第8章
山重水复疑无路 柳暗花明又一村之查找

——刘亮亮

上海对外经贸大学
SHANGHAI UNIVERSITY OF INTERNATIONAL BUSINESS AND ECONOMICS

上章回顾

- 上章介绍了非线性数据结构：图
- 图是一种多对多的关系
- 图的基本概念：结点、度（出度与入度）、网、生成树…
- 图的存储结构与实现：邻接表表示法、邻接矩阵表示法、十字链表表示法
- 图的遍历：图的深度优先、图的广度优先遍历
- 图的最小生成树：克鲁斯卡尔算法、普里姆算法
- 图的最短路径： Dijkstra算法

本章要点

- **生活中总是在查找**
- **查找是一种非常重要、也是非常常见的一种算法：搜索引擎等**
- **静态查找法：**
  - 顺序查找
  - 折半查找
  - 分块查找
- **动态查找法：**
  - 二查排序树
  - 平衡二叉树
  - B+树与B-树
- **哈希查找**

目录

- 基本概念
- 静态查找算法
- 动态查找算法
- 哈希查找算法
- 小结

目录

基本概念

- 数据的组织和查找是大多数应用程序的核心，而查找是所有数据处理中最基本、最常用的操作。
- 查找(Searching)：根据给定的某个值，在查找表中确定一个其关键字等于给定值得数据元素（或记录）。

基本概念

- **查找表(Search Table)**：相同类型的数据元素(对象)组成的集合，每个元素通常由若干数据项构成。
- **关键字(Key，码)**：数据元素中某个(或几个)数据项的值，它可以标识一个数据元素。
  - 若关键字能唯一标识一个数据元素，则关键字称为**主关键字**；
  - 将能标识若干个数据元素的关键字称为**次关键字**。
- **查找/检索(Searching)**：根据给定的K值，在查找表中确定一个关键字等于给定值的记录或数据元素。
  - 查找表中存在满足条件的记录：查找成功；结果：所查到的记录信息或记录在查找表中的位置。
  - 查找表中不存在满足条件的记录：查找失败。

基本概念

- **静态查找(Static Search)**
  - 在查找时只对数据元素进行查询或检索，查找表称为静态查找表。
- **动态查找(Dynamic Search)**
  - 在查找的同时，插入查找表中不存在的记录，或从查找表中删除已存在的某个记录，查找表称为动态查找表。

目录

- 基本概念
- 静态查找算法
- 动态查找算法
- 哈希查找算法
- 小结

目录

- 基本概念
- 静态查找算法
- 动态查找算法
- 哈希查找算法
- 小结

顺序查找

- **顺序查找示例：**
  - 示例1：从散落的一堆图书中找一本书
  - 示例2：电脑里面找文件
  - 示例3：队列中找一个人
  - 。。。。
- **特点：**
  - 没有分类
  - 类似构成线性表
  - 只能逐个比较
- **又称为线性查找**

顺序查找

- **算法思想：**
  - 从表中的第一个（或最后一个）开始，逐个进行关键字的比较
    - ✓若相等，查找成功
    - ✓直到比较到最后一个，还不相等，查找失败

顺序查找

- **算法实现：**

```
//静态查找表的顺序存储结构
Typedef struct
{
    ElemType* elem;
    int length;
}SSTable;
```


```
/*
    顺序查找
    a-数组，n-数组长度，key-待查找的关键字
    查找成功返回序号，否则返回-1
*/
int Search_Seq(SSTable ST, ElemType key)
{
    for(int i=0;i<=ST.length;i++)
    {
        if(ST.elem[i].key==key)
        {
            return i;
        }
    }
    return -1;
}
```

顺序查找

- **算法分析：**
  - 平均查找长度(Average Search Length)：ASL
    - ✓确定记录在查找表中的位置，需要和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的平均查找程度。
  - 表长为n的表，查找每个记录的概率为P_i

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

$$\sum_{i=1}^{n} P_i = 1$$

查找成功的时间复杂度为O(n)

  - 则：$$ASL = nP_1 + (n-1)P_2 + \cdots + 2P_{n-1} + P_n$$
  - 等概率的情况下，$P_i=1/n$

$$ASL_{ss} = \sum_{i=1}^{n} P_i C_i = \frac{1}{n}\sum_{i=1}^{n}(n-i+1)$$

$$ASL_{SS} = \frac{n+1}{2}$$

顺序查找

- **算法优化：**
  - 每次都要越界判断，算法不够完美
  - 可以将数组的第0个作为“哨兵”

```
/*
    顺序查找
    a-数组，n-数组长度，key-待查找的关键字
    查找成功返回序号，否则返回0
*/
int Search_Seq(SSTable ST, ElemType key)
{
      SSTable.elem[0]=key;//哨兵
      for(i=ST.length; ST.elem[i].key!=key;--i);
      return i;
}
```

有序表查找

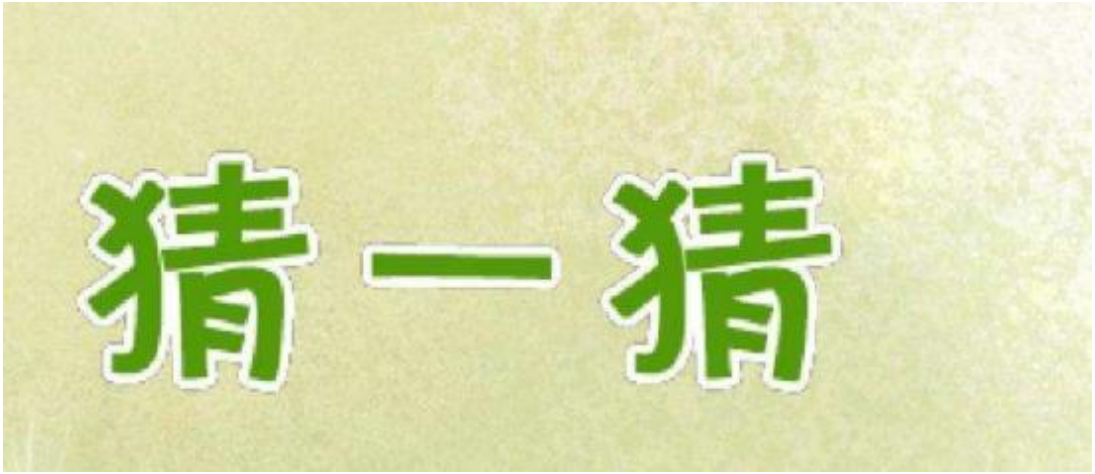
- **示例：**
  - 示例1：图书馆查找一本书
  - 示例2：查拼音字典

- **特点：**
  - 线性表是有序的

有序表查找

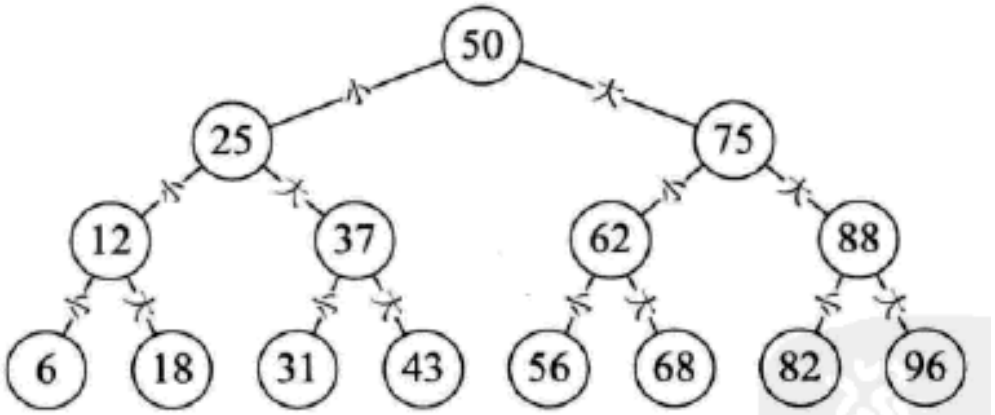
- **一个游戏：**

写了一个100以内的正整数，问你几次可以猜到？如何做？



有序表查找

- **折半查找(Binary Search)：又称为二分查找**
  - 每次取中间的元素进行比较，缩小查找范围
- **前提条件：**
  - 采用顺序存储结构
  - 线性表必须是按关键字有序



有序表查找

- **算法思想：**
  - （1）取中间元素进行比较，如果相等，查找成功

    mid = (low + high)/2
  - （2）若待查找的值小于中间元素，则在左半区进行查找；

    high = mid-1
  - （3）若待查找的值大于中间元素，则在右半区进行查找

    low = mid+1
  - （4）重复(1)(2)(3)，直到查找成功或查找失败

    low > high

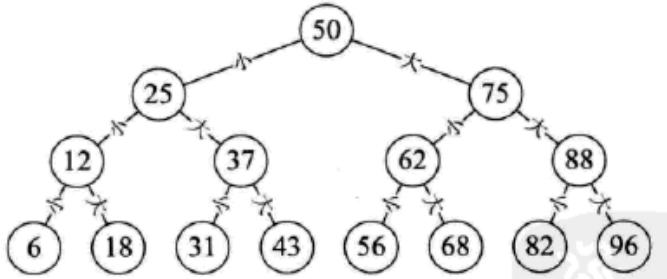
有序表查找

- 算法实现

```
/* 折半查找
*/
int Binary_Search(SSTable ST, ElemType key)
{
    low = 1;
    high = ST.length;
    while(low<=high)
    {
        mid = (low + high) /2;
        if (key ==ST.elem[mid].key)
        {
            return mid;
        }
        else if(key<ST.elem[mid].key)
        {
            high = mid -1;
        }
        else
        {
            low = mid +1;
        }
    }
    return 0;
}
```

有序表查找

- **算法分析：**
  - 查找过程可以用二叉树来描述
  - n个结点的表，判定树的深度为$[\log_2 n]+1$
  - 查找成功时候比较个数最多为$[\log_2 n]+1$
  - 平均查找长度是多少呢？
    - ✓假设有序表的长度$n=2^h-1$，则判定树深度为h



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

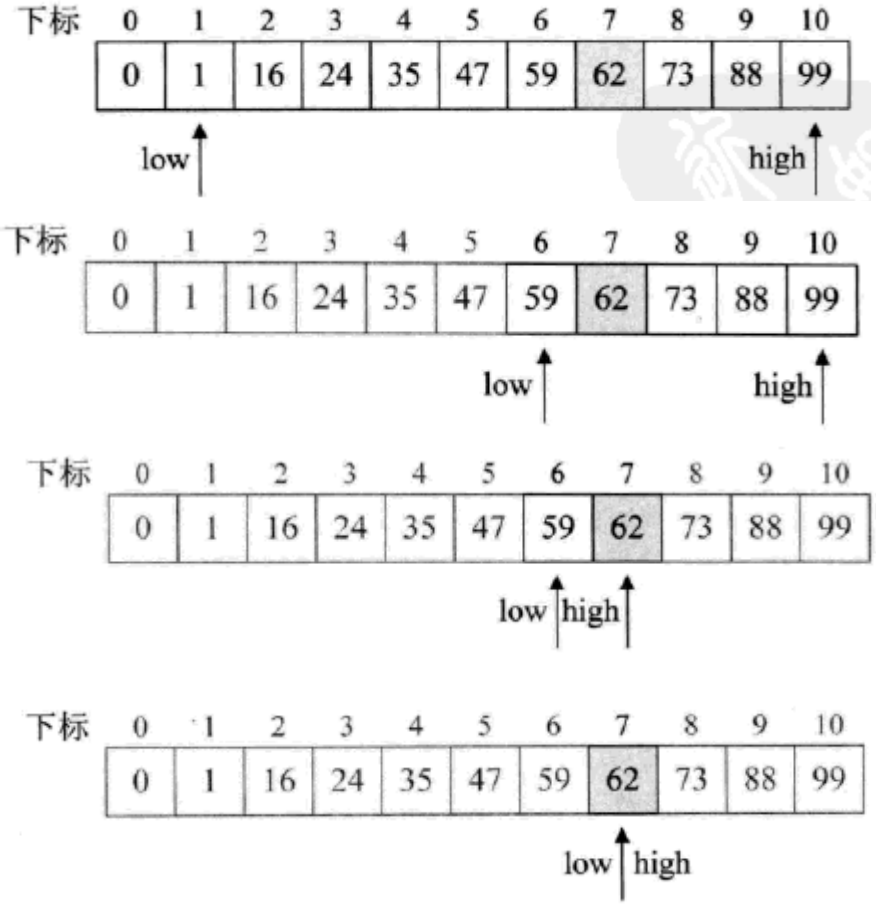
$$= \frac{1}{n}\sum_{j=1}^{h} j \cdot 2^{j-1}$$

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

当n较大时候(n>50)，ASL近似$\log_2(n+1)-1$。

有序表查找

- **图解折半查找：**



目录

- 基本概念
- 静态查找算法
- 动态查找算法
- 哈希查找算法
- 小结

目录

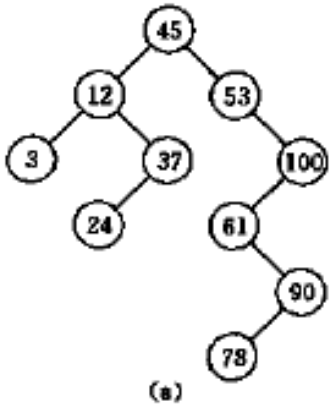
- 基本概念
- 静态查找算法
- 动态查找算法
- 哈希查找算法
- 小结

动态查找表

- **表结构本身是在查找过程中动态生成的**
- **给定值key：**
  - 若表中存在与key相等的记录，查找成功
  - 否则插入关键字等于key的记录
  - 也可以对关键字进行删除
- **算法：**
  - 二叉排序树
  - 平衡二叉树
  - B-树和B+树
  - 键树

二叉排序树

- **定义：或者是一个空树，或者是具有如下性质的二叉树：**
  - 若左子树不空，则左子树上所有结点的值均小于根结点的值；
  - 若右子树不空，则右子树上所有结点的值都大于根节点的值；
  - 左右子树也分别是二叉排序树



又称为
二叉查找树

二叉排序树

- **算法思想：**
  - 待查找的关键字和根结点的关键字比较
    - ✓若相等，则查找成功
    - ✓如果比根结点的关键字小，则在左子树上查找
    - ✓如果比根结点的关键字大，则在右子树上查找
  - 通常，采用二叉链表来存储二叉树

二叉排序树

- 算法实现

```
Status SearchBST(BiTree T, ElemType key, BiTree f, BiTree &p)
{
    if((!T))
    {
        p= f;
        return False;
    }
    else if(key==T->data.kety)
    {
        p = T;
        return TRUE;
    }
    else if (key<T->data.key)
    {
        SearchBST (T->lchild,key,T,p);
    }
    else
    {
        SearchBST (T->rchild, key,T, p);
    }
}
```

查找成功，p指向该节点，如果查找失败，指向查找路径上访问的最后一个结点并返回False。f指向T的双亲

二叉排序树

- **二叉排序树的插入**
  - 动态查找树
  - 特点：树的结构通常不是一次生成的，在查找的过程中，当树中不存在关键字等于给定值的节点时再进行插入。
  - 算法思想：
    - ✓查找不成功时候，将查找的节点插入在查找路径上最后访问的节点的左孩子或右孩子
    - ✓插入的节点一定是新的叶子节点
    - ✓插入后仍然保持有序

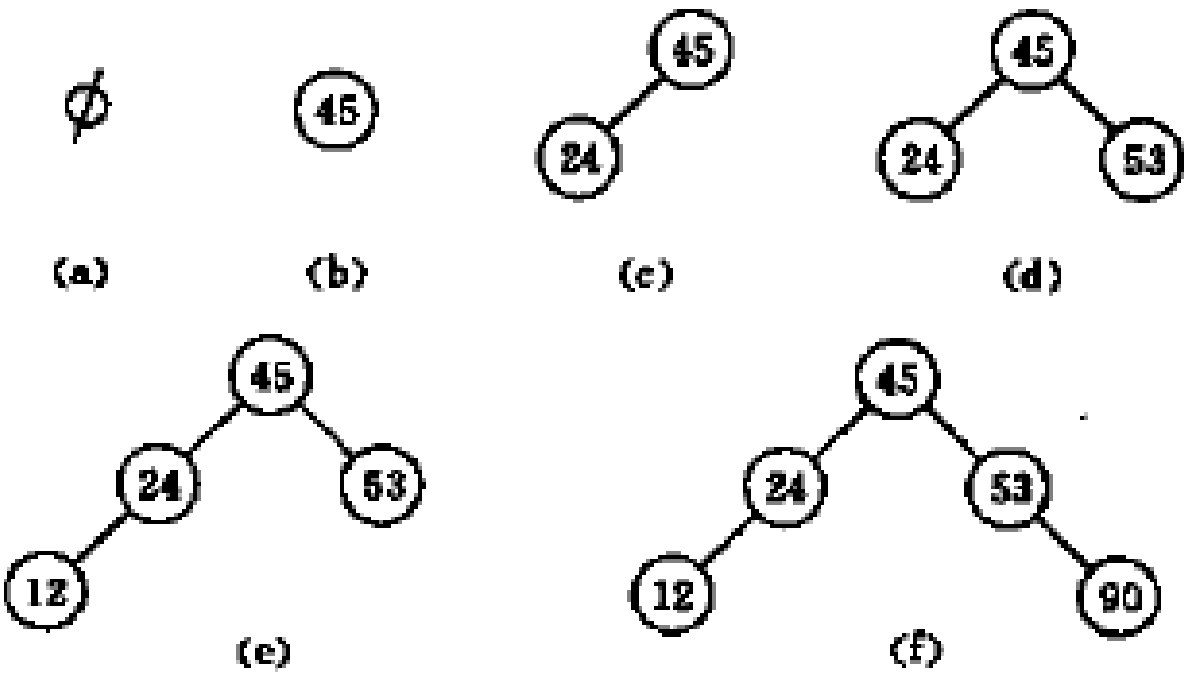
二叉排序树

- 算法描述：

```
Status Insert_BST (BiTree T, ElemType key, BiTree f, BiTree &p)
{
   if(!SearchBST(T, e.key, NULL, p))    //查找失败
   {
       s = (BiTree)malloc(sizeof(BiTNode));  //插入节点
       s->data =e;
       s->lchild = s->rchild = NULL;
       if(!p)                                  //当前树中没有结点，s作为根结点
           T= s;
       else if(e.key <p->data.key)   //插入为左子树
            p->lchlid = s;
       else
            p->rchild = s;                 //插入为右子树
       return TRUE;
    }
    else                                    //查找成功， 不再插入
    {
         return FALSE;
   }
}
```

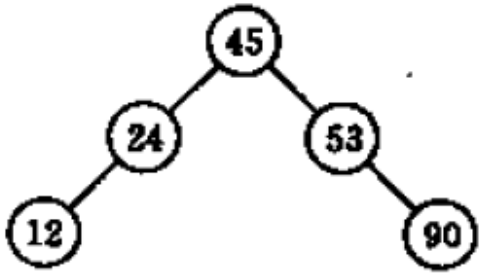
二叉排序树

- 示例：{45, 24, 53, 12, 24, 90}——二叉排序树构造过程



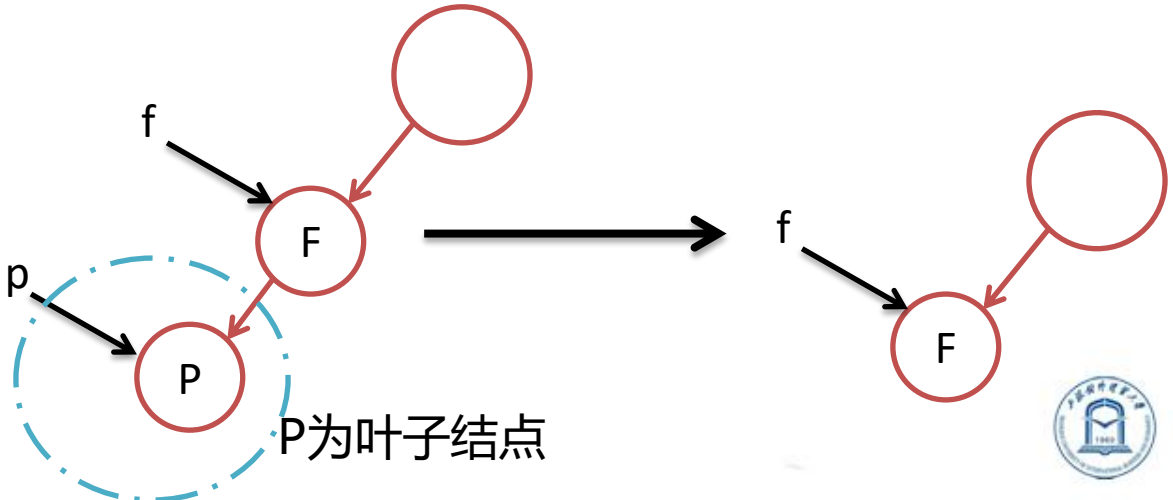
二叉排序树

- **二叉排序树的删除**
  - 中序遍历的二叉排序树得到一个关键字的有序序列
  - **一个无序序列通过构造一棵二叉排序树变成一个有序序列**
  - 删除二叉排序树的节点：
    - ✓删除二叉树的一个结点，会变成森林(左右子树)
    - ✓删除一个记录，要保持二叉排序树的特性。
  - 问题：**如何删除一个节点呢?**



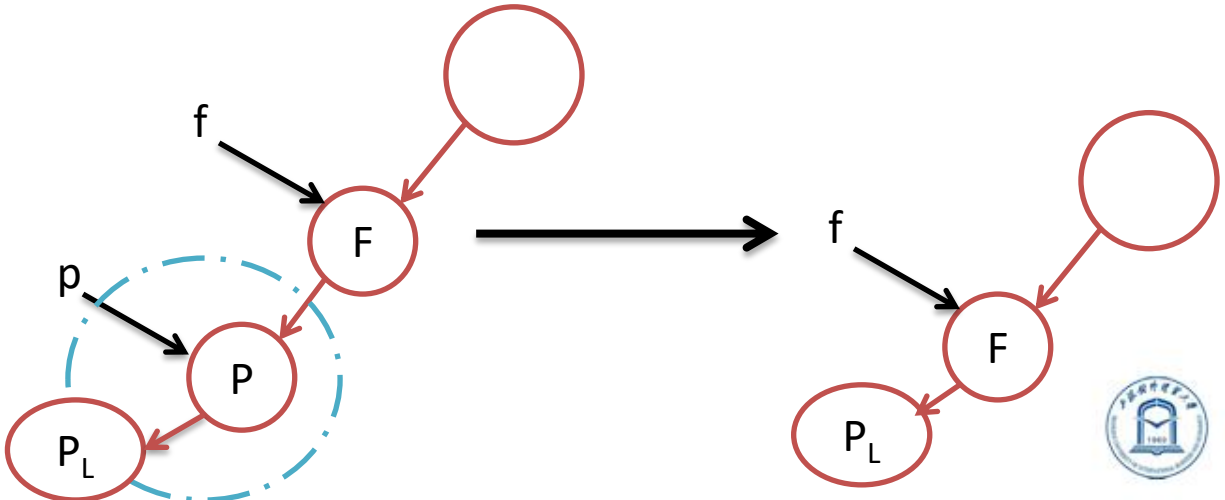
二叉排序树

- **二叉排序树的删除**
  - 假设在二叉树上被删除结点为*p，双亲为f，不失一般性，*p是*f的左孩子
  - **算法描述：**
    - ✓若*p结点为叶子结点，即P_L和P_R均为空树。删除叶子结点不破坏整棵树的结构，修改其双亲结点的指针。



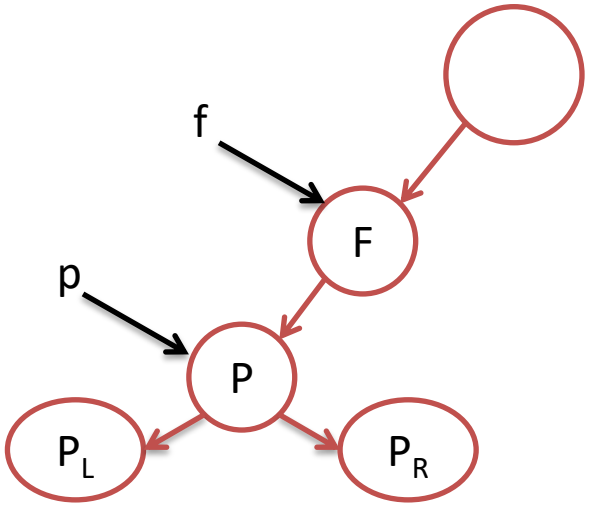
二叉排序树

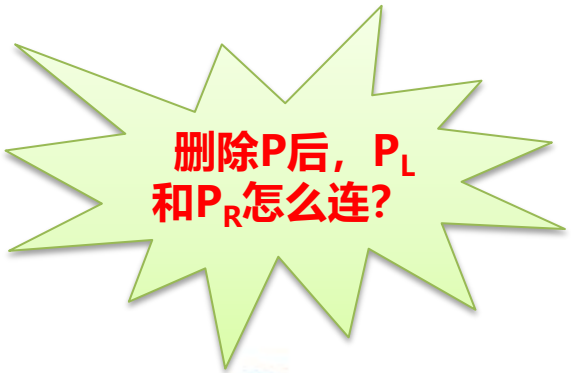
- **二叉排序树的删除**
  - **算法描述：**
    - ✓若*p结点为叶子结点，即P_L和P_R均为空树。删除叶子结点不破坏整棵树的结构，修改其双亲结点的指针。
    - ✓若*p结点只有左子树P_L或右子树P_R，只要将P_L或P_R直接成为其双亲结点*f的左子树即可以。



二叉排序树

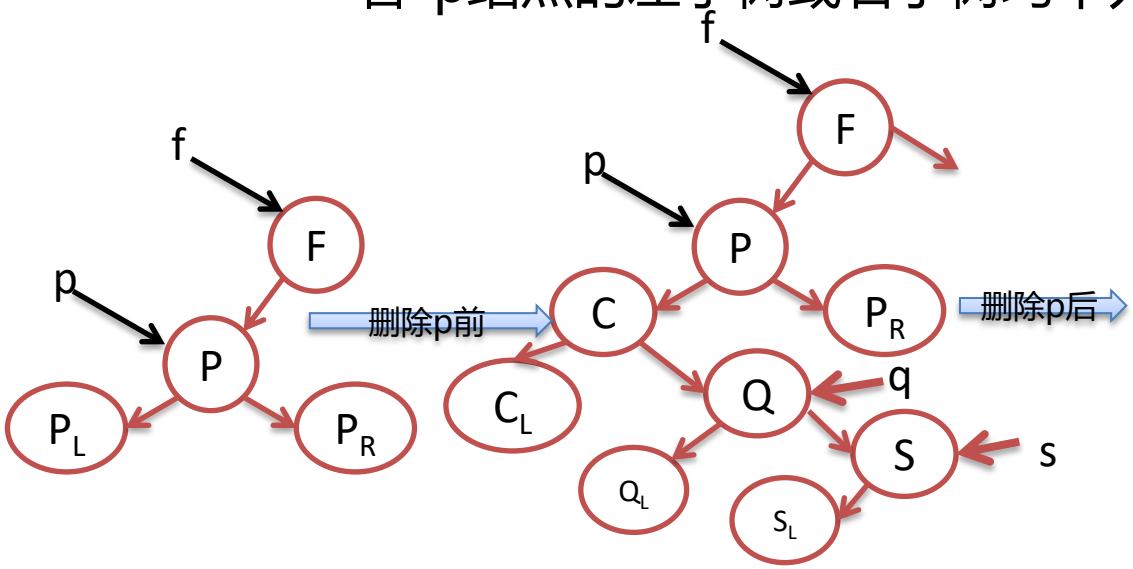
- **二叉排序树的删除**
  - **算法描述：**
    - ✓若*p结点为叶子结点，即P_L和P_R均为空树。删除叶子结点不破坏整棵树的结构，修改其双亲结点的指针。
    - ✓若*p结点只有左子树P_L或右子树P_R，只要将P_L或P_R直接成为其双亲结点*f的左子树即可以。
    - ✓若*p结点的左子树或右子树均不为空



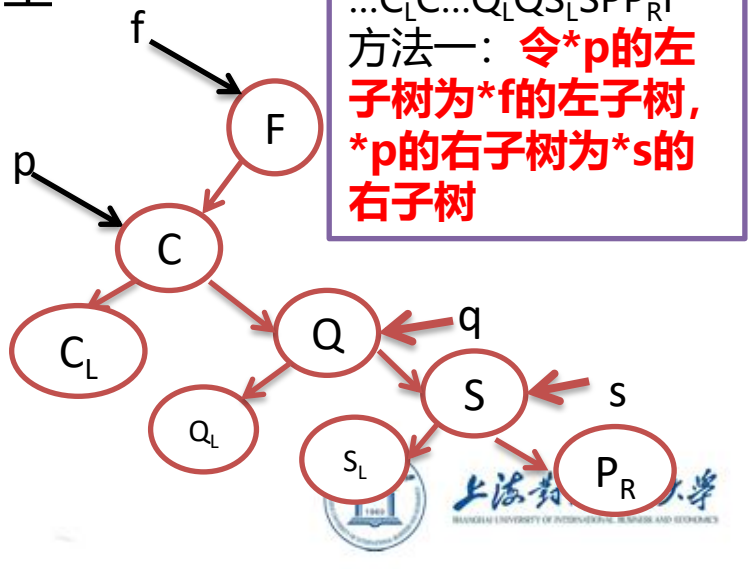


二叉排序树

- **二叉排序树的删除**
  - **算法描述：**
    - ✓若*p结点为叶子结点，即P_L和P_R均为空树。删除叶子结点不破坏整棵树的结构，修改其双亲结点的指针。
    - ✓若*p结点只有左子树P_L或右子树P_R，只要将P_L或P_R直接成为其双亲结点*f的左子树即可以。
    - ✓若*p结点的左子树或右子树均不为空



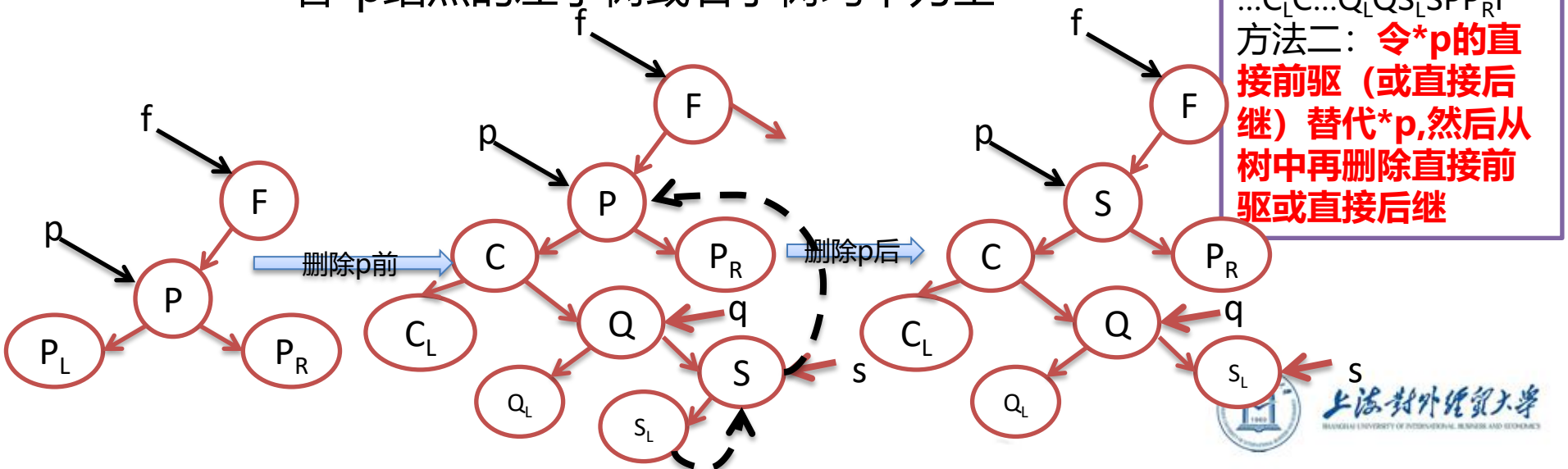
中序遍历序列：
…C_LC…Q_LQS_LSPP_RF
方法一：**令*p的左子树为*f的左子树，*p的右子树为*s的右子树**



二叉排序树

- **二叉排序树的删除**
  - **算法描述：**
    - ✓若*p结点为叶子结点，即P_L和P_R均为空树。删除叶子结点不破坏整棵树的结构，修改其双亲结点的指针。
    - ✓若*p结点只有左子树P_L或右子树P_R，只要将P_L或P_R直接成为其双亲结点*f的左子树即可以。
    - ✓若*p结点的左子树或右子树均不为空

中序遍历序列：
…C_LC…Q_LQS_LSPP$_R$F
方法二：**令*p的直接前驱（或直接后继）替代*p,然后从树中再删除直接前驱或直接后继**



二叉排序树

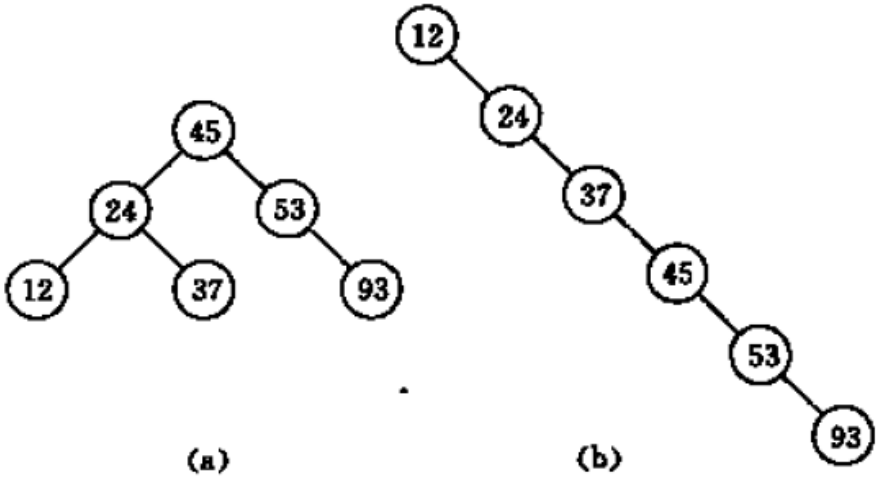
- 二叉排序树的删除

```
Status  DeleteBST(BiTree &T, ElemType key)
{
    if(!T) return FALSE;
    else
    {
        if(key==T->data.key)
        {
            Delete(T); //删除
        }
        else if(key<T->data.key)
        {
            DeleteBST(T->lchild, key);
        }
        else
        {
            DeleteBST(T->rchild,key);
        }
        return TRUE;
    }
}
```

```
//删除p，并重新连接它的左或右子树
void Delete (BiTree &p)
{
   if(!p->rchild)  //右子树为空直接连接左子树
   {
        q=p;
        p=p->lchild;
        free(q);
   }
   else if(!p->lchild) //左子树为空重新连接它的右子树
   {
       q=p;
       p=p->rchild;
       free(q);
   }
   else    //左右子树都都不为空
   {
      q=p;
      s=p->lchild;  //指向左子树
      while(s->rchild)  //转右，找最右边的，删除节点的前驱
      {
          q=s;
          s=s->rchild;
      }
      p->data=s->data;
      if(q!=p) q->rchild = s->lchild;  //重接*q的右子树
      else q->lchild=s->lchild;          //重接*q的左子树
   }
}
```

二叉排序树

- 二叉排序树的查找算法分析



$$ASL_{(a)} = \frac{1}{6}[1 + 2 + 2 + 3 + 3 + 3] = 14/6$$

$$ASL_{(b)} = \frac{1}{6}[1 + 2 + 3 + 4 + 5 + 6] = 21/6$$

二叉排序树

- **二叉排序树的查找算法分析**
  - 树的深度为n,其平均查找长度为(n+1)/2(和顺序查找相同)——单支树
  - 最好的情况是二叉排序树的形态和折半查找的判定树相同，其平均查找长度和$\log_2 n$成正比
  - 在随机的状态下，二叉排序树的平均查找长度和logn是等数量级

平衡二叉树

- **平衡二叉树**
  - 一种二叉排序树，其中每个节点的左右子树的高度之差至多等于1
  - 又称为**AVL树**
  - 二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF(Balance Factor)，平衡二叉树上所有结点的平衡因子只可能是-1,0和1

平衡二叉树

- 平衡二叉树

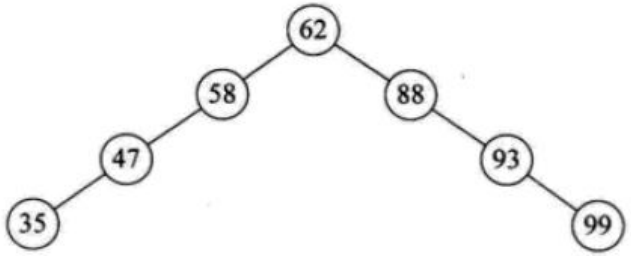


图1 不是平衡二叉树

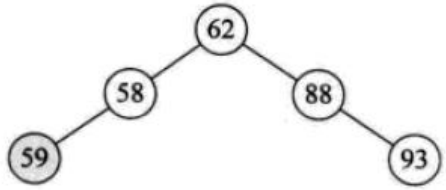


图2 不是平衡二叉树

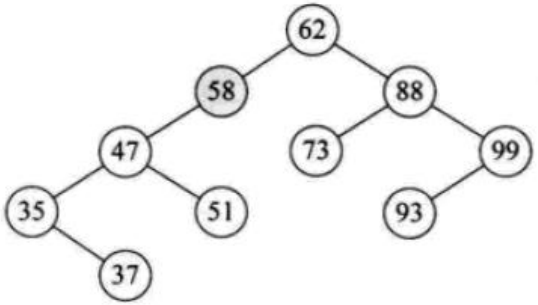


图3 不是平衡二叉树

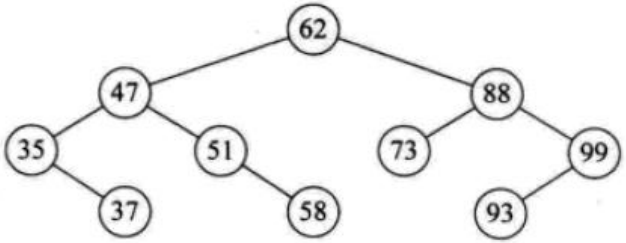
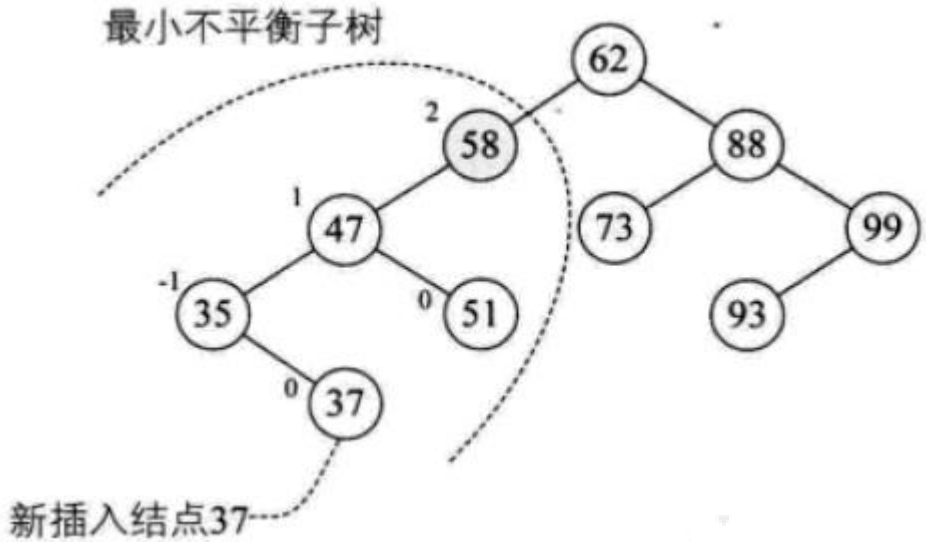


图4 平衡二叉树

平衡二叉树

- **平衡二叉树**
  - 距离插入结点最近的，且平衡因子的绝对值大于1的结点为根的子树，称为**最小不平衡子树**



平衡二叉树

- **构造原理——基本思想：**
  - 在构建二叉排序树的过程中，每当插入一个结点时候，检查是否因插入而破坏树的平衡性.
  - 若是，找出最小不平衡子树，在保持二叉排序树特性的前提下，调整最小不平衡子树中各结点之间的链接关系，进行旋转，使之成为新的平衡子树。

平衡二叉树

- 示例：{3,2,1,4,5,6,7,10,9,8}

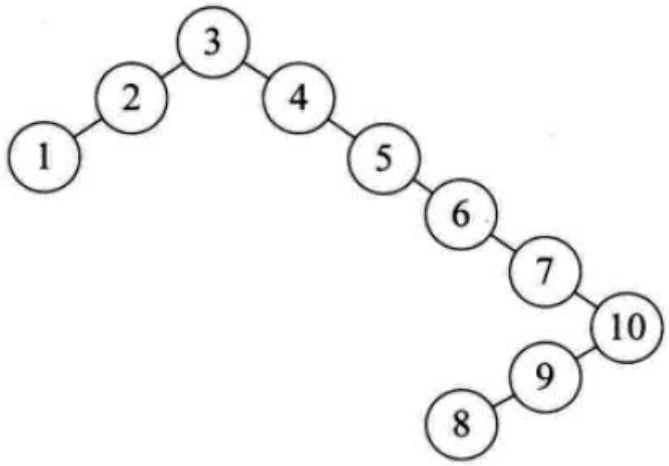


图1

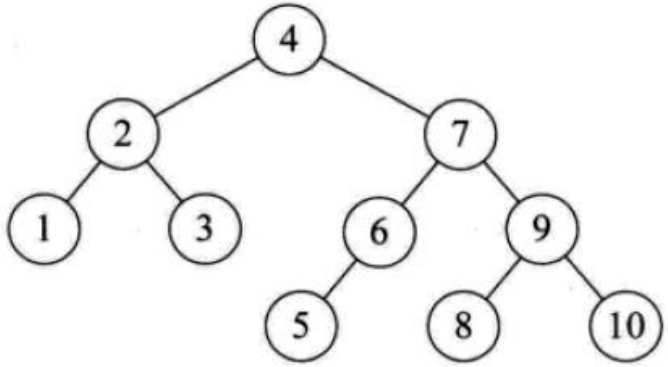


图2

深度不一样，查找效率也不一样

问题是：如何构造图2的二叉排序树?

平衡二叉树

- **示例：{3,2,1,4, 5,6,7,10,9,8}**
  - 插入3,2，正常构建
  - 插入1时候，发现3的平衡因子为2，整个树是最小不平衡子树，此时需要进行调整

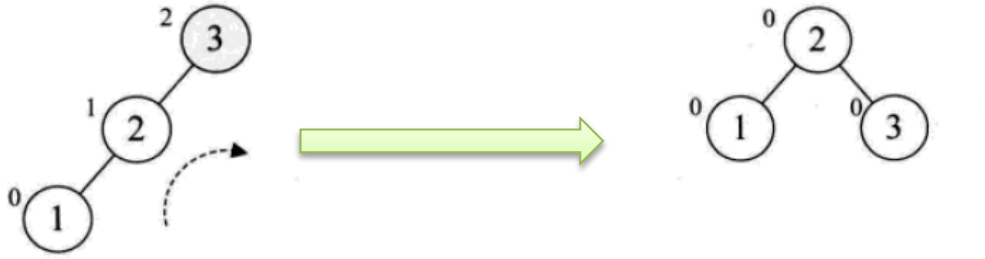


图1

图2

由于BF值为正数，只需要将整个树进行右旋（顺时针转）

平衡二叉树

- **示例：{3,2,1,4, 5,6,7,10,9,8}**
  - 插入4：没变化
  - 插入5：结点3的BF值变成-2,要旋转

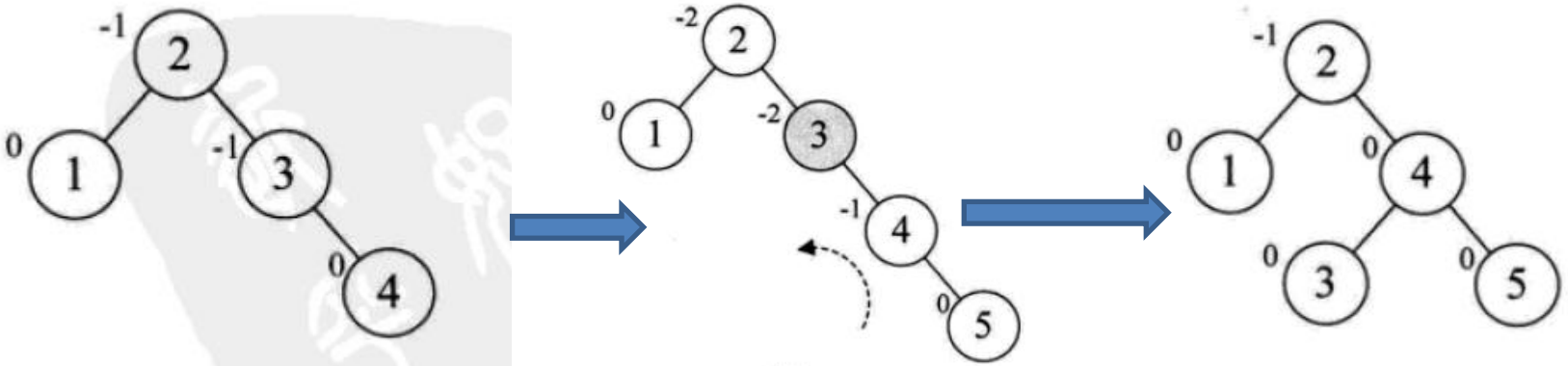


图4

图5

由于BF值为负数，只需要将这个子树进行左旋（逆时针转）

平衡二叉树

- **示例：{3,2,1,4,6,7,10,9,8}**
  - 插入6：根结点2的BF值变成-2

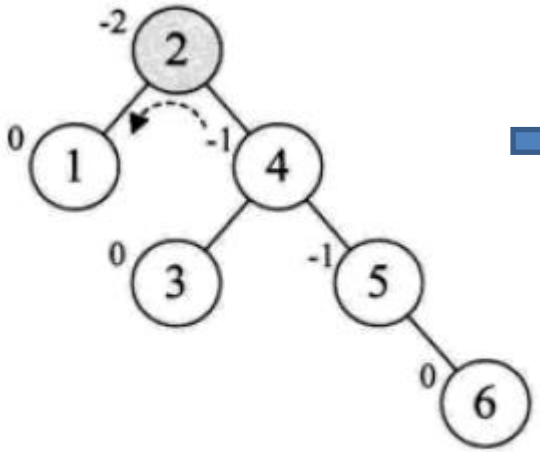


图6

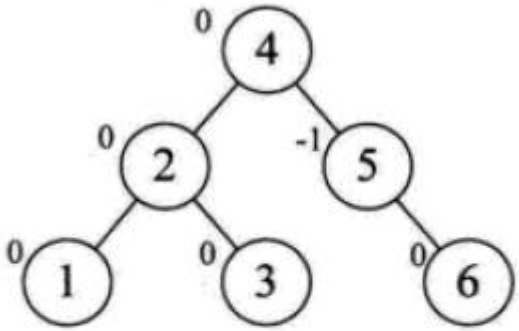


图7

由于BF值为负数，对根结点进行左旋（逆时针转）

平衡二叉树

- **示例：{3,2,1,4,6,7,10,9,8}**
  - 插入7：5的BF值变成-2，左旋5为根的子树

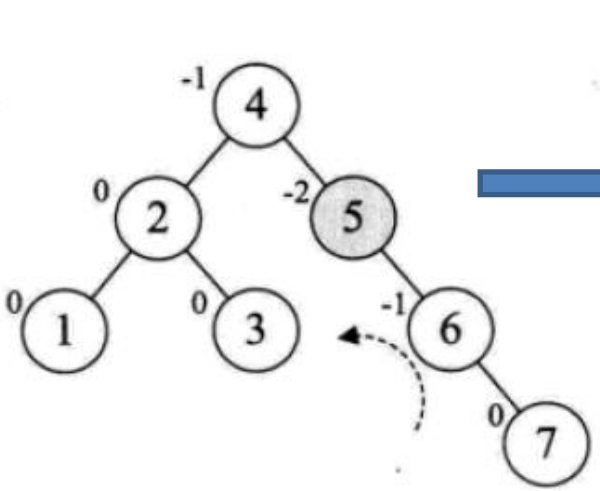


图8

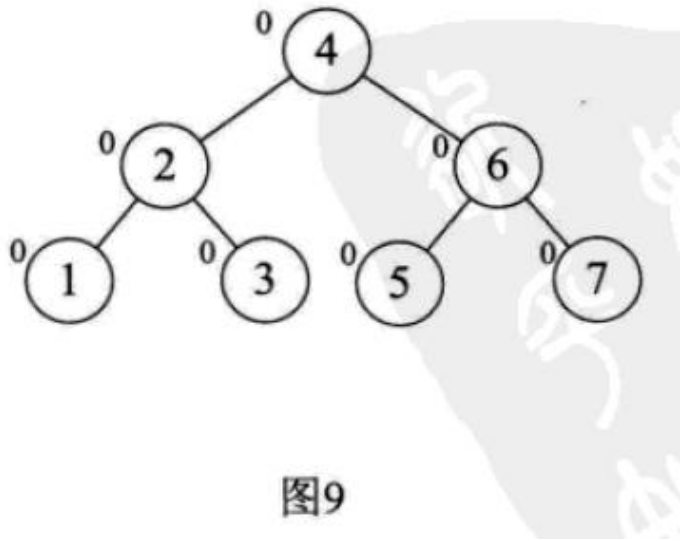


图9

平衡二叉树

- **示例：{3,2,1,4, 5,6,7,10,9,8}**
  - 增加10：结构无变化
  - 增加9：7的BF值为-2，只需要旋转最小不平衡子树7/9/10,但如果左旋转，9就成了10的右子树，不符合二叉排序树

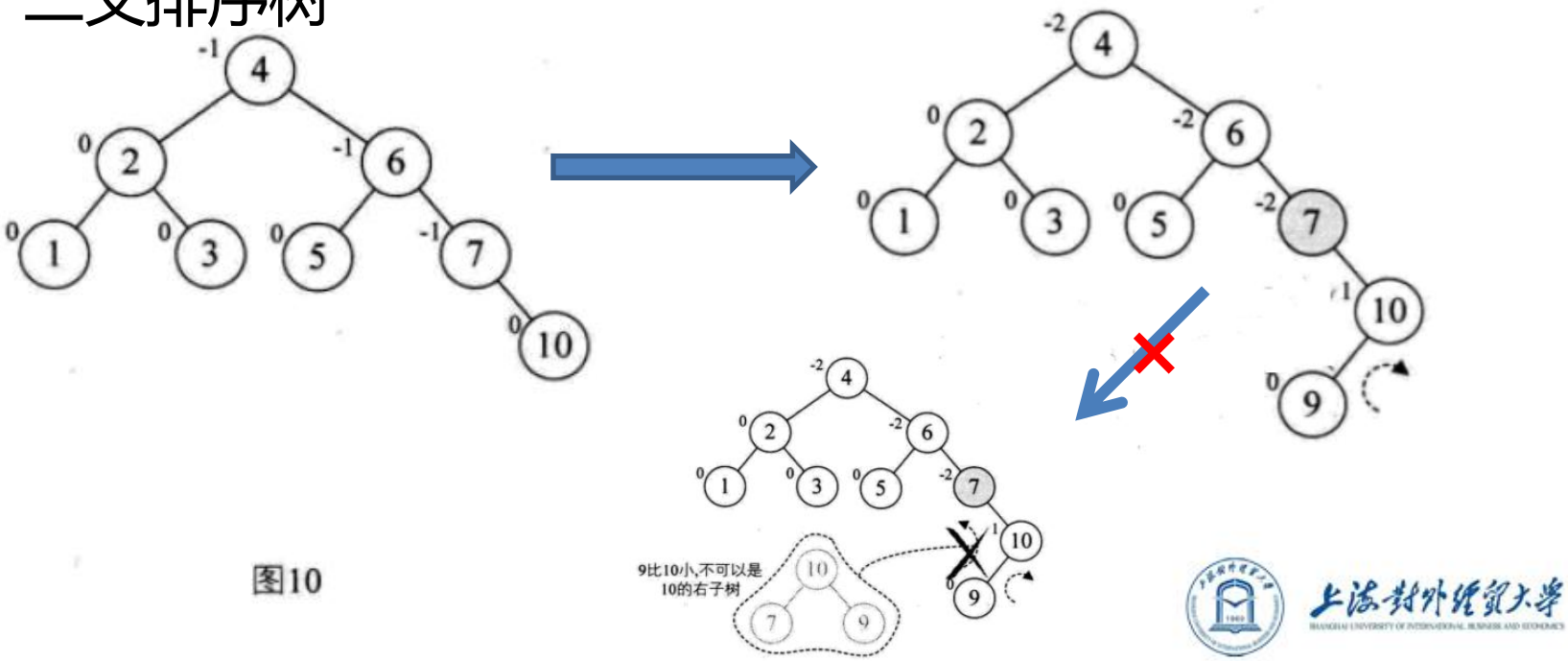


图10

平衡二叉树

- **示例：{3,2,1,4, 5,6,7,10,9,8}**
  - 增加9：不能简单的旋转，7的BF=-2，10的BF=1，这和前面的不同（前面和根节点符号一致），BF的符号不一致，此时（统一符号）：
  - 对9和10进行右旋,10成了9的BF,9的BF=-1和7一致了
  - 再对7为根的最小不平衡子树进行选择

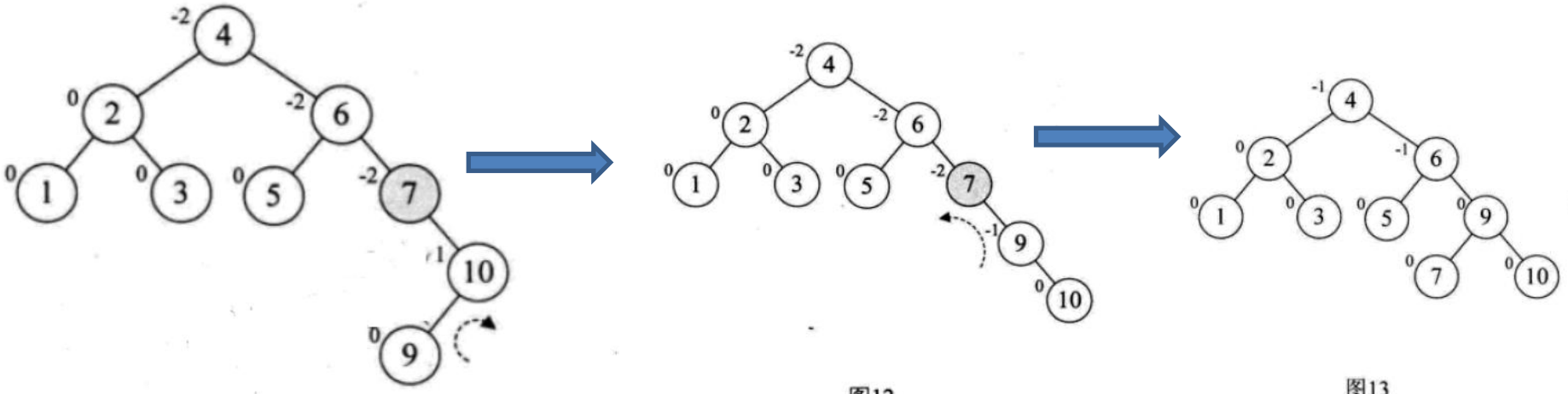


图12　　图13

平衡二叉树

- **示例：{3,2,1,4, 5,6,7,10,9,8}**
  - 增加8：和增加9类似

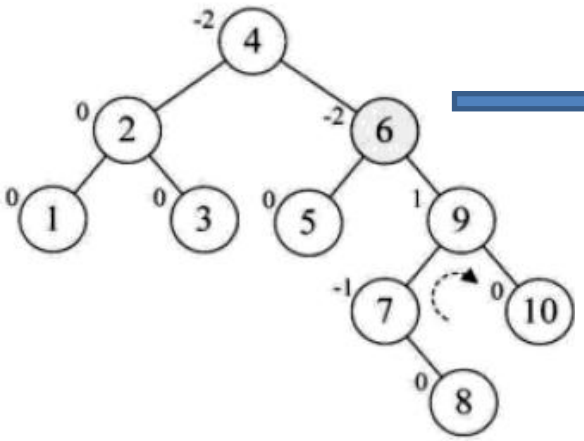


图14

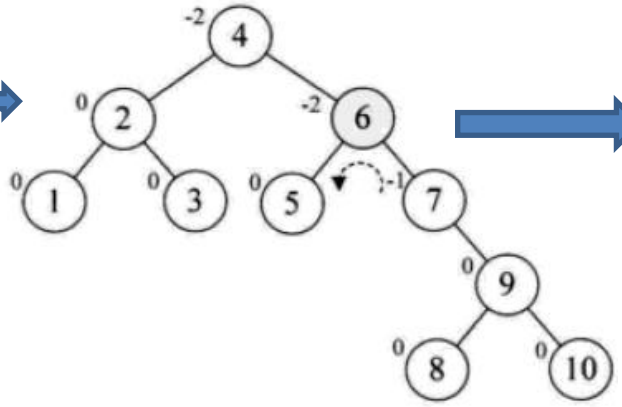


图15

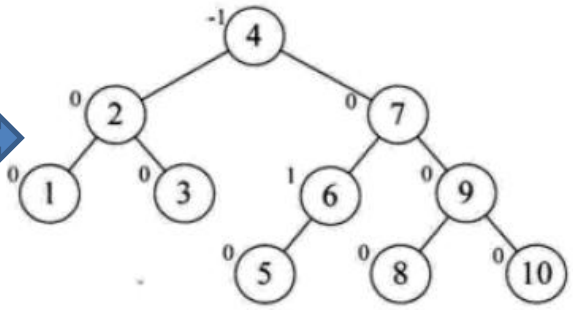
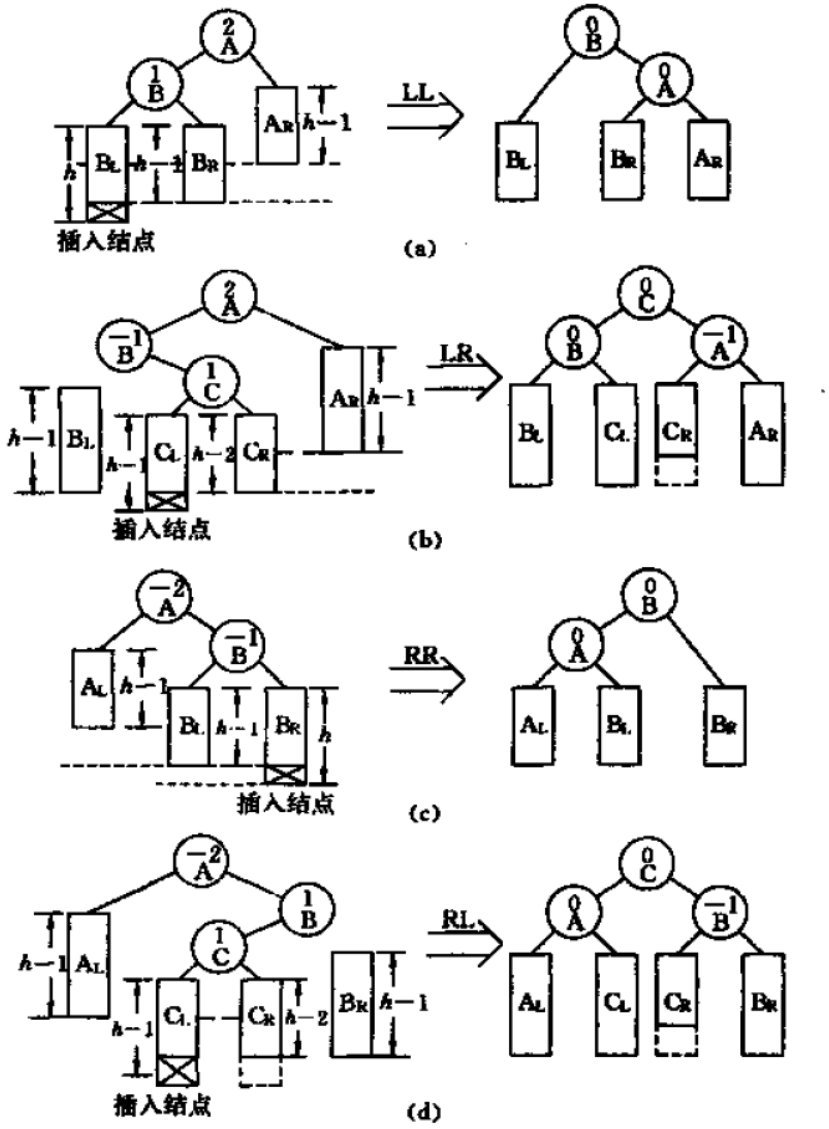


图16

平衡二叉树

- 二叉排序树的平衡旋转



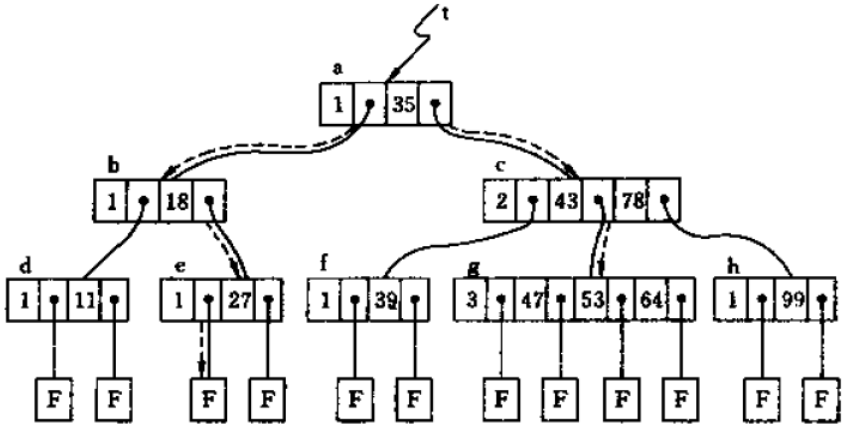
平衡二叉树

- **平衡二叉树的实现**
  - 请参考教材自行学习（略）

- **平衡二叉树的实现**
  - **平均查找的时间复杂度为O(logn)**

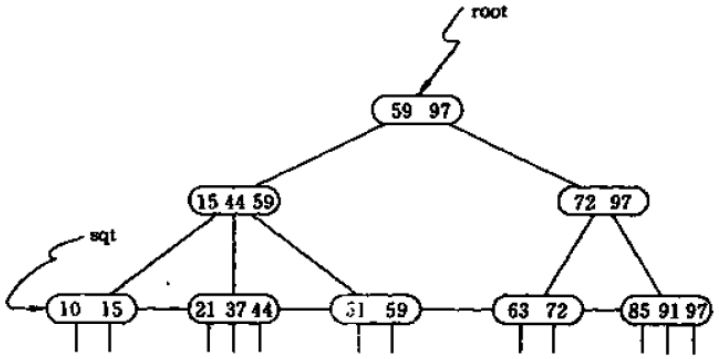
自学内容：其它动态查找树

- **B-树：**
  - 一种平衡的多路查找树，在文件系统中很有用
  - 是一棵m叉树
  - 满足一下特性：
  - 树中的每个结点至多有m棵子树
  - 若根节点不是叶子节点，则至少有两个子树
  - 除根结点之外，所有非终端节点至少有m/2棵子树
  - 所有的非终端节点包含：(n,A0,K1,A1,...Kn,An)
  - 所有的叶子结点在同一层次上



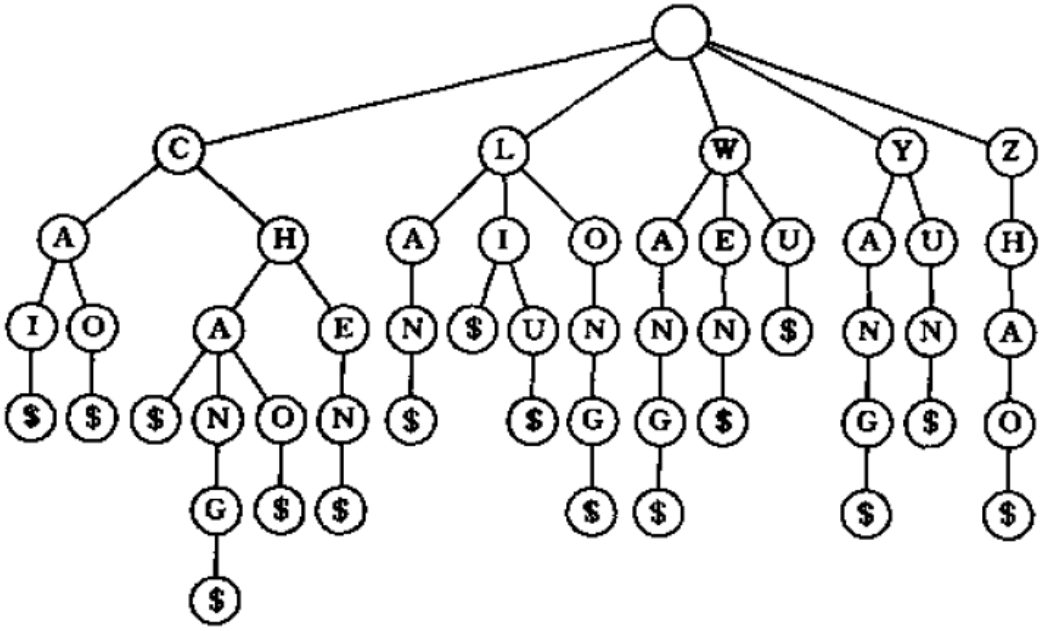
自学内容：其它动态查找树

- **B+树：**
  - B-树的一种变型
  - 有n棵子树的结点中含有n个关键字
  - 所有的叶子结点中包含了全部关键字的信息，及指向这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序连接
  - 所有的非终端节点可以看成是索引部分，节点仅含有子树（根结点）中的最大（或最小）关键字



自学内容：其它动态查找树

- **键树：**
  - 又称为数字查找树（字典树）
  - 一棵度>=2的树，树中的每一个结点不是包含一个或几个关键字，而是只包含组成关键字的符号。



目录

- 基本概念
- 静态查找算法
- 动态查找算法
- 哈希查找算法
- 小结

目录

- 基本概念
- 静态查找算法
- 动态查找算法
- 哈希查找算法
- 小结

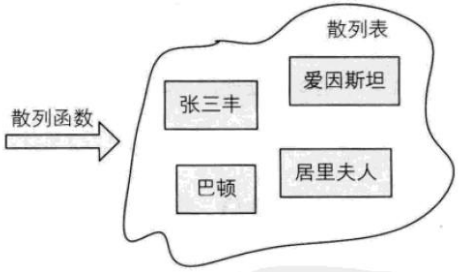
哈希查找算法

- **示例：去找某个老师?**
  - 找他(她)的办公室号
  - “**老师”➔“乐群楼205”（存储地址）➔映射f
  - 查找时，不需要比较，直接通过f计算，得到存储位置
  - 这种存储技术——散列技术
- **散列技术是在记录的存储位置和它的关键字之间确定对应关系f，使得每个关键字key对应一个存储位置f(key).**
- **f称为散列函数，又称为哈希(Hash)函数**
- **采用散列技术将记录在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表(Hash table)**

哈希查找算法

- **散列表查找步骤：**
  - 一是存储：通过散列函数计算记录的散列地址，按地址进行存放

编号	姓名	地址
1	张三丰	体育馆
2	爱因斯坦	图书馆
3	居里夫人	化学实验室
4	巴顿	网吧



  - 二是当查找记录，利用同样的散列函数计算记录的散列地址，按此散列地址访问记录。
- **因此，散列技术既是一种存储方法，也是一种查找方法。**
  - 散列技术最适合的求解问题是查找与给定值相等的记录

哈希查找算法

- **不是所有的关键字适合散列技术**
- **例如：**
  - 性别就不是，用“男”去查找，会查找出很多，显然不合适
  - 查找18-22岁的同学，在散列表中也没法进行
- **冲突：两个关键字key1!=key2，但f(key1)=f(key2)，这种现象称为冲突(collision)**
  - Key1和key2称为这个散列函数的同义词

哈希查找算法

- **散列函数的构造原则**
  - 计算简单
  - 散列函数分布均匀（冲突少）
- **构造方法**
  - 直接定制法：
    - ✓f(key)=key
    - ✓f(key)=a*key+b(a,b是常数)

用age作为关键字

地址	年龄	人数
00	0	500 万
01	1	600 万
02	2	450 万
……	……	……
20	20	1500 万
……	……	……

f(key)=key-1980

地址	出生年份	人数
00	1980	1500 万
01	1981	1600 万
02	1982	1300 万
……	……	……
2000	2000	800 万
……	……	……

哈希查找算法

- **构造方法**
  - **数字分析法**
    - ✓例如：

  - **平方取中法**
    - ✓例如：key=1234，key^2=1522756，取“227”作为地址
  - **除留余数法**：f(key) = key mod p(p<=m，m为表长)
  - **随机数法**：利用关键字的随机数作为散列地址
    - ✓f(key)=random(key)

哈希查找算法

- **不同的情况应采用不同的散列函数，考虑的因素**
  - 计算散列地址所需的时间
  - 关键字的长度
  - 散列表的大小
  - 关键字的分布情况
  - 记录查找的频率

哈希查找算法

- **处理散列冲突的方法：**
  - 方法一：开放地址法——一旦冲突，选择下一个空的地址

$$f_i(key) = (f(key) + d_i) \ MOD \ m \quad (d_i = 1,2,3,\cdots\cdots,m-1)$$

  - 例如：{12,67,56,25,37,22,29,15,47,48,34},表长12

利用散列函数f(key)=key mod 12

✓计算12,67,56,16,25，计算地址，没有冲突，直接存入

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25			16			67	56			

✓计算key=37，f(key)=1，与25冲突

✓利用f(37)=(f(37)+1) mod 12=2，没有冲突，存入2号位置

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25	37		16			67	56		22	

上海对外经贸大学

哈希查找算法

- **处理散列冲突的方法：**
  - 方法一：开放地址法——一旦冲突，选择下一个空的地址
  - 例如：{12,67,56,25,37,22,29,15,47,48,34}
  - 存入22,29,15,47，没有冲突，存入

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25	37	15	16	29		67	56		22	47

  - key=48: f(48)=0冲突，f(48)=(f(48)+1) mod 12 =1 冲突,f(48)=(f(48)+2)mod12=2还是冲突，....f(48)=(f(48)+6) mod 12=6，没冲突，存入
  - 这种解决冲突的开放地址法称为**线性探测法**。

哈希查找算法

- **处理散列冲突的方法：**
  - 其它方法：
    - ✓再散列函数法
    - ✓链地址法
    - ✓公共溢出区法
- **算法实现：**

```
/* 散列表查找关键字 */
Status SearchHash(HashTable H,int key,int *addr)
{
    *addr = Hash(key);              /* 求散列地址 */
    while(H.elem[*addr] != key)     /* 如果不为空，则冲突 */
    {
        *addr = (*addr+1) % m;      /* 开放定址法的线性探测 */
        if (H.elem[*addr] == NULLKEY || *addr == Hash(key))
        {/* 如果循环回到原点 */
            return UNSUCCESS;       /* 则说明关键字不存在 */
        }
    }
    return SUCCESS;
}
```

算法分析：
O(1)

小结

- **静态查找：**
  - 顺序查找
  - 折半查找
- **动态查找：**
  - 二叉排序树
  - 平衡二叉树
  - B树
  - 哈希查找

算法比较

一、顺序查找

条件：无序或有序队列。

原理：按顺序比较每个元素，直到找到关键字为止。

时间复杂度：O(n)

二、二分查找（折半查找）

条件：有序数组

原理：查找过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；

如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。

如果在某一步骤数组为空，则代表找不到。

这种搜索算法每一次比较都使搜索范围缩小一半。

时间复杂度：O(logn)

算法比较

三、二叉排序树查找

条件：先创建二叉排序树：

1. 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
2. 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
3. 它的左、右子树也分别为二叉排序树。

原理：

在二叉查找树 b 中查找 x 的过程为：

1. 若 b 是空树，则搜索失败，否则：
2. 若 x 等于 b 的根节点的数据域之值，则查找成功；否则：
3. 若 x 小于 b 的根节点的数据域之值，则搜索左子树；否则：
4. *查找右子树。*

时间复杂度：$O(\log_2(n))$

算法比较

四、哈希表法（散列表）

条件：先创建哈希表（**散列表**）

原理：根据键值方式(Key value)进行查找，通过散列函数，定位数据元素。

时间复杂度：几乎是 O(1)，取决于产生冲突的多少。

五、分块查找

原理：将 n 个数据元素"按块有序"划分为 m 块（m ≤ n）。

每一块中的结点不必有序，但块与块之间必须"按块有序"；即第1块中任一元素的关键字都必须小于第2块中任一元素的关键字；

而第2块中任一元素又都必须小于第3块中的任一元素，……。

然后使用二分查找及顺序查找。