第7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*)**

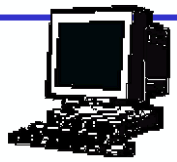
$$ASL = \sum_{i=1}^{n} p_i c_i$$

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



一、顺序查找（线性查找）

二、折半查找（二分或对分查找）

三、分块查找

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;
```

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

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

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。

折半查找

适用范围：顺序表、有序

基本思想（分治法）

- 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

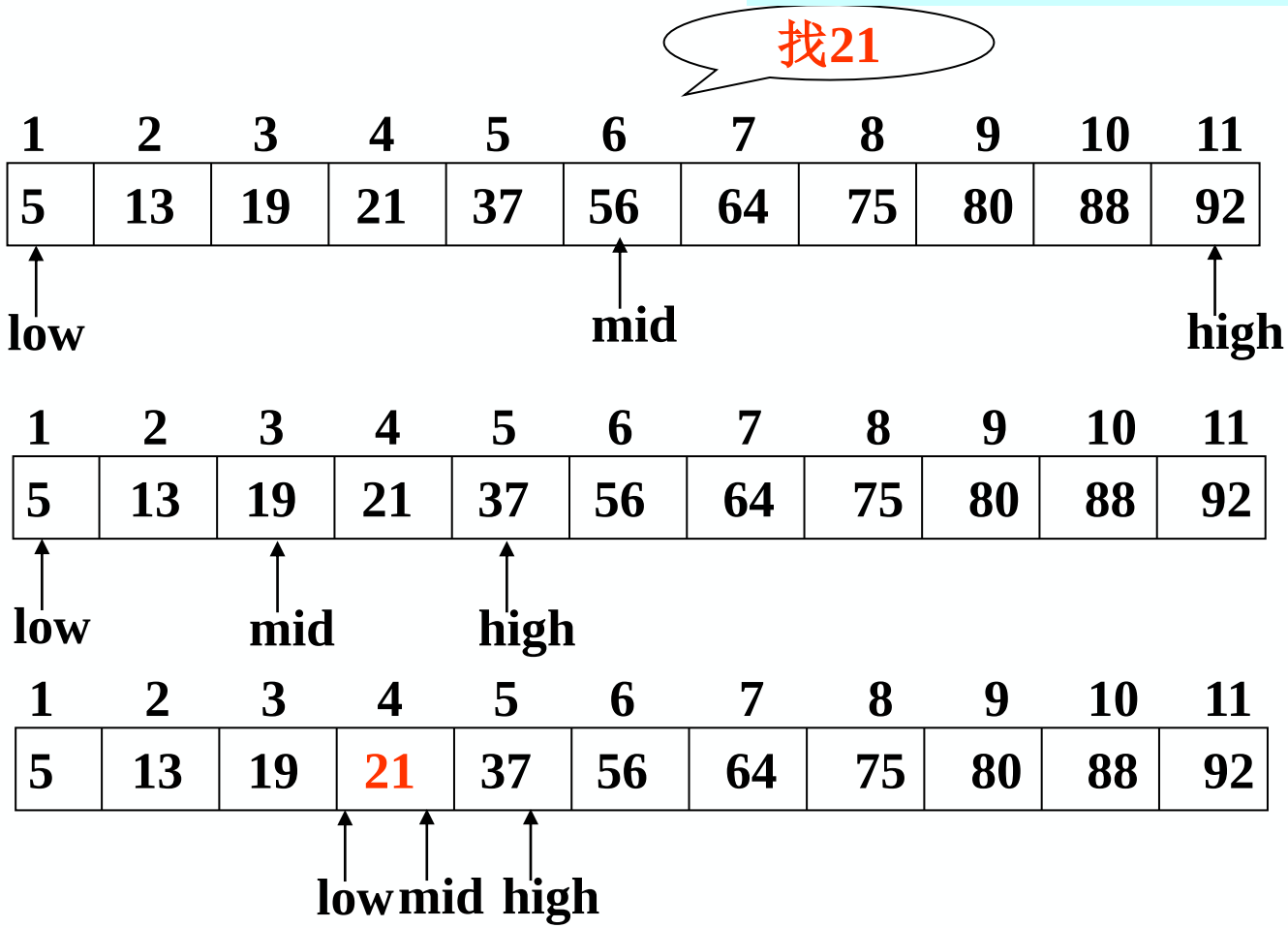
每次查找和中间点元素比较，若查找成功则返回；否则当前查找区间缩小一半，直至查找区间为空查找失败

折半查找

若k==R[mid].key，查找成功

若k<R[mid].key，则high=mid-1

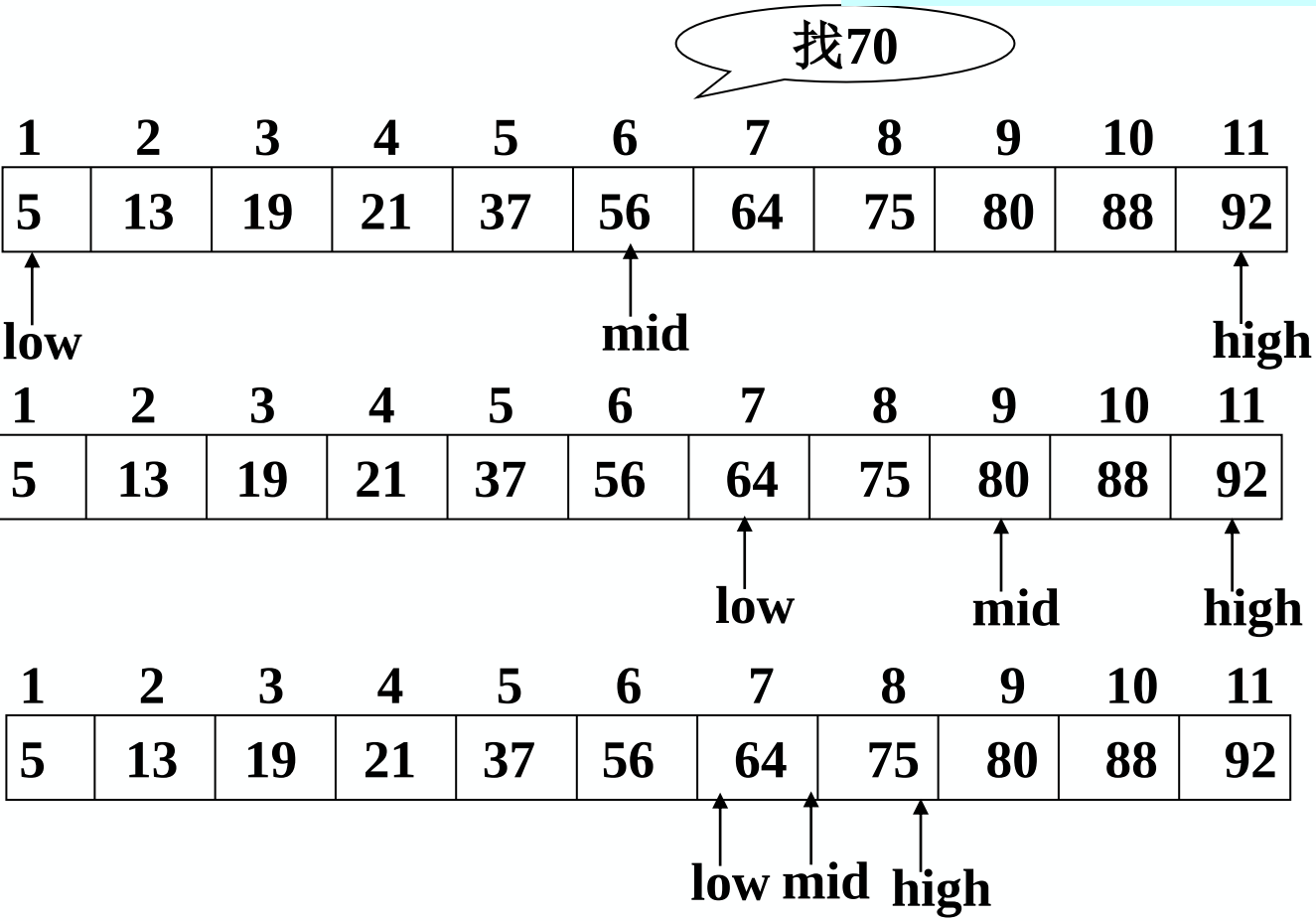
若k>R[mid].key，则low=mid+1



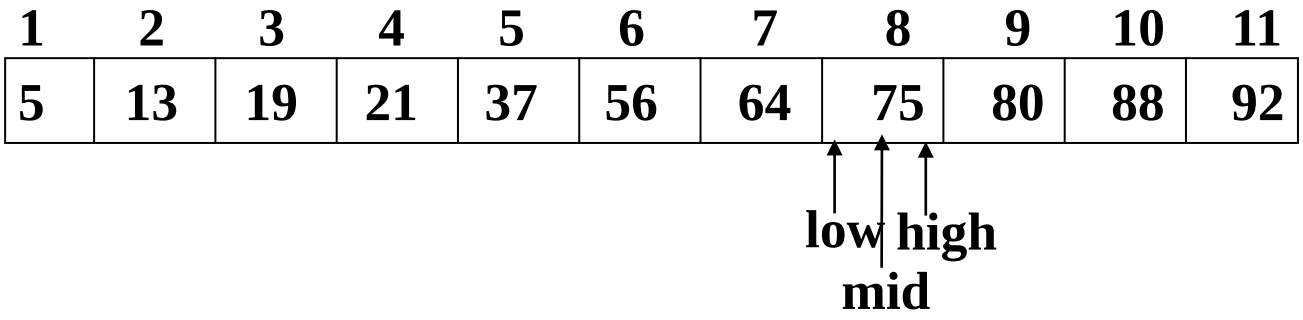
若k==R[mid].key，查找成功

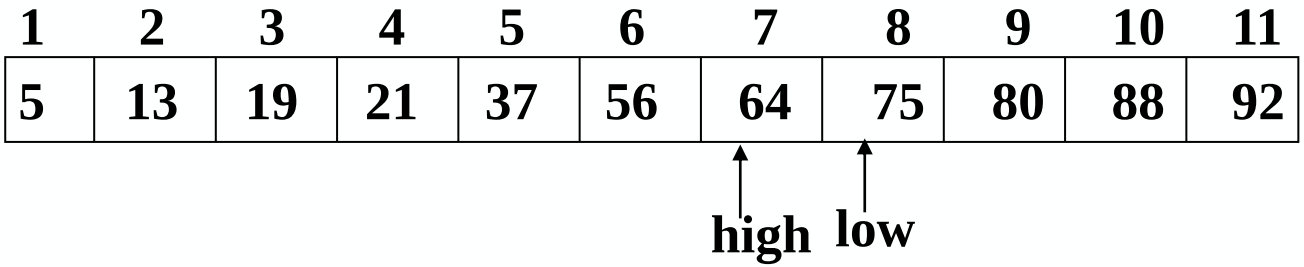
若k<R[mid].key，则high=mid-1

若k>R[mid].key，则low=mid+1



直至low>high时，查找失败





【算法描述】-算法7.3

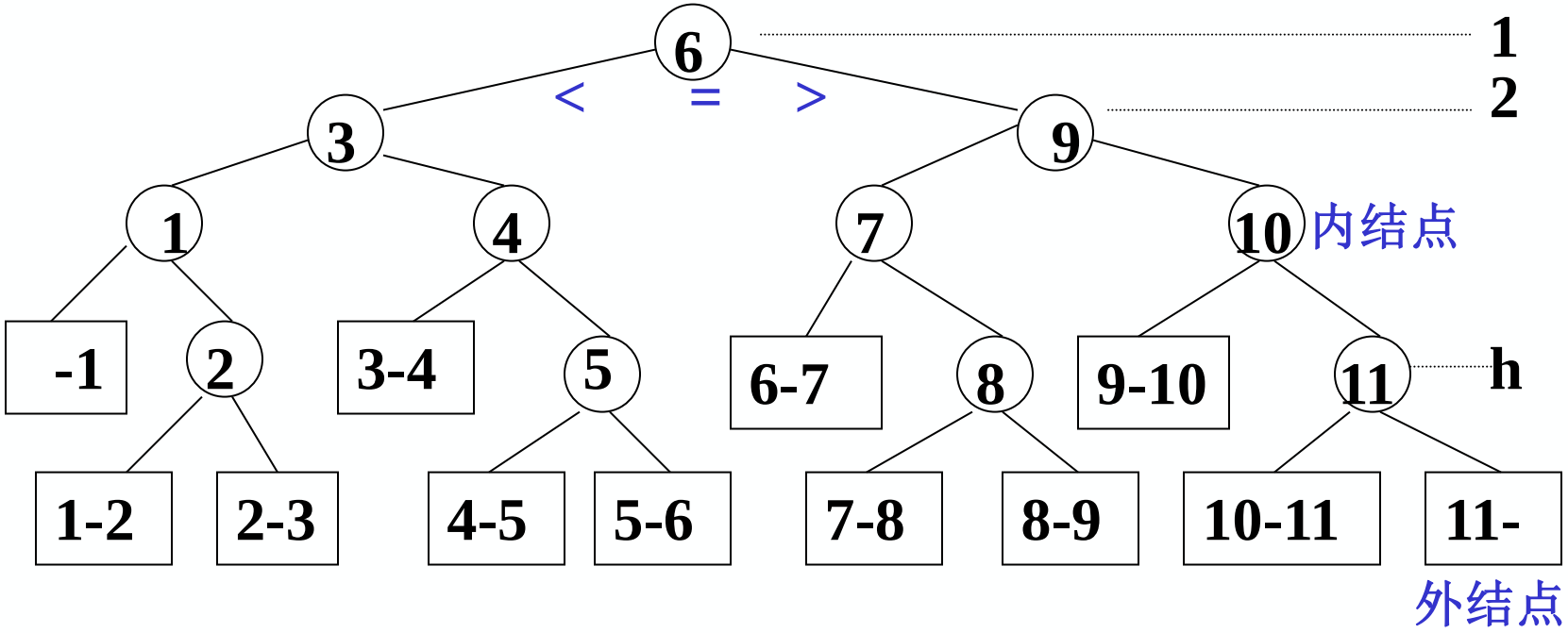
```
int Search_Bin(SSTable ST,KeyType key){
//若找到，则函数值为该元素在表中的位置，否则为0
    low=1;high=ST.length;
    while(low<=high){
        mid=(low+high)/2;
        if(key==ST.R[mid].key) return mid;
        else if(key<ST.R[mid].key) high=mid-1;//前一子表查找
        else low=mid+1;                          //后一子表查找
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

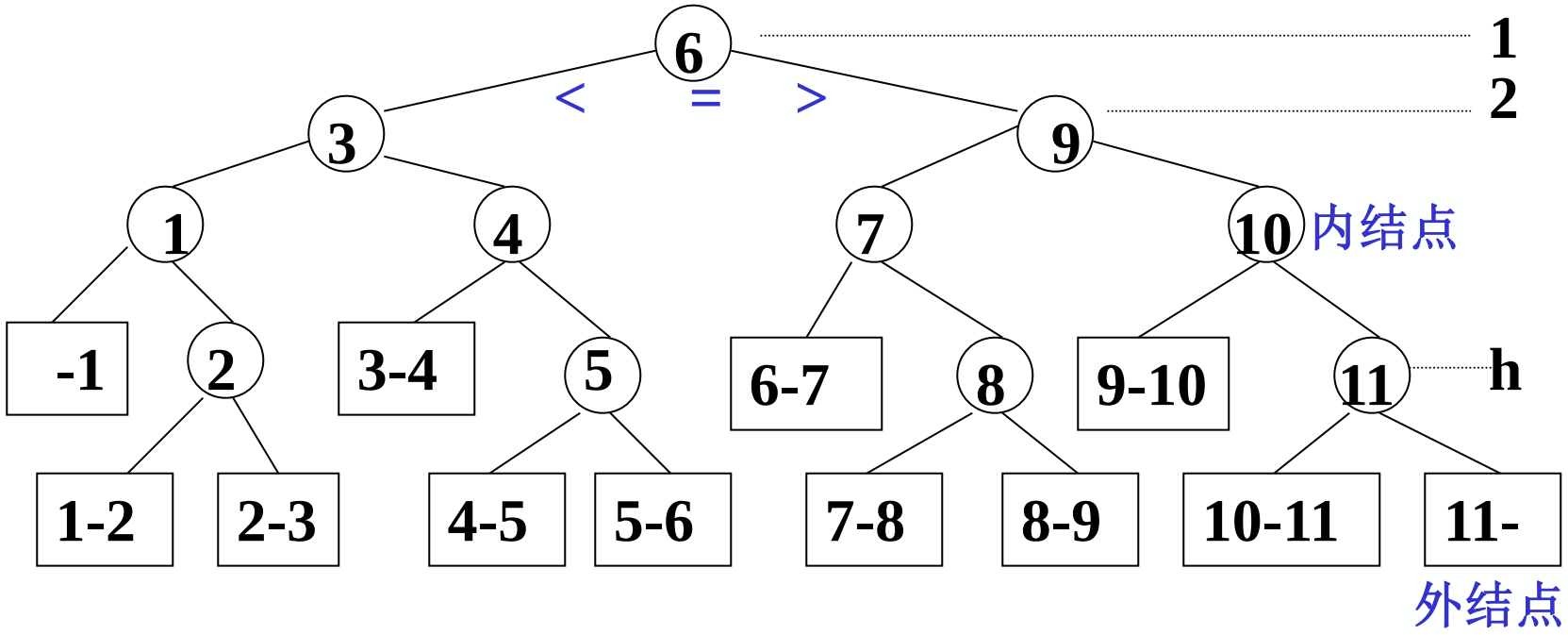
折半查找的性能分析 - 判定树

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

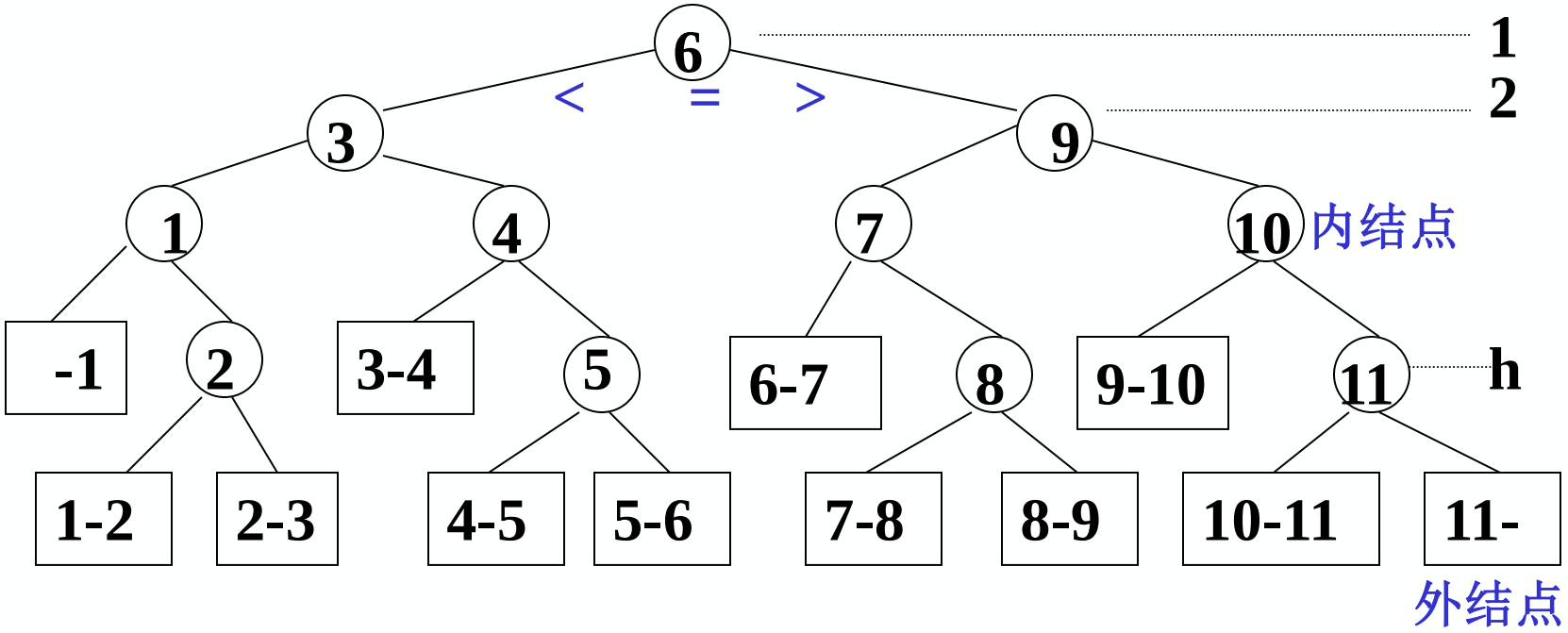


若所有结点的空指针域设置为一个指向一个方形结点的指针，称方形结点为判定树的**外部结点**；对应的，圆形结点为**内部结点**。

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。



$$ASL = 1/11 \times (1\times1 + 2\times2 + 4\times3 + 4\times4) = 33/11 = 3$$



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

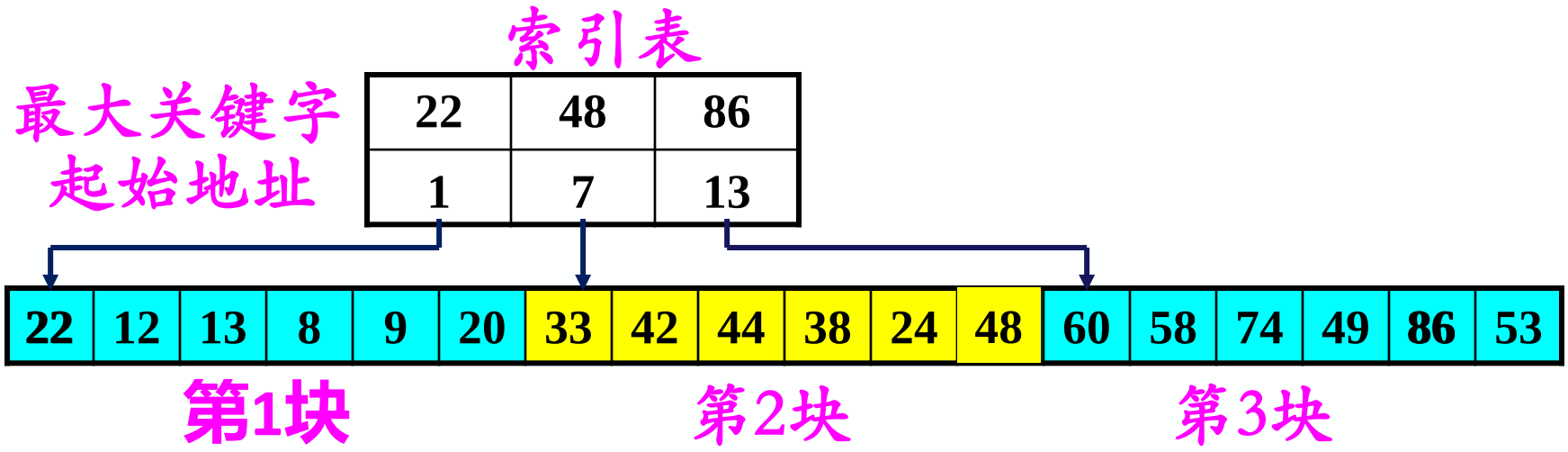
- 查找过程：每次将待查记录所在区间缩小一半，比顺序查找效率高,时间复杂度$O(\log_2 n)$

- 适用条件：采用顺序存储结构的有序表，不宜用于链式结构

分块查找（块间有序，块内无序）

分块有序，即分成若干子表，要求每个子表中的数值都比后一块中数值小（但子表内部未必有序）。

然后将各子表中的最大关键字构成一个索引表，表中还要包含每个子表的起始地址（即头指针）。



分块查找过程

① 对索引表使用折半查找法（因为索引表是有序表）；

② 确定了待查关键字所在的子表后，在子表内采用顺序查找法（因为各子表内部是无序表）；

分块查找性能分析

查找效率：$ASL=L_b+L_w$

对索引表查找的ASL

对块内查找的ASL

$$ASL_{bs} \cong \log_2(\frac{n}{s}+1)+\frac{s}{2} \qquad (\log_2 n \le ASL_{bs} \le \frac{n+1}{2})$$

s为每块内部的记录个数，n/s即块的数目

例如，当n=9，s=3时，ASL_{bs}=3.5,而折半法为3.1，顺序法为5

分块查找优缺点

优点：插入和删除比较容易，无需进行大量移动。

缺点：要增加一个索引表的存储空间并对初始索引表进行排序运算。

适用情况：如果线性表既要快速查找又经常动态变化，则可采用分块查找。

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

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：

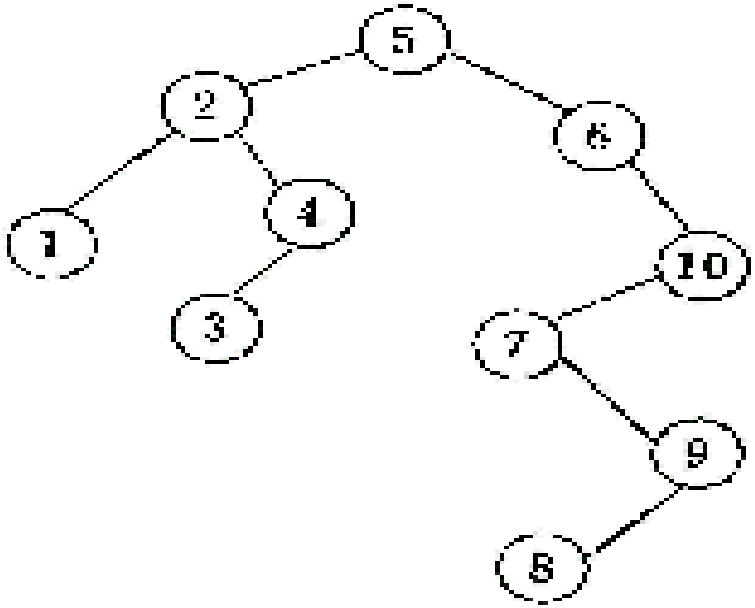
(1)若其左子树非空，则左子树上所有结点的值均小于根结点的值；

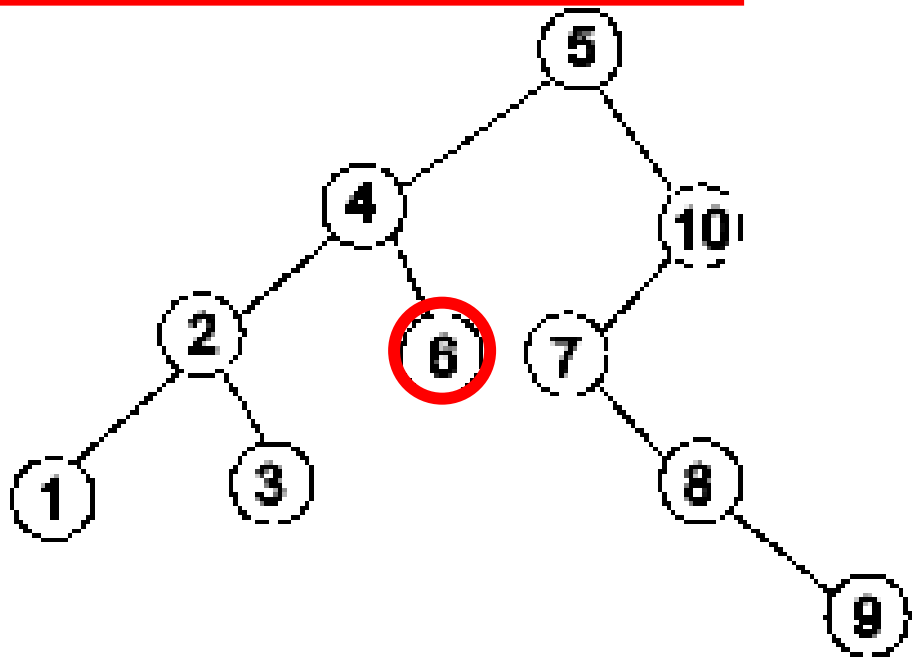
(2)若其右子树非空，则右子树上所有结点的值均大于等于根结点的值；

(3)其左右子树本身又各是一棵二叉排序树

练习

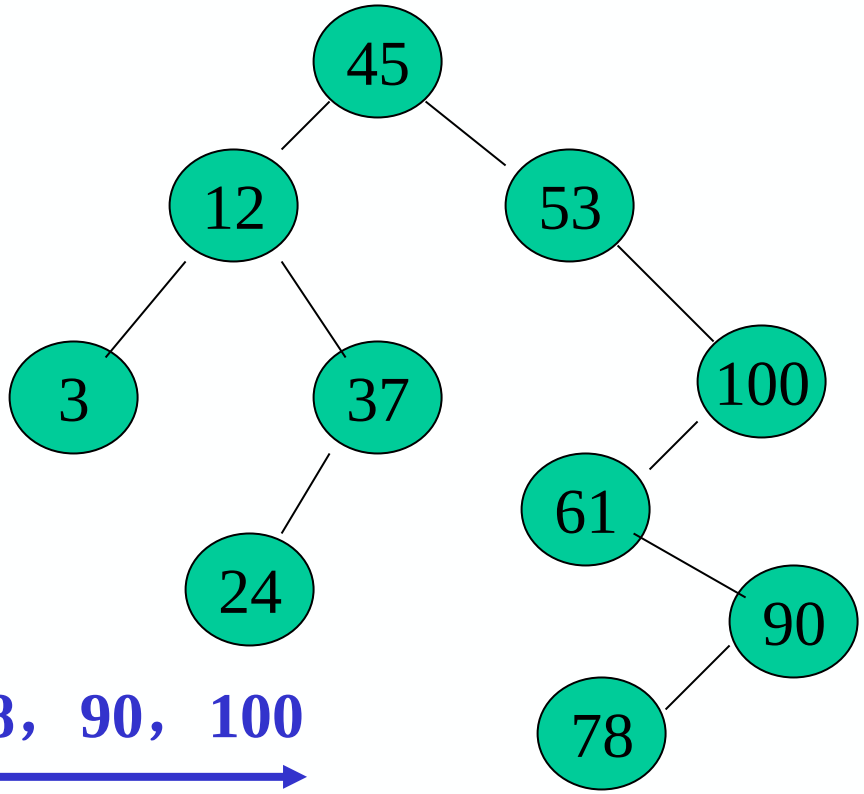
下列图形中，哪个不是二叉排序树？





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

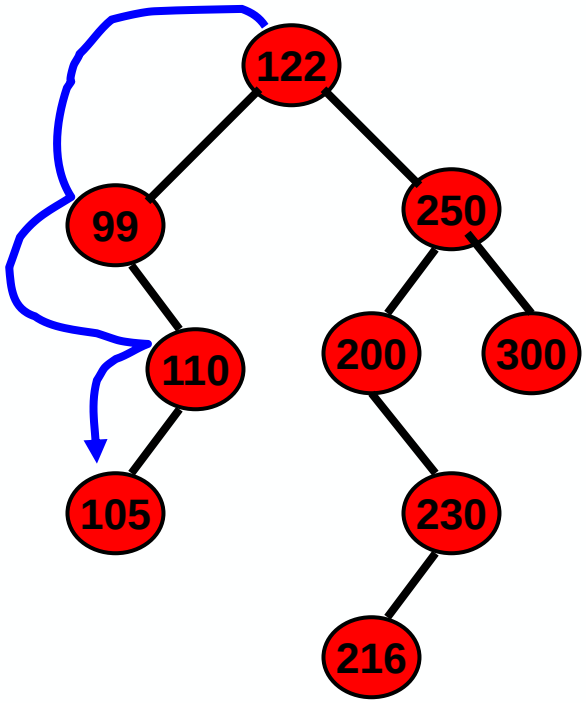
若查找的关键字**等于**根结点，**成功**

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；

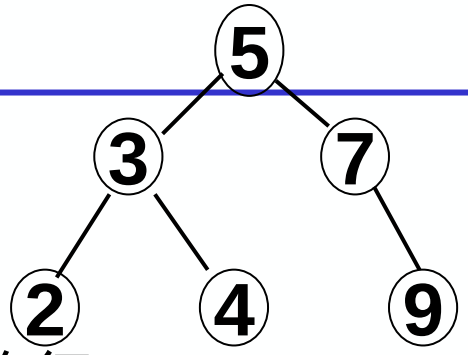
② 若key小于T->data.key，则进一步查找左子树；

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



- **时间分析**

  若成功，则走了一条从根到待查节点的路径

  若失败，则走了一条从根到叶子的路径

  – 上界：O(h)

  – 分析：与树高相关

  ①最坏情况：单支树，ASL=(n+1)/2, 与顺序查找相同

  ②最好情况：ASL≈lgn，形态与折半查找的判定树相似

  ③平均情况：假定n个keys所形成的n!种排列是等概率的，则可证明由这n!个序列产生的n!棵BST（其中有的形态相同）的平均高度为O(lgn)，故查找时间仍为O(lgn)

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

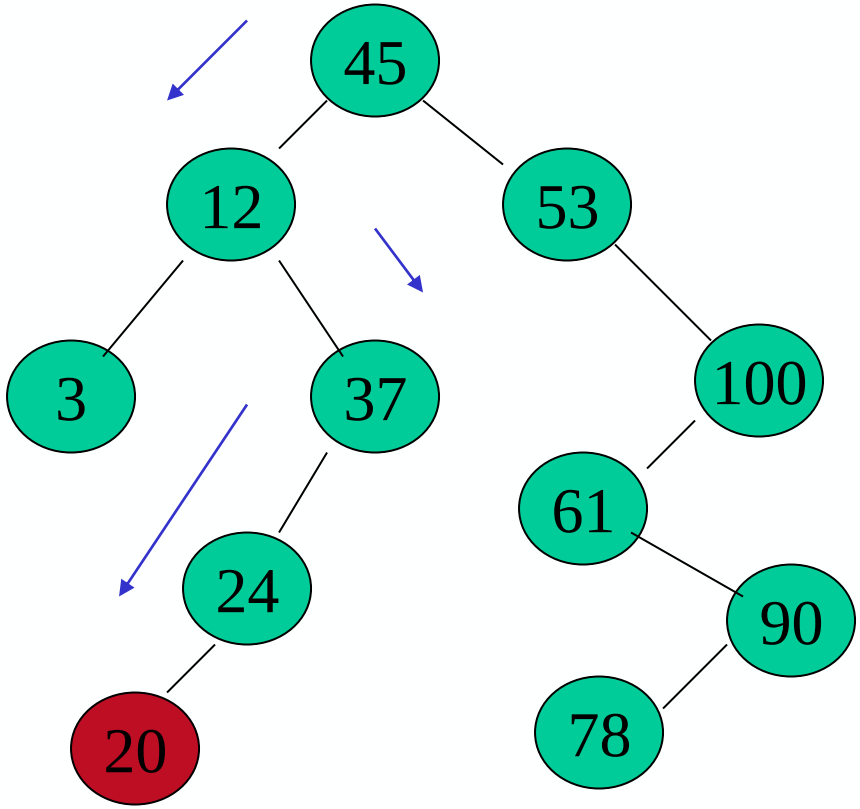
二叉排序树的操作 - 插入

```
void InsertBST(BSTree &T, ElemType e)
{//当二叉排序树T中不存在关键字等于e.key的数据元素时，则插入该元素
    if(!T) {                              //找到插入位置，递归结束
        S=new BSTNode;                    //生成新节点*s
        S->data=e;
        S->lchild=S->rchild=NULL;
        T=S;
    }
    else if(e.key<T->data.key)
        InsertBST(T->lchild,e);           //将*s插入左子树
    else if(e.key>T->data.key)
        InsertBST(T->rchild,e);           //将*s插入右子树
}
```

插入的元素一定在叶结点上

二叉排序树的操作 - 插入

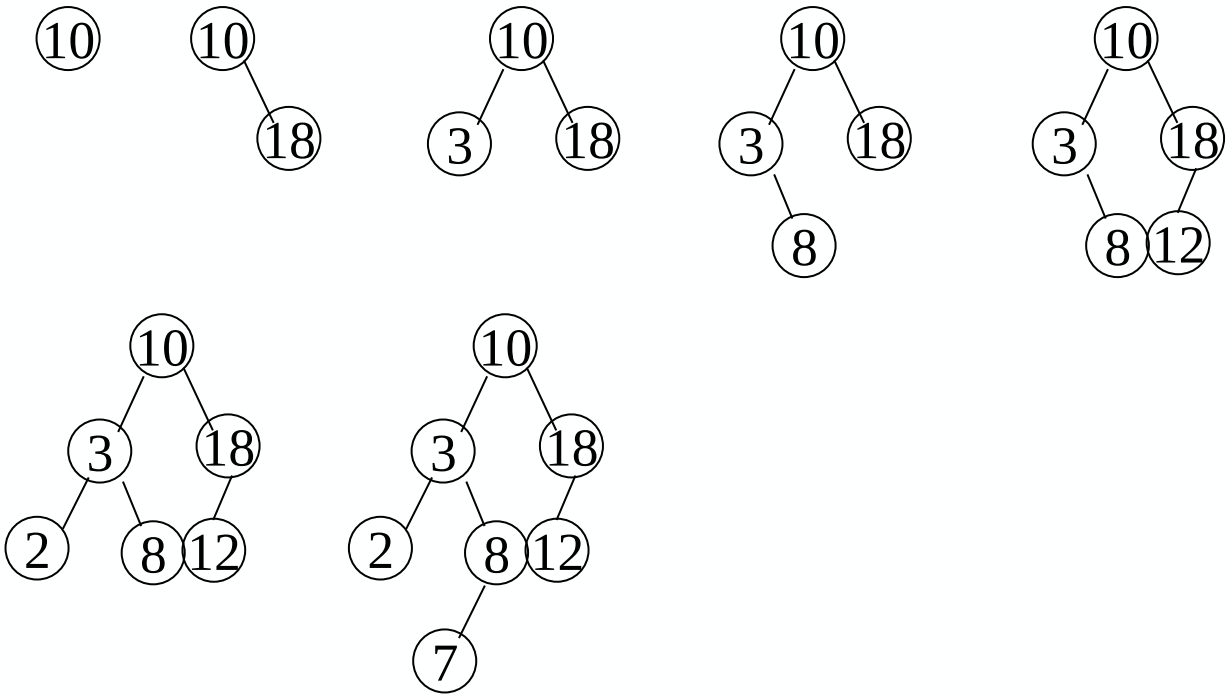
插入结点20



二叉排序树的操作 - 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（具体算法见教材P195）

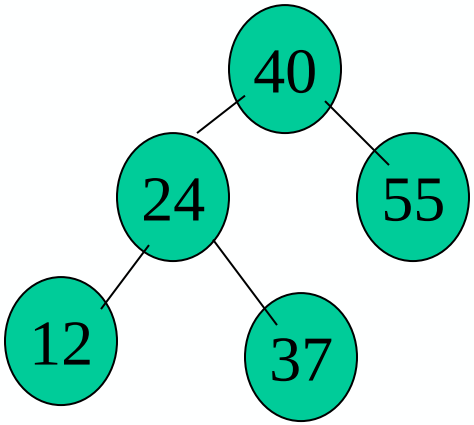
{10， 18， 3， 8， 12， 2， 7}



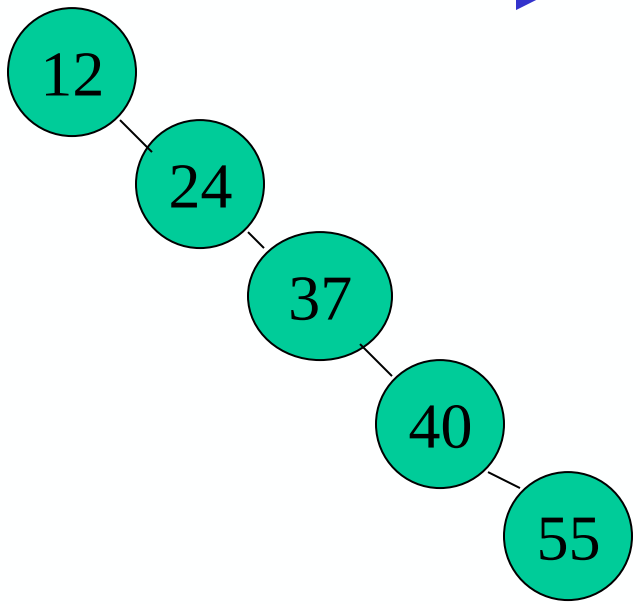
二叉排序树的操作 - 生成

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 - 生成

✓ 一般情况

不同的输入实例（数据集不同、或排列不同），生成的树的形态一般不同。对n个结点的同一数据集，可生成n！棵BST。

✓ 例外情况

但有时不同的实例可能生成相同的BST，例如：（2，3，7，8，5，4）和（2，3，7，5，8，4）可构造同一棵BST。

✓ 排序树名称的由来

因为BST的中序序列有序，所以对任意关键字序列，构造BST的过程，实际上是对其排序。

生成n个结点的BST的平均时间是O(nlgn)，但它约为堆排序的2 - 3倍，因此它并不适合排序。

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

BST的删除

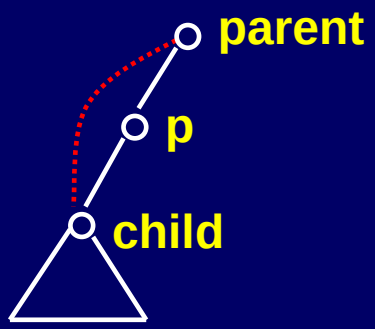
保证删一结点不能将以该结点为根的子树删去，且仍须满足BST性质。即：删一结点相当于删有序序列中的一个结点。

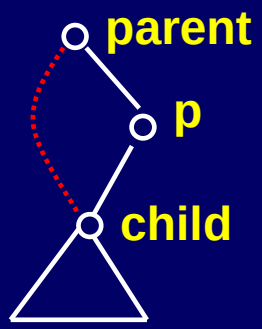
基本思想

- 查找待删结点*p，令parent指向其双亲（初值NULL）；若找不到则返回，否则进入下一步。
- 在删除*p时处理其子树的连接问题，同时保持BST性质不变。

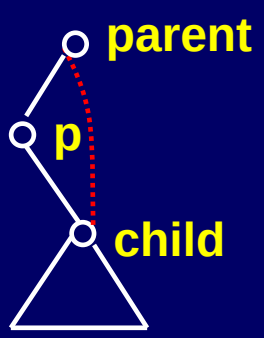
case1:*p是叶子，无须连接子树，只需将*parent中指向*p的指针置空

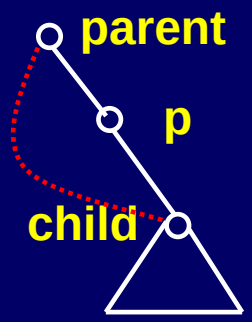
case2:*p只有1个孩子，只须连接唯一的1棵子树，故可令此孩子取代*p与其双亲连接（4种状态）





*p只有左孩子





*p只有右孩子

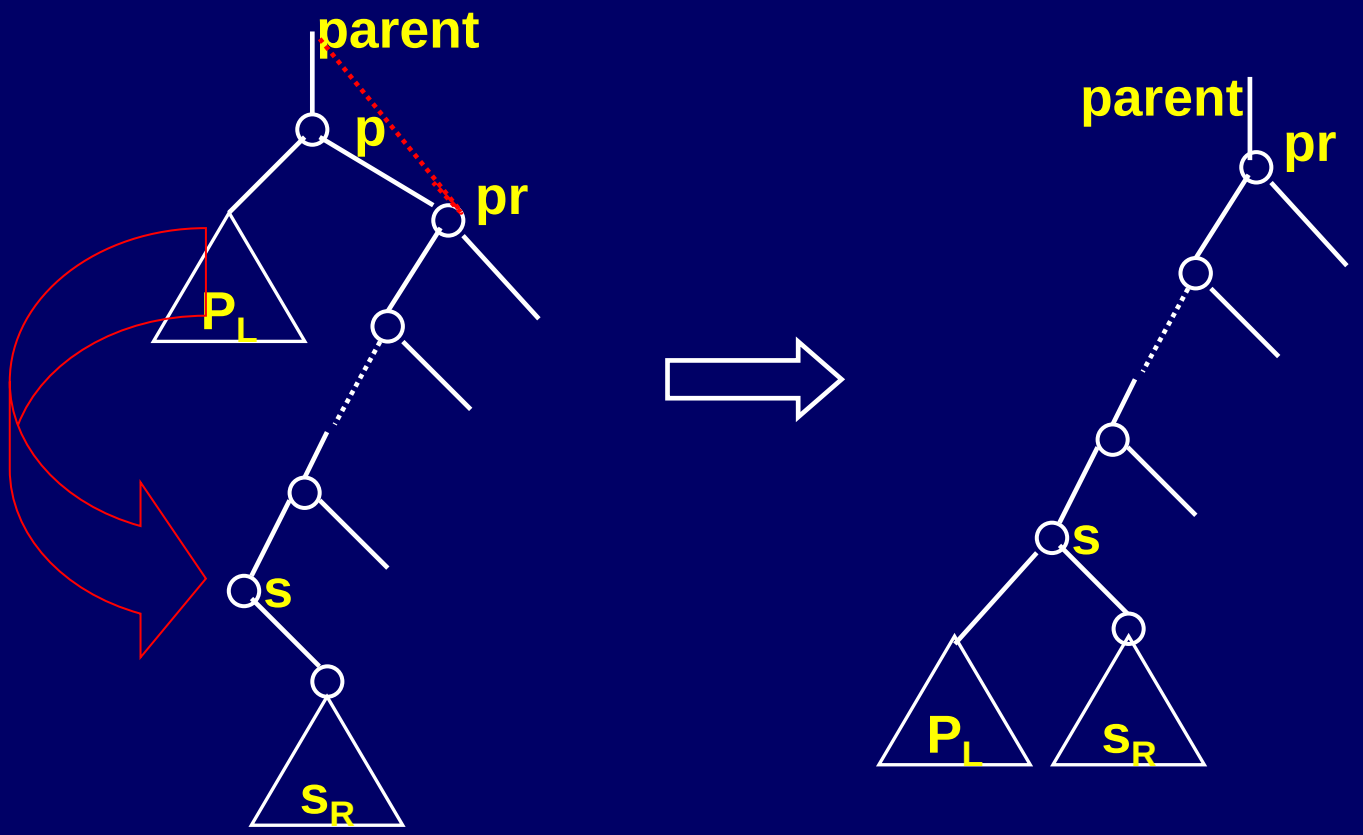
BST的删除

基本思想

case3: *p有2个孩子，有2种处理方式：

①找到*p 的中序后继(或前驱)*s，用*p的右(或左)子树取代*p与其双亲*parent连接；而*p的左(或右)子树P_L则作为*s的左(或右)子树与*s连接。

缺点：树高可能增大。

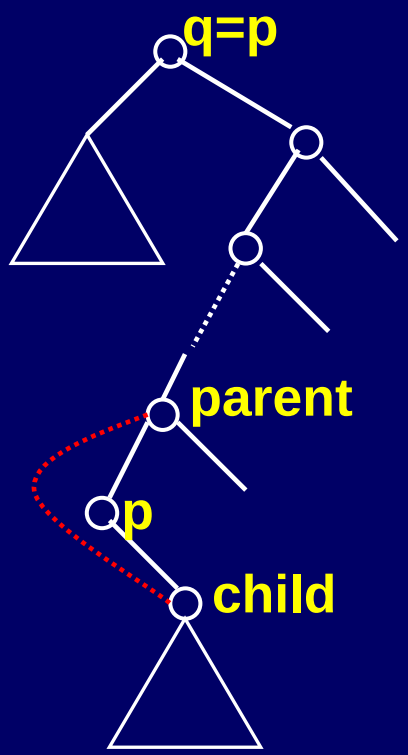


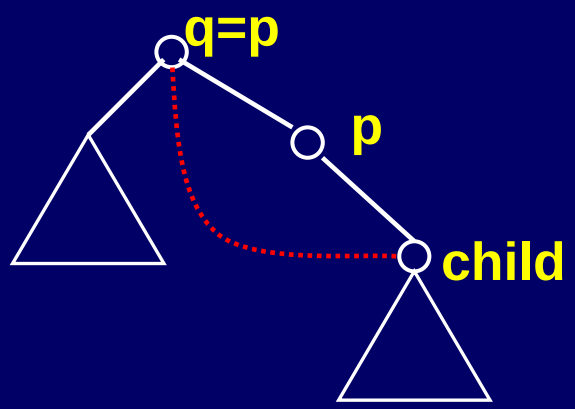
BST的删除

- 基本思想

②令q=p，找*q的中序后继*p，并令parent指向*p的双亲，将*p的右子树取代*p与其双亲*parent连接。将*p的内容copy到*q中，相当于删去了*q，将删*q的操作转换为删*p的操作。对称地，也可找*q的中序前驱

因为*p最多只有1棵非空的子树，属于case2。实际上case1也是case2的特例。因此，case3采用该方式时，3种情况可以统一处理为case2。





*q的中序后继就是其右孩子

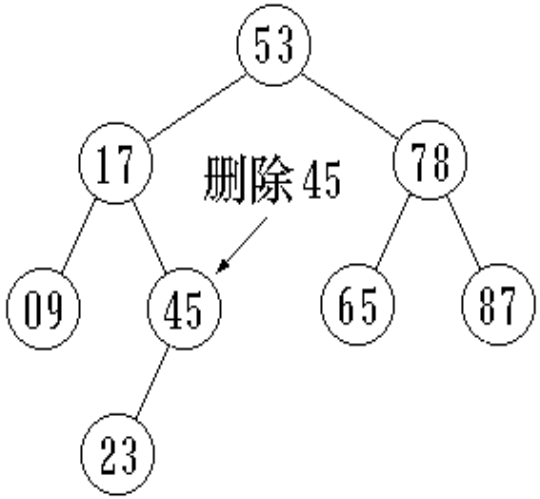
BST的删除

- 算法

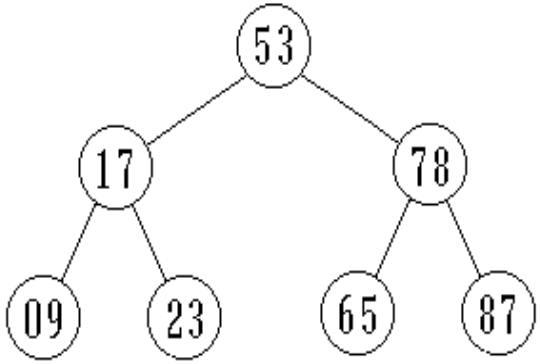
```
void  DelBSTNode( BSTree *T,  KeyType key) {  //*T是根
    BSTree *q, *child, *parent=NULL, *p=*T;
    while ( p ) { //找待删结点
        if  ( p->key ==key ) break;  //已找到
        parent=p; //循环不变式是*parent为*p的双亲
        p＝( key < p->key ) ? p->lchild; p->rchild;
    }
    if ( !p ) return;  //找不到被删结点，返回
    q=p; //q记住被删结点*p
    if (q->lchild && q->rchild ) //case3，找*q的中序后继
        for(parent=q,p=q->rchild; p->lchild; parent=p, p=p->lchild);
```

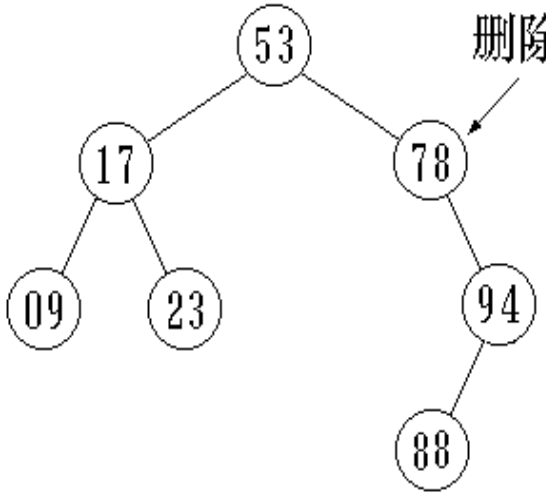
BST的删除

```
//现在3种情况已统一到情况2，被删结点*p最多只有1个非空的孩子
    child=(p->lchild)？ p->lchild; p->rchild; //case1时child空，否则非空
    If ( !parent ) //*p的双亲为空，说明 *p是根，即删根结点
        *T=child; //若是情况1，则树为空；否则*child取代*p成为根
    else { //*p非根，它的孩子取代它与*p的双亲连接，即删 *p
        if ( p==parent->lchild )  parent->lchild=child;
        else   parent->rchild=child;
        if ( p != q ) //情况3，将*p的数据copy到*q中
            q->key=p->key; //若有其他数据亦需copy
    }
    free( p ); //删除*p
} //时间O(h)
```

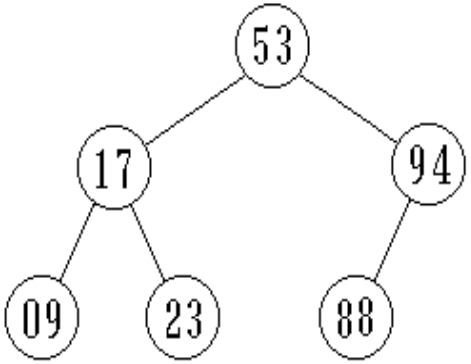


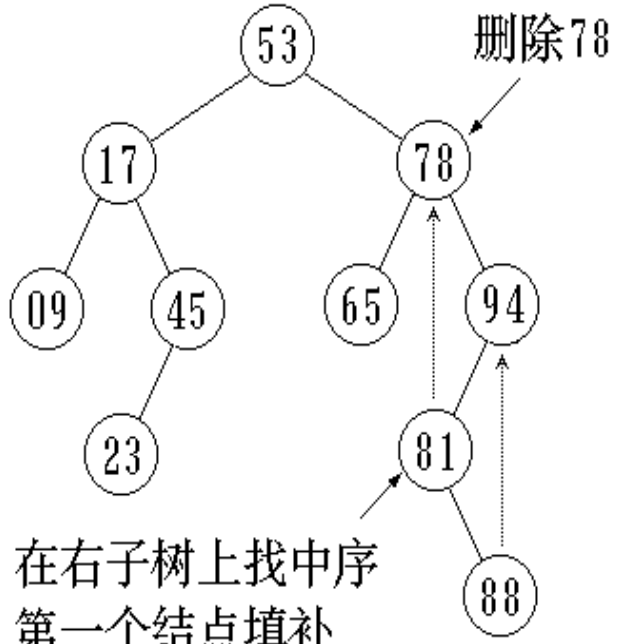
缺右子树用左子女填补



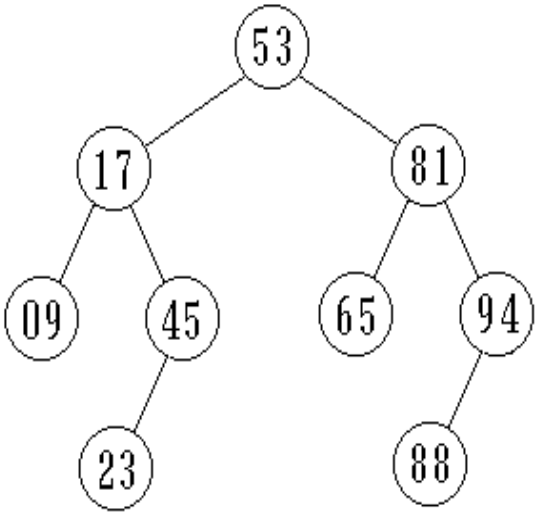


缺左子树用右子女填补

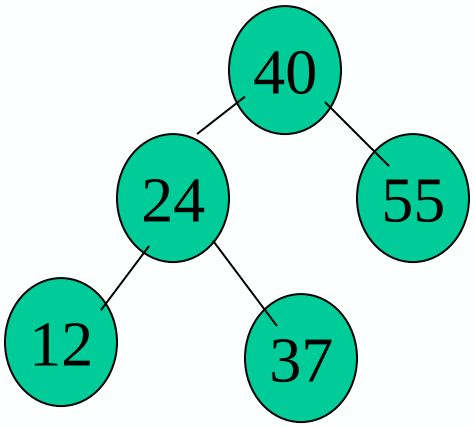


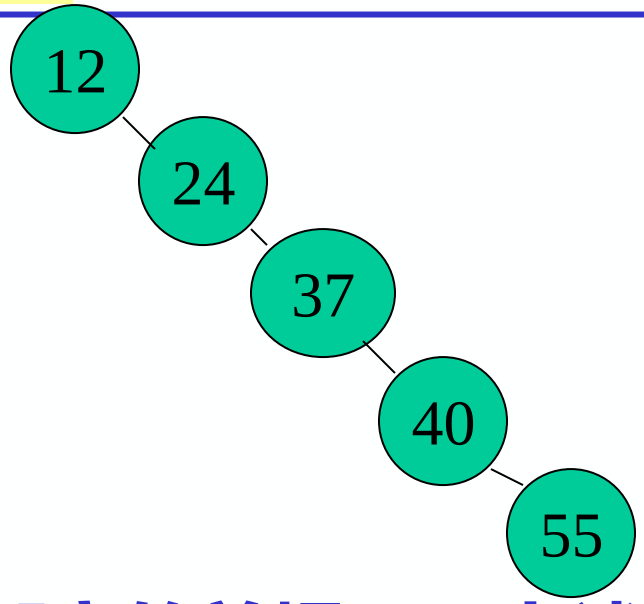


在右子树上找中序
第一个结点填补



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：

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

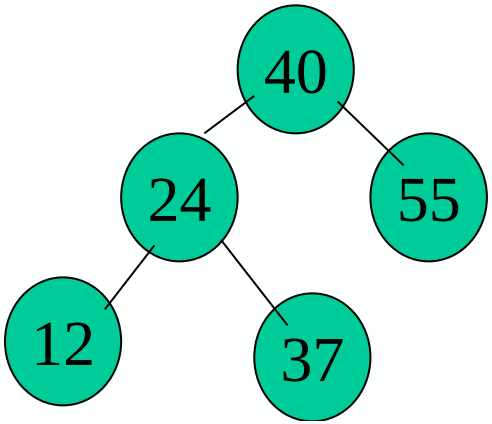
$$\sum_{i=1}^{n} p_i c_i = (1+2+3+4+5)/5 = 3(\text{右图})$$

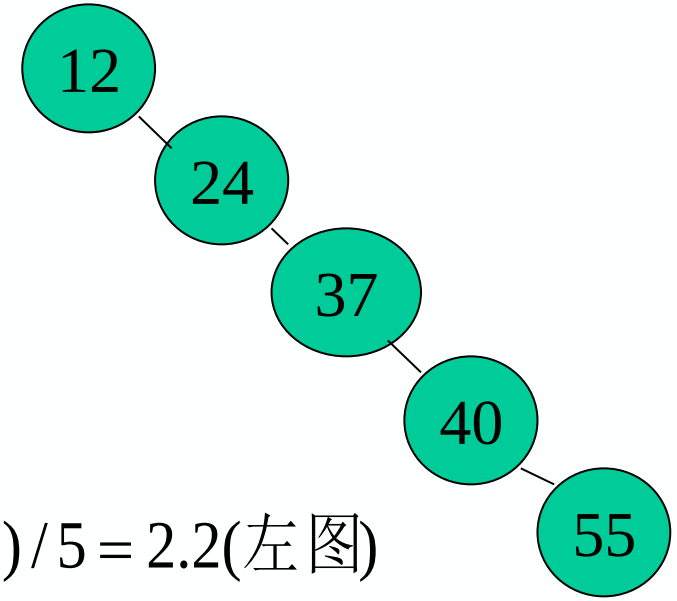
查找的性能分析

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

平衡二叉排序树(Balanced Binary Tree或Height-Balanced Tree)是在1962年由Adelson-Velskii和Landis提出的，又称AVL树。

平衡二叉树的定义

平衡二叉树或者是空树，或者是满足下列性质的二叉树。

⑴：左子树和右子树深度之差的绝对值不大于1；

⑵：左子树和右子树也都是平衡二叉树。

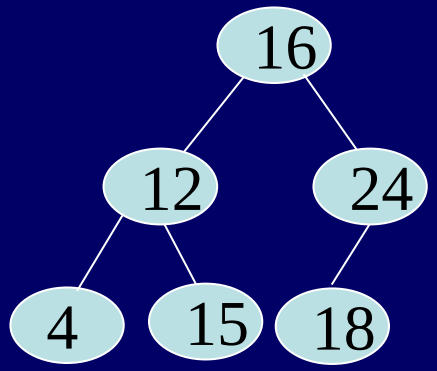
平衡因子(Balance Factor)：二叉树上结点的左子树的深度减去其右子树深度称为该结点的平衡因子。

因此，平衡二叉树上每个结点的平衡因子只可能是-1、0和1，否则，只要有一个结点的平衡因子的绝对值大于1， 该二叉树就不是平衡二叉树。

如果一棵二叉树既是二叉排序树又是平衡二叉树，称为平衡二叉排序树(Balanced Binary Sort Tree)。

结点类型定义如下：

```
typedef  struct  BNode
  {  KeyType  key ;    /*  关键字域  */
     int  Bfactor ;    /*  平衡因子域  */
     …      /*  其它数据域  */
     struct  BNode  *Lchild , *Rchild ;
  }BSTNode ;
```



平衡二叉树

在平衡二叉排序树上执行查找的过程与二叉排序树上的查找过程完全一样，则在AVL树上执行查找时，和给定的K值比较的次数不超过树的深度。

设深度为h的平衡二叉排序树所具有的最少结点数为N_h，则由平衡二叉排序树的性质知：

$N_0=0$，$N_1=1$，$N_2=2$，…，$N_h=N_{h-1}+N_{h-2}+1$

该关系和Fibonacci数列相似。根据归纳法可证明，当$h\geq 0$时，$N_h=F_{h+2}-1$，…而

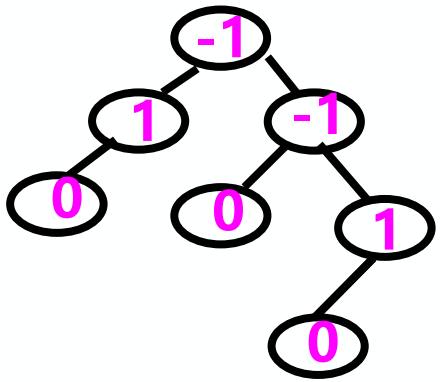
$$F_h\approx\frac{\Phi^h}{\sqrt{5}}\quad 其中\ \Phi=\frac{1+\sqrt{5}}{2}\qquad 则N_h\approx\frac{\Phi^{h+2}}{\sqrt{5}}-1$$

这样，含有n个结点的平衡二叉排序树的最大深度为

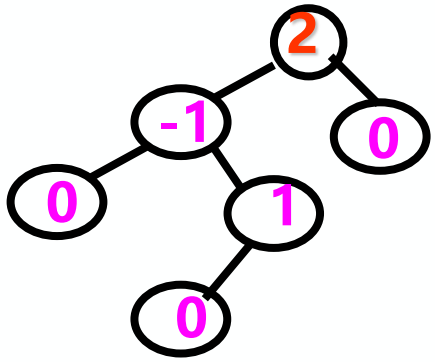
$$h\approx\log_{\Phi}(\sqrt{5}\times(n+1))-2$$

则在平衡二叉排序树上进行查找的平均查找长度和$\log_2 n$是一个数量级的，平均时间复杂度为$O(\log_2 n)$。

练习：判断下列二叉树是否AVL树?



(a) 平衡树



(b) 不平衡树

平衡化旋转

一般的二叉排序树是不平衡的，若能通过某种方法使其既保持有序性，又具有平衡性，就找到了构造平衡二叉排序树的方法，该方法称为平衡化旋转。

在对AVL树进行插入或删除一个结点后，通常会影响到从根结点到插入（或删除）结点的路径上的某些结点，这些结点的子树可能发生变化。以插入结点为例，影响有以下几种可能性

- 以某些结点为根的子树的深度发生了变化；
- 某些结点的平衡因子发生了变化；
- 某些结点失去平衡。

沿着插入结点上行到根结点就能找到某些结点，这些结点的平衡因子和子树深度都会发生变化，这样的结点称为失衡结点。

1 LL型平衡化旋转

(1) 失衡原因

在结点a的左孩子的左子树上进行插入，插入使结点a失去平衡。a插入前的平衡因子是1，插入后的平衡因子是2。设b是a的左孩子，b在插入前的平衡因子只能是0，插入后的平衡因子是1(否则b就是失衡结点或者a不会成为失衡节点)。

(2) 平衡化旋转方法

通过顺时针旋转操作实现。

用b取代a的位置，a成为b的右子树的根结点，b原来的右子树作为a的左子树。

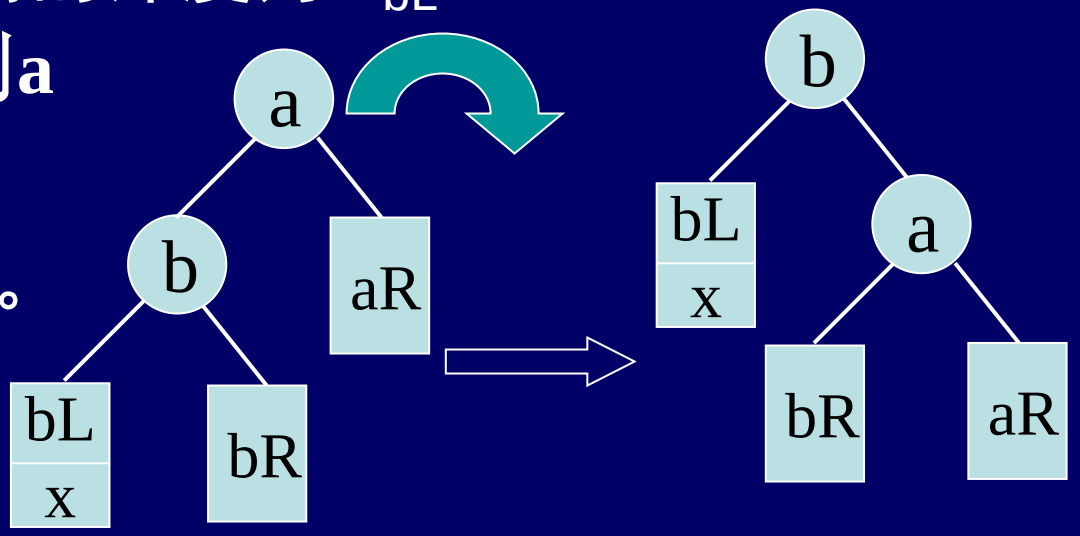
(3) 插入后各结点的平衡因子分析

① 旋转前的平衡因子

设插入后b的左子树的深度为H_{bL}，则其右子树的深度为H_{bL}-1； a的左子树的深度为H_{bL}+1。

a的平衡因子为2，则a的右子树的深度为：

$$H_{aR}=H_{bL}+1-2=H_{bL}-1。$$



LL型平衡化旋转示意图

② 旋转后的平衡因子

a的右子树没有变，而左子树是b的右子树，则平衡因子是：$H_{aL}- H_{aR}=(H_{bL}-1)-(H_{bL}-1)=0$

即a是平衡的，以a为根的子树的深度是H_{bL}。

b的左子树没有变化，右子树是以a为根的子树，则平衡因子是： $H_{bL}-H_{bL}=0$

即b也是平衡的，以b为根的子树的深度是$H_{bL}+1$，与插入前a的子树的深度相同，则该子树的上层各结点的平衡因子没有变化，即整棵树旋转后是平衡的。

(4) 旋转算法

```
BBSTNode * LL_rotate(BBSTNode *a)
   {  BBSTNode *b ;
      b=a->Lchild ; a->Lchild=b->Rchild ;
      b->Rchild=a ;
      a->Bfactor=b->Bfactor=0 ;
      return b;
   }
```

2 LR型平衡化旋转

(1) 失衡原因

在结点a的左孩子的右子树上进行插入，插入使结点a失去平衡。a插入前的平衡因子是1，插入后a的平衡因子是2。设b是a的左孩子，c为b的右孩子， b在插入前的平衡因子只能是0，插入后的平衡因子是-1；c在插入前的平衡因子只能是0，否则，c就是失衡结点。

(2) 插入后结点c的平衡因子的变化分析

①插入后c的平衡因子是1：即在c的左子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}-1；b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2。

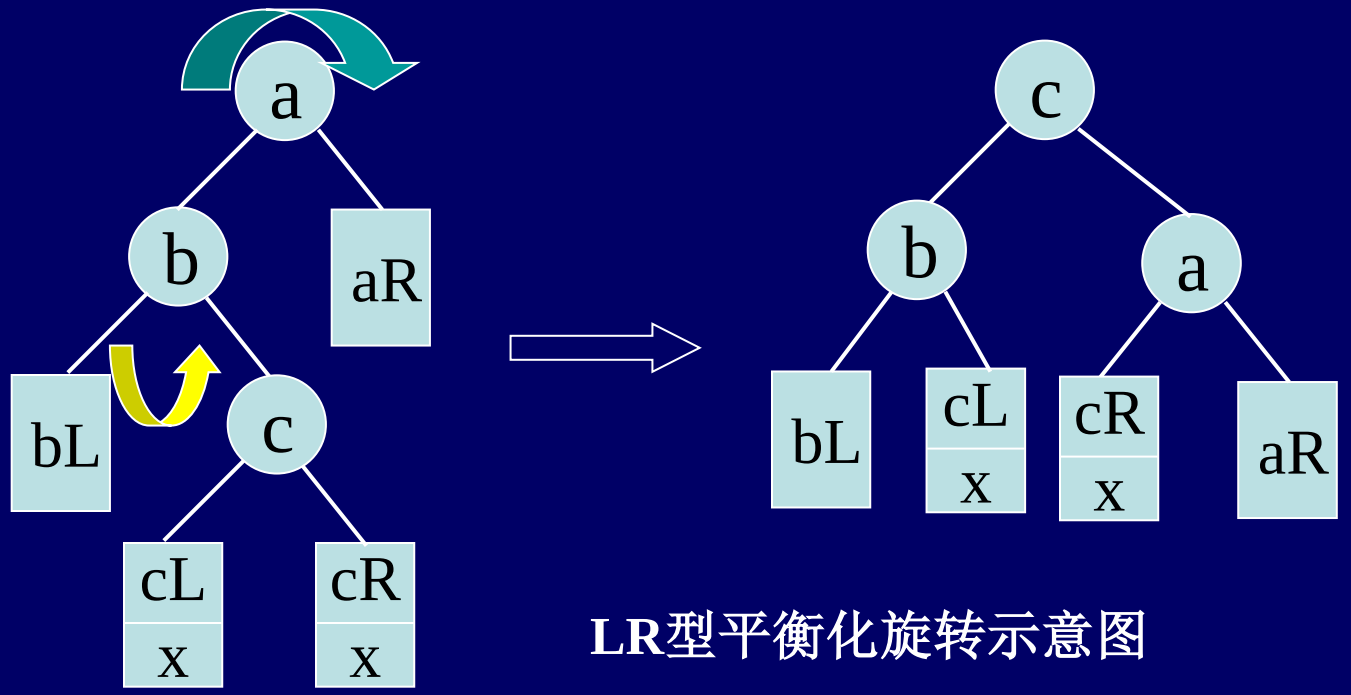
因插入后a的平衡因子是2 ，则a的右子树的深度是H_{cL}。

② 插入后c的平衡因子是0：c本身是插入结点。设c的左子树的深度为H_{cL}，则右子树的深度也是H_{cL}；因b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2；插入后a的平衡因子是2 ，则a的右子树的深度是H_{cL}。

③ 插入后c的平衡因子是-1：即在c的右子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}+1 ，以c为根的子树的深度是H_{cL}+2；因b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}+1，以b为根的子树的深度是H_{cL}+3；则a的右子树的深度是H_{cL}+1。

(3) 平衡化旋转方法

先以b进行一次逆时针旋转(将以b为根的子树旋转为以c为根)，再以a进行一次顺时针旋转，如图所示。将整棵子树旋转为以c为根，b是c的左子树，a是c的右子树；c的右子树移到a的左子树位置， c的左子树移到b的右子树位置。



LR型平衡化旋转示意图

(4) 旋转后各结点(a,b,c)平衡因子分析

① 旋转前 (插入后)c的平衡因子是1：

a的左子树深度为H_{cL}-1，其右子树没有变化，深度是H_{cL}，则a的平衡因子是-1；b的左子树没有变化，深度为H_{cL}，右子树是c旋转前的左子树，深度为H_{cL}，则b的平衡因子是0； c的左、右子树分别是以b和a为根的子树，则c的平衡因子是0 。

② 旋转前 (插入后)c的平衡因子是0：

旋转后a，b，c的平衡因子都是0 。

③ 旋转前 (插入后)c的平衡因子是-1：

旋转后a，b，c的平衡因子分别是0，1，0 。

综上所述，即整棵树旋转后是平衡的。

(5) 旋转算法

```
BBSTNode * LR_rotate(BBSTNode *a)
  {  BBSTNode *b,*c  ;
     b=a->Lchild ; c=b->Rchild ;      /*  初始化  */
     a->Lchild=c->Rchild ;  b->Rchild=c->Lchild ;
     c->Lchild=b ;  c->Rchild=a ;
     if (c->Bfactor==1) {a->Bfactor=-1 ;b->Bfactor=0 ; }
     else if (c->Bfactor==0)  a->Bfactor=b->Bfactor=0 ;
     else {   a->Bfactor=0 ; b->Bfactor=1 ;   }
     c->Bfactor=0; return c;
  }
```

3 RL型平衡化旋转

(1) 失衡原因

在结点a的右孩子的左子树上进行插入，插入使结点a失去平衡，与LR型正好对称。对于结点a，插入前的平衡因子是-1，插入后a的平衡因子是-2。设b是a的右孩子，c为b的左孩子，b在插入前的平衡因子只能是0，插入后的平衡因子是1；同样，c在插入前的平衡因子只能是0，否则，c就是失衡结点。

(2) 插入后结点c的平衡因子的变化分析

① 插入后c的平衡因子是1：在c的左子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}-1。

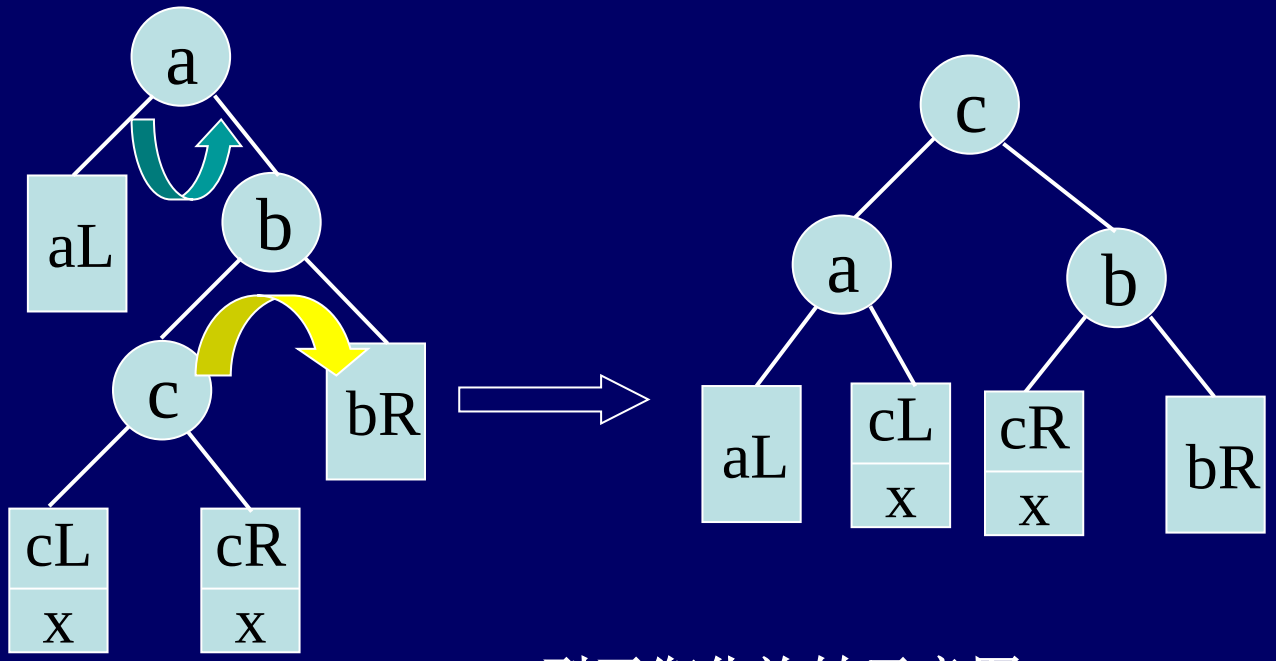
因b插入后的平衡因子是1，则其右子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2；因插入后a的平衡因子是-2，则a的左子树的深度是H_{cL}。

② 插入后c的平衡因子是0：c本身是插入结点。设c的左子树的深度为H_{cL}，则右子树的深度也是H_{cL}；因b插入后的平衡因子是1，则b的右子树的深度为H_{cL}，以b为根的子树的深度是H_{cL}+2；因插入后a的平衡因子是-2，则a的左子树的深度是H_{cL}。

③ 插入后c的平衡因子是-1：在c的右子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为H_{cL}+1，以c为根的子树的深度是H_{cL}+2；因b插入后的平衡因子是1，则b的右子树的深度为H_{cL}+1，以b为根的子树的深度是H_{cL}+3；则a的右子树的深度是H_{cL}+1。

(3) 平衡化旋转方法

先以b进行一次顺时针旋转，再以a进行一次逆时针旋转，如图所示。即将整棵子树(以a为根)旋转为以c为根，a是c的左子树，b是c的右子树；c的右子树移到b的左子树位置，c的左子树移到a的右子树位置。



RL型平衡化旋转示意图

(4) 旋转后各结点(a, b, c)的平衡因子分析

① 旋转前 (插入后)c的平衡因子是1：

a的左子树没有变化，深度是H_{cL}，右子树是c旋转前的左子树，深度为H_{cL}，则a的平衡因子是0；b的右子树没有变化，深度为H_{cL}，左子树是c旋转前的右子树，深度为H_{cL}-1，则b的平衡因子是-1； c的左、右子树分别是以a 和b为根的子树，则c的平衡因子是0 。

② 旋转前 (插入后)c的平衡因子是0：

旋转后a，b，c的平衡因子都是0 。

③ 旋转前 (插入后)c的平衡因子是-1：

旋转后a，b，c的平衡因子分别是1，0，0 。

综上所述，即整棵树旋转后是平衡的。

(5) 旋转算法

```
BBSTNode * RL_rotate(BBSTNode *a)
  {  BBSTNode *b,*c  ;
     b=a->Rchild ; c=b->Lchild ;      /*  初始化  */
     a->Rchild=c->Lchild ;  b->Lchild=c->Rchild ;
     c->Lchild=a ;  c->Rchild=b ;
     if (c->Bfactor==1){ a->Bfactor=0 ; b->Bfactor=-1 ; }
     else if (c->Bfactor==0)  a->Bfactor=b->Bfactor=0 ;
     else {   a->Bfactor=1 ;b->Bfactor=0 ;   }
     c->Bfactor=0; return c;
  }
```

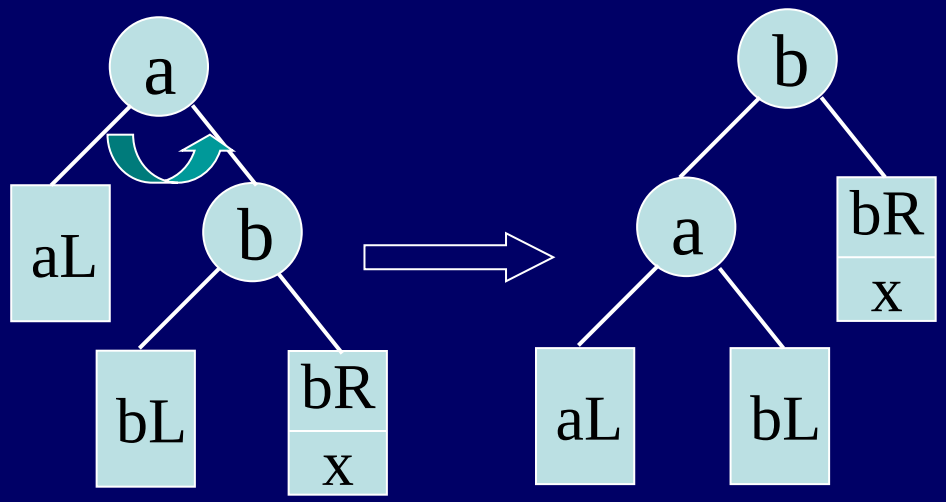
4 RR型平衡化旋转

(1) 失衡原因

在结点a的右孩子的右子树上进行插入，插入使结点a失去平衡。要进行一次逆时针旋转，和LL型平衡化旋转正好对称。

(2) 平衡化旋转方法

设b是a的右孩子，通过逆时针旋转实现，如图所示。用b取代a的位置，a作为b的左子树的根结点，b原来的左子树作为a的右子树。



RR型平衡化旋转示意图

(3) 旋转算法

```
BBSTNode  *RR_rotate(BBSTNode *a)
   {  BBSTNode *b ;
      b=a->Rchild ; a->Rchild=b->Lchild ; b->Lchild=a ;
      a->Bfactor=b->Bfactor=0 ; return b ;
   }
```

对于上述四种平衡化旋转，其正确性容易由“遍历所得中序序列不变”来证明。并且，无论是哪种情况，平衡化旋转处理完成后，形成的新子树仍然是平衡二叉排序树，且其深度和插入前以a为根结点的平衡二叉排序树的深度相同。所以，在平衡二叉排序树上因插入结点而失衡，仅需对失衡子树做平衡化旋转处理。

平衡二叉排序树的插入

平衡二叉排序树的插入操作实际上是在二叉排序插入的基础上完成以下工作：

⑴：判别插入结点后的二叉排序树是否产生不平衡?

⑵：找出失去平衡的最小子树；

⑶：判断旋转类型，然后做相应调整。

失衡的最小子树的根结点a在插入前的平衡因子不为0，且是离插入结点最近的平衡因子不为0的结点的。

1 算法思想(插入结点的步骤)

①：按照二叉排序树的定义，将结点s插入；

②：在查找结点s的插入位置的过程中，记录离结点s最近且平衡因子不为0的结点a，若该结点不存在，则结点a指向根结点；

③： 修改结点a到结点s路径上所有结点的平衡因子；

④：判断是否产生不平衡，若不平衡，则确定旋转类型并做相应调整。

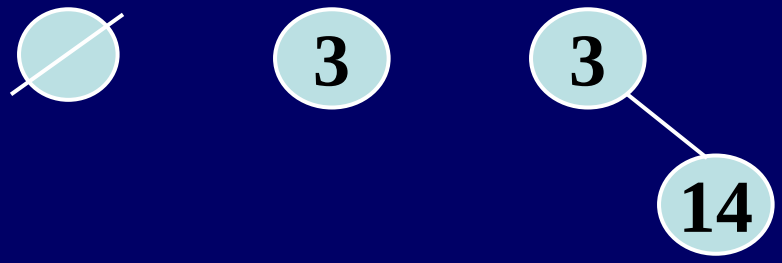
2 算法实现

```
void  Insert_BBST(BBSTNode *T, BBSTNode *S)
  {   BBSTNode *f,*a,*b,*p,*q;
      if  (T==NULL)  {   T=S ; T->Bfactor=0 ; return ;  }
      a=p=T ;     /* a指向离s最近且平衡因子不为0的结点 */
      f=q=NULL ;       /*   f指向a的父结点,q指向p父结点   */
      while (p!=NULL)
         {  if (S->key==p->key)  return ;    /*  结点已存在  */
            if (p->Bfactor!=0)  {  a=p ; f=q ;  }
            q=p ;
            if (S->key < p->key)  p=p->Lchild ;
            else p=p->Rchild ;     /*  在右子树中搜索  */
         }     /*  找插入位置  */
```

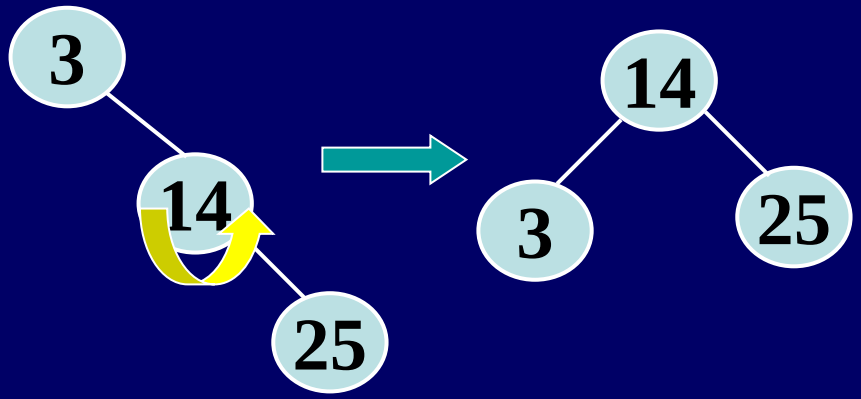
```
if (S->key < q->key)  q->Lchild=S ;/*  s为左孩子  */
else   q->Rchild=S ;      /*  s插入为q的右孩子  */
p=a ;
while (p!=S)
   {   if (S->key < p->key )
          {   p->Bfactor++ ;  p=p->Lchild ;   }
       else {   p->Bfactor-- ; p=p->Rchild ;  }
   }     /*  插入到左子树,平衡因子加1,插入到左子树,减1  */
if (a->Bfactor>-2&& a->Bfactor<2)
   return ;   /*   未失去平衡,不做调整  */
if (a->Bfactor==2)
   {   b=a->Lchild ;
```

```
        if (b->Bfactor==1)    p=LL_rotate(a) ;
        else p=LR_rotate(a) ;
    }
else
    {  b=a->Rchild ;
        if (b->Bfactor==1)    p=RL_rotate(a) ;
        else  p=RR_rotate(a) ;
    }  /*  修改双亲结点指针  */
if (f==NULL)  T=p ;      /*  p为根结点  */
else    if (f->Lchild==a) f->Lchild=p ;
        else  f->Rchild=p ;
}
```

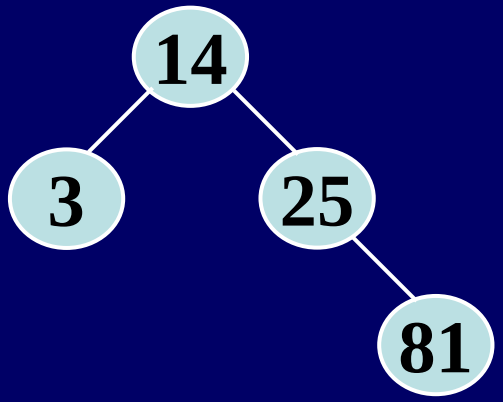
例： 设要构造的平衡二叉树中各结点的值分别是(3, 14, 25, 81, 44)， 平衡二叉树的构造过程如下。



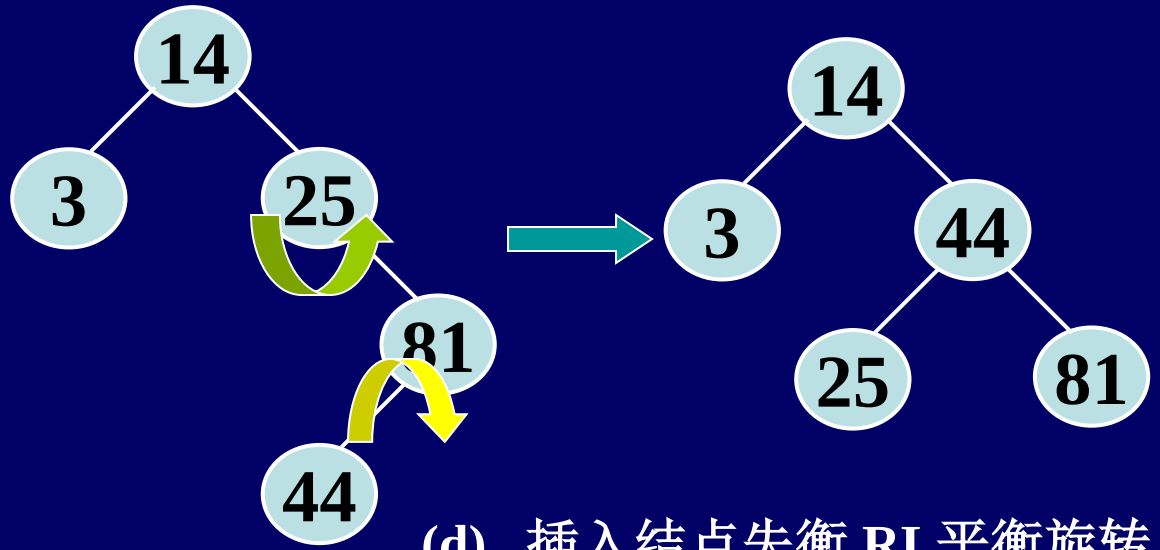
(a) 插入不超过两个结点



(b) 插入新结点失衡,RR平衡旋转



(c) 插入新结点未失衡



(d) 插入结点失衡,RL平衡旋转

平衡二叉树的构造过程

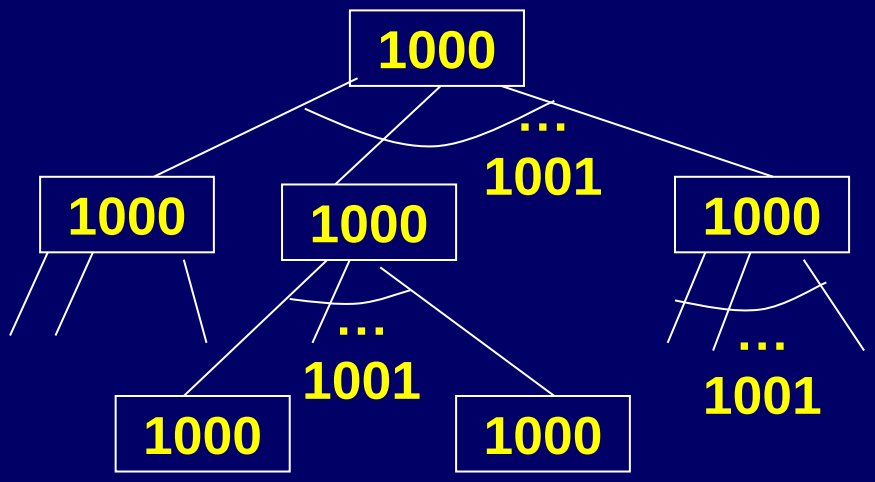
§ 9.3.2 B-树

1.概念

- B-树是一种完全平衡的多阶查找树，主要是作为磁盘文件的索引组织，用于外部查找。
- 基本想法是增大结点来降低树高，减少访问外存的次数

一棵m阶B-树，每个结点最多有m个孩子，m一般为50－2000

例如，1棵1001阶B-树，3层即可包含10亿以上的Keys，当根结点置于内存时，查找任一Key至多只要访问2次外存。



1个结点
1000个关键字

1001个结点
1,001,000个关键字

1,002,001个结点
1,002,001,000个关键字

§ 9.3.2 B-树

- **定义：一棵m(m≥3)阶的B-树是满足下述性质的m叉树：**
  - **性质1：** 每个结点至少包含下列数据域：$(j, P_0, K_1, P_1, K_2, \ldots, K_j, P_j)$

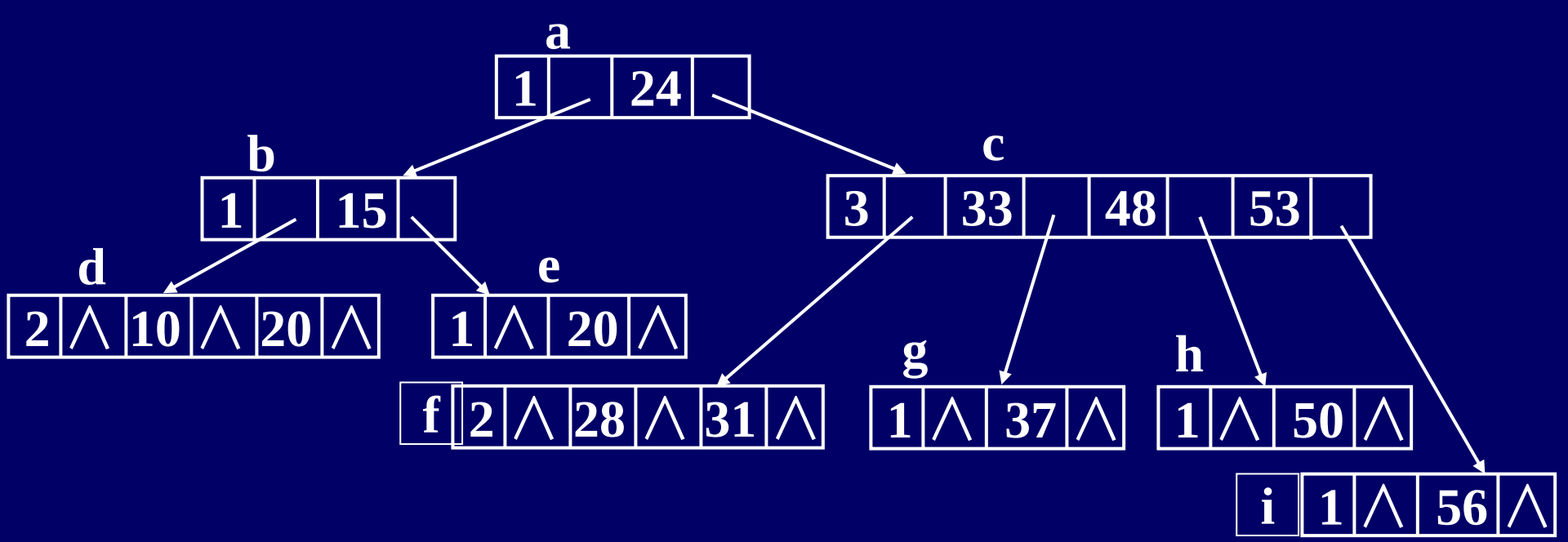
    j：keys总数， K_i：关键字， P_i：指针

    $keys(P_0) < K_1 < keys(P_1) < K_2 < \ldots < K_j < keys(P_j)$
  - **性质2：** 所有叶子在同一层上，叶子层数为树高h，叶子中的指针为空。
  - **性质3：** 每个非根结点中的关键字数目j满足：

$$\lceil m/2 \rceil - 1 \le j \le m - 1$$

    即：每个非根的内部结点的子树数在m和$\lceil m/2 \rceil$之间
  - **性质4：** 非空的B－树中，根至少有1个关键字。

    即：根不是叶子时，子树数为：2～m之间



一棵包含13个关键字的4阶B_树

B-树

运算

- **查找**

  多路查找：先在结点内查找（折半或顺序），后在子结点中查找（读盘）

- **插入和生成**

  平衡机制：满时插入分裂结点，从叶子 ⟹ 根，树高可能长一层

- **删除**

  平衡机制：半满时删除，可能要合并结点，从叶子 ⟹ 根，树高可能减一层

根据m阶B_树的定义，结点的类型定义如下：

```
#define M    5      /*  根据实际需要定义B_树的阶数  */
typedef  struct BTNode
  {  int   keynum ;   /*   结点中关键字的个数   */
     struct BTNode  *parent ;    /*   指向父结点的指针   */
     KeyType  key[M+1] ;     /*   关键字向量,key[0]未用   */
     struct BTNode  *ptr[M+1] ; /*   子树指针向量   */
     RecType   *recptr[M+1] ;
        /*   记录指针向量,recptr[0]未用   */
  }BTNode ;
```

B_树的查找

由B_树的定义可知，在其上的查找过程和二叉排序树的查找相似。

(1) 算法思想

① 从树的根结点T开始，在T所指向的结点的关键字向量key[1…keynum]中查找给定值K(用折半查找)：

若key[i]=K(1≤i≤keynum)，则查找成功，返回结点及关键字位置；否则，转(2)；

② 将K与向量key[1…keynum]中的各个分量的值进行比较，以选定查找的子树：

- 若K<key[1]：T=T->ptr[0]；

- 若key[i]<K<key[i+1](i=1, 2, …keynum-1)：
  T=T->ptr[i]；
- 若K>key[keynum]：T=T->ptr[keynum]；

转①，直到T是叶子结点且未找到相等的关键字，则查找失败。

(2) 算法实现

```
int  BT_search(BTNode *T, KeyType K, BTNode *p)
  /*  在B_树中查找关键字K, 查找成功返回在结点中的位置  */
 /*  及结点指针p; 否则返回0及最后一个结点指针   */
 {   BTNode *q ; int n ;
     p=q=T ;
```

```
    while  (q!=NULL)
       {  p=q ; q->key[0]=K ;      /*   设置查找哨兵   */
          for (n=q->keynum ; K<q->key[n] ; n--);
          if (n>0&&EQ(q->key[n], K) )    return n ;
          q=q->ptr[n] ;
       }
    return 0 ;
  }
```

(3) 算法分析

在B_树上的查找有两中基本操作：

- 在B_树上查找结点(查找算法中没有体现)；

◆ 在结点中查找关键字：在磁盘上找到指针ptr所指向的结点后，将结点信息读入内存后再查找。因此，磁盘上的查找次数(待查找的记录关键字在B_树上的层次数)是决定B_树查找效率的首要因素。

根据m阶B_树的定义，第一层上至少有1个结点，第二层上至少有2个结点；除根结点外，所有非终端结点至少有$\lceil m/2 \rceil$棵子树，…，第h层上至少有$\lceil m/2 \rceil^{h-2}$个结点。在这些结点中：根结点至少包含1个关键字，其它结点至少包含$\lceil m/2 \rceil$-1个关键字，设s=$\lceil m/2 \rceil$，则总的关键字数目n满足：

$$n \geqq 1+(s-1)\sum_{i=2}^{h} 2s^{i-2}=1+\ 2(s-1)\frac{s^{h-1}-1}{s-1}\ =2s^{h-1}-1$$

因此有： $h \leqq 1+\log_s((n+1)/2)=1+\log_{\lceil m/2 \rceil}((n+1)/2)$

即在含有n个关键字的B_树上进行查找时，从根结点到待查找记录关键字的结点的路径上所涉及的结点数不超过$1+\log_{\lceil m/2 \rceil}((n+1)/2)$。

B_树的插入

B_树的生成也是从空树起，逐个插入关键字。插入时不是每插入一个关键字就添加一个叶子结点，而是首先在最低层的某个叶子结点中添加一个关键字，然后有可能“分裂”。

(1) 插入思想

① 在B_树的中查找关键字K，若找到，表明关键字已存在，返回；否则，K的查找操作失败于某个叶子结点，转 ②；

② 将K插入到该叶子结点中，插入时，若：

- 叶子结点的关键字数<m-1：直接插入；
- 叶子结点的关键字数=m-1：将结点“分裂”

⑵ 结点“分裂”方法

设待“分裂”结点包含信息为：

$(m, A_0, K_1, A_1, K_2, A_2, \ldots, K_m, A_m)$，从其中间位置分为两个结点：

$(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \ldots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$

$(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \ldots, K_m, A_m)$

并将中间关键字$K_{\lceil m/2 \rceil}$插入到p的父结点中，以分裂后的两个结点作为中间关键字$K_{\lceil m/2 \rceil}$的两个子结点。

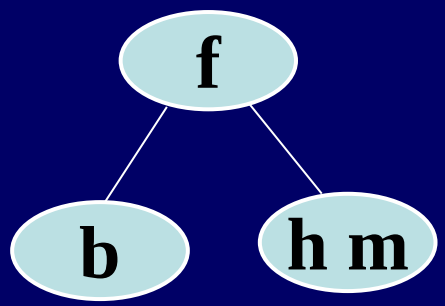
当将中间关键字$K_{\lceil m/2 \rceil}$插入到p的父结点后，父结点也可能不满足m阶B_树的要求(分枝数大于m)，则必须对父结点进行“分裂”，一直进行下去，直到没有父结点或分裂后的父结点满足m阶B_树的要求。

当根结点分裂时，因没有父结点，则建立一个新的根，B_树增高一层。

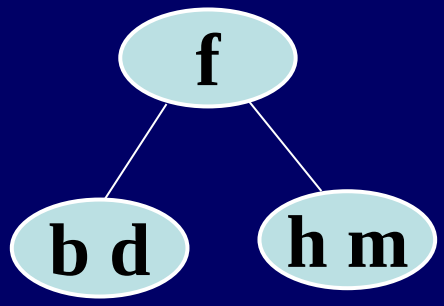
例（下页）：在一个3阶B_树(2-3树)上插入结点的过程。

(3) 算法实现

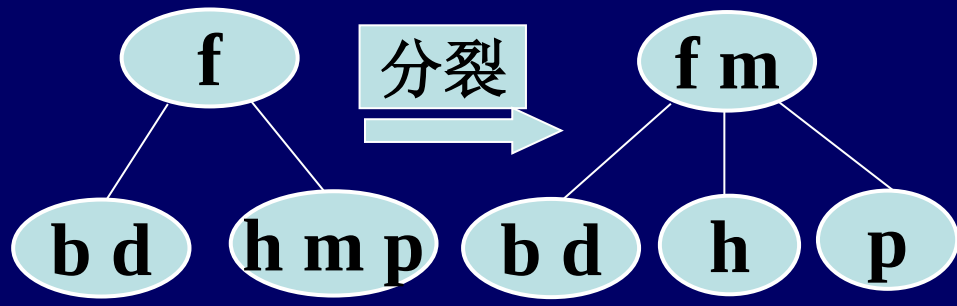
要实现插入，首先必须考虑结点的分裂。设待分裂的结点是p，分裂时先开辟一个新结点，依此将结点p中后半部分的关键字和指针移到新开辟的结点中。分裂之后，而需要插入到父结点中的关键字在p的关键字向量的p->keynum+1位置上。



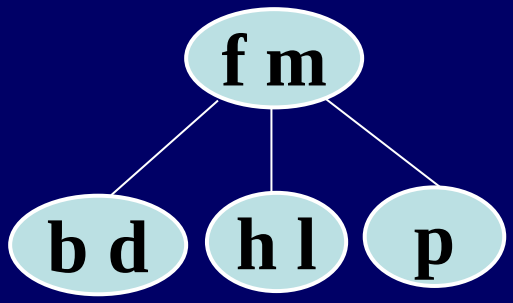
(a) 一棵2-3树



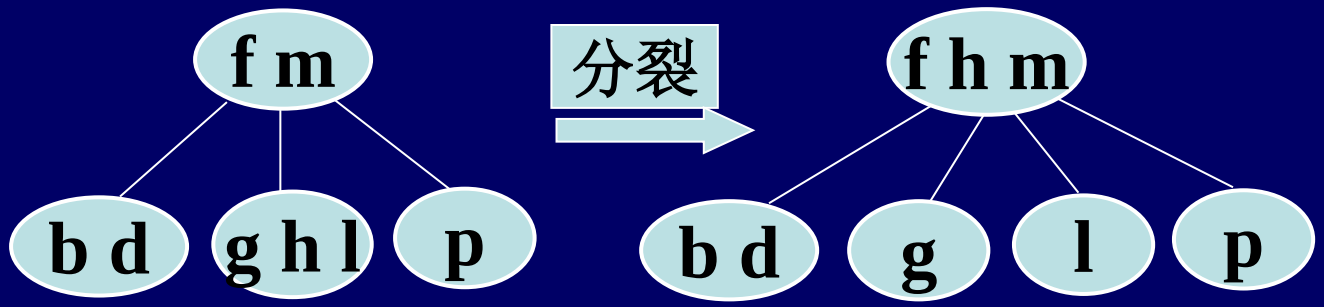
(b) 插入d后



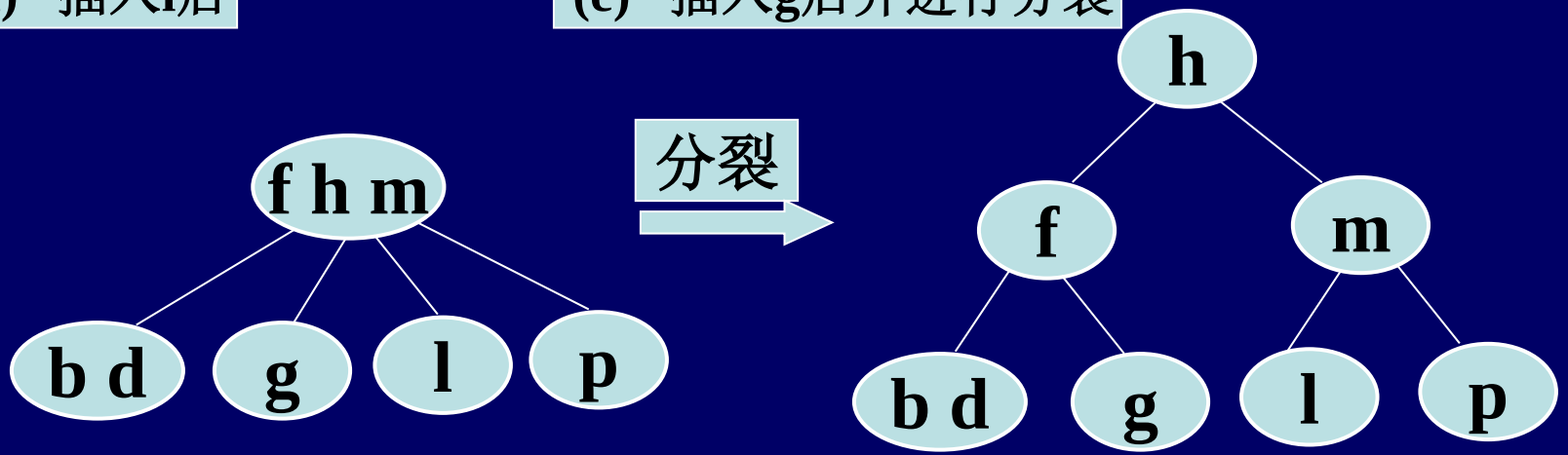
(c) 插入p后并进行分裂



(d) 插入l后



(e) 插入g后并进行分裂



(f) 继续进行分裂

在B_树中进行插入的过程

```
BTNode  *split(BTNode *p)
  /*  结点p中包含m个关键字，从中分裂出一个新的结点  */
  {  BTNode *q ;  int k, mid, j ;
     q=(BTNode *)malloc(sizeof( BTNode)) ;
     mid=(m+1)/2 ;   q->ptr[0]=p->ptr[mid] ;
     for (j=1,k=mid+1; k<=m; k++)
        {   q->key[j]=p->key[k] ;
            q->ptr[j++]=p->ptr[k] ;
        }  /*  将p的后半部分移到新结点q中  */
     q->keynum=m-mid ;  p->keynum=mid-1 ;
     return(q) ;
  }
```

```
void  insert_BTree(BTNode *T, KeyType  K)
    /*   在B_树T中插入关键字K，*/
   {   BTNode *q, *s1=NULL, *s2=NULL ;
       int n ;
       if  (!BT_search(T, K, p))     /*  树中不存在关键字K  */
          {   while (p!=NULL)
              {  p->key[0]=K ;     /*   设置哨兵   */
                 for (n=p->keynum ; K<p->key[n] ; n--)
                    {   p->key[n+1]=p->key[n] ;
                        p->ptr[n+1]=p->ptr[n] ;
                    }    /*   后移关键字和指针   */
                 p->key[n+1]=K ; p->ptr[n]=s1 ;
```

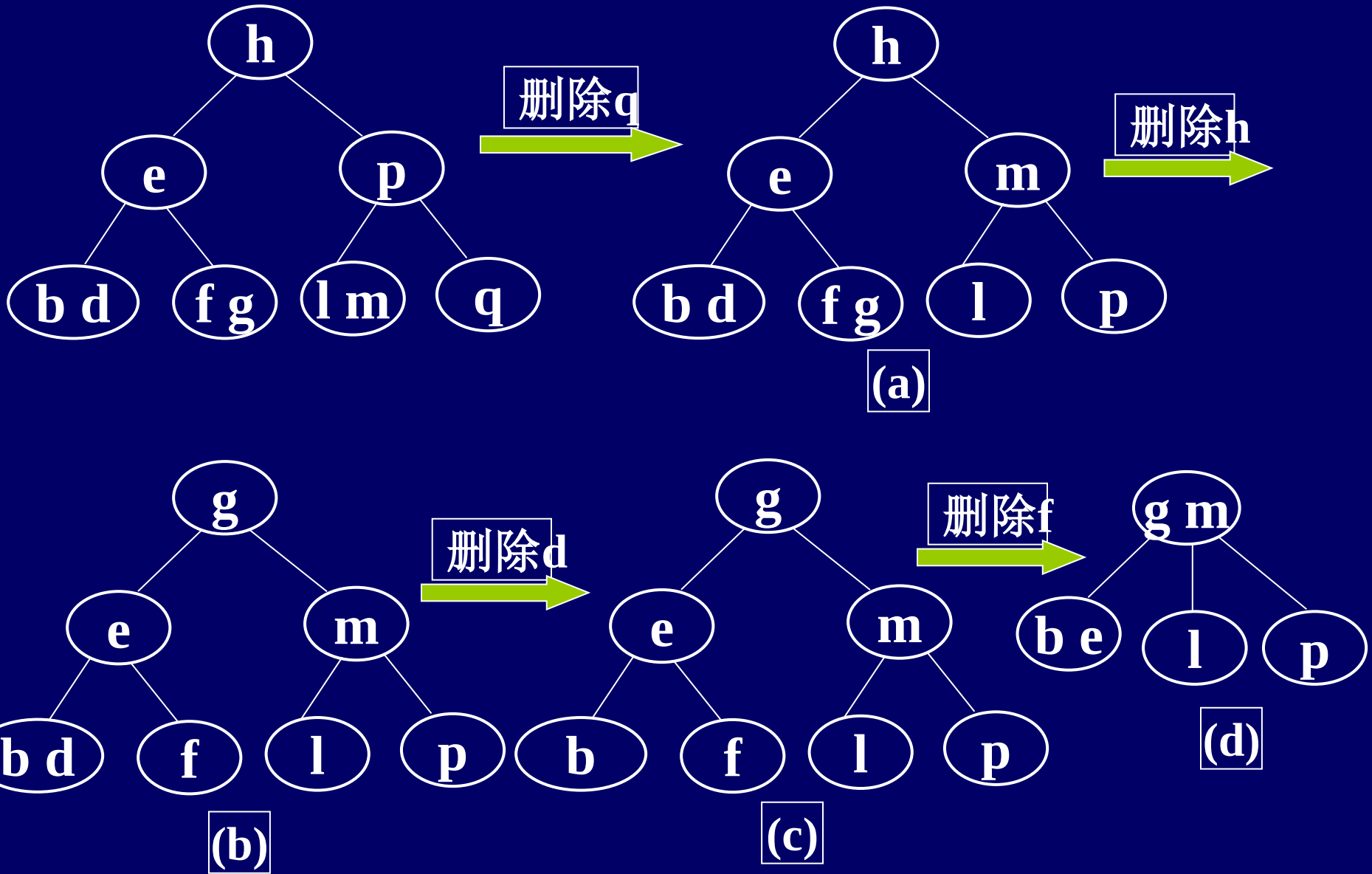
```
            p->ptr[n+1]=s2 ;   /*  置关键字K的左右指针  */
        if (++(p->keynum )<m)  break ;
        else {  s2=split(p) ; s1=p ;  /*  分裂结点p  */
                K=p->key[p->keynum+1] ;
                p=p->parent ;    /*  取出父结点*/
              }
    }
        if (p==NULL)       /*  需要产生新的根结点  */
           {  p=(BTNode*)malloc(sizeof( BTNode)) ;
              p->keynum=1 ; p->key[1]=K ;
              p->ptr[0]=s1 ; p->ptr[1] =s2 ;
            }
    } }
```

利用m阶B_树的插入操作，可从空树起，将一组关键字依次插入到m阶B_树中，从而生成一个m阶B_树。

B_树的删除

在B_树上删除一个关键字K，首先找到关键字所在的结点N，然后在N中进行关键字K的删除操作。

若N不是叶子结点，设K是N中的第i个关键字，则将指针A_{i-1}所指子树中的最大关键字(或最小关键字)K'放在(K)的位置，然后删除K'，而K'一定在叶子结点上。如图，删除关键字h，用关键字g代替h的位置，然后再从叶子结点中删除关键字g。



在B_树中进行删除的过程

从叶子结点中删除一个关键字的情况是：

⑴ 若结点N中的关键字个数>⌈m/2⌉-1：在结点中直接删除关键字K，如图 (b)~©所示。

⑵ 若结点N中的关键字个数=⌈m/2⌉-1：若结点N的左(右)兄弟结点中的关键字个数>⌈m/2⌉-1，则将结点N的左(或右)兄弟结点中的最大(或最小)关键字上移到其父结点中，而父结点中大于(或小于)且紧靠上移关键字的关键字下移到结点N，如图 (a)。

⑶ 若结点N和其兄弟结点中的关键字数=⌈m/2⌉-1：删除结点N中的关键字，再将结点N中的关键字、指针与其兄弟结点以及分割二者的父结点中的某个关键字K_i，合并为一个结点，若因此使父结点中的关键字个数<⌈m/2⌉-1，则依此类推，如图 (d)。

算法实现

在B_树上删除一个关键字的操作，针对上述的(2)和(3)的情况，相应的算法如下：

```
int  BTNode  MoveKey(BTNode *p)
  /*  将p的左(或右)兄弟结点中的最大(或最小)关键字上移  */
  /*   到其父结点中,父结点中的关键字下移到p中  */
  {  BTNode *b , *f=p->parent ;   /*  f指向p的父结点  */
     int k, j ;
     for (j=0; f->ptr[j]!=p; j++) ;   /*  在f中找p的位置  */
     if (j>0)        /*  若p有左邻兄弟结点  */
          {  b=f->ptr[j-1] ;     /*  b指向p的左邻兄弟  */
```

```
if (b->keynum>(m-1)/2)
        /*   左邻兄弟有多余关键字  */
    {  for (k=p->keynum; k>=0; k--)
          {   p->key[k+1]=p->key[k];
              p->ptr[k+1]=p->ptr[k];
           }      /*   将p中关键字和指针后移   */
        p->key[1]=f->key[j];
        f->key[j]=b->key[keynum] ;
     /*  f中关键字下移到p, b中最大关键字上移到f  */
        p->ptr[0]= b->ptr[keynum] ;
        p->keynum++ ;
        b->keynum-- ;
```

```
        return(1) ;
    }
if (j<f->keynum)     /*   若p有右邻兄弟结点   */
   {  b=f->ptr[j+1] ;      /*   b指向p的右邻兄弟   */
      if (b->keynum>(m-1)/2)
          /*   右邻兄弟有多余关键字  */
         {  p->key[p->keynum]=f->key[j+1] ;
            f->key[j+1]=b->key[1];
            p->ptr[p->keynum]=b->ptr[0];
  /*  f中关键字下移到p, b中最小关键字上移到f  */
            for (k=0; k<b->keynum; k++)
```

```
                    {   b->key[k]=b->key[k+1];
                        b->ptr[k]=b->ptr[k+1];
                    }      /*  将b中关键字和指针前移  */
                p->keynum++ ;
                b->keynum-- ;
                return(1) ;
            }
        }
    return(0);
}   /*  左右兄弟中无多余关键字,移动失败  */
}
```

```
BTNode  *MergeNode(BTNode *p)
   /*  将p与其左(右)邻兄弟合并,返回合并后的结点指针  */
  {   BTNode *b, f=p->parent ;
      int j, k ;
      for (j=0; f->ptr[j]!=p; j++);   /*  在f中找出p的位置  */
      if (j>0) b=f->ptr[j-1];    /*  b指向p的左邻兄弟  */
      else {  b=p; p=f->ptr[j+1];  }    /*  p指向p的右邻  */
      b->key[++b->keynum]=f->key[j] ;
      b->ptr[b->keynum]=p->ptr[0] ;
      for (k=1; k<=p->keynum ; k++)
         {   b->key[++b->keynum]=p->key[k] ;
             b->ptr[b->keynum]=p->ptr[k] ;
         }     /*  将p中关键字和指针移到b中  */
```

```
    free(p);
    for (k=j+1; k<=f->keynum ; k++)
       {   f->key[k-1]=f->key[k] ;
           f->ptr[k-1]=f->ptr[k] ;
       }     /*   将f中第j个关键字和指针前移   */
    f->keynum-- ;
    return(b) ;
}
```

```
void  DeleteBTNode(BTNode *T, KeyType K)
  {  BTNode  *p, *S ;
     int j,n ;
     j=BT_search(T, K, p) ; /* 在T中查找K的结点  */
     if (j==0)  return(T) ;
     if (p->ptr[j-1])
        {  S=p->ptr[j-1] ;
           while (S->ptr[S->keynum])
                S=S->ptr[S->keynum] ;
            /*  在子树中找包含最大关键字的结点   */
           p->key[j]=S->key[S ->keynum] ;
           p=S ; j=S->keynum ;
        }
```

```
for (n=j+1; n<p->keynum; n++)
    p->key[n-1]=p->key[n] ;
        /*  从p中删除第m个关键字   */
p->keynum-- ;
while (p->keynum<(m-1)/2&&p->parent)
    {   if (!MoveKey(p) )  p=MergeNode(p);
        p=p->parent ;
    }    /*  若p中关键字树目不够,按(2)处理     */
 if (p==T&&T->keynum==0)
    {   T=T->ptr[0] ;   free(p) ;    }
}
```

B^+树

在实际的文件系统中，基本上不使用B_树，而是使用B_树的一种变体，称为m阶B^+树。 它与B_树的主要不同是叶子结点中存储记录。在B^+树中，所有的非叶子结点可以看成是索引，而其中的关键字是作为“分界关键字”，用来界定某一关键字的记录所在的子树。一棵m阶B^+树与m阶B_树的主要差异是：

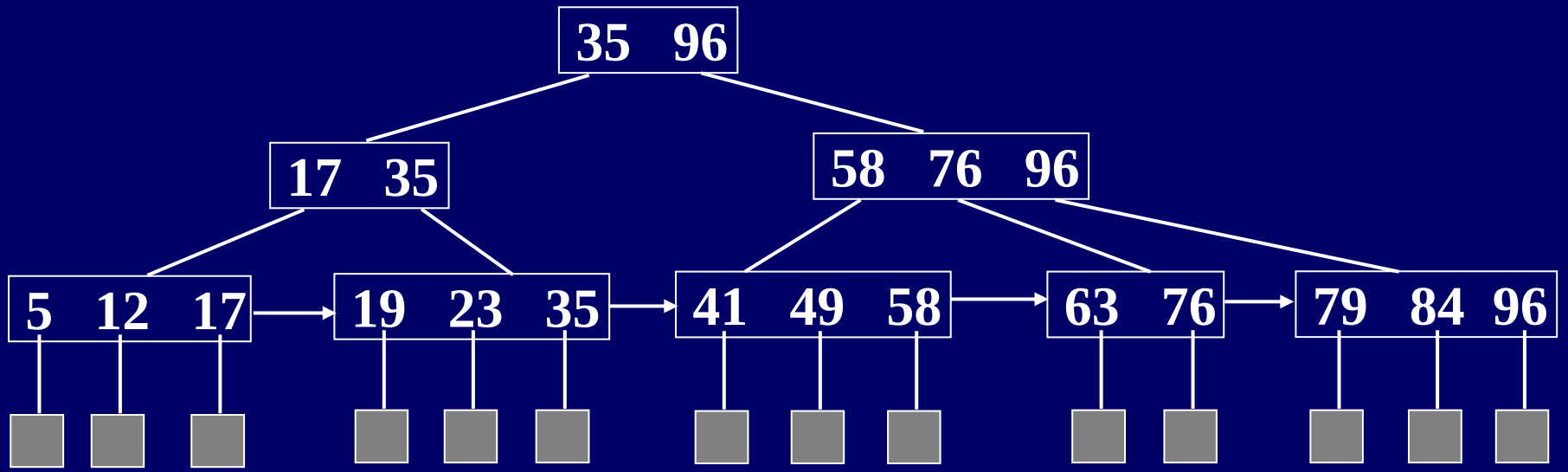
（1）若一个结点有n棵子树，则必含有n个关键字；

（2）所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，而且叶子结点按关键字的大小从小到大顺序链接；

(3) 所有的非叶子结点可以看成是索引的部分，结点中只含有其子树的根结点中的最大(或最小)关键字。

如图所示是一棵3阶B+树。

由于B+树的叶子结点和非叶子结点结构上的显著区别，因此需要一个标志域加以区分，结点结构定义如下：



一棵3阶B+树

```
typedef  enum{branch, left}  NodeTag ;
typedef  struct BPNode
   {   NodeTag   tag ;     /*  结点标志  */
       int   keynum ;   /*  结点中关键字的个数  */
       struct BTNode  *parent ;    /*  指向父结点的指针  */
       KeyType  key[M+1] ;    /*  组关键字向量,key[0]未用
       */
       union pointer
          {   struct BTNode  *ptr[M+1] ; /*  子树指针向量  */
              RecType   *recptr[M+1] ;  /*  recptr[0]未用  */
          }ptrType ;    /*  用联合体定义子树指针和记录指针  */
   }BPNode ;
```

与B_树相比，对B^+树不仅可以从根结点开始按关键字随机查找，而且可以从最小关键字起，按叶子结点的链接顺序进行顺序查找。在B^+树上进行随机查找、插入、删除的过程基本上和B_树类似。

在B^+树上进行随机查找时，若非叶子结点的关键字等于给定的K值，并不终止，而是继续向下直到叶子结点(只有叶子结点才存储记录)，即无论查找成功与否，都走了一条从根结点到叶子结点的路径。

B^+树的插入仅仅在叶子结点上进行。当叶子结点中的关键字个数大于m时，“分裂”为两个结点，两个结点中所含有的关键字个数分别是$\lfloor (m+1)/2 \rfloor$和$\lceil (m+1)/2 \rceil$，且将这两个结点中的最大关键字提升到父结点中，用来替代原结点在父结点中所对应的关键字。提升后父结点又可能会分裂，依次类推。

红黑树定义

- **Red-Black tree, 简称RB-Tree**
- **它是在1972年由鲁道夫·贝尔发明的，他称之为“对称二叉B树”，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于1978年写的一篇论文中获得的**
- **特点：利用对树中的结点“红黑着色”的要求，降低了平衡性的条件，达到局部平衡，有着良好的最坏情况运行时间，它可以在O(log n)时间内做查找，插入和删除，这里的n是树中元素的数目。**

红黑树的应用

- C++ STL中的关联式容器：集合set、多重集合multiset、映射map、多重映射multimap
- JAVA集合框架：TreeSet和TreeMap
- 在Linux内核中，用于组织虚拟区间的数据结构也是红黑树

代码参见：

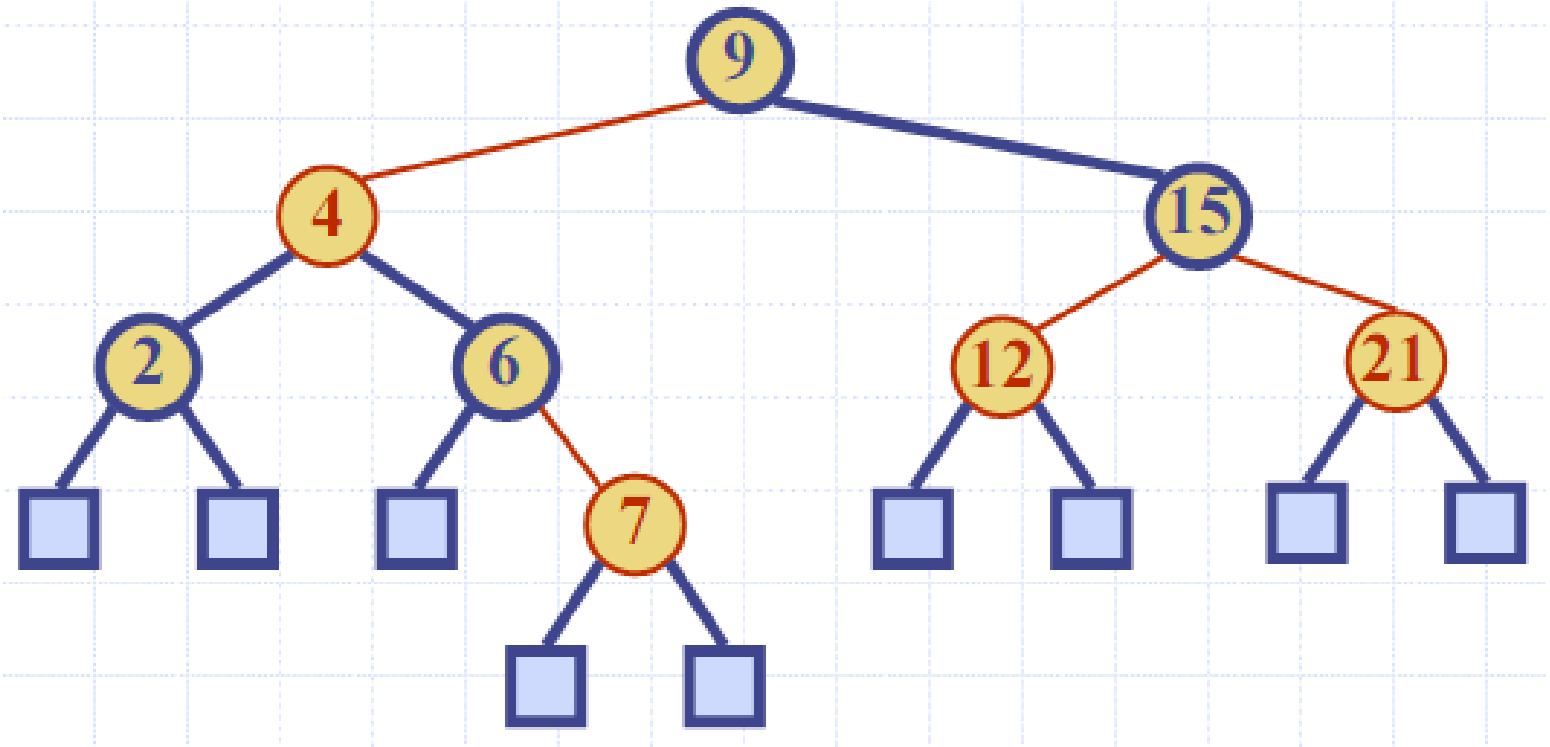
linux/include/linux/rbtree.h

linux/lib/rbtree.c

红黑树的定义

- 平衡的扩充二叉搜索树，满足下面条件：
  - 颜色特征：每个结点为“黑色”或“红色”
  - 根特征：根结点永远是“黑色”的
  - 外部特征：扩充外部叶结点都是空的“黑色”结点
  - 内部特征：“红色”结点的两个子结点都是“黑色”的，即：不允许两个连续的红色结点
  - 深度特征：对于每个结点，从该结点到其所有子孙叶结点的路径中所包含的黑色结点数量必须相同

红黑树示例



树结点的结构

Key	Data	Color	parent	lchild	rchild

7.3 哈希表的查找

上述查找均是基于key的比较，由判定树可证明，在平均和最坏情况下所需的比较次数的下界是：

$$\lg n + O(1)$$

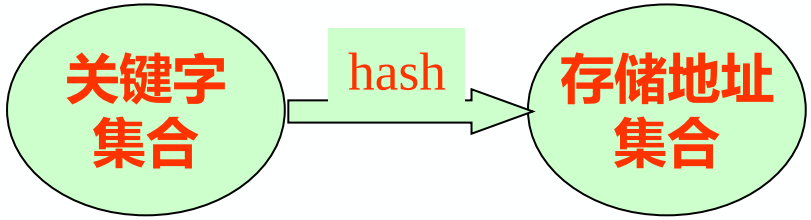
要突破此界，就不能只依赖于比较来进行。

直接寻址：当结点的key和存储位置间建立某种关系时，无须比较即可找到结点的存储位置。

例如，若500个结点的keys取值为1 ~ 500间互不相同的整数，则keys可作为下标，将其存储到T[1..500]之中，寻址时间为O (1)。

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200101181102**01**的所有信息存入V[01]单元；
将200101181102**02**的所有信息存入V[02]单元；

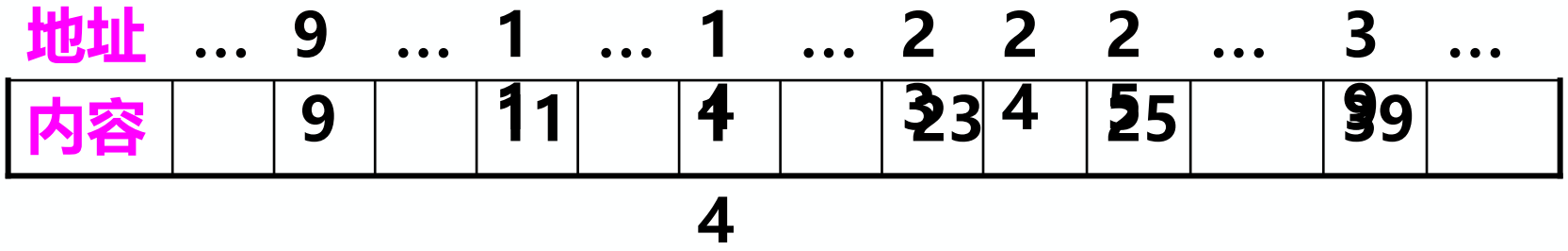
……

将200101181102**31**的所有信息存入V[31]单元。

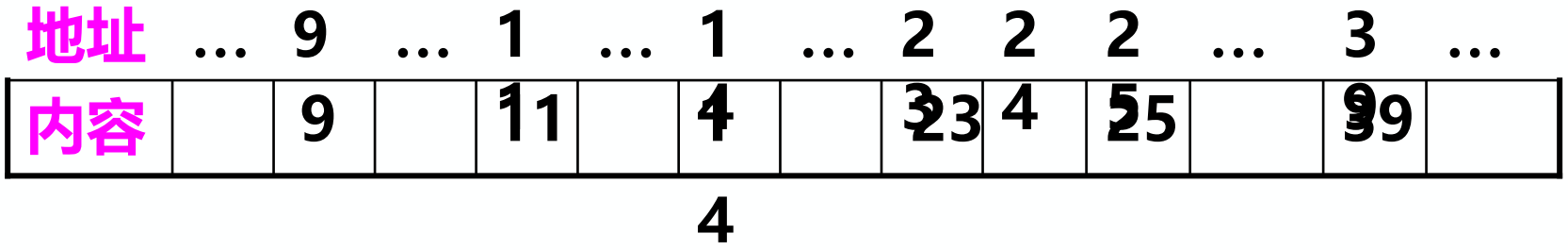
查找200101181102**16**的信息，可直接访问V[16]！

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。



如何查找

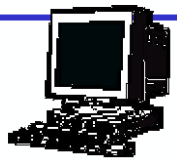


根据哈希函数H（k）＝k

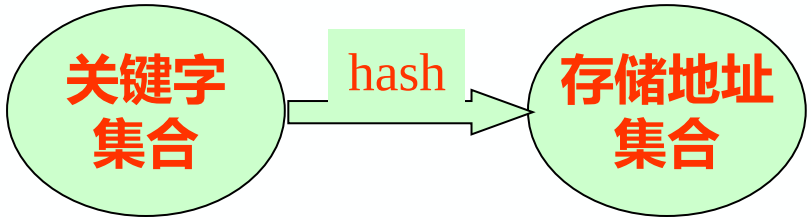
查找key=9,则访问H(9)=9号地址，若内容为9则成功；

若查不到，则返回一个特殊值，如空指针或空记录。

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**



- **一般情况:**
  全集U：可能发生的关键字集合，即关键字的取值集合

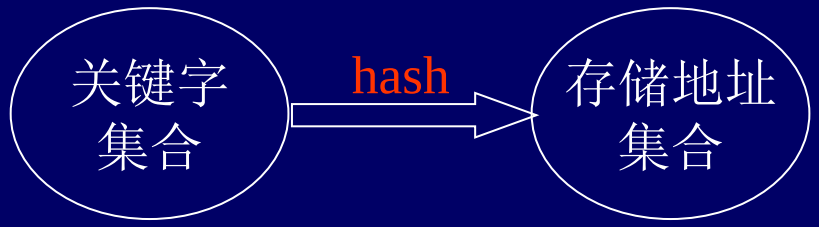
  实际发生集K：实际需要存储的关键字集合

  ∵ |K| << |U|，当表T的规模取|U|浪费空间，有时甚至无法实现

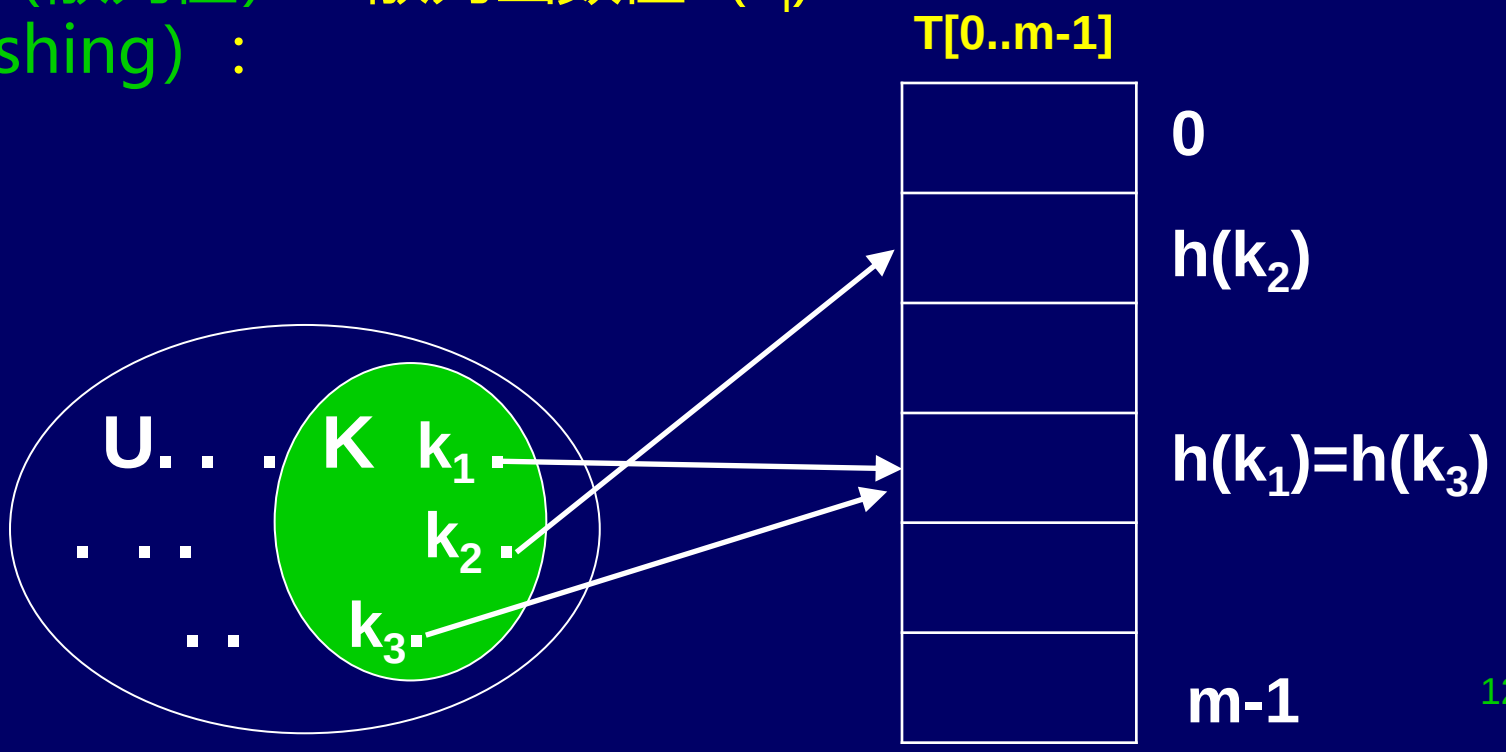
  ∴ T的规模一般取O(K), 但压缩存储后失去了直接寻址的功能

有关术语

- 散列（hash，哈希，杂凑）函数h
  在keys和表地址之间建立一个对应关系h，将全集U映射到T[0..m-1]的下标集上：
  h：U→{ 0, 1, … , m-1 }
  $\forall k_i \in U$，$h(k_i) \in \{ 0,1,\ldots,m-1 \}$



- 散列表（哈希表）：T
- 散列地址（散列值）：散列函数值$h(k_i)$
- 散列（hashing）：



有关术语

冲 突：不同的关键码映射到同一个哈希地址

$$key1 \neq key2，但H(key1)=H(key2)$$

同义词：具有相同函数值的两个关键字

能完全避免冲突吗?

须满足：① $|U| \leq m$ ② 选择合适的hash函数

否则只能设计好的h使冲突尽可能少

解决冲突： 须确定处理冲突的方法

装填因子： 冲突的频繁程度除了与h的好坏相关，还与表的填满程度相关

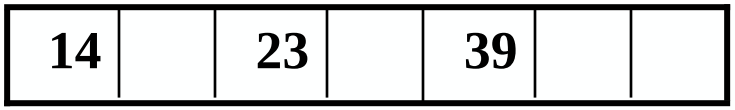
$$\alpha = n/m, \quad 0 \leq \alpha \leq 1$$

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!



9

25

11

H(14)=14%7=0

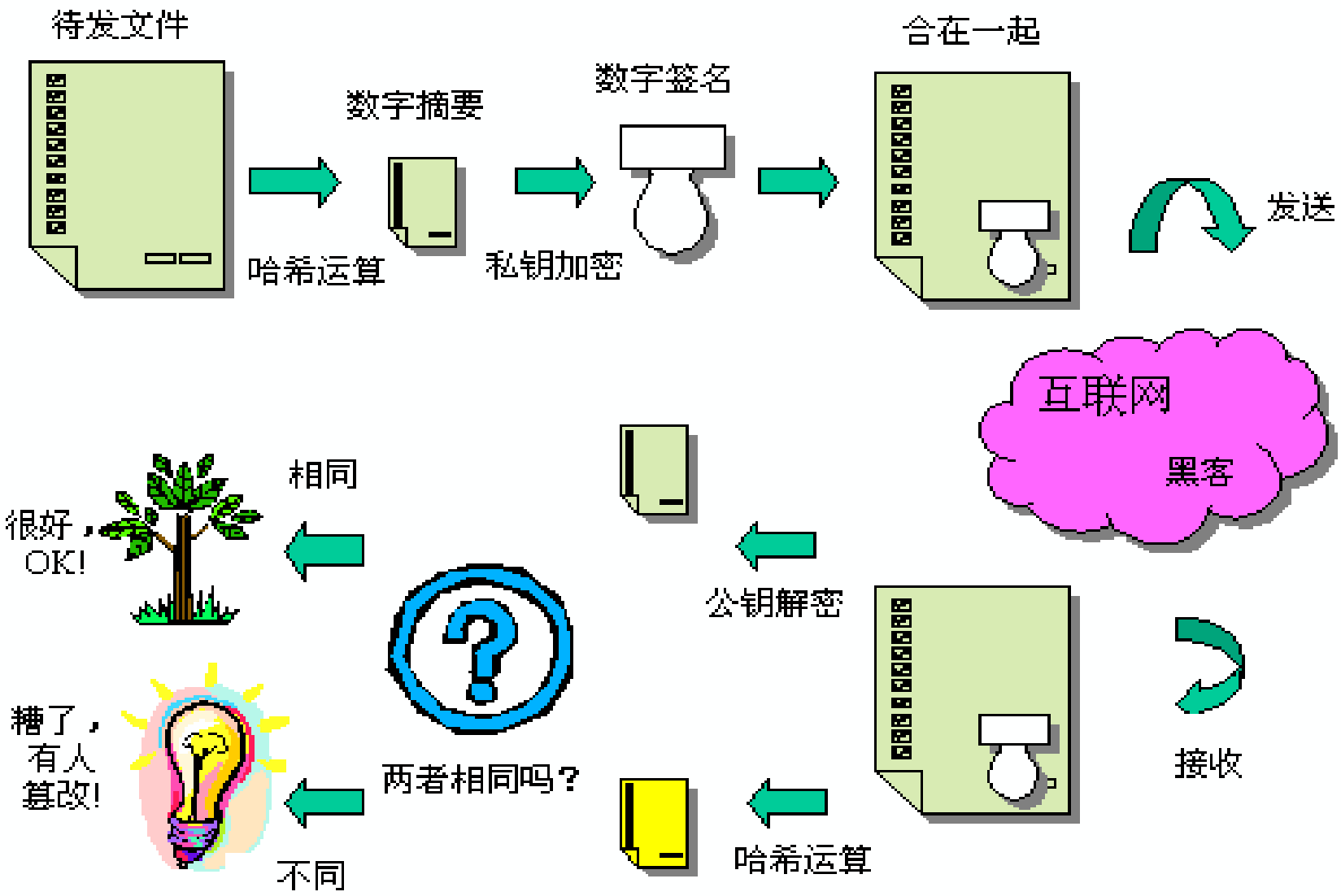
H(25)=25%7=4

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，
构造原则：简单、均匀

简单 - - 计算快速

均匀 - - 冲突最小化

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**

假定：关键字定义在自然数集合上，否则将其转换到自然数集合上

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。

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

平方取中法

平方取中法

先通过平方扩大相近数的差别，然后取中间的若干位作为散列地址。因为中间位和乘数的每一位相关，故产生的地址较为均匀。

```
int Hash( int k ) {
    k *=k;    k /=100;  //去掉末尾两位数
    return k%1000;  //取中间3位，T的地址为0~999
  }
```

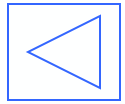
除留余数法 （最常用重点掌握）

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
优点：最简单，无需定义为函数，直接写到程序里使用

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 记录的查找频率。



处理冲突的方法

1.开放地址法

2.链地址法

处理冲突的方法

1、开放地址法（闭散列）

- **基本思想：**当发生冲突时，使用某种探测技术在散列表中形成一个**探测序列**，沿此序列逐个单元查找，直到找到给定的key，或碰到1个**开放地址**（空单元）为止
  - 插入时，探测到开地址可将新结点插入其中
  - 查找时，探测到开地址表示查找失败
  - 开地址表示：与应用相关，如key为串时，用空串表示开地址
- **一般形式**

  $h_i = (h(k) + d_i) \% m \qquad 1 \le i \le m-1$

  探测序列：$h(k), h_1, h_2, \ldots, h_{m-1}$

  开地址法要求：$\alpha < 1$，　实用中 $0.5 \le \alpha \le 0.9$

处理冲突的方法

- **开放地址法分类：**按照形成探测序列的方法不同，可将其分为3类：线性探测、二次探测、伪随机探测

（1）线性探测法

- **基本思想：**将T[0..m-1]看作一循环向量，令$d_i=i$，即

$$h_i=(h(k)+i)\%m \qquad 0\le i\le m-1$$

若令d＝h(k)，则最长的探测序列为：

d,d+1,…,m-1,0,1,…,d-1

探测过程终止于

- 探测到空单元：查找时失败，插入时写入
- 探测到含有k的单元：查找时成功，插入时失败
- 探测到T[d-1] 但未出现上述2种情况：查找、插入均失败

处理冲突的方法

- **例1：** 已知一组keys为（26,36,41,38,44,15,68,12,06,51），用除余法和线性探测法构造散列表

解： ∵ $\alpha<1$，n＝10，不妨取m＝13，此时$\alpha\approx0.77$

h(k)= k%13　　keys：(26,36,41,38,44,15,68,12,06,51)

对应的散列地址：(0,10, 2, 12, 5, 2, 3, 12, 6, 12)

	0	1	2	3	4	5	6	7	8	9	10	11	12
T[0..12]	26		41			44					36		38
	26	12	41	15	68	44	06	51			36		38
探测次数	1	3	1	2	2	1	1	9			1		1

处理冲突的方法

- **非同义词冲突**：上例中，h(15)=2，h(68)=3，15和68不是同义词，但是15和41这两个同义词处理过程中，15先占用了T[3]，导致插入68时发生了非同义词的冲突。
- **聚集（堆积）**：一般地，用线性探测法解决冲突时，当表中i,i+1,…,i+k的位置上已有结点时，一个散列地址为i,i+1,…,i+k,i+k+1的结点都将争夺同一地址：i＋k＋1。这种不同地址的结点争夺同一后继地址的现象称为**聚集**。

  聚集增加了探测序列的长度，使查找时间增加。应**跳跃式**地散列。

	0	1	2	3	4	5	6	7	8	9	10	11	12
T[0..12]	26		41			44					36		38
	26	12	41	15	68	44	06	51			36		38
探测次数	1	3	1	2	2	1	1	9			1		1

处理冲突的方法

（2）二次探测法：

探测序列为(增量序列为$d_i = i^2$)：

h_i＝（h(k)＋i*i）％m　　0 ≤i ≤ m－1

缺点：不易探测到整个空间

（3）伪随机探测法：

$h_i = h(k)+d_i$ ％ m　　(0≤ 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

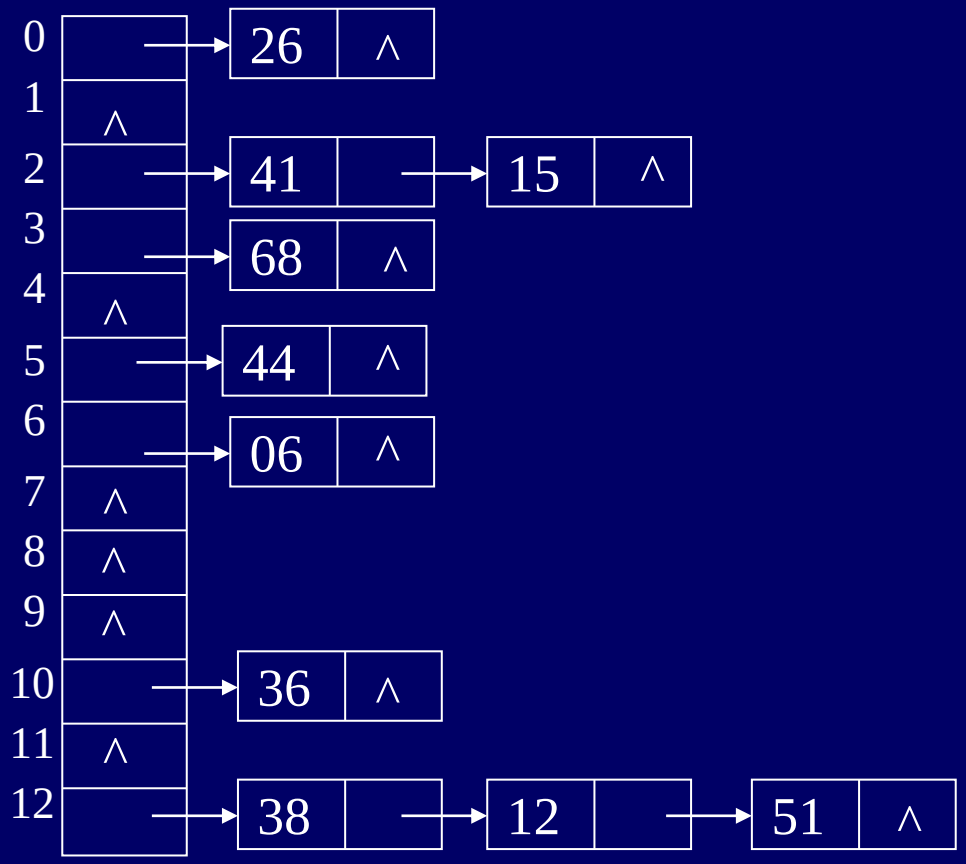
- step1 取数据元素的关键字key，计算其哈希函数值（地址）。若该地址对应的存储空间还没有被占用，则将该元素存入；否则执行step2解决冲突。
-
- step2 根据选择的冲突处理方法，计算关键字key的下一个存储地址。若下一个存储地址仍被占用，则继续执行step2，直到找到能用的存储地址为止。

处理冲突的方法

2、拉链法（开散列，链地址法）

- **基本思想：**将所有keys为同义词的结点连接在同一个单链表中，若散列地址空间为0～m-1，则将表定义为m个头指针组成的指针数组T[0..m-1]，其初值为空。

例2：keys和hash函数同前例



比较次数：　1　　2　　3

处理冲突的方法

- 特点
  - 无堆积现象、非同义词不会冲突、ASL较短
  - 无法预先确定表长度时，可选此方法
  - 删除操作易于实现，开地址方法只能做删除标记
  - 允许$\alpha>1$，当n较大时，节省空间

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。

散列表上的运算

仅给出开放定址法处理冲突时的相关运算

- 存储结构

```
#define NIL -1  //空结点标记
#define m  997 //表长，依赖应用和α
typedef struct {
    KeyType key;
    //….
}NodeType, HashTable[m];
```

1、查找运算

和建表类似，使用的函数和处理冲突的方法应与建表一致

- 开地址可统一处理

散列表上的运算

```
int Hash( KeyType K, int i){
    return ( h(K)+ Increment( i ) )%m; //Increment相当于d_i
}
Int HashSearch( HashTable T, KeyType K, int *pos){
   int i=0; //探测次数计数器
   do {  *pos=Hash( K,i ); //求探测地址h_i
         if ( T[*pos].key==K ) return 1; //成功
         if ( T[*pos].key==NIL ) return 0; //失败
   } while (++i<m); //最多探测m次
   return -1; //可能是表满，失败
}
```

散列表上的运算

2、插入和建表

- **插入：**先查找，找到开地址成功，否则(表满、K存在)失败

```
void HashInsert( HashTable T, NodeType new ){
    int pos, sign;
    sign=HashSearch( T, new.key, &pos ) ;
    if ( !sign ) //标记为0，是开地址
            T[pos]=new; //插入成功
    else
            if ( sign>0 ) printf("duplicate key") ; //重复，插入失败
            else  Error("overflow"); //表满，插入失败
 }
```

- **建表：**将表中keys清空，然后依次调用插入算法插入结点

散列表上的运算

3、删除

- 开地址法不能真正删除（置空），而应该置特定标记Deleted
  - 查找时：探测到Deleted标记时，继续探测
  - 插入时：探测到Deleted标记时，将其看作开地址，插入新结点
- 当有删除操作时，一般不用开地址法，而用拉链法

4、性能分析

只分析查找时间，插删取决于查找，因为冲突，仍需比较。

- 例子（见例1和例2图）
  - 成功

线性探测：ASL＝(1*6+2*2+3*1+9*1)/10=2.2 //n=10

拉链法： ASL＝(1*7+2*2+3*1)/10=1.4 //n=10

散列表上的运算

- 不成功: 查找不成功时对关键字需要执行的平均比较次数

线性探测：直到探测到开地址为止

若h(K)=0，则必须依次将T[0..8]中的关键字和K比较

ASL＝(9+8+7+6+5+4+3+2+1+1+2+1+10)/13=4.54 //m=13

	0	1	2	3	4	5	6	7	8	9	10	11	12
T[0..12]	26	12	41	15	68	44	06	51			36		38
探测次数	9	8	7	6	5	4	3	2	1	1	2	1	10

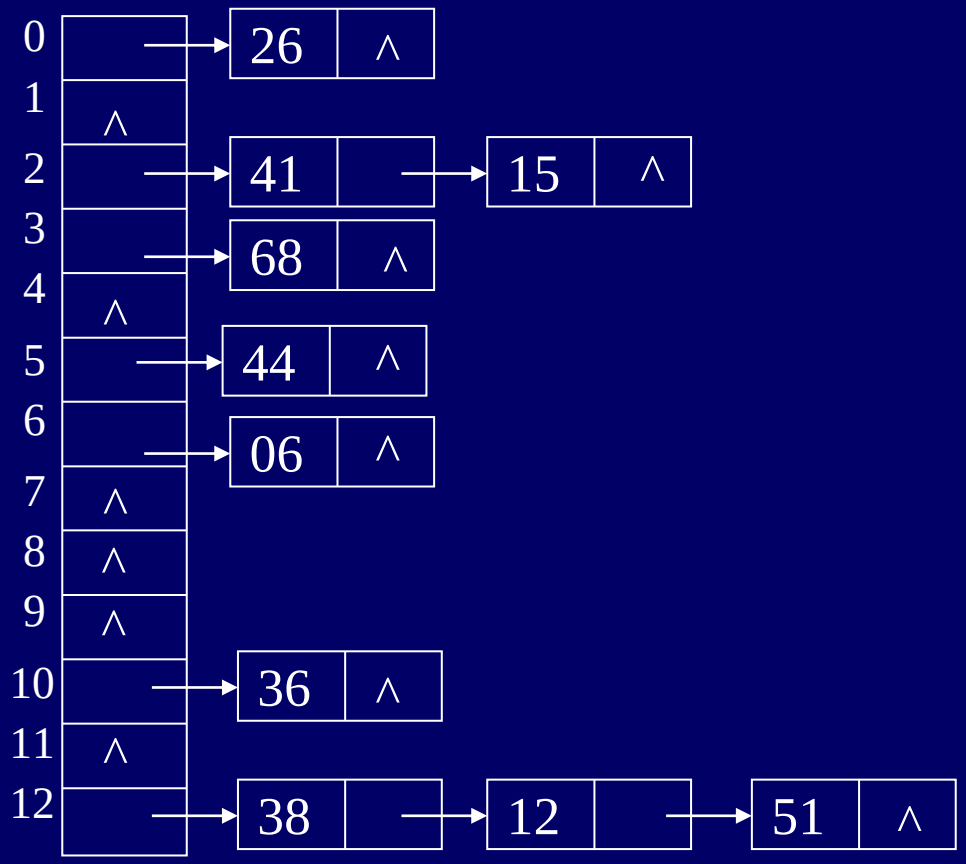
注意：当m>p（p＝|散列地址集|）时，分母应该为p

散列表上的运算

- 不成功

拉链法：不包括空指针判定

ASL＝(1+0+2+1+0+1+1+0+0+0+1+0+3)/13=0.77 //m=13



比较次数：　1　　2　　3

散列表上的运算

- 一般情况
  - 线性探测

    成功：ASL＝（1＋1/(1－α)）/2

    失败：ASL＝（1＋1/(1－α)^2）/2

    显然，只与α相关，只要α合适，ASL＝O(1)。
  - 其他方法优于线性探测：见教材P219

几点结论

- 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

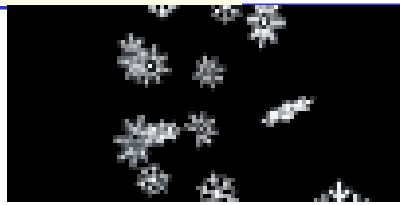
哈希表应用举例

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）

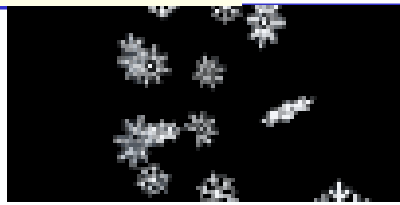
POJ 3349 Snowflake Snow Snowflakes



- ACM题目描述

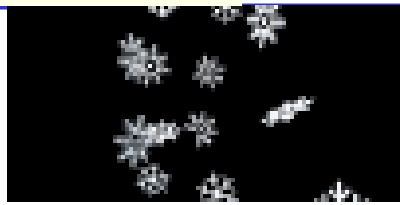
- 写一个程序判断是否有两片雪花是否相同。
- 每片雪都有六个角，每个角都有长度。如果两片雪花相对应的角的长度都相同，则认为这两片雪是相同的。

POJ 3349 Snowflake Snow Snowflakes



- **输入**
- 第一行为雪花数n(0<n<=100000)，接下来n行，每行描述一片雪花，包括六个整数，代表六个角的长度，其长度大于等于0，小于10000000，该六个整数都是按照时针顺序描述的，可以是顺序针，也可以是反时针，同时可以从任意位置开始，即1 2 3 4 5 6 和4 3 2 1 6 5可以认为是相同的雪花。

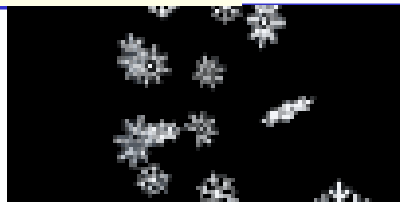
POJ 3349 Snowflake Snow Snowflakes



- 输出

- 如果所有的雪花都是不相同的，则输出
- “No two snowflakes are alike.”,
- 如果存在两片雪花是相同的，则输出
- “Twin snowflakes found.”。

POJ 3349 Snowflake Snow Snowflakes

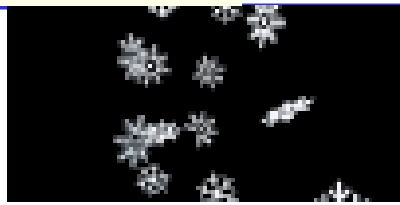


- **解题思路**

- 根据题意，雪花数最多可达100000片，而每两片雪花的比较次数最多可达12种，每种情况要比较6个数，如果采用逐个比较，则最多情况要求比较100000*100000*12*6=720000000000，即7200亿次，这显然是一个天大的数字。

- 每片雪花有六个角，将六个角的长度累加，得到一个代表该雪花的总长度。初步设想是将总长度相同的雪花放在一起，如果要判断某片雪花，只要和总长度与它相同的组中的每片雪花进行比较，如果都不相同，就将该雪花也放入该组。

POJ 3349 Snowflake Snow Snowflakes

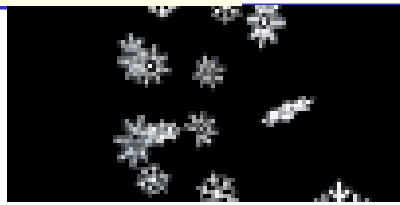


- **解题思路**

- 如果每个组都代表不同的总长度，由于雪花各个角的长度取值范围是0至10000000，则每片雪花的总长度取值范围达0至60000000，采用不同总长度设一个组，将要一个长度为60000000的数组，显然也是不合理的。

POJ 3349 Snowflake Snow Snowflakes

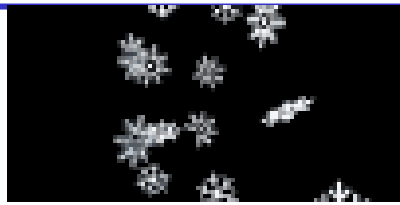
- **解题思路**



- 采用一种折中的方案，取一个自定义的素数，尽可能大，但不又超大，如99991，即定义一个99991个空间的数组，将各雪花总长度与99991取模，作为该雪花的存放组，而每一个组采用升序链表存储。该方法即采用**哈希查找**方法，**哈希函数为H(Key)=K%99991**。

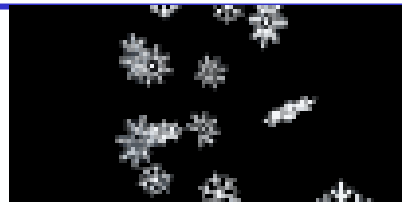
- 如果要判断某片雪花，先求得其总长度，并与99991取模，找到其所在的链表，在该链表找到总长与之相同的雪花进行比较，如果成功，直接输出并结束。如果总长度相同的全比较完而没有找到匹配的，则直接插入该处，以保持链表的升序。

参考代码



```
typedef struct node
{
    int arm[6];  //六个角的长度
    struct node *next;
    int sum;  //雪花六个角的总长度
}LNode;
```

参考代码



```
//判断是否有两片雪花相同，如果有，返回1，
否则返回0并插入该雪花
int isOk(LNode *p,int loc)
{
    用b数组存放p的排序结果
    while(链表中总长度不大于p的每片雪花q)
    {
        如果总长度不相同，直接比较下一个
        如果总长相同，则作如下操作
        用a存放q的排序结果
            如果a与b相同，则轮换进行比较，相同则返回1
            如果a与b不相同，则比较下一个
    }
    //如果前面没有返回，则插入该结点并保持有序
    return 0;
}
```

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