算法与数据结构

主讲教师：陈娜
联系方式：chenna@stdu.edu.cn


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

第7章　查找

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，了解二叉排序树的删除算法；

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：**
  对查找表没有修改操作
- **动态查找表：**
  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*)

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

一、顺序查找（线性查找）

二、折半查找（二分或对分查找）

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;
```

回顾顺序表的查找过程：

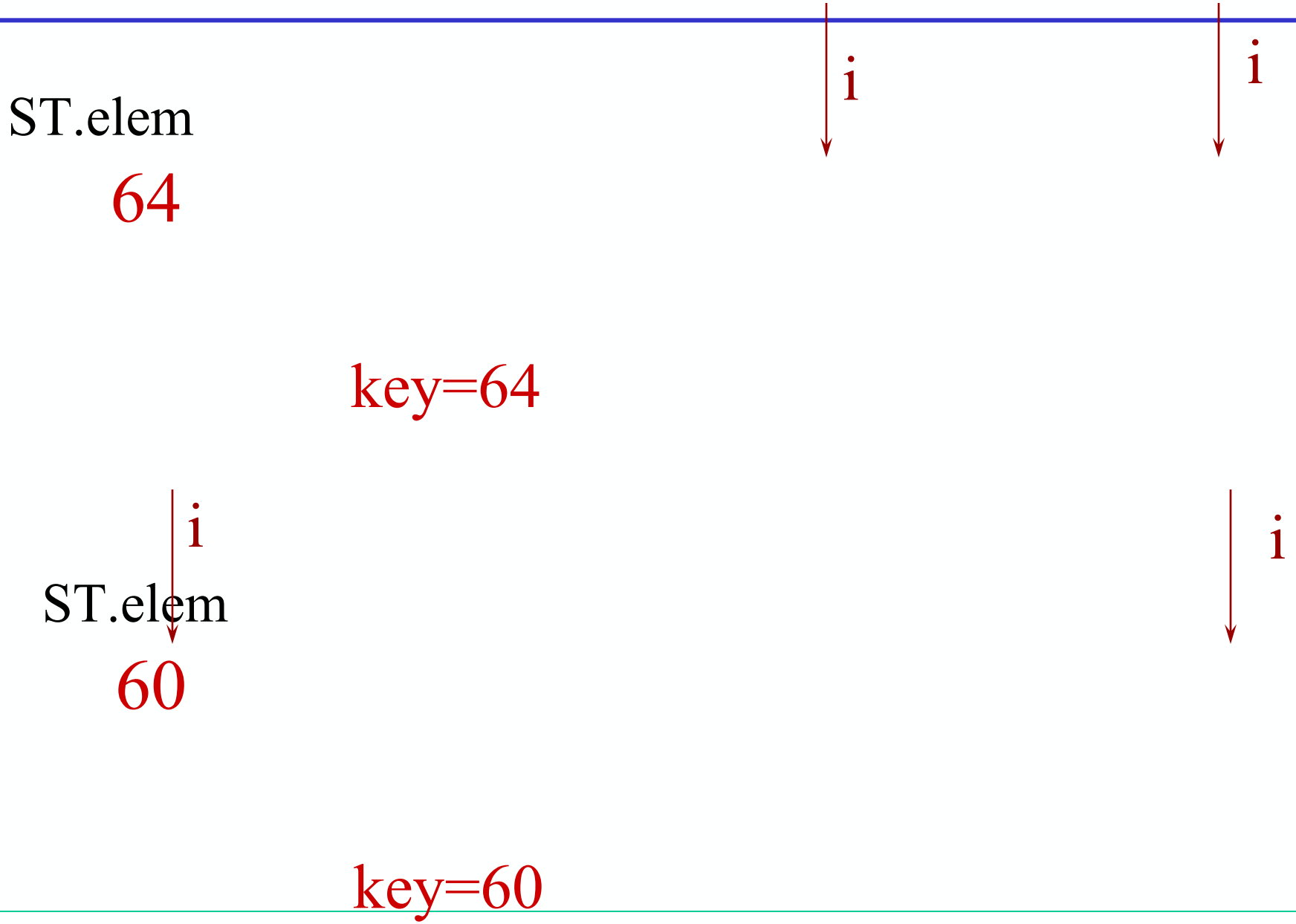
假设给定值 e=64,

要求 ST.elem[k] = e, 问: k = ?

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。


i
i
ST.elem
64
key=64
i
i
ST.elem
60
key=60

```
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  );

  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。

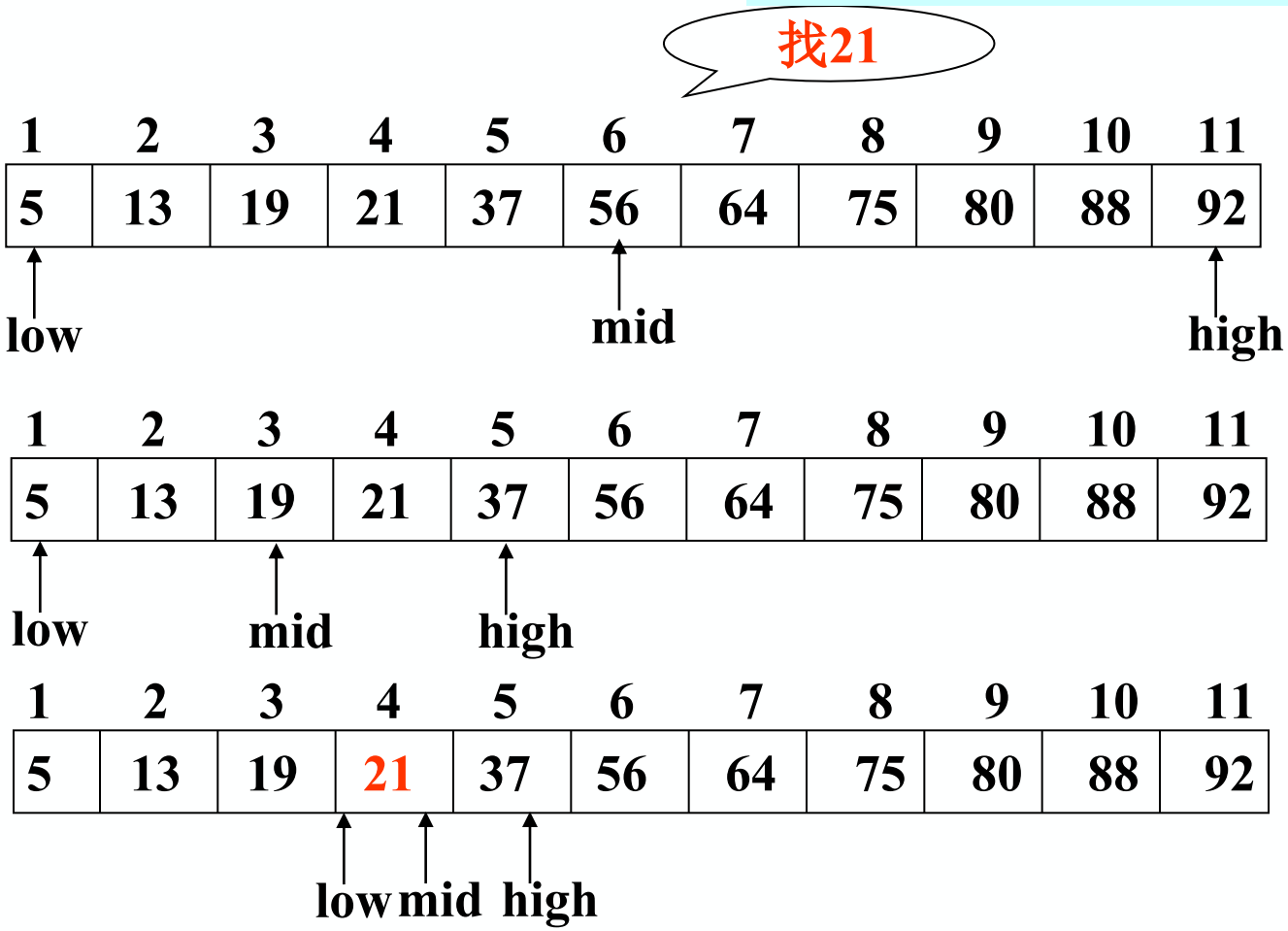
查找概率相等时，ASL相同；
查找概率不等时，如果从前向后查找，则按查找概率由大到小排列的有序表其ASL要比无序表ASL小。

折半查找

若k==R[mid].key，查找成功

若k<R[mid].key，则high=mid-1

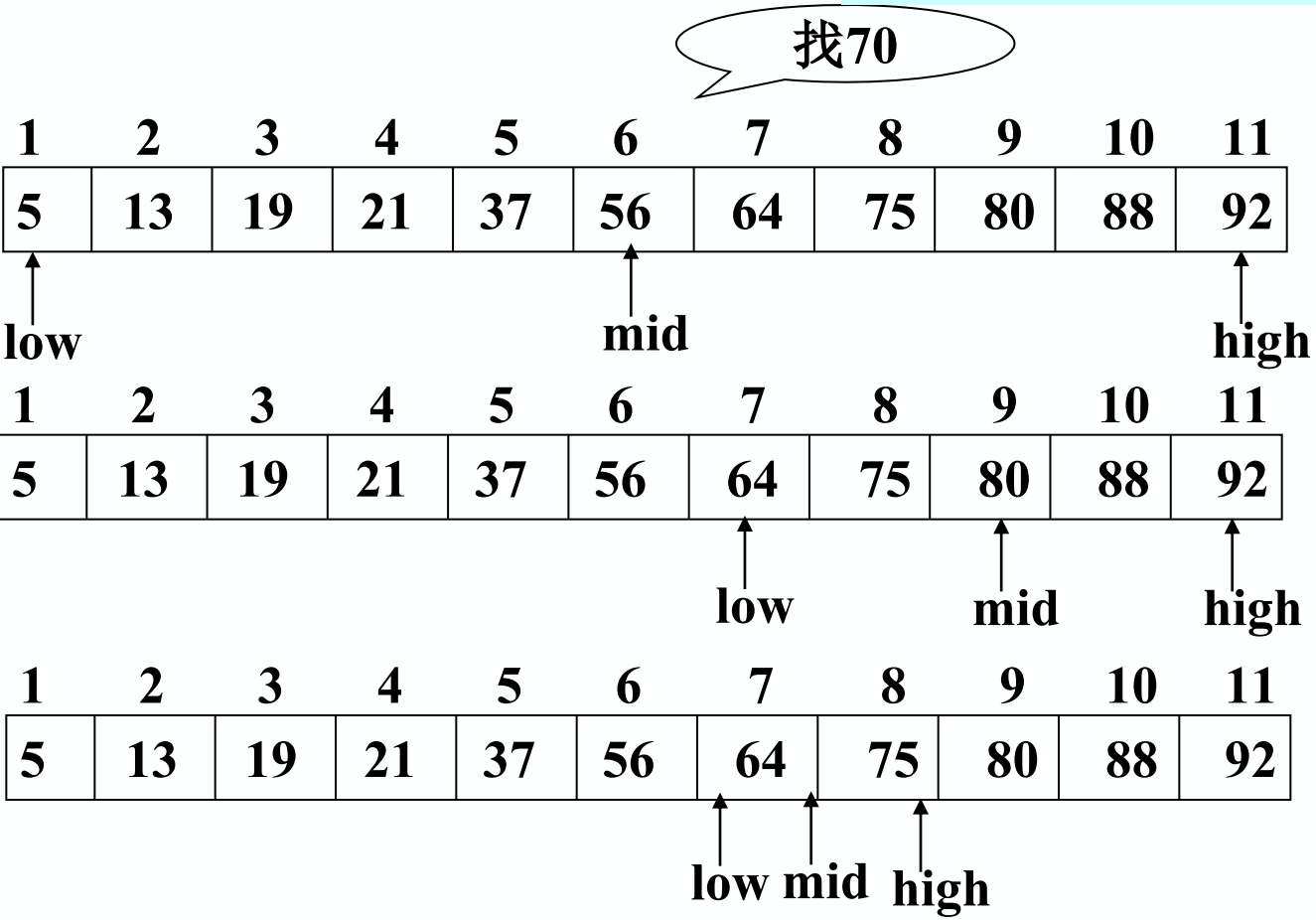
若k>R[mid].key，则low=mid+1



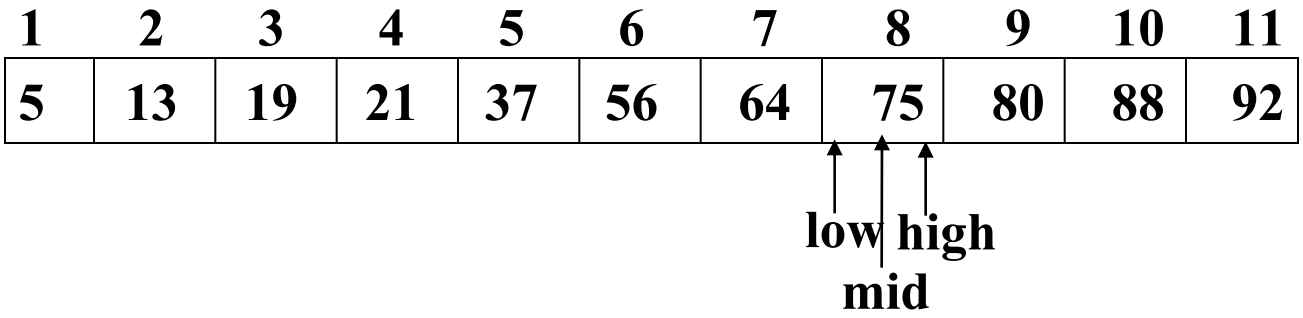
若k==R[mid].key，查找成功

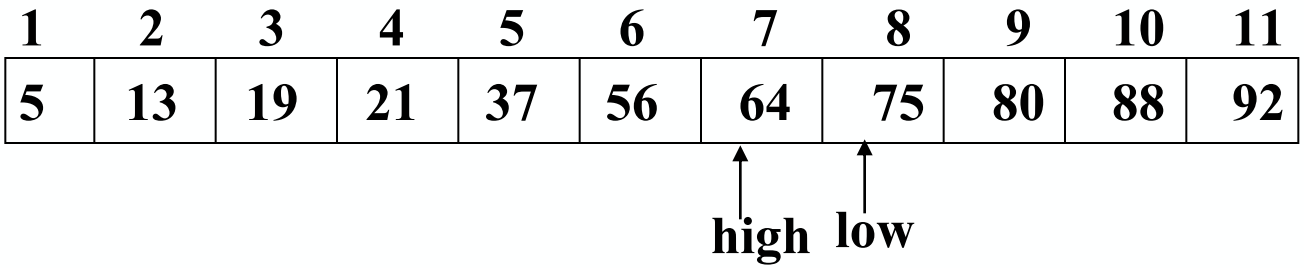
若k<R[mid].key，则high=mid-1

若k>R[mid].key，则low=mid+1



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

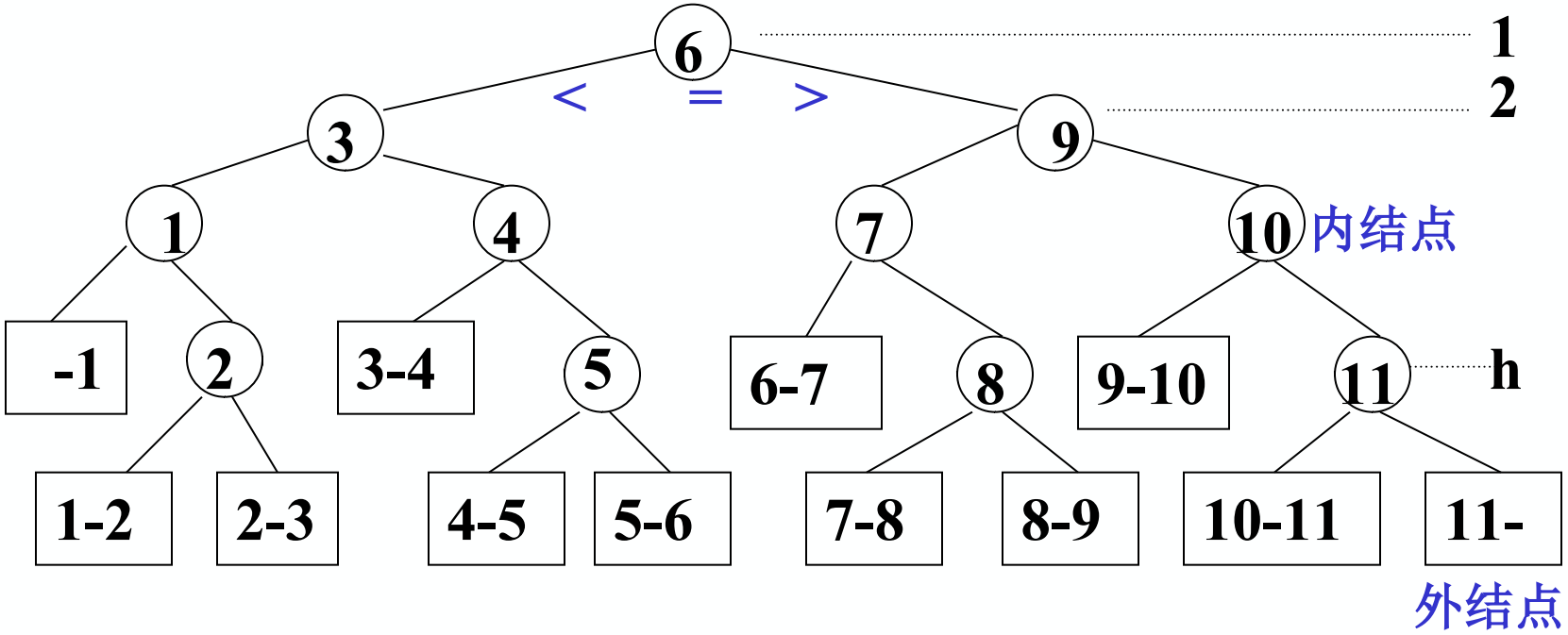
```
int Search_Bin ( SSTable ST, KeyType key ) {
  low = 1;  high = ST.length;     // 置区间初值
  while (low <= high) {
    mid = (low + high) / 2;
    if （key==ST.elem[mid].key)
      return  mid;          // 找到待查元素
    else  if (key<ST.elem[mid].key)
      high = mid - 1;       // 继续在前半区间进行查找
    else  low = mid + 1; // 继续在后半区间进行查找
  }
  return 0;                 // 顺序表中不存在待查元素
} // Search_Bin
```

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
}
```

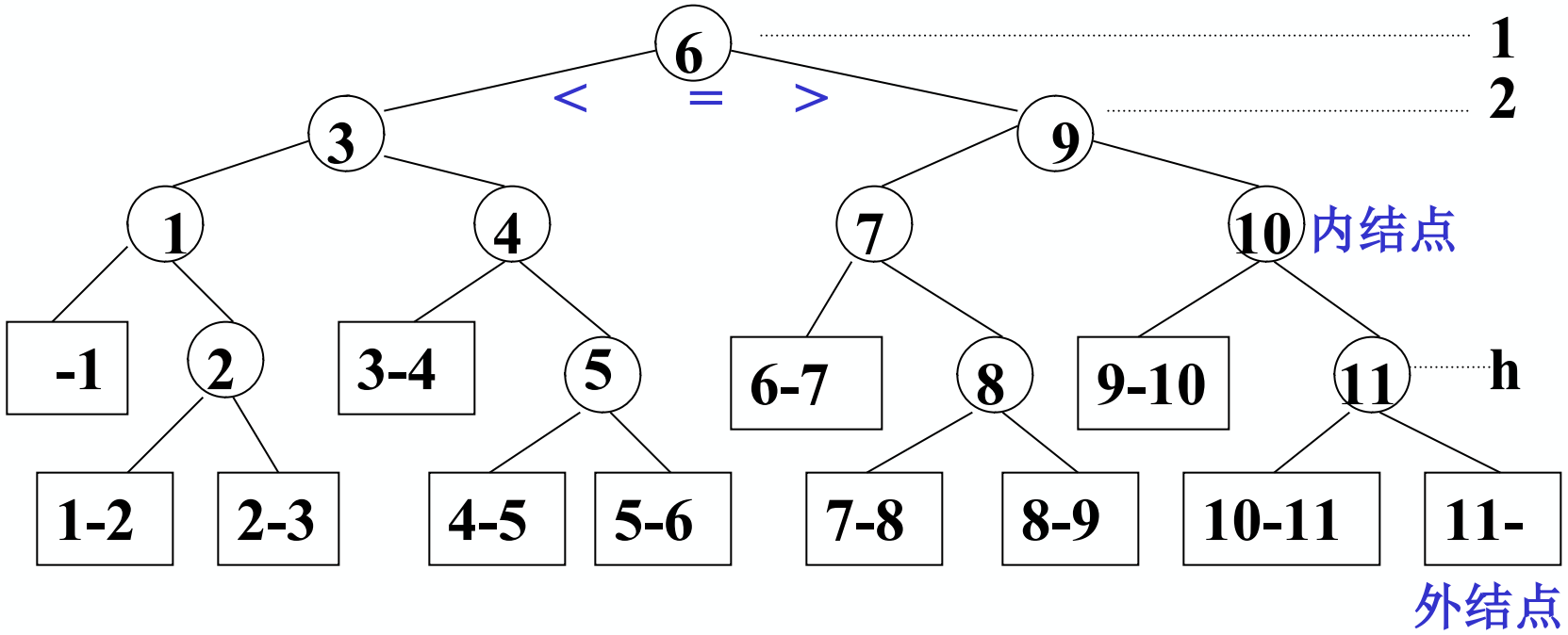
折半查找的性能分析—判定树

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

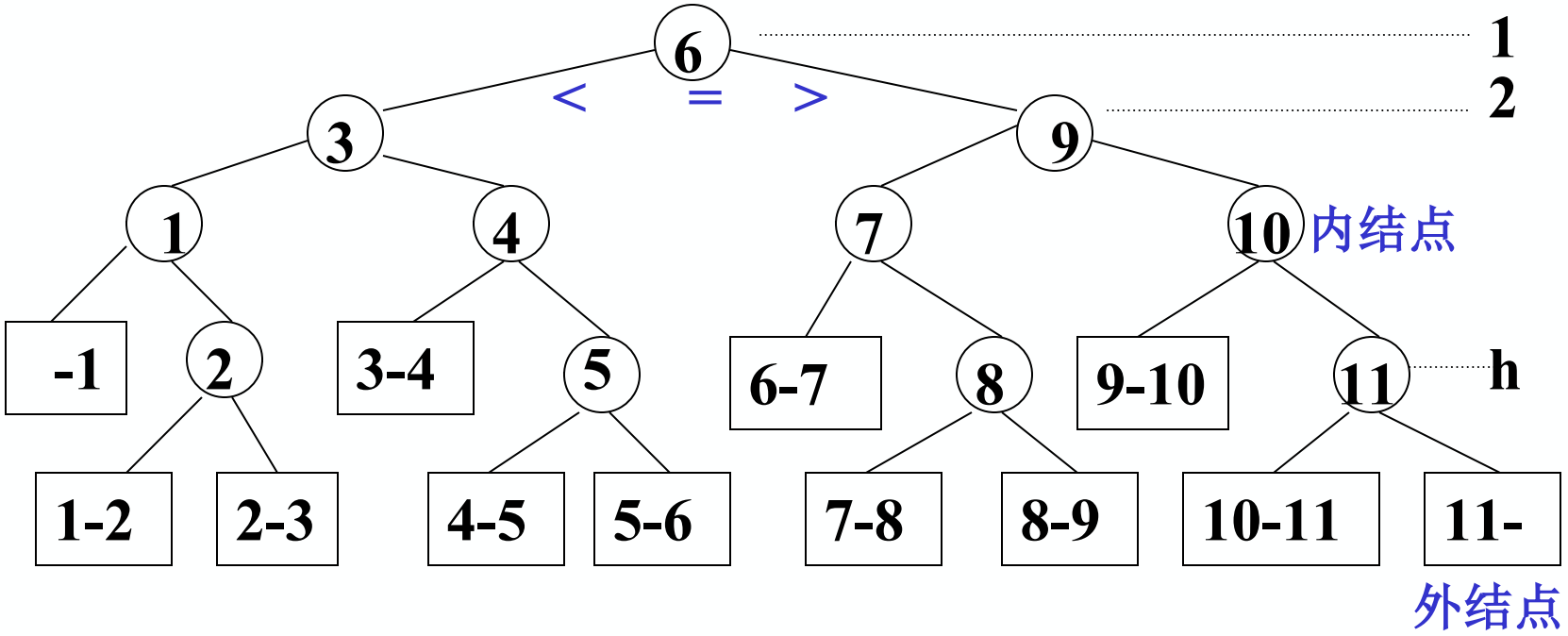


若所有结点的空指针域设置为一个指向一个方形结点的指针，称方形结点为判定树的外部结点；对应的，圆形结点为内部结点。

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。

一般情况下，表长为 n 的折半查找的判定树的深度和含有 n 个结点的完全二叉树的深度相同。

假设 $n=2^h-1$ 并且查找概率相等

则

在n>50时，可得近似结果

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

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

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

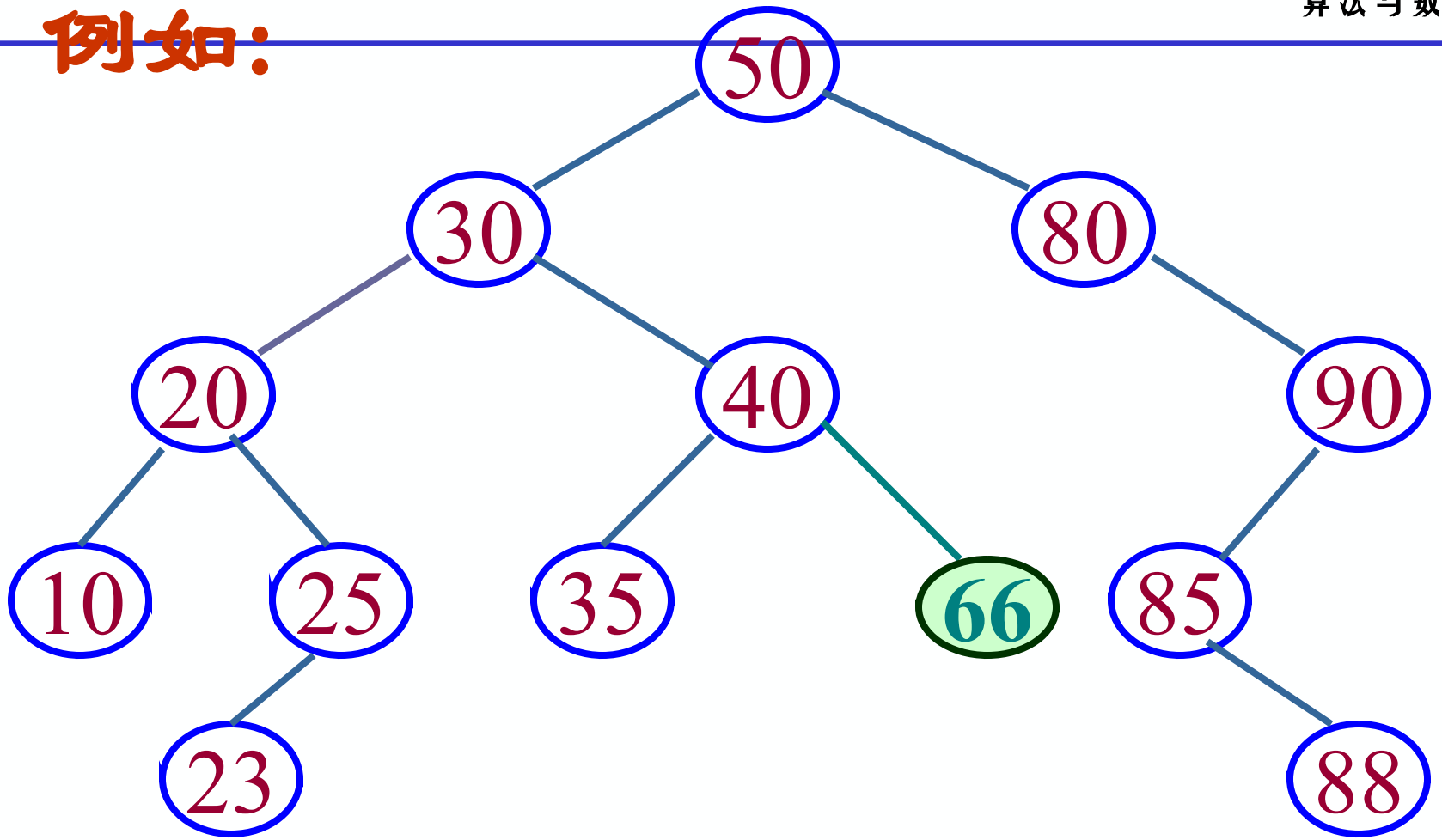
二叉排序树或是空树，或是满足如下性质的二叉树：

(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；

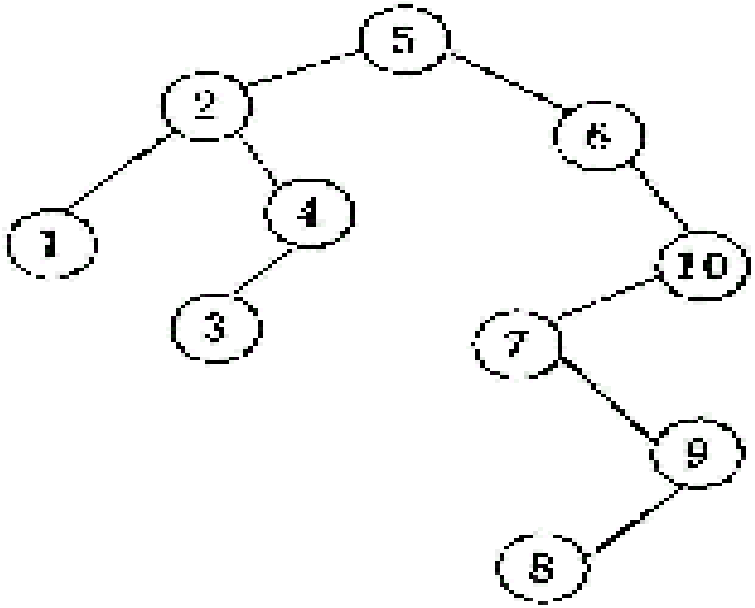
(3)其左右子树本身又各是一棵二叉排序树

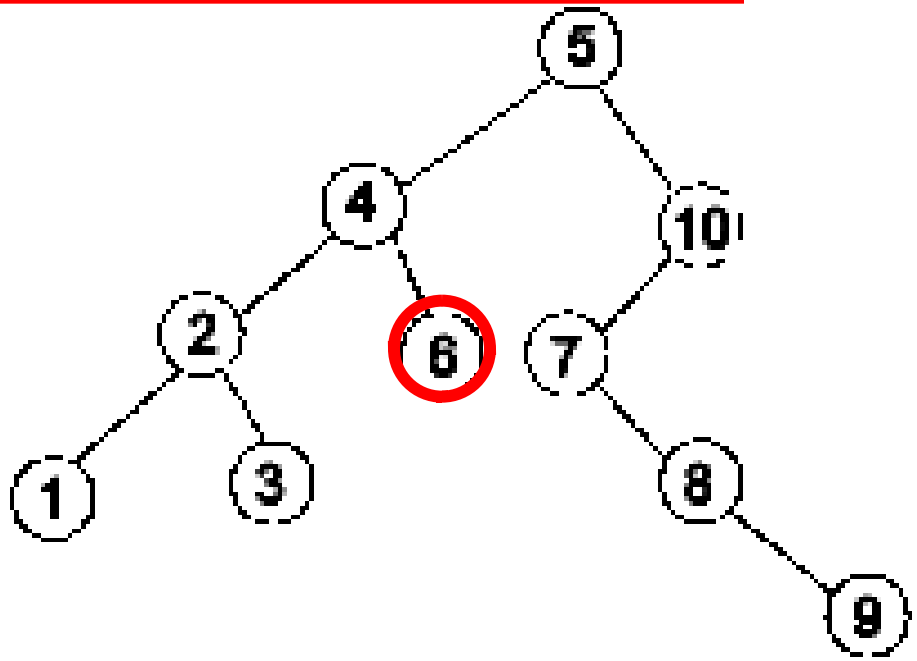
例如：



不是二叉排序树。

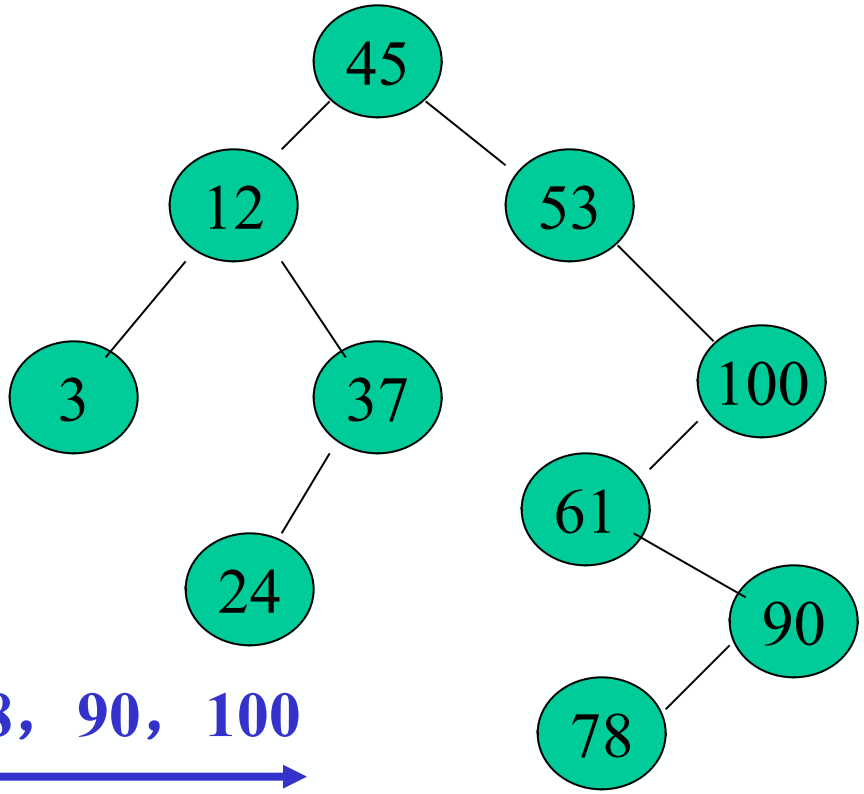
下列图形中，哪个不是二叉排序树 ？





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

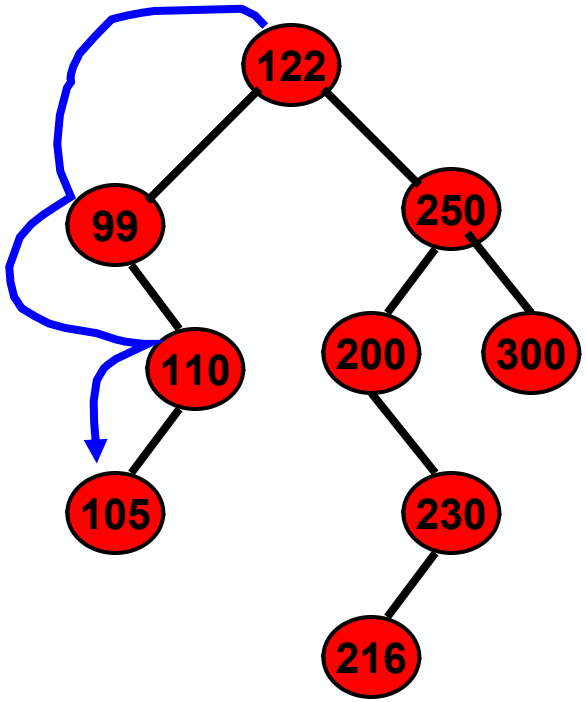
若查找的关键字等于根结点，成功

否则

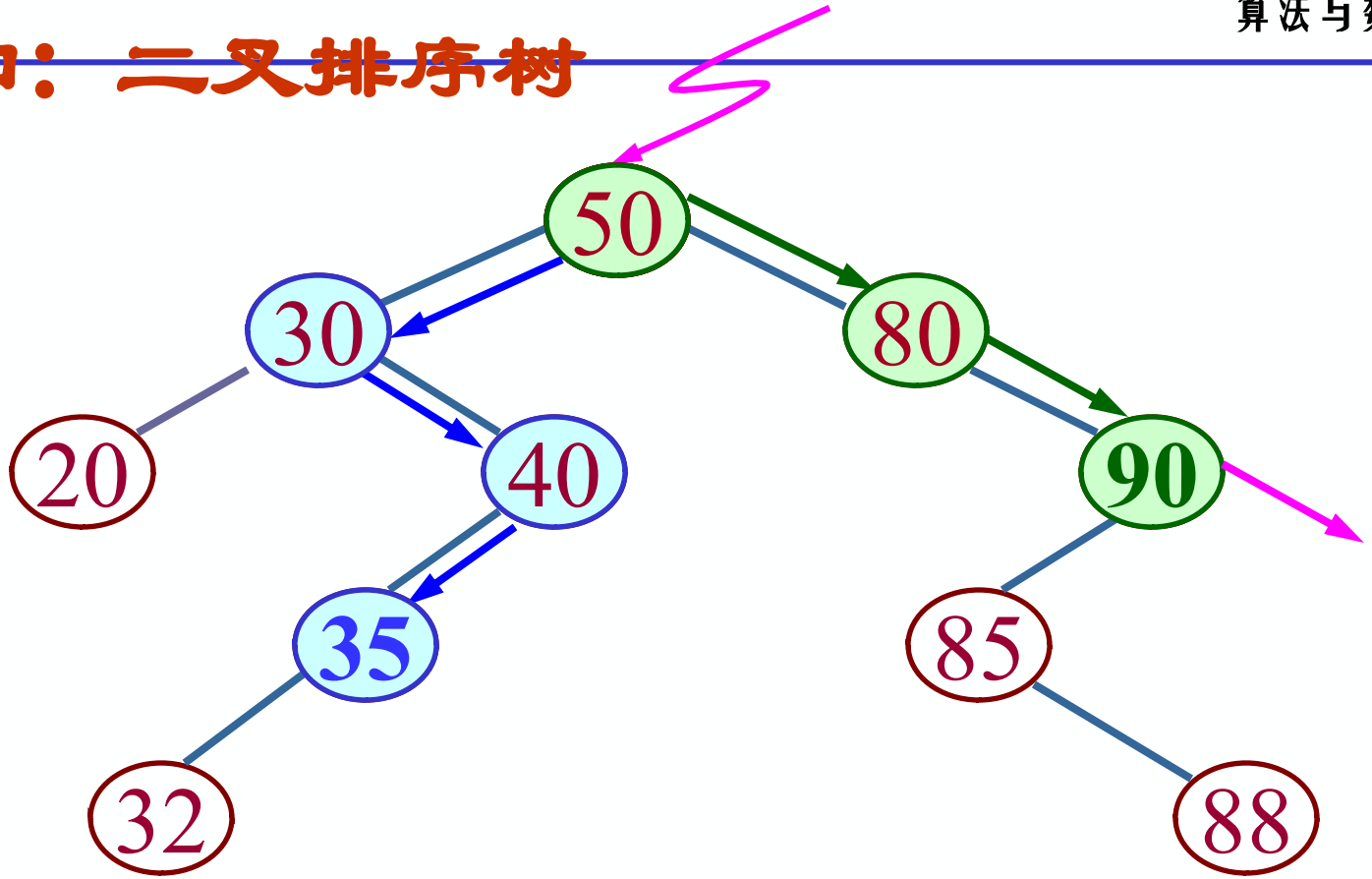
若小于根结点，查其左子树

若大于根结点，查其右子树

在左右子树上的操作类似



例如：二叉排序树



查找关键字

== 50 , 35 , 90 , 95 ,

从上述查找过程可见，

在查找过程中，生成了一条查找路径：

从根结点出发，沿着左分支或右分支逐层向下直至关键字等于给定值的结点；

——查找成功

或者

从根结点出发，沿着左分支或右分支逐层向下直至指针指向空树为止。

——查找不成功

【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

- **根据**动态查找表的定义，**“插入”操作在查找不成功时才进行**；
- 若二叉排序树为**空树**，则新插入的结点为**新的根结点**；否则，新插入的结点必为一个**新的叶子结点**，其**插入位置**由查找过程得到。

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

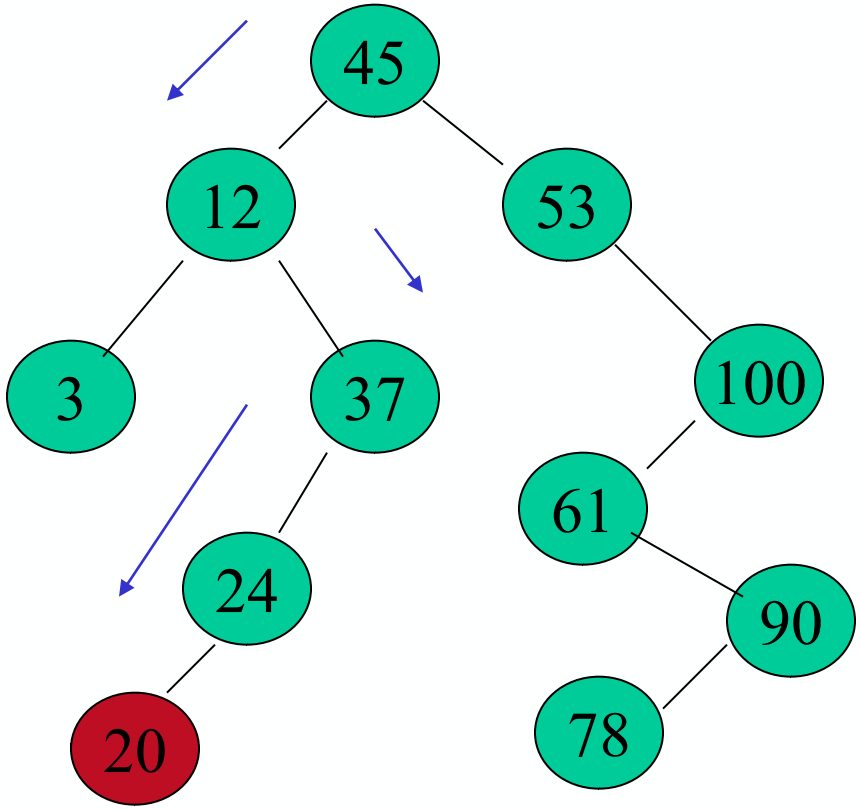
否则，继续在其左、右子树上查找

- ✓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



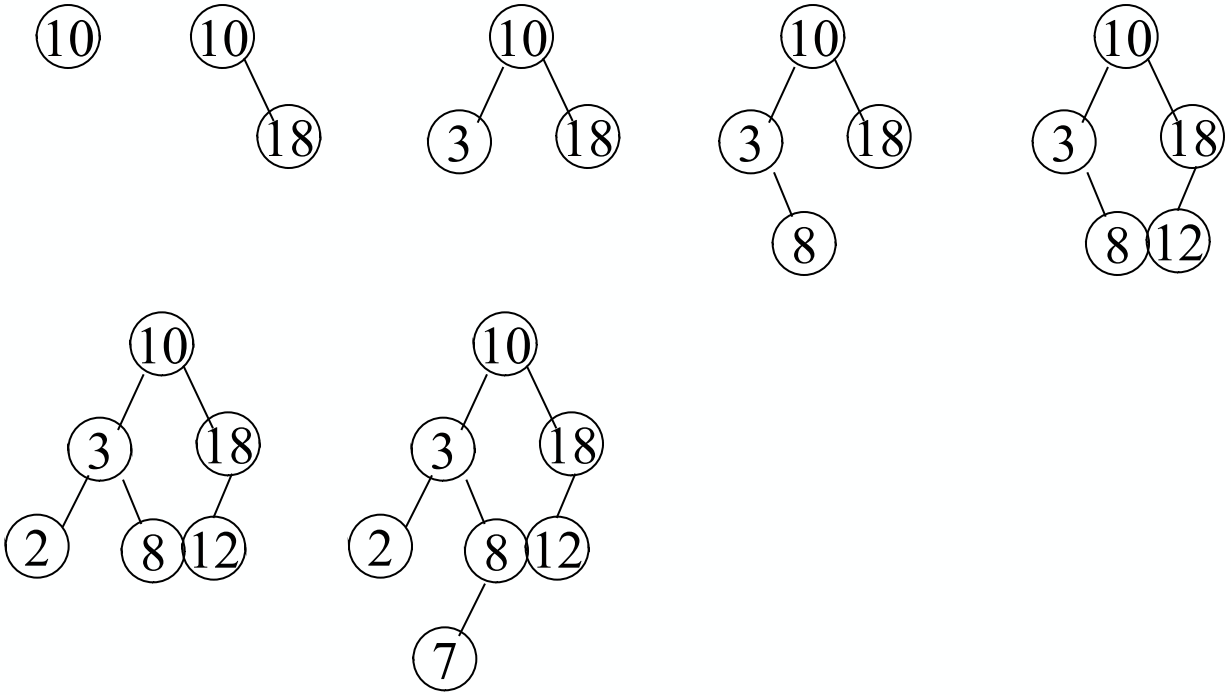
算法描述如下：

```
Status InsertBST (BiTree T, ElemType e ) {
  //当二叉排序树T中不存在关键字为e.key的数据元素时，则插入
      if (!T){
       S=new BSTNode;
       S->data=e;
       S->lchild=s->rchild=NULL;
      T=S;  }
else  if (e.key<T->data.key) InsertBST (T->lchild,e);
Else if (e.key>T->data.key) InsertBST (T->rchild,e);
} // InsertBST
```

二叉排序树的操作—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

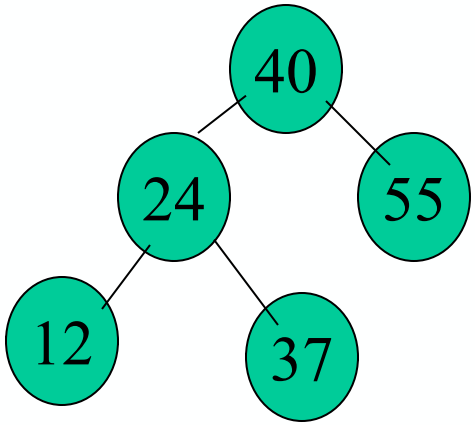
{10，18，3，8，12，2，7}



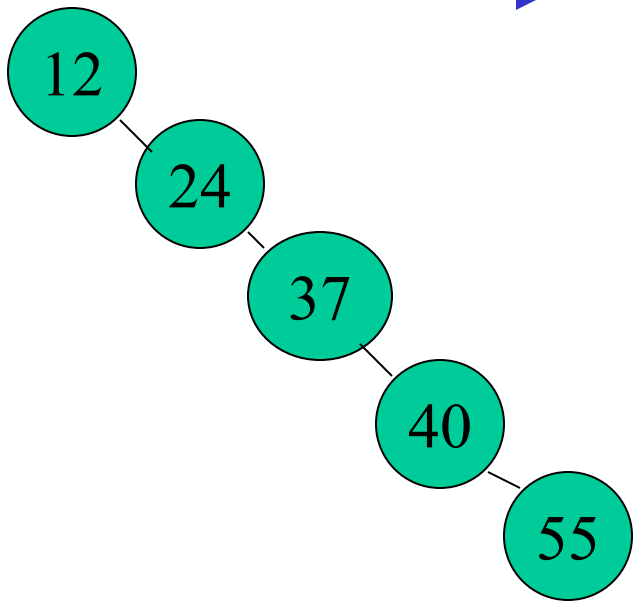
二叉排序树的操作一生成

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



二叉排序树的操作一生成

算法思想：

（1）将二叉排序树T初始化为空树；

（2）读入一个关键字key的结点，将此节点插入到T中。

（3）重复执行，直到输入结束。

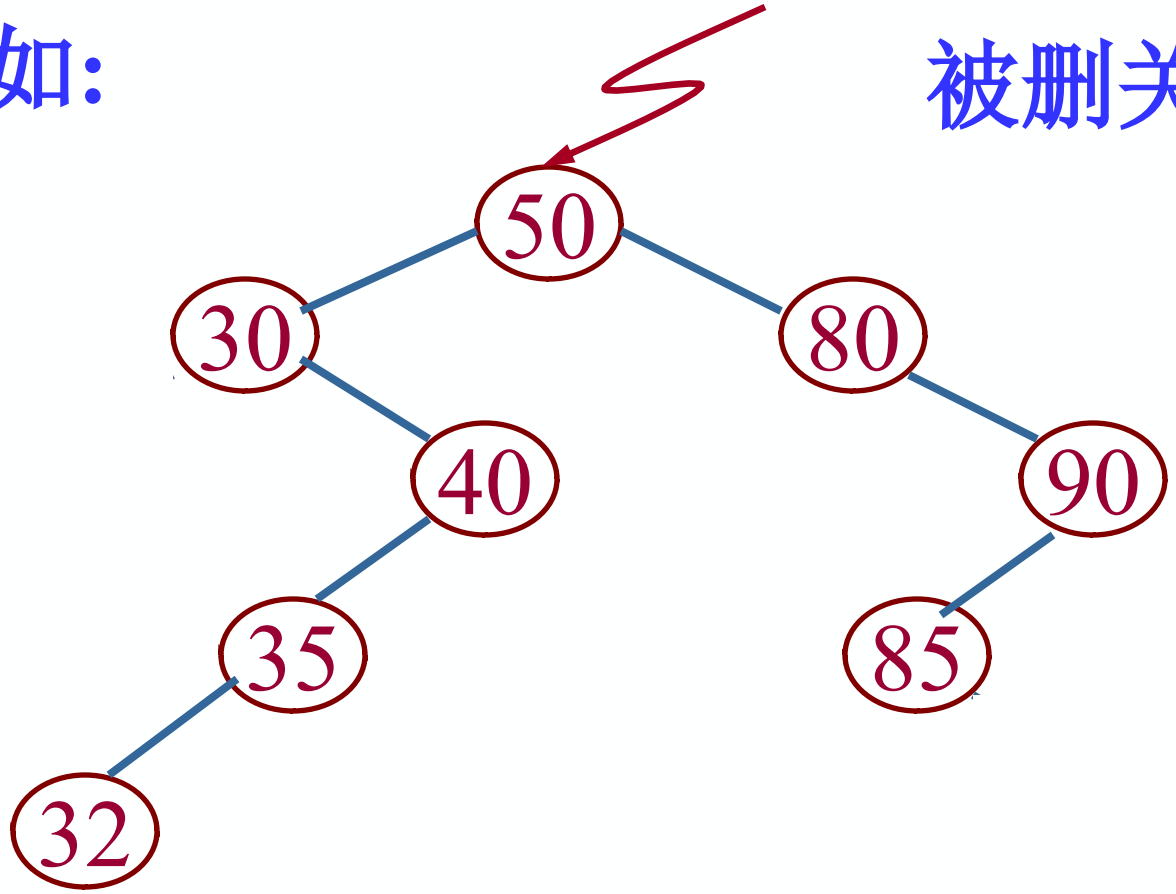
二叉排序树的操作—删除

- 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

（1）被删除的结点是叶子结点

例如：

被删关键字 = 88

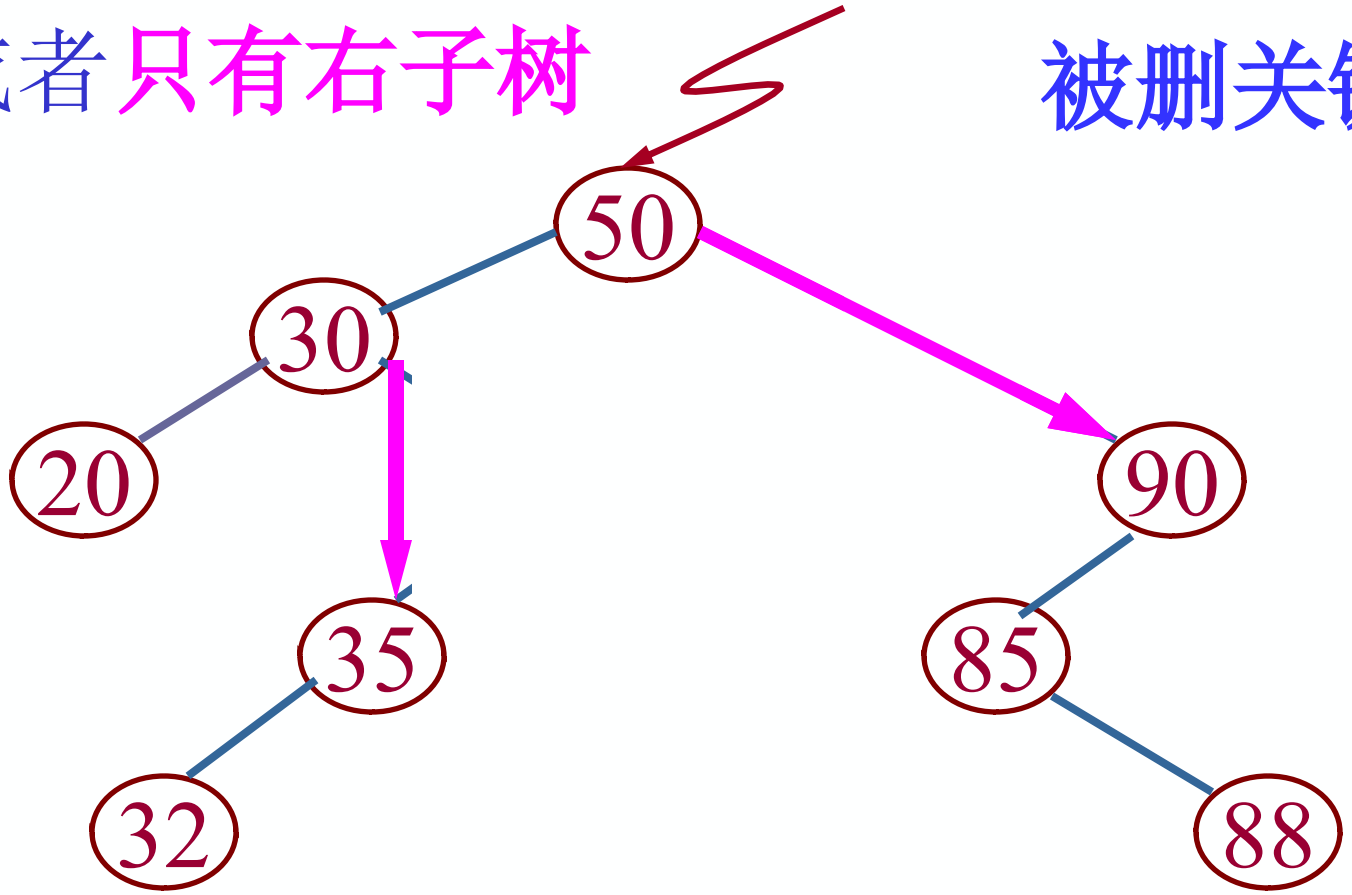


其双亲结点中相应指针域的值改为“空”

（2）被删除的结点只有左子树

或者只有右子树

被删关键字 = 80

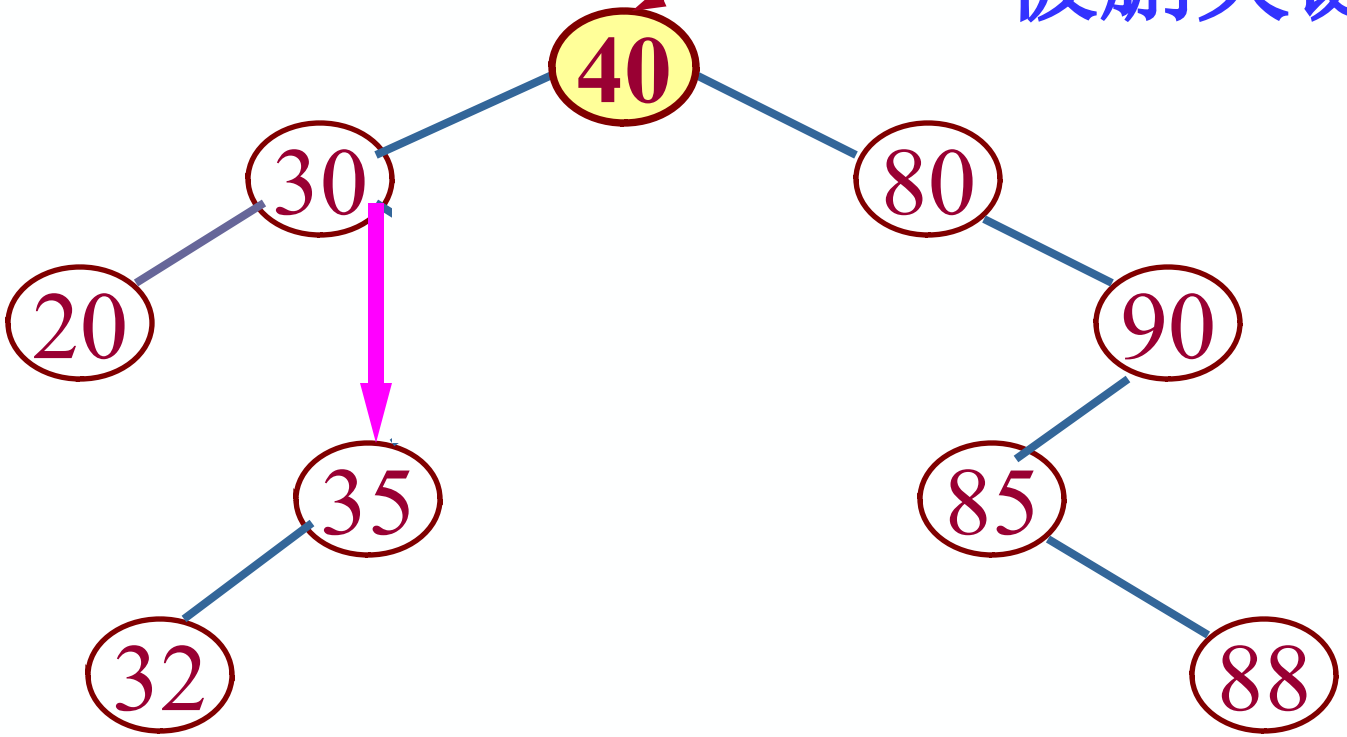


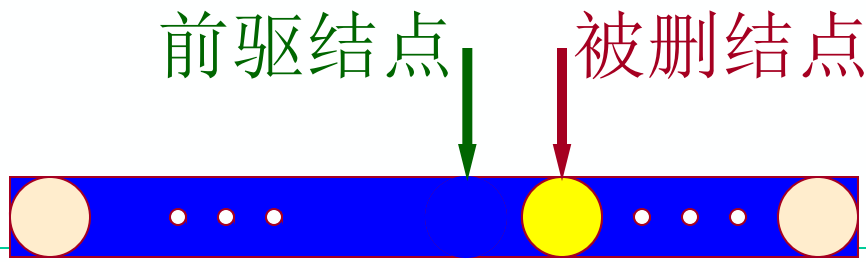
用被删除结点的左孩子或右孩子代替它

”。

（3）被删除的结点既有左子树，也有右子树

被删关键字 = 50





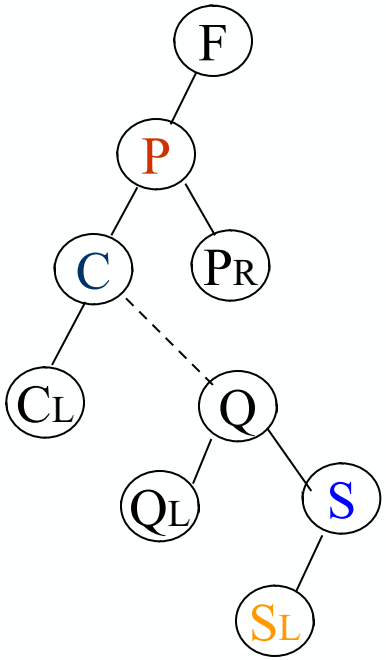
以其前驱替代之，然后再删除该前驱结点

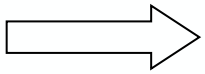
p左、右子树均非空

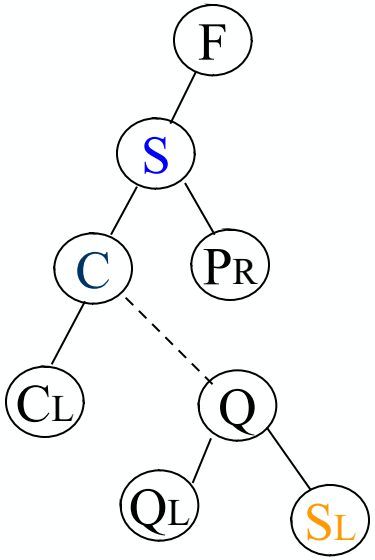
（1）沿p左子树的根C的右子树分支找到S，S的右子树为空，将S的左子树成为S的双亲Q的右子树，用S取代p

（2）若C无右子树，用C取代p

中序遍历：C_L C ……Q_L Q S_L S P P_R F

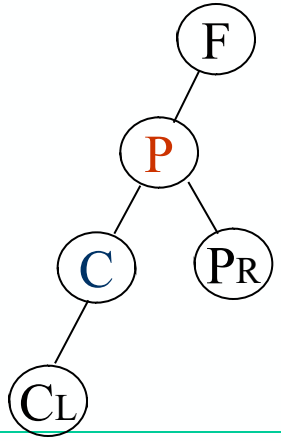


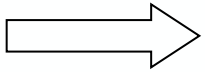


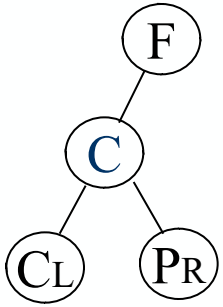


(5)

中序遍历：C_L C ……Q_L Q S_L S P_R F







中序遍历：C_L C P_R F

中序遍历：C_L C P P_R F (6)

–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。

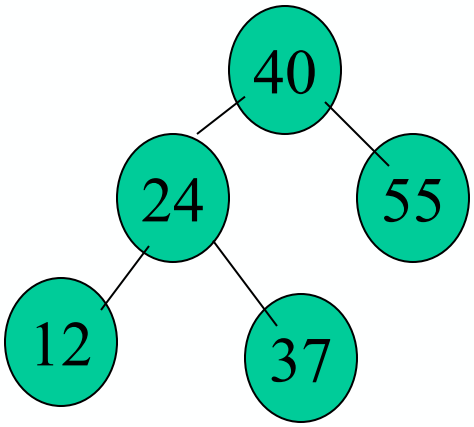
– 被删结点左、右子树都存在，可以在它的左子树中寻找中序下的最后一个结点(关键码最大)，用它的值填补到被删结点中，再来处理这个结点的删除问题。

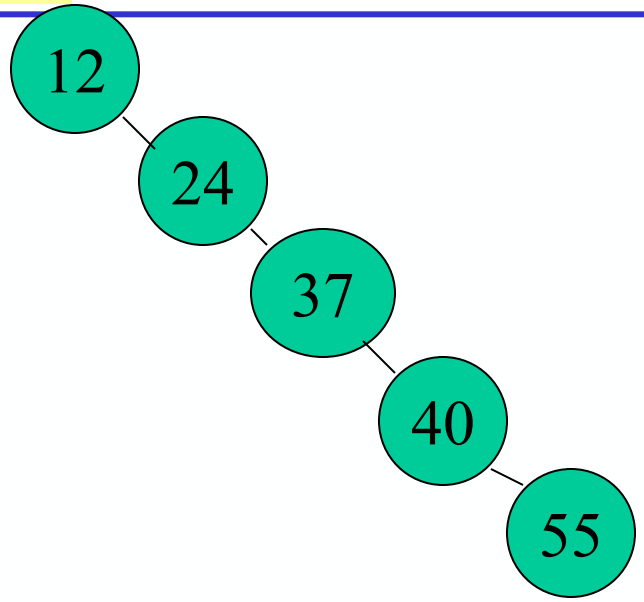
```
void DeleteBST(BSTree &T,Keytype key)
//从二叉排序树T中删除关键字等于key的结点
p=T;f=NULL;
//从根开始查找
while (p){
 if (p->data.key==key) break;
  f=p;
 if (p->data.key>key) p=p->lchild;
  else p=p->rchild;
}

if (!p) return;  //找不到则返回
```

```
//左右子树均不空
if((p->lchild)&&(p->rchild))
 {
  q=p;  s=p->lchild;
  while (s->rchild)
   {q=s;  s=s->rchild; }
 p->data=s->data;
  if(q!=p) q->rchild=s->lchild;
  else  q->lchild=s->lchild;

delete s;
}
```

```
else if(!p->rchild)//无右子树
{  q=p;  p=p->lchild;}

  else if (!p->lchild)//无左子树
 {   q=p;  p=p->rchild;}


if(!f)  T=p;   //删除的是根
else {
 if(q==f->rchild) {f->lchild=p; delete q;}
 if(q==f->lchild) {f->rchild=p; delete q;}
}//
```

查找的性能分析





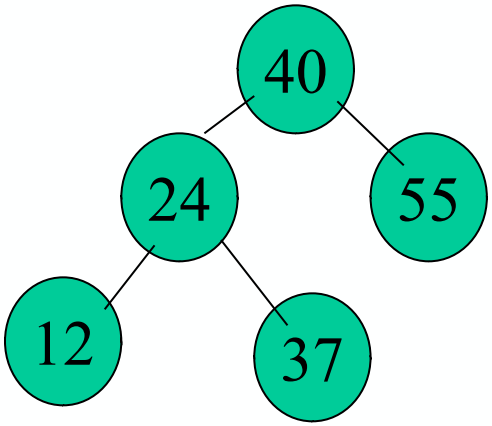
第i层结点需比较i次。在等概率的前提下，上述两图的平均查找长度为：

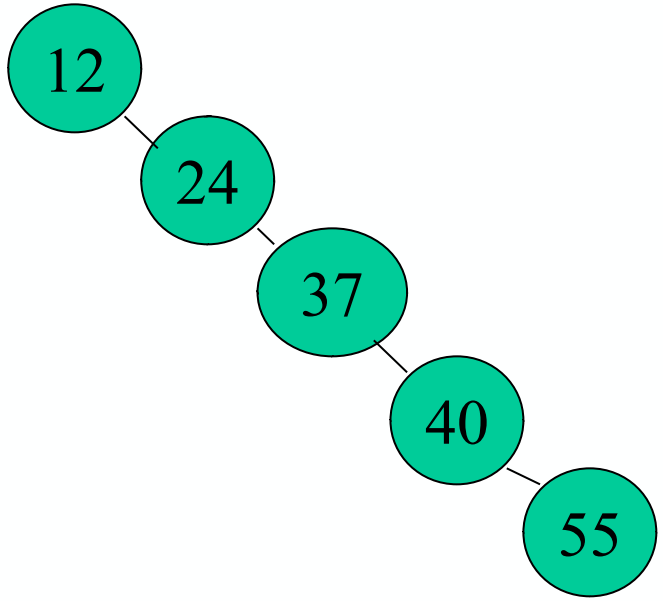
查找的性能分析

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）





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

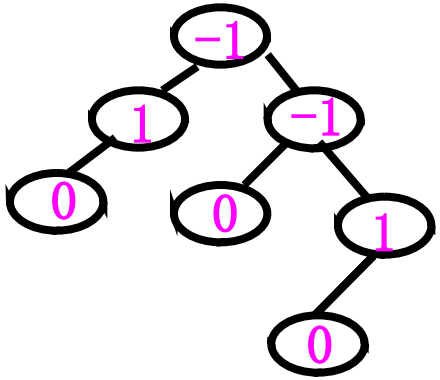
- 左、右子树是平衡二叉树；
- 所有结点的左、右子树深度之差的绝对值≤ 1

平衡因子:该结点左子树与右子树的高度差

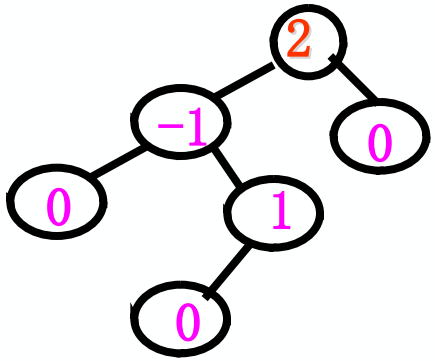
- 任一结点的平衡因子只能取：-1、0 或 1；如果树中任意一个结点的平衡因子的绝对值大于1，则这棵二叉树就失去平衡，不再是*AVL*树；

- 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。

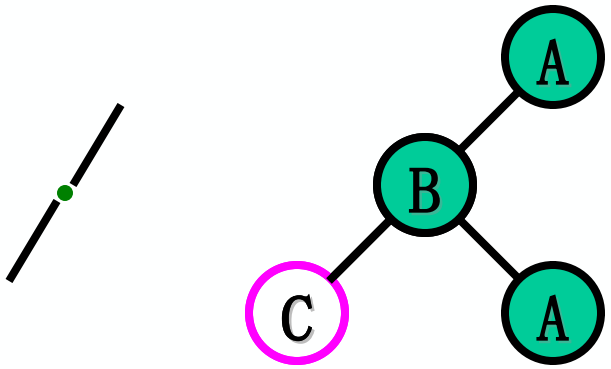
- ✓LL平衡旋转
- ✓RR平衡旋转
- ✓LR平衡旋转
- ✓RL平衡旋转

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。

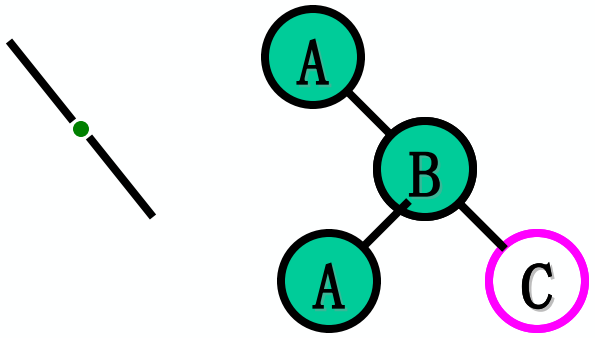
（以B为旋转轴）



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。

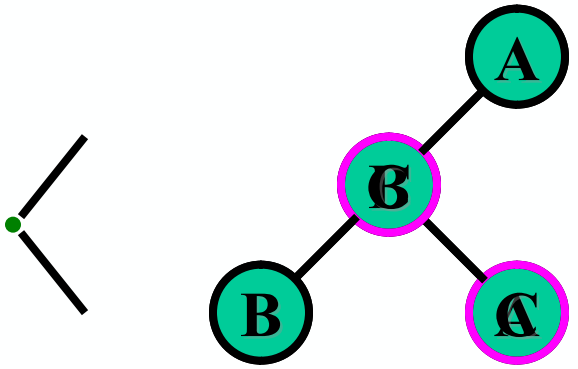
（以B为旋转轴）



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。

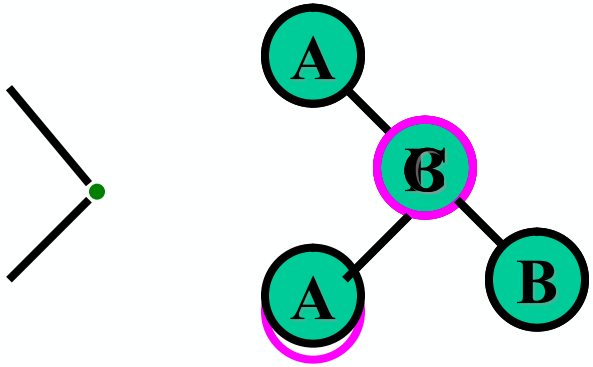
（以插入的结点C为旋转轴）



4）RL平衡旋转：

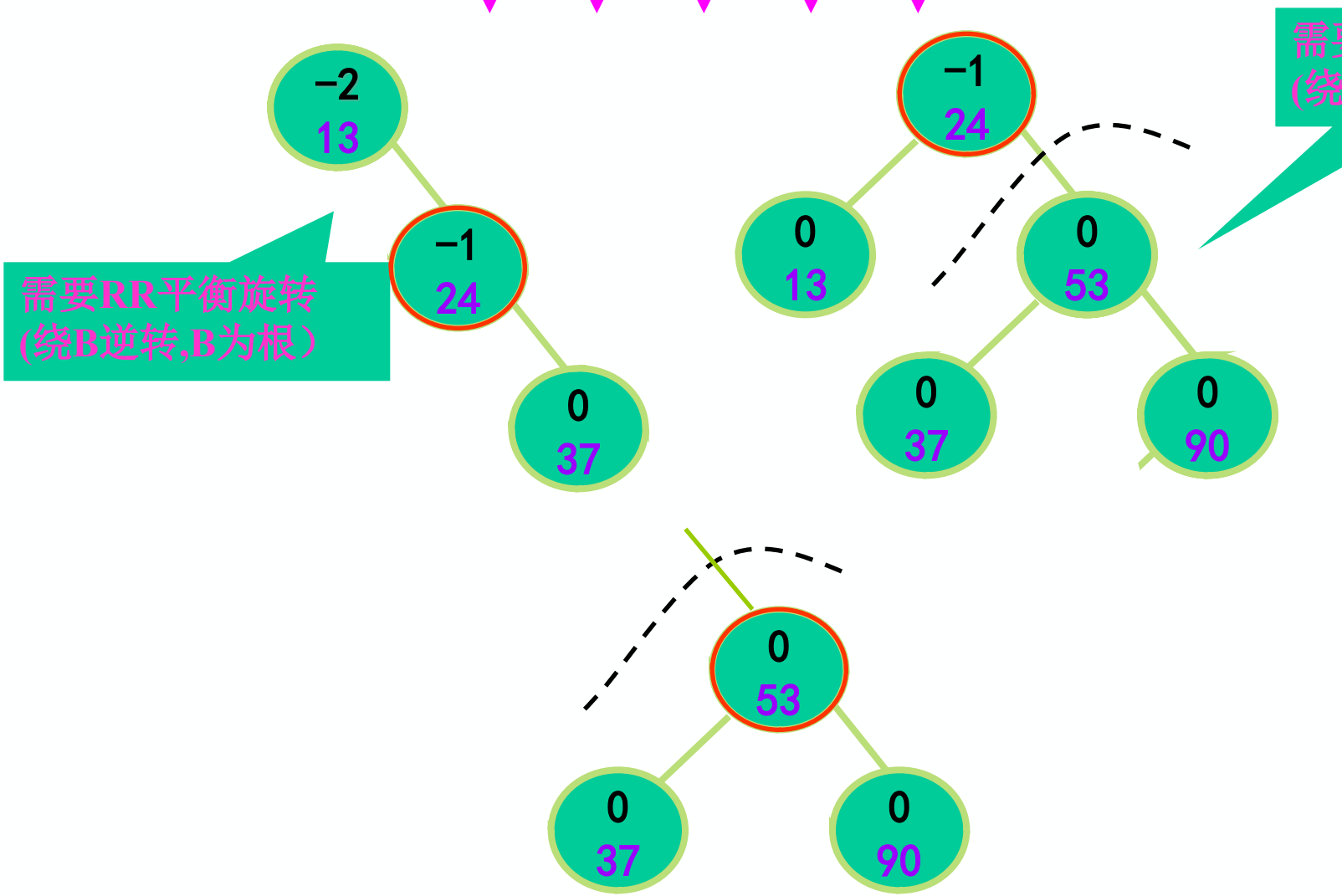
若在A的右子树的左子树上插入结点，使A的平衡因子从-1增加至-2，需要先进行顺时针旋转，再逆时针旋转。

（以插入的结点C为旋转轴）



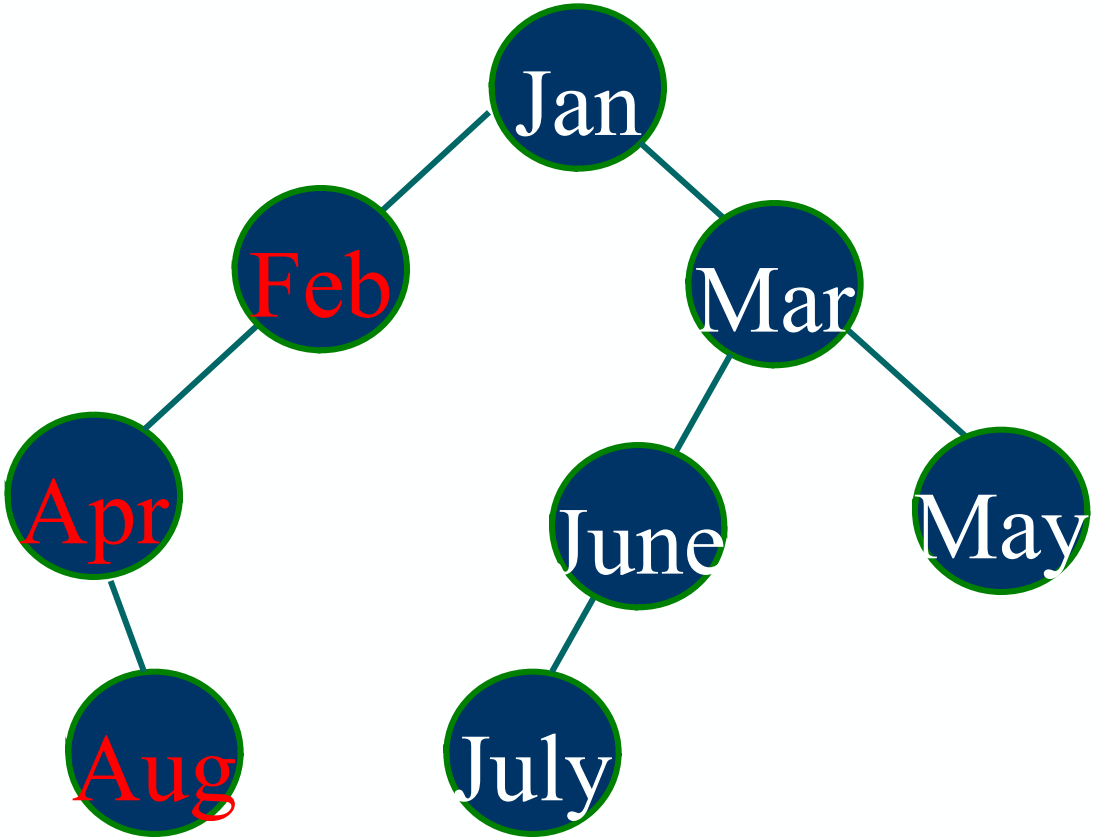
练习：请将下面序列构成一棵平衡二叉排序树

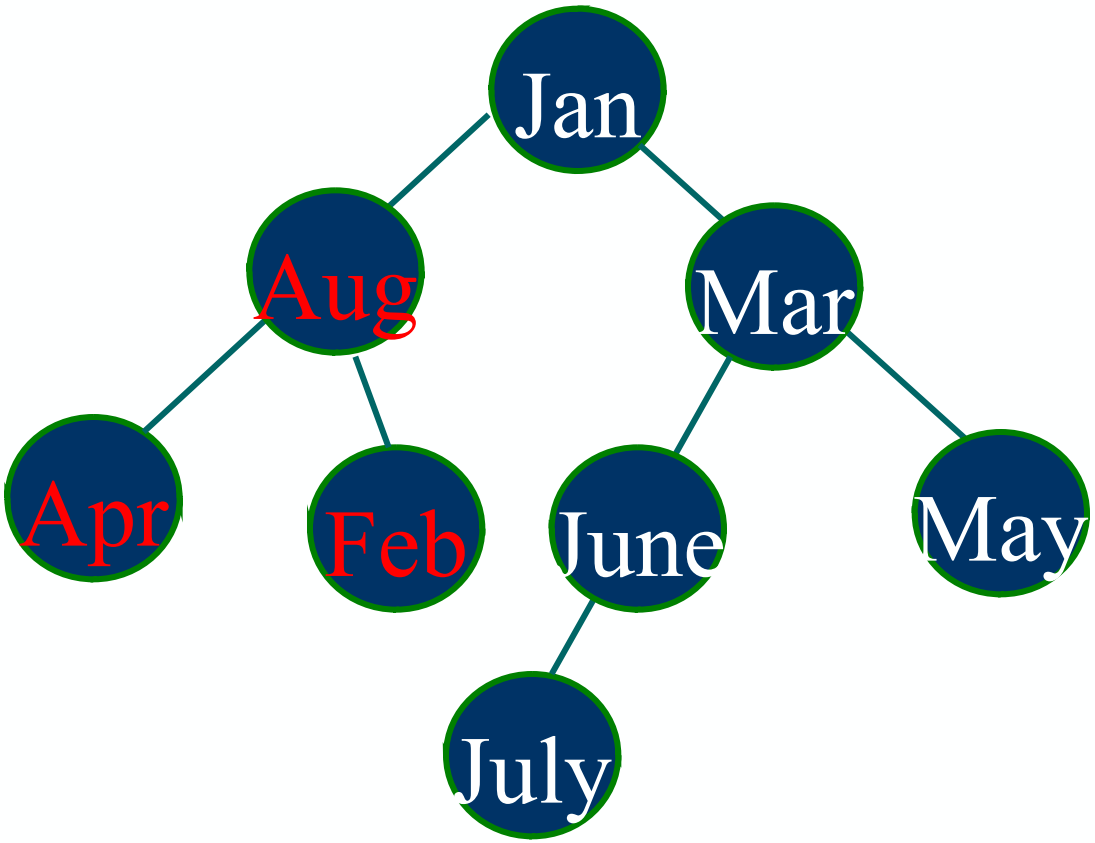
（ 13，24，37，90，53）

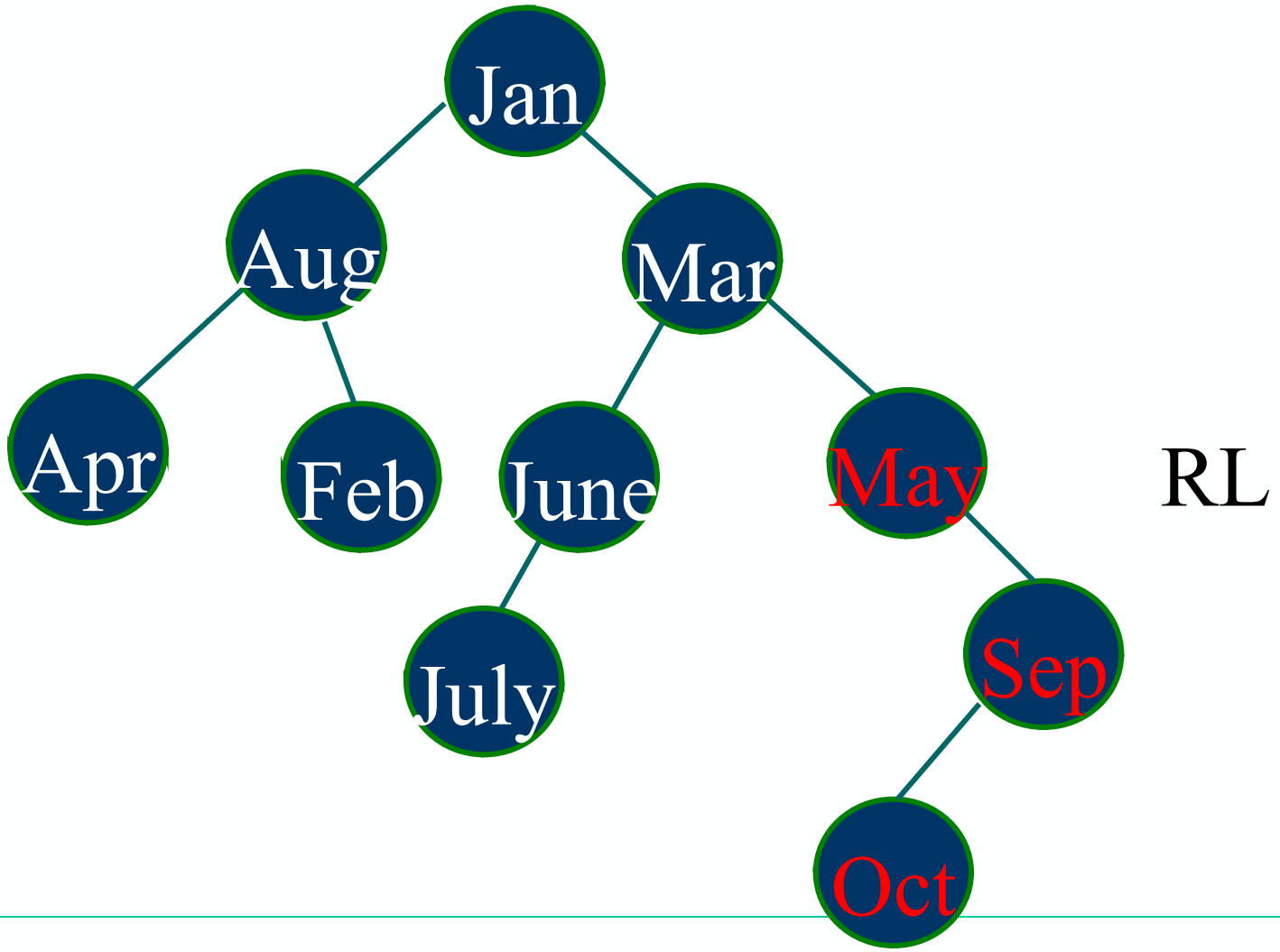


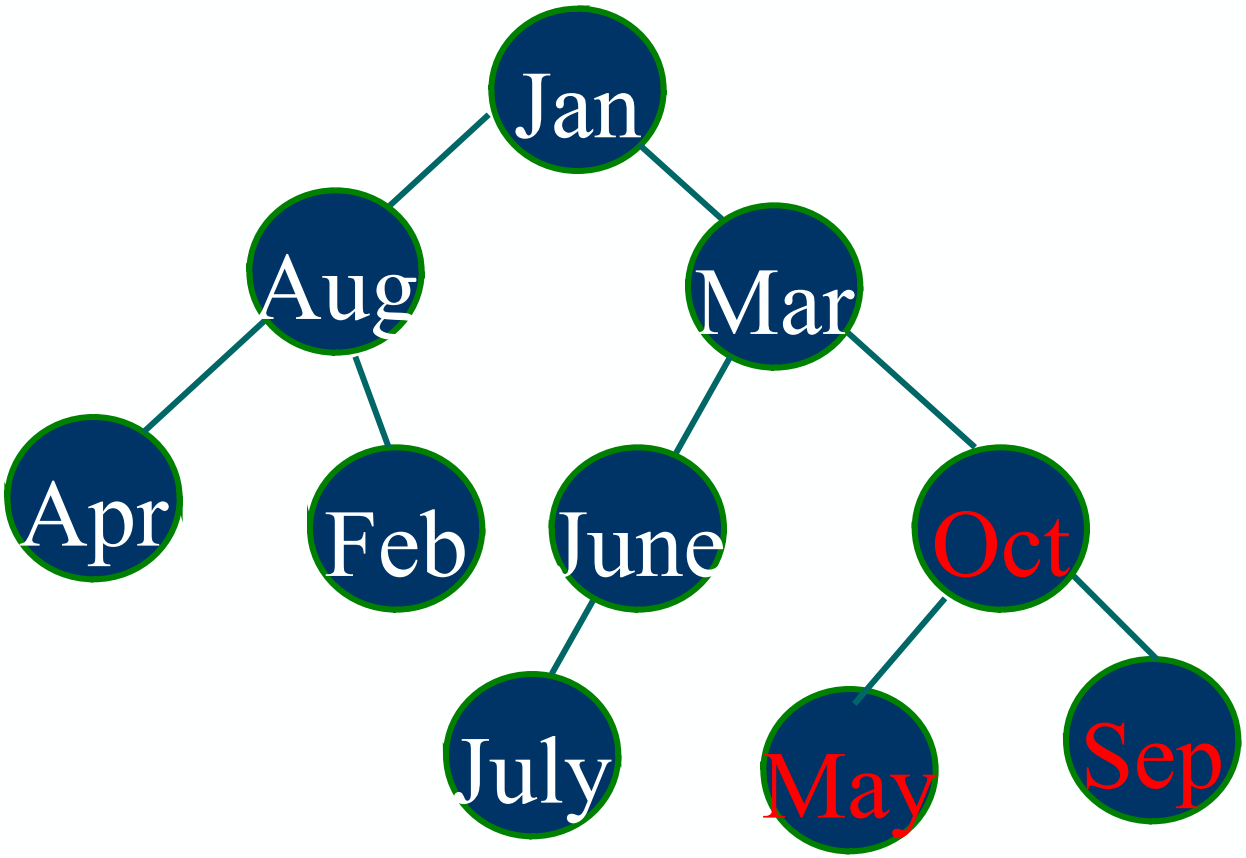
(Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec)

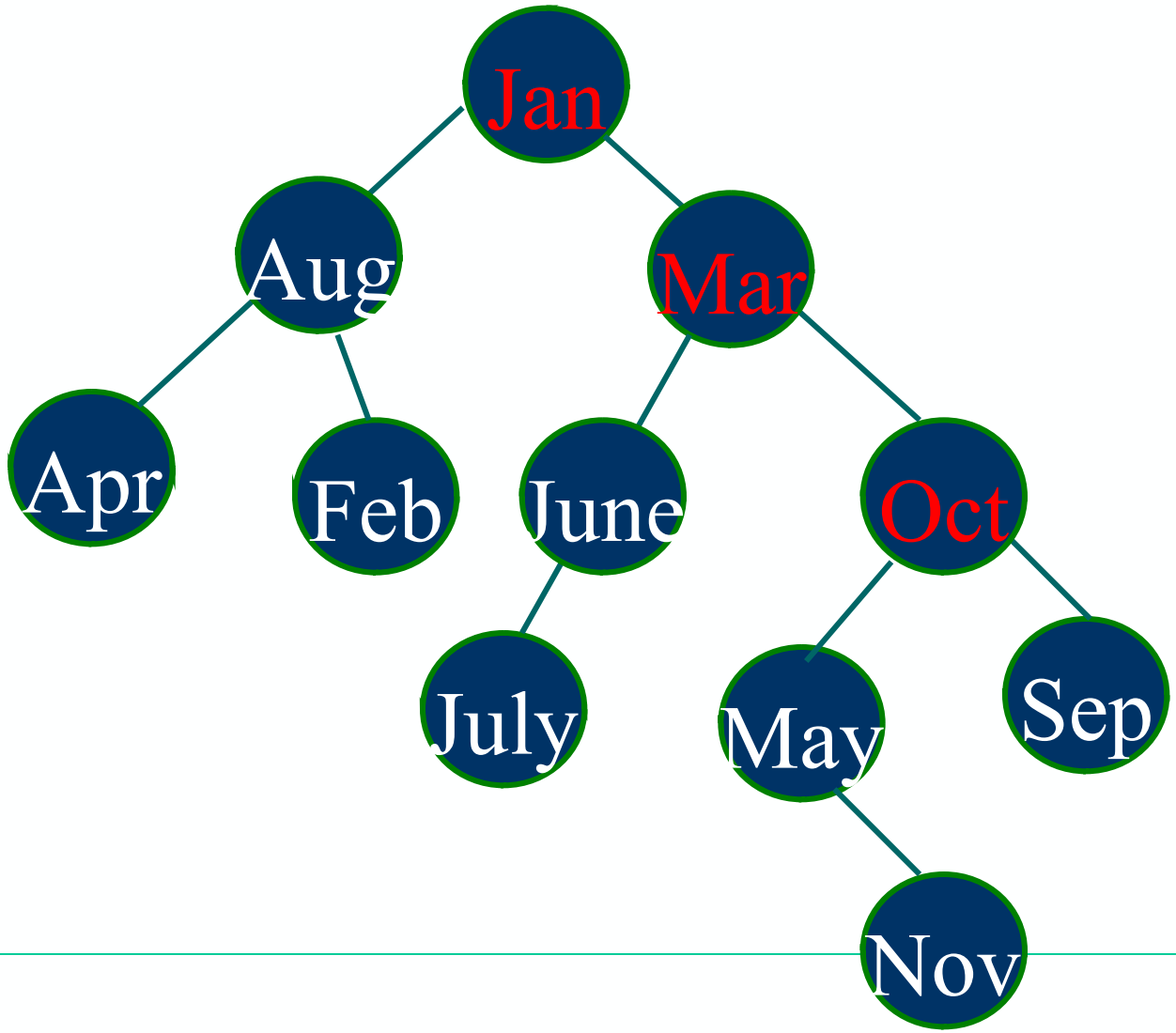
LR



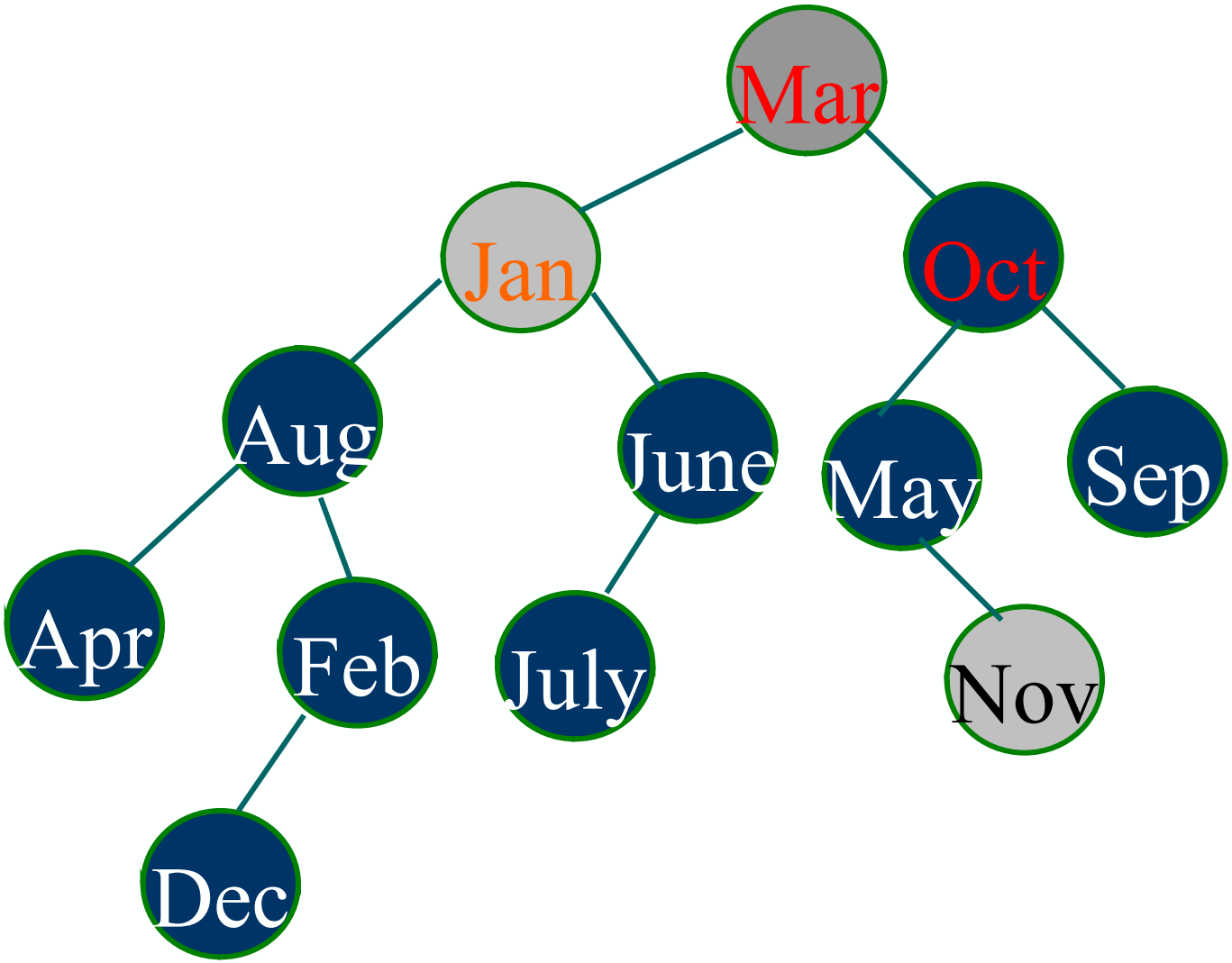

Jan
Aug
Mar
Apr
Feb
June
May
July


Jan
Aug
Mar
Apr
Feb
June
May
July
Sep
Oct
RL


Jan
Aug
Mar
Apr
Feb
June
Oct
July
May
Sep



RR


Mar
Jan
Oct
Aug
June
May
Sep
Apr
Feb
July
Nov
Dec

在平衡的二叉排序树BBST上插入一个新的数据元素e的递归算法：

（1）若BBST为空树，则插入的新结点为BBST的根结点，树的深度增加1。

（2）若e的关键字和BBST的根结点的关键字相等，则不进行插入。

（3）若e的关键字小于BBST的根结点的关键字，而且在BBST的左子树中不存在和e相同的结点，则将e插入到左子树中，并且当插入之后的左子树深度增加1时，分别考虑下列情况：

① BBST的根结点的平衡因子为-1，则将根节点的平衡因子更改为0，BBST的深度不变。

②BBST的根结点的平衡因子为0，则将根节点的平衡因子更改为1，BBST的深度增1。

③ BBST的根结点的平衡因子为1，

- 若BBST插入后的左子树根结点的平衡因子为1，则需进行LL平衡处理，处理后，将根结点和其右子树根结点的平衡因子更改为0，树的深度不变。
- 若BBST插入后的左子树根结点的平衡因子为-1，则需进行LR平衡处理，处理后，修改根结点和其左右子树根结点的平衡因子，树的深度不变。

（4）若e的关键字大于BBST的根结点的关键字，而且在BBST的右子树中不存在和e相同的结点，则将e插入到右子树中，并且当插入之后的右子树深度增加1时，分别考虑不同的情况进行处理（类似（3））

平衡树查找的分析：

在平衡树上进行查找的过程和二叉排序树相同，因此，查找过程中和给定值进行比较的关键字的个数不超过平衡树的深度。

问：含n个关键字的二叉平衡树可能达到的最大深度是多少？

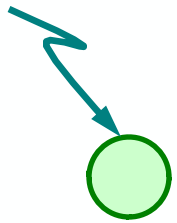
先看几个具体情况：

$n = 0$

空树

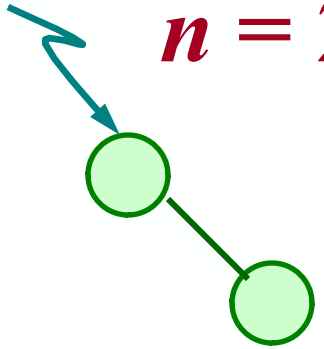
最大深度为 0

$n = 1$



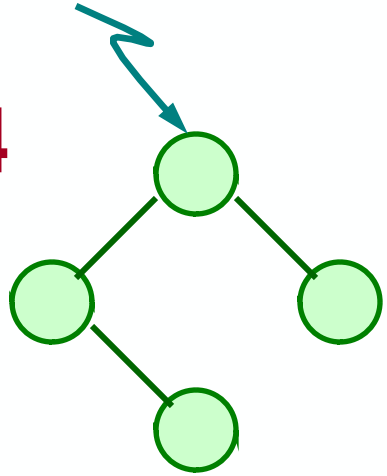
最大深度为 1

$n = 2$



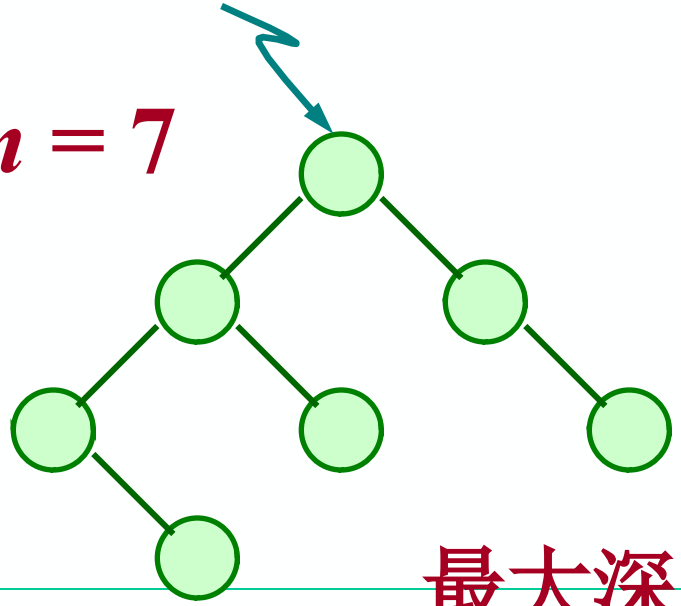
最大深度为 2

$n = 4$



最大深度为 3

$n = 7$



最大深度为 4

在**二叉平衡树**上进行查找时，

查找过程中和给定值**进行比较的关键字的个数和** ***log(n)*** 相当。

1. B-树的定义

B-树是一种 平衡 的 多路 查找 树：

在 m 阶的B-树上，每个非终端结点可能含有：

n 个关键字 K_i（$1 \le i \le n$）　$n<m$

n 个指向记录的指针 D_i（$1 \le i \le n$）

$n+1$ 个指向子树的指针 A_i（$0 \le i \le n$）

——— 多叉树的特性

- 非叶结点中的多个关键字均自小至大有序排列，即：$K_1 < K_2 < \ldots < K_n$ ；
- A_{i-1} 所指子树上所有关键字均小于 K_i ；
- A_i 所指子树上所有关键字均大于 K_i ；

——— 查找树的特性

- 树中所有叶子结点均不带信息，且在树中的同一层次上；
- 根结点或为叶子结点，或至少含有两棵子树；
- 其余所有非叶结点均至少含有$\lceil m/2 \rceil$棵子树，至多含有 m 棵子树；

——— 平衡树的特性

2.查找过程：

从根结点出发，沿指针**搜索结点**和**在结点内进行**顺序（或折半）**查找** 两个过程交叉进行。

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**；

若**查找不成功**，则**返回插入位置**。

B+树

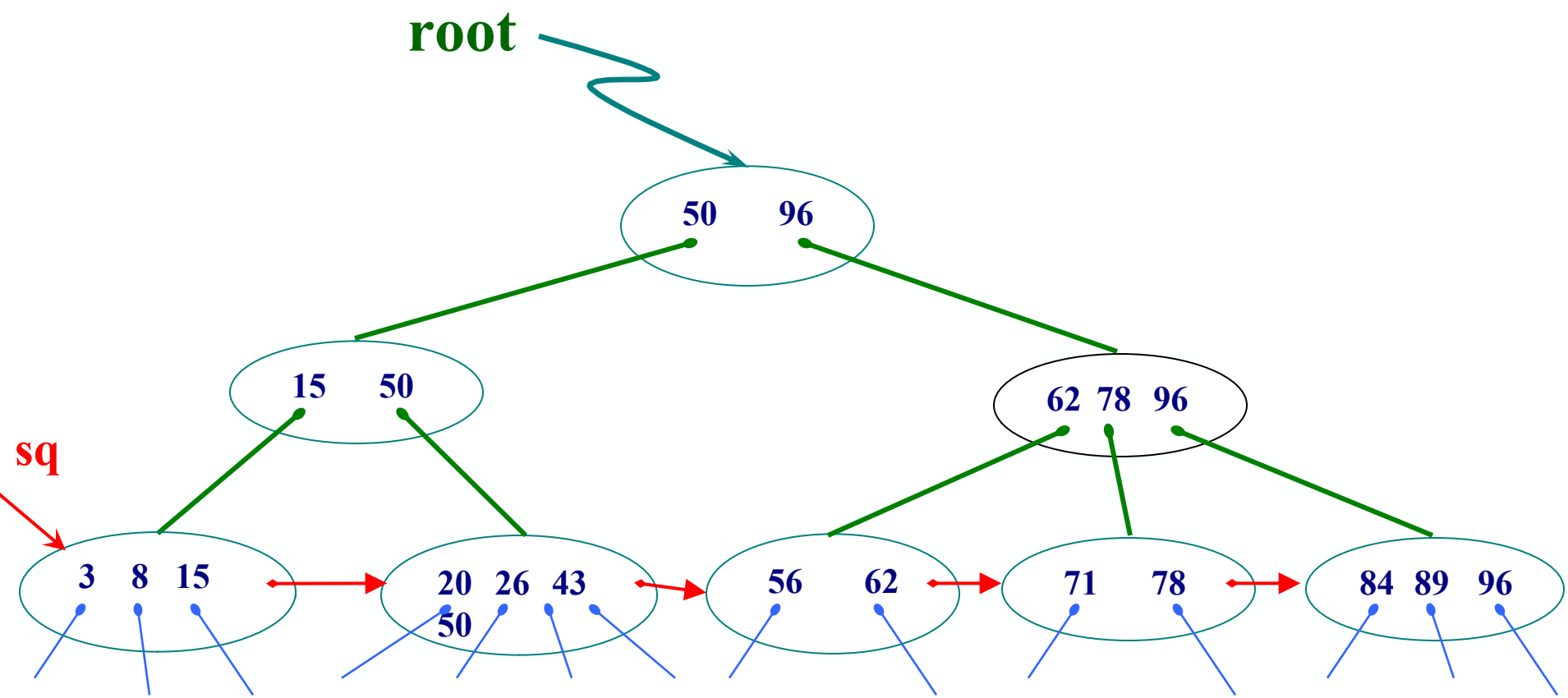
是B-树的一种变型

1. B^+树的结构特点：

※ 每个叶子结点中含有 ***n*** 个关键字和 ***n*** 个指向记录的指针；并且，所有**叶子**结点彼此相**链接**构成一个有序链表，其头指针指向含最小关键字的结点；

※ 每个非叶结点中的关键字 K_i 即为其相应指针 A_i 所指子树中关键字的最大值；

※ 所有叶子结点都处在同一层次上，每个叶子结点中关键字的个数均介于 $\lceil m/2 \rceil$ 和 m 之间。

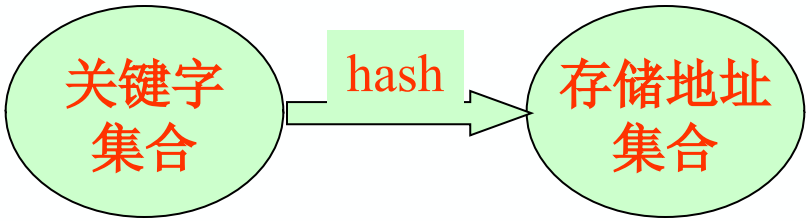

root
50 96
15 50
62 78 96
sq
3 8 15
20 26 43
50
56 62
71 78
84 89 96

知识回顾

- 静态查找表
  - 顺序查找
  - 折半查找
- 动态查找表
  - 二叉排序树
  - 平衡二叉树
  - B-树
  - B+树

7.3 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200101181020**1**的所有信息存入V[01]单元；
将200101181020**2**的所有信息存入V[02]单元；
……
将200101181023**1**的所有信息存入V[31]单元。

查找200101181021**6**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、散列函数)：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

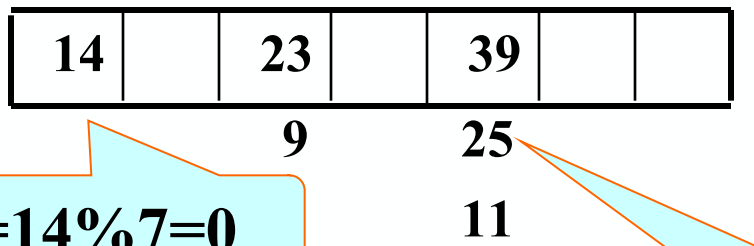
key1≠key2，但H(key1)=H(key2)

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



6个元素用7个地址应该足够!

H(14)=14%7=0

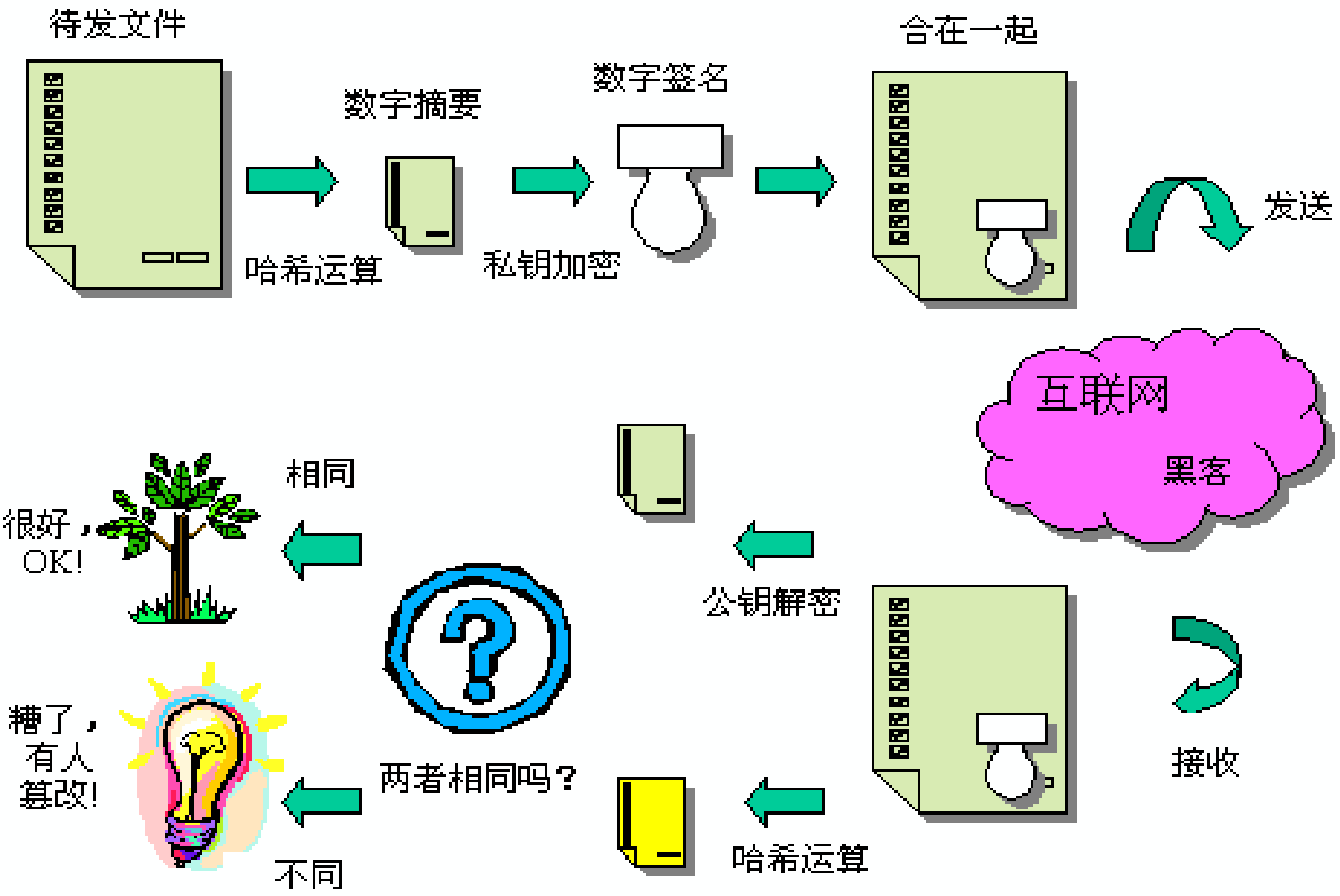
H(25)=25%7=4

H(11)=11%7=4

同义词

有冲突

哈希函数在信息安全领域中的应用



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）

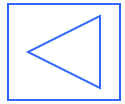
Hash(key)=key mod p （p是一个整数)

关键： 如何选取合适的p？

技巧： 设表长为m，取p≤m且为质数

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

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

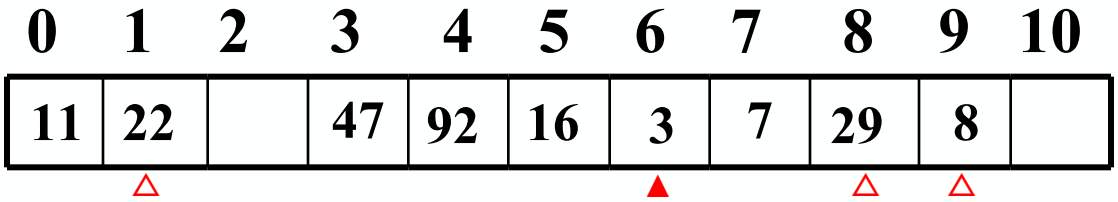
其中：m为哈希表长度

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

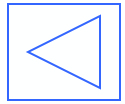
③ 3 连续移动了两次

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

$H_i=(Hash(key)\pm d_i) \bmod m$
其中：m为哈希表长度，m要求是某个4k+3的质数；
　　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	

△ ▲ △ △

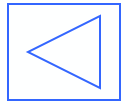
Hash(3)=3，哈希地址冲突，由
$H_1=(Hash(3)+1^2) \bmod 11=4$，仍然冲突；
$H_2=(Hash(3)-1^2) \bmod 11=2$，找到空的哈希地址，存入。

伪随机探测法

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

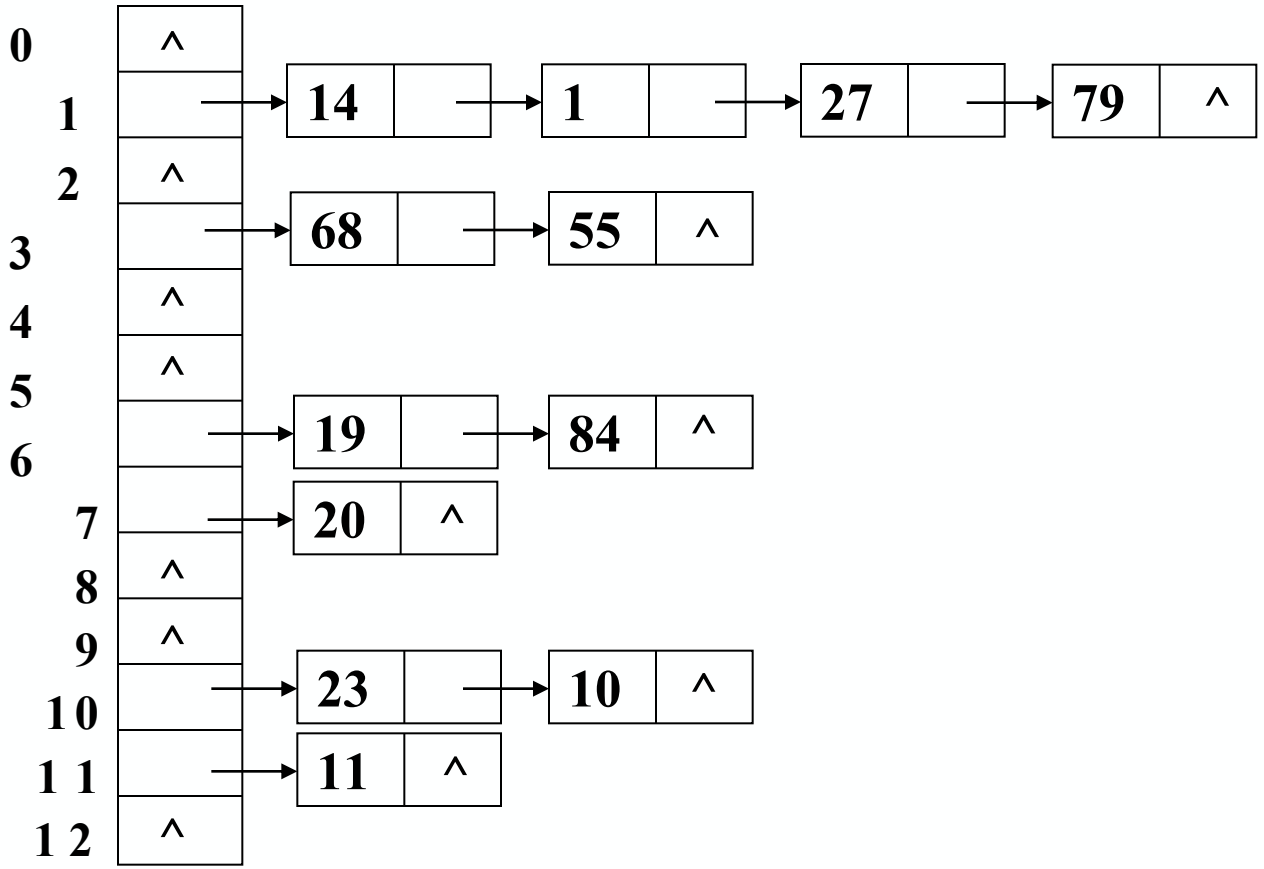
其中：m为哈希表长度

d_i 为随机数



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

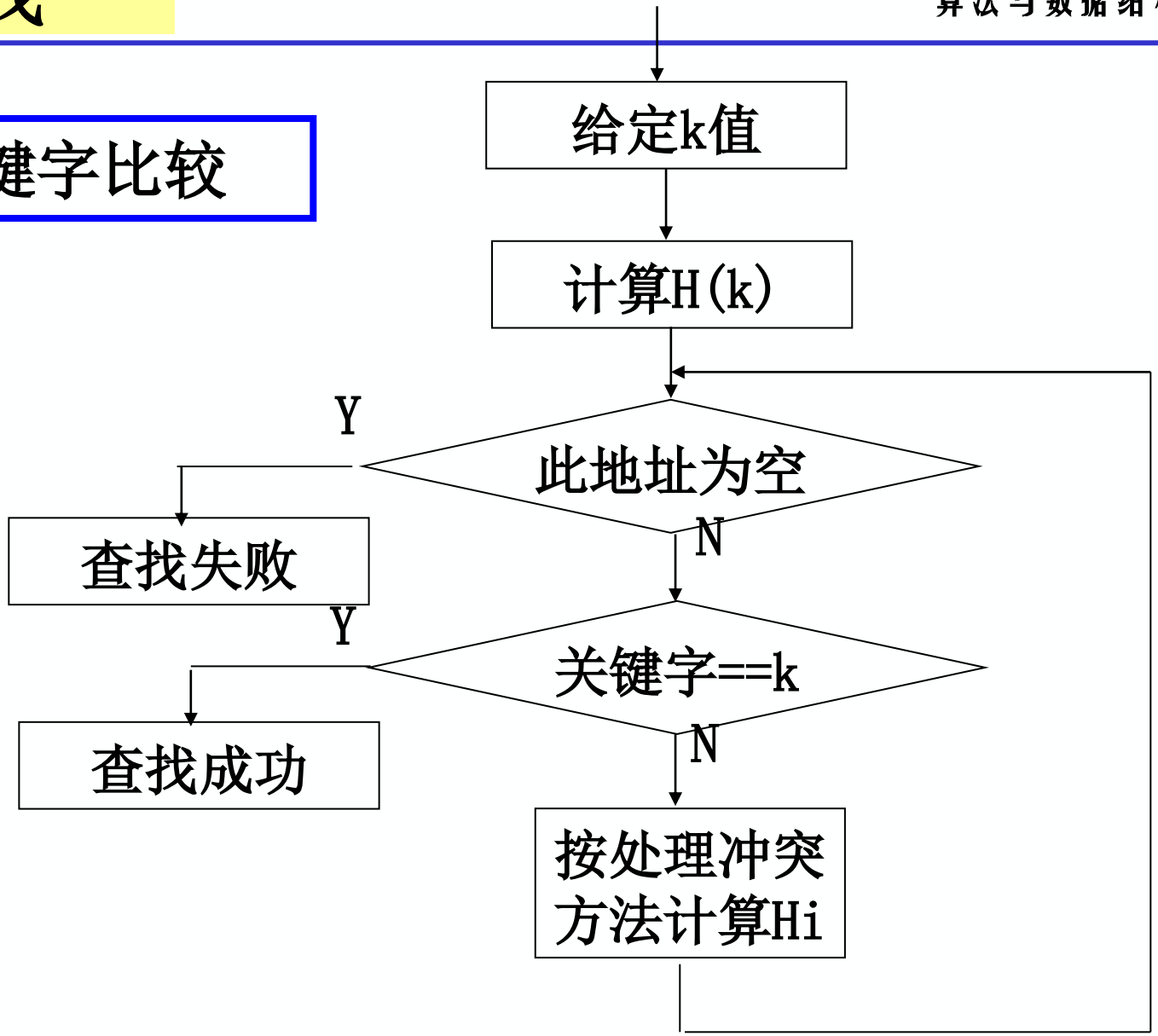


链地址法的优点：

- 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

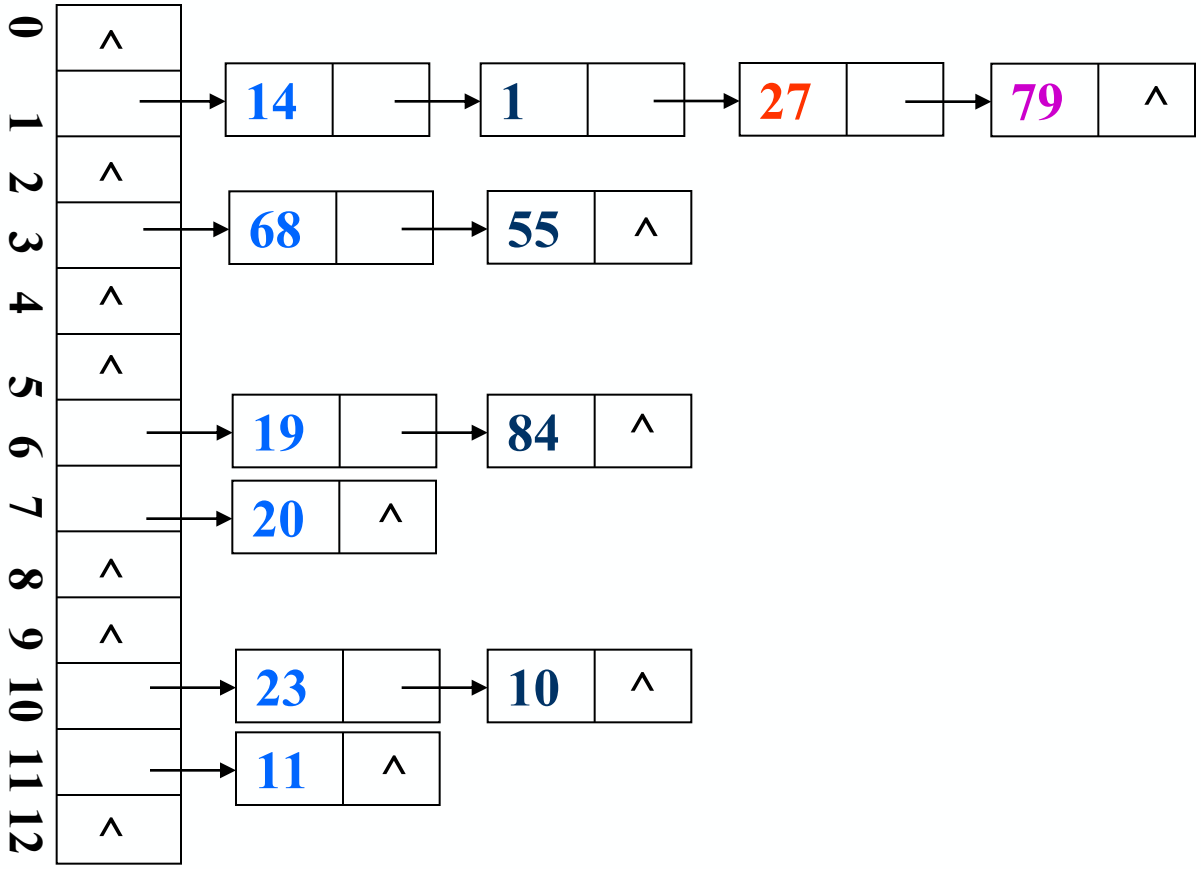
H(27)=1 冲突，H1=(1+1)MOD16=2
冲突，H2=(1+2)MOD16=3
冲突，H3=(1+3)MOD16=4

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！既不是口格的O(1)，也不是O(n)

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
- 链地址法优于开地址法
- 除留余数法作哈希函数优于其它类型函数

练习

设一组关键字为（7，15，20，31，48，53，64，19，82，91），Hash函数H（key）= key MOD 11，Hash表表长m=11，用线性探测法解决冲突，试构造Hash表。

并：（1）给出构造过程

（2）求出平均查找长度ASL

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.掌握二叉排序树的插入算法，了解二叉排序树的删除算法；

4.熟练掌握平衡二叉树的旋转方法

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