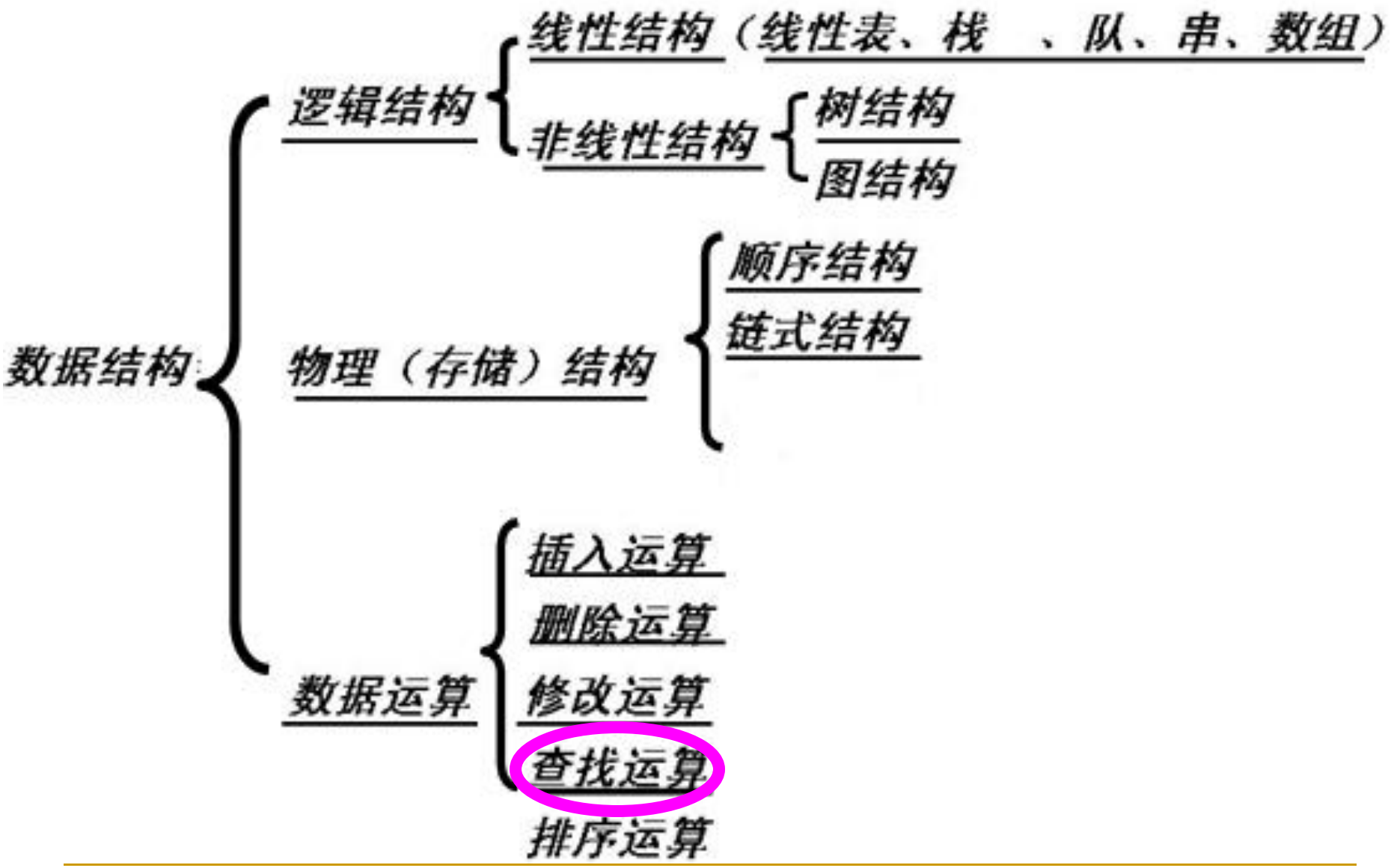
第七章 查找


数据结构
逻辑结构
线性结构（线性表、栈 、队、串、数组）
非线性结构
树结构
图结构
物理（存储）结构
顺序结构
链式结构
数据运算
插入运算
删除运算
修改运算
查找运算
排序运算

教学内容

7.1　查找的基本概念

7.2　线性表的查找

7.3　树表的查找

7.4　哈希表的查找

教学目标

1. 熟练掌握顺序表和有序表（折半查找）的查找算法及其性能分析方法；
2. 熟练掌握二叉排序树的构造和查找算法及其性能分析方法；
3. 掌握二叉排序树的插入算法，掌握二叉排序树的删除方法；
4. 熟练掌握哈希函数（除留余数法）的构造
5. 熟练掌握哈希函数解决冲突的方法及其特点

7.1 查找的基本概念

是一种数据结构

- 查找表:
  由同一类型的数据元素（或记录）构成的集合
- 静态查找表:
  查找的同时对查找表不做修改操作（如插入和删除）
- 动态查找表:
  查找的同时对查找表具有修改操作
- 关键字
  记录中某个数据项的值，可用来识别一个记录
- 主关键字:
  唯一标识数据元素
- 次关键字:
  可以标识若干个数据元素

查找算法的评价指标

关键字的平均比较次数，也称平均搜索长度 *ASL*(*Average Search Length*)

$$ASL = \sum_{i=1}^{n} p_i c_i$$

n：记录的个数

pi：查找第i个记录的概率（通常认为pi =1/n）

ci：找到第i个记录所需的比较次数

7.2 线性表的查找

1.顺序表的查找（线性查找）

2.有序表的查找（折半查找）

1. 顺序查找

应用范围：

顺序表或线性链表表示的静态查找表

表内元素之间无序

顺序表的表示

```
typedef struct {
        ElemType   *R; //表基址
        int        length;    //表长
}SSTable;
```

第2章在顺序表L中查找值为e的数据元素

```
int LocateELem(SqList L,ElemType e)
{  for (i=0;i< L.length;i++)
      if (L.elem[i]==e) return i+1;
  return 0;}
```

改进：把待查关键字key存入表头（“哨兵”），从后向前逐个比较，可免去查找过程中每一步都要检测是否查找完毕，加快速度。

```
int Search_Seq( SSTable  ST , KeyType  key ){
  //若成功返回其位置信息，否则返回0
 ST.R[0].key =key;
 for( i=ST.length; ST.R[ i ].key!=key;  - - i  );
 //不用for(i=n; i>0; - -i) 或 for(i=1; i<=n; i++)
   return i;


}
```

顺序查找的性能分析

- 空间复杂度：一个辅助空间。
- 时间复杂度：

1） 查找成功时的平均查找长度

设表中各记录查找概率相等

$ASL_s(n)=(1+2+\ ...\ +n)/n\ =(n+1)/2$

2)查找不成功时的平均查找长度　　$ASL_f=n+1$

顺序查找算法有特点

- 算法简单，对表结构无任何要求（顺序和链式）
- n很大时查找效率较低

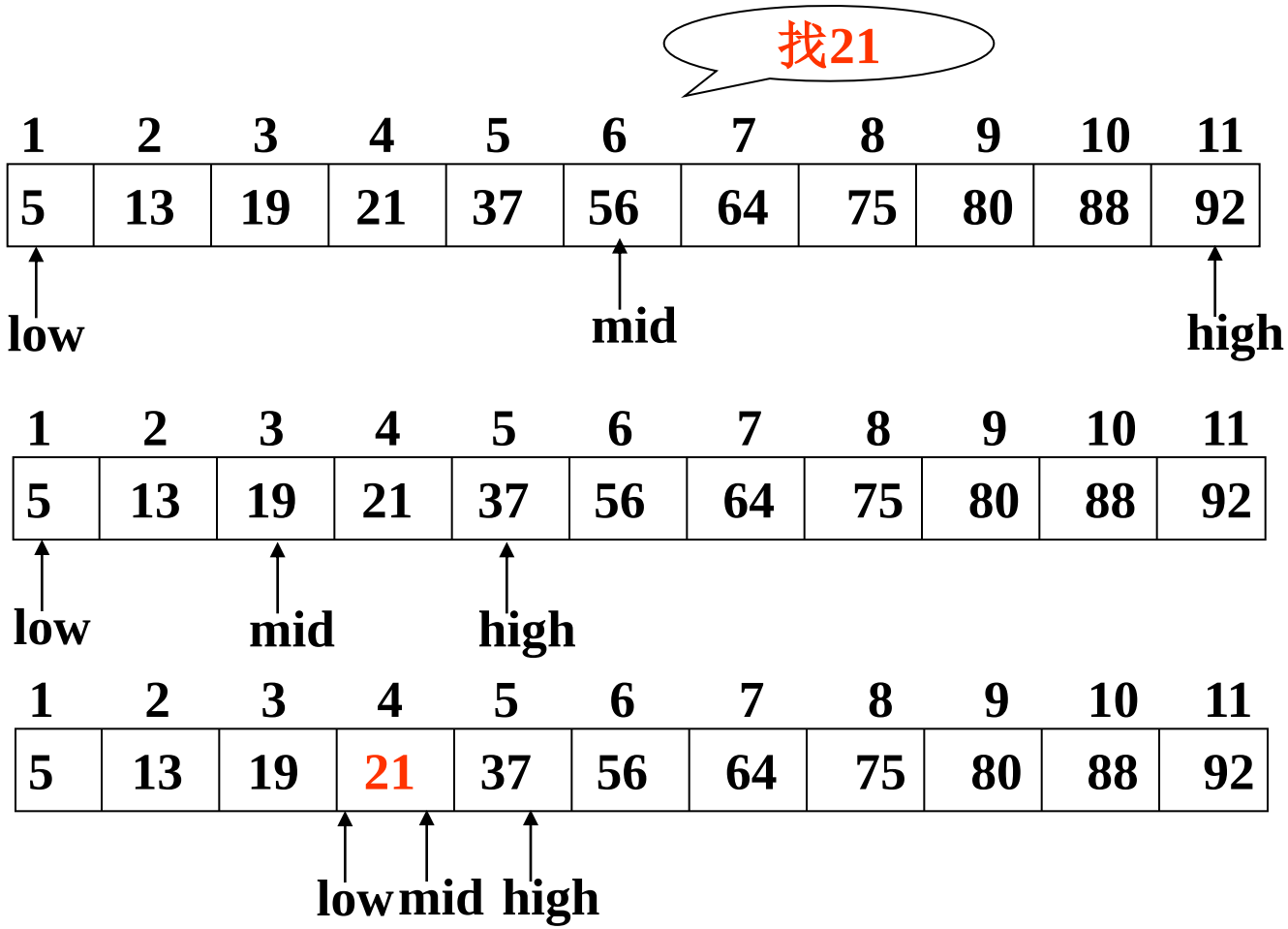
- 改进措施：非等概率查找时，可按照查找概率进行排序。

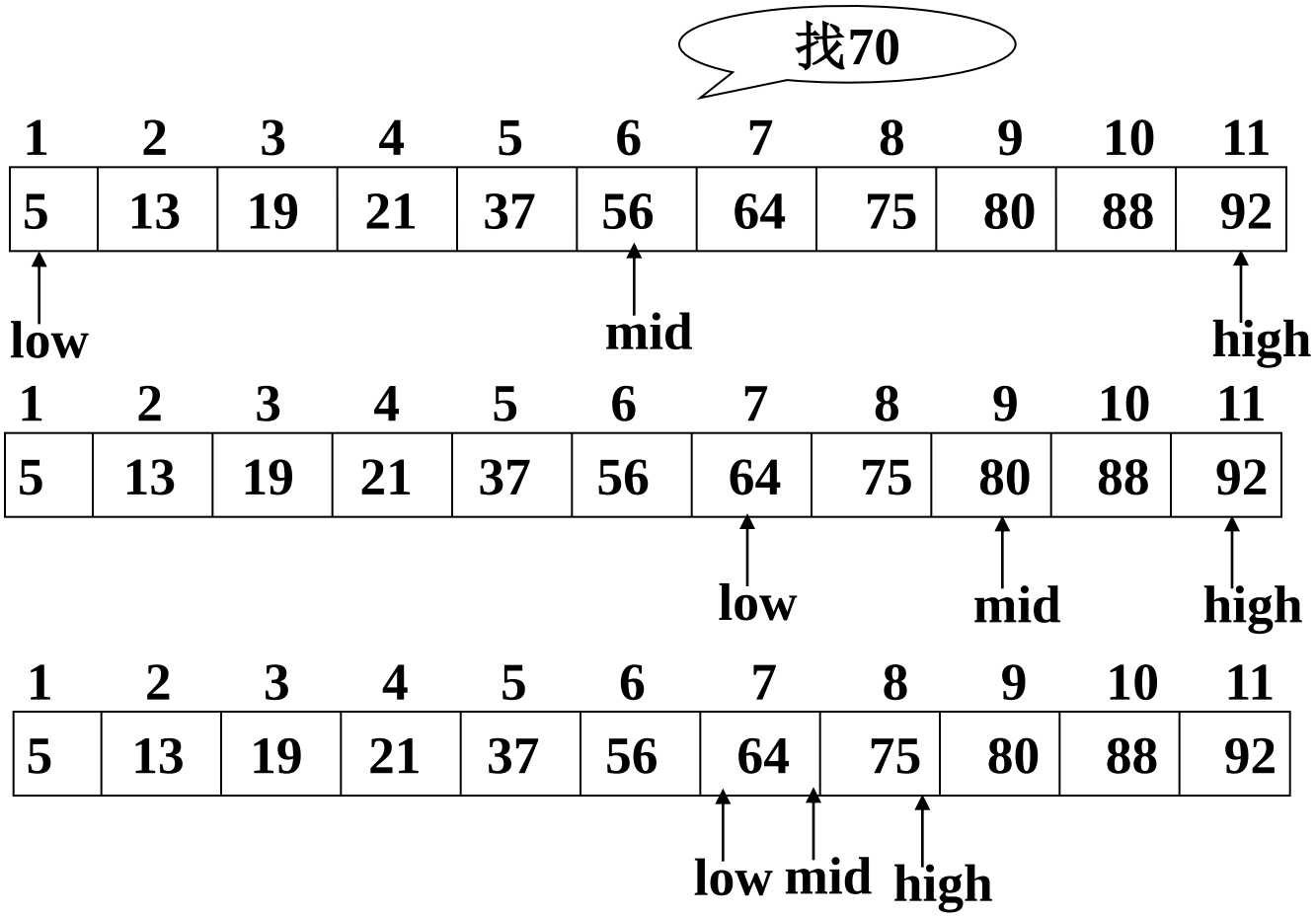
练习:判断对错

n个数存在一维数组A[1..n]中，在进行顺序查找时，这n个数的排列有序或无序其平均查找长度ASL不同。

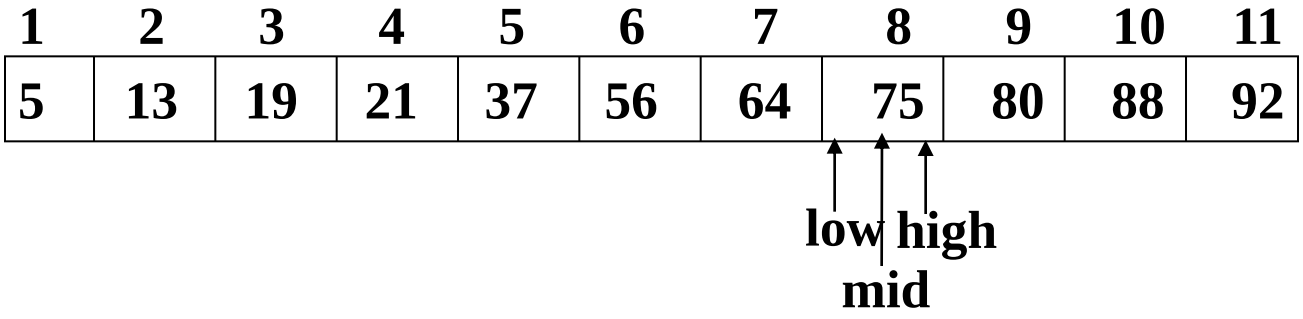
查找概率相等时，ASL相同；
查找概率不等时，如果从前向后查找，则按查找概率由大到小排列的有序表其ASL要比无序表ASL小。

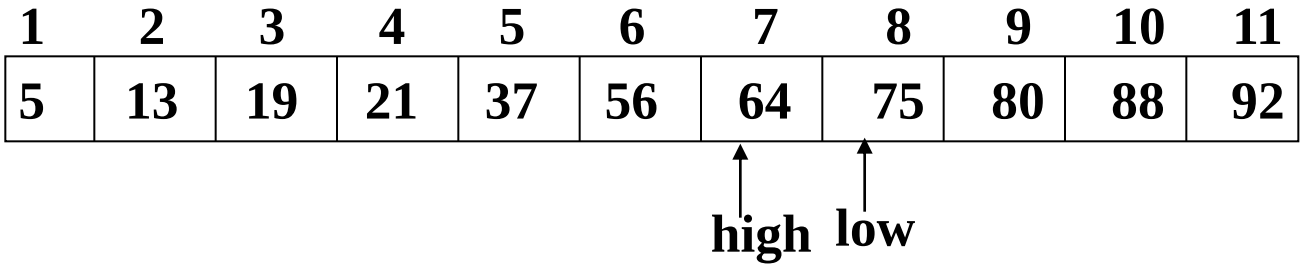
2. 有序表的查找（折半查找）




找70
1 2 3 4 5 6 7 8 9 10 11
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
1 2 3 4 5 6 7 8 9 10 11
5 13 19 21 37 56 64 75 80 88 92
low
mid
high
1 2 3 4 5 6 7 8 9 10 11
5 13 19 21 37 56 64 75 80 88 92
low
mid
high

直至low>high时，查找失败





折半查找（非递归算法）

- 设表长为n，low、high和mid分别指向待查元素所在区间的上界、下界和中点,k为给定值
- 初始时，令low=1,high=n,mid=⌊(low+high)/2⌋
- 让k与mid指向的记录比较
  - 若k==R[mid].key，查找成功
  - 若k<R[mid].key，则high=mid-1
  - 若k>R[mid].key，则low=mid+1
- 重复上述操作，直至low>high时，查找失败

【算法描述】

```
int Search_Bin(SSTable ST,KeyType key){
//若找到，则函数值为该元素在表中的位置，否则为0
    low=1;high=ST.length;
    while(low<=high){
        mid=(low+high)/2;
        if(key==ST.R[mid].key) return mid;
        else if(key<ST.R[mid].key) high=mid-1;//前一子表查找
        else low=mid+1;                          //后一子表查找
    }
    return 0;              //表中不存在待查元素
}
```

折半查找（递归算法）

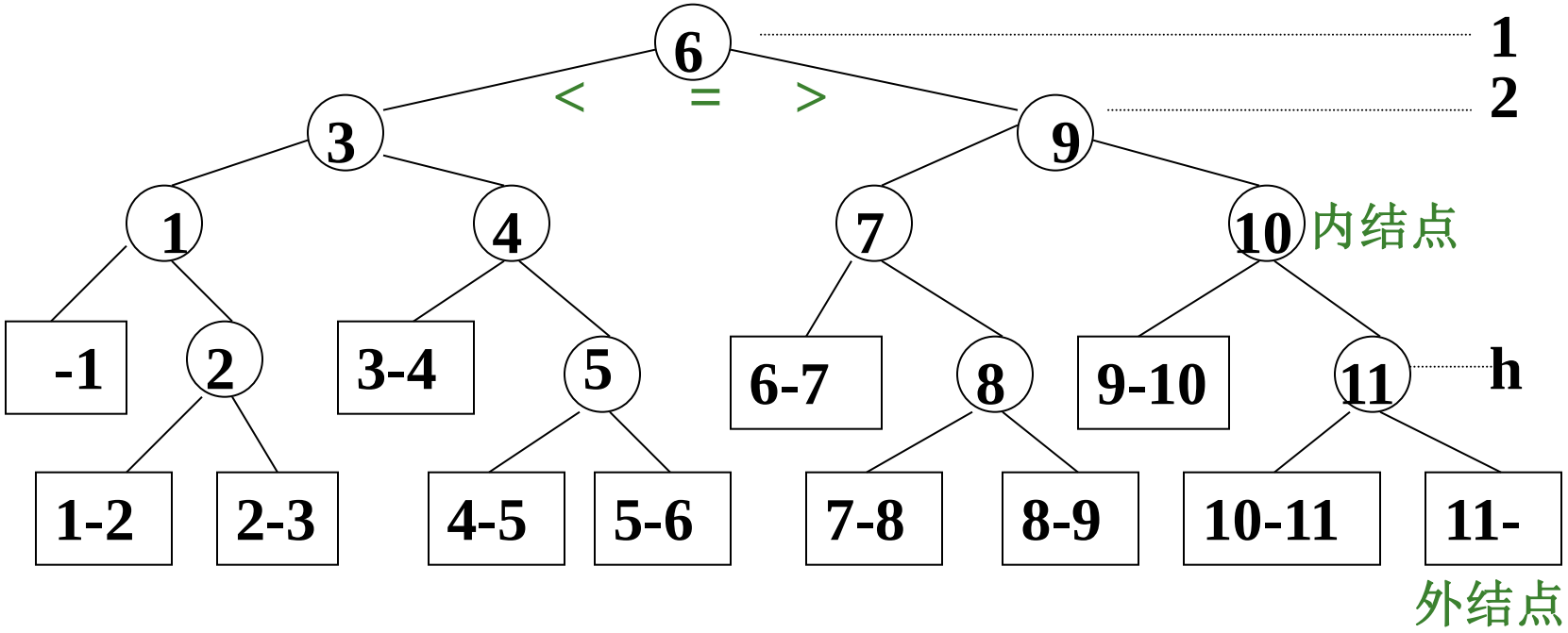
```
int Search_Bin (SSTable ST, keyType key, int low, int high)
{
  if(low>high) return 0;   //查找不到时返回0
  mid=(low+high)/2;
  if(key等于ST.elem[mid].key)  return mid;
  else if(key小于ST.elem[mid].key)
    ……..//递归
  else…….  //递归
}
```

Facebook
- Software Engineer position

Given the numbers 1 to 1000, what is the minimum number of guesses needed to find a specific number, if you are given the hint "higher" or "lower" for each guess you make ??

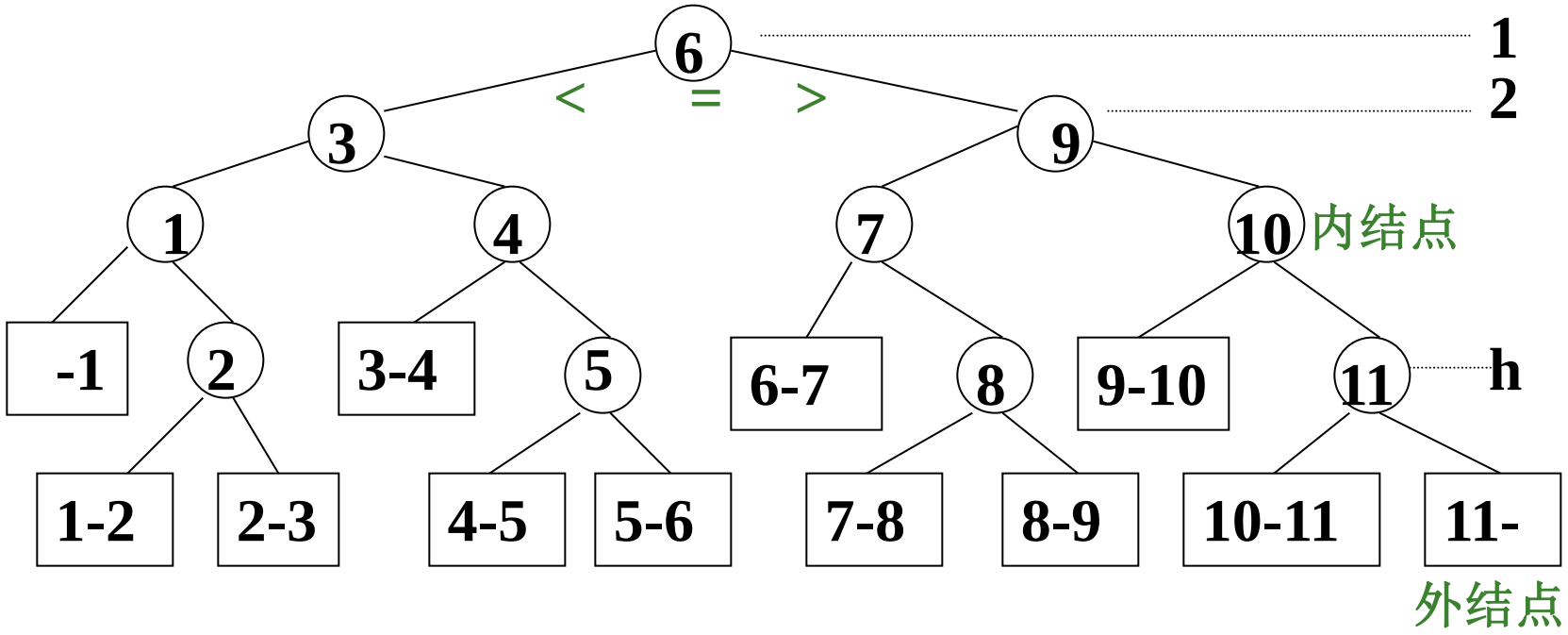
折半查找的性能分析—判定树

1	2	3	4	5	6	7	8	9	10	11
5	13	19	21	37	56	64	75	80	88	92

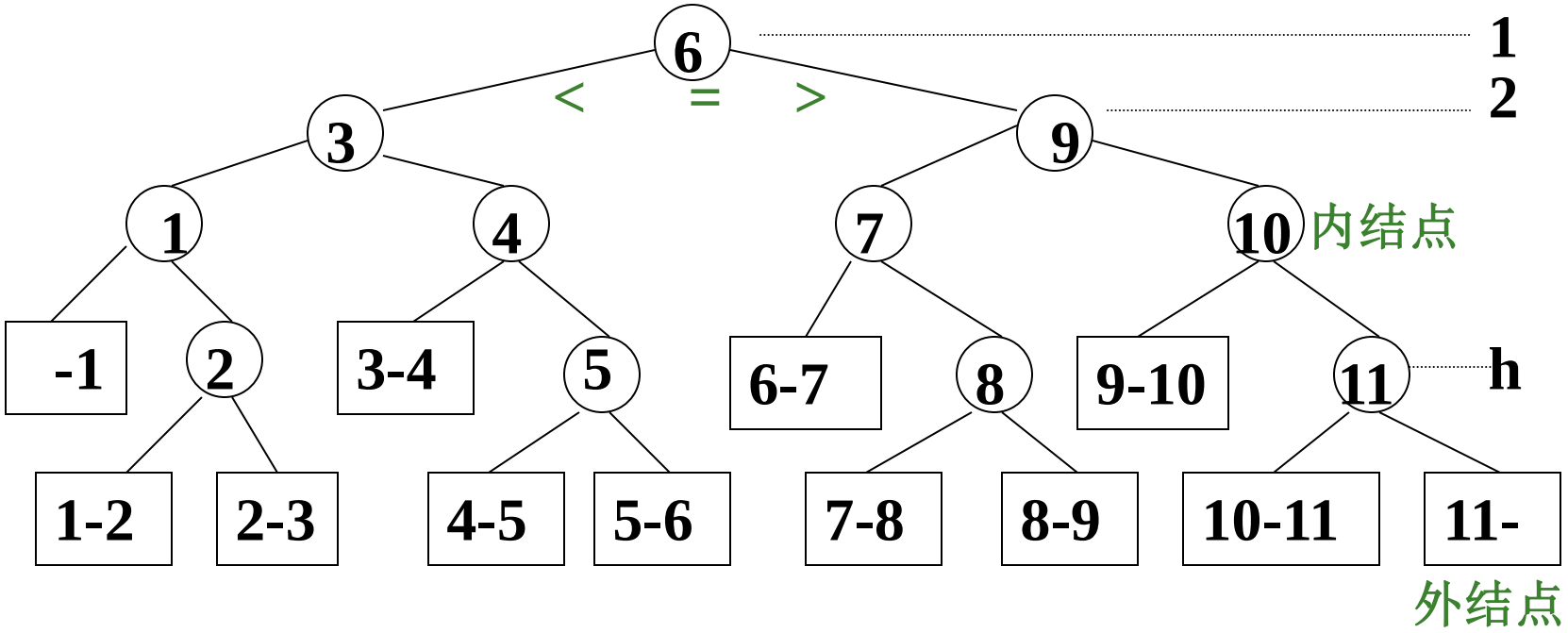


若所有结点的空指针域设置为一个指向一个方形结点的指针，称方形结点为判定树的外部结点；对应的，圆形结点为内部结点。

练习：假定每个元素的查找概率相等，求查找成功时的平均查找长度。



ASL＝1/11*(1*1＋2×2＋4×3+4*4)＝33/11=3



查找成功时比较次数：为该结点在判定树上的层次数，不超过树的深度　$d = \lfloor \log_2 n \rfloor + 1$

查找不成功的过程就是走了一条从根结点到外部结点的路径d或d+1。

折半查找的性能分析

- 查找过程：每次将待查记录所在区间缩小一半，比顺序查找效率高，时间复杂度**O(log2 n)**

- 适用条件：采用顺序存储结构的有序表，不宜用于链式结构

Facebook
- Software Engineer position

Given the numbers 1 to 1000, what is the minimum number of guesses needed to find a specific number, if you are given the hint "higher" or "lower" for each guess you make ??

7.3 树表的查找

表结构在查找过程中动态生成

对于给定值key
若表中存在，则成功返回；
否则插入关键字等于key 的记录

二叉排序树
平衡二叉树
B-树
B^+树
键树

二叉排序树

二叉排序树或是空树，或是满足如下性质的二叉树：

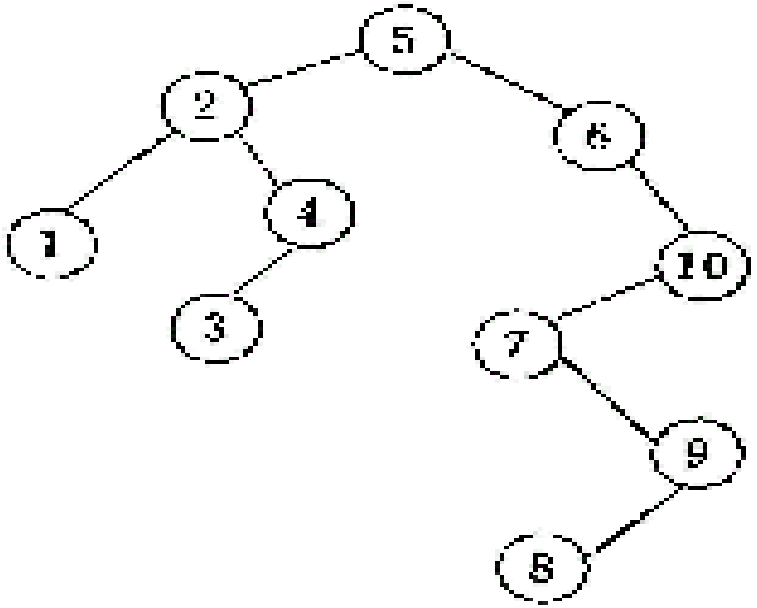
(1)若其左子树非空，则左子树上所有结点的值均小于根结点的值；

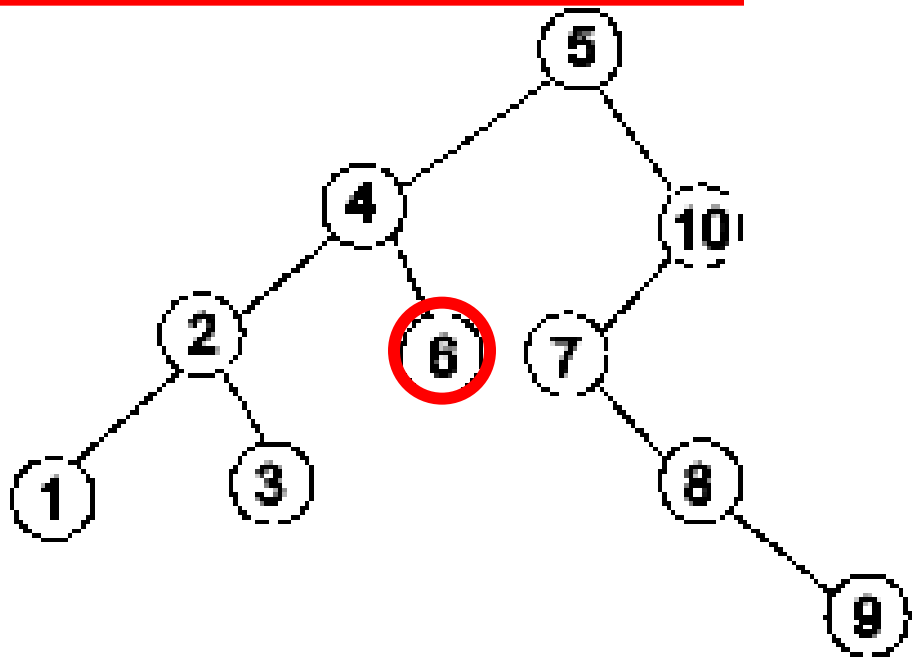
(2)若其右子树非空，则右子树上所有结点的值均大于等于根结点的值；

(3)其左右子树本身又各是一棵二叉排序树

练习

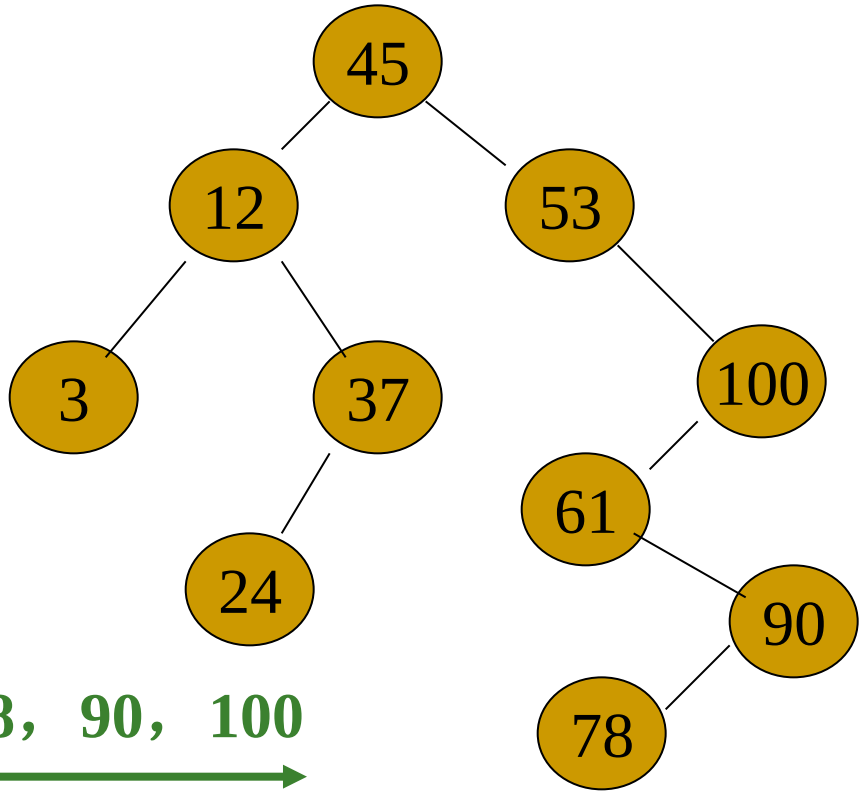
下列图形中，哪个不是二叉排序树 ？





练习

中序遍历二叉排序树后的结果有什么规律？



3，12，24，37，45，53，61，78，90，100

递增

得到一个关键字的递增有序序列

二叉排序树的操作－查找

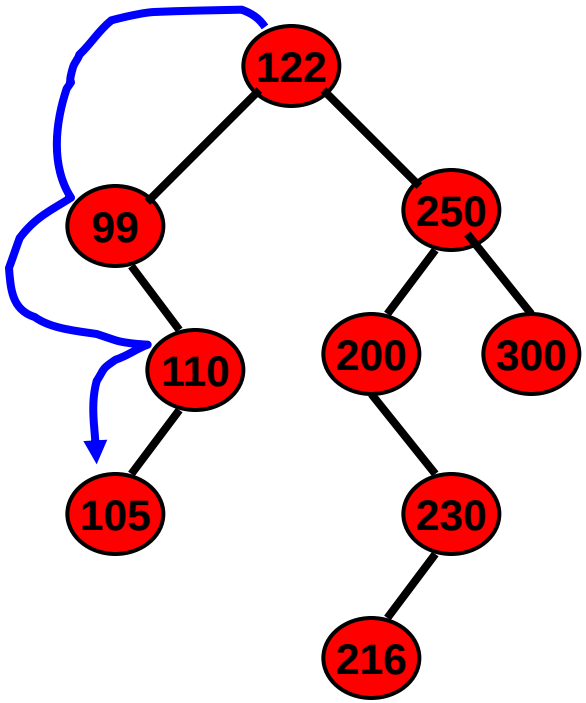
若查找的关键字等于根结点，成功

否则

　若小于根结点，查其左子树

　若大于根结点，查其右子树

在左右子树上的操作类似



【算法思想】

（1）若二叉排序树为空，则查找失败，返回空指针。

（2）若二叉排序树非空，将给定值key与根结点的关键字T->data.key进行比较：

① 若key等于T->data.key，则查找成功，返回根结点地址；

② 若key小于T->data.key，则进一步查找左子树；

③ 若key大于T->data.key，则进一步查找右子树。

【算法描述】

```
BSTree SearchBST(BSTree T,KeyType key) {
   if((!T) || key==T->data.key) return T;
   else if (key<T->data.key)  return SearchBST(T->lchild,key);
          //在左子树中继续查找
   else return SearchBST(T->rchild,key);
          //在右子树中继续查找
} // SearchBST
```

二叉排序树的操作—插入

若二叉排序树为空，则插入结点应为根结点

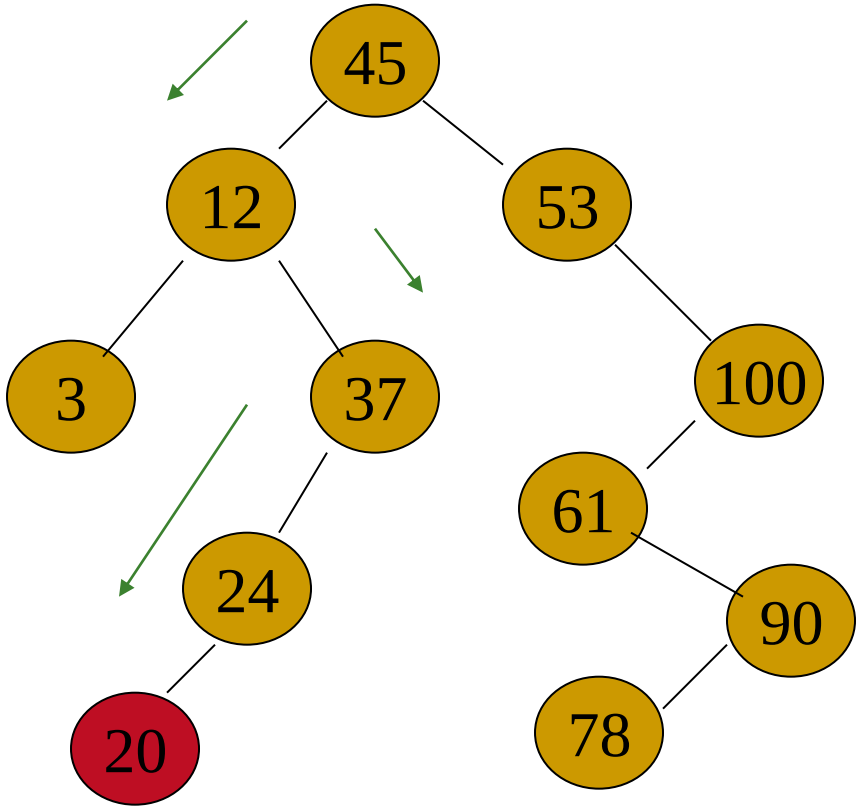
否则，继续在其左、右子树上查找

- ✓树中已有，不再插入
- ✓树中没有，查找直至某个叶子结点的左子树或右子树为空为止，则插入结点应为该叶子结点的左孩子或右孩子

插入的元素一定在叶结点上

二叉排序树的操作—插入

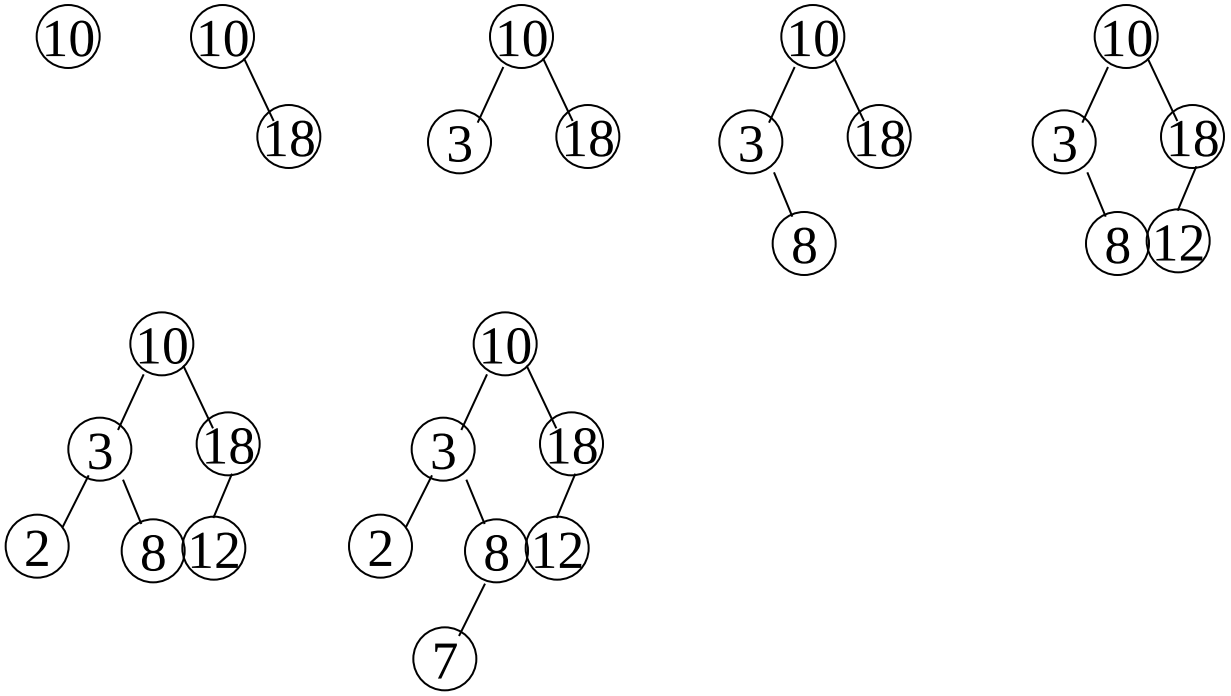
插入结点20



二叉排序树的操作一生成

从空树出发，经过一系列的查找、插入操作之后，可生成一棵二叉排序树

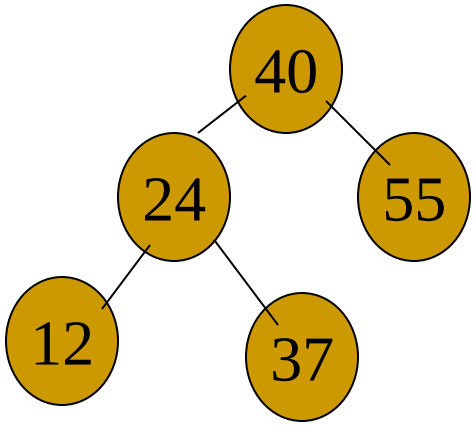
{10，18，3，8，12，2，7}



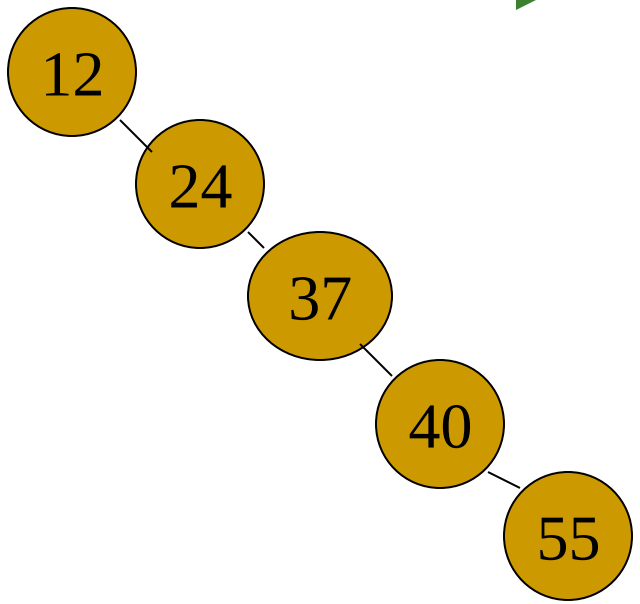
二叉排序树的操作一生成

不同插入次序的序列生成不同形态的二叉排序树

40，24，12，37，55

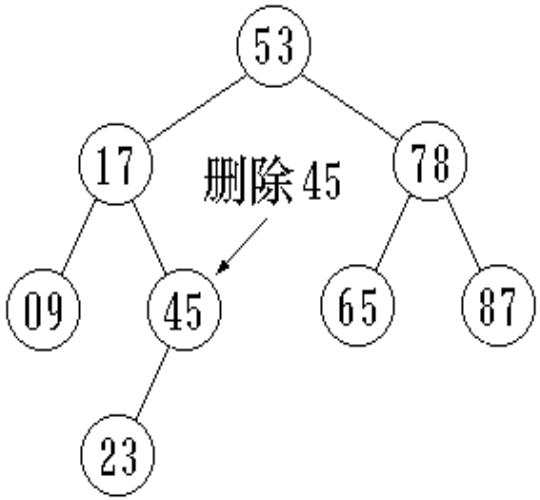


12，24，37，40，55

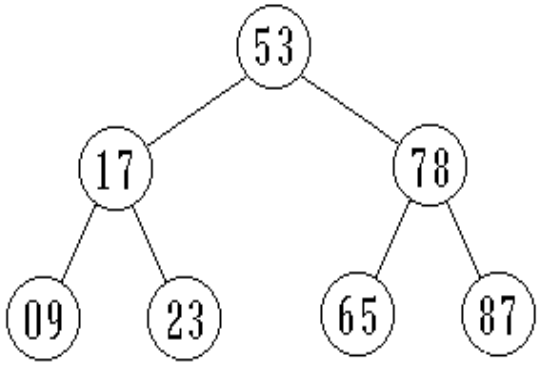


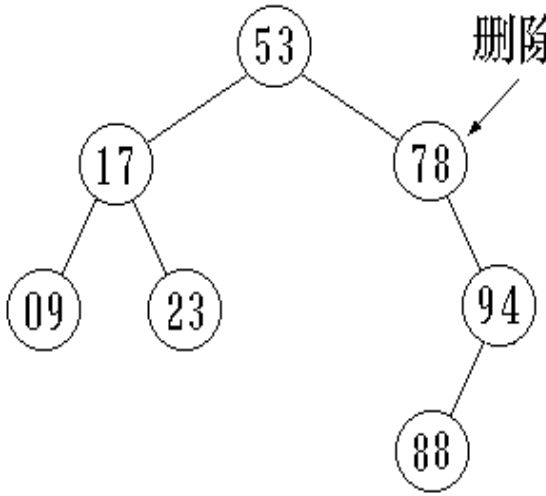
二叉排序树的操作—删除

- 将因删除结点而断开的二叉链表重新链接起来
- 防止重新链接后树的高度增加

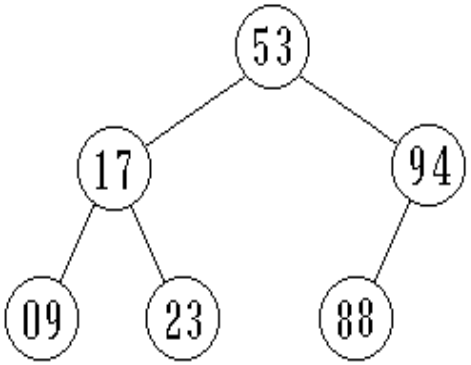


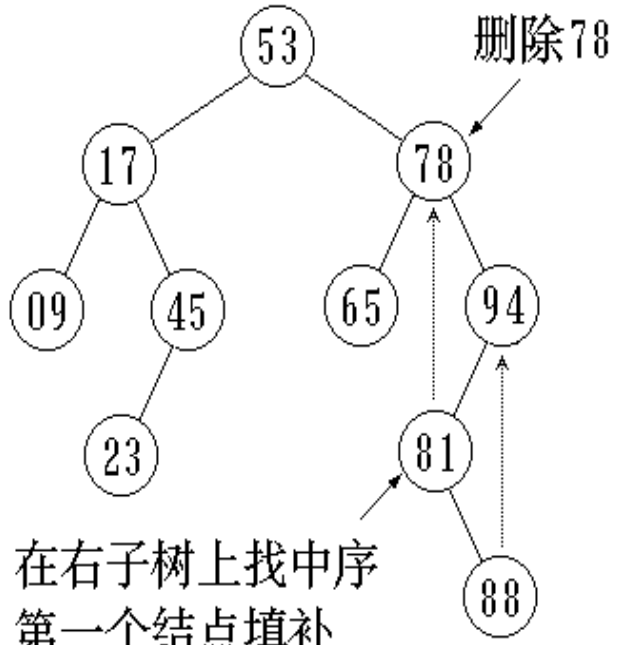
缺右子树用左子女填补



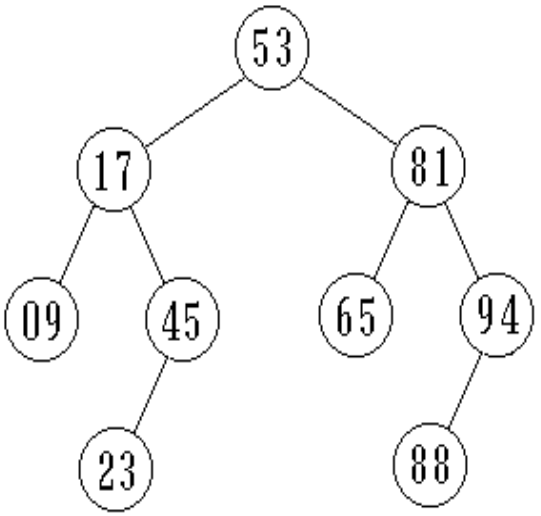


缺左子树用右子女填补





在右子树上找中序
第一个结点填补



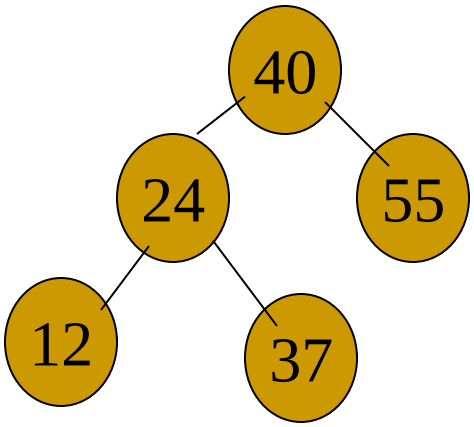
–删除叶结点，只需将其双亲结点指向它的指针清零，再释放它即可。

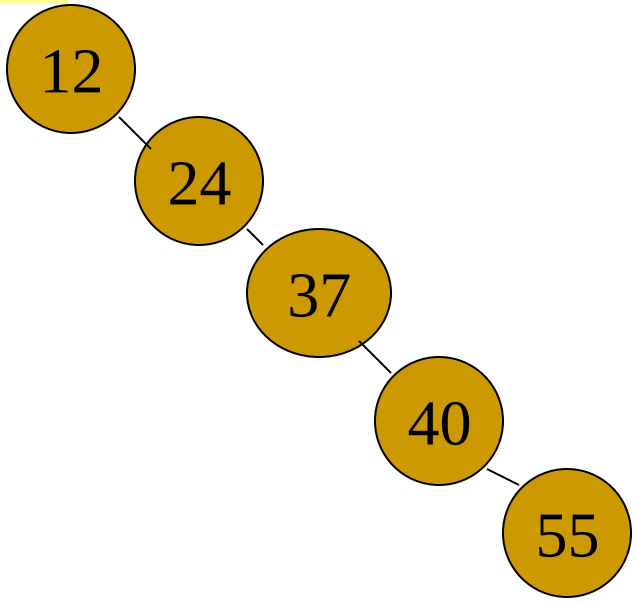
–被删结点缺右子树，可以拿它的左子女结点顶替它的位置，再释放它。

–被删结点缺左子树，可以拿它的右子女结点顶替它的位置，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(关键码最小)，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。

查找的性能分析





第i层结点需比较i次。在等概率的前提下，上述两图的平均查找长度为：

$$\sum_{i=1}^{n} p_i c_i = (1+2\times 2+3\times 2)/5 = 2.2(\text{左图})$$

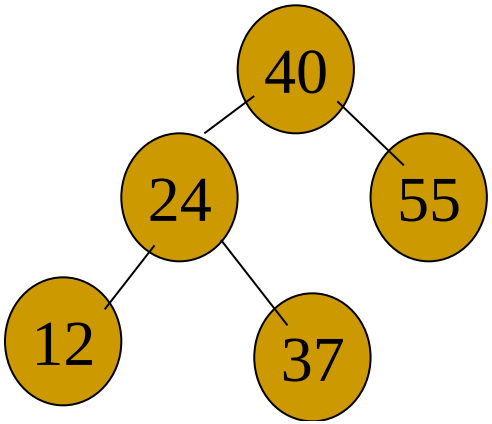
$$\sum_{i=1}^{n} p_i c_i = (1+2+3+4+5)/5 = 3(\text{右图})$$

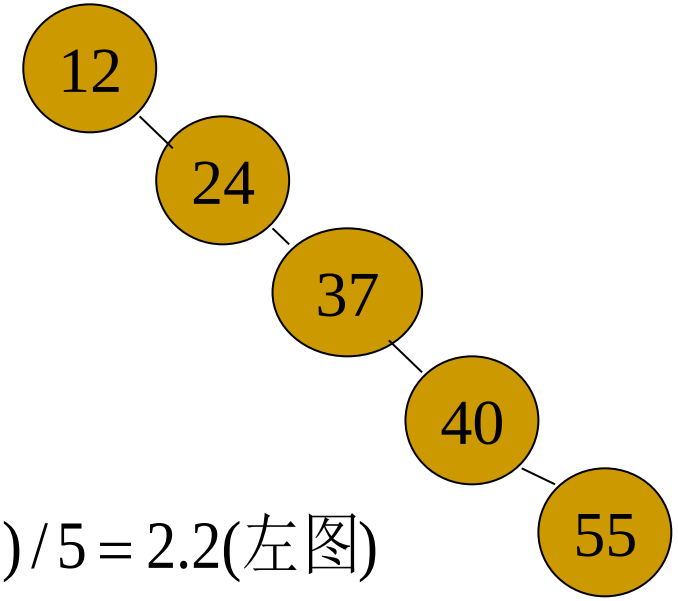
查找的性能分析

平均查找长度和二叉树的形态有关，即，

最好：$\log_2 n$（形态匀称，与二分查找的判定树相似）

最坏：　（n+1)/2（单支树）





$$\sum_{i=1}^{n} p_i c_i = (1+2\times 2+3\times 2)/5 = 2.2(\text{左图})$$

$$\sum_{i=1}^{n} p_i c_i = (1+2+3+4+5)/5 = 3(\text{右图})$$

问题：如何提高二叉排序树的查找效率？

尽量让二叉树的形状均衡

平衡二叉树

- 左、右子树是平衡二叉树；
- 所有结点的左、右子树深度之差的绝对值≤ 1

平衡因子:该结点左子树与右子树的高度差

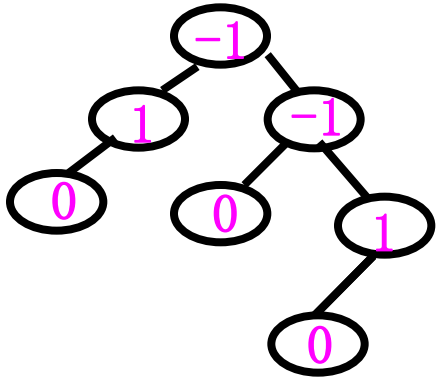
1989奥斯卡是最佳短片奖--《Balance》

世界需要平衡，破坏平衡的一方，也许会一时很强势的称霸，最终的结局逃不过孤立和落空

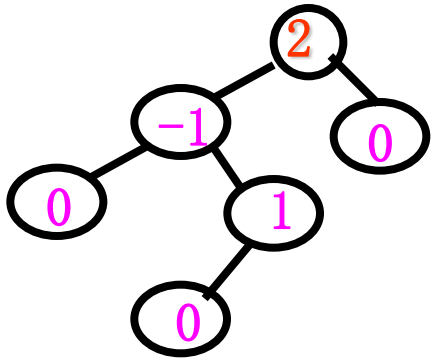
- 任一结点的平衡因子只能取：-1、0 或 1；如果树中任意一个结点的平衡因子的绝对值大于1，则这棵二叉树就失去平衡，不再是*AVL*树；

- 对于一棵有*n*个结点的*AVL*树，其高度保持在$O(\log_2 n)$数量级，*ASL*也保持在$O(\log_2 n)$量级。

练习：判断下列二叉树是否AVL树？



（a）平衡树



（b）不平衡树

如果在一棵AVL树中插入一个新结点，就有可能造成失衡，此时必须重新调整树的结构，使之恢复平衡。我们称调整平衡过程为平衡旋转。

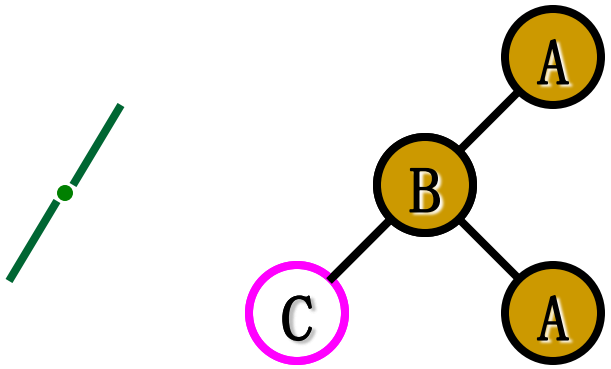
- ✓LL平衡旋转
- ✓RR平衡旋转
- ✓LR平衡旋转
- ✓RL平衡旋转

保证二叉排序树的次序不变

1）LL平衡旋转：

若在A的左子树的左子树上插入结点，使A的平衡因子从1增加至2，需要进行一次顺时针旋转。

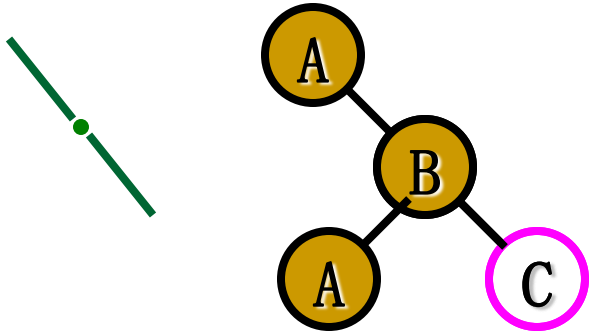
（以B为旋转轴）



2）RR平衡旋转：

若在A的右子树的右子树上插入结点，使A的平衡因子从-1增加至-2，需要进行一次逆时针旋转。

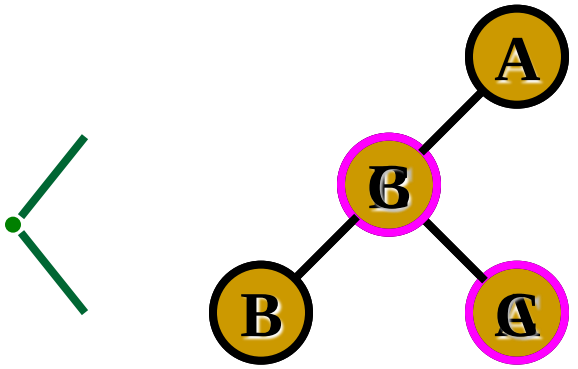
（以B为旋转轴）



3）LR平衡旋转：

若在A的左子树的右子树上插入结点，使A的平衡因子从1增加至2，需要先进行逆时针旋转，再顺时针旋转。

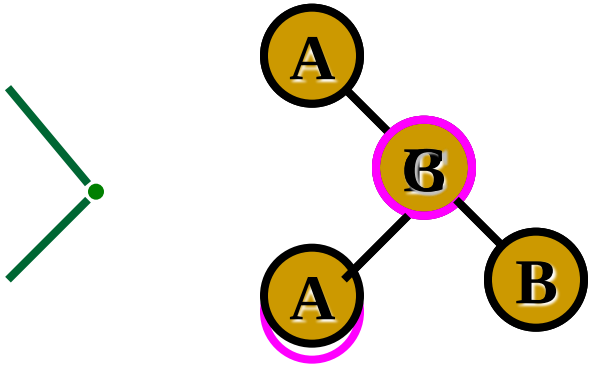
（以插入的结点C为旋转轴）



4）RL平衡旋转：

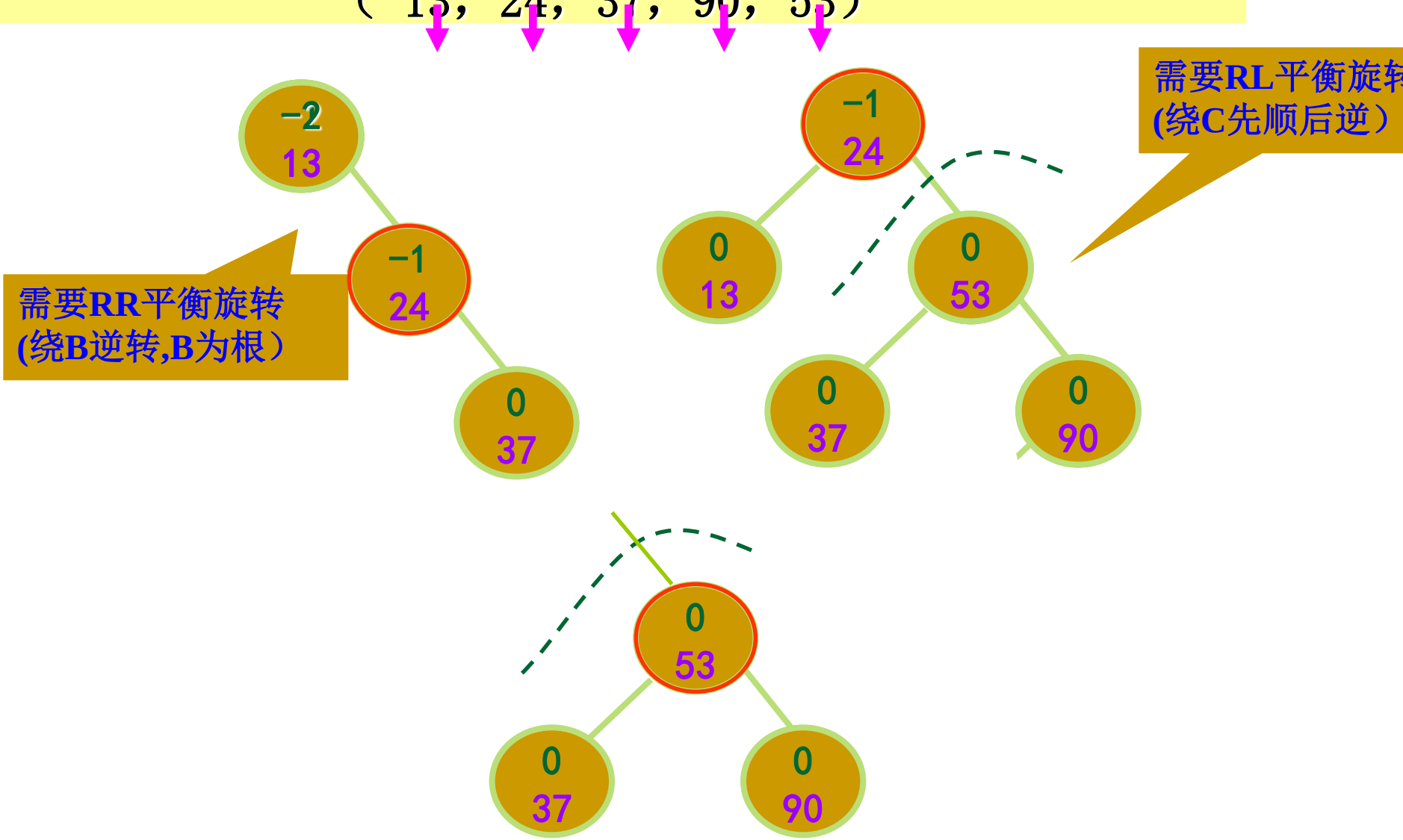
若在A的右子树的左子树上插入结点，使A的平衡因子从-1增加至-2，需要先进行顺时针旋转，再逆时针旋转。

（以插入的结点C为旋转轴）



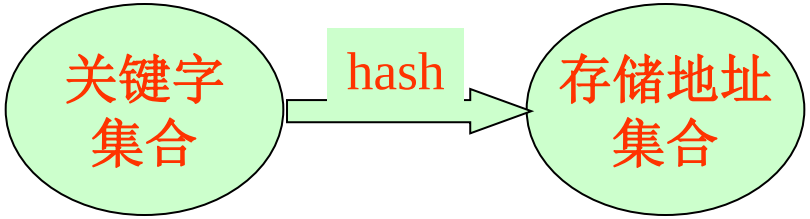
练习：请将下面序列构成一棵平衡二叉排序树

（ 13，24，37，90，53）



7.4 哈希表的查找

- 基本思想：记录的存储位置与关键字之间存在对应关系，Loc(i)=H(keyi) → 哈希函数



- 优点：查找速度极快O(1),查找效率与元素个数n无关

例1

若将学生信息按如下方式存入计算机，如：
将20010118102**01**的所有信息存入V[01]单元；
将20010118102**02**的所有信息存入V[02]单元；
……
将20010118102**31**的所有信息存入V[31]单元。

查找20010118102**16**的信息，可直接访问V[16]！

例2

数据元素序列(14，23，39，9，25，11)，若规定每个元素k的存储地址H（k）＝k，请画出存储结构图。

地址	…	9	…	11	…	14	…	23	24	25	…	39	…
内容		9		11		14		23		25		39	

如何查找

地址	…	9	…	11	…	14	…	23	24	25	…	39	…
内容		9		11		14		23		25		39	

根据哈希函数**H（k）＝k**

查找key=9,则访问H(9)=9号地址，若内容为9则成功；

若查不到，则返回一个特殊值，如空指针或空记录。

有关术语

哈希（hash)方法(杂凑法)

选取某个函数，依该函数按关键字计算元素的存储位置，并按此存放；

查找时，由同一个函数对给定值k计算地址，将k与地址单元中元素关键码进行比，确定查找是否成功。

哈希函数(杂凑函数)： 哈希方法中使用的转换函数

有关术语

哈希表(杂凑表)： 按上述思想构造的表

地址	…	9	…	11	…	14	…	23	24	25	…	39	…
内容		9		11		14		23		25		39	

冲　突： 不同的关键码映射到同一个哈希地址

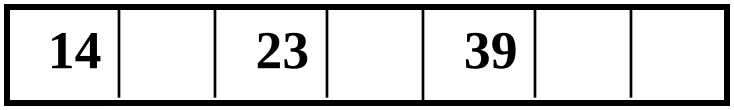
key1≠key2，但H(key1)=H(key2)

同义词： 具有相同函数值的两个关键字

冲突现象举例

（14，23，39，9，25，11）

哈希函数：H(k)=k mod 7



9

25

11

6个元素用7个地址应该足够!

H(14)=14%7=0

H(25)=25%7=4

H(11)=11%7=4

同义词

有冲突

如何减少冲突

冲突是不可能避免的

构造好的哈希函数

制定一个好的解决冲突方案

哈希函数的构造方法

根据元素集合的特性构造

地址空间尽量小

均匀

1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
6. 随机数法

直接定址法

Hash(key) = a ⁎key + b　　（a、b为常数）

优点：以关键码key的某个线性函数值为哈希地址，不会产生冲突。

缺点：要占用连续地址空间，空间效率低。

直接定址法

例： {100，300，500，700，800，900}，
哈希函数Hash(key)=key/100

0	1	2	3	4	5	6	7	8	9
	100		300		500		700	800	900

除留余数法 （最常用重点掌握）

Hash(key)=key mod p (p是一个整数)

关键：如何选取合适的**p**？

技巧：设表长为**m**，取**p≤m**且为质数

除留余数法 （最常用重点掌握）

- 设关键字集={ 15, 45, 18, 39, 24, 33, 21 }，哈希表表长m=20。

则若取p=9(含质因子3)，则哈希地址为{ 6, 0, 0, 3, 6, 6, 3 }，冲突现象严重；

若取p=11(质数)，则哈希地址为{ 4, 1, 7, 6, 2, 0, 10 }，没有产生冲突现象。

构造哈希函数考虑的因素

① 执行速度（即计算哈希函数所需时间）；
② 关键字的长度；
③ 哈希表的大小；
④ 关键字的分布情况；
⑤ 查找频率。

处理冲突的方法

1.开放定址法

2.链地址法

1. 开放定址法（开地址法）

基本思想： 有冲突时就去寻找下一个空的哈希地址，只要哈希表足够大，空的哈希地址总能找到，并将数据元素存入。

- 线性探测法
- 二次探测法
- 伪随机探测法

线性探测法

$H_i=(Hash(key)+d_i) \bmod m$　　$(1 \le i < m)$

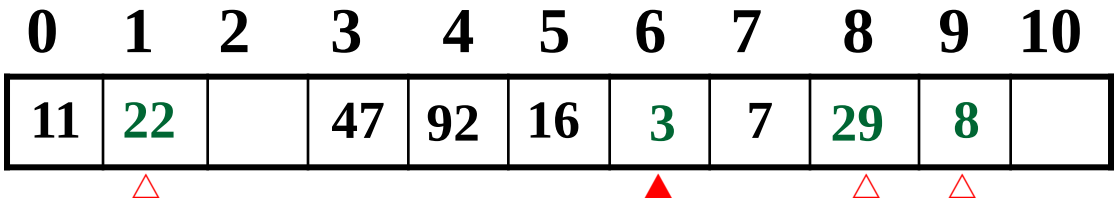
其中：m为哈希表长度

d_i 为增量序列 1，2　　m-1，且$d_i=i$

一旦冲突，就找下一个空地址存入

线性探测法

关键码集为 {47，7，29，11，16，92，22，8，3}，
设：哈希表表长为m=11；
哈希函数为Hash(key)=key mod 11



① 47、7、11、16、92没有冲突

② Hash(29)=7，有冲突，由H_1=(Hash(29)+1) mod 11=8，哈希地址8为空，因此将29存入

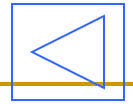
③ 3 连续移动了两次

线性探测法的特点

优点：只要哈希表未被填满，保证能找到一个空地址单元存放有冲突的元素。

缺点：可能使第i个哈希地址的同义词存入第i+1个地址，这样本应存入第i+1个哈希地址的元素变成了第i+2个哈希地址的同义词，……，产生“聚集”现象，降低查找效率。

解决方案：二次探测法



二次探测法

关键码集为 {47，7，29，11，16，92，22，8，3}，
设：　　哈希函数为Hash(key)=key mod 11

$H_i=(Hash(key)\pm d_i)\ mod\ m$
其中：m为哈希表长度；
　　d_i为增量序列 1^2，-1^2，2^2，-2^2，…，q^2

0	1	2	3	4	5	6	7	8	9	10
11	22	3	47	92	16		7	29	8	
	△	▲						△	△	

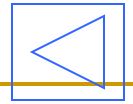
Hash(3)=3，哈希地址冲突，由
$H_1=(Hash(3)+1^2)\ mod\ 11=4$，仍然冲突；
$H_2=(Hash(3)-1^2)\ mod\ 11=2$，找到空的哈希地址，存入。

伪随机探测法

$H_i=(Hash(key)+d_i) \bmod m$　　$(1 \le i < m)$

其中：m为哈希表长度

d_i 为随机数

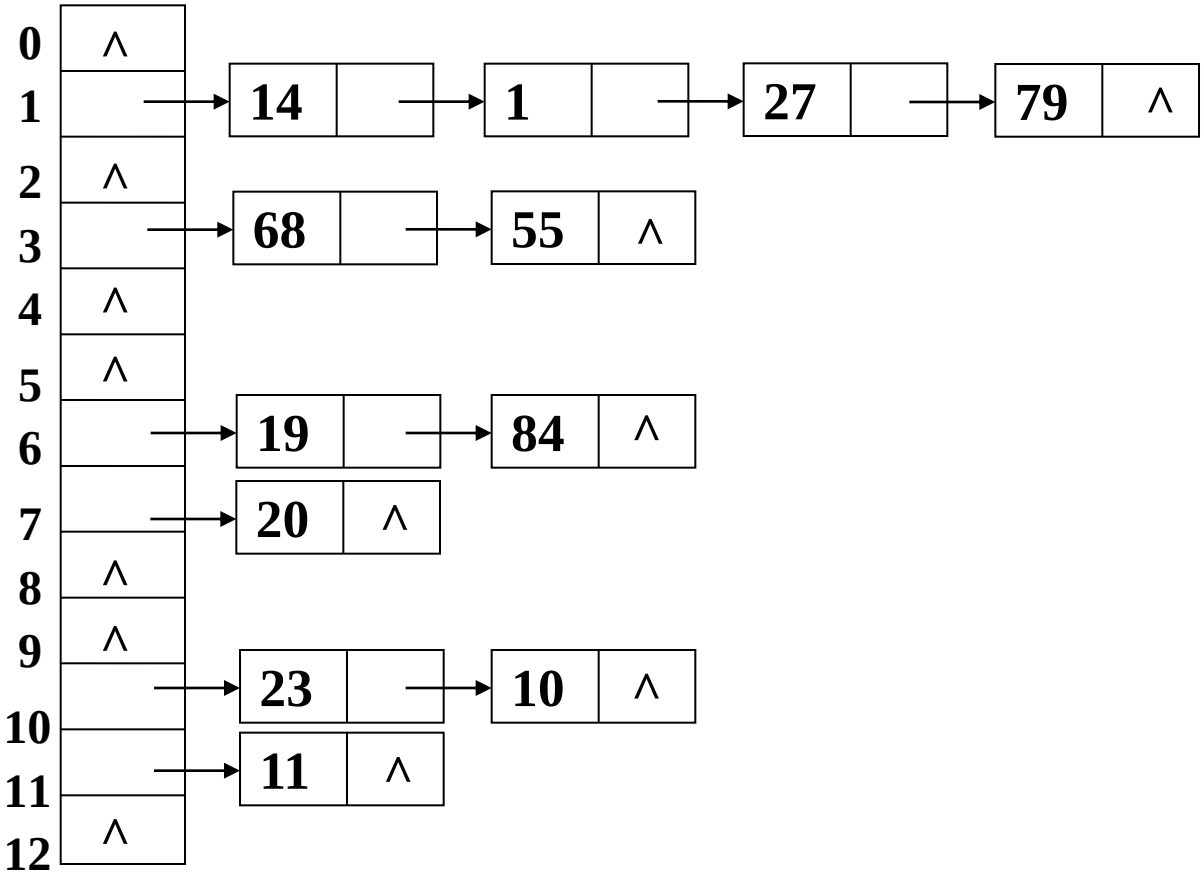


开放地址法建立哈希表步骤

- step1 取数据元素的关键字key，计算其哈希函数值（地址）。若该地址对应的存储空间还没有被占用，则将该元素存入；否则执行step2解决冲突。
- step2 根据选择的冲突处理方法，计算关键字key的下一个存储地址。若下一个存储地址仍被占用，则继续执行step2，直到找到能用的存储地址为止。

2. 链地址法(拉链法)

基本思想：相同哈希地址的记录链成一单链表，m个哈希地址就设m个单链表，然后用用一个数组将m个单链表的表头指针存储起来，形成一个动态的结构



链地址法建立哈希表步骤

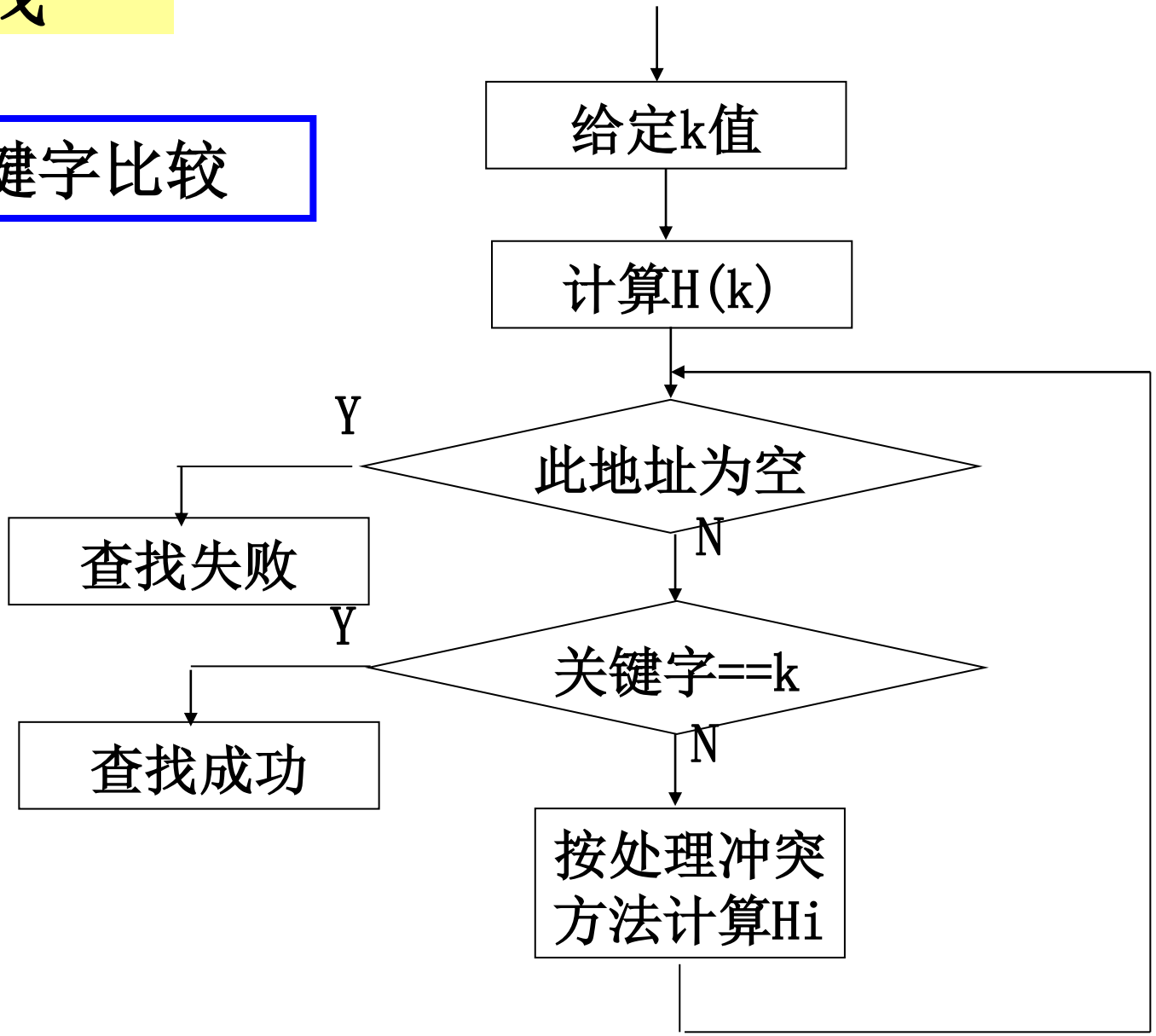
- step1 取数据元素的关键字key，计算其哈希函数值（地址）。若该地址对应的链表为空，则将该元素插入此链表；否则执行step2解决冲突。
- step2 根据选择的冲突处理方法，计算关键字key的下一个存储地址。若该地址对应的链表为不为空，则利用链表的前插法或后插法将该元素插入此链表。

链地址法的优点：

- 非同义词不会冲突，无“聚集”现象
- 链表上结点空间动态申请，更适合于表长不确定的情况

哈希表的查找

给定值与关键字比较



哈希表的查找

ASL=(1*6+2+3*3+4+9)/12=2.5

已知一组关键字(19,14,23,1,68,20,84,27,55,11,10,79)
哈希函数为：H(key)=key MOD 13，哈希表长为m=16，
设每个记录的查找概率相等

(1) 用线性探测再散列处理冲突，即Hi=(H(key)+di) MOD m

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	1	68	27	55	19	20	84	79	23	11	10			
	1	2	1	4	3	1	1	3	9	1	1	3			

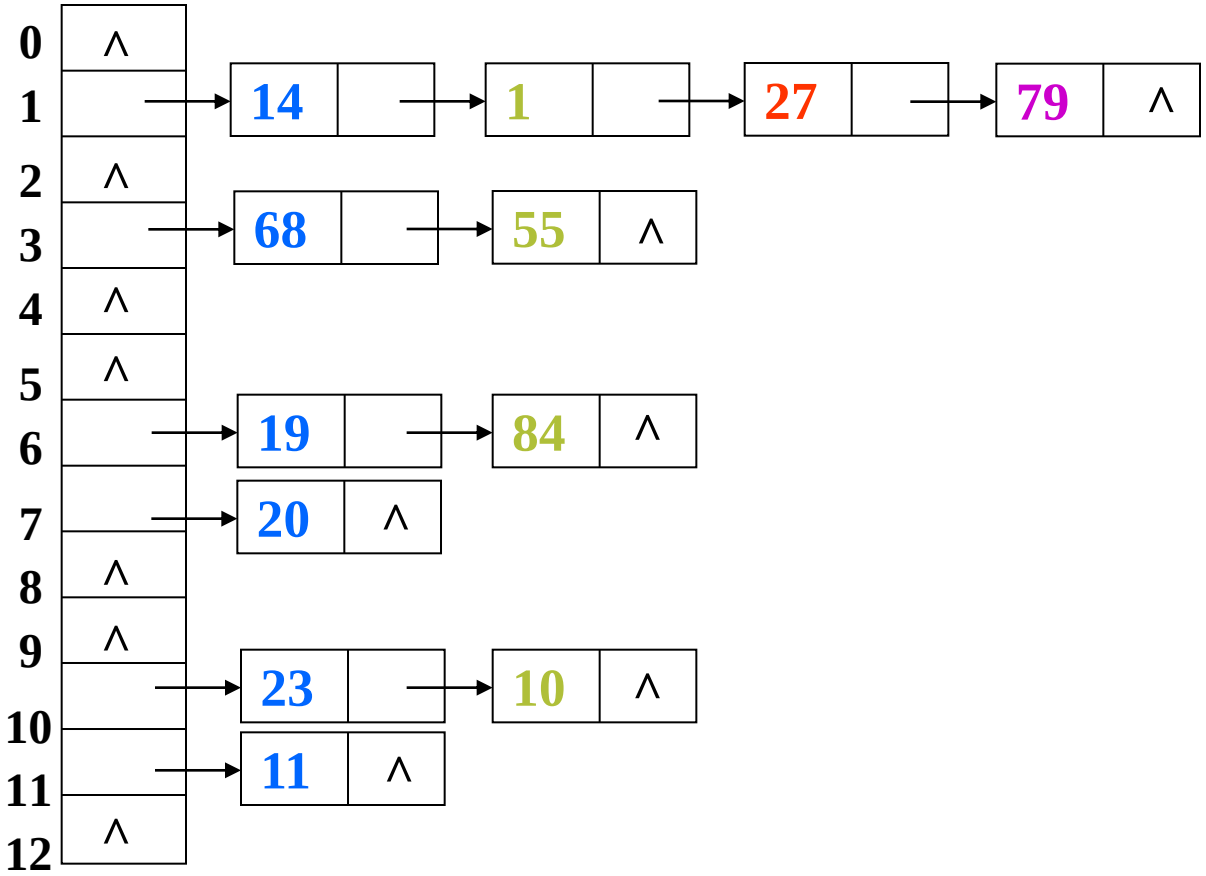
H(19)=6
H(14)=1
H(23)=10
H(1)=1 冲突，H1=(1+1) MOD16=2
H(68)=3
H(20)=7
H(27)=1 冲突，H1=(1+1)MOD16=2
冲突，H2=(1+2)MOD16=3
冲突，H3=(1+3)MOD16=4

哈希表的查找

ASL=(1*6+2*4+3+4)/12=1.75

(2) 用链地址法处理冲突

关键字(19,14,23,1,68,20,84,27,55,11,10,79)



思考

关键字(19,14,23,1,68,20,84,27,55,11,10,79)

无序表查找ASL?

有序表折半查找ASL?

哈希表的查找效率分析

O(1)?

使用平均查找长度ASL来衡量查找算法，ASL取决于

- ✓ 哈希函数
- ✓ 处理冲突的方法
- ✓ 哈希表的装填因子

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$$

α 越大，表中记录数越多，说明表装得越满，发生冲突的可能性就越大，查找时比较次数就越多。

哈希表的查找效率分析

ASL与装填因子α有关！既不是严格的O(1)，也不是O(n)

$$ASL \approx 1 + \frac{\alpha}{2}$$ （拉链法）

$$ASL \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$ （线性探测法）

$$ASL \approx -\frac{1}{\alpha}\ln(1-\alpha)$$ （随机探测法）

几点结论

- 对哈希表技术具有很好的平均性能，优于一些传统的技术
- 链地址法优于开放地址法
- 除留余数法作哈希函数优于其它类型函数

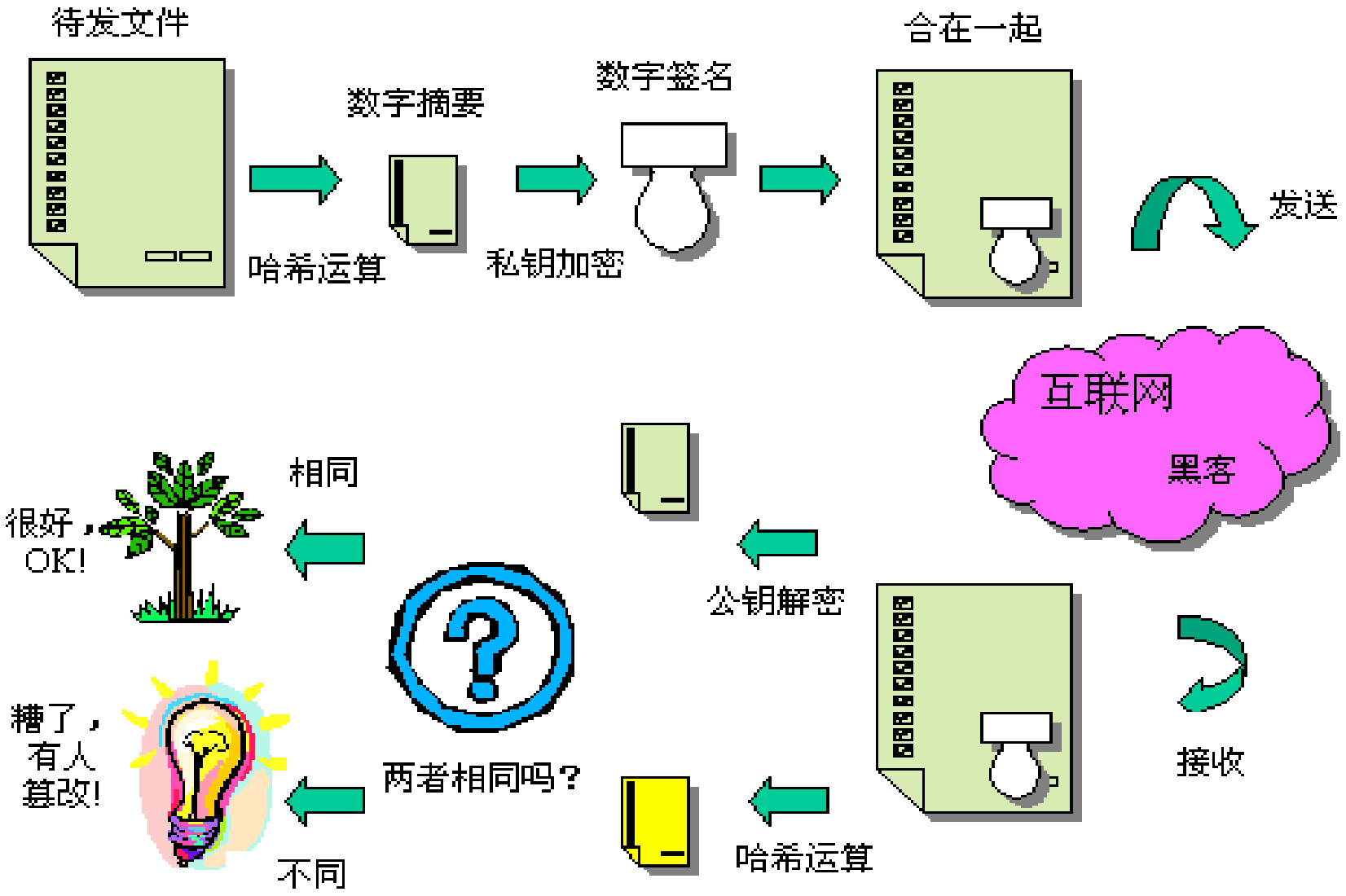
哈希表应用举例

编译器对标识符的管理多是采用哈希表

构造Hash函数的方法：

- 将标识符中的每个字符转换为一个非负整数
- 将得到的各个整数组合成一个整数（可以将第一个、中间的和最后一个字符值加在一起，也可以将所有字符的值加起来）
- 将结果数调整到0~M-1范围内，可以利用取模的方法，Ki%M（M为素数）

哈希函数在信息安全领域中的应用



小结

1. 熟练掌握顺序表和有序表（折半查找）的查找算法及其性能分析方法；

2.熟练掌握二叉排序树的构造和查找算法及其性能分析方法；

3.熟练掌握二叉排序树的插入算法，掌握删除方法；

4.掌握平衡二叉树的定义

5.熟练掌握哈希函数（除留余数法）的构造

6.熟练掌握哈希函数解决冲突的方法及其特点

- 开放地址法（线性探测法、二次探测法）
- 链地址法
- 给定实例计算平均查找长度ASL，ASL依赖于装填因子α