数据结构与算法

Data Structures and Algorithms

第五部分 查找

数据结构考查内容

一、线性表
（一）线性表的基本概念
（二）线性表的实现
（三）线性表的应用

二、栈、队列和数组
（一）栈和队列的基本概念
（二）栈和队列的顺序存储结构
（三）栈和队列的链式存储结构
（四）多维数组的存储
（五）特殊矩阵的压缩存储
（六）栈、队列和数组的应用

三、树与二叉树
（一）树的基本概念
（二）二叉树
（三）树、森林
（四）树与二叉树的应用

四、图
（一）图的基本概念
（二）图的存储及基本操作
（三）图的遍历
（四）图的基本应用

数据结构

五、查找
（一）查找的基本概念
（二）顺序查找法
（三）分块查找法
（四）折半查找法
（五）B树及其基本操作，B+树的基本概念
（六）散列（Hash）表
（七）字符串模式匹配
（八）查找算法分析及应用

六、排序
（一）排序的基本概念
（二）插入排序
（三）起泡排序
（四）简单选择排序
（五）希尔排序
（六）快速排序
（七）堆排序
（八）二路归并排序
（九）基数排序
（十）外部排序

算 法

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。

- 查找分类
  - 传统的查找方法
    - 线性查找
    - 折半查找
    - 分块查找
    - 二叉查找树
  - 较先进的查找方法（散列法,哈希表）

重点：顺序查找、二分(折半)查找、二叉排序树查找以及散列表构造和散列方法实现。

难点：二叉排序树的删除算法和平衡二叉树的构造算法。

几个概念：

- ✓ 查找表：由同一类型的数据元素（或记录）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；
- ✓ 查找方式：根据查找方法取决于记录的键值还是记录的存储位置？
  - 基于关键字比较的查找
    - ◆ 顺序查找、折半查找、分块查找、BST&AVL、B-树 和B+树
  - 基于关键字存储位置的查找
    - ◆ 散列法
- ✓ 根据被查找的数据集合存储位置
  - 内查找：整个查找过程都在内存进行；
  - 外查找：若查找过程中需要访问外存，如B树和B+树
- ✓ 根据查找方法是否改变数据集合？

静态查找：查找+提取数据元素属性信息，被查找后数据并不改变。

动态查找：查找+（插入或删除元素），被查找后数据可能改变，可以插入/删除记录。

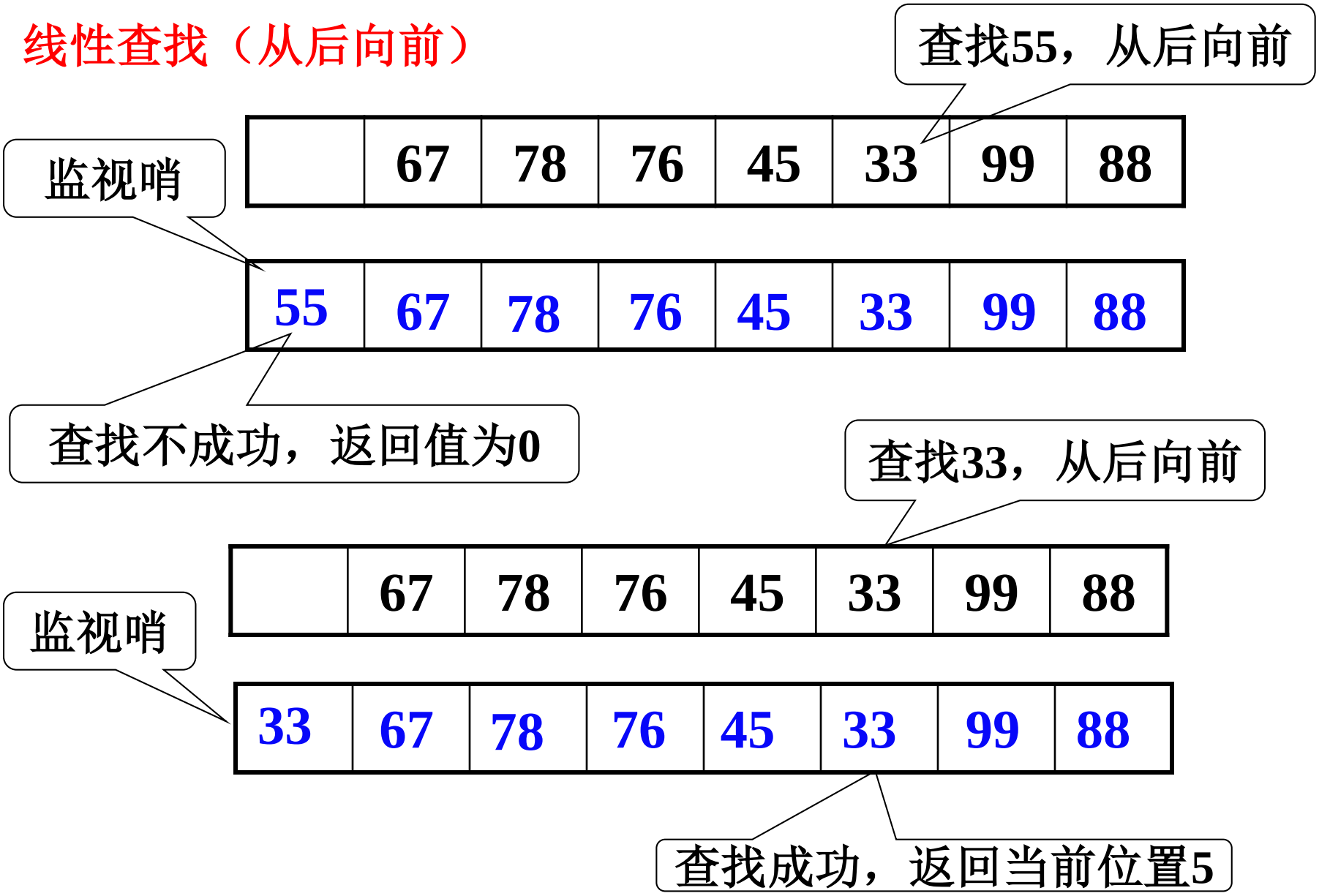
5.1 线性查找

【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 ;
```

线性查找（从后向前）


查找55，从后向前
监视哨
67 78 76 45 33 99 88
55 67 78 76 45 33 99 88
查找不成功，返回值为0
查找33，从后向前
监视哨
67 78 76 45 33 99 88
33 67 78 76 45 33 99 88
查找成功，返回当前位置5

【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：对于查找通常不采用时间复杂度分析，而是“比较”作为衡量算法的主要指标。

【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为关键字的比较次数，n为表长。

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

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

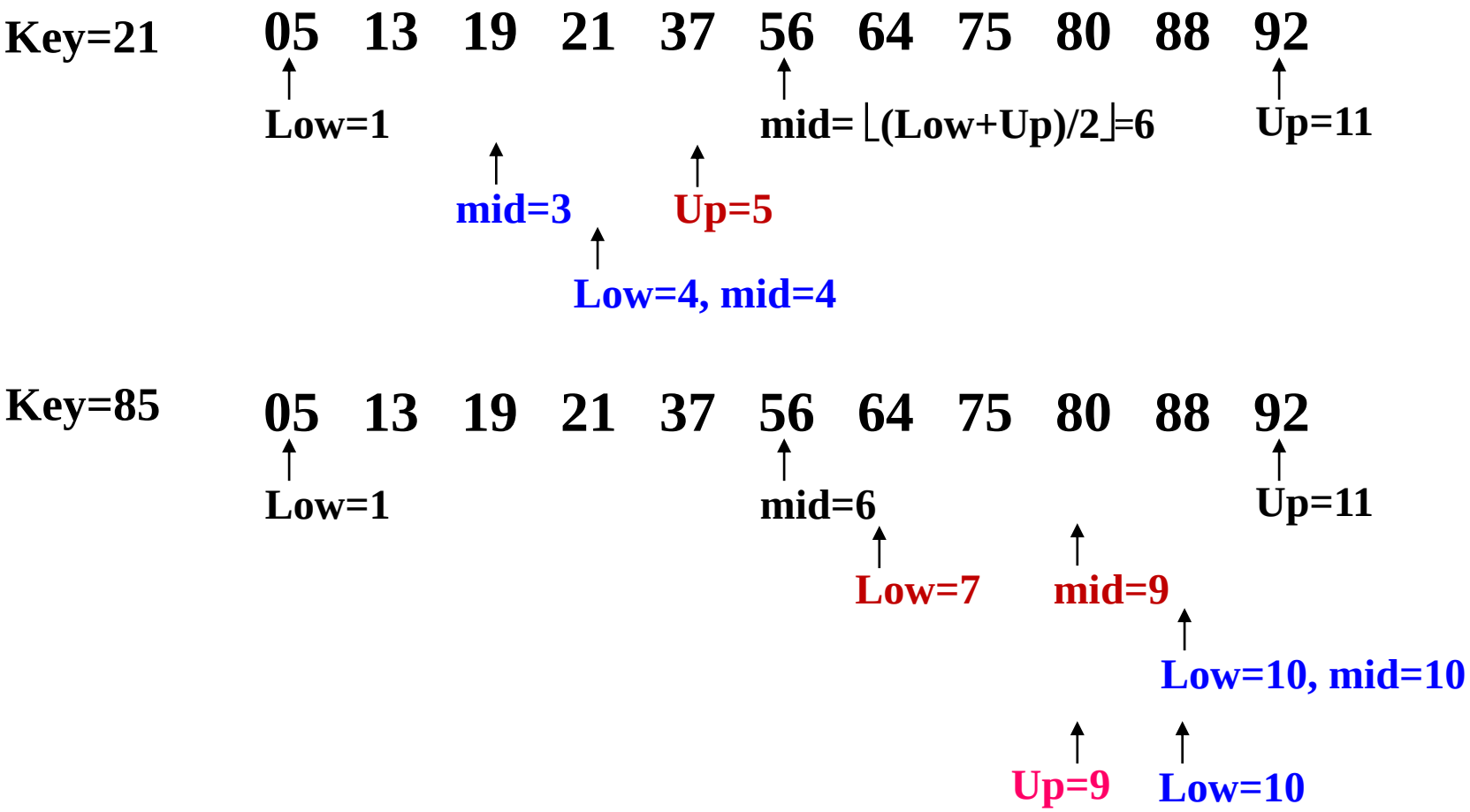
$$ASL = nP_1+(n-1)P_2+\ldots+2P_{n-1}+P_n$$

$$= 1/n \cdot \sum_{i=1}^{n}(n-i+1)$$

$$= (n+1)/2$$

5.2 折半查找

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

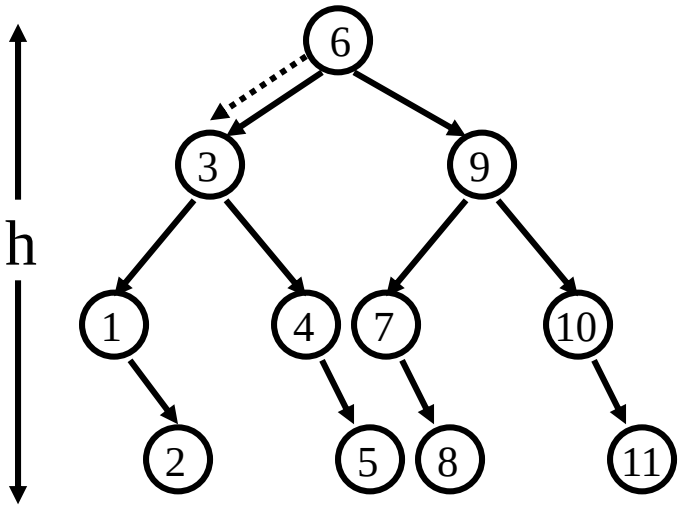


折半查找算法

```
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



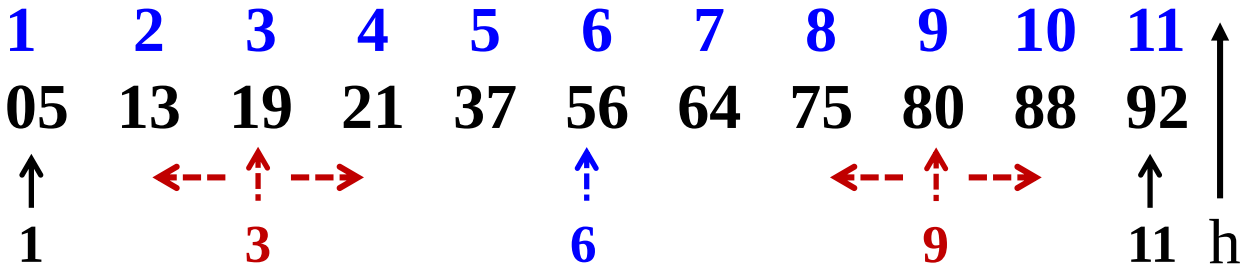
$n=2^h-1 \quad (h=\log_2(n+1))$

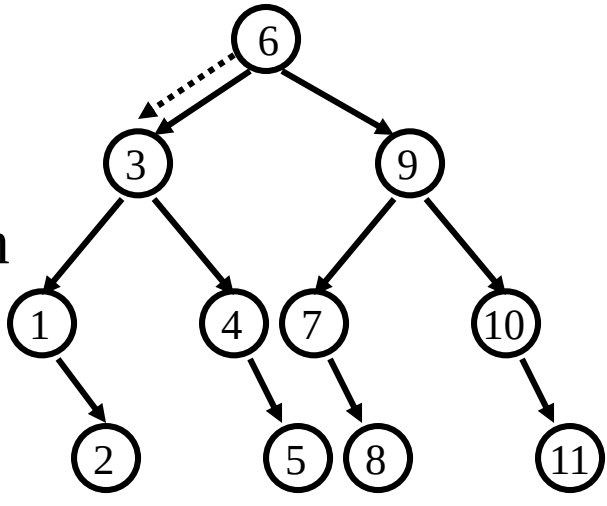
$$ASL_{bs}=\sum_{i=1}^{n} P_i \cdot C_i \qquad P_i=1/n$$

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

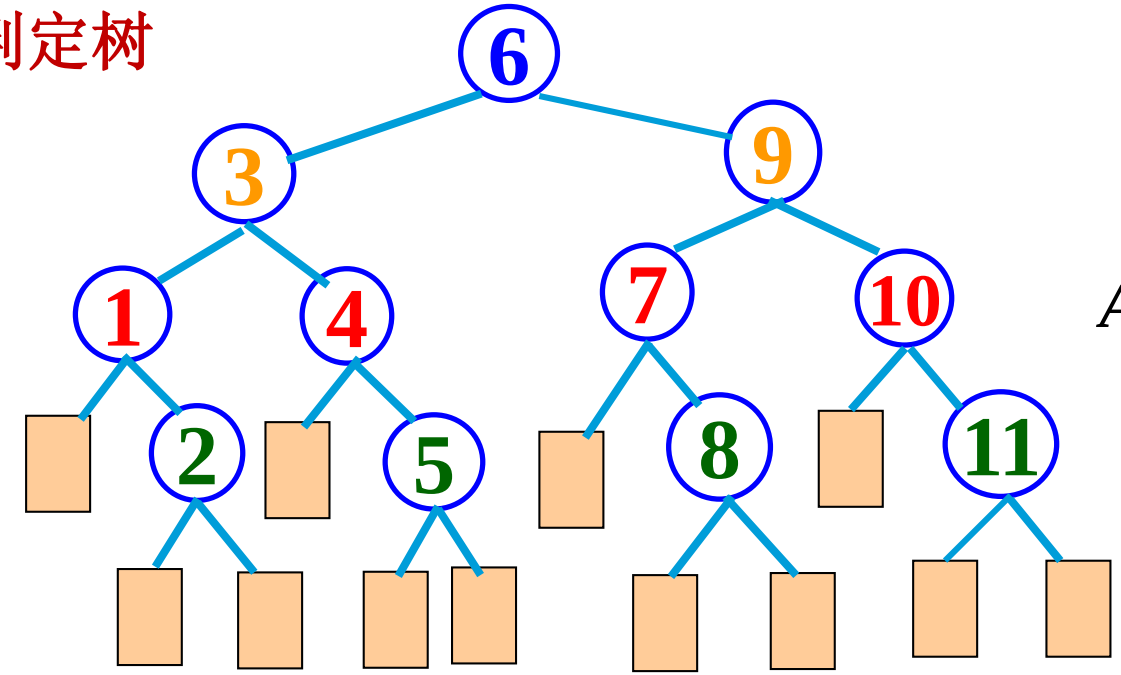
$$=(n+1)/n\cdot\log_2(n+1)-1$$

$$ASL_{bs}\approx\log_2(n+1)-1$$





判定树



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

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

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

$$\approx \log_2(n+1)-1$$

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4

```
int  Bsearch( F , i , j , k )                    //折半查找的递归算法
{   int m;
    if( i > j ) return( -1 ); //判断 i, j 是否合法
    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) );
           }
}
```

调用： Bsearch(F,1,n,k)

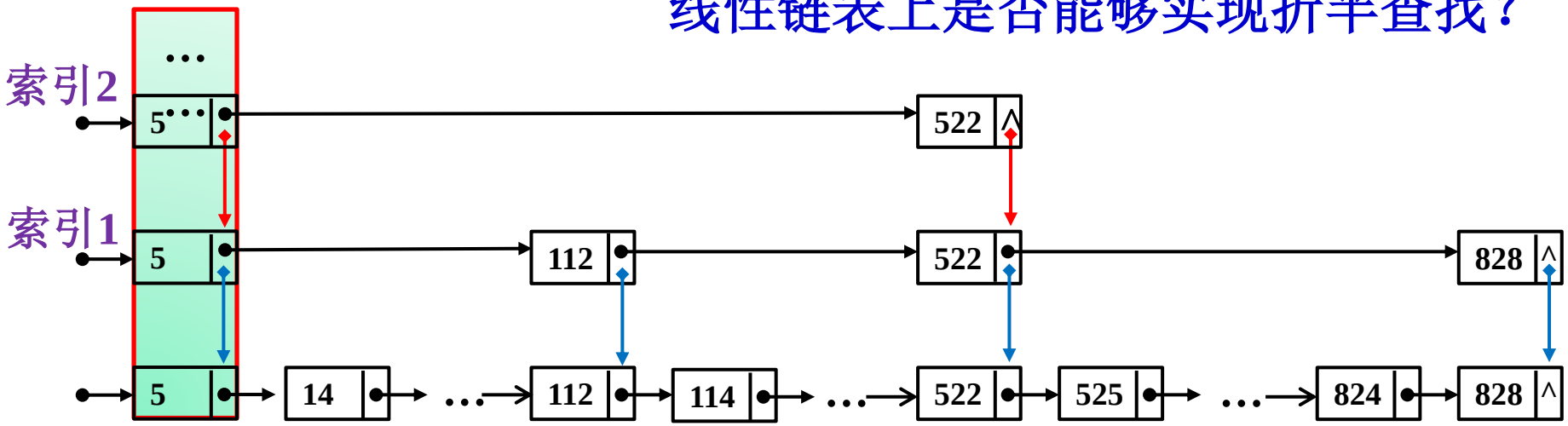
顺序表和有序表上查找算法对比：

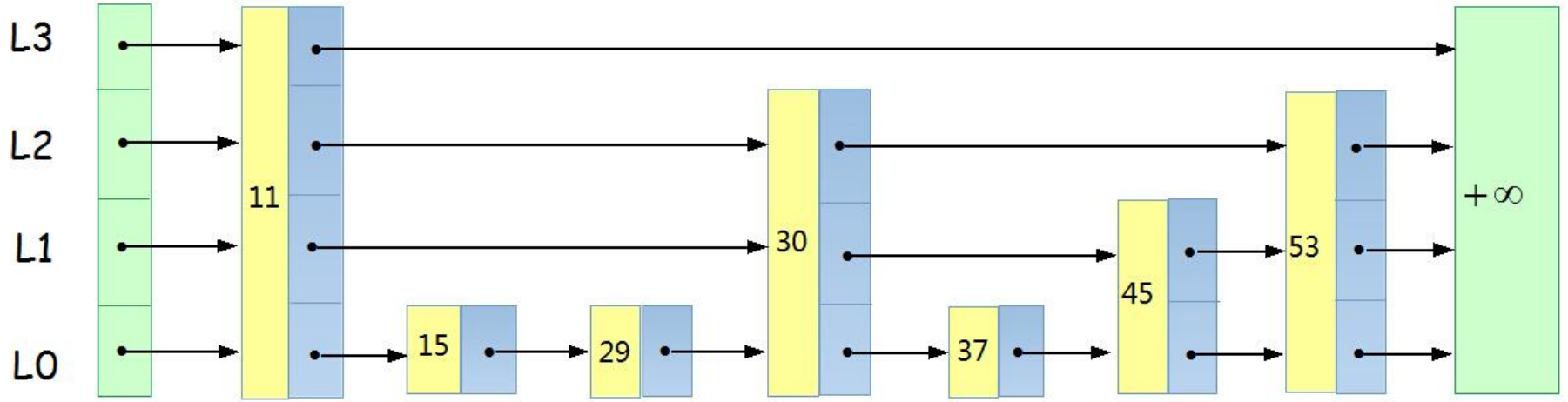
	顺序表	有序表
表的特征	表中元素无序	表中元素有序
存储结构	顺序和链表结构	顺序结构
查找方法	顺序查找	折半查找
插删操作	方便	移动元素
ASL	大	小
适用范围	表长较短	表长较长 有序

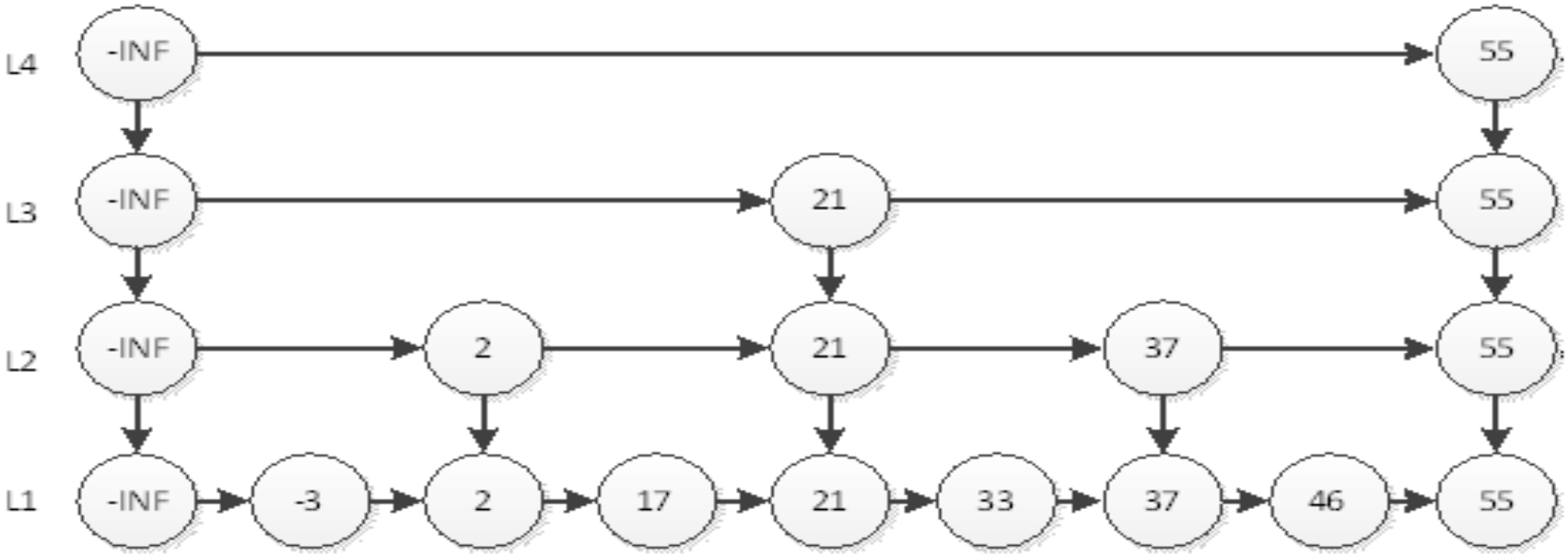
思考题：

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，一个数num的算术平方根sqrt一定在0~num/2之间，并且满足sqrt=num/sqrt，可以利用二分查找在0~num/2之间查找sqrt。

线性链表上是否能够实现折半查找？







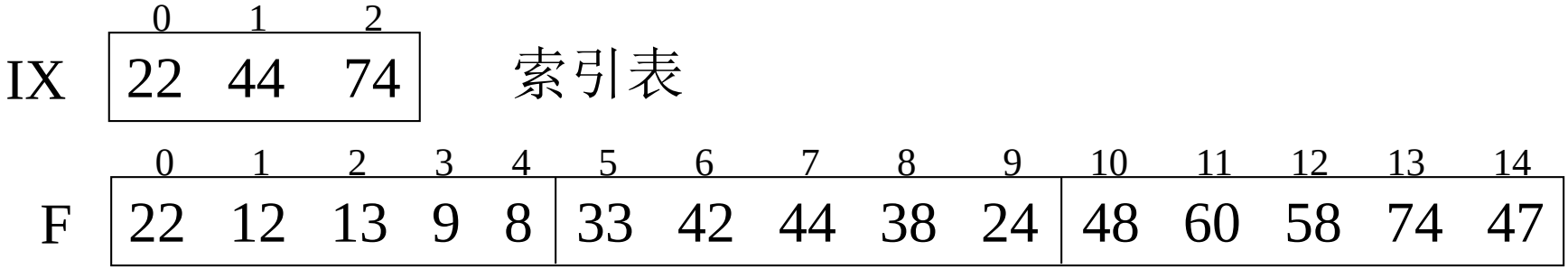
Skip List

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(logn)；**

5.3 分块查找（线性查找+折半查找）



建立要求：

(1) 查找表要求顺序存储

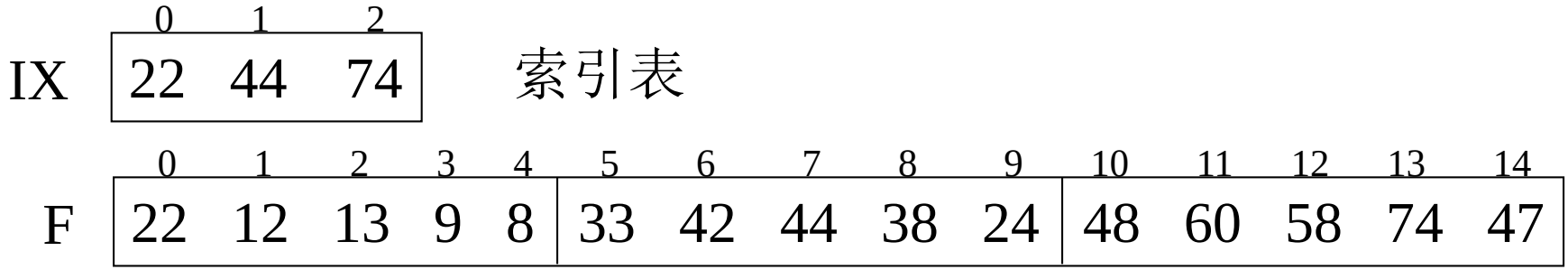
(2)查找表分成n块，当块序号i > j时，第i块中的最小元素大于第j块中的最大元素。

查找思想：

(1) 首先确定所要查找关键字在哪一块中。

(2) 在所确定的块中用顺序查找查找关键字。

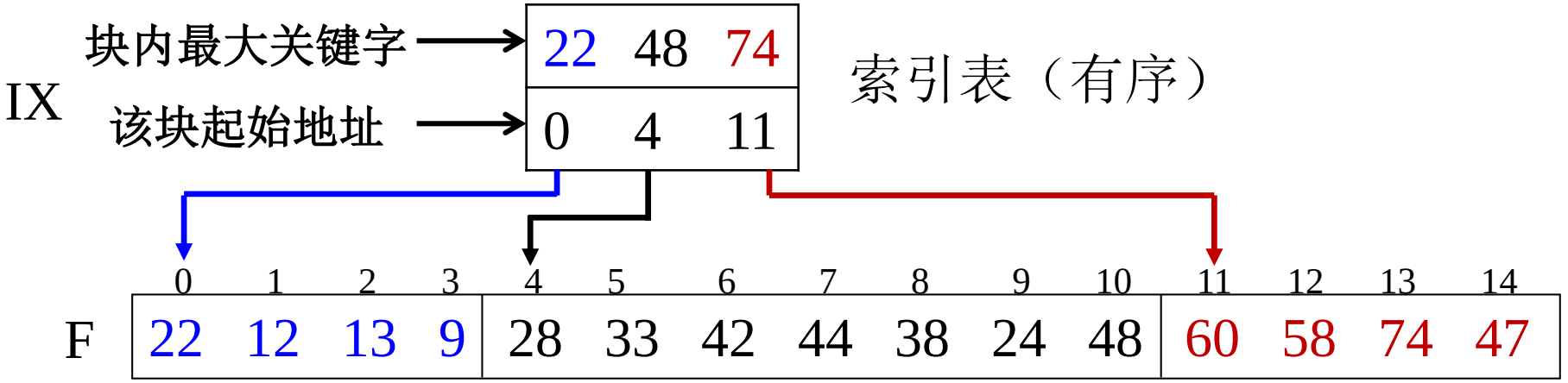
分块查找的过程是一个“缩小区间”的查找过程。




```
Int   index_search(F, k,  last, blocks, ix, L)
//查找表；查找的关键字；查找表总长；块数；索引表；每块长度
keytype  k ; int  last, blocks; index ix; LIST  F ; int L ;
{   int  i, j ;
    i = 0; //块号从0开始
    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 ( j == last )          return -1;
        if ( k == F[ j ].key )  return j ;
     }
    return –1 ;
}
```

Typedef keytype index[maxblock]

分块查找



线性表：块内可无序、块大小可不一致(可以顺序存储和链式存储)

```
typedef struct
{
    keyType key;
    int addr;
}indexType;
```

```
typedef struct
{
   indexType index[maxblock];
   int block;
} indexTable;
indexTable ix;
```

```
int SEARCH(List F, int last, indexTable IX, keyType key )
{
  int i = 0, j, e;
  while (( key > IX.index[i].key )&&(i < IX.block)) //查找块号
          i++;
  if ( i<IX.block )
  {
     j = IX.index[i].addr; //当前块起始位置
    if ( i != IX.block )     e = IX.index[i+1].addr-1;   //当前块结束位置
    while ( key!=F[j].key && j<=e && (j<last) ) //查找
             j=j+1;
    if ( j == last ) return -1;
    if ( k == F[ j ].key )  return j ;
   }
  return -1
}
```

例 已知一个长度为16的顺序表L，其元素关键字有序排列，若采用折半查找一个L中不存在的元素，则关键字的比较次数最多是（B）

A. 4 B. 5 C. 6 D. 7

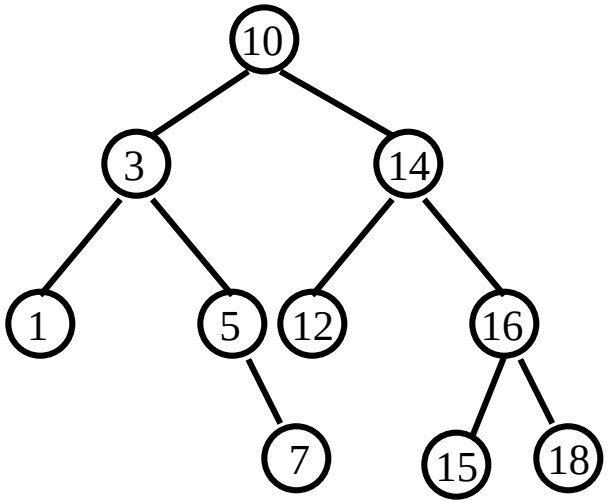
例 如果索引表长度为b，每块平均长度为L，均以线性查找的平均查找长度是多少？

(b+1)/2+(L+1)/2

例 长度为n的线性表，分成多少块平均查找长度数最少？

$\sqrt{n}$

5.4 二叉查找树(二叉排序树)

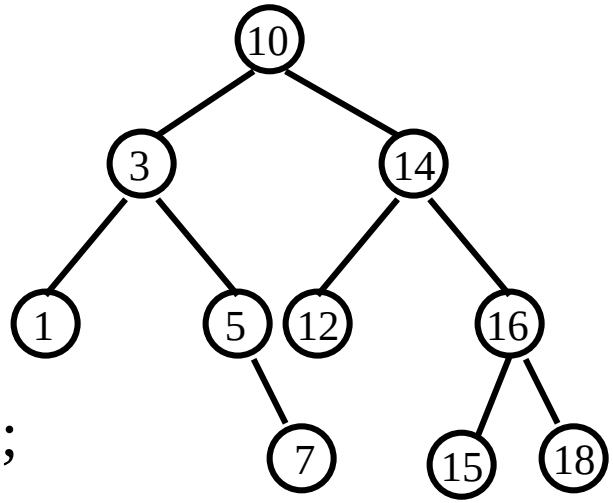


二叉查找树是一种特殊的二叉树，又称二叉排序树。二叉排序树要么是一棵空树，要么满足如下性质的非空二叉树：

- 对于每个结点，如果其左子树非空，则左子树的所有结点的关键值都小于该(根)结点的关键值；
- 如果其右子树非空，则右子树的所有结点的关键值都大于该(根)结点的关键值。
- 左、右子树本身又是一棵二叉排序树。

简单地讲：左子树结点值 < 根结点值 < 右子树结点值
递归定义的数据结构

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



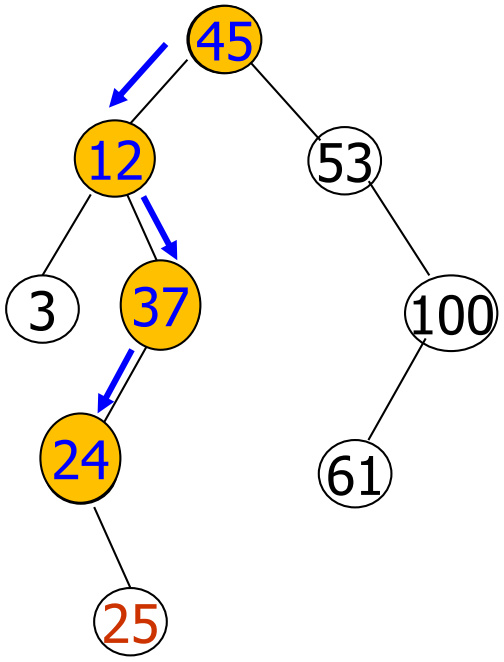
1、查找

```
BST   search( keyType  k, BST  F );
{   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 ) ) ;
}
```

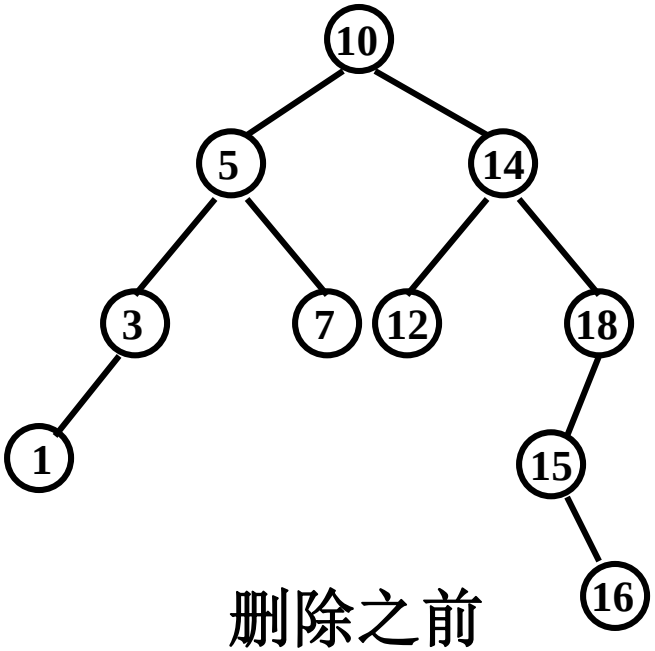
2、在二叉查找树中插入新结点

```
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 )
}
```

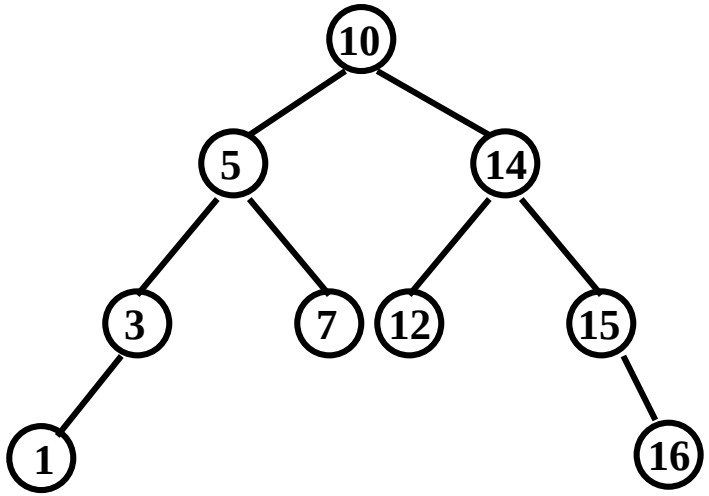
插入的位置只发生在叶子结点的位置



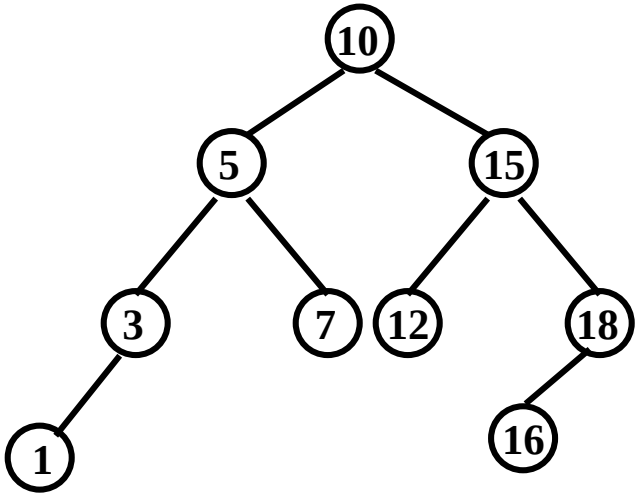
3、在二叉查找树中删除结点



删除之前



删除18之后



删除14之后

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

四种删除的方式(删除结点p)

(1) p为叶结点

释放结点p，修改其双亲结点f的相应指针。

(2) p的右子树空

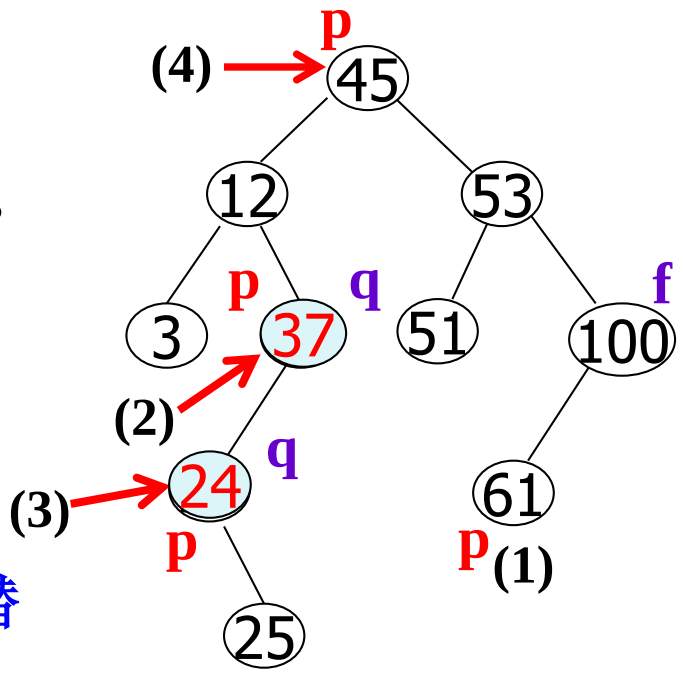
p的右子树空，则用左子树顶替

q = p; p = p->lchild; free(q);

(3) p的左子树为空

删除方法： p的左子树空，则用右子树顶替

q = p; p = p->rchild; free(q);



(4) p的左右子树都非空

把左子树作为右子树中最小结点的左子树。

或者 把右子树作为左子树中最大结点的右子树。

替换的策略：

- 用左子树中最大的结点（前驱）替换；
- 右子树中最小的结点（后继）替换；

树高增加了，怎么办？

```
q = p; s = p->lchild;
while (!s->rchild)
{ q = s; s = s->rchild; }
p->data = s->data;
q->rchild = s->lchild;
```

二叉排序树的结点删除算法-非递归算法

```
Status Delete(BST &p)
 { BST q, s;
   if (!p->rchild)                    //右子树为空，把左子树挂上
      {q = p;  p = p->lchild;  free(q); }
     else if (!p->lchild)             //左子树为空，把右子树挂上
      {q = p;  p = p->rchild;  free(q);}
     else {                           //左右子树均不空
     q = p;  s = p->lchild;           //找在左子树的中序前驱
     while (!s->rchild)               //前驱在左子树的最右下
          {q = s;  s = s->rchild; }   //顺着右子树往下走
           p->data = s->data;
           if (q != p) q->rchild = s->lchild;
           else q->lchild = s->lchild;
           free(s);
          }
  return TRUE;
} // Delete
```

二叉查找树删除结点(递归)

在F所指的当前非空二叉树中关键字最小的结点，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 )
{  右子树最左边结点来替换（中序后继）
  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;// min可设置成极小数
    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
}
```

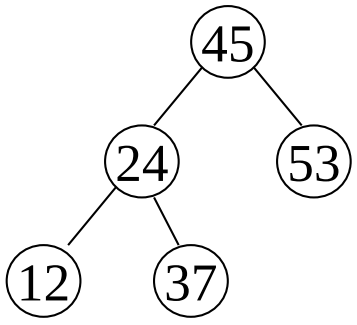
二叉排序树的优点

◆用二叉排序树作为目录树，把一个记录的关键码和记录的地址作为二叉排序树的结点，按关键码值建成二叉排序树。

◆能像有序表那样进行高效查找；

◆能像链表那样灵活插入、删除。

二叉排序树的查找算法分析



平均查找长度

$O(\log_2 n)$

ASL = （1+2+2+3+3）/ 5
= 2.2

平均查找长度

O(n)

ASL = （1+2+3+4+5）/ 5
= 3

如何来保证平均查找长度是$O(\log_2 n)$呢？

平衡二叉树

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·阿德尔森-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。

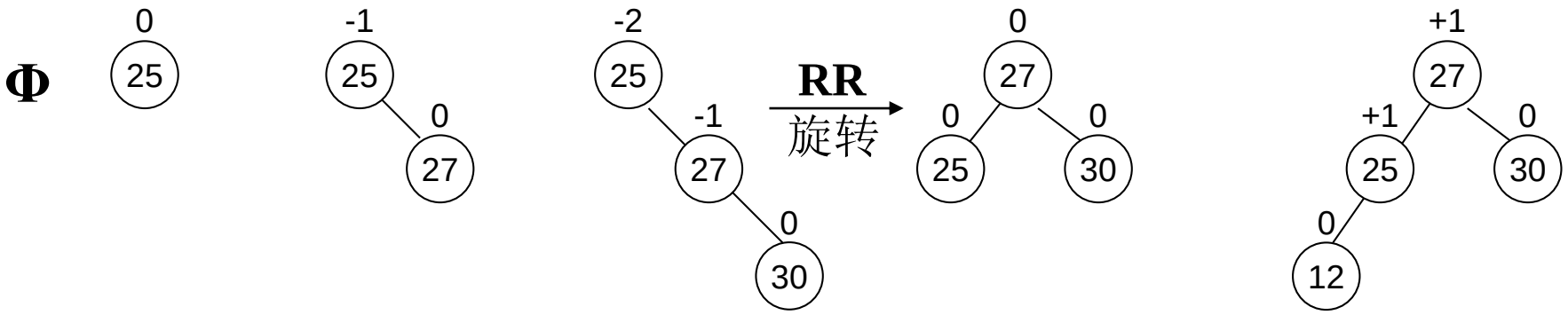
【定义】AVL树或者是一棵空二叉树，或者具有如下性质的二叉查找树：

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。

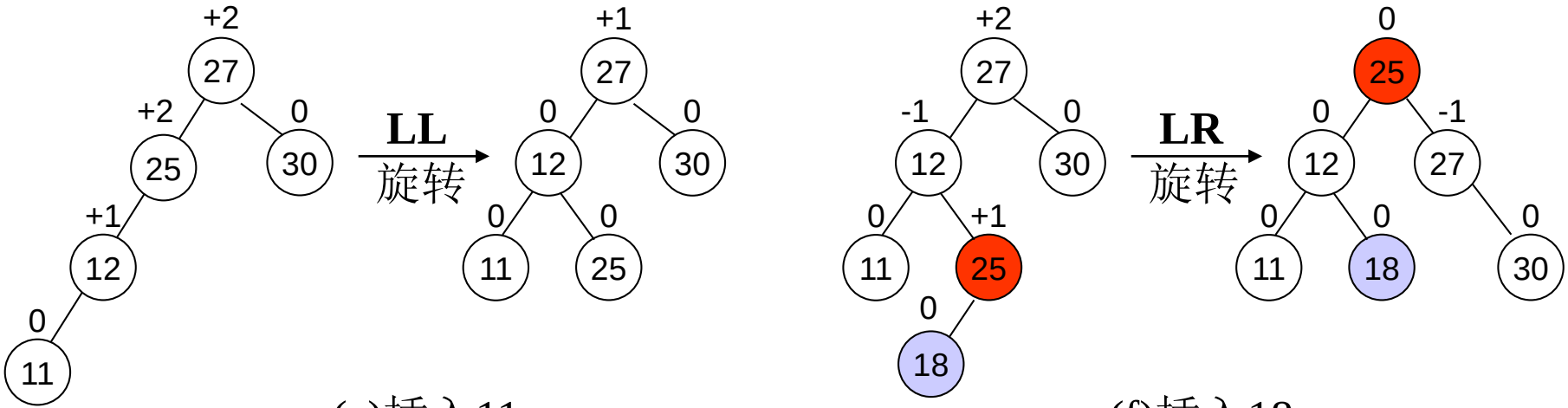
结点的平衡因子 BF (Banlanced Factor) 定义为：结点左子树与右子树的高度之差。

AVL中的任意结点的 BF 只可能是 -1，0，+1

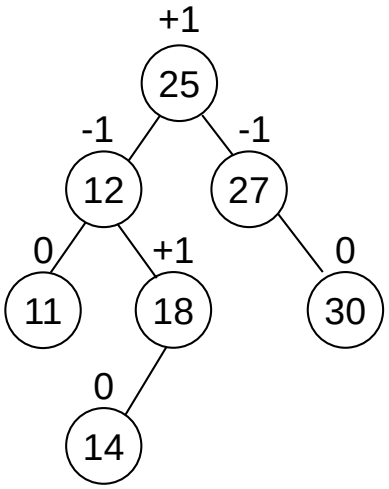
输入={25, 27, 30, 12, 11, 18, 14, 20, 15, 22}



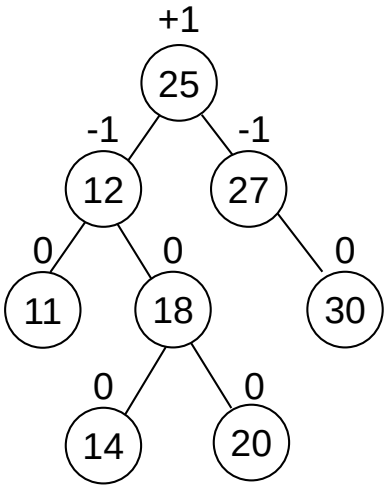
(a)插入25 (b)插入27 (c)插入30 (d)插入12



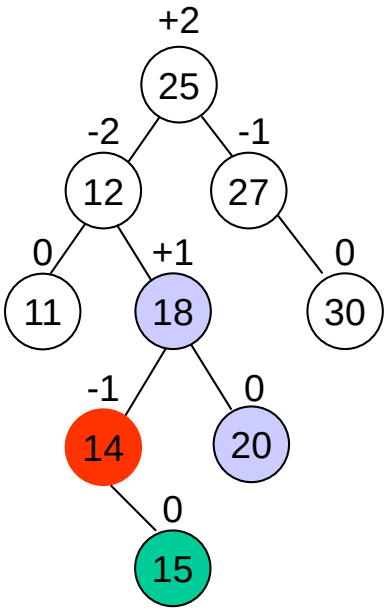
(e)插入11 (f)插入18



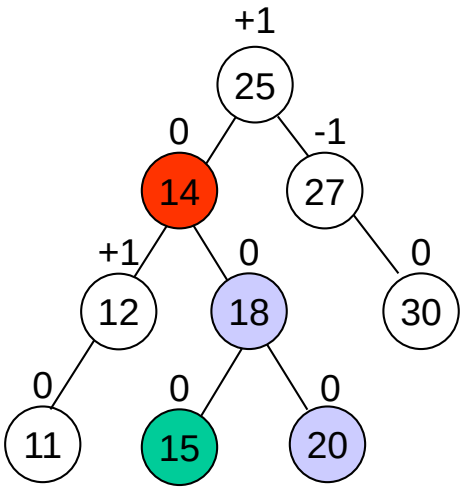
(g)插入14



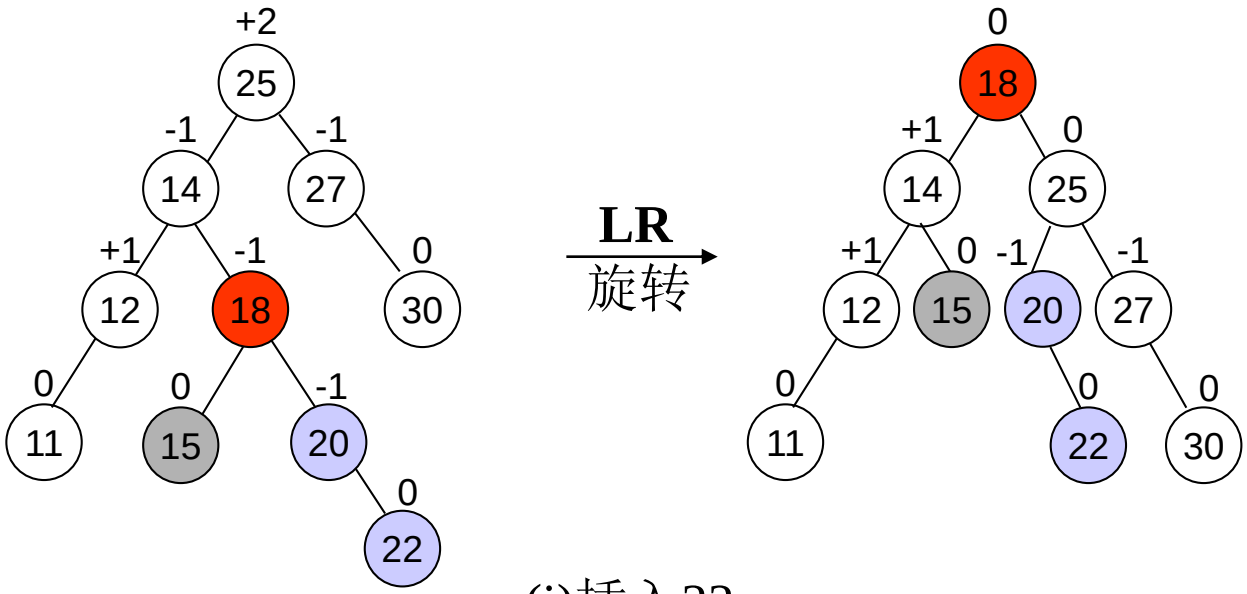
(h)插入20



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；

AVL树的平衡化处理

■在一棵AVL树上插入结点可能会破坏树的平衡性，需要平衡化处理恢复平衡，且保持BST的结构性质。

■若用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。

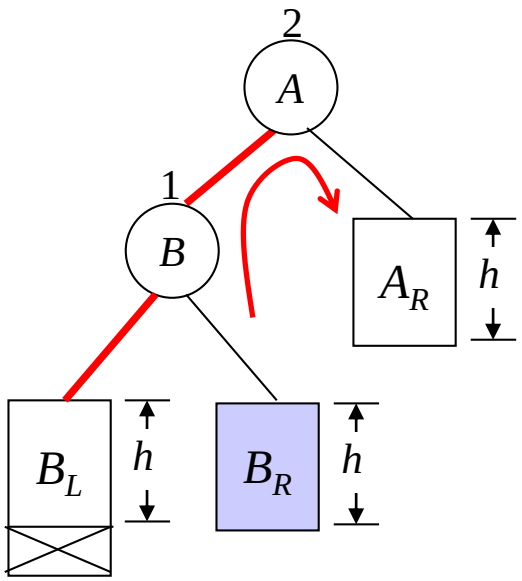
■可以用4种旋转进行平衡化处理：

①LL型：新结点Y 被插入到A 的左子树的左子树上（顺）

②RR型：新结点Y 被插入到A 的右子树的右子树上（逆）

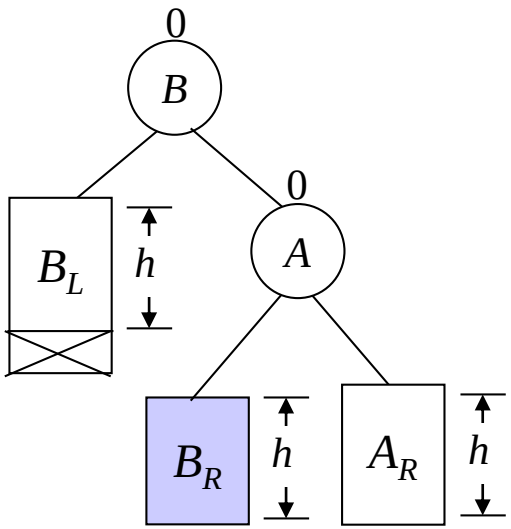
③LR型：新结点Y被插入到A的左子树的右子树上（逆、顺）

④RL型：新结点Y 被插入到A 的右子树的左子树上（顺、逆）

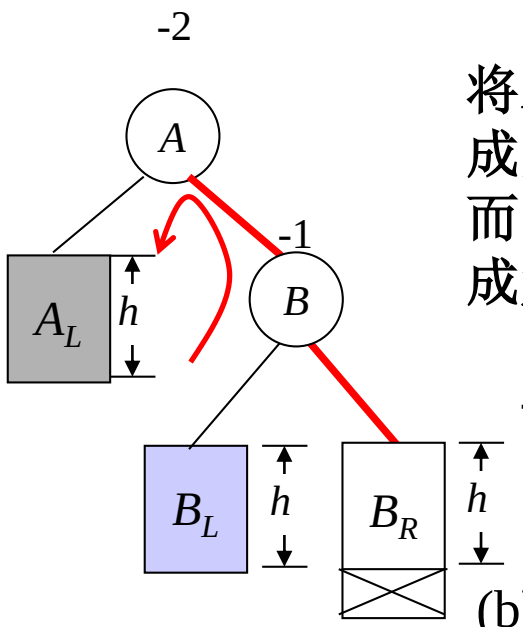


将A顺时针旋转，
成为B的右子树，
而原来B的右子树
成为A的左子树。

LL型

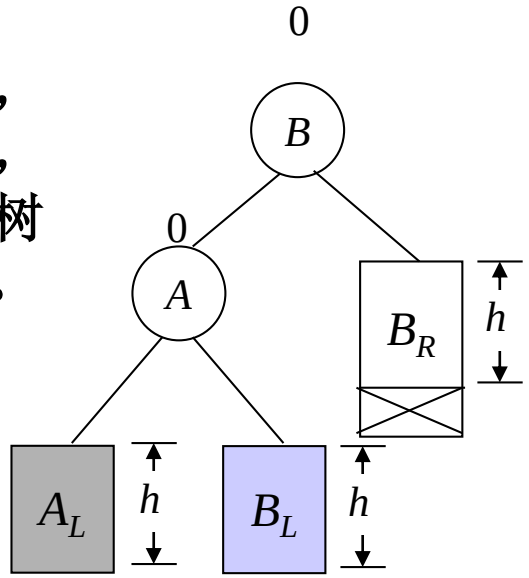


(a) LL型的旋转

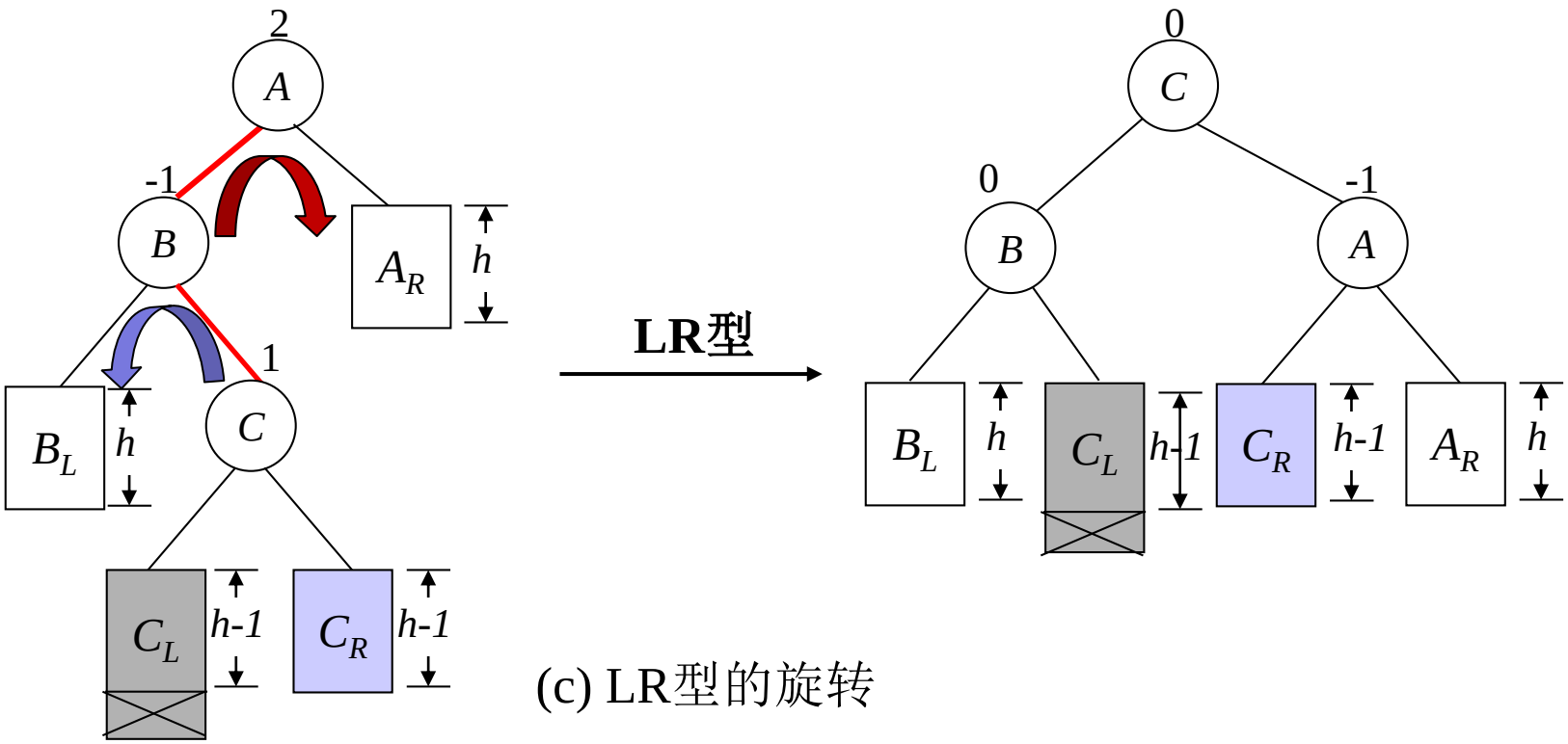


将A逆时针旋转，
成为B的左子树，
而原来B的左子树
成为A的右子树。

RR型



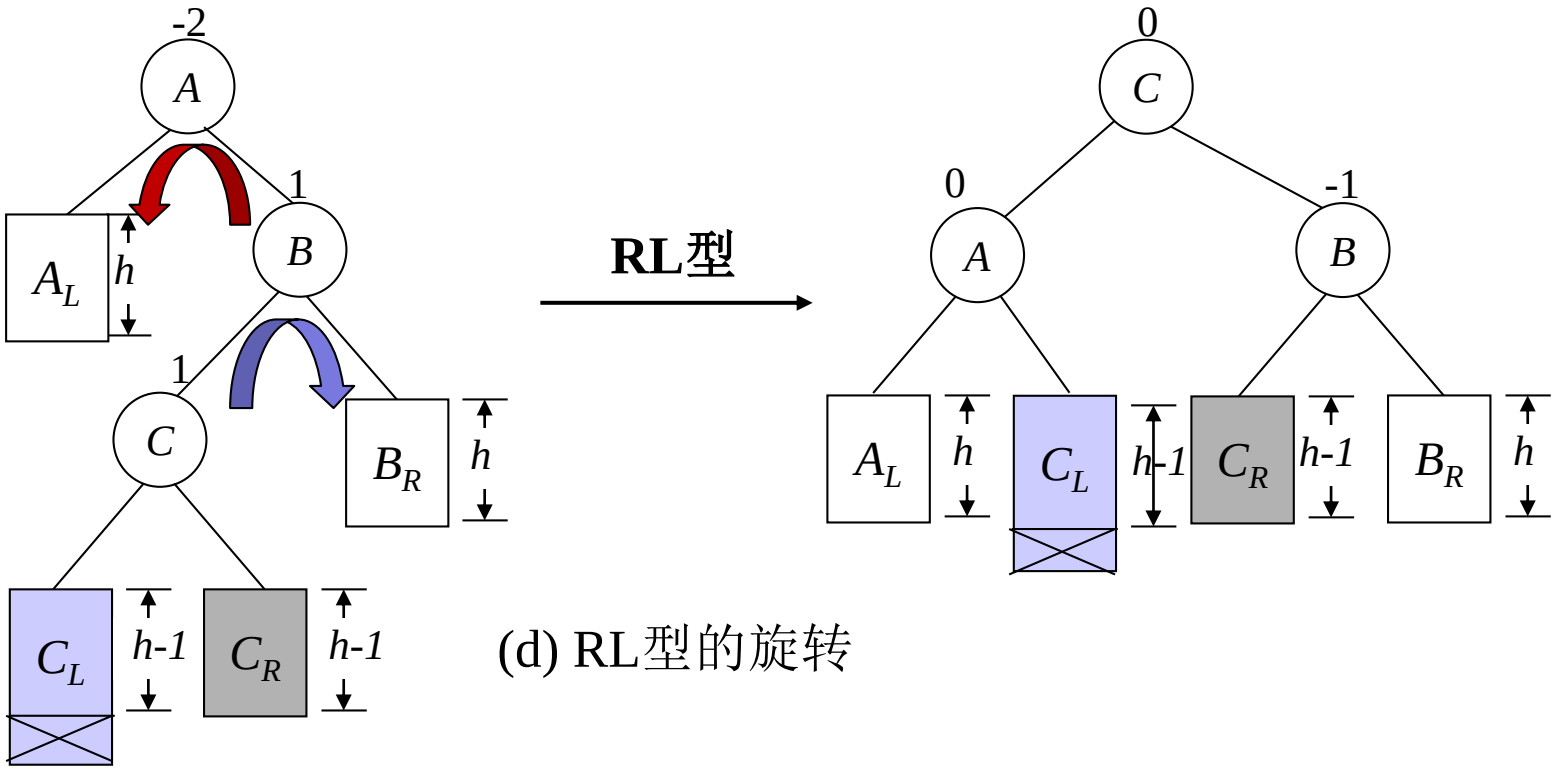
(b) RR型的旋转



(c) LR型的旋转

（1）绕 *C*，将 *B* 逆时针旋转，C_L作为 B 的右子树；

（2）绕 *C*，将 *A* 顺时针旋转，C_R作为 A 的左子树。



(d) RL型的旋转

（1）绕 *C*，将 *B* 顺时针旋转，C_R作为 B 的左子树；

（2）绕 *C*，将 *A* 逆时针旋转，C_L作为 A 的右子树。

AVL树的插入操作与建立

■对于一组关键字的输入，从空开始不断插入结点， 最后构成AVL树；

■每插入一个结点后，就应判断从该结点到根的路径上平衡因子有无结点发生不平衡；

■如有不平衡问题，利用旋转方法进行树的调整，使之平衡化；

■建AVL树过程是不断插入结点和必要时进行平衡化的过程。

AVL 树的结构类型

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