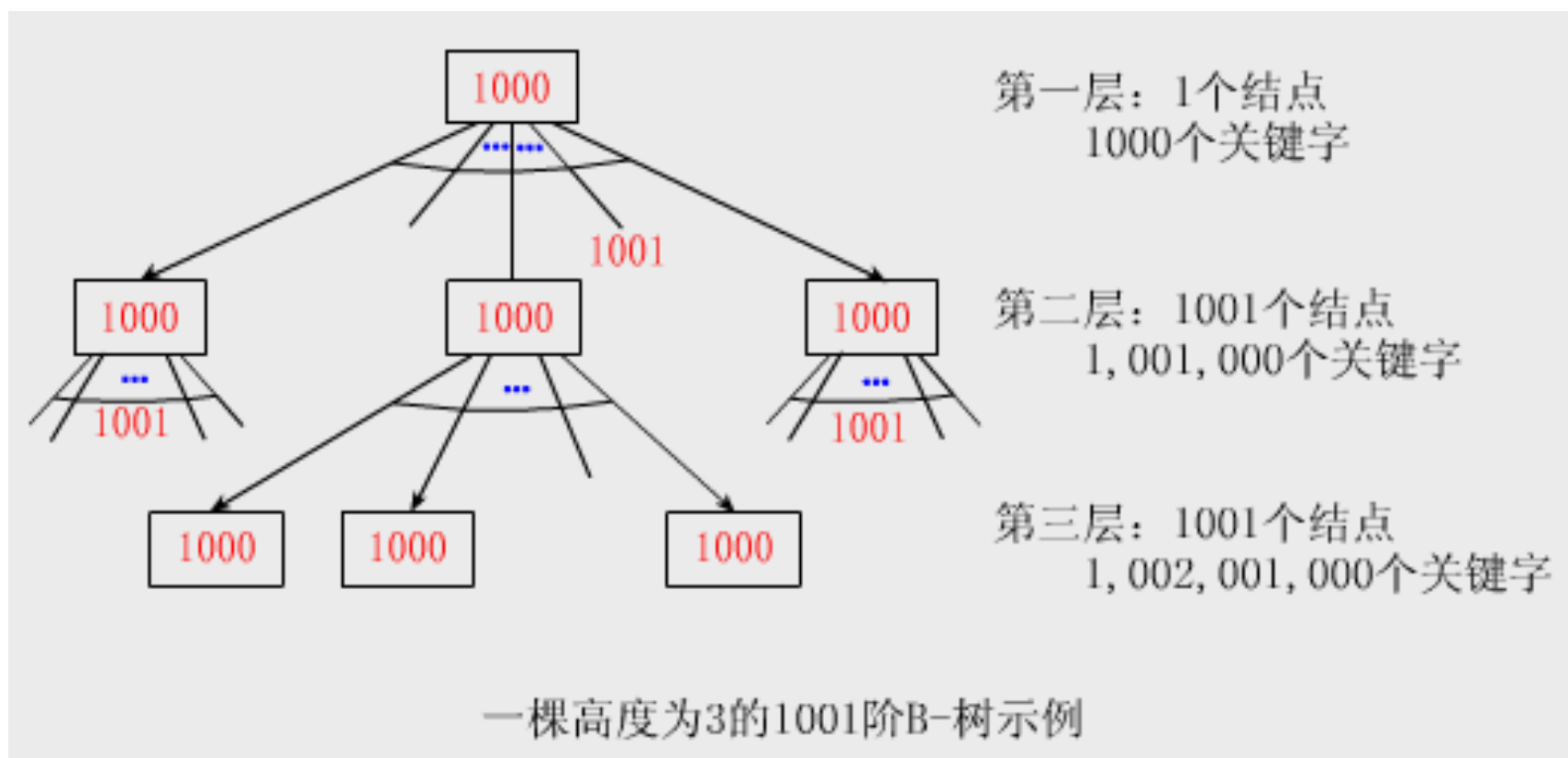
结点	n	A0	(K1,A1)	(K2,A2)
a	2	b	(20,C)	(40,d)
b	2	^	(10, ^)	(15, ^)
c	2	^	(25, ^)	(30,e)
d	2	^	(45, ^)	(50, ^)
e	1	^	(35, ^)	

【重申】 m 阶的 B- 树 T 是这样一棵 m 分支查找树： T 或者为空；或者其高度不小于 1 且满足下述性质：

- (1) 根结点至少有两个儿子；
- (2) 除根和叶子结点外的所有结点，至少有 $m/2$ 个儿子；
- (3) 所有的叶子结点都在同一层上。



【例5-9】一棵高度为3的1001阶B-树。



【说明】

- ①每个结点包含1000个关键字，故在第三层上有100多万万个叶结点，这些叶节点可容纳10亿多个关键字；
- ②图中各结点内的数字表示关键字的数目；
- ③通常根结点可始终置于主存中，因此在这棵B-树中查找任一关键字至多只需二次访问外存。

3、B-树的存储结构

```
#define Max 1000
```

//结点中关键字的最大数目: $\text{Max} = m - 1$, m 是B-树的阶

```
#define Min 500
```

//非根结点中关键字的最小数目: $\text{Min} = \lceil m/2 \rceil - 1$

```
typedef int KeyType; //KeyType应由用户定义
```

```
typedef struct node //结点定义中省略了指向关键字代表的记录的指针
```

```
{ int keynum //结点中当前拥有的关键字的个数,  $\text{keynum} < \text{Max}$ 
```

```
    KeyType key[Max+1] //关键字向量为 $\text{key}[1..\text{keynum}]$ ,  $\text{key}[0]$ 不用
```

```
    struct node *parent; //指向双亲结点
```

```
    struct node *son[Max+1]; //孩子指针向量为 $\text{son}[0..\text{keynum}]$ 
```

```
} BTreeNode;
```

```
typedef BTreeNode *BTree;
```

4、B-树的查找

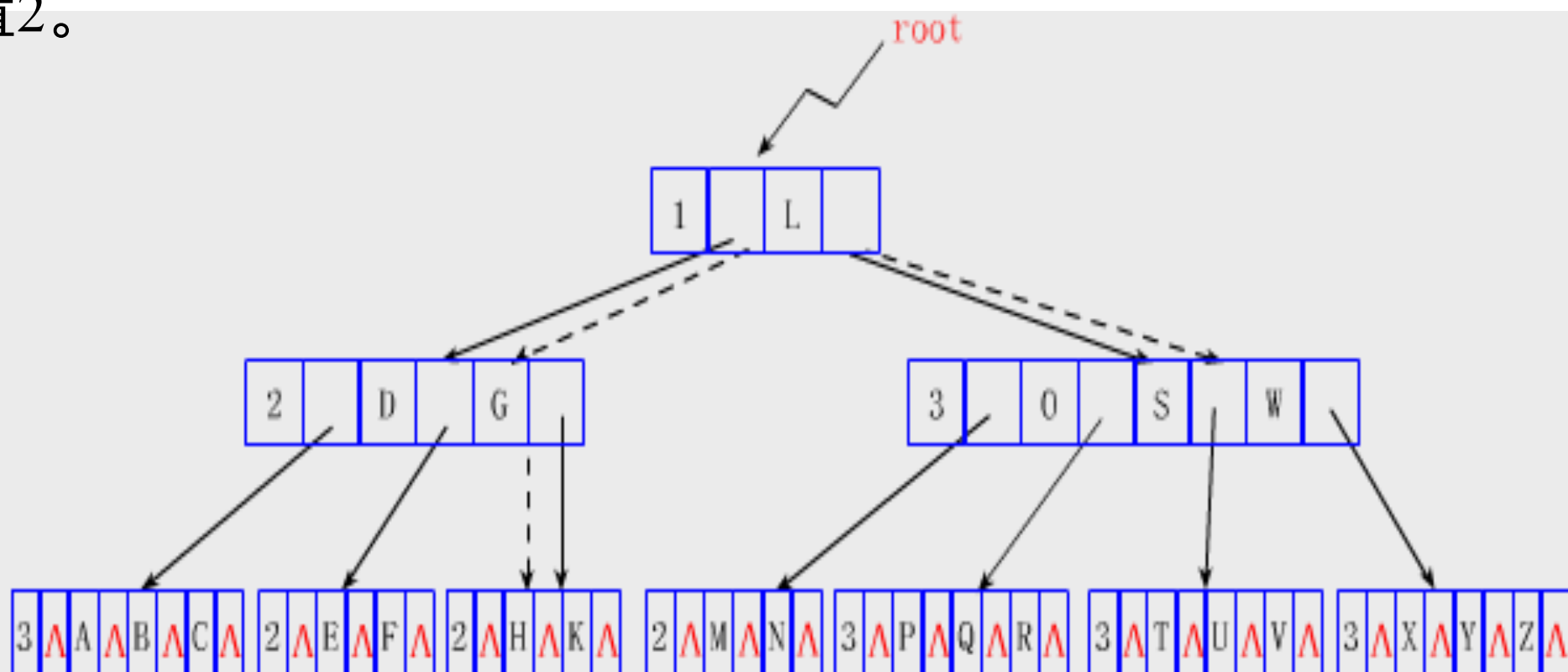
(1) B-树的查找方法

在B-树中查找给定关键字的方法类似于二叉排序树上的查找。不同的是在每个结点上确定向下查找的路径不一定是二路而是keynum+1路的。

➤对结点内的存放有序关键字序列的向量key[1..keynum] 用顺序查找或折半查找方法查找；

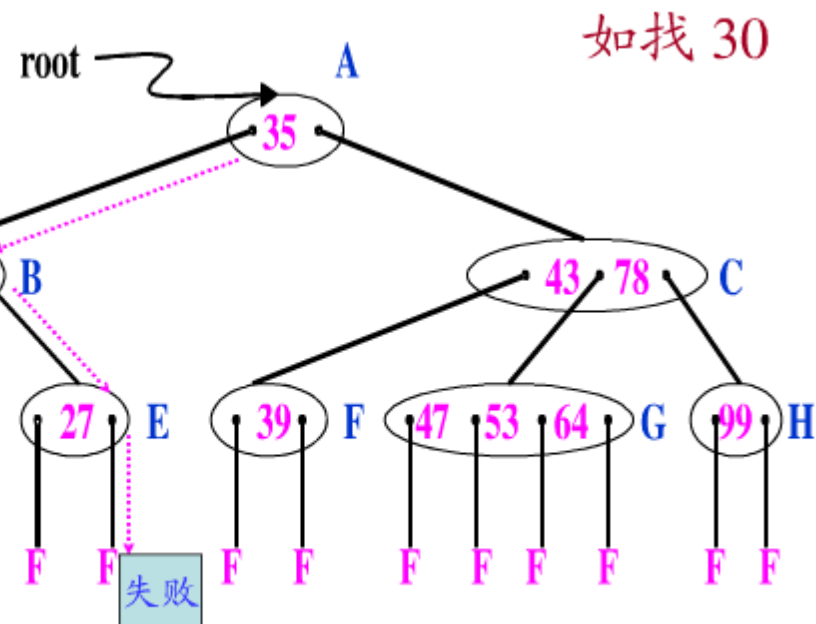
➤若在 某结点内找到待查的关键字K，则返回该结点的地址及K在key[1..keynum]中的位置； 否则，确定K在某个key[i]和key[i+1]之间结点后，从磁盘中读son[i]所指的结点继续找.....。

【例5-10】下图中左边的虚线表示查找关键字 l 的过程，它失败于叶结点的H和K之间空指针上；右边的虚线表示查找关键字 S 的过程，并成功地返回 S 所在结点的地址和 S 在 $key[1..keynum]$ 中的位置2。

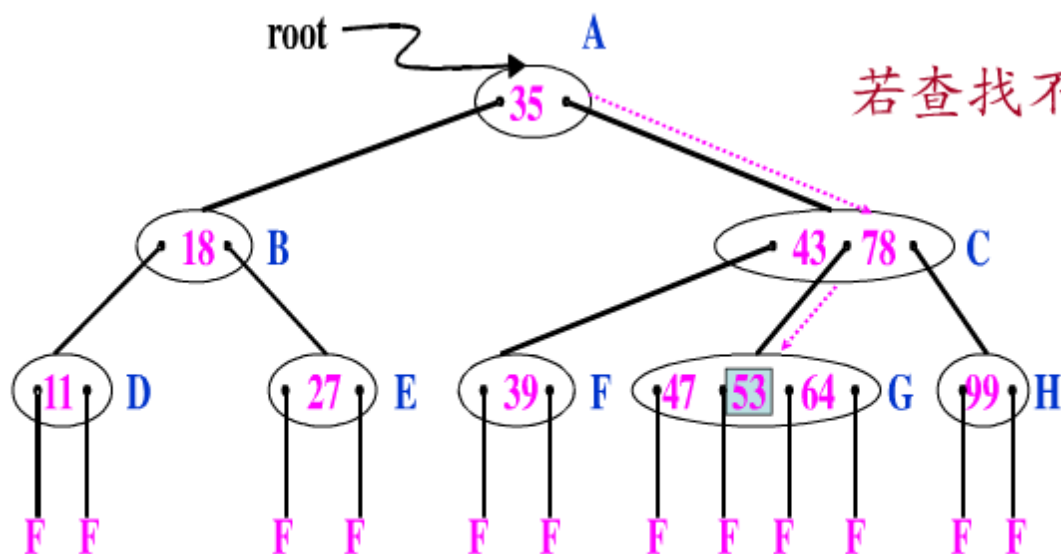


一棵5阶的B-树

【例5-11】



若查找不成功，则返回插入位置。



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

(2) B-树的查找算法

```
BTreeNode *SearchBTree(BTree T, KeyType K, int *pos)
{ //在B-树T中查找关键字K, 成功时返回找到的结点的地址及K在其中的位置*pos
  //失败则返回NULL
  int i;
  T->key[0]=k; //设哨兵. 下面用顺序查找key[1..keynum]
  for(i=T->keynum; K<t->key[i];i--); //从后向前找第1个小于等于K的关键字
  if(i>0 && T->key[i]==1){ //查找成功, 返回T及i
    *pos=i;
    return T;
  } //结点内查找失败, 但T->key[i]<K<T->key[i+1], 下一个查找的结点应为son[i]
  if(!T->son[i]) // *T为叶子, 在叶子中仍未找到K, 则整个查找过程失败
    return NULL;
  //查找插入关键字的位置, 则应令*pos=i, 并返回T, 见后面的插入操作
  DiskRead(T->son[i]); //在磁盘上读入下一查找的树结点到内存中
  return SearchBTree(T->Son[i], k, pos); //递归地继续查找于树T->son[i]
}
```

(3) 查找操作的时间开销

B-树上的查找有两个基本步骤：

- ①在B-树中查找结点，该查找涉及读盘操作，属外部查找；
- ②在结点内查找，该查找属内查找。

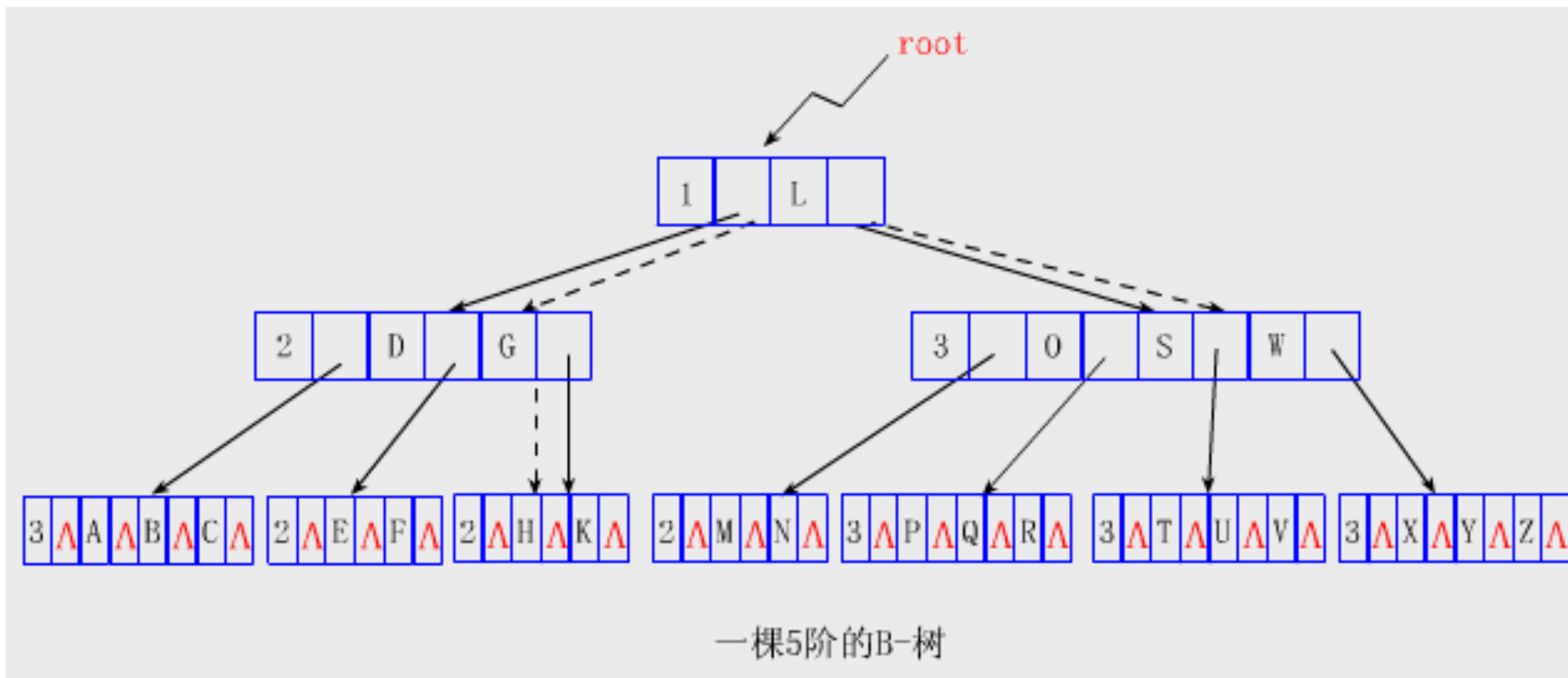
查找操作的时间为：

- ①外查找的读盘次数不超过树高 h ，故其时间是 $O(h)$ ；
- ②内查找中，每个结点内的关键字数目 $keynum < m$ (m 是B-树的阶数)，故其时间为 $O(nh)$ 。

注意：

- ①实际上外部查找时间可能远远大于内部查找时间；
- ②B-树作为数据库文件时，打开文件之后就必須将根结点读入内存，而直至文件关闭之前，此根一直驻留在内存中，故查找时可以不计读入根结点的时间。

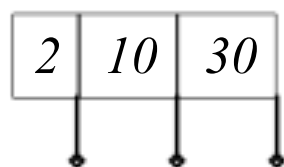
【例5-12】一棵包含24个英文字母的5阶B-树的存储结构图。



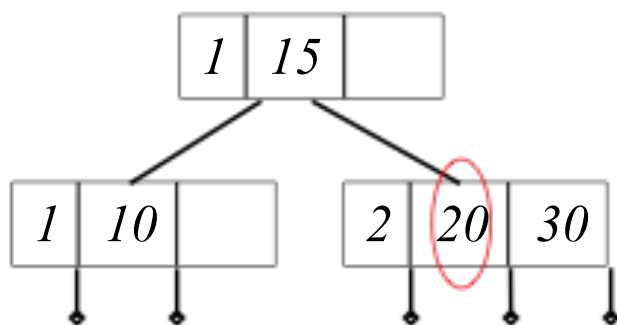
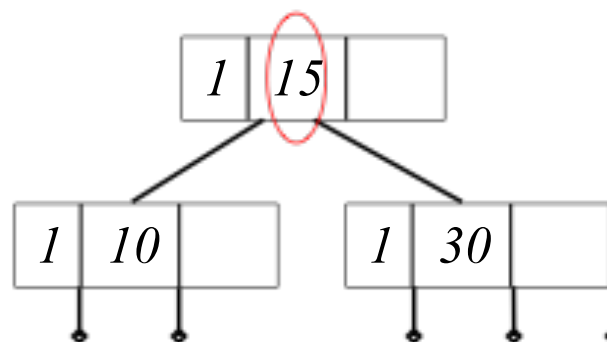
按照定义，在5阶B-树里，根中的关键字数目可以是1~4，子树数可以是2~5；其它的结点中关键字数目可以是2~4，若该结点不是叶子，则它可以有3~5棵子树。

5、B-树的插入操作

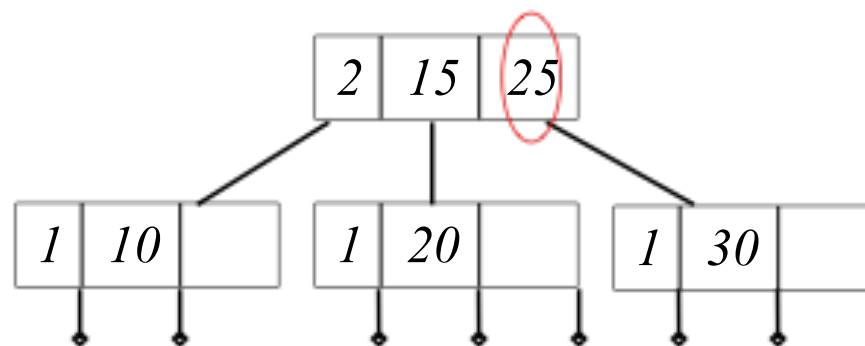
【例5-13】



未分裂

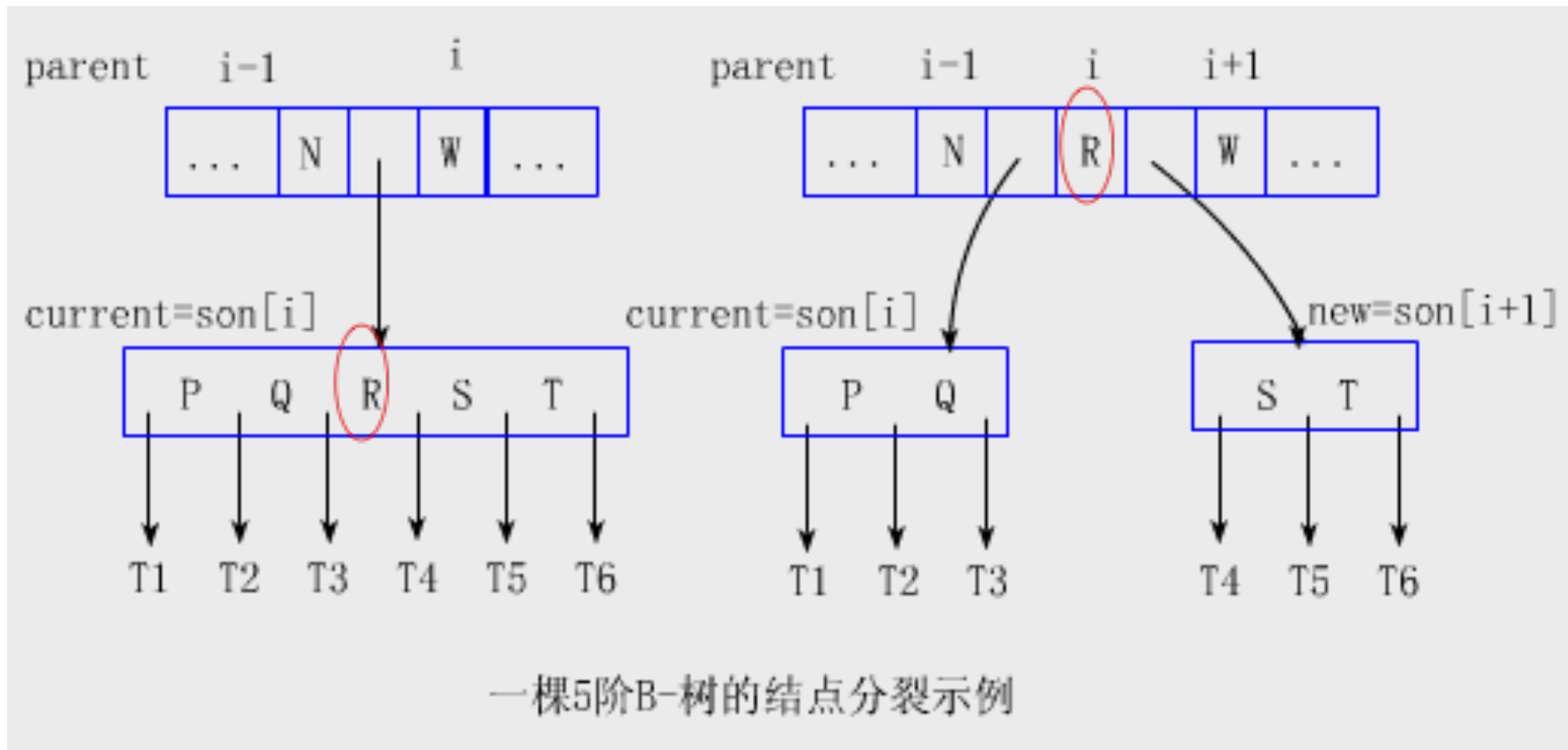


分裂1次



分裂2次

【例5-14】



【例5-15】以关键字序列：

(a, g, f, b, k, d, h, m, i,
e, s, i, r, x, c, l, n, t, u, p)

建立一棵5阶B-树的生长过程。

注意：

①当一结点分裂时所产生的两个结点大约是半满的，这就为后续的插入腾出了较多的空间,尤其是当 m 较大时，往这些半满的空间中插入新的关键字不会很快引起新的分裂；

②向上插入的关键字总是分裂结点的中间位置上的关键字，它未必是正待插入该分裂结点的关键字。因此，无论按何次序插入

关键字序列 : $(a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p)$

(1)

<i>a</i>

(2)

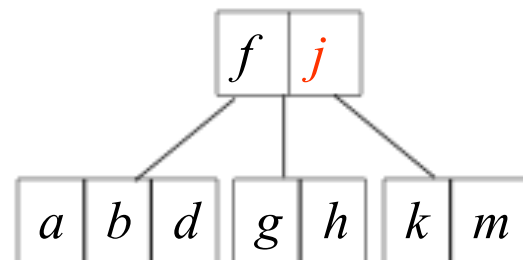
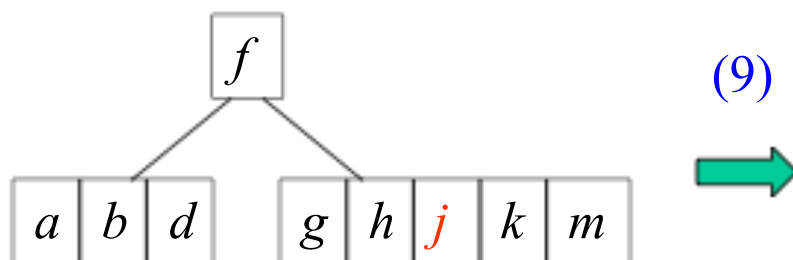
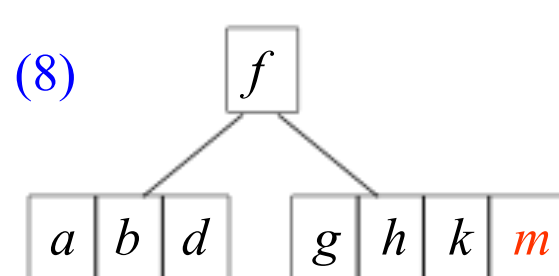
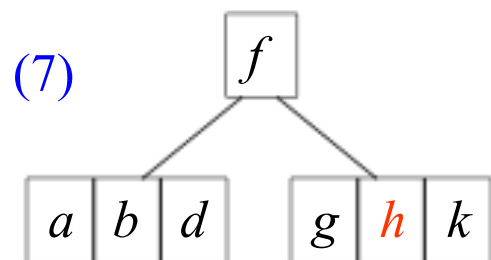
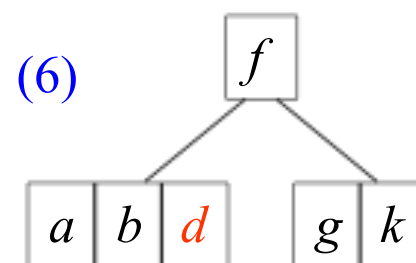
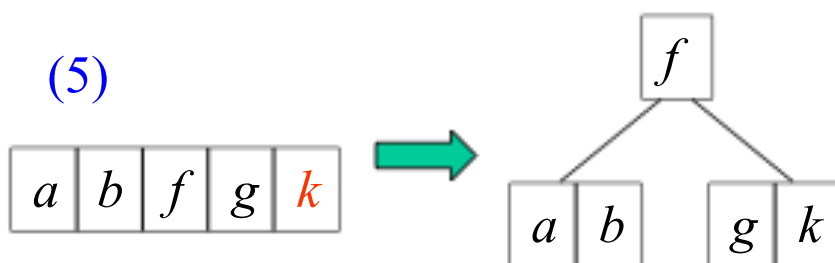
<i>a</i>	<i>g</i>
----------	----------

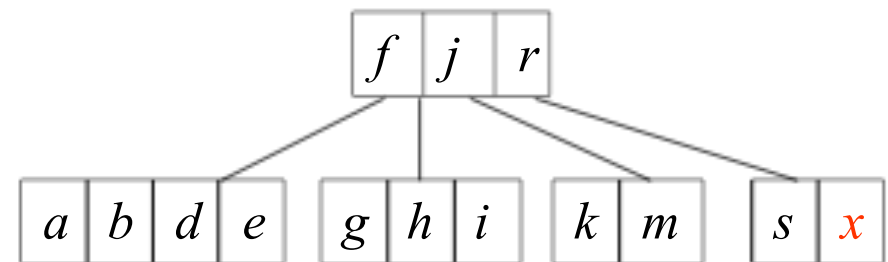
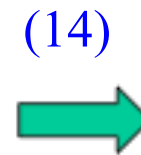
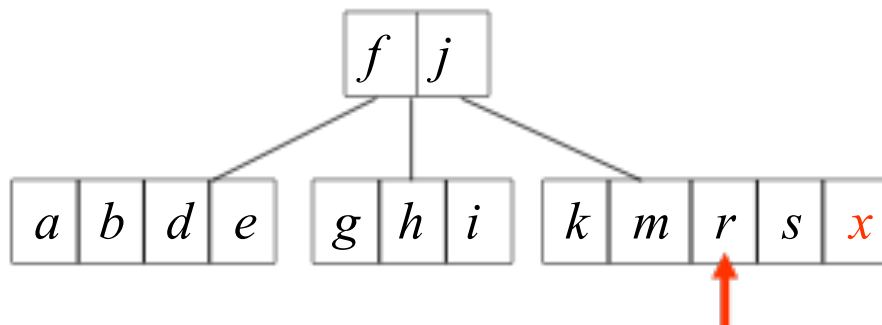
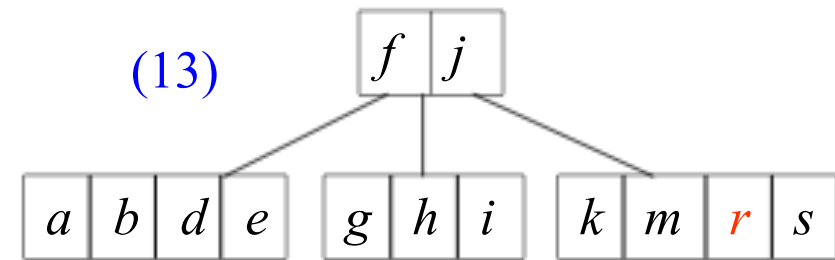
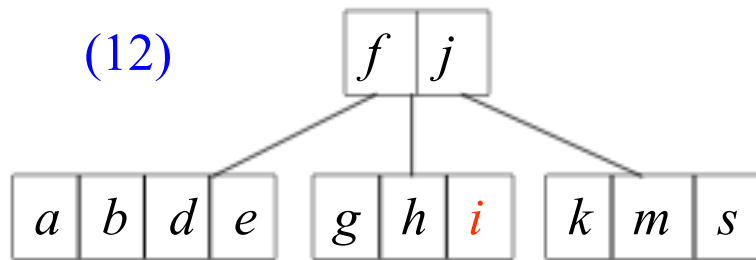
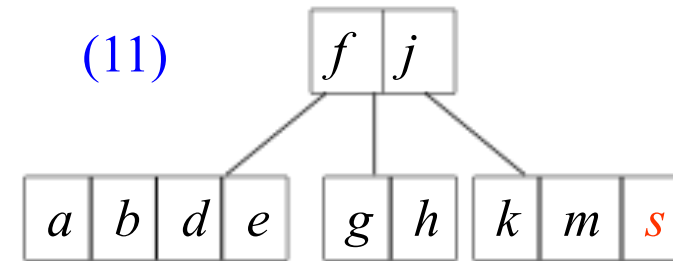
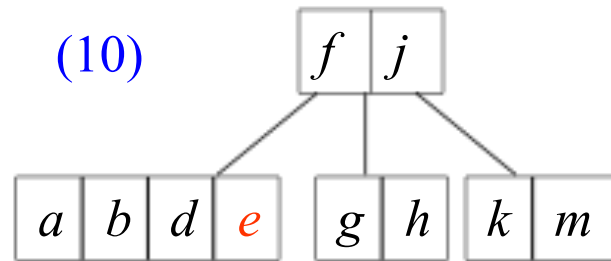
(3)

<i>a</i>	<i>f</i>	<i>g</i>
----------	----------	----------

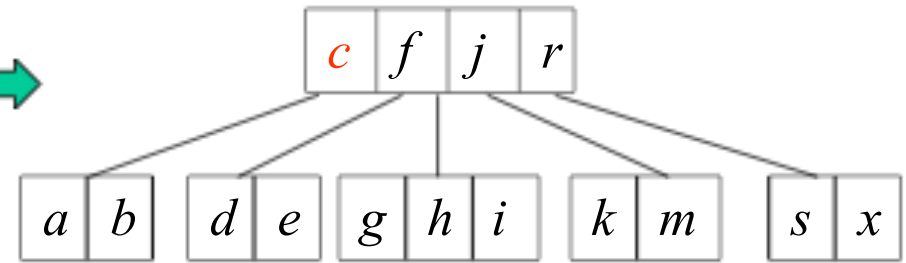
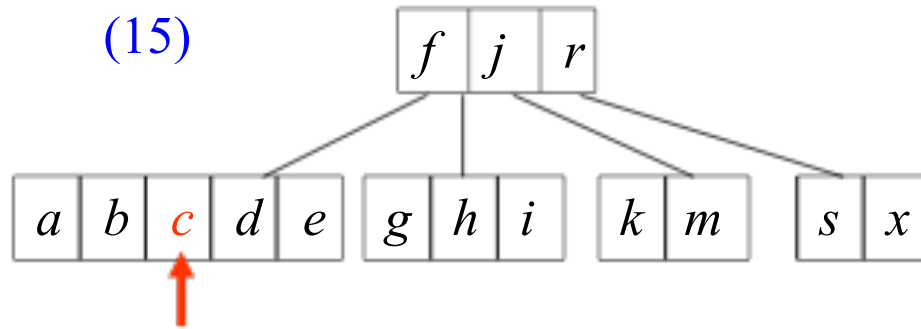
(4)

<i>a</i>	<i>b</i>	<i>f</i>	<i>g</i>
----------	----------	----------	----------

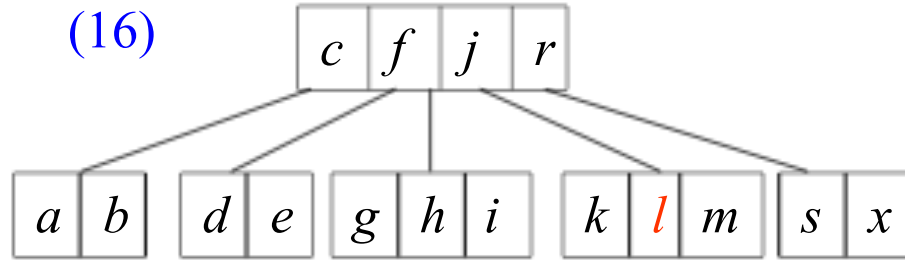




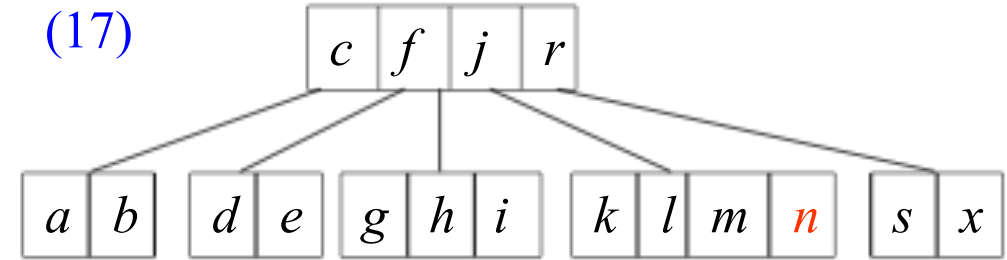
(15)



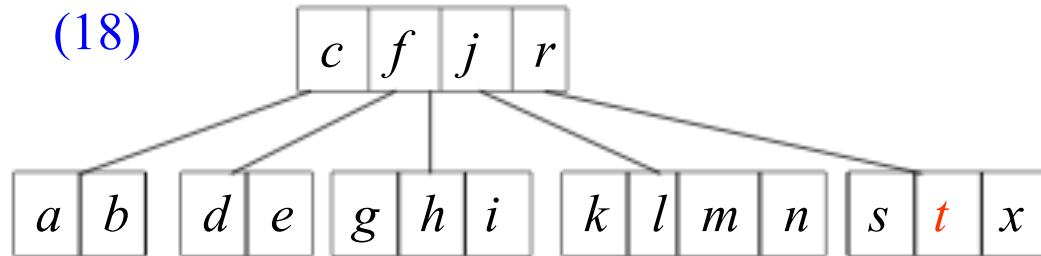
(16)



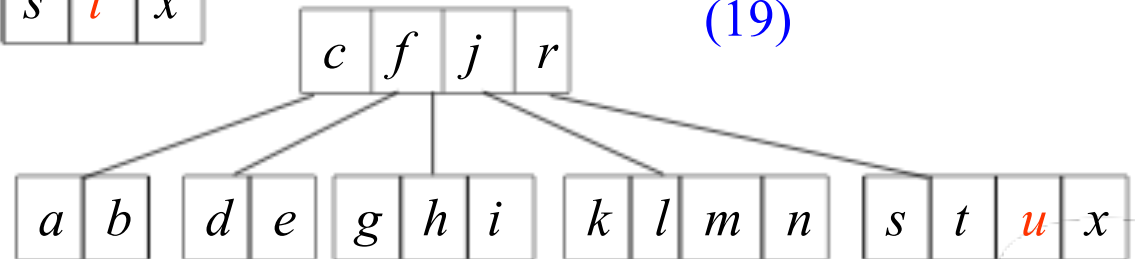
(17)

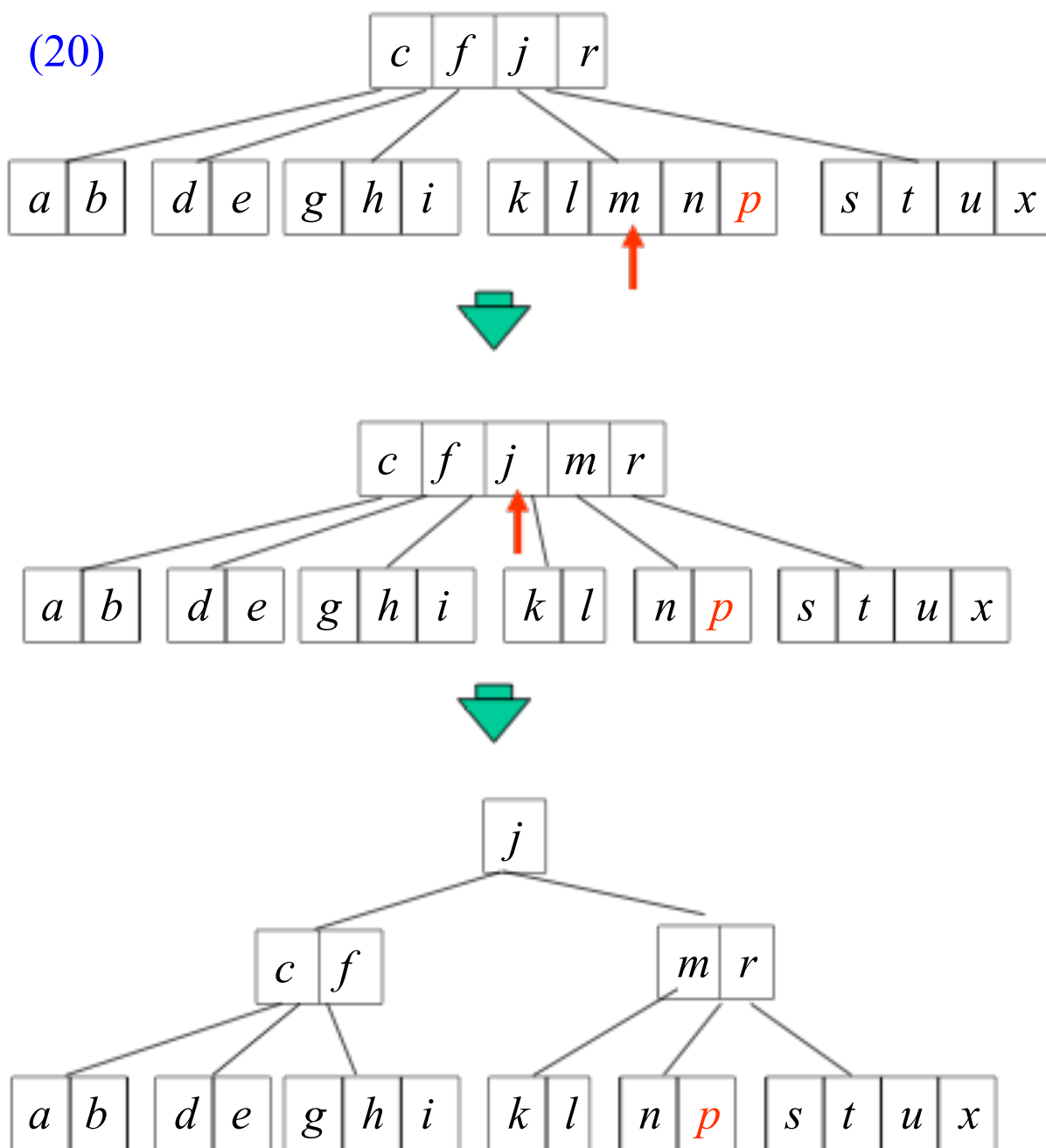


(18)



(19)





重申：

①当一结点分裂时所产生的两个结点大约是半满的，这就为后续的插入腾出了较多的空间，尤其是当 m 较大时，往这些半满的空间中插入新的关键字不会很快引起新的分裂；

②向上插入的关键字总是分裂结点的中间位置上的关键字，它未必是正待插入该分裂结点的关键字。因此，无论按何次序插入关键字序列，树都是平衡的。

插入结点小结

首先执行插入操作以确定可以插入新关键字的叶结点 p 。

如果在结点 p 上插入新关键字是结点 p 的关键字达到 m ，则需要分裂结点 p 。

否则，只需将新结点 p 写入磁盘上，完成插入操作。

分裂节点：假定插入新关键字后，结点 p 具有如下结构：

$m, A0, (K1, A1), (K2, A2), \dots, (Km, Am)$, 且 $K_i < K_{i+1}, 0 \leq i < m$

分裂后，形成具有如下格式的两个结点 p 和 q ：

结点 p : $m/2, A0, (K1, A1), \dots, (Km/2-1, Am-1)$

结点 q : $m-m/2, Am/2, (Km/2, Am/2+1), \dots, (Km, Am)$

剩下的关键字 $K_{m/2}$ 和指向新结点 q 的指针形成一个二元组 $(K_{m/2}, q)$ 。

该二元组将被插入到 p 的父结点中。插入前，将结点 p 和 q 写盘。

插入父结点有可能会将这个父结点的分裂，而且此分裂过程可能会一直向上传播，直到根结点为止。

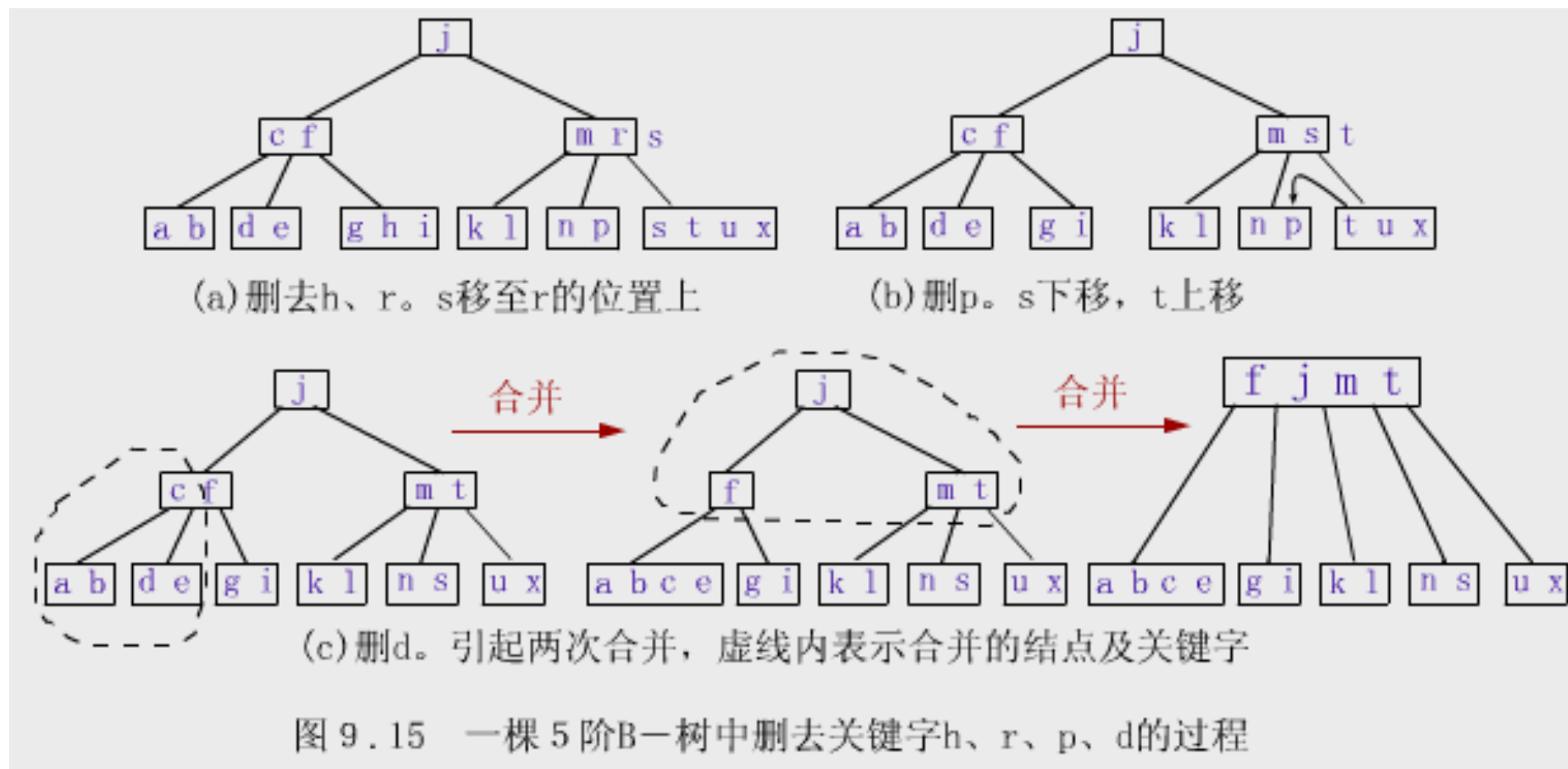
根结点分裂时，创建一个只包含一个关键字的新根结点，B-树的高度增1。



6、B-树的删除操作

- 首先找到要删除的关键字 x 所在的结点 z ;
- 若 z 不是叶结点, 则用该B-树中某个叶结点中的关键字来代替结点 z 中的 x ;
- 假定 x 是 z 中的第 i 个关键字 ($x=K_i$), 就可以用子树 A_i 中的最小关键字或者子树 A_{i-1} 中的最大关键字来替换 x , 而这两个关键字均在叶结点中。 \Rightarrow 把从非叶结点中删除 x 的操作转化成从叶结点中删除 x 的操作。

【例5-16】删除5阶B-树的h、r、p、d等关键字的过程



例15的过程分析：

第1个被删的关键字h是在叶子中，且该叶子的关键字个数 $>m/2-1$ (5阶B-树的 $\text{Min}=2$)，故直接删去即可；

第2个删去的r不在叶子中，故用中序后继s取代r，即把s复制到r的位置上，然后从叶子中删去s；

第3个删去的p所在的叶子中的关键字数目是最小值 Min ，但其右兄弟的关键字个数 $>m/2-1$ ，故可以通过左移，将双亲中的s移到p所在的结点，而将右兄弟中‘最小(即最左边)’的关键字t上移至双亲取代s；

当删去d时，d所在的结点及其左右兄弟均无多余的关键字，故需将删去d后的结点与这两个兄弟中的一个(图中是选择左兄弟(ab))及其双亲中分隔这两个被合并结点的关键字c合并在一起形成一个新结点(abce)。但因为双亲中失去c后关键字个数 $<m/2-1$ ，故必须对该结点做调整操作，此时它只有一个右兄弟，且右兄弟无多余的关键字，不可能通过移动关键字来解决。因此引起再次合并，因根只有一个关键字，故合并后树高度减少一层，从而得到上图的最后一个图。

删除小结

从叶结点 p 删除关键字后，分为 4 种不同情况分别进行处

理。

- (1) p 同时也是根结点，若删除 x 后还至少剩一个关键字，写盘，结束；否则，B-树变成空树；

- (1) 删除 x 后结点 p 至少还有 $m/2-1$ 个关键字，写盘，结束；

- (1) 删除 x 后结点 p 有 $m/2-2$ 个关键字，其邻近的右兄弟（左兄弟）结点 q 至少还有 $m/2$ 个关键字。

假定 r 是 p 和 q 的父结点，设 K_i 是 r 中大于（或小于） x 的最小（或最大）关键字，将 K_i 下移至 p ，把 q 的最小（或最大）关键字上移到 r 的 K_i 处，把 q 的最左（或最右）子树指针平移到 p 中最后（或最前）子树的指针处，在 q 中，将被移走的关键字和指位置有剩余的關鍵字和指针补齐，再将其中的关键字个数减1。将 p 、 q 、 r 写盘，结束；

删除小结（续）

- (1) 删除 x 后结点 p 有 $m/2-2$ 个关键字，其邻近的右兄弟（左兄弟）结点 q 至少还有 $m/2-1$ 个关键字。

将 p 、 q 和关键字 K_i 合并，形成一个结点，有少于 $m-1$ 个关键字。合并操作使父结点 r 的关键字个数减1。若 r 中关键字个数仍满足要求（根至少1个关键字，非根结点至少 $m/2-1$ 个关键字），写盘，结束。否则，父结点 r 的关键字不足，若是根，其中无关键字，删除；若 r 不是根结点，其关键字个数应为 $m/2-2$ 。要与其兄弟结点合并，并沿B-树向上进行，直到根结点的儿子被合并为止。

B-树的高度及性能分析

B-树上操作的时间由存取磁盘的时间和CPU计算时间构成。

- B-树上大部分基本操作所需访问盘的次数均取决于树高 h 。关键字总数相同的情况下B-树的高度越小，磁盘I/O所花的时间越少；
- 与高速的CPU计算相比，磁盘I/O要慢得多，所以可忽略CPU的计算时间，只分析算法所需的磁盘访问次数：

(磁盘访问次数) \times (读写盘的平均时间) 。

(1) B-树的高度

若 $n \geq 1$, $m \geq 3$ ，则对任意一棵具有 n 个关键字的 m 阶B-树，其树高 h 至多为： $\log_t((n+1)/2)+1$ 。

t 是每个(除根外)内部结点的最小度数，即 $\lceil m/2 \rceil$

- B-树的高度为 $O(\log_t n)$ ；
- 在B-树上查找、插入和删除的读写 盘的次数为 $O(\log_t n)$ ；
- CPU计算时间为 $O(m \log_t n)$ 。

(2) 性能分析

(1) n 个结点的平衡的二叉排序树的高度 H （即 $\lg n$ ）比B-树的高度 h 约大 $\lg t$ 倍。

【例5-17】若 $m=1024$ ，则 $\lg t = \lg 512 = 9$ 。此时若B-树高度为4，则平衡的二叉排序树的高度约为36。

显然，若 m 越大，则B-树高度越小；

(1) 若要作为内存中的查找表，B-树却不一定比平衡的二叉排序树好，尤其当 m 较大时更是如此。因为查找等操作的CPU计算时间在B-树上是： $O(m \log t n) = O(\lg n \cdot (m / \lg t))$ ；而 $m / \log t > 1$ ，所以 m 较大时 $O(m \log t n)$ 比平衡的二叉排序树上相应操作的时间 $O(\lg n)$ 大得多。因此，**仅在内存中使用的B-树必须取较小的 m 。**

(1) 通常取最小值 $m=3$ ，此时B-树中每个内部结点可以有2或3个孩子，这种3阶的B-树称为2-3树。

B+树

B+树是B-树的一种变形，二者区别在于：

- (1) 有 k 个子结点的结点必然有 k 个关键码；
- (2) 非叶子结点仅具有索引作用，与记录有关的信息均放在叶结点中。

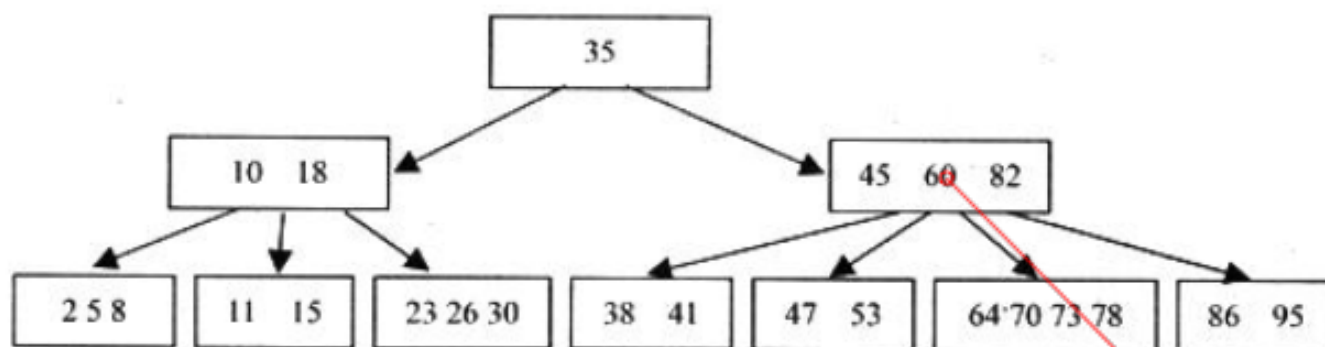
查找： B+树查找过程中，即使找到关键码，也必须向下找到叶子结点；

插入： 插入在叶子上，分裂，改变上层两个最大关键码；

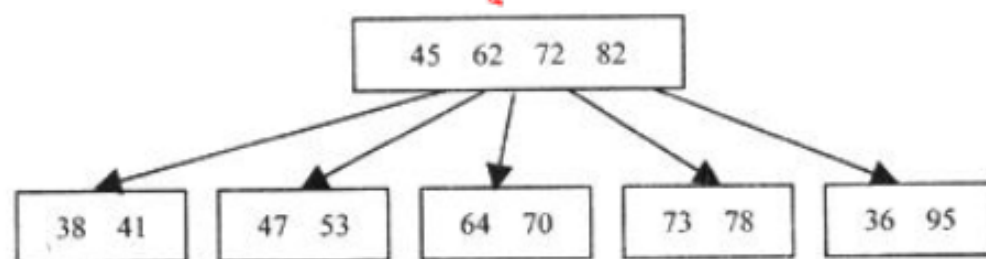
删除： 删除在叶结点，上层关键码可以保留，删除后若关键字少于 $m/2$ ，则与B-同样处理。

B-树只适合随机检索，但B+树同时支持随机检索和顺序检索。

试题基于以下的5阶B树结构，该B树现在的层数为2。

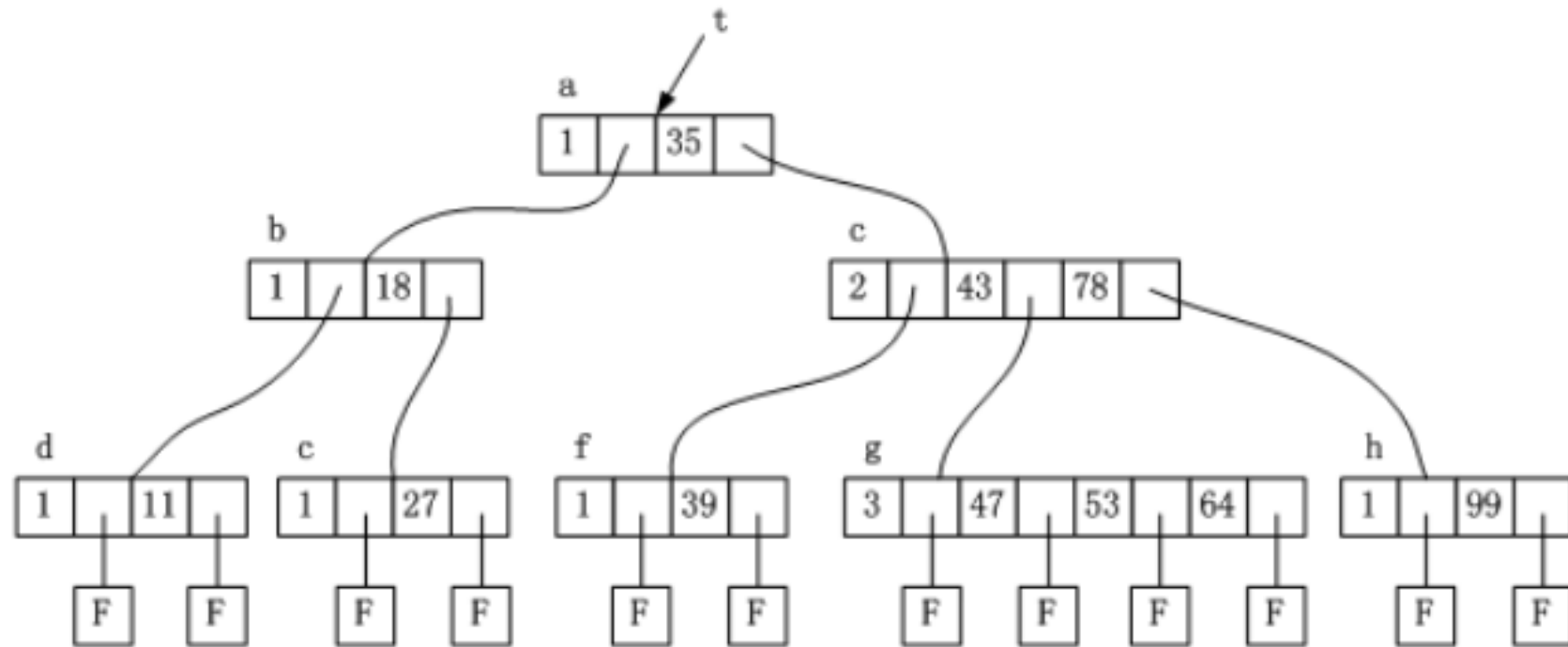


往该B树中插入关键码72后，该B树的第2层的结点数为_____。

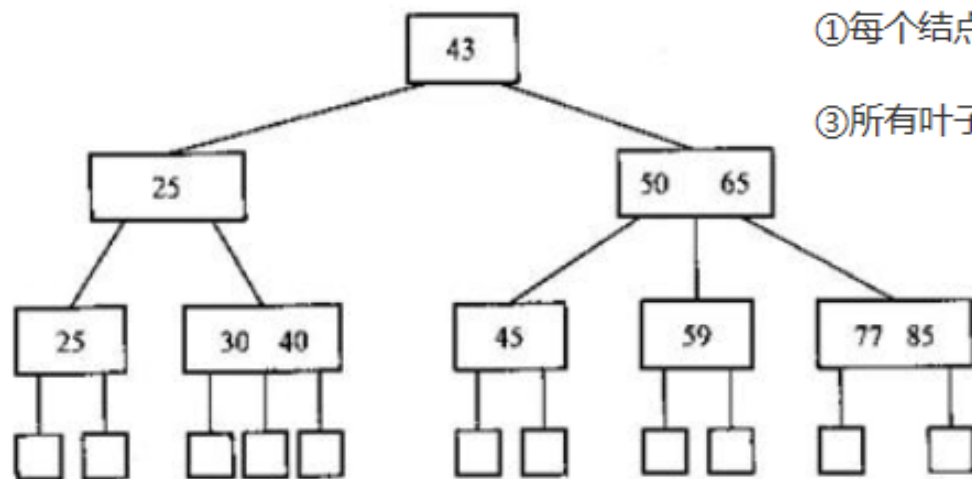


从该B树中删除关键码15后，该B树的第2层的结点数为_____。

下图所示为一棵M阶B-树，M最有可能的值为（ ）。



设有一3阶B树，如下图所示。



下面关于m阶B树说法正确的是()

- ①每个结点至少有两棵非空子树; ②树中每个结点至多有m—1个关键字;
③所有叶子在同一层上; ④当插入一个数据项引起B树结点分裂后, 树长高一层。

- (1) 在该B树上插入关键字35、80，画出两次插入后的B树。
(2) 从得到的B树上依次删除77、43，画出两次删除后的B树。

已知一个5阶B树有53个关键字，并且每个节点的关键字都达到最少状态，则它的深度是

- A. 3
B. 4
C. 5
D. 6

答案 b 根结点最少一个关键字 非根结点最少 $m/2$ 向上取整再-1个关键字 第一层: 1 第二层: $2*2$
第三层: $3*2*2$ 第四层: $3*6*2$ $1+2*2+3*2*2+3*6*2 = 53$, 正好四层

在高度为 5 的 4 阶 B 树中，所有结点关键字个数最少为 (②) 个。

- A. 64 B. 63 C. 32 D. 31

若元素的输入顺序为 (46, 79, 56, 38, 40, 84)，则利用堆排序方法建立的初始堆为 (③)。

- A. 79, 46, 56, 38, 40, 80 B. 38, 40, 56, 79, 46, 84

- C. 84, 79, 56, 46, 40, 38 D. 84, 56, 79, 40, 46, 38

在有向图 G 的拓扑序列中，若顶点 u 在顶点 v 之前，则下列情形不可能出现的是 (④)。

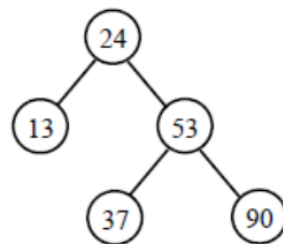
- A. G 中有弧 $\langle u, v \rangle$ B. G 中有一条从 u 到 v 的路径

- C. G 中没有弧 $\langle u, v \rangle$ D. G 中有一条从 v 到 u 的路径

假设在平衡二叉树中插入一个结点后造成了不平衡，若距插入点最近的不平衡祖先结点为 A，且 A 的左孩子的平衡因子为 1，右孩子的平衡因子为 0，则应作 (⑤) 型调整以使其平衡。

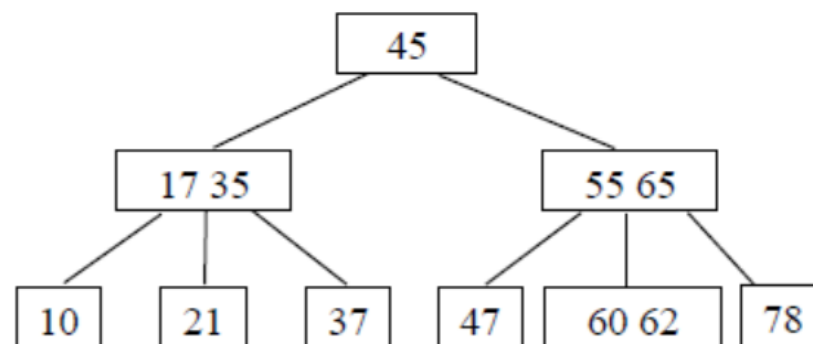
- A. LL B. LR C. RL D. RR

在下图所示的平衡二叉树中，插入关键字 48 后得到一棵新平衡二叉树。在新平衡二叉树中，关键字 37 所在结点的左、右子结点中保存的关键字分别是



- A. 13、48 B. 24、48 C. 24、53 D. 24、90

设有一棵 3 阶 B 树，如下图所示。删除关键字 78 得到一棵新 B 树，其最右叶结点所含的关键字是



- A. 60 B. 60, 62 C. 62, 65 D. 65

5.7 散列法——哈希 (Hash) 查找

散列法——地址散列法

被查找元素的存储地址 = Hash (Key)

【例5-18】 全国30个地区的人口统计，每个地区为一个记录，内容如下：

编号	地区名	总人口	汉族	回族	...
----	-----	-----	----	----	-----

$f1(key)$: 取关键字第一个字母在字母表中的序号；

$f2(key)$: 求第一和最后字母在字母表中序号之和，然后取30的余数；

$f3(key)$: 将第一个字母的八进制看成十进制，再取30的余数；

Key	BEIJING (北京)	TIANJIN (天津)	HEBEI (河北)	SHANXI (山西)	SHANGHAI (上海)	SHANGDONG (山东)	HENAN (河南)	SICHUAN (四川)
$f1(key)$	02	20	08	19	19	19	08	19
$f2(key)$	09	04	17	28	28	26	22	03
$f3(key)$	04	26	02	13	23	17	16	16

Hash查找的关键问题

①构造Hash函数

②制订解决冲突的方法

散列 (Hash) 函数 (表) 分类

内散列表 (数组)

外散列表 (链表)

具有不同关键字而具有相同散列地址的元素称为**同义词**，即 $key1 \neq key2$ ，但 $h(key1) = h(key2)$ 。由同义词引起的冲突称作**同义词冲突**。

构造**散列函数 (Hash)** 的几种方法：

- 直接定址法： $Hash(key) = key$ ；或 $Hash(key) = a \cdot key + b$ ；
- 质数除余法；
- 平方取中法；
- 折叠法；
- 数字分析法；

解决**冲突**的方法：

- 开放定址法
- 再散列法
- 链地址法

➤ 直接定址法: $\text{Hash}(key) = key$; 或 $\text{Hash}(key) = a \cdot key + b$;

【例5-19】: $\text{Hash}(key) = key$

地址	01	02	03	04	05	...	25	26	27	...	100
年龄	1	2	3	4	5	...	25	26	27	...	100
人数	3000	2000	5000			...	6000		
:											

【例5-20】 $\text{Hash}(key) = key + (-1949) + 1$

地址	01	02	03	04	05	...	25	26	27	...	100
年份	1949	1950	1951			...	1973			...	
人数	3000	2000	5000			...	6000		
:											

平方取中法

记录	key	key2	Hash
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741
Q3	2163	4 745 651	745

➤ 质数除余法

设桶数B, 取质数 $m \leq B$

$$\text{Hash}(key) = key \% m$$

➤ 平方取中法

取 key^2 的中间的几位数

➤ 例：散列地址

图书编号: 0-442-20586-

$$\begin{array}{r}
 4 \quad 5864 \qquad \qquad 5864 \\
 \quad 4220 \qquad \qquad 0224 \\
 + \qquad \qquad \qquad + \\
 \hline
 04 \qquad \qquad \qquad 04 \\
 10088 \qquad \qquad 6092
 \end{array}$$

左：移位叠加

$$\text{Hash}(key) = 0088$$

右：间界叠加

$$\text{Hash}(key) = 6092$$

➤ 数字分析法

如右图所示

假设散列表长为10010

可取中间4位中的两位为散列地址。

8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	5	4	1	5	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5

➤ 随机数法

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

构造Hash函数应注意以下几个问题：

- 计算Hash函数所需时间；
- 关键字的长度；
- 散列表的大小；
- 关键字的分布情况；
- 记录的查找频率；

解决冲突的几种方法：

➤ 开放定址法

$$H_i = (H(\text{key}) + d_i) \text{ MOD } m \quad i = 1, 2, \dots, k \quad (k \leq m-2)$$

$H(\text{key})$ 为哈希函数， m 为表长， d_i 为增量序列：

- (1) $d_i = 1, 2, 3, \dots, m-1$
- (2) $d_i = 12, -12, 22, -22, \dots, \pm k2 \quad (k \leq m/2)$
- (3) d_i 为伪随机序列，称为伪随机探测再散列

➤ 再散列法

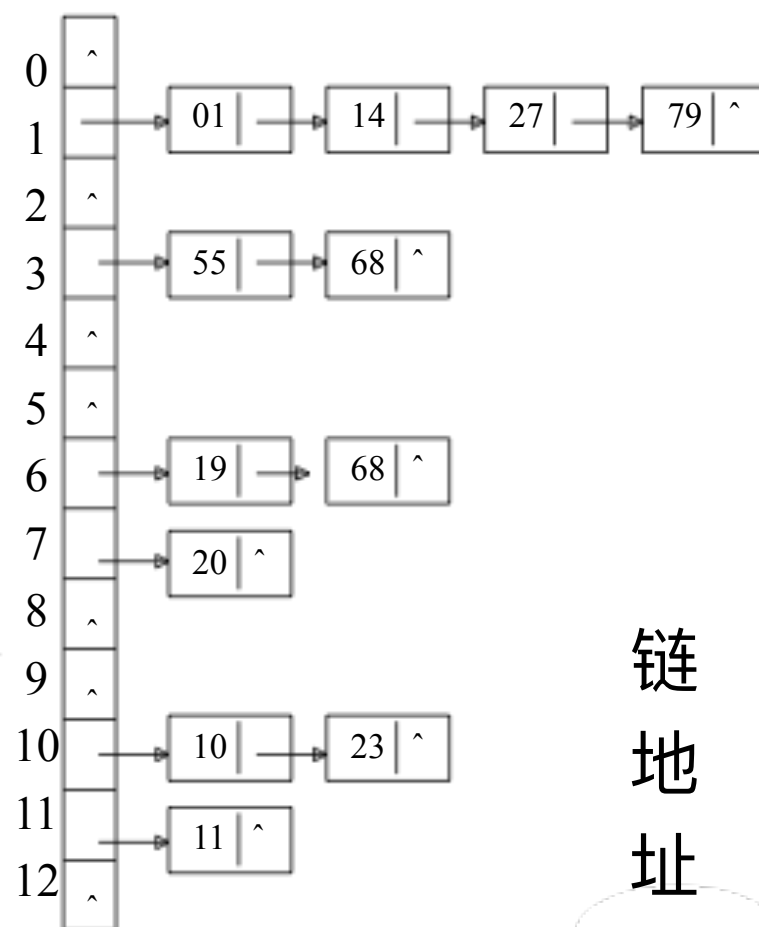
$$H_i = R_{hi}(\text{key}) \quad i = 1, 2, \dots, k$$

➤ 链地址法

例：(19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79)

$$H(\text{key}) = \text{key} \text{ MOD } 13$$

➤ 建立一个公共溢出区



装填因子： $\alpha = \frac{\text{表中装入的记录数}}{\text{哈希表的长度}}$

装填因子 α 标志着哈希表的装满程度， α 越小，发生冲突的可能性越小，反之，发生冲突的可能性越大。

成功查找平均查找长度： ASL_s

查找到散列表中已存在结点的平均比较次数。

失败查找平均查找长度： ASL_u

查找失败，但找到插入位置的平均比较次数。

几种处理冲突方法的平均查找长度

处理冲突方法	查找成功	查找失败（插入记录）
线性探测法	$\frac{1}{2}(1 + \frac{1}{1-\alpha})$	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
二次探测法 再散列法	$-\left(\frac{1}{\alpha}\right)\ln(1-\alpha)$	$\frac{1}{1-\alpha}$
链地址法	$1 + \frac{\alpha}{2}$	$\alpha + e^{-\alpha}$

【例5-21】将序列13, 15, 22, 8, 34, 19, 21插入到一个初始时为空的哈希表, 哈希函数采用 $H(x)=1+(x\%7)$:

- (1)使用线性探测法解决冲突;
- (2)使用步长为3的线性探测法解决冲突;
- (3)使用再哈希法解决冲突, 冲突时的哈希函数 $H(x)=1+(x\%6)$ 。

解: 设哈希表长度为8。

(1) 使用线性探测法解决冲突，即步长为1，对应地址为：

$$H(13)=1+(13\%7)=7;$$

$$H(15)=1+(15\%7)=2;$$

$$H(22)=1+(22\%7)=2(\text{冲突}), \quad H1(22)=(2+1)\%8=3;$$

$$H(8)=1+(8\%7)=2(\text{冲突}), \quad H1(8)=(2+1)\%8=3 \quad (\text{仍冲突})$$

$$H2(8)=(3+1)\%8=4;$$

$$H(34)=1+(34\%7)=7(\text{冲突}), \quad H1(34)=(7+1)\%8=0;$$

$$H(19)=1+(19\%7)=6;$$

$$H(21)=1+(21\%7)=1;$$

哈希表

地址	0	1	2	3	4	5	6	7
Key	34	21	15	22	8		19	13
探测次数	2	1	1	2	3		1	1

(2) 使用步长为3的线性探测法解决冲突，对应地址为：

$$H(13)=1+(13\%7)=7;$$

$$H(15)=1+(15\%7)=2;$$

$$H(22)=1+(22\%7)=2(\text{冲突}), \quad H1(22)=(2+3)\%8=5;$$

$$H(8)=1+(8\%7)=2 \quad (\text{冲突}), \quad H1(8)=(2+3)\%8=5 \quad (\text{仍冲突})$$

$$H2(8)=(5+3)\%8=0;$$

$$H(34)=1+(34\%7)=7(\text{冲突}), \quad H1(34)=(7+3)\%8=2 \quad (\text{仍冲突})$$

$$H2(34)=(2+3)\%8=5 \quad (\text{仍冲突})$$

$$H3(34)=(5+3)\%8=0 \quad (\text{仍冲突})$$

$$H4(34)=(0+3)\%8=3;$$

$$H(19)=1+(19\%7)=6;$$

$$H(21)=1+(21\%7)=1;$$

哈希表

地址	0	1	2	3	4	5	6	7
key	8	21	15	34		22	19	13
探测次数	3	1	1	5		2	1	1

(3) 使用再哈希法解决冲突，再哈希函数为 $H(x)=1+(x\%6)$

$$H(13)=1+(13\%7)=7;$$

$$H(15)=1+(15\%7)=2;$$

$$H(22)=1+(22\%7)=2(\text{冲突}),$$

$$H(8)=1+(8\%7)=2(\text{冲突}),$$

$$H(34)=1+(34\%7)=7(\text{冲突}),$$

再哈希函数
 $\Rightarrow 1+(x\%7).$

$$H_1(22)=1+(22\%6)=5;$$

$$H_1(8)=1+(8\%6)=3;$$

$$H_1(34)=1+(34\%6)=5(\text{仍冲突})$$

$$H_2(34)=1+(34\%5)=5(\text{仍冲突})$$

$$H_3(34)=1+(34\%4)=3(\text{仍冲突})$$

$$H_4(34)=1+(34\%3)=2(\text{仍冲突})$$

$$H_5(34)=1+(34\%2)=1;$$

$$H(19)=1+(19\%7)=6;$$

$$H(21)=1+(21\%7)=1(\text{冲突}), \quad H_1(21)=1+(21\%6)=4;$$

哈希表

地址	0	1	2	3	4	5	6	7
Key		34	15	8	21	22	19	13
探测次数		6	1	2	2	2	1	1

【例5-22】 已知一组记录的键值为{1,9,25,11,12,35,17,29},
请构造一个散列表。

(1) 采用除留余数法构造散列函数, 线性探测法处理冲突, 要求新插入键值的平均探测次数不多于2.5次, 请确定散列表的长度m及相应的散列函数。分别计算查找成功和查找失败的平均查找长度;

(2) 采用 (1) 中的散列函数, 但用链地址法处理冲突, 构造解: 散列表, 分别计算查找成功和查找失败的平均查找长度。

因为 $8/m \leq 1/2$, 所以 $m \geq 16$

取 $m=16$, 散列函数为 $H(\text{key})=\text{key}\%13$

给定键值序列为: {1,9,25,11,12,35,17,29}

散列地址为: d:1,9,12,11,12,9,4,3

用线性探测解决冲突, 见下表:

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key		1		29	17					9	35	11	25	12		
成功探测次数		1		1	1					1	2	1	1	2		
失败探测次数	1	2	1	3	2	1	1	1	1	6	5	4	3			

查找成功的平均查找长度：

$$ASL_s = (6+4)/8 = 1.25$$

查找失败的平均查找长度：

$$ASL_u = (6 \times 1 + 2 + 3 + 2 + 6 + 5 + 4 + 3) / 13 = 2.4$$

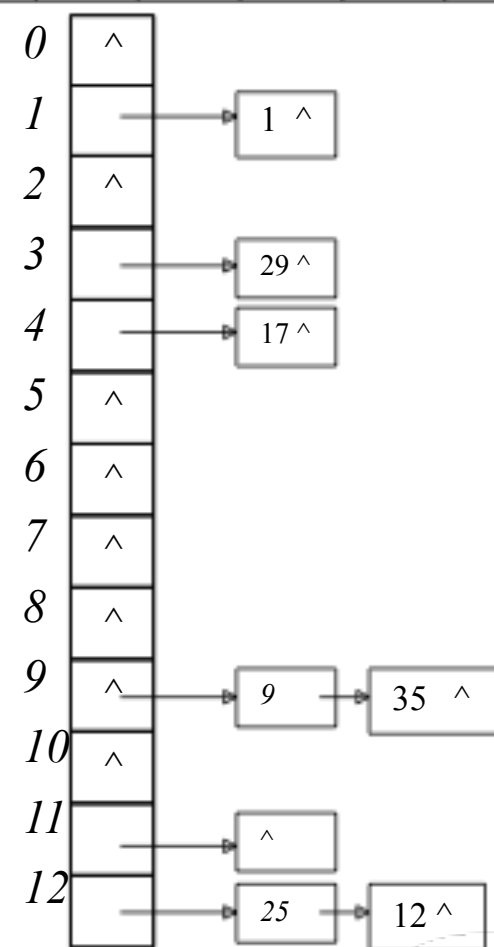
(2)使用链地址法处理冲突构造散列表。

查找成功的平均查找长度：

$$ASL_s = (6+4)/8 = 1.25$$

查找失败的平均查找长度：

$$ASL_u = (4 \times 1 + 2 \times 2) / 13 \approx 2.4$$



【例5-23】 将关键字序列 (7, 8, 30, 11, 18, 9, 14) 散列存储到散列表中，散列表的存储空间是一个下标从0开始的一维数组，散列函数为： $H(\text{key}) = (\text{key} \times 3) \text{ MOD } 7$ ，处理冲突采用线性探测再散列法，要求装填（载）因子为0.7。

- (1) 请画出所构造的散列表；
- (2) 分别计算等概率情况下查找成功和查找不成功的平均查找长度。

下标	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	

查找成功的平均查找长度：ASL成功=12/7

查找不成功的平均查找长度：ASL不成功=18/7

关键字	7	8	30	11	18	9	14
散列地址	0	3	6	5	5	6	0
探查次数	1	1	1	1	3	3	2

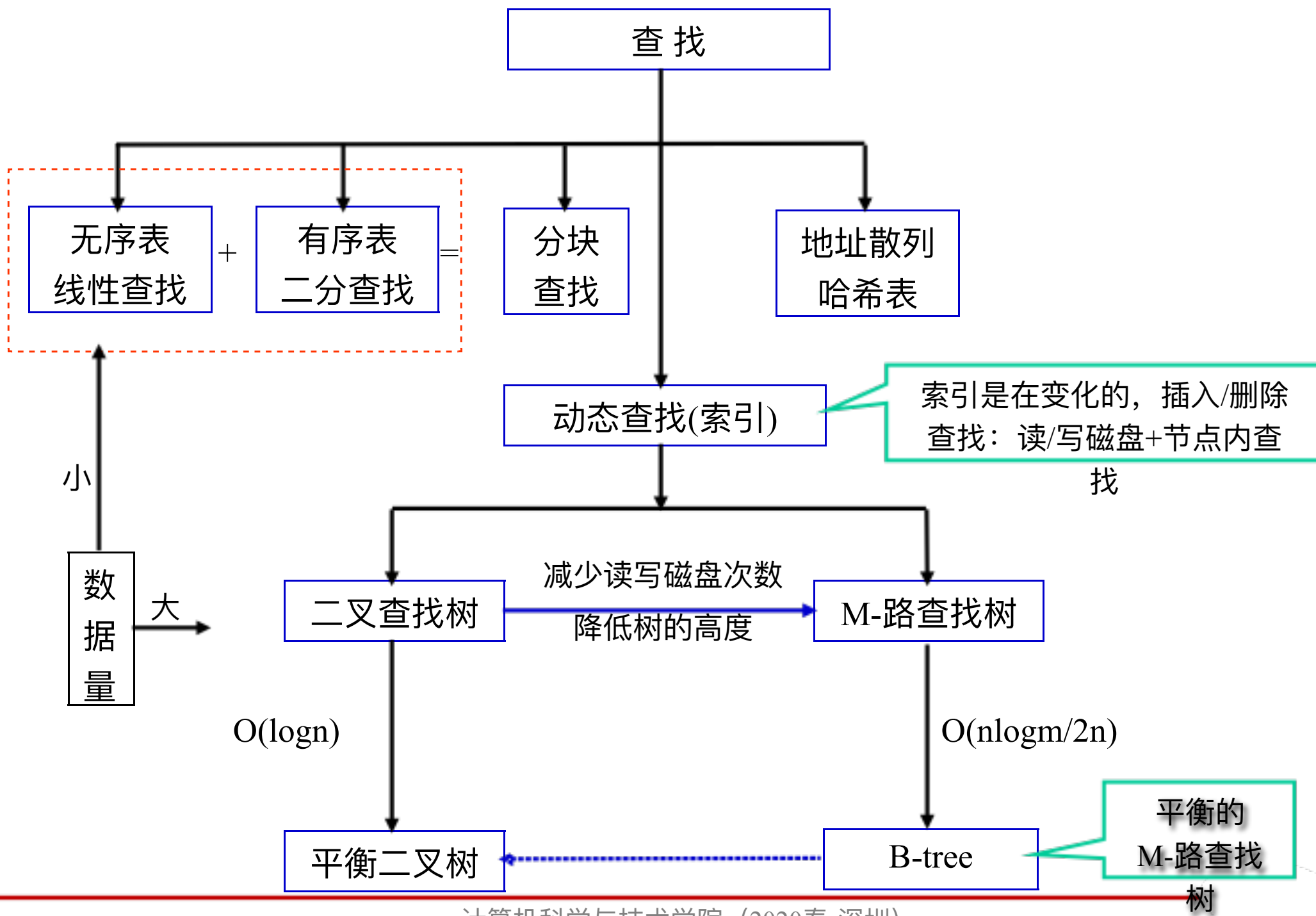
下标	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	

$$ASL_{成功} = (1 \text{ 次} \times 4 + 2 \text{ 次} \times 1 + 3 \text{ 次} \times 2) / 7 = 12/7$$

各位置对应的探查次数如下表所示：

下标	0	1	2	3	4	5	6
探查次数	3	2	1	2	1	5	4

$$ASL_{不成功} = (3 + 2 + 1 + 2 + 1 + 5 + 4) / 7 = 18/7$$



B-tree (Balance-tree)

Rudolf Bayer, Edward M. McCreight(1970)写的一篇文章《[Organization and Maintenance of Large Ordered Indices](#)》中首次提出。

Definition[edit]

According to Knuth's definition, a B-tree of order m is a tree which satisfies the following properties:

1. Every node has at most m children.
2. Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ children.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains $k-1$ keys.
5. All leaves appear in the same level.

B-tree是一种平衡的 m -路查找树。

B树的特性：

- (1) 关键字集合分布在整棵树中；
- (2) 任何一个关键字出现且只出现一个结点中；
- (3) 搜索有可能在叶子结点结束；
- (4) 其搜索性能等价于在关键字全集内
做一次二分查找；
- (5) 自动层次控制。

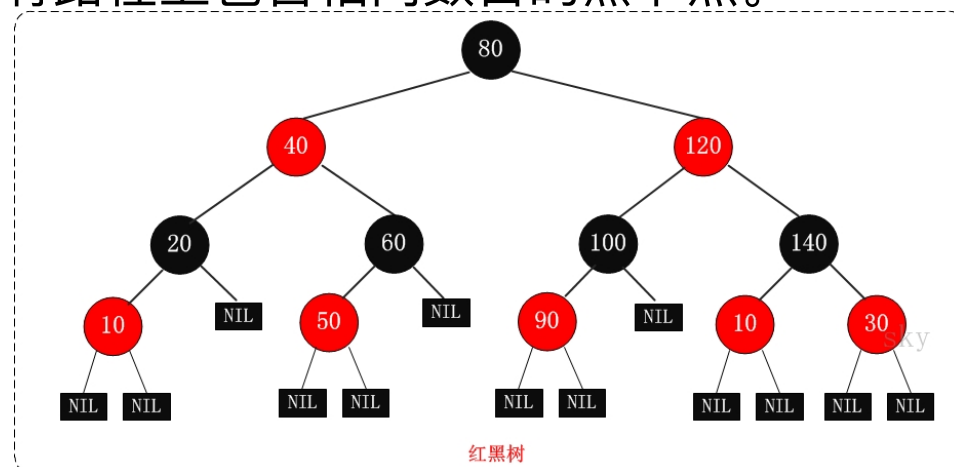
红-黑树 (R-B Tree)

全称是Red-Black Tree，又称为“红黑树”，它是一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树是一种近似平衡的二叉查找树，能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一陪。红黑树的特性：

- (1) 每个节点或者是黑色，或者是红色。
- (2) 根节点是黑色。
- (3) 每个叶子节点 (NIL) 是黑色。
- (4) 如果一个节点是红色的，则它的子节点必须是黑色的。
- (5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是 $O(\lg n)$ 。



几种查找树的应用：

“高度严格”的平衡二叉树
理想化的概念

AVL树: 最早的平衡二叉树之一。应用相对其他数据结构比较少。windows对进程地址空间的管理用到了AVL树。

红黑树: 平衡二叉树，广泛用在C++的STL中。如map和set都是用红黑树实现的。

B/B+树: 用在磁盘文件组织 数据索引和数据库索引。

Trie树(字典树): 用在统计和排序大量字符串，如自动机。

“大致上”平衡树
是AVL的具体应用