第九章 查找

- **查找表**：由同一类型的数据元素（记录）组成的集合。

  记作：ST={a1, a2, . . . , an}

学生成绩表

序号	学　号	姓　名	性别	数学	外语
1	200041	刘大海	男	80	75
2	200042	王　伟	男	90	83
3	200046	吴晓英	女	82	88
4	200048	王　伟	女	80	90
n	……	…..	…	….	…
	数据项1（主关键字）	数据项2			数据项5

- **关键字**：　可以标识一个记录的数据项
- **主关键字**：　可以唯一地标识一个记录的数据项
- **次关键字**：　可以识别若干记录的数据项

- 查找表的操作：

  生成查找表

  查找元素(记录)x在是否在表ST中

  查找元素(记录)x的属性

  插入新元素(记录)x

  删除元素(记录)x

  ……

- **查找**----根据给定的某个关键字值，在查找表中确定一个其关键字等于给定值的记录或数据元素。

  设k为给定的一个关键字值，R[1..n]为n个记录的表,若存在R[i].key=k,1≤i≤n,称**查找成功**；否则称**查找失败**。

- **静态查找**：查询某个特定的元素，检查某个特定的数据元素的属性,不插入新元素或删除元素(记录)。
- **动态查找**：在查找过程中，同时插入查找表中不存在的数据元素(记录)。

- 查找表的类型及其查找方法

(1) **静态查找表**

- 顺序表，用顺序查找法
- 线性链表，用顺序查找法
- 有序的顺序表，用：折半查找法；

  **斐班那契查找法；插值查找法；
- 索引顺序表/分块表，用分块查找法。

(2) **动态查找表**

- 二叉排序树，平衡二叉树(AVL树)

**- B树，B+树，键树

(3) **哈希(Hash)表**

- **平均查找长度**----查找一个记录时比较关键字次数的平均值。

$$ASL=\sum_{i=1}^{n} P_iC_i$$

P_i --- 查找r[i]的概率

C_i --- 查找r[i]所需比较关键字的次数

9.1 静态查找表

9.1.1 顺序表与顺序查找法

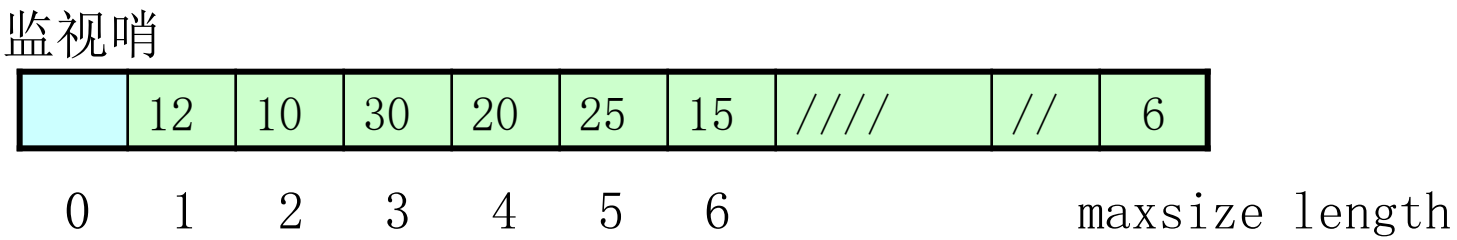
1. 顺序表的描述

例1 元素类型为记录(结构)

```
#define maxsize 100 //表长100
typedef struct node
  { keytype key ;  //关键字类型
    char name[6];  //姓名
    ......         //其它
  } ElemType;
typedef struct
  {
   ElemType elem[maxsize+1]; //maxsize+1个记录,
                             //elem[0]为监视哨
   int length;
  } SSTable;
SSTable ST1,ST2;
```

例2 元素类型为整型

```
#define maxsize 100 //表长100
typedef int ElemType;
typedef struct
 {
  ElemType elem[maxsize+1]; //maxsize+1个记录,
                            //elem[0]为监视哨
  int length;
  } SSTable;
SSTable ST1,ST2;
```



2. 顺序查找法(sequential search)算法设计

算法1：假定不使用监视哨elem[0]

基本思想：

将关键字k依次与记录的关键字:

elem [n].key, elem[n-1].key，..., elem[1].key

比较。

如果找到一个记录elem[i],有:

elem[i].key＝k (1≤i≤n)，

则查找成功，停止比较,返回记录的下标i;

否则,查找失败,返回0。

输入：查找表ST，查找条件（关键字）k
输出：成功时：记录序号，失败时：0

```
int sequsearch(SSTable ST,keytype k)
{
  int i=ST.length;          //从第n个记录开始查找
  while (i>=1 && k!=ST.elem[i].key)
       i--;                                  //继续扫描
  if (i)
     printf(" success\n");                   //查找成功
  else   printf(" fail\n");                  //查找失败
  return i;                              //返回记录的下标i
 }
```

算法2：假定使用监视哨elem[0]

基本思想：

先将关键字k存入elem[0].key,再将k依次与elem[n].key,...,elem[1].key,elem[0].key进行比较。

如果找到一个记录,有：

k=elem [i].key, (0≤i≤n)，则停止比较。

如果 i>0,则查找成功；否则,查找失败。

输入：查找表ST，查找条件（关键字）k

输出：成功时：记录序号，失败时：0

```
int sequsearch(SSTable ST,keytype k)
{
  int i=ST.length;          //从第n个记录开始查找
  ST.elem[0].key=k;          //k填入ST.elem[0].key
  while( k!=ST.elem[i].key )
    i-- ;                              //继续扫描
  return i;                            //返回记录的下标i
 }
```

3 查找算法性能分析：

对n个记录的表，所需比较关键字的次数

- 只考虑查找成功：最少为1，最多为n
  假定每个记录的查找概率相等，
  即 $P_1 = P_2 = \ldots = P_n = 1/n$

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

$$= \frac{1}{n}(n + (n-1) + \ldots + 1)$$

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

- 考虑查找失败：**使用监视哨elem[0],为 n+1**
  不使用监视哨elem[0],为 n

假定查找成功和失败的机会相同，对每个记录的查找概率相等，

$P_i=1/(2*n)$，则

$$ASL=\frac{1}{2n}\sum_{i=1}^{n} C_i + \frac{n+1}{2} = \frac{n+1}{4} + \frac{n+1}{2} = 3(n+1)/4$$

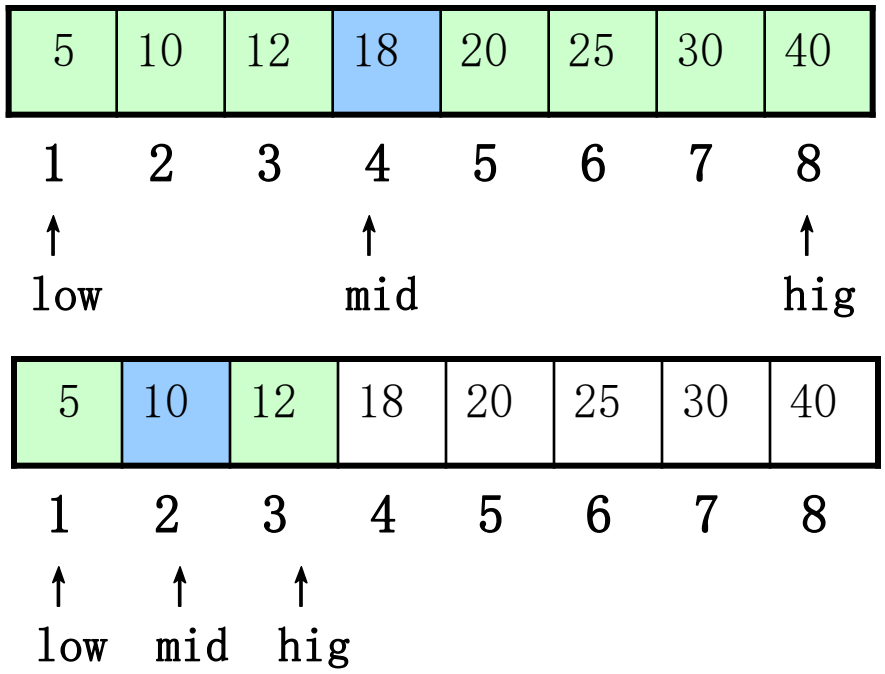
9.1.2 有序的顺序表的查找与折半查找法

1. 有序表

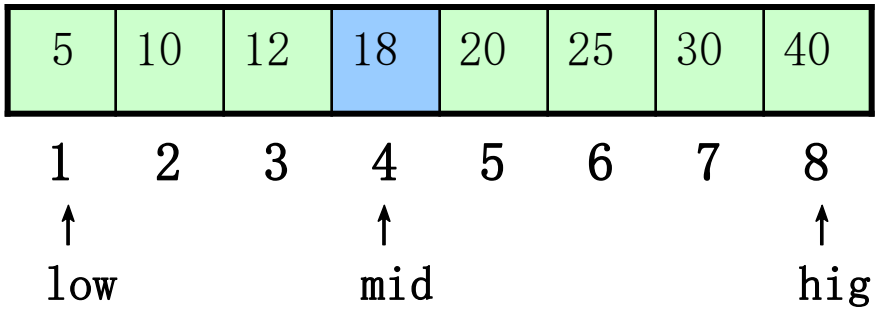
elem[1].key≤elem[2].key≤...≤elem[n].key

2. 折半查找(binary search,对半查找，二分查找)

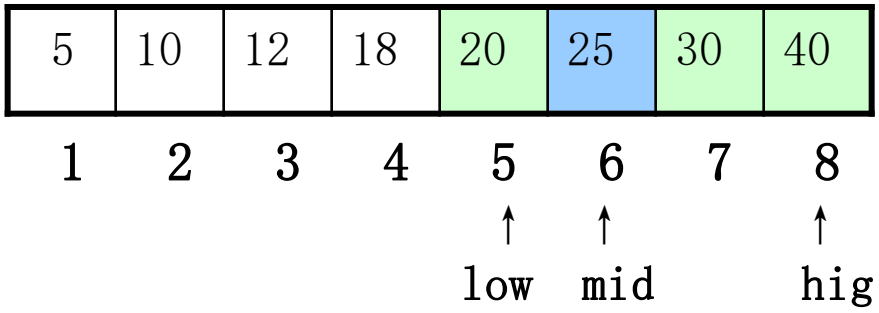
假定 k=10



假定k=40

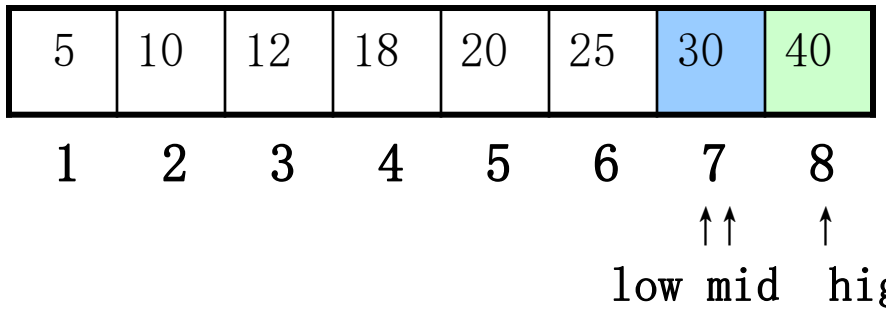


low=1,hig=8
mid=(low+hig)/2=4

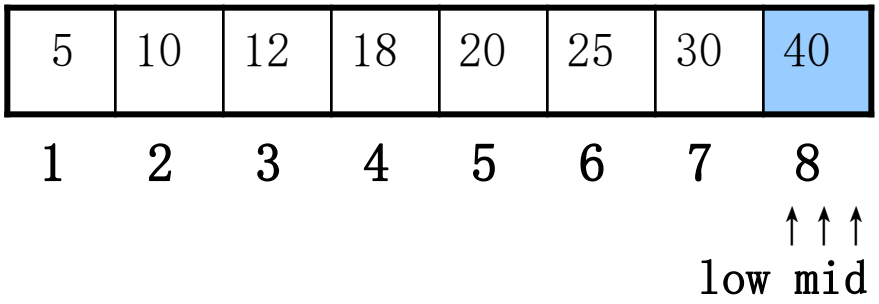


low=5,hig=8
mid=(low+hig)/2=6

假定 k=40（续）

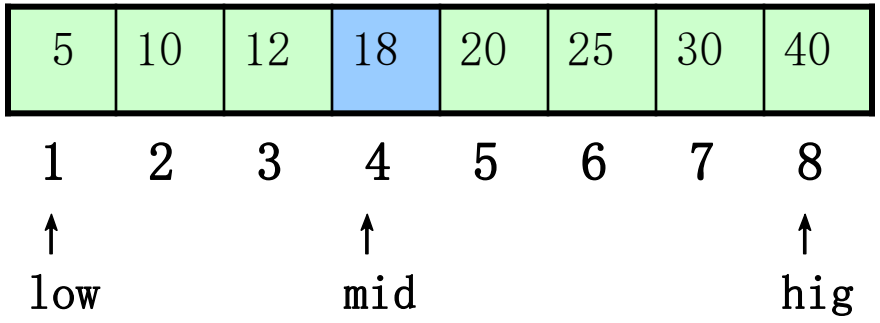


low=7,hig=8
mid=(low+hig)/2=7

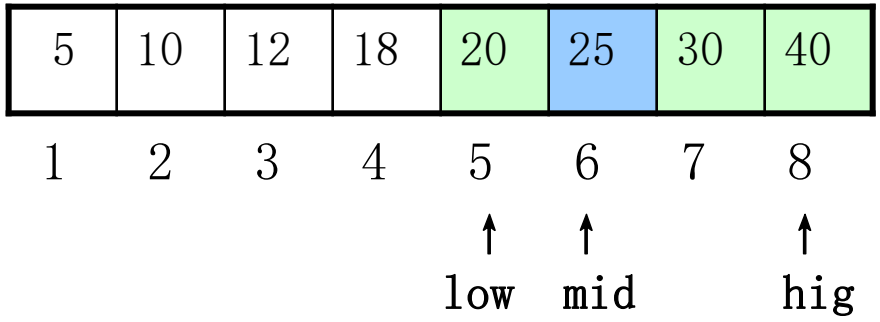


low=8,hig=8
mid=(low+hig)/2=8

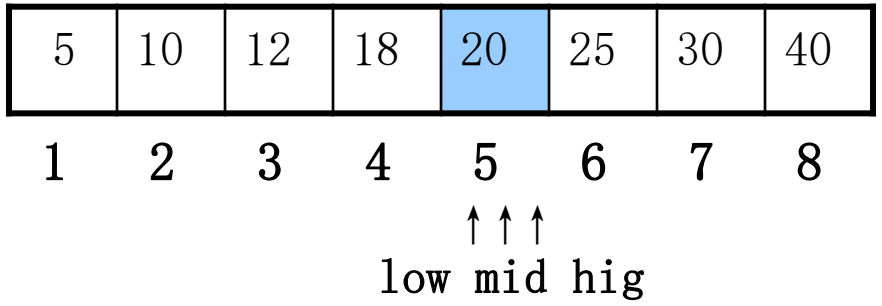
假定 k=22



low=1,hig=8
mid=(low+hig)/2=4

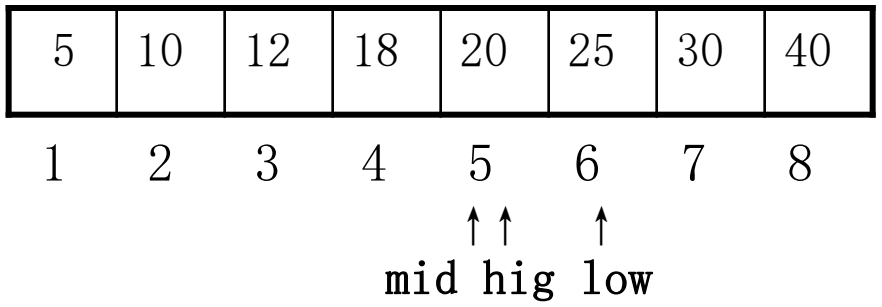


low=5,hig=8
mid=(low+hig)/2=6

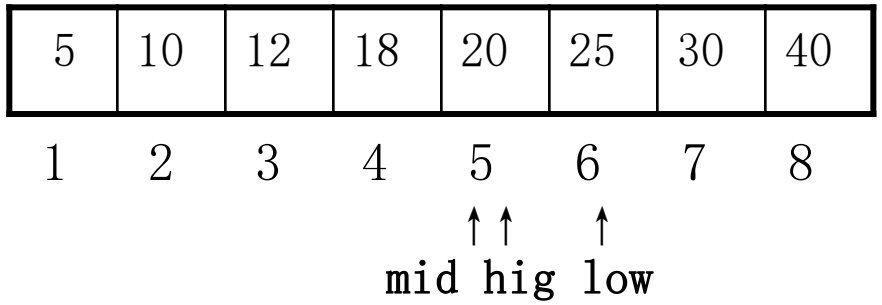


low=5,hig=5
mid=(low+hig)/2=5

假定 k=22（续）



mid=5
low=6,hig=5



mid=5
low=6,hig=5

hig<low,查找失败

3 折半查找算法1

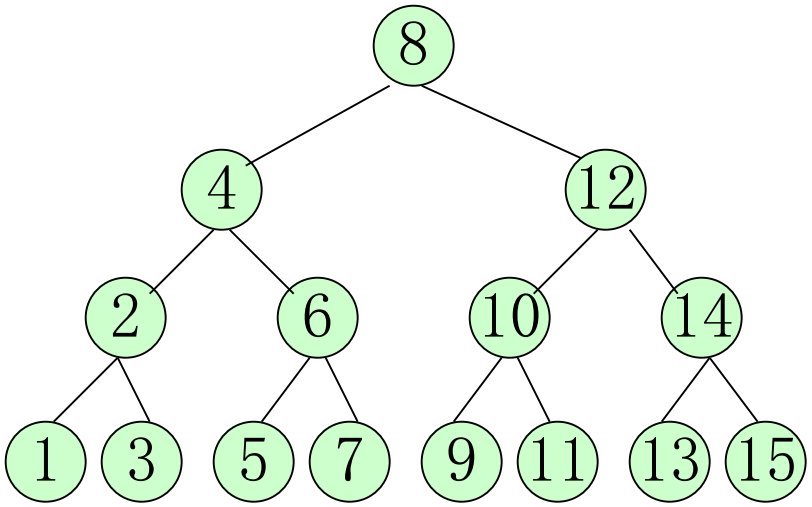
```
int binsrch(SSTable  ST,keytype k)
{ int low,mid,hig;
  low=1;
  hig=ST.length;
  while (low<=hig)
  {  mid=(low+hig)/2;        //计算中间记录的地址
     if (k<ST.elem[mid].key)
           hig=mid-1;           //查左子表
     else if (k==ST.elem[mid].key)
           break;               //查找成功,退出循环
     else low=mid+1;    //查右子表
   }
```

```
if (ST.elem[mid].key==k)
      { printf("success\n");    //查找成功
        return mid;
       }
 else
      { printf("fail\n");       //查找失败
        return 0;
       }
 }
```

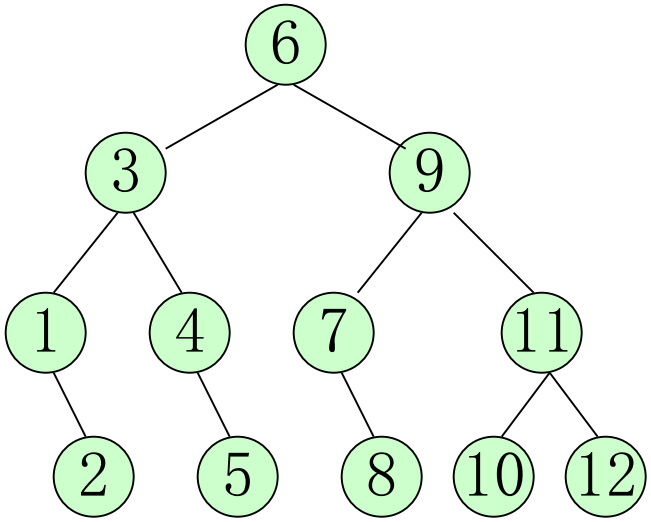
折半查找算法2

```
int binsrch(SSTable  ST,keytype k)
{int low,mid,hig;
 low=1;  hig=ST.length;
 while (low<=high)
 { mid=(low+high)/2;
   if (k<ST.elem [mid].key) hig=mid-1; //查左子表
   else if (k==ST.elem [mid].key)
          return mid;        //查找成功，返回mid
   else low=mid+1;     //查右子表
 }
 return 0 ;                  //查找失败，返回0
}
```

4. 判定树（描述折半查找过程的二叉树）



n=15, 满二叉树



n=12, 非满二叉树

- 结点内的数据表示数据元素的序号（如例中表示有15个元素组成的有序表的判定树）
- 根结点表示首先要和关键字k进行比较的数据元素的序号（如8），比较相等时，查找成功，否则，当k小于根结点对应元素的关键字时，下步就和左子结点（如序号4）对应元素的关键字比较，否则，下步就和右子结点（如序号12）对应元素的关键字比较。

● 若 $n = 2^k - 1$, 则判定树为满二叉树, 其深度为 $k=\log_2(n+1)$

假定$P_i=1/n(i=1,2,\ldots,n)$, 比较关键字的次数:

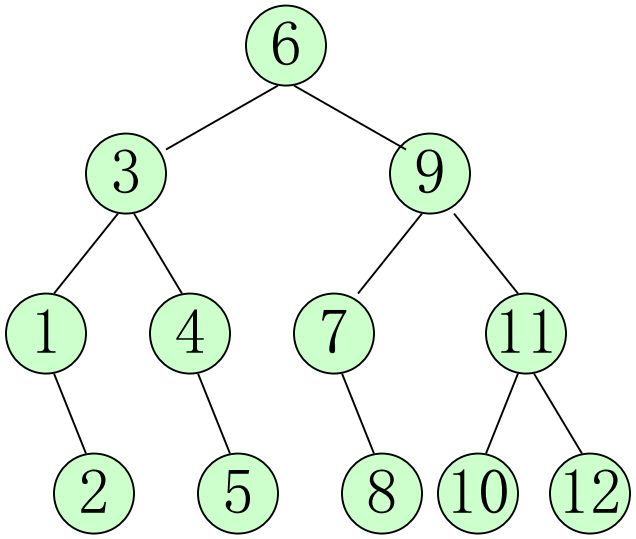
- 最少 $Cmin=1$
- 最多 $Cmax=\log_2(n+1)$
- $ASL=\frac{n+1}{n}-\log_2(n+1)-1$

设$n=15$ $ASL=\frac{15+1}{15}-\log_2(15+1)-1=\frac{16}{15}*4-1\approx3.3$

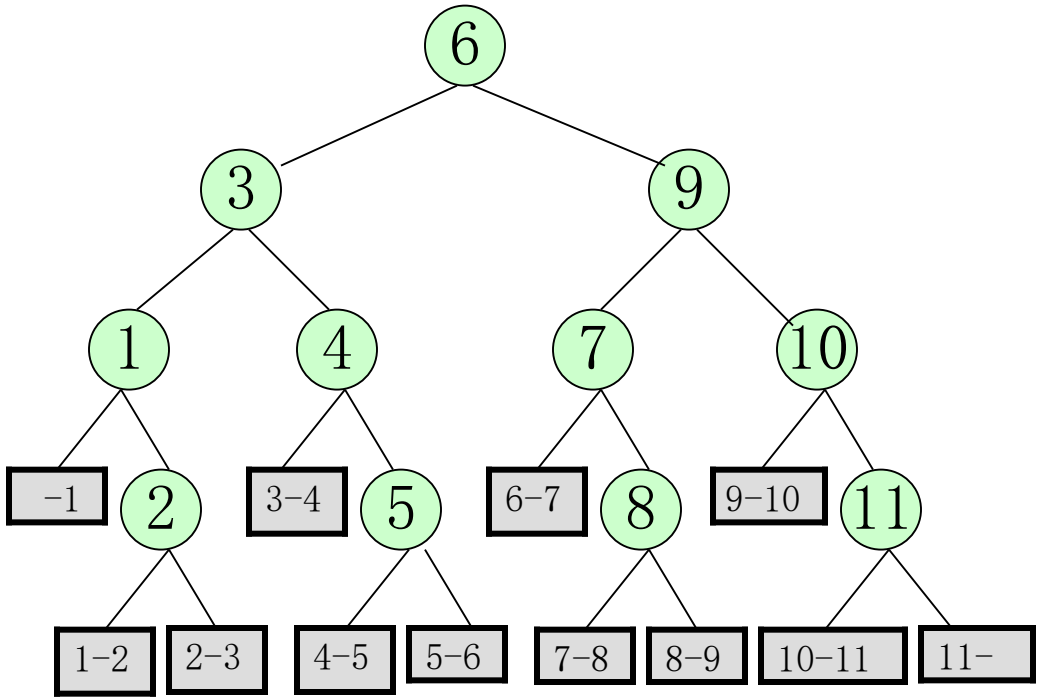
对任意的n

$$ASL \approx \frac{n+1}{n}\log_2(n+1)-1 = O(\log_2 n)$$

设n=12， $ASL = \frac{1+2+2+3+3+3+3+4+4+4+4+4}{12} = \frac{37}{12} \approx 3.1$

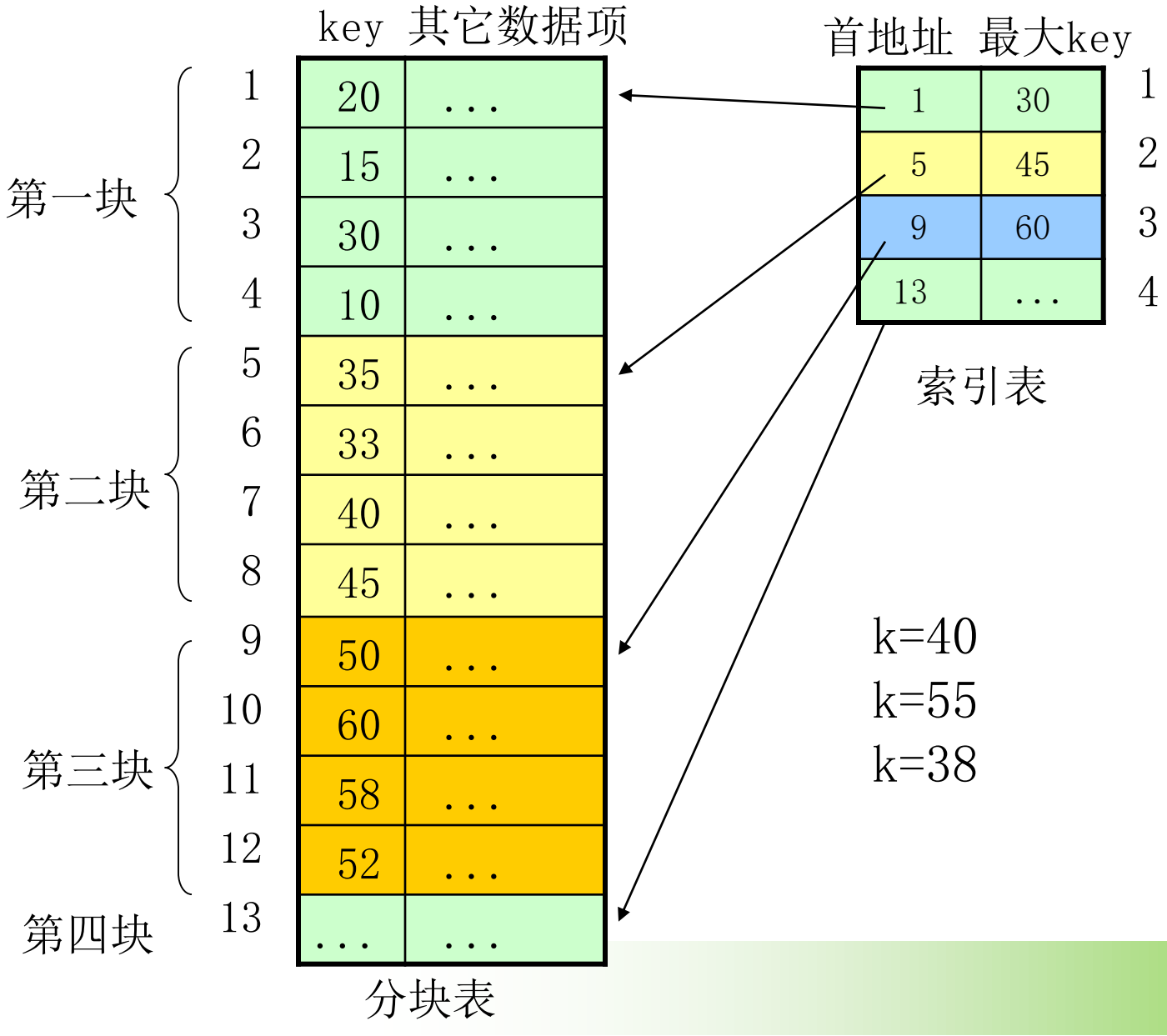


n=12，判定树



n=11，加外部结点的判定树

9.1.3 索引顺序表(分块表)与分块查找法



- 条件

  (1)分块表"按块有序"，索引表"按key有序"

  (2)设n个记录分为b个块，每块的记录数s=n/b

- 查找方法与ASL

  (1)顺序查找(或折半查找)索引表

  确定k值所在的块号或块的首地址

  $$ASL(1)=Lb=\frac{b+1}{2}$$

  (2)在某一块中顺序查找

  $$ASL(2)=Lw=\frac{s+1}{2}$$

- $$ASL=Lb+Lw=\frac{b+1}{2}+\frac{s+1}{2}=\frac{1}{2}(b+s)+1=\frac{1}{2}(\frac{n}{s}+s)+1$$

- 最佳分块

  $$s=\sqrt{n} \qquad b=\sqrt{n}$$

  $$ASL_{min}=\sqrt{n}+1=O(\sqrt{n})$$

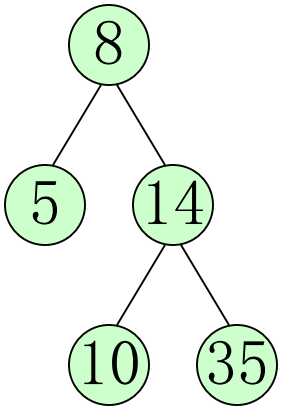
9.2 动态查找表

1. 二叉排序树(二叉查找树)

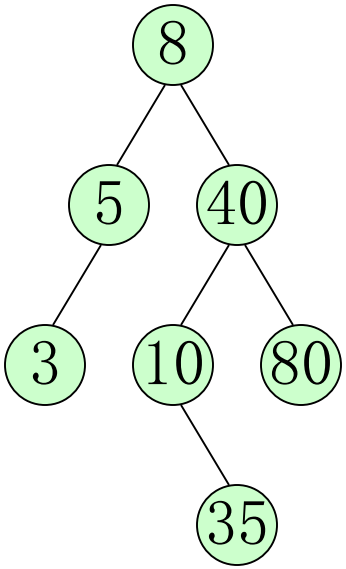
(1) 二叉排序树的定义

如果二叉树的任一结点大于其非空左子树的所有结点，而小于其非空右子树的所有结点，则这棵二叉树称为**二叉排序树**。

对一棵二叉排序树进行中序遍历，所得的结点序列一定是递增有序的。

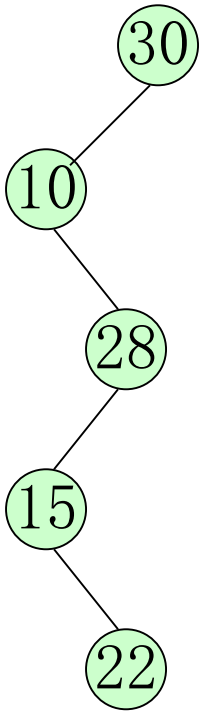


LDR：5, 8, 10, 14, 35

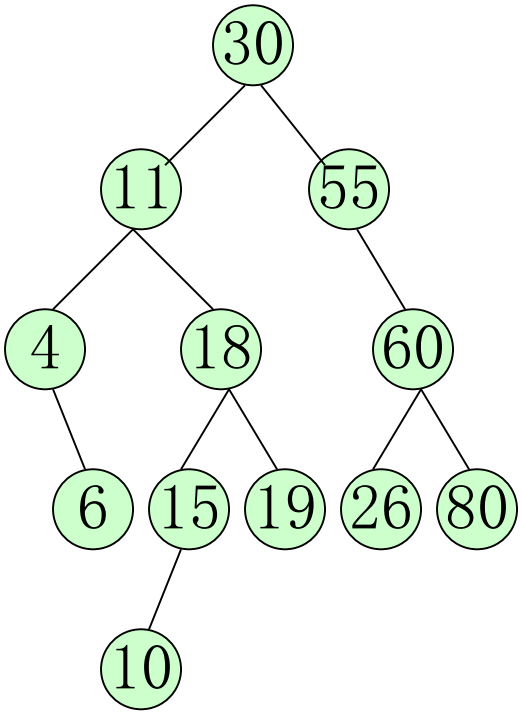


LDR：3, 5, 8, 10, 35, 40, 80

下列二叉树是否为二叉排序树？



T1

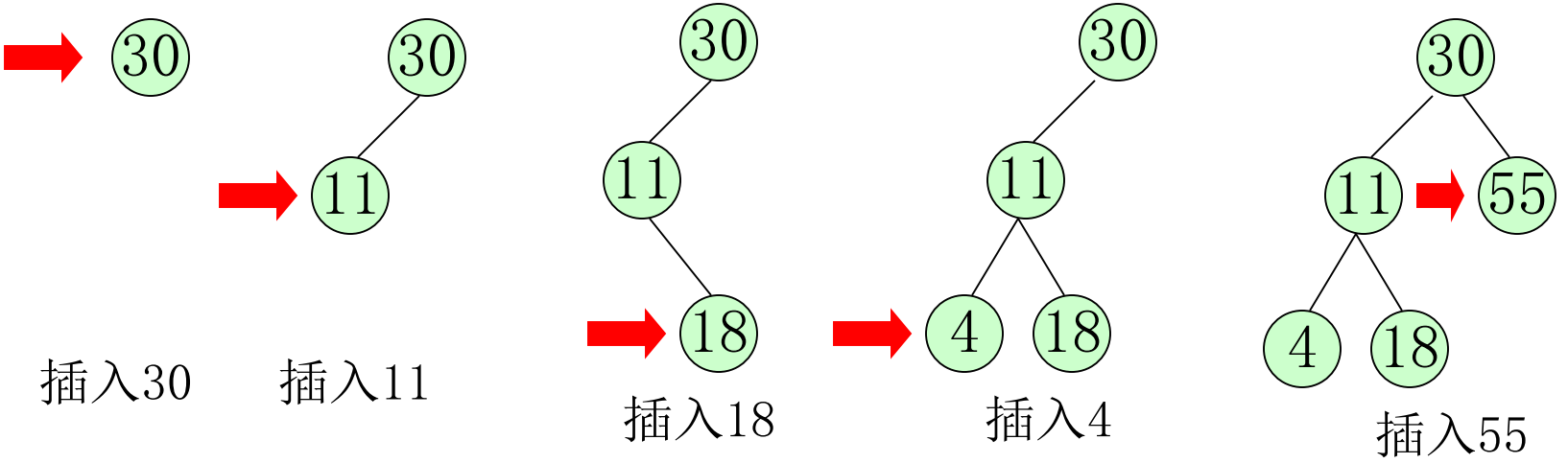


T2

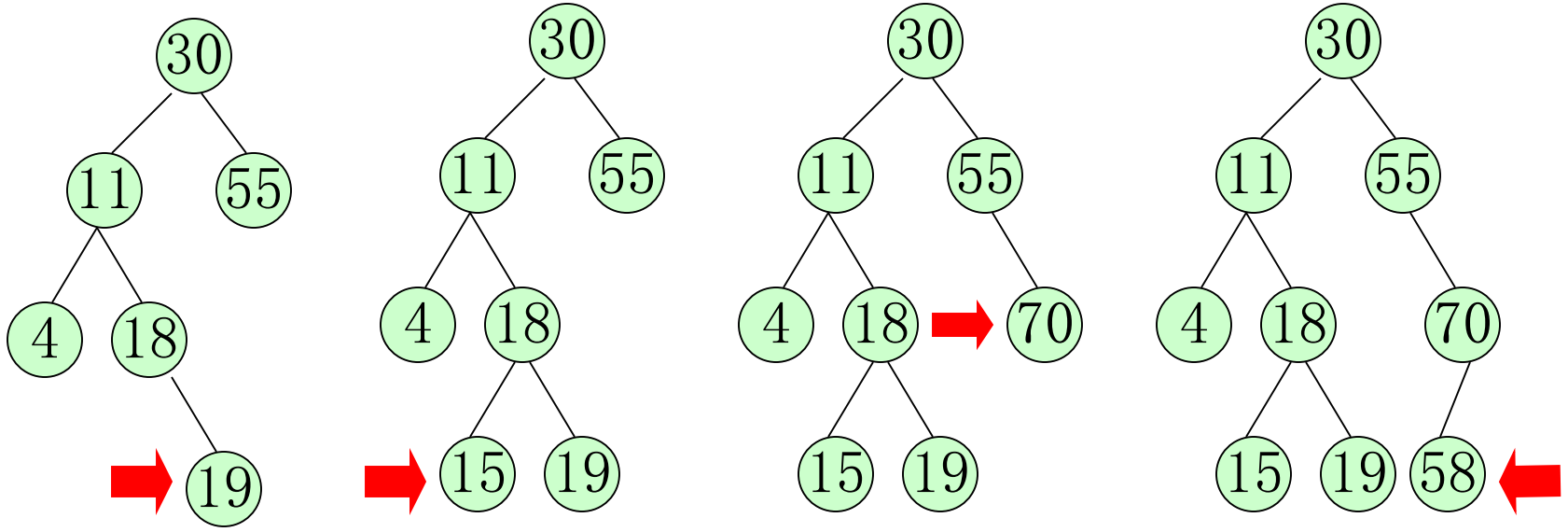
T3

（2）二叉排序树的生成

设输入序列为：30, 11, 18, 4, 55, 19, 15, 70, 58



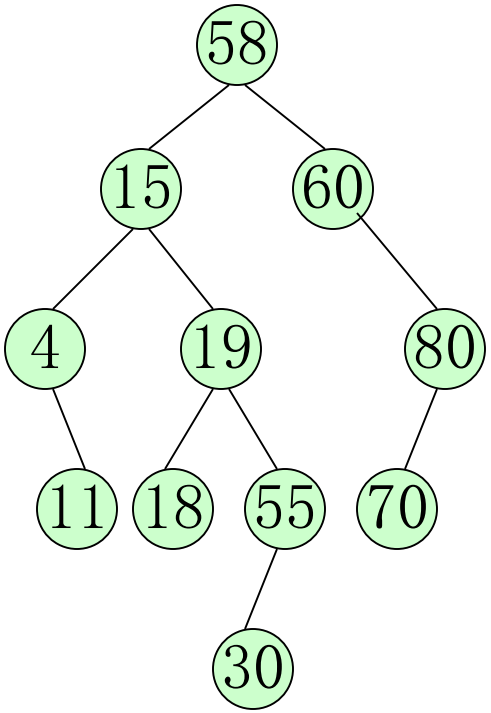
插入30　插入11　插入18　插入4　插入55



插入19　插入15　插入70　插入58

课堂练习：

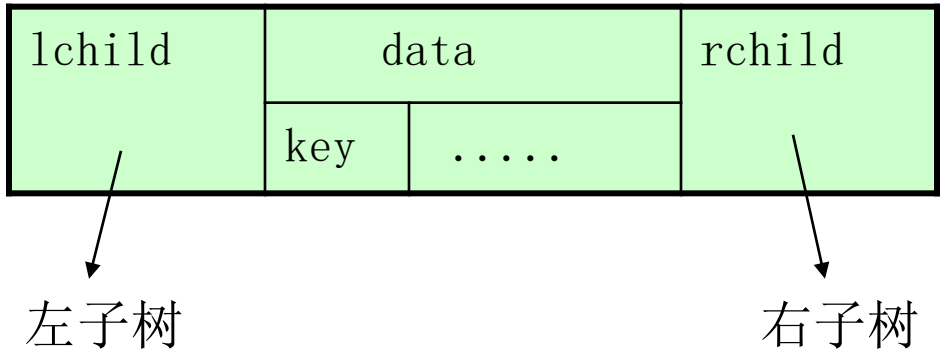
设输入关键字序列为:58,60,15,80,19,55,4,18,70,11,30,生成二叉排序树,试画出二叉排序树；假定查找每个结点(关键字)的概率相同，计算查找成功时的平均查找长度ASL。



$$ASL=\frac{1+2+2+3+3+3+4+4+4+4+5}{11}=\frac{35}{11}\approx 3.18$$

(3)二叉排序树的存储结构

结点形式：



结点类型定义：

```
struct  node
{ struct
   { int key ;                    //关键字
     .....                        //其它数据项
    } data ;
 struct node *lchild,*rchild ; //左右子树的指针
 } *root,*t;
```

(4) 插入1个元素到二叉排序树的算法

```
struct node *intree(struct node *t,ElemType x)
{ if (t==NULL)          //t是指向二叉树根的指针
   {
   t=(struct node *)malloc(sizeof(struct node));
   t->data=x;                          //生成并插入结点x
   t->lchild=t->rchild=NULL;    //为叶子结点
   }
  else if(x.key<t->data.key)
         t->lchild=intree(t->lchild,x); //插入左子树
       else
         t->rchild=intree(t->rchild,x); //插入右子树
  return t;
  }
```

(5)二叉排序树的查找算法

（返回值　失败：NULL　成功：非NULL，结点指针）

a) **递归算法**

```
struct node *search_tr(struct node *t, keytype k)
{ if (t==NULL) return NULL;              //查找失败
  else
    if (k==t->data.key)
            return t;  //查找成功
    else
      if (k<t->data.key)
        return search_tr(t->lchild,k); //查左子树
      else
        return search_tr(t->rchild,k); //查右子树
 }
```

b) **非递归算法**

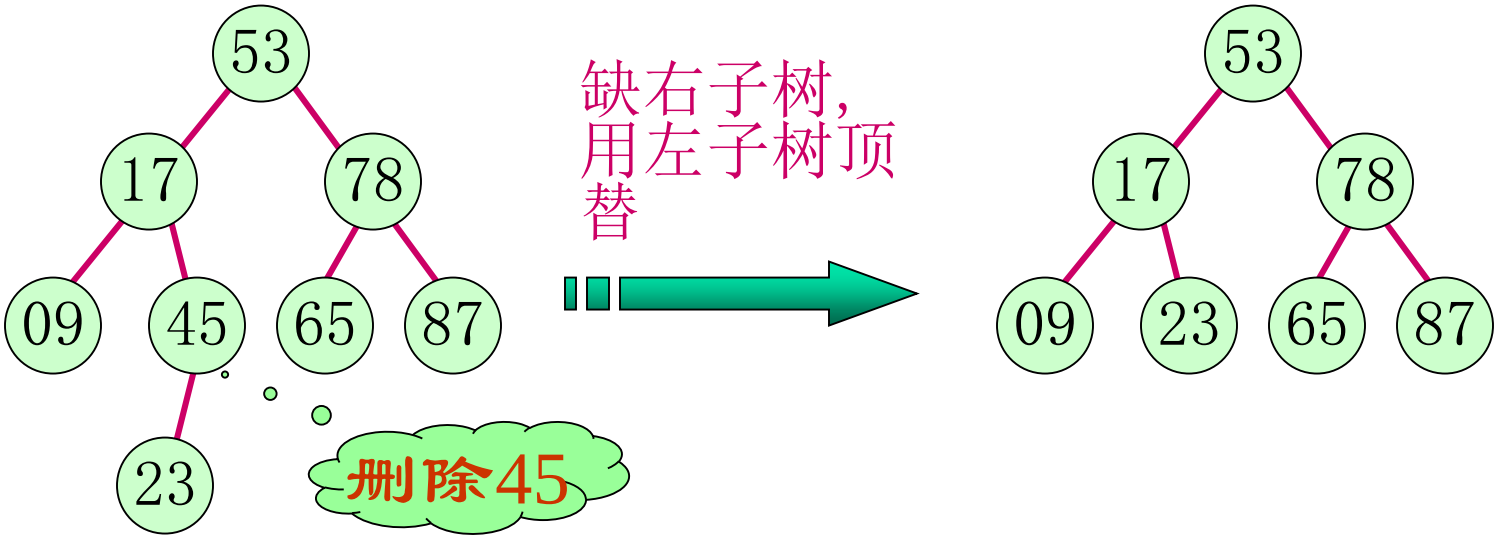
```
struct node *search_tree(struct node *t,keytype k)
  { while (t!=NULL)
      if (k==t->data.key)
          return t;                    //查找成功
      else
       if (k<t->data.key)
            t=t->lchild;               //查左子树
       else
            t=t->rchild;               //查右子树
    return t;                          //查找失败
   }
```

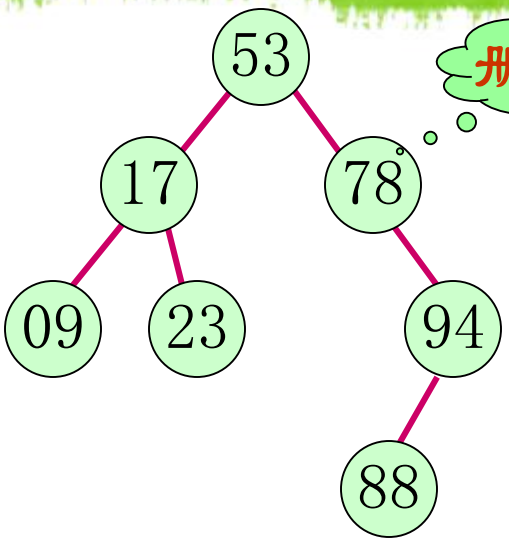
(6) 二叉排序树的删除

- 在二叉排序树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来，同时确保二叉排序树的性质不会失去。

- 为保证在删除节点后二叉排序树的性质不会丢失：
  - **删除叶结点**，只需将其双亲结点指向它的指针置空，再释放它即可。
  - **被删结点缺左子树（或右子树）**，可以用被删节点的右子树（或左子树）顶替它的位置，再释放它。

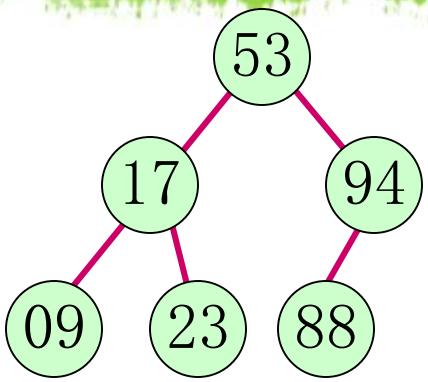
- **被删结点左、右子树都存在**，可以在它的右子树中寻找中序下的第一个结点(关键值最小)，用它的值填补到被删结点中，再来处理这个结点的删除问题。

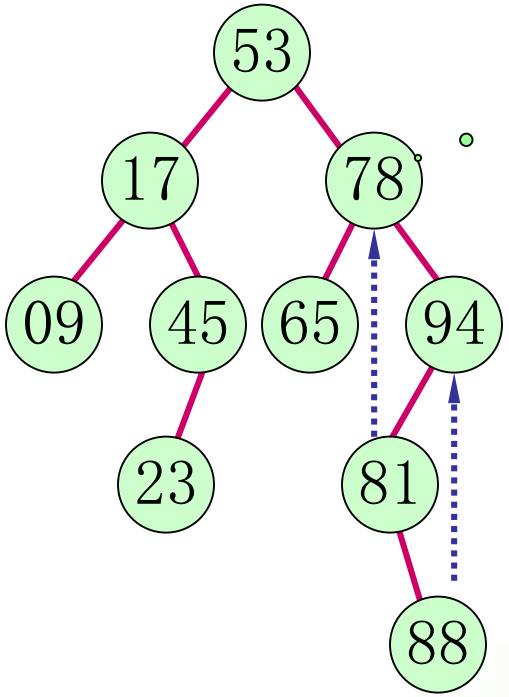




删除78

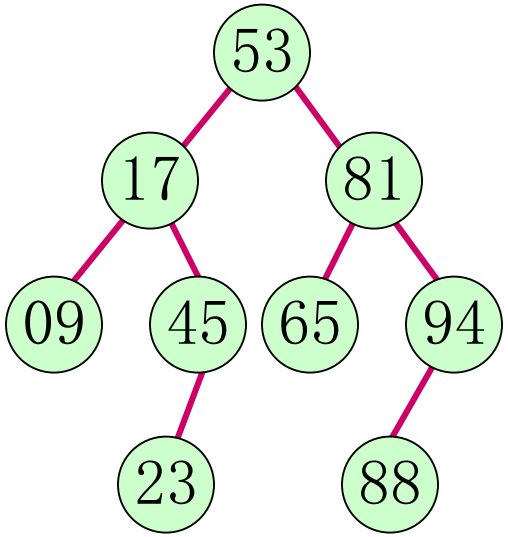
缺左子树，用右子树顶替





删除78

在右子树上找中序下第一个结点填补



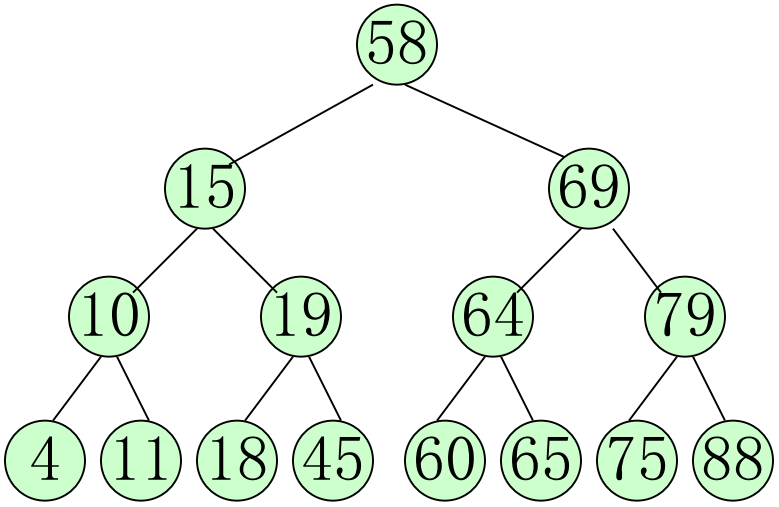
(7) 查找性能分析

最好情况(为满二叉树)

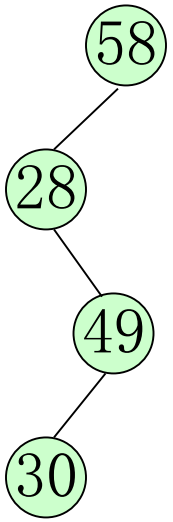
$$ASL = \frac{n+1}{n}\log_2(n+1) - 1 = O(\log_2 n)$$

最坏情况(为单枝树)： $ASL=(1+2+...+n)/n=(n+1)/2$

平均值： $ASL \approx O(\log_2 n)$



满二叉树



单枝树

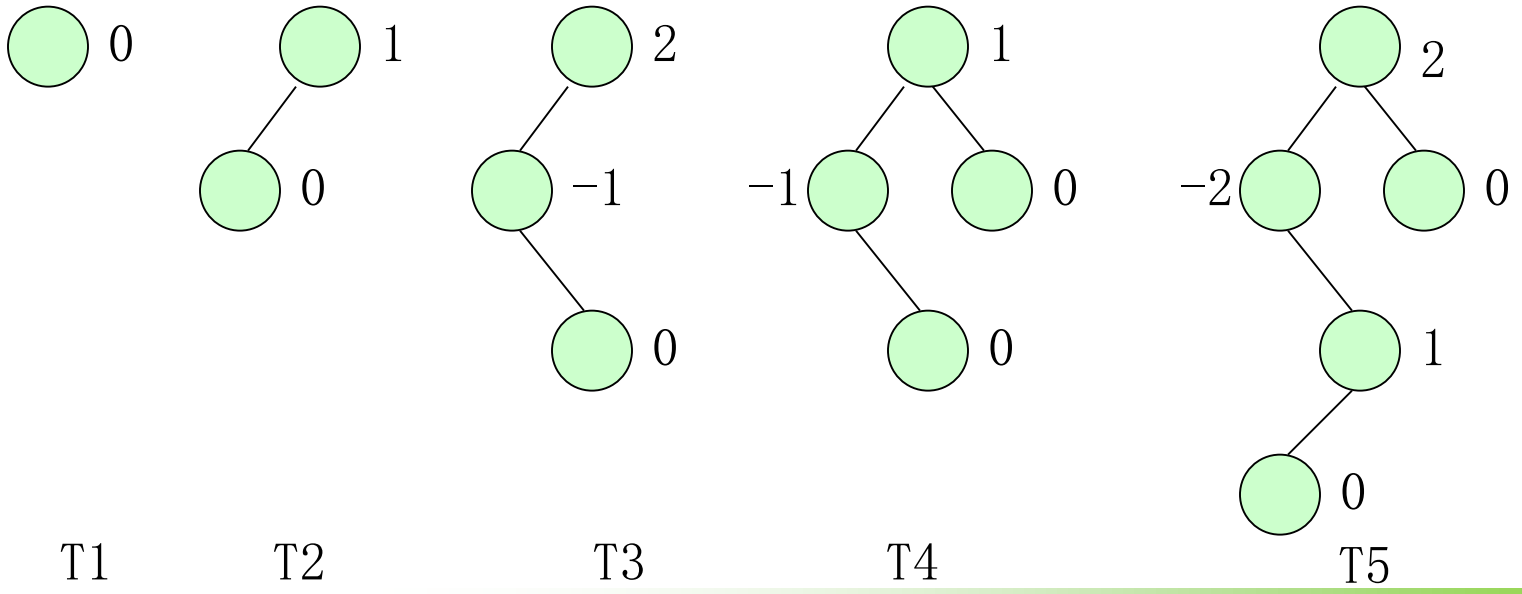
$ASL=(15+1)/15*\log_2(15+1)-1 \approx 3.3$

$ASL=(1+2+3+4)/4=2.5$

2.平衡二叉树(高度平衡二叉树)

（1）AVL树的定义

- AVL树：由G.M.Adelson-Velskii和E.M.Landis提出。
- 结点的平衡因子：结点的左右子树的深度之差。
- 平衡二叉树：任意结点的平衡因子的绝对值小于等于1的二叉树。



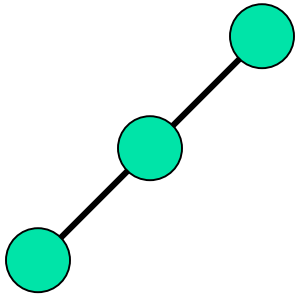
（2）AVL树的存储结构：

```
typedef int DataType;      //结点数据类型

typedef struct node {      //AVL树结点定义
    DataType data;         //结点数据域
    int balance;           //结点平衡因子域
    struct node *leftChild, *rightChild;
                           //结点左、右子树指针域
} AVLNode;

typedef  AVLNode * AVLTree;      //AVL树
```

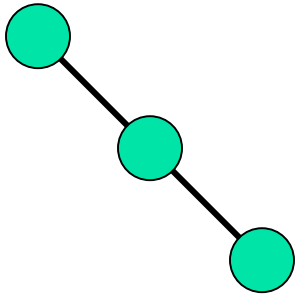
（3）平衡化旋转

- 如果在一棵平衡的二叉搜索树中插入一个新结点，造成了不平衡。此时必须调整树的结构，使之平衡化。
- 平衡化旋转有两类：
  - 单旋转（左旋和右旋）
  - 双旋转（左平衡和右平衡）
- 每插入一个新结点时，AVL 树中相关结点的平衡状态会发生改变。因此,在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子。

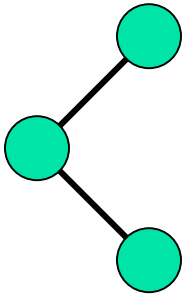
- 如果在某一结点发现高度不平衡，停止回溯。从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。单旋转可按其方向分为左单旋转和右单旋转， 其中一个是另一个的镜像，其方向与不平衡的形状相关。
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为先左后右和先右后左两类。



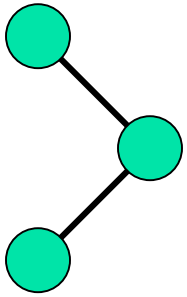
右单旋转



左单旋转

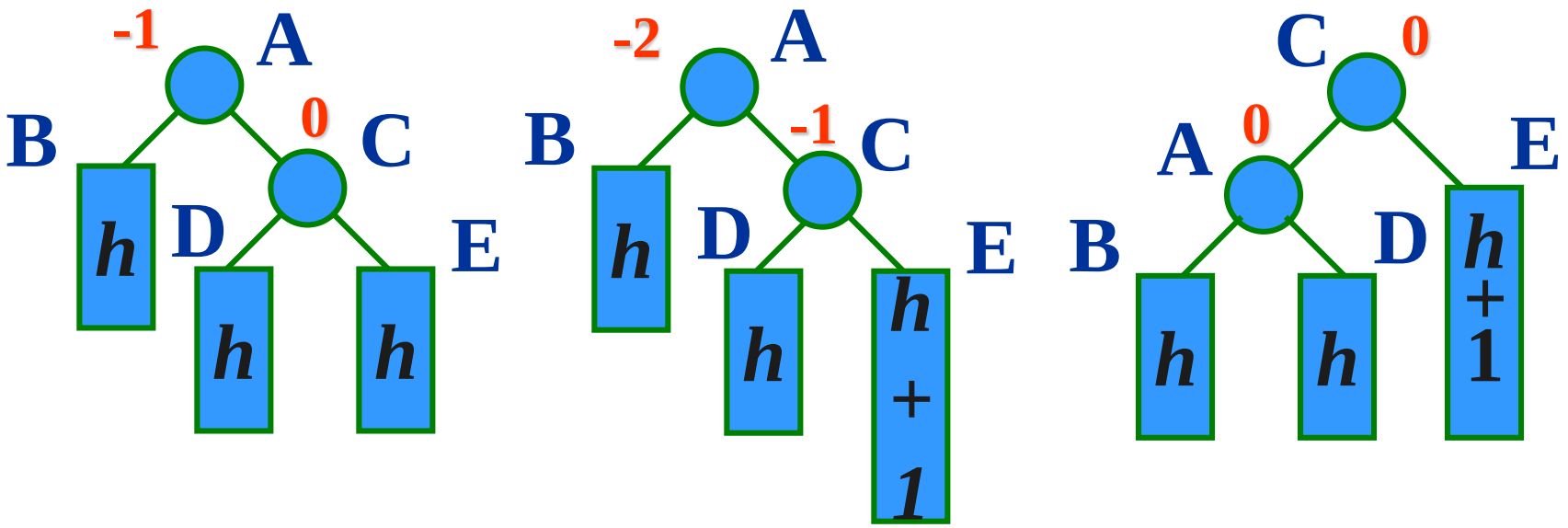


左右双旋转



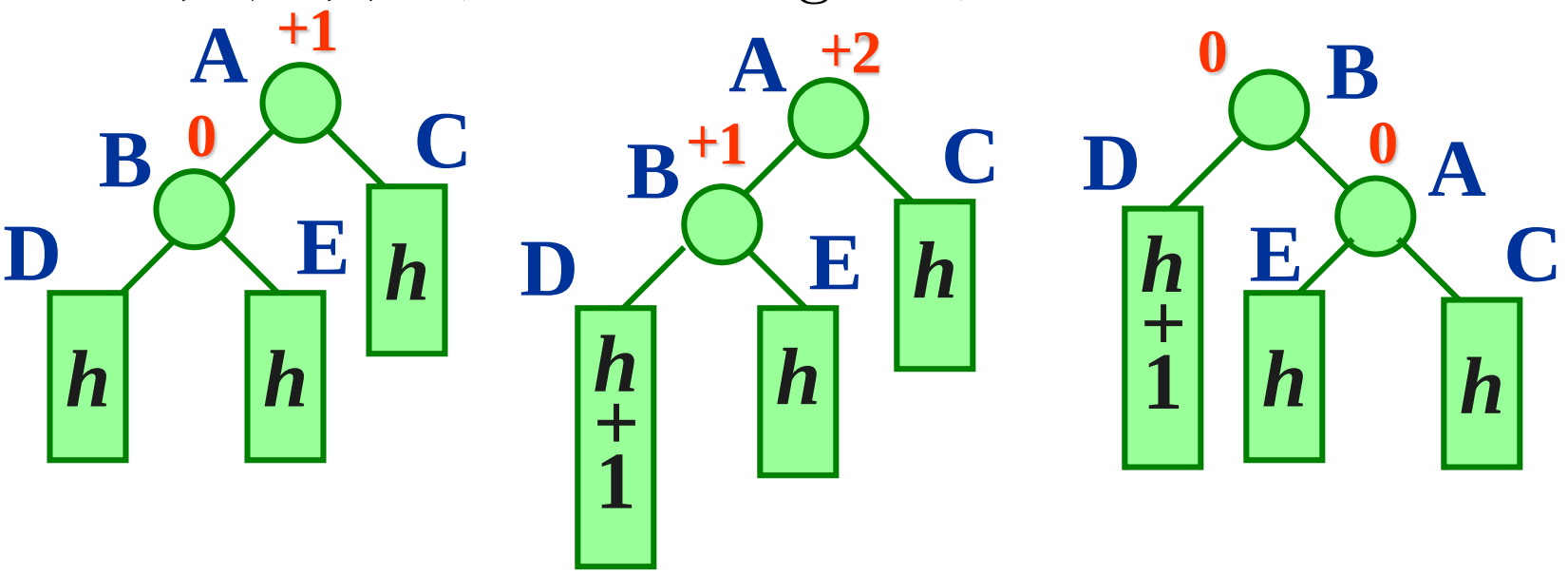
右左双旋转

（a）左单旋转（RotateLeft ）



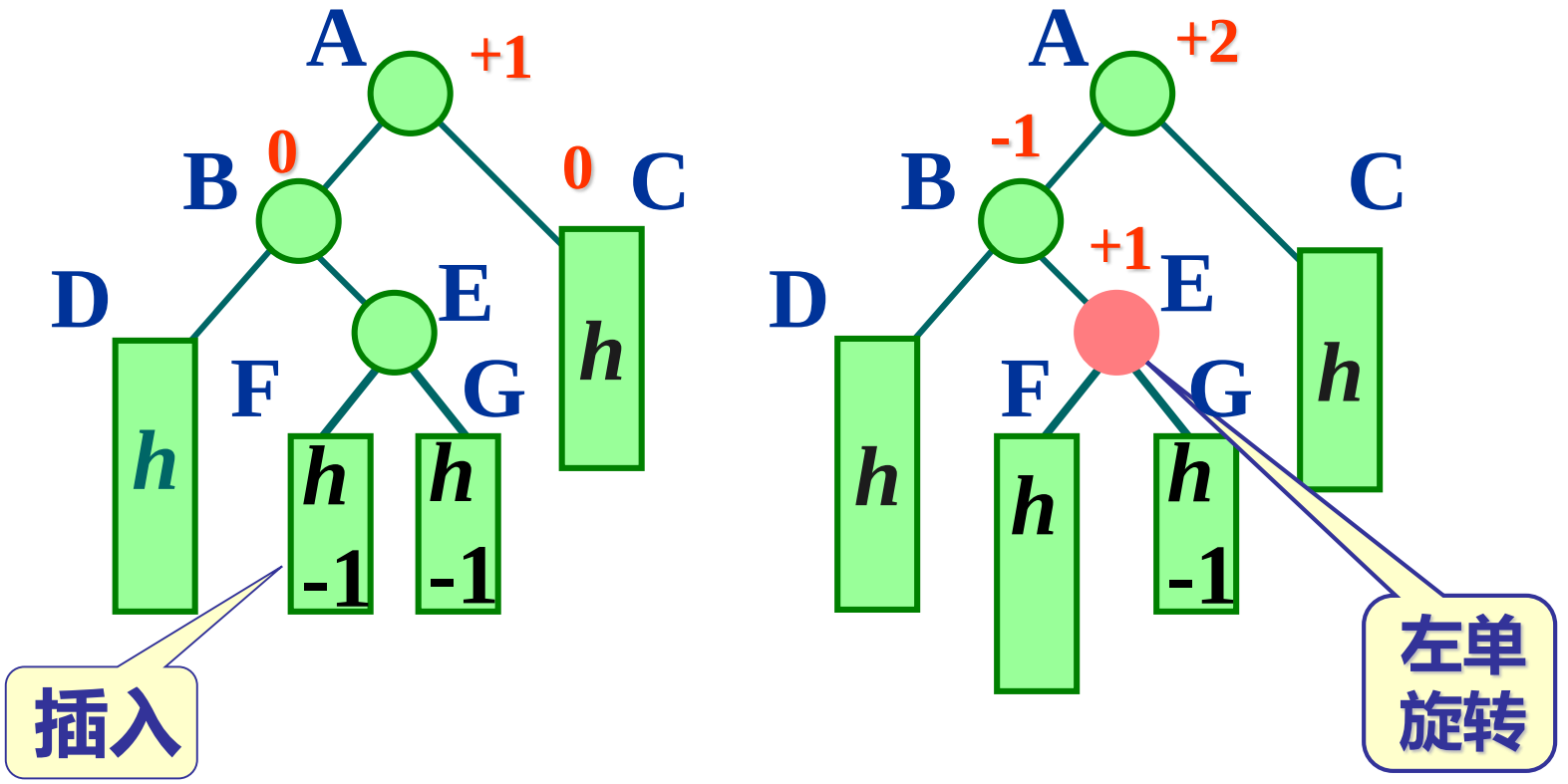
- 在子树E中插入新结点，该子树高度增1导致结点A的平衡因子变成-2，出现不平衡。
- 沿插入路径检查三个结点A、C和E。它们处于方向为“\”的直线上，需做左单旋转。
- 以结点C为旋转轴，让结点A逆时针旋转。

（b）右单旋转（RotateRight ）

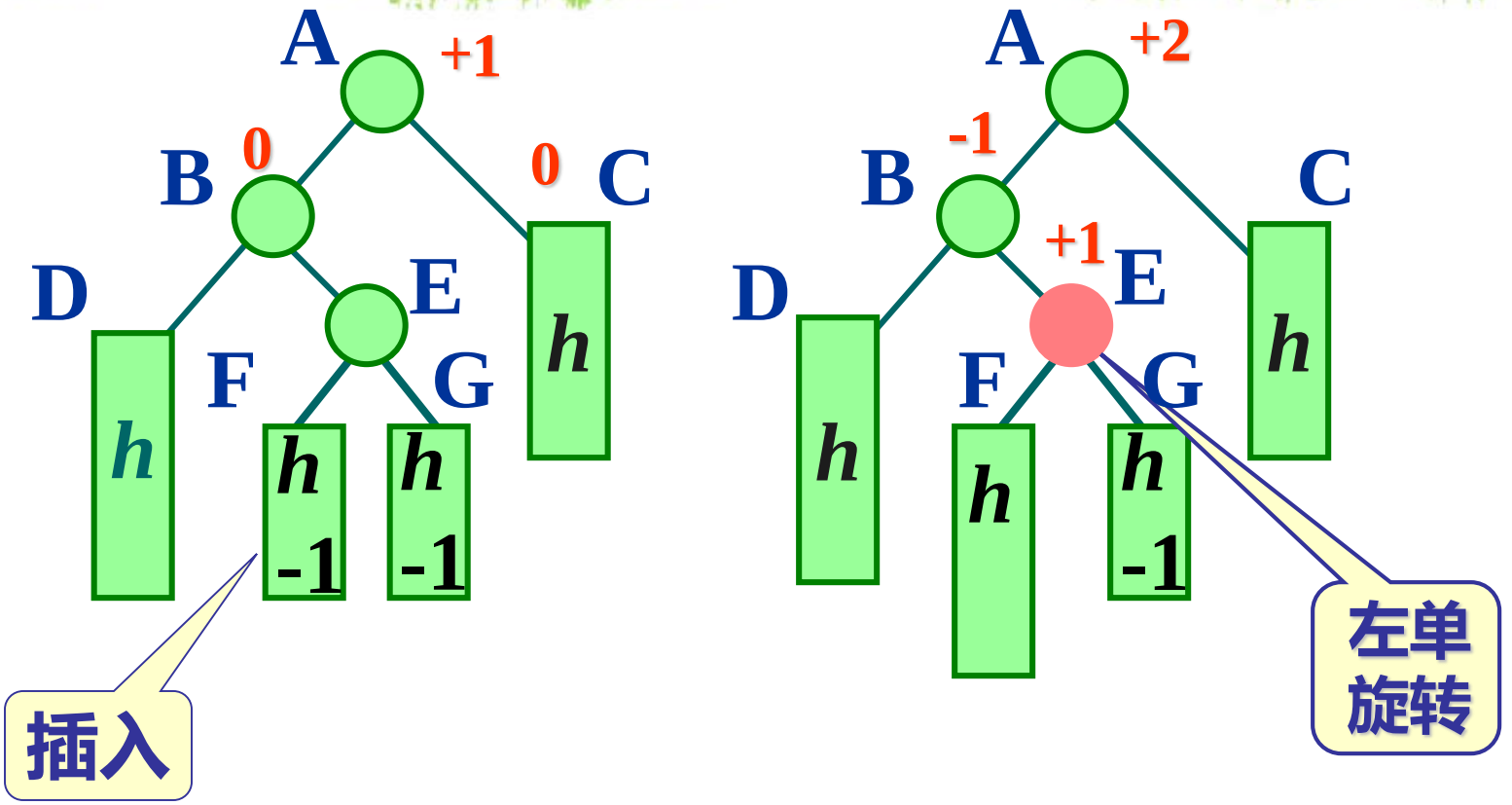


- 在子树D中插入新结点，该子树高度增1导致结点A的平衡因子变成+2，出现不平衡。
- 沿插入路径检查三个结点A、B和D。它们处于方向为“/”的直线上，需做右单旋转。
- 以结点B为旋转轴，让结点A顺时针旋转。

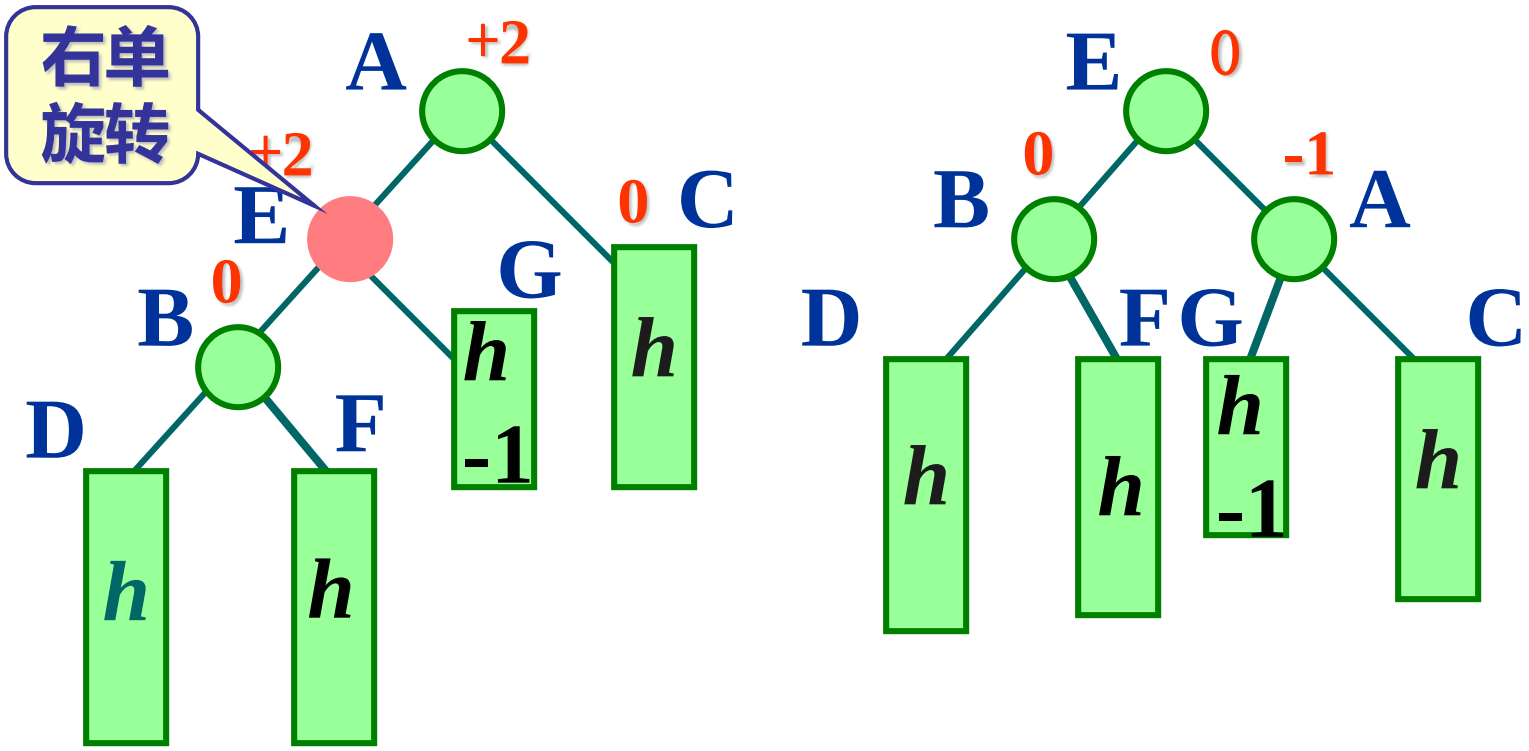
（c）先左后右双旋转 (RotationLeftRight)



- 在子树F或G中插入新结点，该子树的高度增1。结点A的平衡因子变为+2，发生了不平衡。

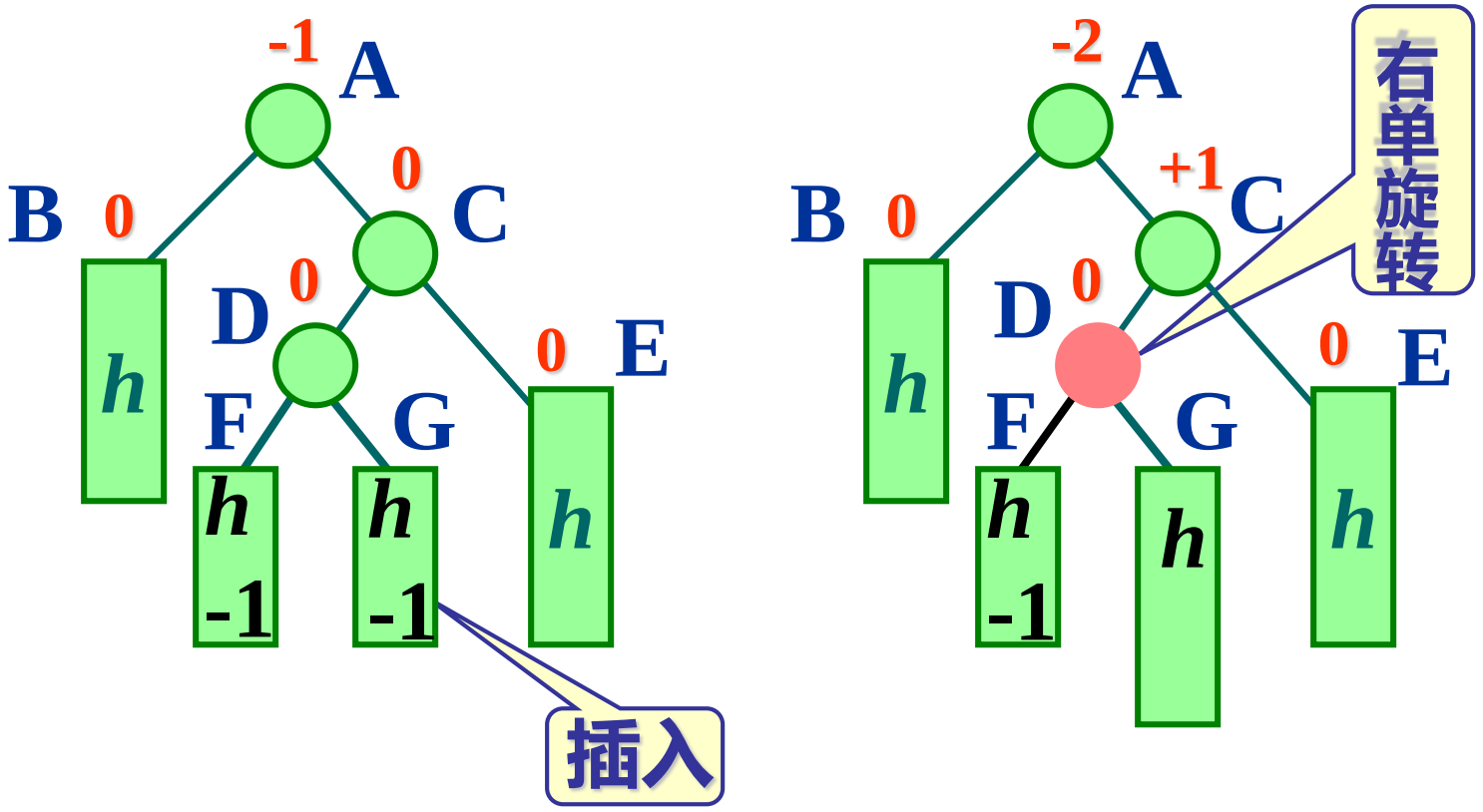


- 从结点A起沿插入路径选取3个结点A、B和E，它们位于一条形如“〈”的折线上，因此需要进行先左后右的双旋转。
- 以结点E为旋转轴，将结点B逆时针旋转。

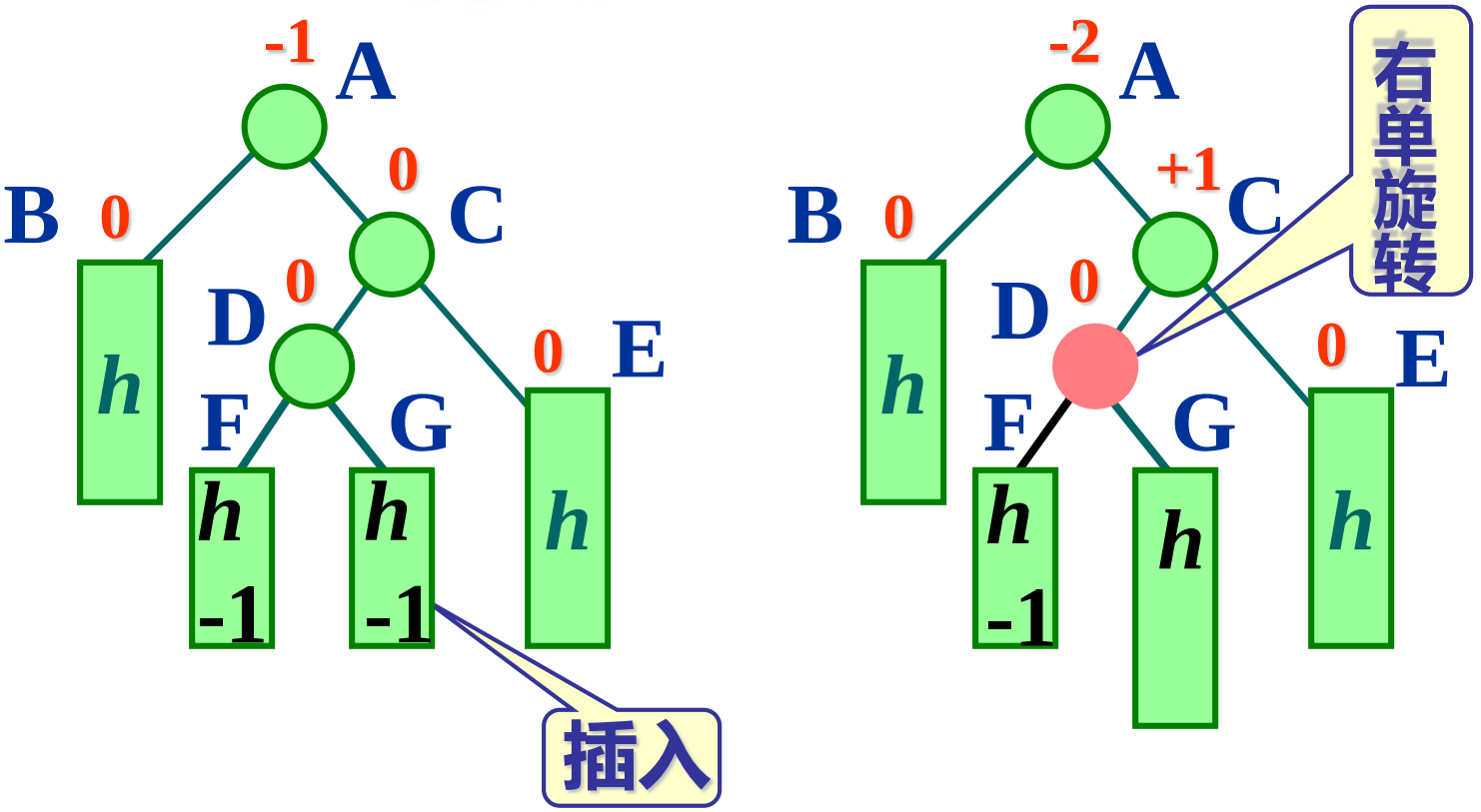


- 再以结点E为旋转轴，将结点A顺时针旋转。

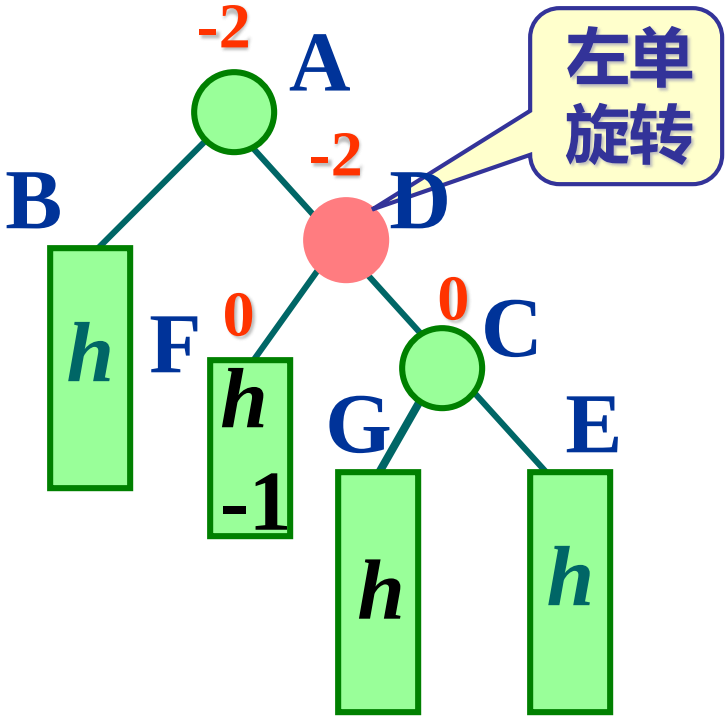
（d）先右后左双旋转（RotationRightLeft）

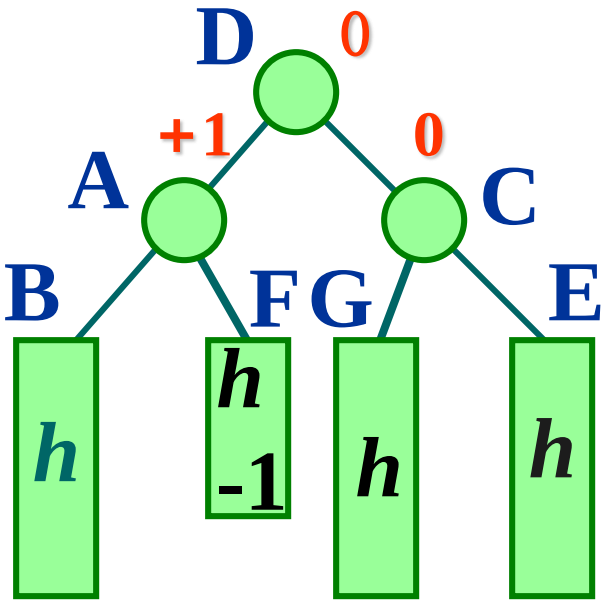


- 右左双旋转是左右双旋转的镜像
- 在子树F或G中插入新结点，该子树高度增1，A的平衡因子变为+2，发生了不平衡。



- 从结点A起沿插入路径选取3个结点A、C和D，它们位于一条形如“〉”的折线上，因此需要进行先右后左的双旋转。
- 以结点D为旋转轴，将结点C顺时针旋转。



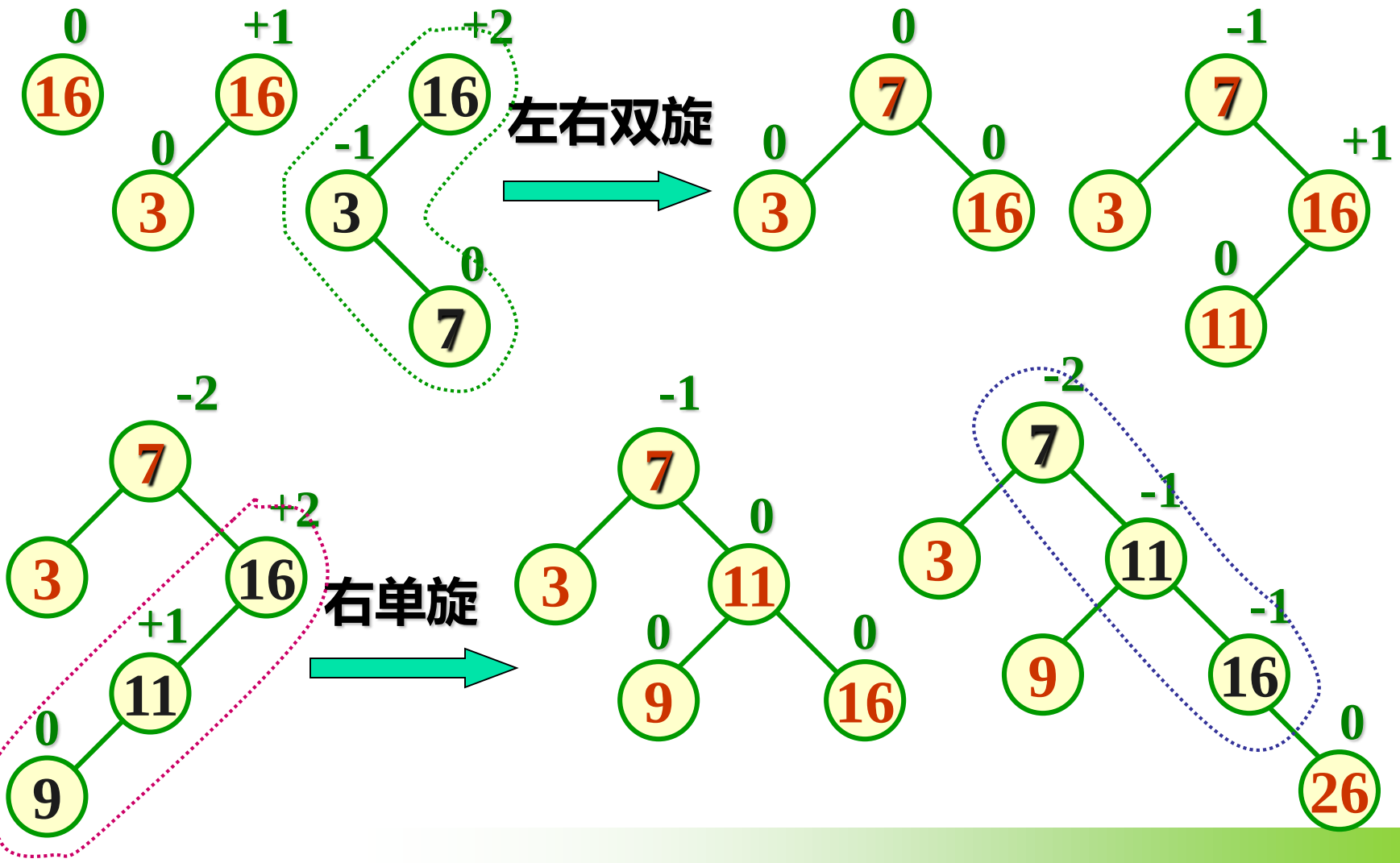


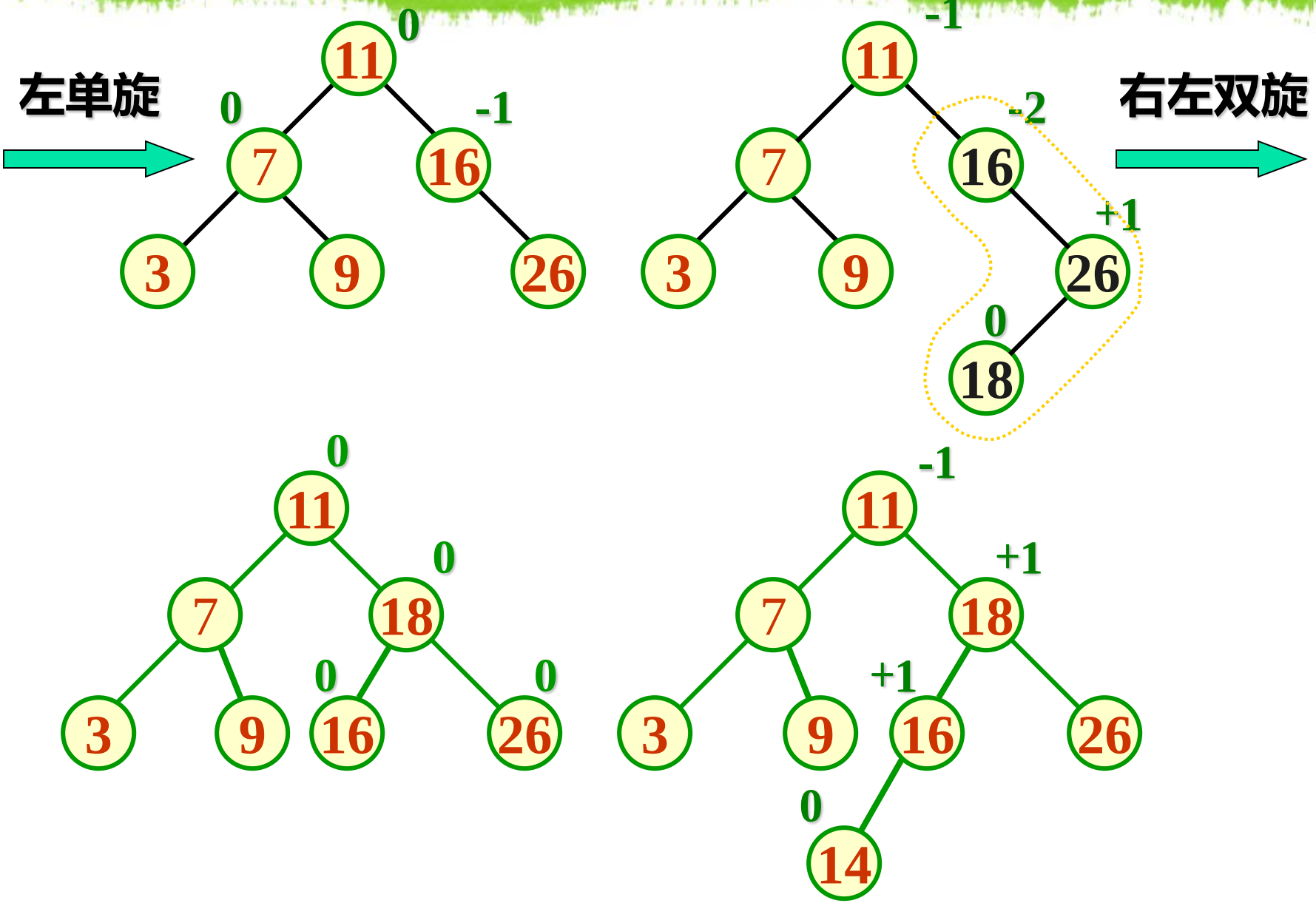
- 再以结点D为旋转轴，将结点A逆时针旋转。

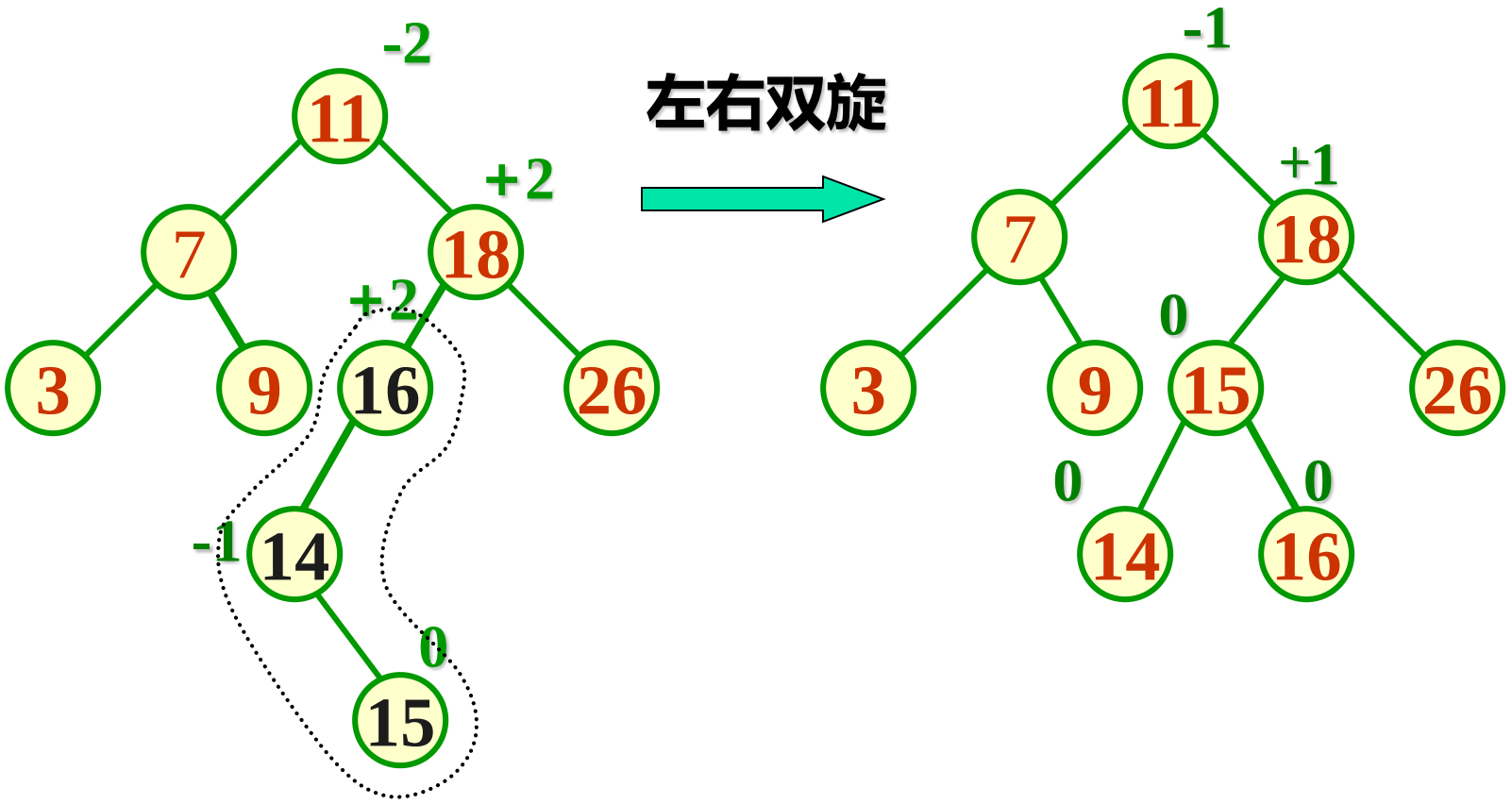
（3）AVL树的插入

- 在向一棵本来是高度平衡的AVL树中插入一个新结点时，如果树中某个结点的平衡因子的绝对值 |balance| ＞ 1，则出现了不平衡，需要做平衡化处理。
- 算法从一棵空树开始，通过输入一系列对象关键码，逐步建立AVL树。在插入新结点时使用平衡旋转方法进行平衡化处理。

例，输入关键码序列为｛16，3，7，11，9，26，18，14，15｝，插入和调整过程如下。

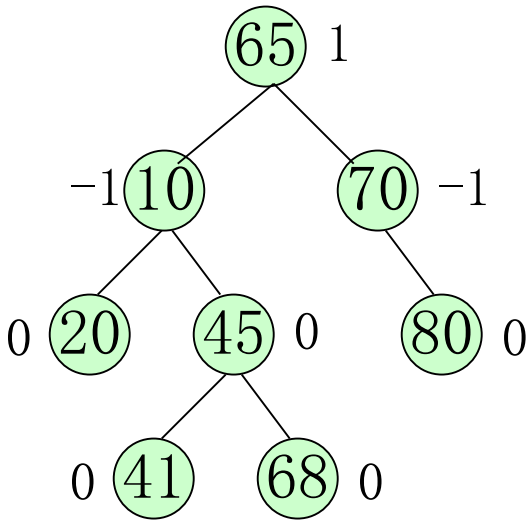



左单旋
0
11
0
7
-1
16
3
9
26
-1
11
-2
7
16
+1
3
9
26
0
18
右左双旋
0
11
0
7
18
0
0
3
9
16
26
-1
11
+1
7
18
+1
3
9
16
26
0
14

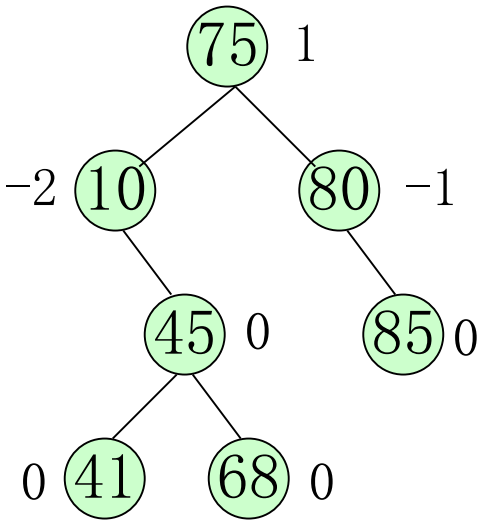


从空树开始的建树过程

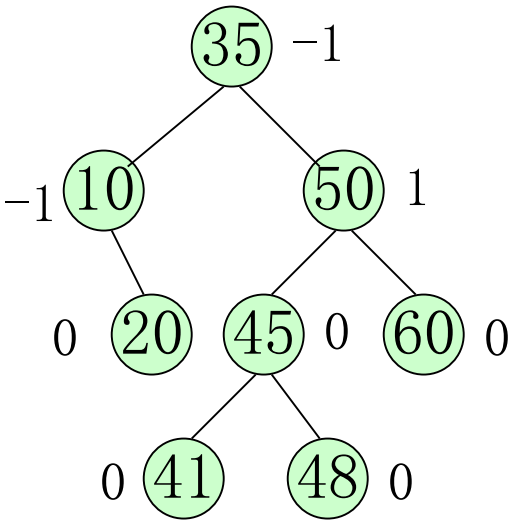
平衡二叉树、二叉排序树、平衡二叉排序树的区别：



平衡二叉树



二叉排序树



平衡二叉排序树

静态和动态查找表查找方法

静态查找表和动态查找表通过比较关键字进行查找：

（1）顺序表，对数据元素的存储一般有两种形式：

(a) 是按到来次序连续存放，查找时顺序比较查找；

(b) 按关键字的相对关系整理后以递增或递减形式连续存放，则查找时使用顺序法或二分法比较查找。

（2）二叉排序树，从根开始进行比较查找。

不足：查找时无法根据关键字的值估计数据元素可能在的位置。

9.3 哈希(Hash)表和哈希法

存储数据元素时，利用一个Hash函数根据数据元素的关键字计算出该数据元素的存储位置，查找时，也是根据给定的数据元素的关键字计算出该数据元素可能存储位置，这样一来，存储和查找的效率相当高，

哈希表也称为散列表，其数据元素的存储一般是不连续的。 通过Hash函数计算出的地址称为哈希地址或散列地址。

9.3.1 哈希表相关术语

Hash函数实现的是将一组关键字映象到一个有限的地址区间上，理想的情况是不同的关键字得到不同的地址。

设K_1和K_2为关键字，若$K_1 \neq K_2$, $H(K_1)=H(K_2)$, 则称K_1, K_2为**同义词**, K_2与K_1发生了**冲突**

例 设 H(k)=k%17

k1=5　　k2=22

∵ H(5)=5%17=5　　H(22)=22%17=5

H(5)=H(22)=5

∴ 5和22是同义词, 5和22发生冲突

9.3.1 哈希表相关术语

采用哈希表进行存储和查找需要着重考虑两个问题：

（a）选择一个好的哈希（散列）函数；

（b）选择一种解决冲突（碰撞）的方法。

9.3.2 构造哈希函数的方法

1. 直接定址法

取关键字或关键字的某个线性函数值为哈希地址

H(key)=key

H(key)=a.key+b

例1 人口统计表

序号(地址)	年 龄	人 数(万)
1	1	10.5
2	2	12.6
3	3	11.0
4	4	20.8
...	...	...
150	150	...

key

H(key)=key = 地址

H(年龄) = 年龄

例2 学生成绩表

序号(地址)	学　号	姓　名	性别	数学	外语
1	200041	刘大海	男	80	75
2	200042	王　伟	男	90	83
3	200043	吴晓英	女	82	88
4	200044	王　伟	女	80	90
	……	……	…	..	..
n	……	……	…	..	..

key

H(key) = key - 200040 = 地址

H(学号)= 学号- 200040

例3 标识符表

序号	标识符(key)
1	ABC
2	
3	CAD
4	DAT
5	
6	FOX
25	YAB
26	ZOO

H(key)=**key的第一个字母在字母表中的序号**

key	=ABC	CAD	DAT	FOX	YAB	ZOO
H(key)	=1	3	4	6	25	26

2. 除留余数法

设哈希表HT[0..m-1]的表长为m，哈希地址为key除以p所的的余数：

H(key)=key MOD p //PASCAL语言

H(key)=key % p //C语言

其中：MOD,%为“取模”或“求余”

p<=m ，p为接近m的质数(素数)，如：

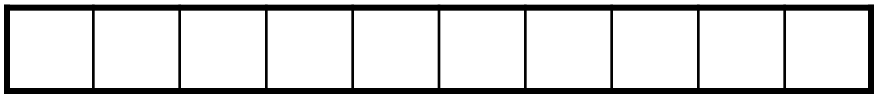
3,5,7,11,13,17,19,23,29,31,37.....

或不包含小于20的质因子的合数，如：

713(=23*31)

例1　设 m=130, 取p=127,

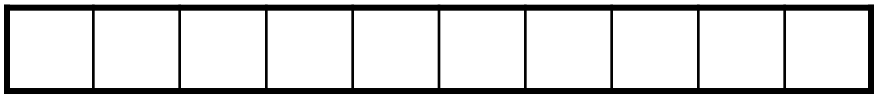
H(key)=key % 127



0　1　2　.....　129

例2　设 m=256　取 p=251

H(key)=key % 251



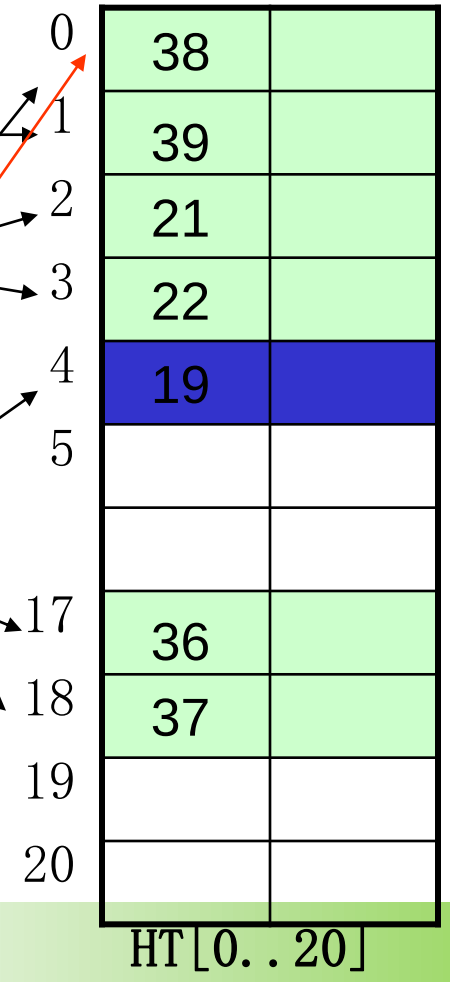
0　1　2　.....　255

例　设哈希表的地址范围为0～20，哈希函数为

H(K)=K % 19

输入关键字序列：39,22,21,37,36,38,19,解决冲突的方法为**线性探测再散列(哈希)**，构造哈希表HT[0..20]。

关键字K	H(K)=K%19
39	39%19=1
22	22%19=3
21	21%19=2
37	37%19=18
36	36%19=17
38	38%19=0
19	19%19=0



19与38冲突，再与39，21，22冲突，存入HT[4]

再输入17, 56, 55

17%19=17

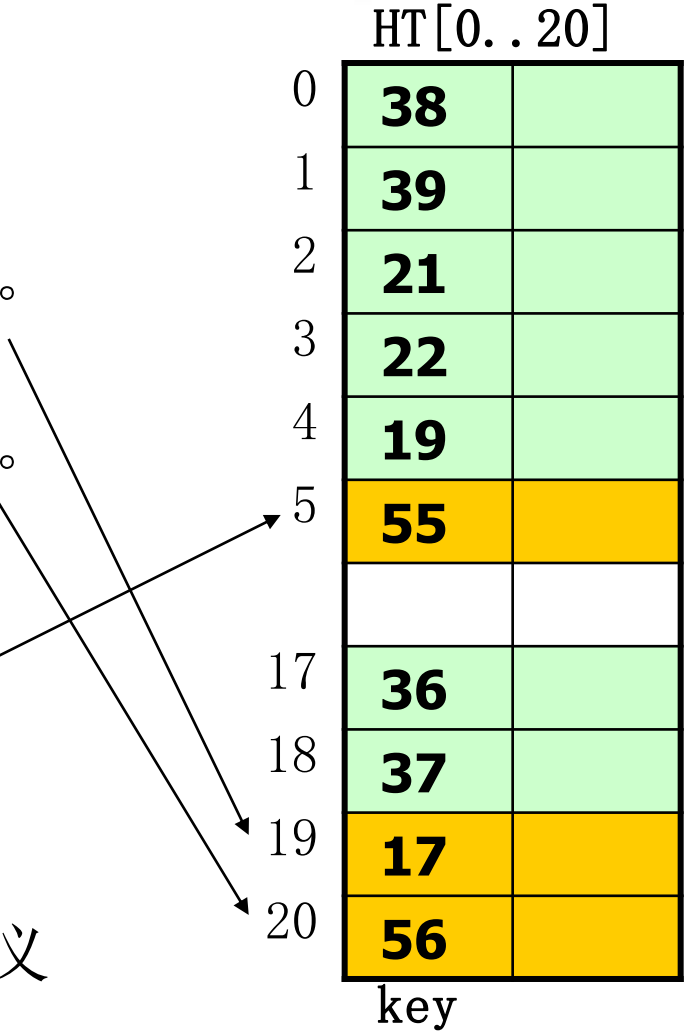
17与36冲突, 再与37冲突, 存入HT[19]。

56%19=18

56与37冲突, 再与17冲突, 存入HT[20]。

55%19=17

55与36冲突, 再与37, 17, 56冲突, 再与38, 39, 21, 22, 19冲突, 存入HT[5]。



对于H(k)=k % 19，所有的19a+b为同义词, 0<b<19

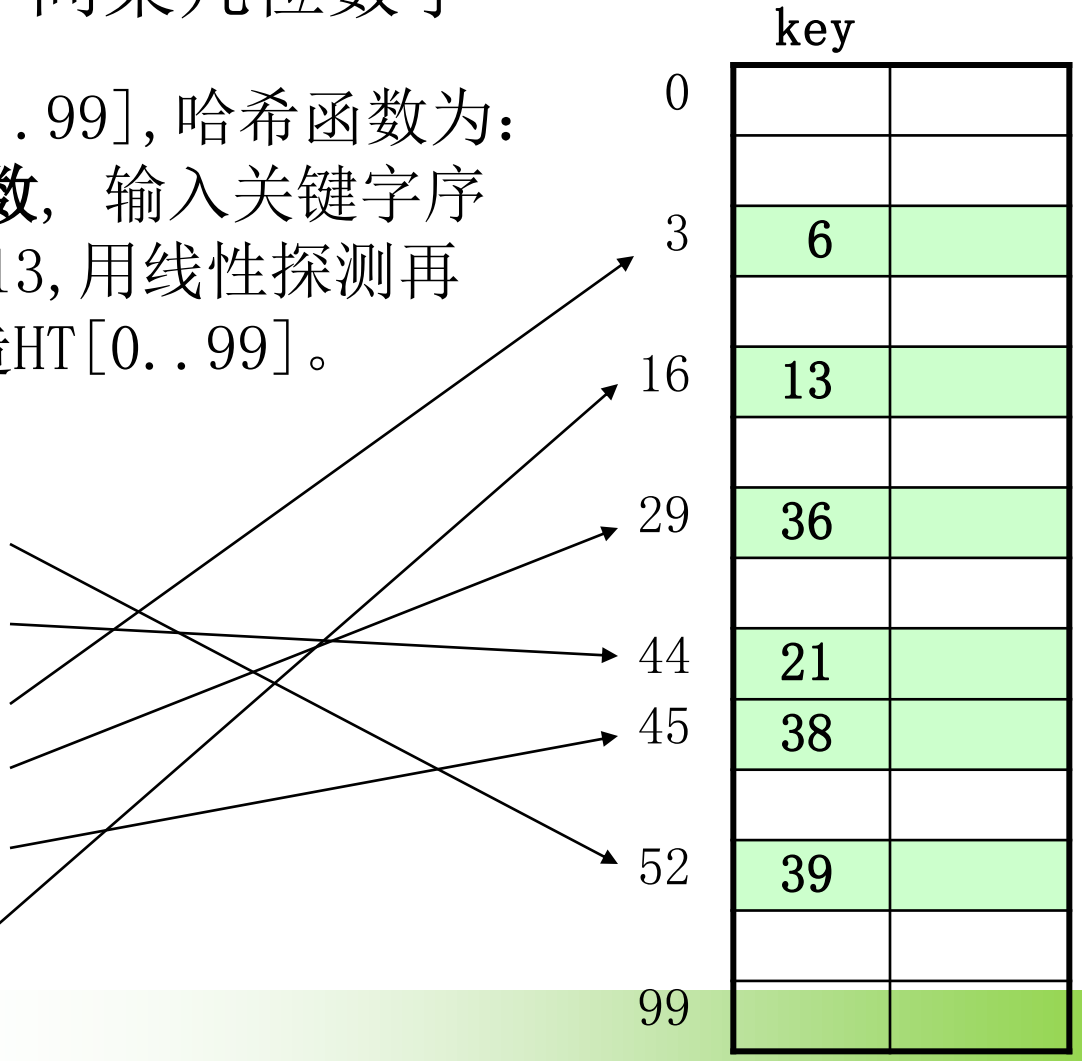
如：5, 24, 43, 62, 81, ...

3. **平方取中法----**取关键字平方后的中间某几位为哈希地址，即：

$H(k)$=取k^2的中间某几位数字

例. 设哈希表为HT[0..99]，哈希函数为：**$H(K)$=取k^2的中间2位数**，输入关键字序列：39, 21, 6, 36, 38, 13，用线性探测再散列法解决冲突，构造HT[0..99]。

K	k^2	H(K)
39	1521	52
21	0441	44
6	0036	03
36	1296	29
38	1444	44
13	0169	16



4.折叠法

将关键字分割成位数相同的几部分,然后取这几部分的叠加和作为哈希地址。

(1)边界折叠法

设表地址范围为0～999

- k1=056439527

  650 + 439 + 725 =1814

  H(k1)=814

- k2=123486790

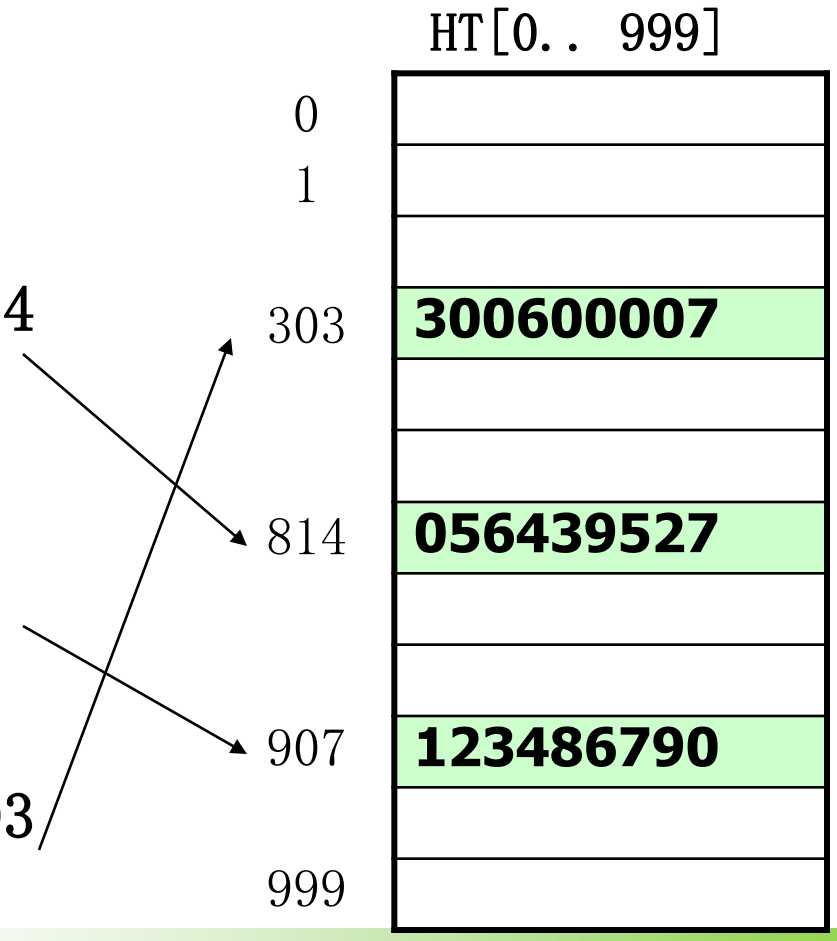
  321 + 486 + 097 =907

  H(k2)=907

- k3=300600007

  003 + 600 + 700 =1303

  H(k3)=303



4.折叠法

将关键字分割成位数相同的几部分,然后取这几部分的叠加和作为哈希地址。

(2)移位折叠法(移位法)

设表地址范围为0～999

- k1=056439527

  056 + 439 + 527 =1022

  H(k1)=022

- k2=123486790

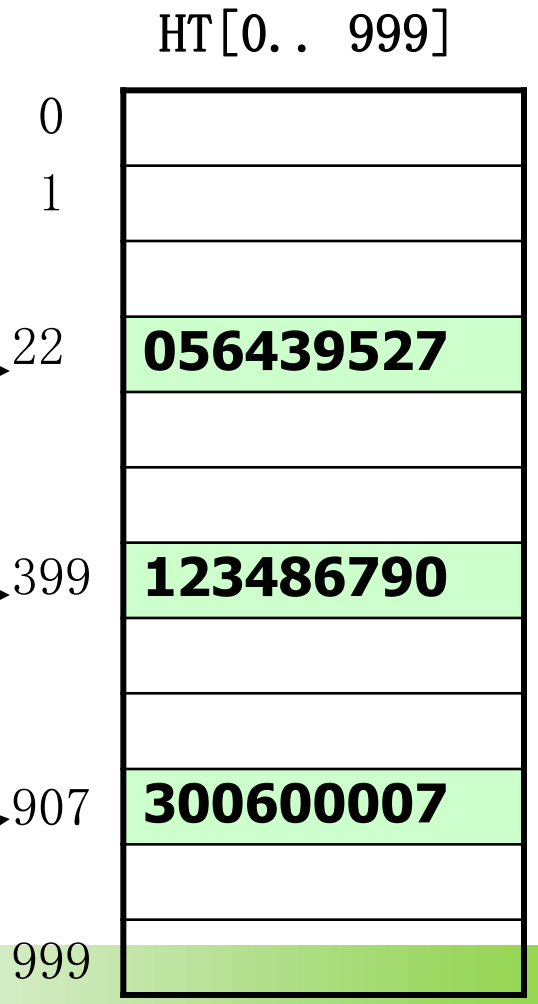
  123 + 486 + 790 =1399

  H(k2)=399

- k3=300600007

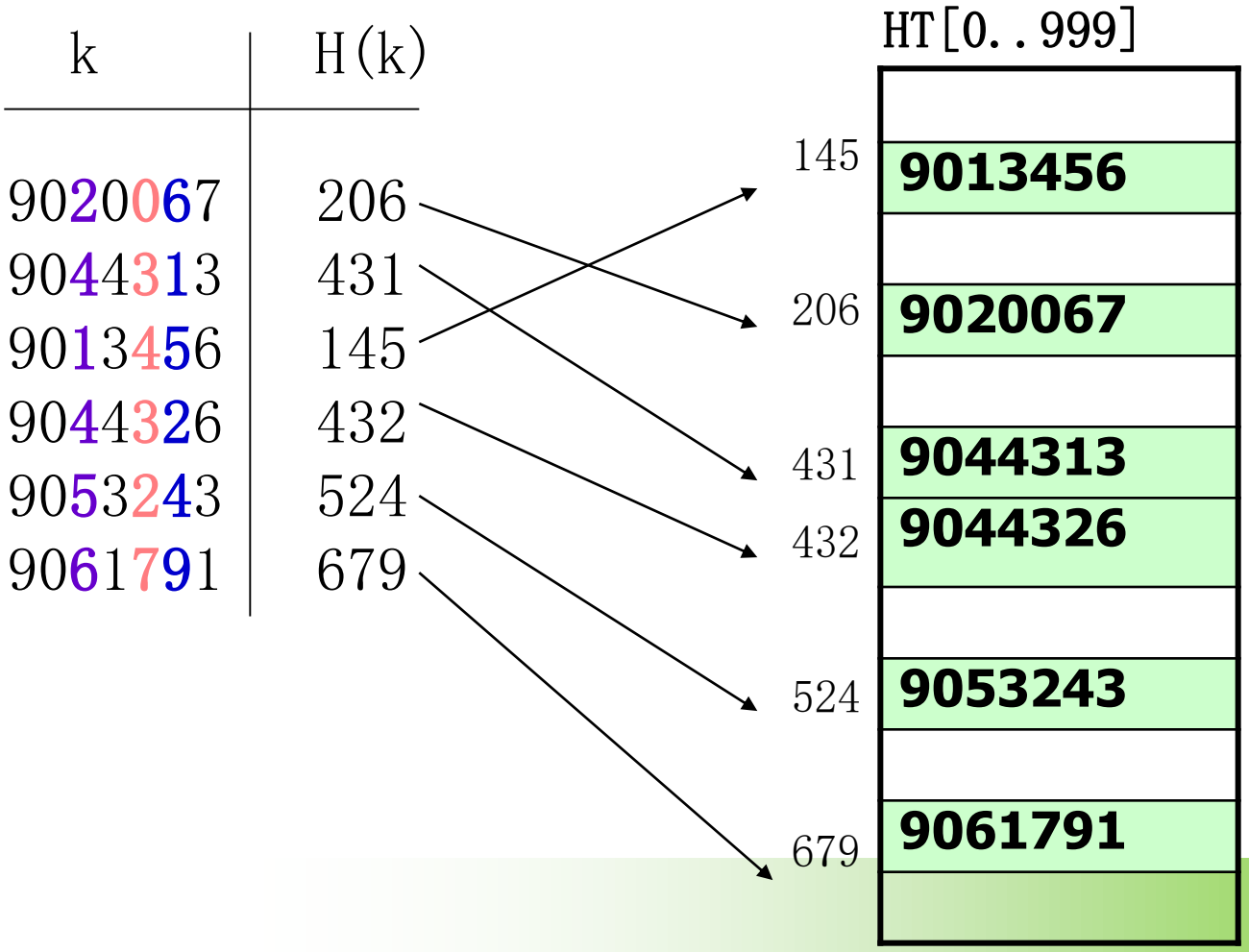
  300 + 600 + 007 =907

  H(k3)=907



5. 数字分析法

设哈希表中可能出现的关键字都是事先知道的，则可取关键字的**若干分布均匀的位组成哈希地址。**



6. 随机数法

H(key)=random(key)

random(key)为产生伪随机数的函数

7. 灵活构造哈希函数

例. 设哈希表为HT[0..40],哈希函数为:

H(K)=取k^2的中间2位数*40/99

其中40/99将其00～99压缩到00～40之内,

输入关键字序列：39,21,6,36,38,13,

用**线性探测再散列**法解决冲突。

K	k^2	H(K)
39	1521	52*40/99=21
21	0441	44*40/99=17
6	0036	03*40/99=1
77	5929	92*40/99=37
38	1444	44*40/99=17
13	0169	16*40/99=6

	key
0	
1	**6**
3	
6	**13**
17	**21**
18	**38**
21	**39**
37	**77**
40	

9.3.3 如何解决冲突

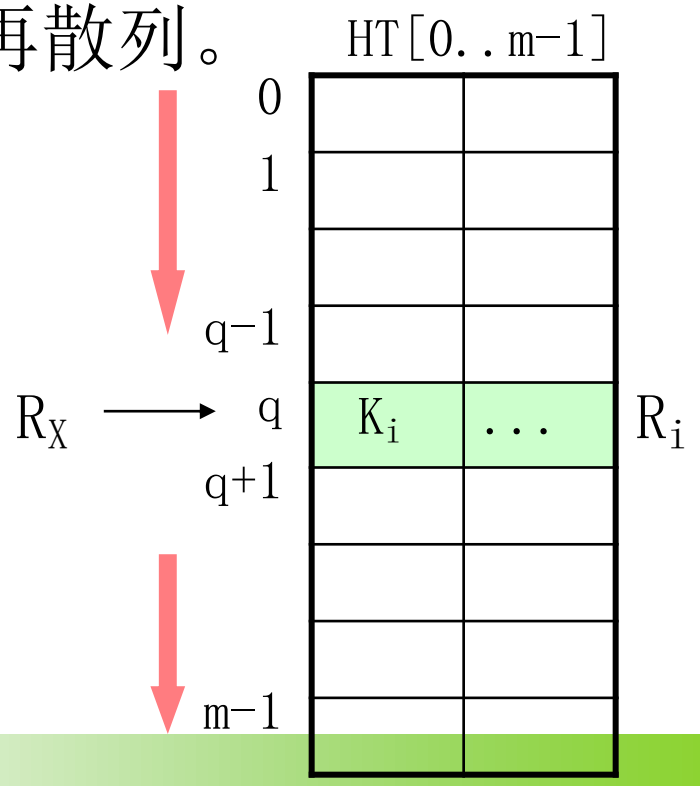
1. 开放地址法(开式寻址法)

假定记录R_i, R_X的关键字K_i, K_X为同义词，散列地址为q, R_i已存入HT[0..m-1]中的HT[q]中，则R_X 存入HT中的某个空位上。依次在地址:

q+1, q+2, . . . , m-1, 0, 1, . . . , q-1

中寻找一个空位，叫做线性探测再散列。

(1) 线性探测再散列



课堂练习：设 **H(k)=k的首字母在字母表中的序号**,用线性探测再散列法解决冲突，依次用下列关键字，构造哈希表 HT[0..28]。

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4

HT[0..28]

0	
1	
2	
3	
4	
5	
6	
7	
12	
13	
25	
26	
27	
28	

例　设 H(k)=**k的首字母在字母表中的序号**，用线性探测再散列 法解决冲突，依次用下列关键字，构造哈希表HT[0..28]。

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4

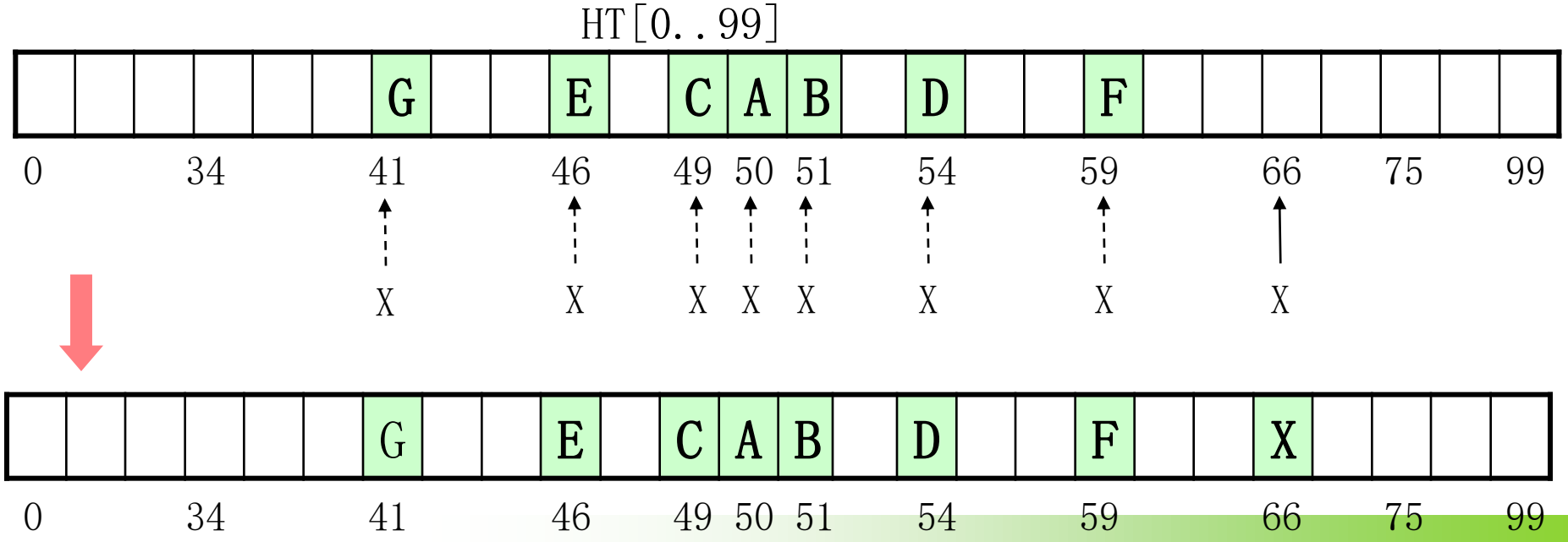
HT[0..28]

0	YES
1	A
2	ANT
3	CAD
4	DEC
5	DAB
6	ZY
7	DE
12	LL
25	YY
26	ZMN
27	ZE
28	ZOO

(2) 二次探测再散列

假定记录R_i和R_j的关键字K_i和K_j为同义词，散列地址为q，R_i已存入HT[0..m-1]中的HT[q]中。若依次在地址 $q+1^2, q-1^2, q+2^2, q-2^2, \ldots, q+i^2, q-i^2, \ldots$ 中寻找一个空位，叫做二次探测再散列。

例：设记录X和A为同义词，散列地址为50，二次探测再散列的地址序列为：51,49,54,46,59,41,66,34,75,.....

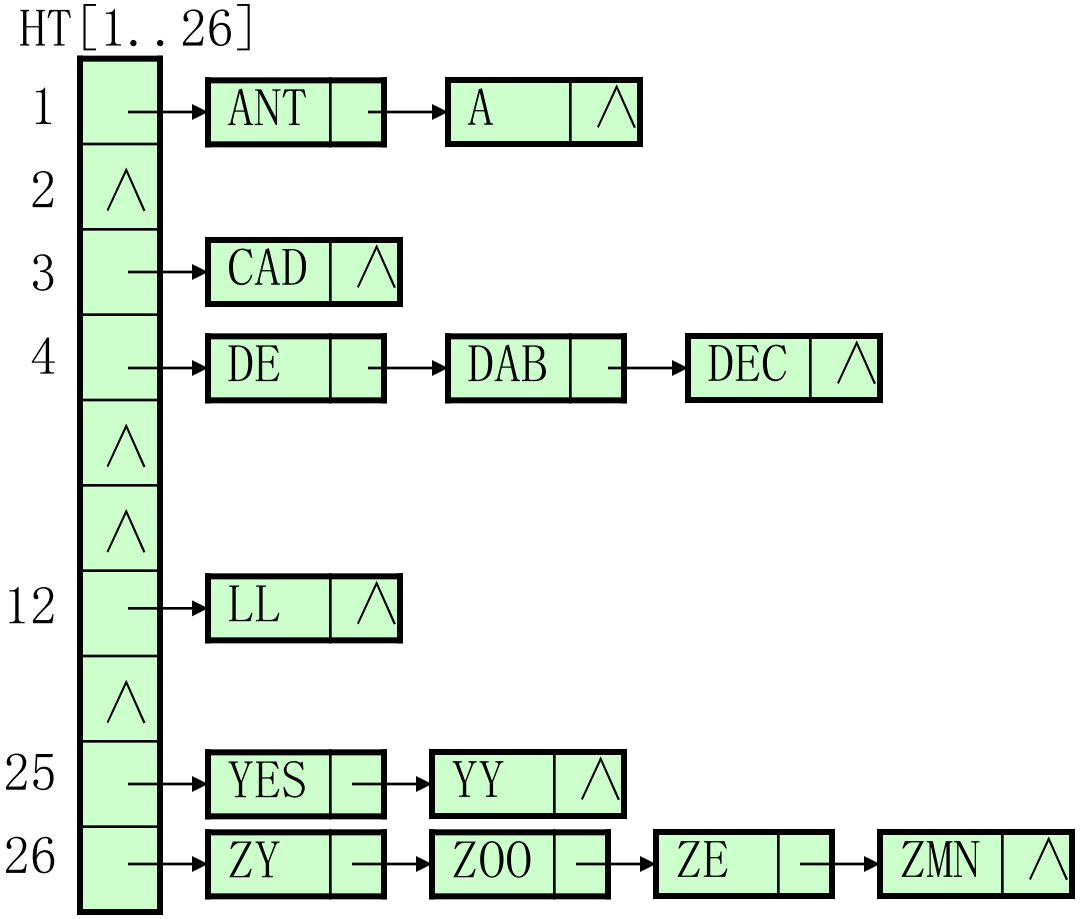


2. 链地址法

将关键字为同义词的所有记录存入同一链表中。（表头插入）

例　**设H(k)=k的首字母在字母表中的序号**,用下列关键字造哈希表HT[1..26]

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4

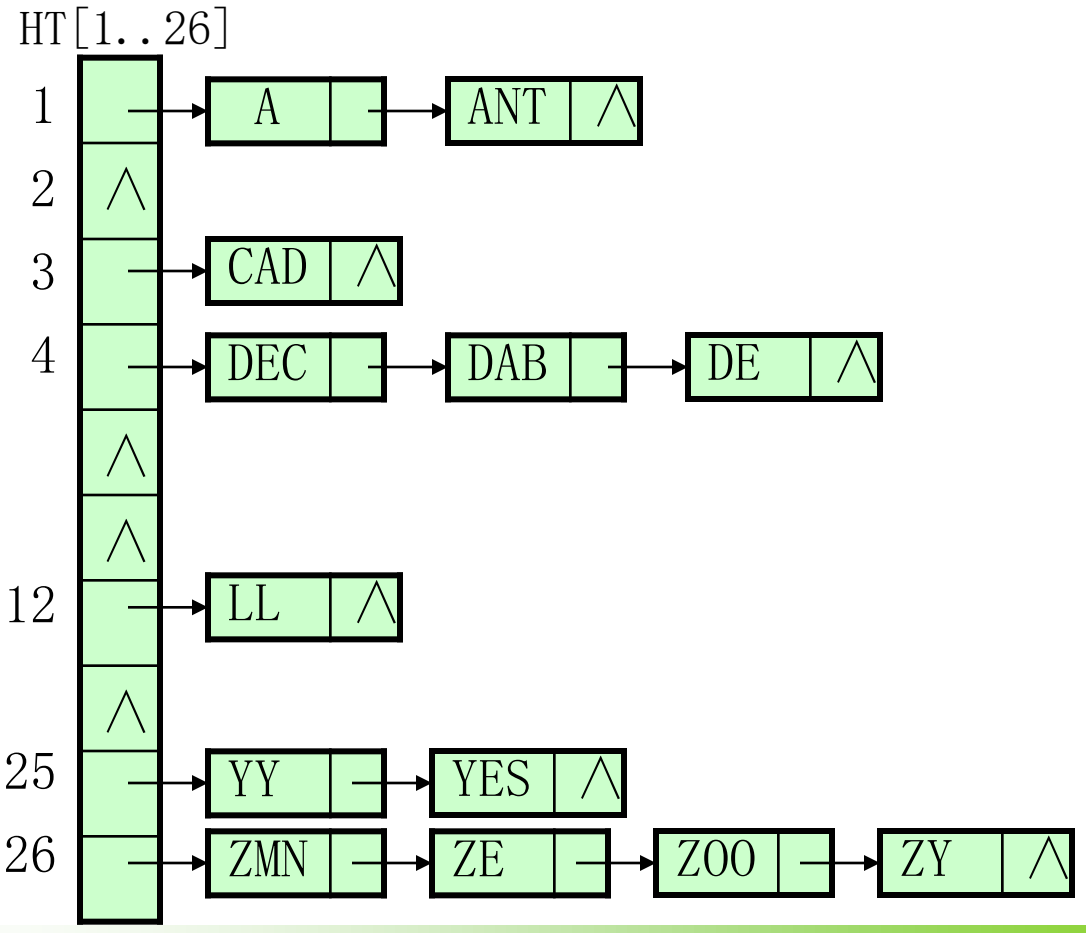


2. 链地址法

将关键字为同义词的所有记录存入同一链表中。(表尾插入)

例　设**H(k)=k的首字母在字母表中的序号**,用下列关键字造哈希表 HT[1..26]

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4



3. 建立公共溢出区

将发生冲突的所有记录存入一个**公共溢出表OT[0..v]**中。

例　设**H(k)=k的首字母在字母表中的序号,**用下列关键字生成基本表HT[1..26]和溢出表OT[0..50]

	k	H(k)
1	A	1
2	DEC	4
3	ZMN	26
4	DAB	4
5	ZE	26
6	ANT	1
7	YY	25
8	ZOO	26
9	CAD	3
10	YES	25
11	ZY	26
12	LL	12
13	DE	4

HT[1..26]

1	A
2	
3	CAD
4	DEC
12	LL
25	YY
26	ZMN

OT[0..50]

1	DAB
2	ZE
3	ANT
4	ZOO
5	YES
6	ZY
7	DE
50	

4. **再哈希法**

发生冲突时，使用下一个哈希函数计算哈希地址：

j1=H1(K)； j2=H2(K)； j3=H3(K)；.....

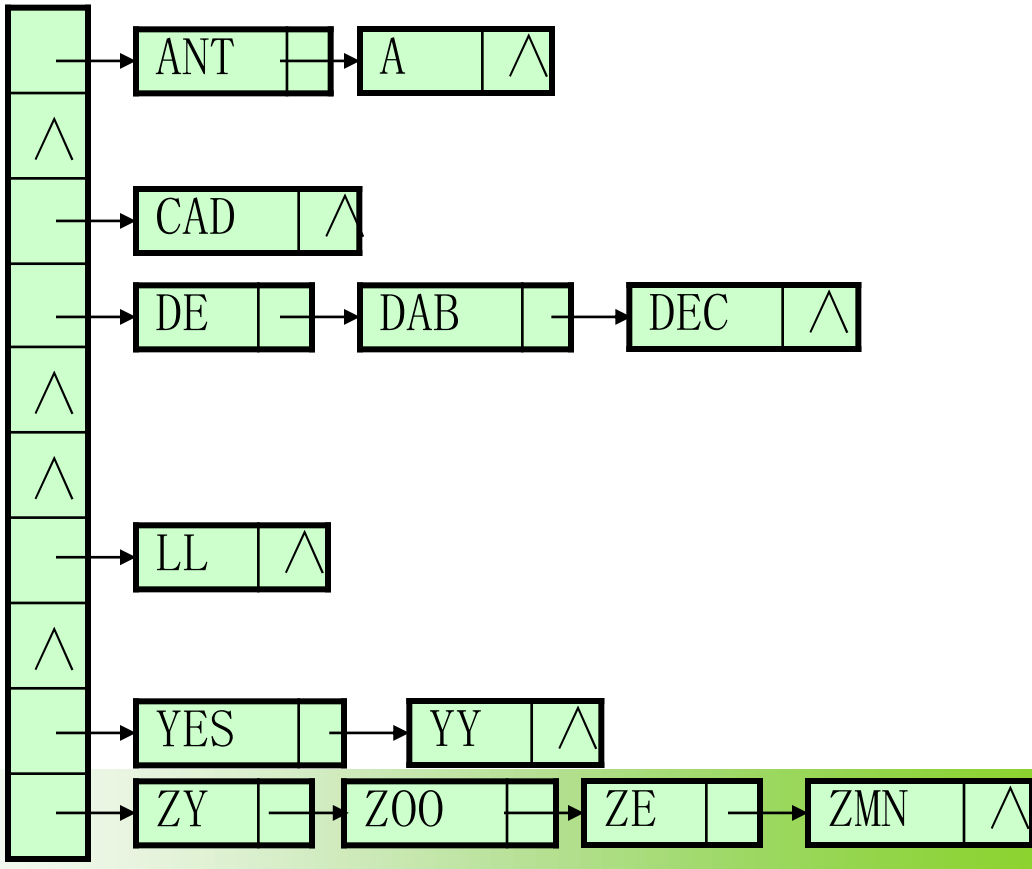
9.3.4 哈希表的查找及其分析

例1(**链地址法**)假定每个记录的查找概率相等，查找成功时的平均查找长度：

ASL=(1+2+1+1+2+3+1+1+2+1+2+3+4)/13

= 24/13

≈1.85



例2 （给定线性探测再散列得到的哈希表如右所示)假定每个记录的查找概率相等，查找成功时的平均查找长度.

关键字K	H(K)	比较次数
YES	25	5
A	1	1
ANT	1	2
CAD	3	1
DEC	4	1
DAB	4	2
ZY	26	10
DE	4	4
LL	12	1
YY	25	1
ZMN	26	1
ZE	26	2
ZOO	26	3
	合计	34

$ASL_{succ}=34/13\approx2.62$

HT[0..28]

0	YES
1	A
2	ANT
3	CAD
4	DEC
5	DAB
6	ZY
7	DE
12	LL
13	
25	YY
26	ZMN
27	ZE
28	ZOO

例2(续) (线性探测再散列)查找不成功时的平均查找长度. 需统计不成功时比较次数

H(K)	比较次数	H(K)	比较次数
0	8		
1	7	15	0
2	6	16	0
3	5	17	0
4	4	18	0
5	3	19	0
6	2	20	0
7	1	21	0
8	0	22	0
9	0	23	0
10	0	24	0
11	0	25	12
12	1		
13	0		
14	0		

$ASL_{unsucc}=49/26 \approx 1.885$

HT[0..28]

0	YES
1	A
2	ANT
3	CAD
4	DEC
5	DAB
6	ZY
7	DE
12	LL
13	
25	YY
26	ZMN
27	ZE
28	ZOO

一般情况：平均查找长度依赖于哈希表的装填因子：

α=(表中填入记录数)/(哈希表的长度)

解决冲突的方法	平均查找长度	
	查找成功	查找失败
线性探测再散列	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}[1+\frac{1}{(1-\alpha)^2}]$
二次探测再散列	$\frac{\ln(1-\alpha)}{\alpha}$	$\frac{1}{1-\alpha}$
链地址	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$