
# 数据结构

## Data Structure

张先宜

手机：18056307221 13909696718

邮箱：zxianyi@163.com

QQ：702190939

QQ群：XC数据结构交流群 275437164

# 第7章 查找

# 7.1 概述

- **查找是现实生活和软件设计中最常用的运算，例如，**
  - 查字典、辞典、电话号码；
  - 高考成绩查询，考试成绩查询；
  - 数据库中的各种查询；
  - **Google、Baidu…**
- **因此，查找方法的性能会影响到软件系统的性能。**
- **查找涉及到的几个相关问题：**
  - 如何设计查找方法？
  - 如何评价查找性能？
  - 如何组织数据，以便能更好地提高查找的性能？

- **本章主要内容：**
  - 介绍查找的相关概念。
  - 围绕几种常见的数据表，讨论查找的方法，并分析其性能。

- 查找（搜索）-- 在数据集（表）中找出一个特定元素（的位置）。

- 查找涉及的基本概念和相关问题：

## 1. 查找表

- 同类型的数据元素（记录）所构成的集合。常见的查找表有：
  - 线性表：数据元素构成一个线性序列。
  - 树　表：以树结构的形式组织。
  - 散列表：以某种函数的方式来确定元素的地址，实现数据表的组织。
  - 索引表：为元素建立索引，以提高查找的速度。

## 2. 关键字(Key Word)

- 数据元素（记录）中可以标识数据元素（记录）的数据项（字段）称为关键字。

- 关键字分为两种：

① 主关键字：唯一标识一个元素的字段。

  - 例如，高考成绩信息中的“准考证号”字段。

② 次关键字：可能标识到多个元素的字段。

  - 例如，高考成绩信息中的“考生姓名”字段，
  - 因为现实中有太多同名同姓的人。

- 查找的过程即对给定的值与关键字进行比较的过程。对给定的数据集，找到两者相等的记录称查找成功，否则查找失败。

## 3. 静态查找表和动态查找表

- 静态查找表 — 表中数据元素（记录）保持不变，查找中不涉及元素的插入、删除等操作。
- 动态查找表 – 表中数据元素（记录）是动态变化的，查找过程中伴随着元素的插入和删除操作等。

# 4. 查找长度（Search Length）

- 查找过程中所做的关键字的比较次数。

① 平均查找长度ASL（ Average Search Length ）

- 查找成功需要的平均比较次数。

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

- 假设查找表有n个记录，则：$\sum_{i=1}^{n} P_i = 1$

- 其中：$P_i$为查找表中第i个记录的查找概率，且
- $C_i$为成功查找第i个元素的比较次数。

② 最大（最坏）查找长度

③ 失败查找长度

- **5. 数据元素（记录）类型的定义**
  - 结构类型，或者类
  - 增加关键字字段

【元素（记录）类型描述】

```
typedef struct
{
        KeyType key;            //关键字域
        InfoType otherInfo;     //其它信息域

}elementType;
```

## 【一个元素类型实例】

```
typedef struct
{
        string sID;          //学号，作为主关键字
        string sName;        //其它信息域
        int sAge;
        int sGender;
          //…，可能还有其他字段
}student;
```

临渊羡鱼，不如退而结网。

# 7.2 线性表的查找

- **问题描述：**
  - 查找表为线性表，可能是顺序表，也可能是链表。
    - **典型地是一维数组；也可能是链表；或顺序文件。**
    - **本章更多的讨论是面向一维数组。**
  - 要求在此表中找出关键字的值为x的元素（记录）的位置，
    - **若查找成功，则返回其位置（数组下标，或结点指针），**
    - **否则，返回一个表示元素不存在的下标（如0、-1或空指针）。**

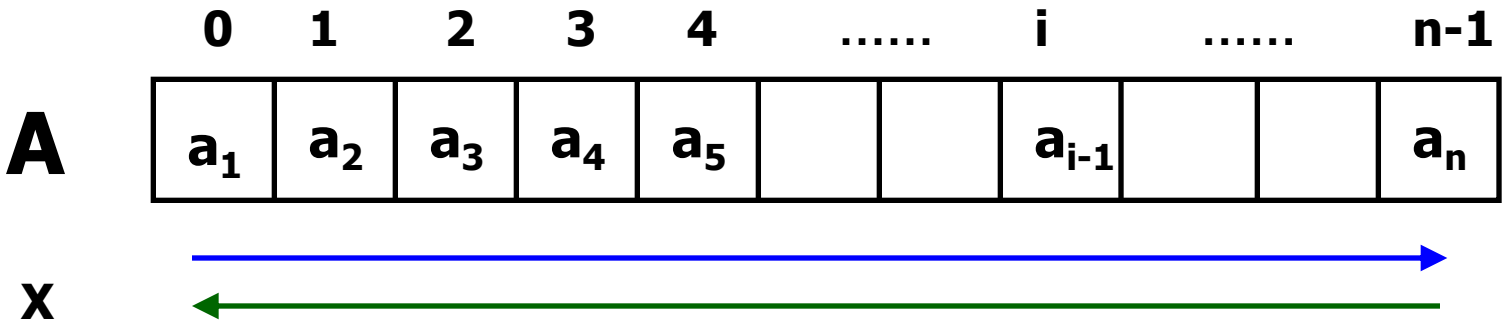
- **按照查找表中数据的特性，以及对应的查找方法，可分为以下几种：**
  - 顺序查找 -- 没有任何关于数据元素分布的特性
  - 有序表查找 -- 二分查找（折半查找）
  - 索引表查找 -- 数据表分块有序

# 7.2.1 顺序查找

## 1. 问题描述:

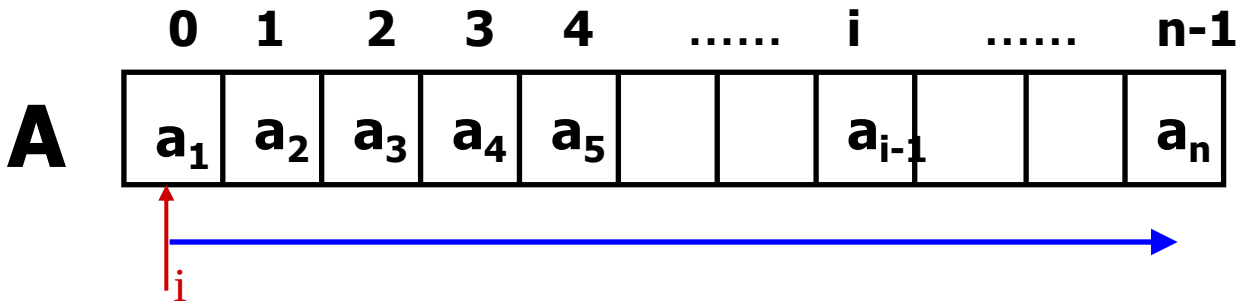
- 在数组A[n]中查找关键字为x的元素（记录），
- 若查找成功，则返回元素的下标，否则返回-1。

- **分析：**
  - 显然只能依次搜索（比较元素）。
- **问题：**
  - 按照什么样的查找顺序？即从前往后还是从后往前？



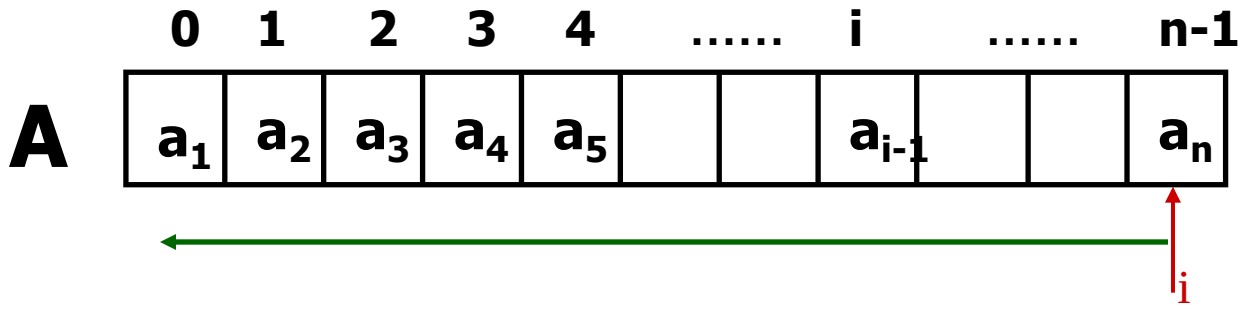
## 2.【算法描述—从左往右】

```
int  search ( elementType A[]，keyType x )
{   int i;
    for(i=0;i<n;i++)      //i为数组下标
        if (A[i].key==x)  return  i;   //查找成功
    return  -1;                        //查找失败
}
```



# 3.【算法描述—从右往左】

```
int  search ( elementType A[]，keyType x )
{   int i;
    for(i=n-1;i>=0;i--)                    //i为数组下标
        if (A[i].key==x)  return  i;       //查找成功
    return  -1;                            //查找失败
}
```



## 4. 设置监视哨

- A[0]单元存放搜索关键字x，为监视哨。

【有监视哨的从右往左搜索】

```
int  search ( elementType A[]，keyType x )
{   int i;
    A[0].key=x;
    for( i=n; A[i].key!=x; i-- ) ;   //本句独立运行
          //i>=1时，A[i].key==x，找到x，返回i
          //否则，失败，但i=0时一定有A[0].key==x
    return  i;   //查找成功，返回i；查找失败，返回0
}
```

- 设置监视哨的好处
  - 省去了每次循环中检查数组下标是否越界，即检查（比较）**i>=0**。
  - 实践证明，当**n>1000**时，设置监视哨平均查找时间会减少一半。
- 【思考问题】
  - 监视哨是否可以设置在高下标处（比如**A[n]**处）？

  【答】可以，从左往右搜索。

## 5. 算法分析

① 时间复杂度：**O(n)**，同线性表搜索

② 空间复杂度：**O(1)**

③ 平均查找长度：$ASL = \sum_{i=1}^{n} P_i C_i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$

④ 最大查找长度：**n**

⑤ 失败查找长度：**n+1**

## 6. 适用存储结构

- 顺序结构（数组）
- 链式结构

单选题 1分

设置

下面（　　）算法适合构造一个稠密图G的最小生成树。

A．Prim算法　　B．Kruskal算法
C．Floyd算法　　D．Dijkstra算法

A Prim算法

B Kruskal算法

C Floyd算法

D Dijkstra算法

提交

# 7.2.2 有序表的二分查找

- 也称：折半查找（**Binary Search**）

## 1. 问题描述

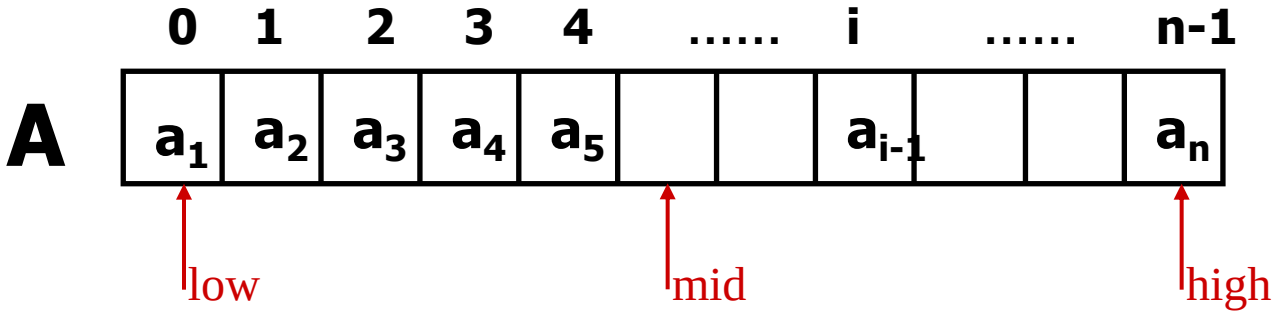
- 专门针对有序顺序表。不妨设为递增有序顺序表。

■ **【分析】**

- 用什么方法查找呢？
  - **简单的顺序查找当然可以，但没有利用“有序”的条件。**
  - **折半查找！**

## 2. 折半查找方法描述

- 设查找区域的首尾下标分别为 **low** 和 **high**（初值分别为0和n-1）。
- 中间元素的下标 **mid=(low+high)/2**

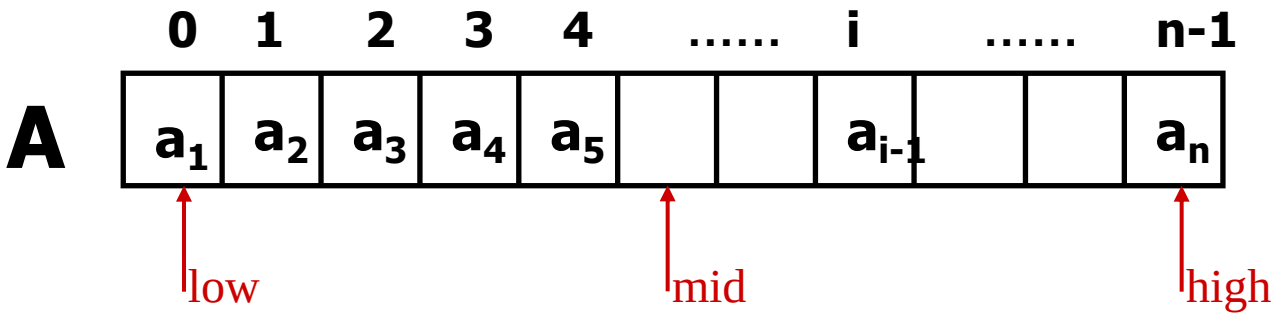


- 将待查关键字x和比较区域的中间元素 (其下标mid=(low+high)/2) 的关键字进行比较，并根据比较的结果分别作如下处理：

① x==A[mid].key: 查找成功，返回mid的值。

② x<A[mid].key: 说明待查元素只可能在左边区域，应在此区域继续查找，更新查找区域：

- low不变
- high=mid-1
- mid=(low+high)/2



③ **x>A[mid].key**: 说明待查元素只可能在右边区域，应在此区域继续查找，更新查找区域：

- **low=mid+1**
- **high**不变
- **mid=(low+high)/2**

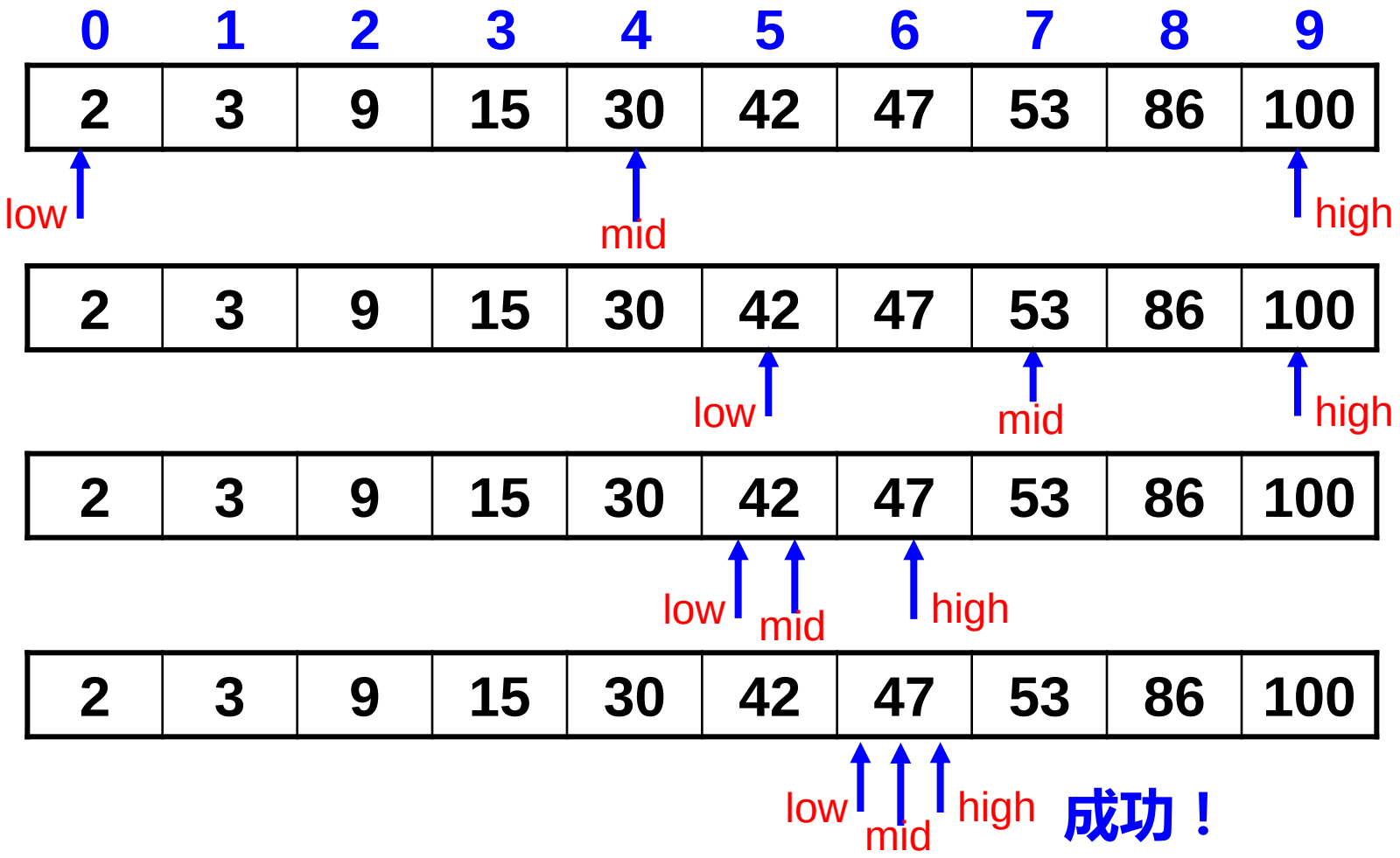
☞ 若表中存在所要查找的元素，则经上述若干次后，即可找到。此时（成功查找条件）：

**low<=high && x==A[mid].key** 。

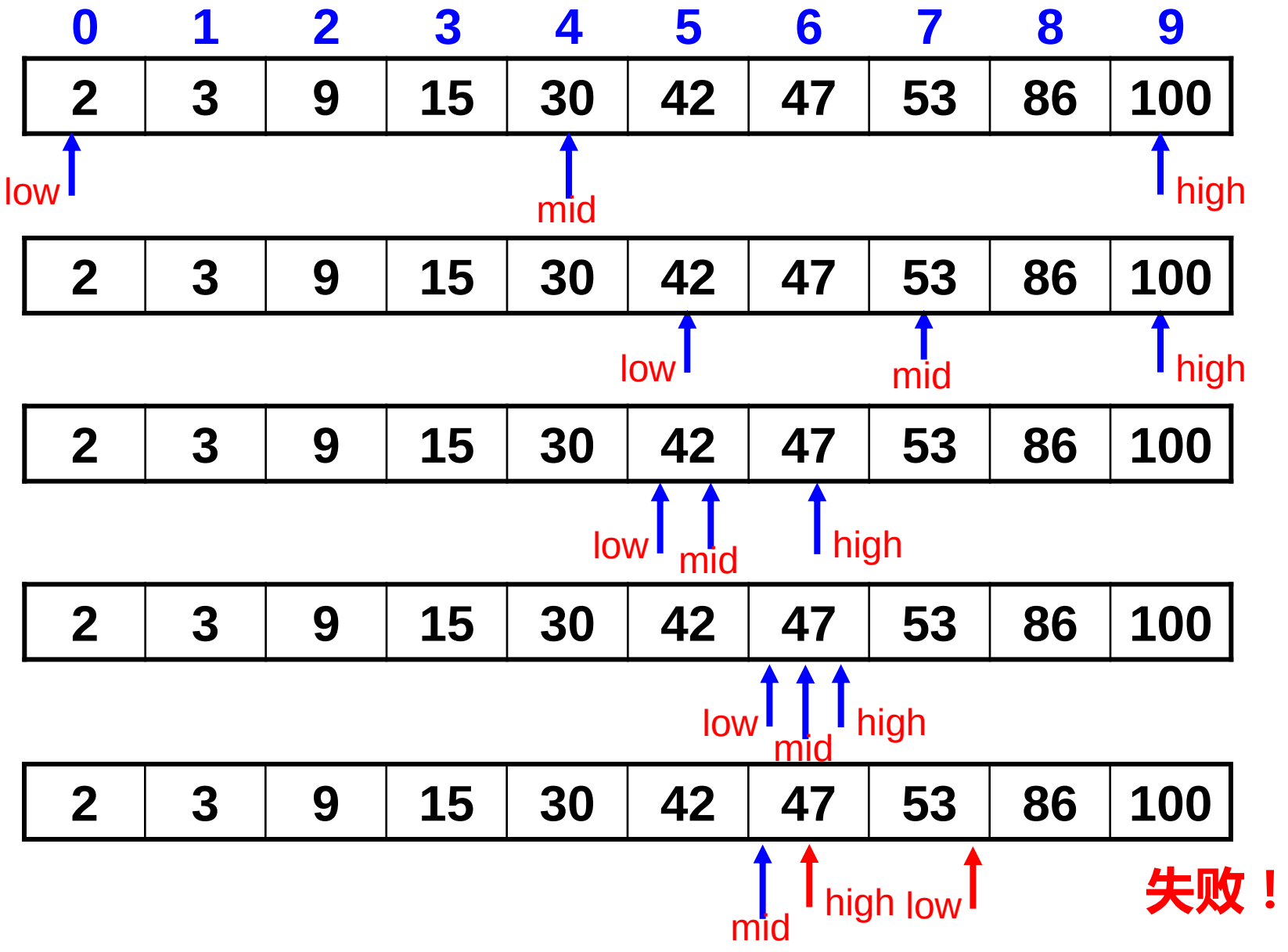
☞ 问题是：如果不存在指定的元素（查找失败），如何能判断出来？

## 3. 折半查找实例演示

- 成功查找：在下图有序表中查找元素x=47。



- 失败查找：在下图有序表中查找元素x=50。



## 4. 二分查找算法描述

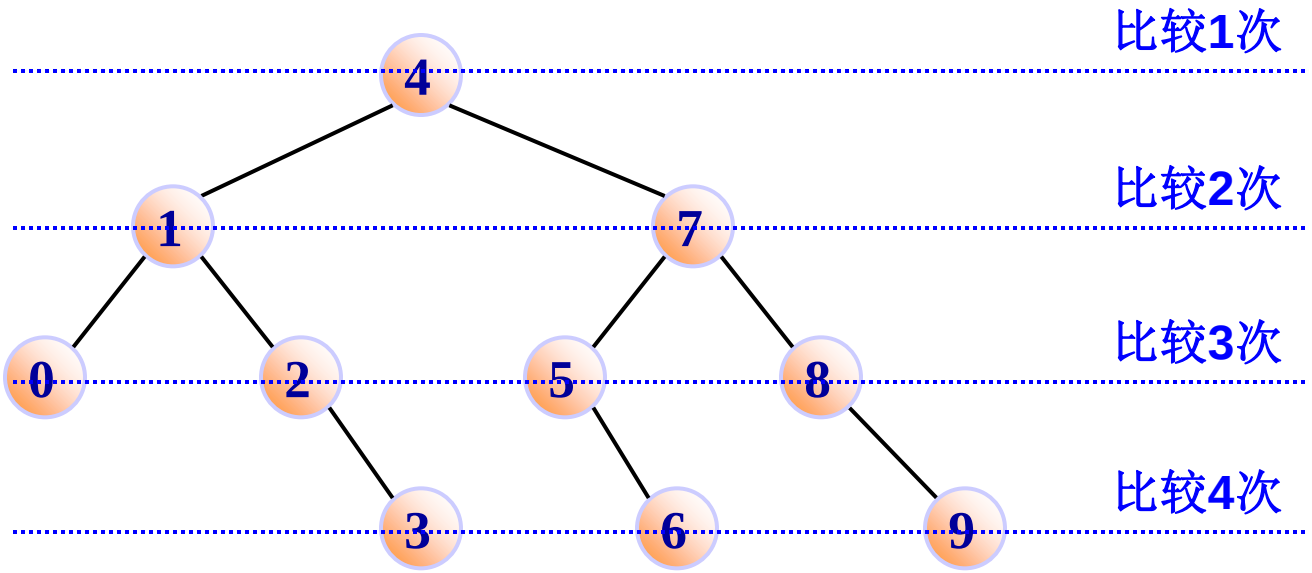
```
int BiSearch(elementType A[ ], keyType x)
 {  int mid, low=0, high=n-1;      //下标初始化
    while ( low<=high)
    {
       mid=(low+high)/2;    //计算中间点下标
       if (x==A[mid].key)        return  mid;  //成功
       else if (x<A[mid].key)  high=mid-1;  //左区
       else                              low=mid+1;   //右区
     }
     return  -1;          //查找失败
}
```

## 5. 二分查找的递归算法描述

```
int BiSearch(elementType A[ ], keyType x, int low, int
    high)
 {
    if( low>high )  return -1;                    //查找失败
    else
    {
        mid=(low+high)/2;
        if (x==A[mid].key)        return  mid;  //成功
        else if (x<A[mid].key)
            return BiSearch(A, x, low, mid-1);   //搜索左半区
        else
            return BiSearch(A, x, mid+1, high);  //搜索右半区
    }
}
```

# 6. 二分查找判定树

- 描述二分查找过程的二叉树。
- 二叉树中一个结点对应表中一条记录。
- 树中结点标注记录的下标（非元素值）。
- 以mid下标为树根；左子表对应左子树；右子表对应右子树。
- 前面实例对应的二叉树如下图：



- 二分查找判定树，除了最后一层结点，为满二叉树。那么二分查找树是否一定时完全二叉树呢？
  - 二分查找树不一定是完全二叉树，但除了最后一层结点，为满二叉树。
- 通过二分查找判定树，可以直接确定元素的查找次数，即结点在二叉树上的层次。
- 给出序列的起、止下标可以直接画出二分查找判定树。
- 同一序列，起止下标不同，画出的二分查找判定树形态相同，只是结点编号不同。

- 对二叉查找树进行中序遍历结果如何？
  - 中序遍历序列与数组次序相同。

- 【课堂练习】
  - 画出A[0…19]和A[1…20]的二分查找树。

- 【思考问题】
  - 1.利用二分查找树如何求ASL？
  - 2.给出求A[7]的比较序列？

# 7. 算法分析

- 可见二分查找，成功查找的比较次数最多不会超过二叉树的深度，即<=$\lfloor \log_2 n \rfloor + 1$
- 平均查找长度：
  - 假设表的记录数为$n=2^h-1$个，则判定树是深度$h=\log_2(n+1)$的满二叉树。表中记录查找概率相同，为1/n，则平均查找长度为：

$$ASL = \sum_{i=1}^{n} P_i C_i = \frac{1}{n} \sum_{j=1}^{h} j \times 2^{j-1} = \frac{n+1}{n} \log_2(n+1) - 1$$

  - 当n较大时：$ASL = \log_2(n+1) - 1$

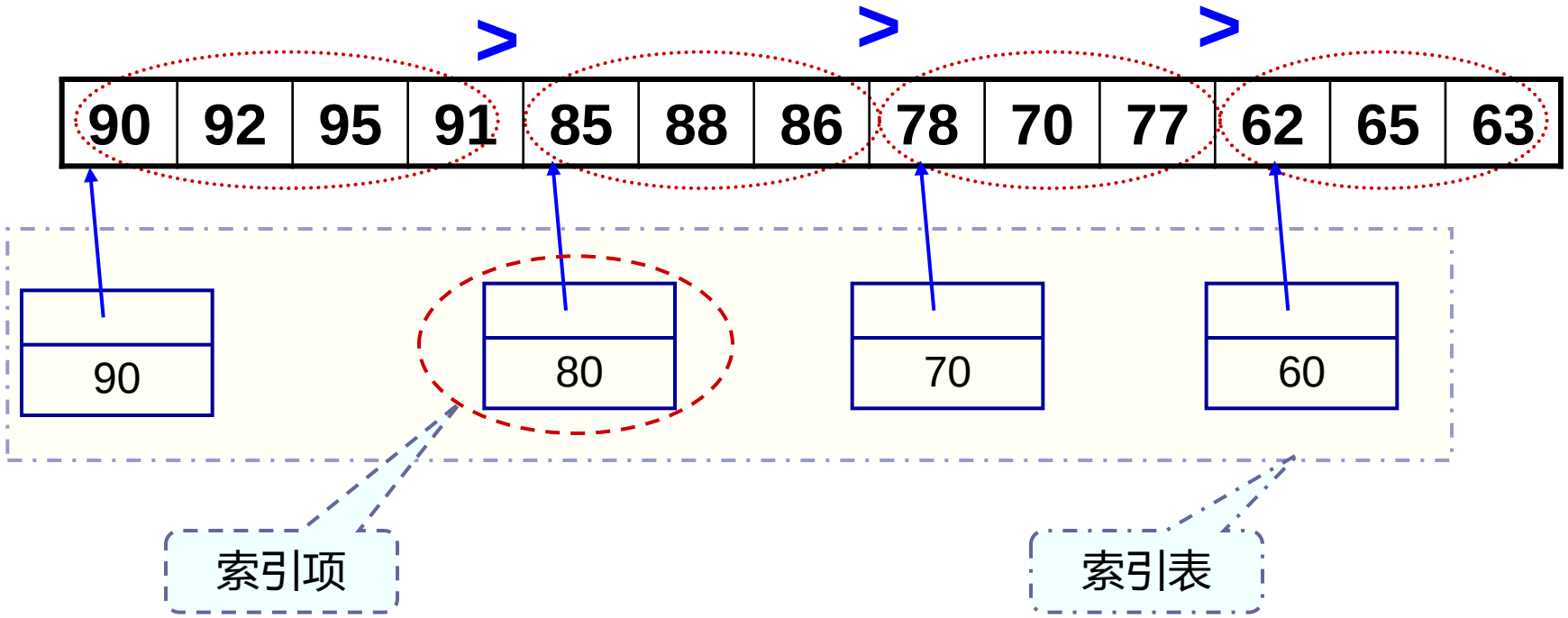
- 最大（最坏）查找长度：$\lfloor \log_2 n \rfloor + 1$
- 时间复杂度：**$O(\log_2 n)$**

## 8. 适用存储结构：

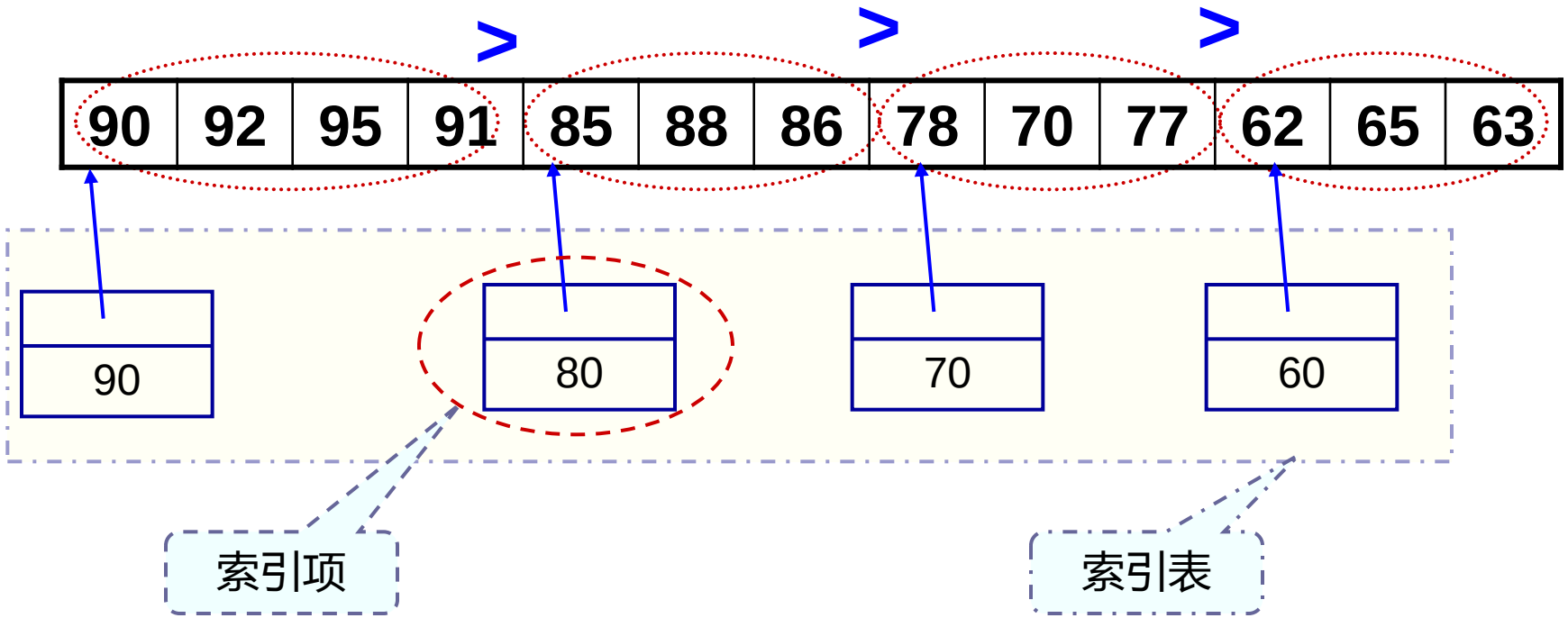
- 有序顺序表
- 不能用于链式存储结构（有序链表可用跳表**SkipList**快速查找）

# 7.2.3 索引表的查找

- **一些数据表具有这样的特性：**
  - 分块有序 —— 块间有序，块内无序。
  - 如下图所示：



- **为便于查找，可这样处理：**
  - 为每一块设立一个块首指针，
  - 并标注对应块中的最大（小）关键字（可能是潜在的，不一定出现）
  - 将这两项内容合并为一个索引项
  - 各索引项合在一起构成索引表。



- 索引表的查找方法：
  - 分两步进行：
  - ① 首先要在索引表中查找以确定元素所在的块；
  - ② 然后在所确定的块中进行查找。

其中：

- 块间查找的实现：
  - 二分查找（索引表有序）
  - 简单顺序查找
- 块内查找的实现：
  - 只能采用简单顺序查找。

- 索引表查找的性能分析：
  - 介于简单顺序查找和二分查找之间

**The deeper you go, the deeper you are able to go, and the deeper you go, the deeper you want to go, and the more enjoyable the experience becomes.**

单选题 1分

设置

图的BFS生成树的树高比DFS生成树的树高（　　）。

A．矮　　B．相等

C．矮或相等　　D．高或相等

A 矮

B 相等

C 矮或相等

D 高或相等

提交

# 7.3 树表的查找

- 二分查找效率较高，但只适用有序顺序表；
- 顺序表不适合频繁的插入、删除操作
- 顺序表只适合作静态查找表；
- 本节介绍的几种树表结构，由于采用链式存储结构，适合作动态查找表。

# 7.3.1 二叉排序树

- **Binary Search Tree，又叫二叉查找树。**

**1.** 二叉排序树定义：

- 二叉排序树是一棵二叉树，或者为空，或者满足以下条件：

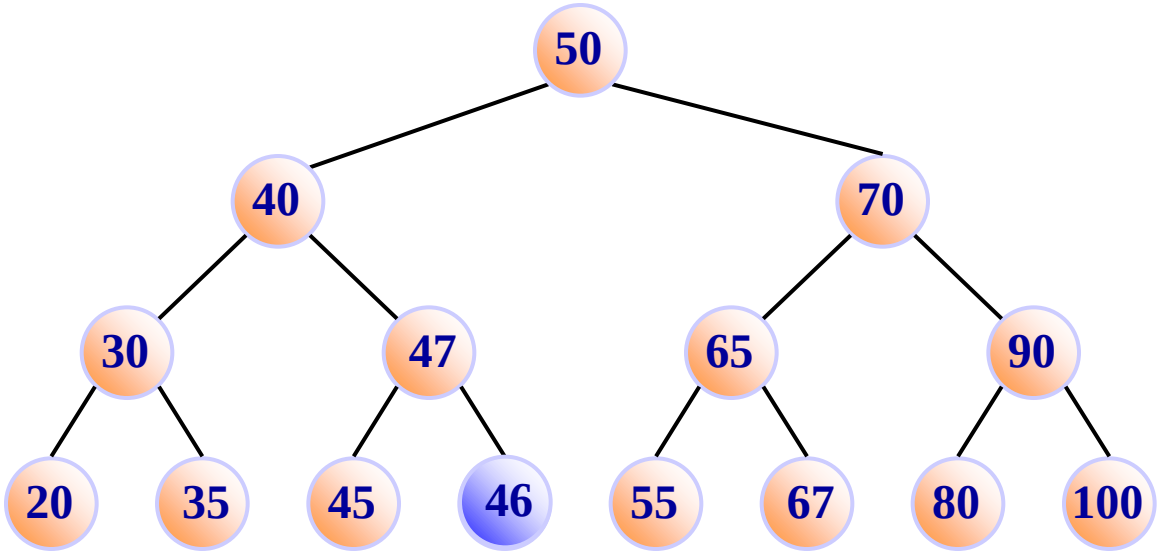
① 若左子树不空，则左子树中所有结点的值小于根结点的值；

② 若右子树不空，则右子树中所有结点的值不小于根结点的值（大于等于）；

③ 左、右子树都为二叉排序树。

- 显然，定义是递归的。
- 二叉排序树特点：中序序列是非降序列。

# 二叉排序树实例




- 此树中若将结点**49**更换成**46**，如图，则不是一棵二叉排序树。

【思考问题】二叉排序树是否可以改为以下定义呢？

① 二叉排序树是满足如下条件的二叉树：其中每个结点的值

- 大于其左子树中所有结点的值，
- 小于或等于其右子树中所有结点的值。

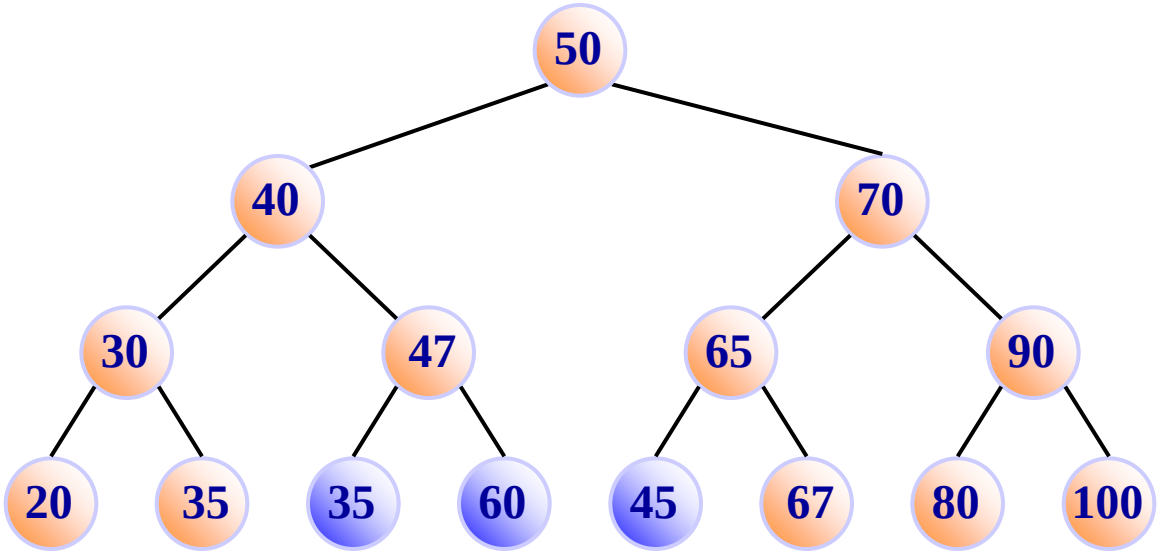
【答】正确

② 二叉排序树是满足如下条件的二叉树：其中每个结点的值

- 大于其左孩子结点的值，
- 小于或等于其右孩子结点的值。

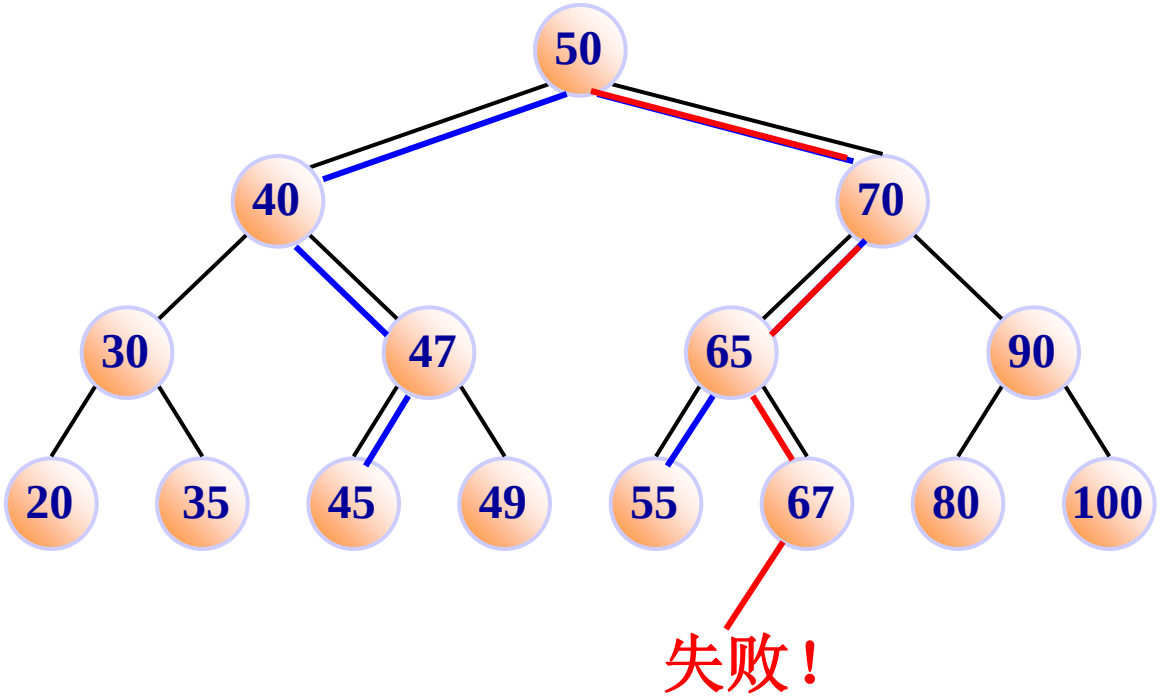
【答】错误，如下图所示。

# 非二叉排序树实例



## 2. 二叉排序树的查找

- 例，在下图中查找45、55、66



- 可见二叉排序树的查找与折半查找非常相似。
- 将待查关键字x与T的根结点关键字比较，出现如下情况：
  ① x==T->key，查找成功，返回根结点指针；
  ② x<T->key，x只可能在T的左子树中，应在左子树上继续查找；
  ③ x>T->key，x只可能在T的右子树上，应在右子树上继续查找。

## 3. 二叉排序树查找的非递归算法描述

```
btNode *bstSearch(btNode *T， elementType x)
  {
      p=T;
      while ( p != NULL )
      {
         if ( p -> key = = x )
              return p;                  //查找成功
         if ( x < p -> key )
                p = p -> lChild;   //继续在左子树查找
         else  p = p -> rChild;   //继续在右子树查找
       }
       return NULL;    //失败
  }
```

## 4. 二叉排序树查找的递归算法描述

```
btNode *bstSearch(btNode *T，  elementType x)
{
    if ( T = = NULL )
        return T;              //失败
    else if( T->key==x )
        return T;              //成功
    else if( T->key>x )
        return bstSearch(T->lChild,x); //搜左子树
    else
        return bstSearch(T->rChild,x); //搜右子树
}
```

## 3. 二叉排序树中插入结点

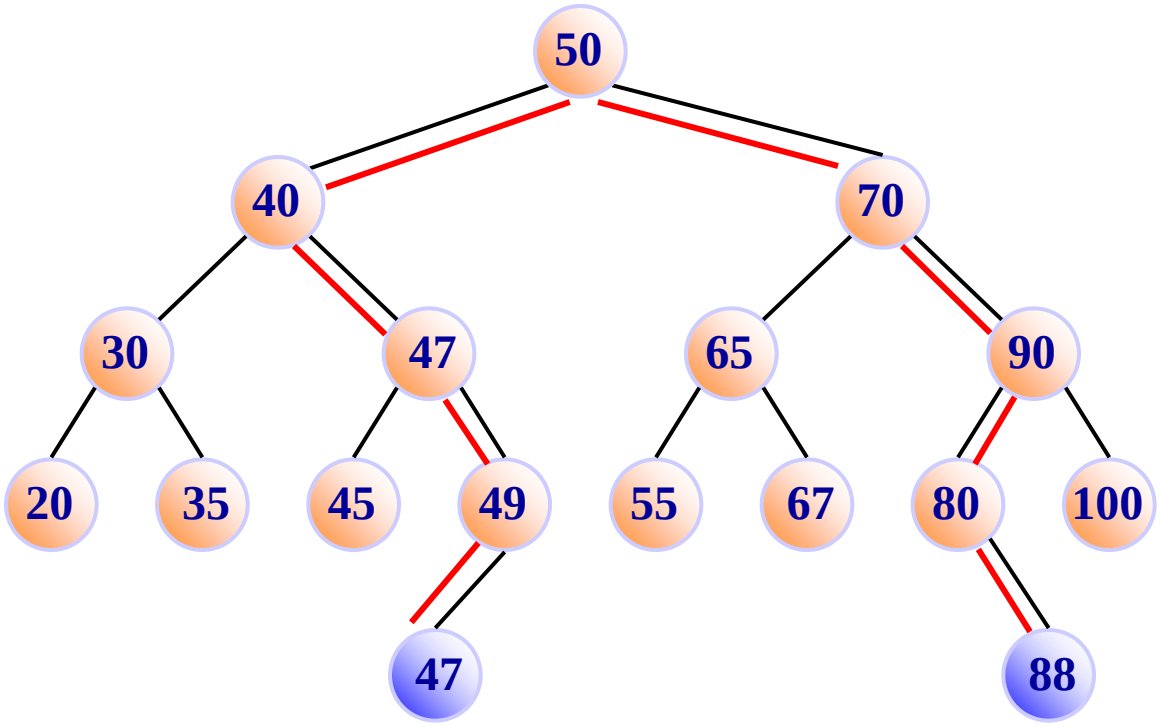
☞ 为了保持二叉排序树的特征，必须：

① 若待插结点的值小于根结点的值，则往左子树中插入

② 若待插结点的值大于等于根结点的值，则往右子树中插入。

③ 按此方式递归调用若干次后，可以搜索到一个空子树位置，即要插入的位置。即插入的结点一定为叶子结点。

- 例在下图中插入结点47、88



- 【二叉排序树结点插入算法描述】

```
        //将结点s插入到以T为根指针的二叉排序树中
void insert(btNode *& T, btNode *s)
{
    if ( T==NULL )      // T 为空时
        T == s;             // 新结点s作为根结点
    else    if (  s->data < T->data  )
        insert( T->lchild, s );     //s插入左子树
    else
        insert( T->rchild, s );     //s插入右子树
}
```

【二叉排序树结点插入非递归算法描述】

```
void InsertNodeNR( btNode *&T,  btNode *s )
{   if( T==NULL ) { T=s;} else
    while( T )
    {
        if( T->data>s->data )    //s插左子树
            if( T-> lChild )   T=T->lChild;
            else { T->lChild=s; break; }
        else  // T->data<s->data，s插右子树
            if( T-> rChild )   T=T->rChild;
            else { T->rChild=s; break; }
} }
```

## 4. 二叉排序树的构造（创建）

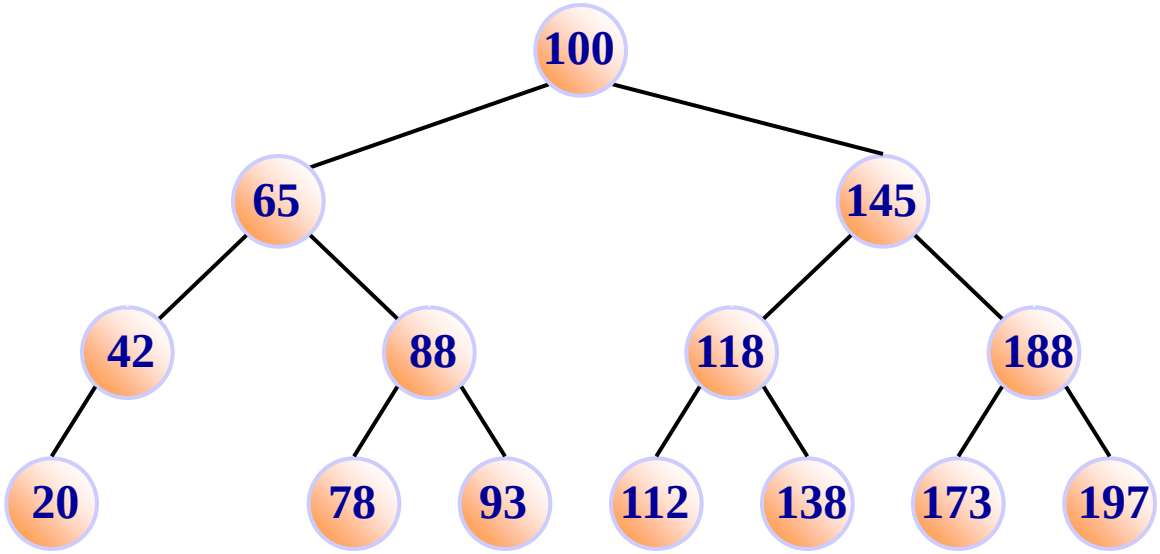
- 从空树开始，保持二叉排序树的特征，反复将新结点作为叶结点插入树中。

【二叉排序树构造算法描述】

```
void  createBst(btNode *&T)
            // 接受读入数据，从空树开始构造二叉排序树
{
        T=NULL;
        cin>>x;
        while (x!=结束符)
        {
            s=new btNode;       //申请新结点
            s->data=x;
            s->lChild=NULL;
            s->rChild=NULL;
            insert( T, s );          //调用插入结点算法
            cin>>x;
        }
 }
```

- 二叉排序树构造分析：
  - 以下列数据序列作为输入构造一棵二叉排序树。

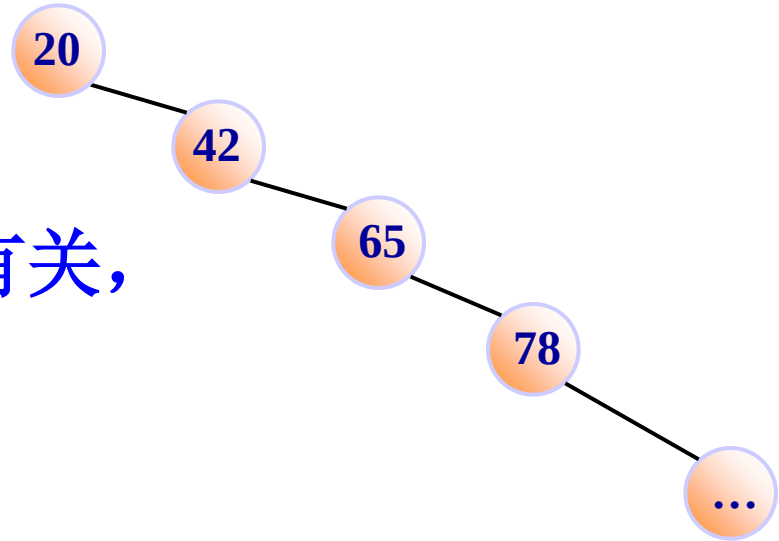
    **100，65，88，93，145，118，138，112，188，173，42，78，20，197**
  - 平均查找长度：
    - **ASL =（1＋2×2＋3×4＋4×7）/14 =45/14**



- 再以下列数据序列作为输入构造一棵二叉排序树。

  **20，42，65，78，88，93，100，112，118，138，145，173，188,197**

- 与上例是同样一组数据，只是排列顺序不同，这里为递增有序序列。
- 构造的二叉排序树为单分支树。退化为线性表。
- 平均查找长度：
  - （1＋2＋…＋14）/14 =105/14




- 平均查找长度与树的高度有关，
- ---平衡二叉树

单选题 1分

设置

一棵哈夫曼树，有n个2度结点，则此树总共有（ ）个结点。

A 2n-1

B 2n

C 2n+1

D 2n+2

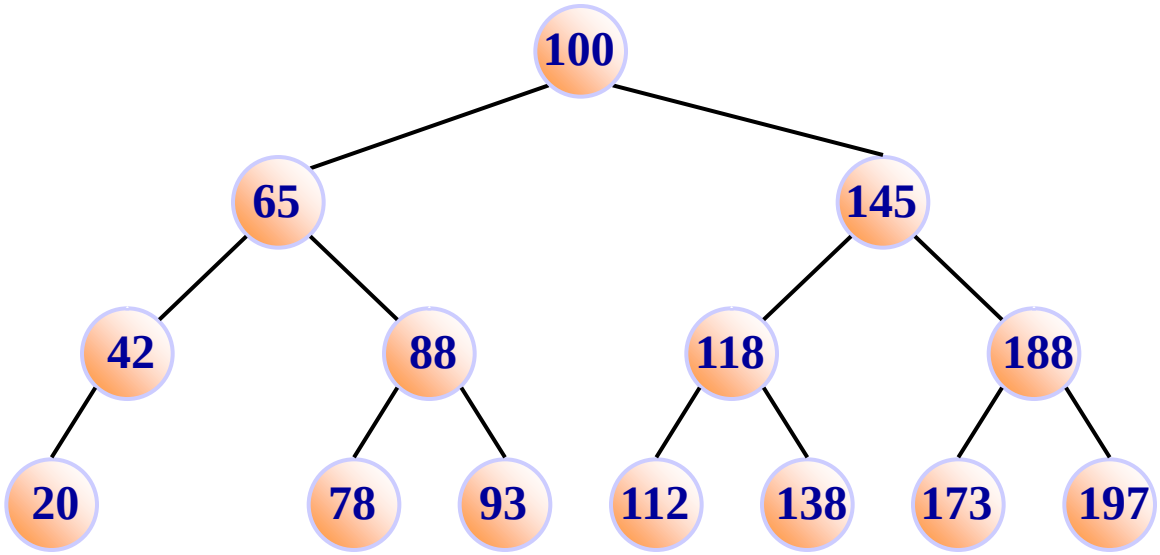
提交

## 5. 二叉排序树删除结点

- 设待删除结点指针为p，其父结点指针为fp。

### ① 删除叶结点

- 直接删除，不改变其有序性。
- 例：删除下图中的结点78。

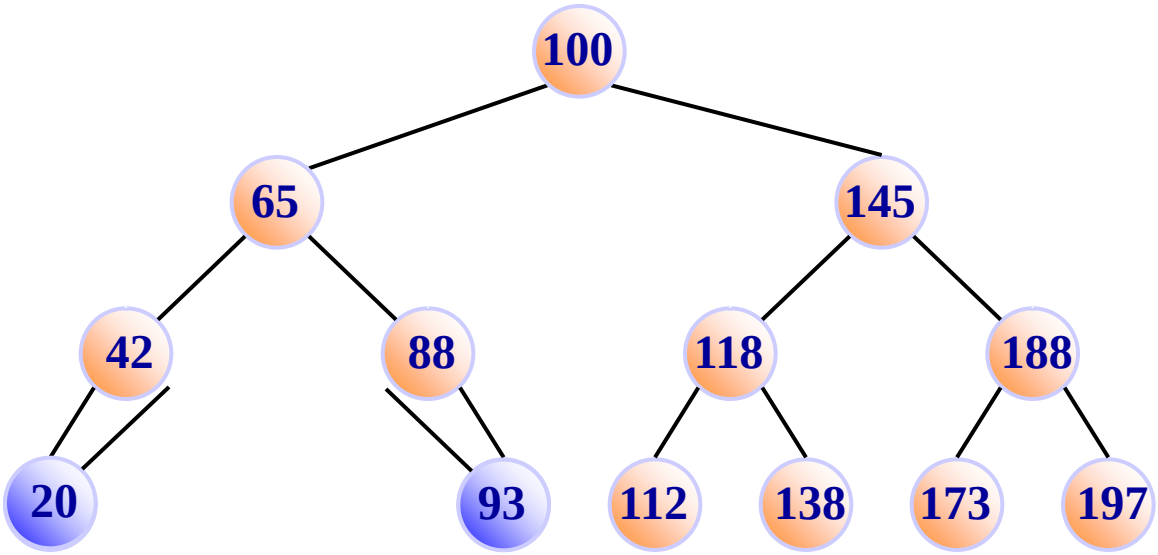



删除叶子结点78

② p 只有左子树 pl，用p左子树的根结点直接取代 p。

③ p 只有右子树 pr，用p右子树的根结点直接取代 p。

- 【例】删除下图中的结点42和88

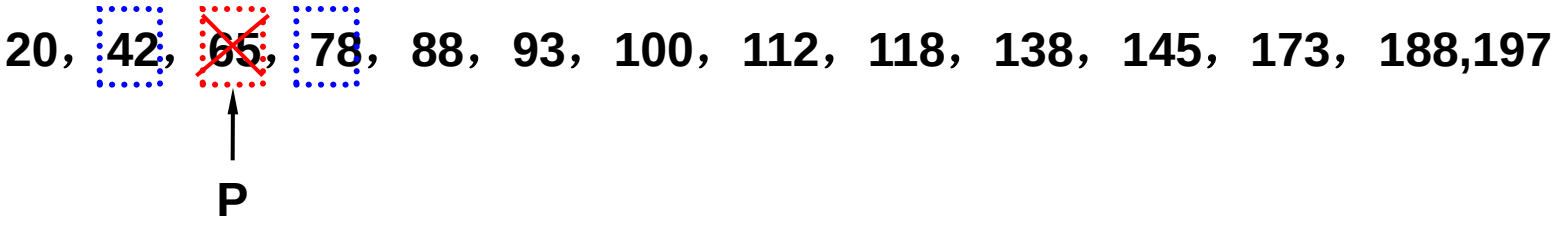



删除结点42和88

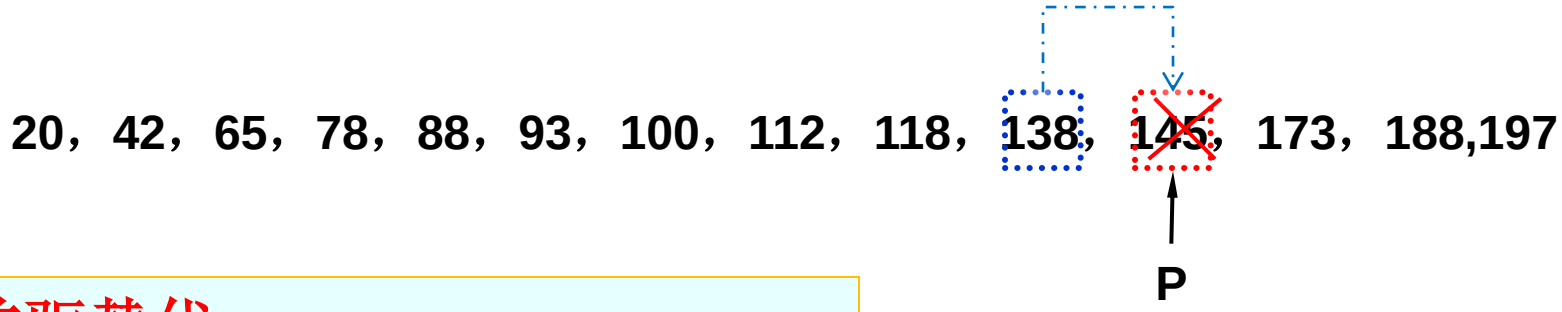
④ p 的左、右子树都不空。

(1) 替代法（顶替法）

- 用 p 的中序序列的直接前驱或直接后继顶替删除结点。
- 并不真正删除目标节点p，而是把前驱或后继结点的值写入结点p，然后删除这个前驱或后继结点s。



- 直接前驱顶替
  - 用p的直接前驱s的值顶替p的值；重接s左子树；删除s。例：

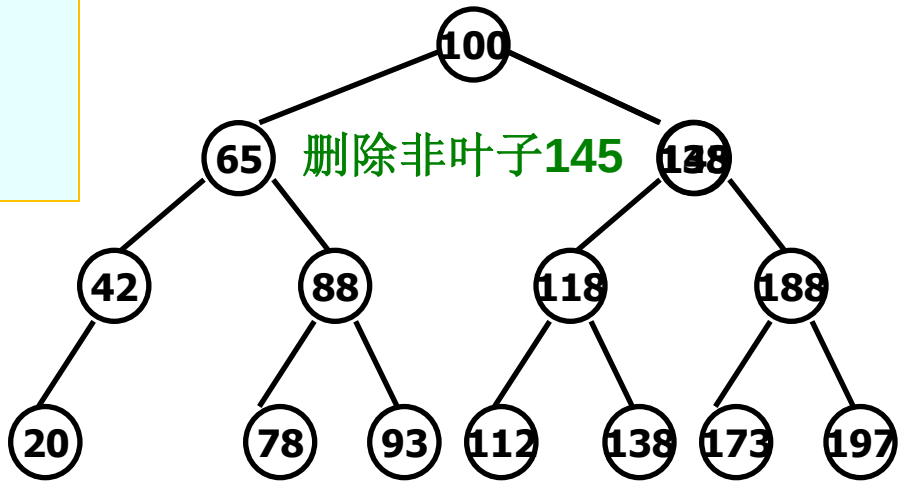



直接前驱替代：

P 左子树的中序遍历最后结点替代，

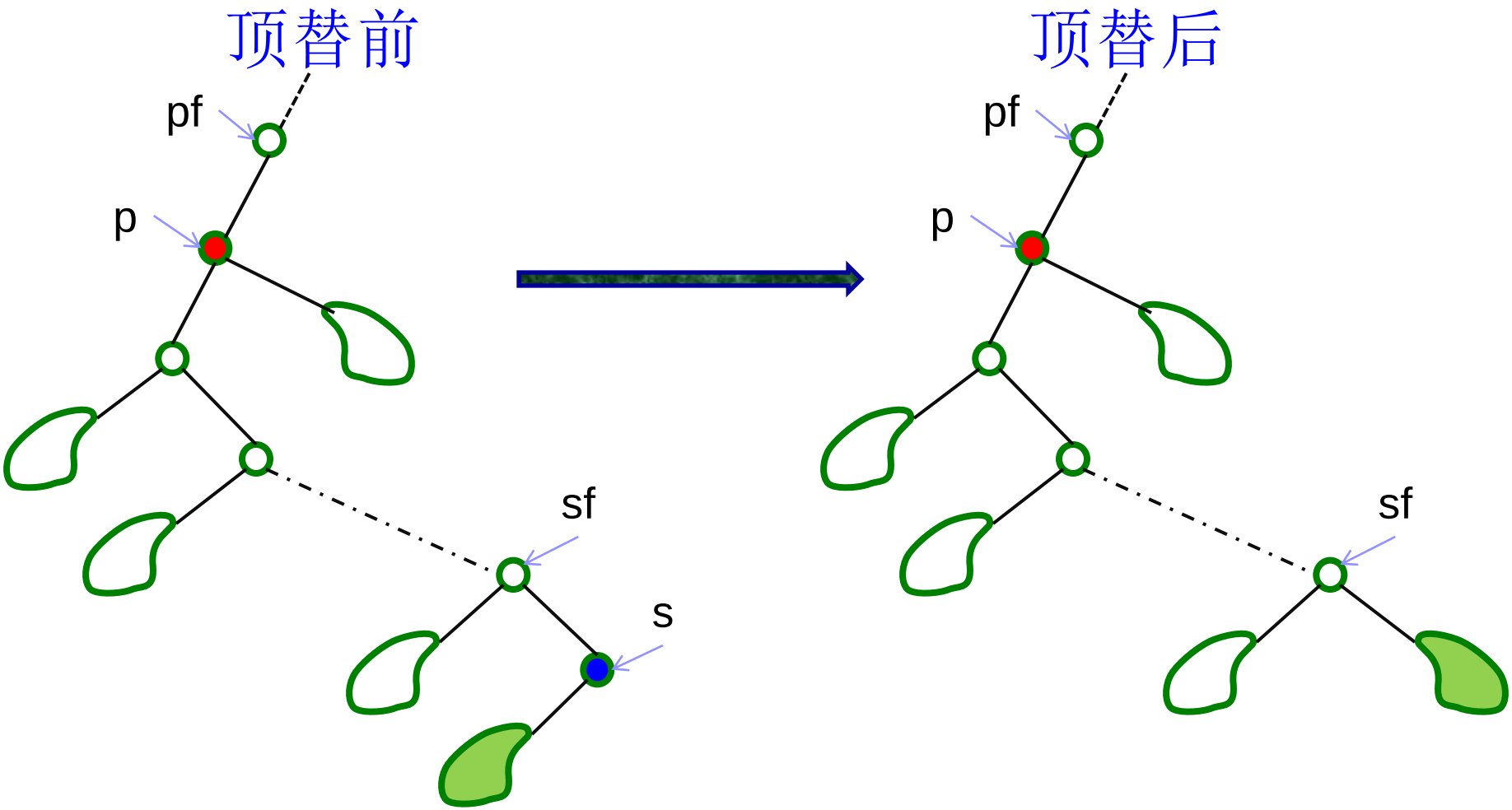
或，左子树最右结点，

即：左子树中的最大值。



正常情况执行：

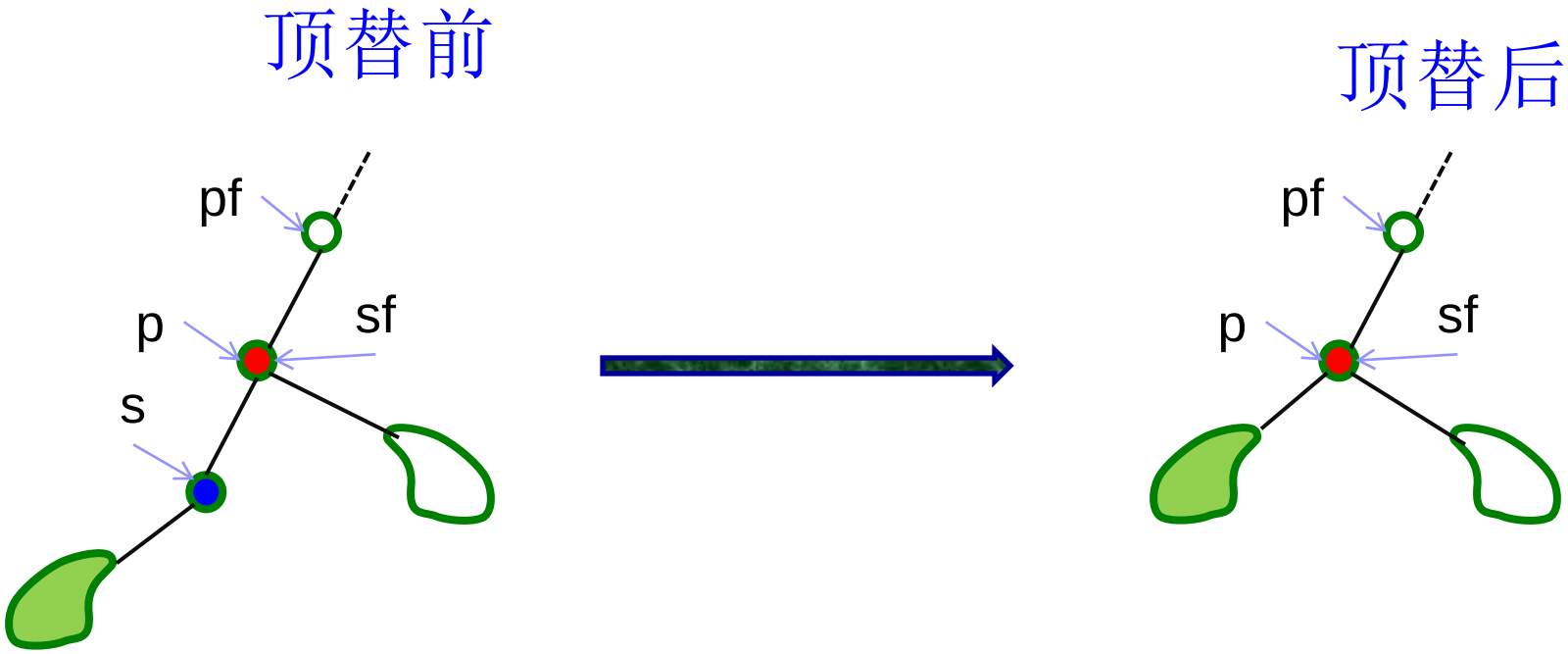
```
p->data=s->data;
sf->rChild=s->lChild;
delete s
```



- 特殊情况

特殊情况sf==p执行：

```
p->data=s->data;
sf->lChild=s->lChild;
delete s
```



- 直接前趋顶替操作：
  - p是待删除结点；s是p的直接前趋结点，sf是s的父结点

① 搜索p的直接前驱s，sf指向s的父结点；

② s的值赋给p：p->data=s->data；

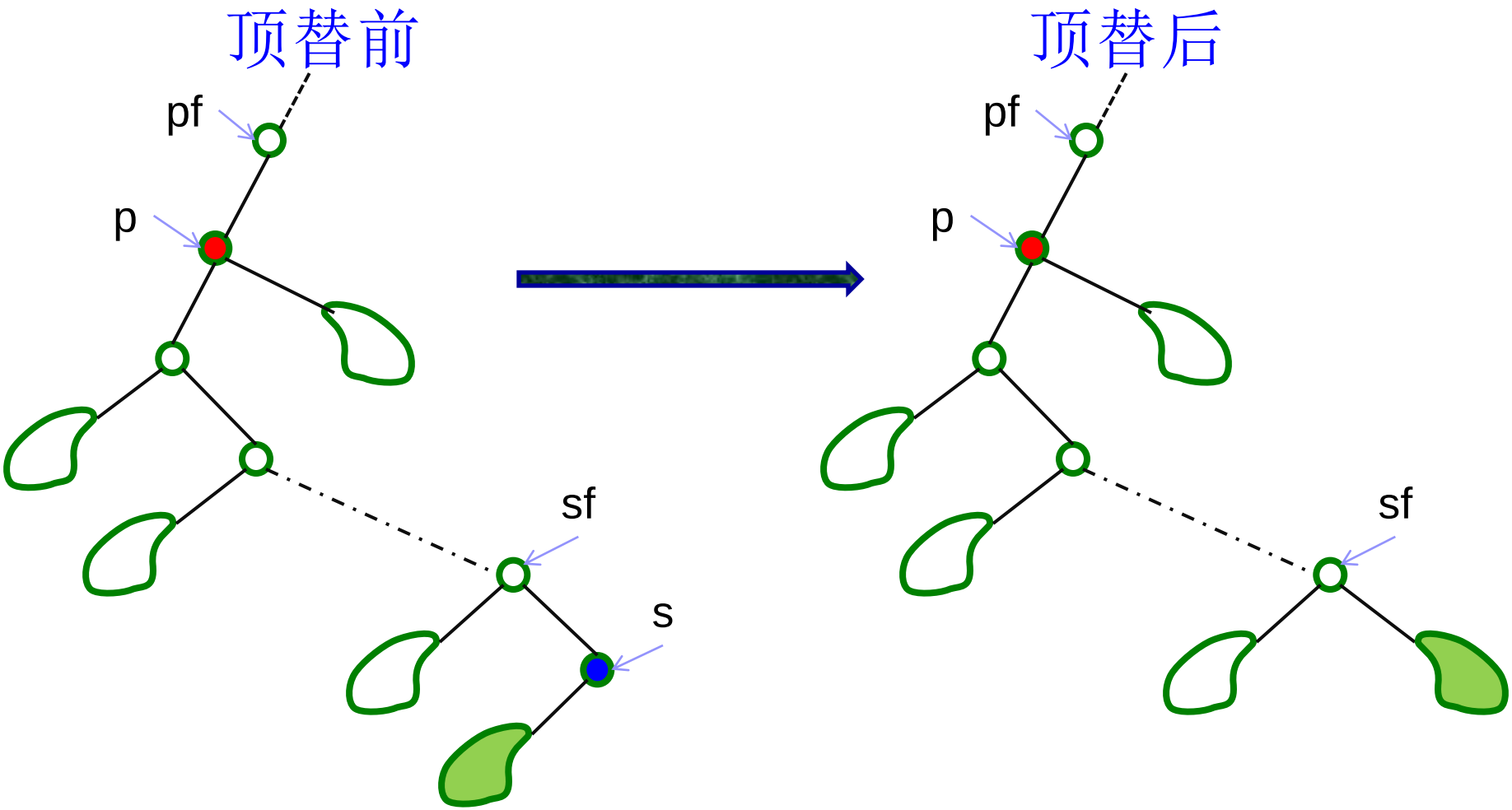
③ s左子树重接为sf的右子树：sf->rChild=s->lChild；

④ 特殊情况p==sf处理
  - s一定是p的左孩子，且s没有右子树，此时，要把s的左子树接续为p的左子树，即：
  - sf->lChild=s->lChild;或p->lChild=s->lChild;

⑤ 删除s结点：delete s

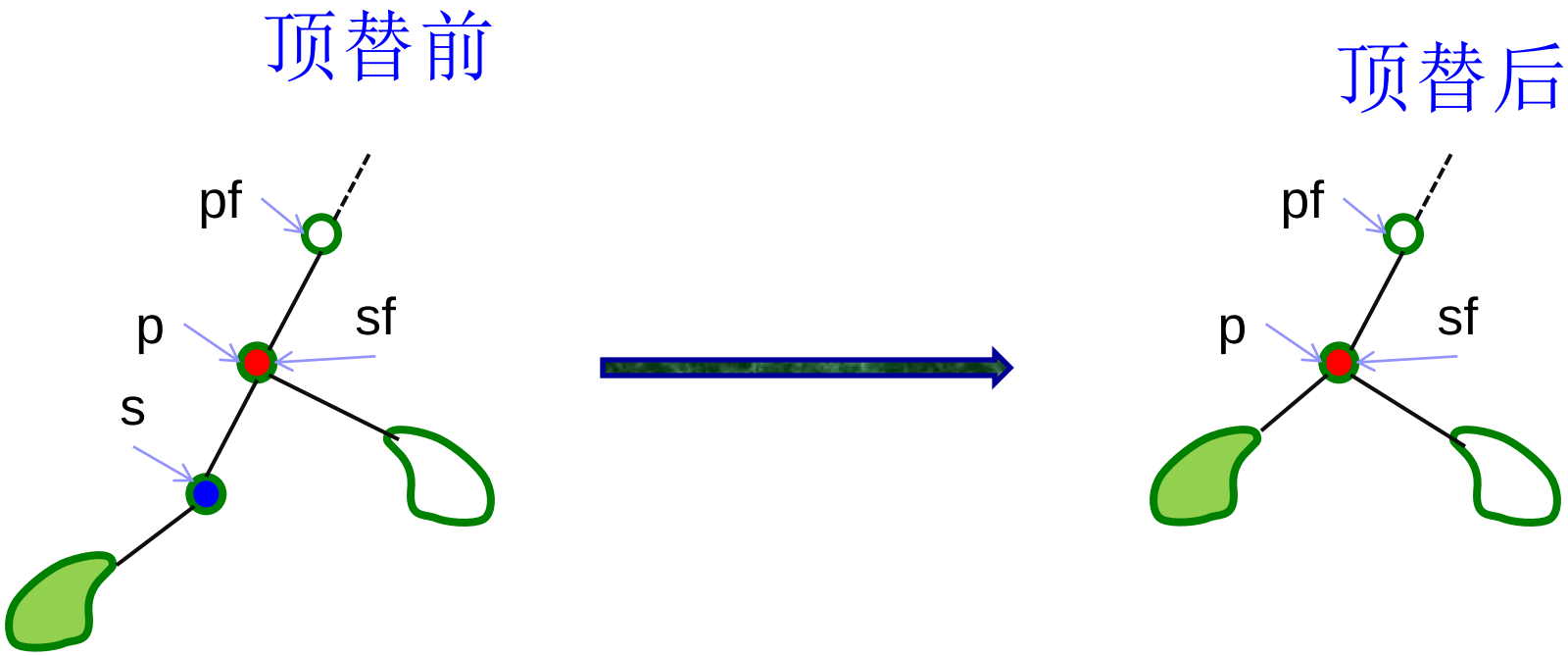
正常情况执行：

```
p->data=s->data;
sf->rChild=s->lChild;
delete s
```



特殊情况sf==p执行：

```
p->data=s->data;
sf->lChild=s->lChild;
delete s
```



- 关键是如何找到目标结点p的中序直接前驱结点s？
  - s是P左子树中序遍历序列最后结点。
  - s是p左子树上值最大的结点；
  - s是P的左子树上最右的结点，s一定没有右子树，可能有左子树；

- 寻找p的直接前驱
  - 先找到p左子树根结点，从这个根结点开始，反复寻找右分支：s=s->rChild，直到右分支最后一个结点。搜索中sf保存s父结点指针，代码如下：

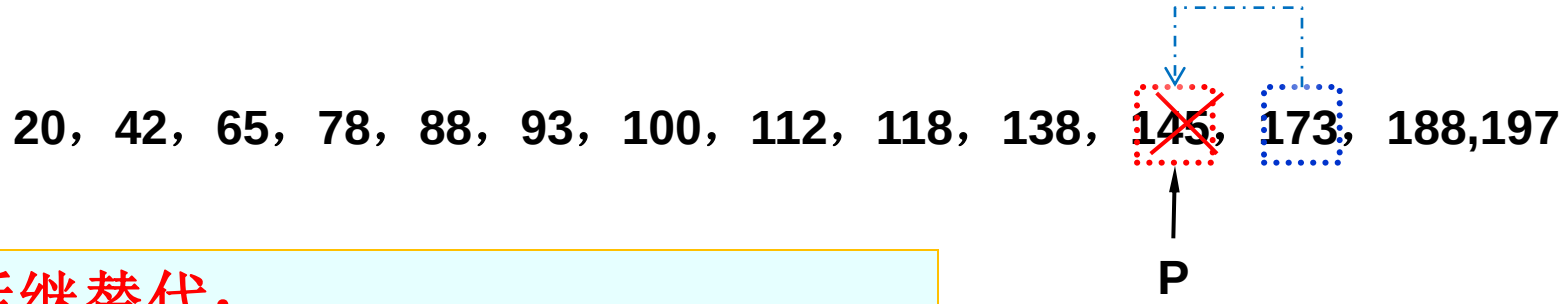
```
sf=p;                    //sf初始指向p
s=p->lChild;             //s初始指向p的左孩子
while(s->rChild)         //反复搜索右分支
{
        sf=s;            //sf保存s父结点指针
        s=s->rChild;     //s右移
}                        //循环结束，s即为最右结点指针
```

- 直接前驱顶替完整代码：

```
{       sf=p;                   s=p->lChild;
        while(s->rChild)
        {       sf=s;
                s=s->rChild;            }
        p->data=s->data;
        if(sf==p)   //s为p（sf）左子树的根结点。
                sf->lChild=s->lChild;
        else
                sf->rChild=s->lChild;
        delete s;
  }
```

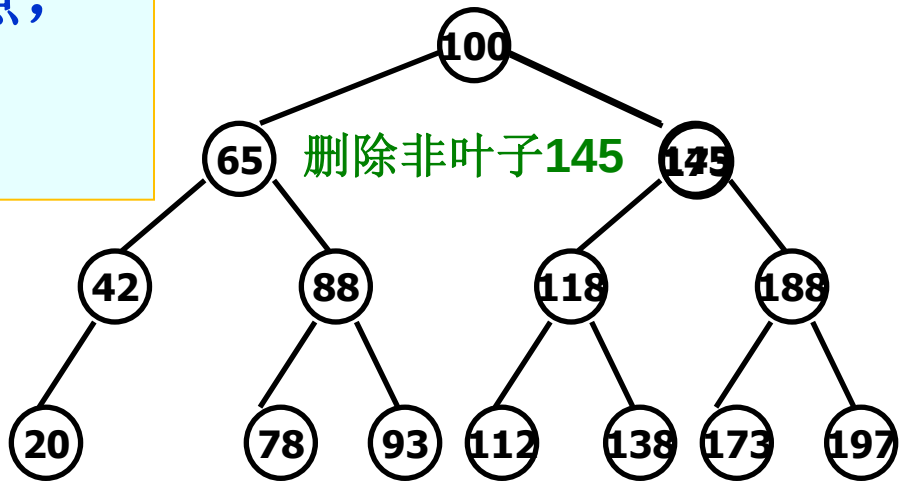
## 直接后继顶替

- 与直接前驱顶替类似，用直接后继s的值顶替p；重接s的右子树；删除s。例：




直接后继替代：

P 右子树的中序遍历第一个结点，
或，右子树最左结点，
即：右子树中的最小值。



- 直接后继顶替操作：
  - p是待删除结点；s是p的直接后继结点，sf是s的父结点

① 搜索p的直接后继s，sf指向s的父结点；

② s的值赋给p：p->data=s->data；

③ s右子树重接为sf的左子树：sf->lChild=s->rChild；

④ 特殊情况p==sf处理
  - s是p的右孩子，且s没有左子树，此时，要把s的右子树接续为p的右子树，即：
  - sf->rChild=s->rChild;或p->rChild=s->rChild;

⑤ 删除s结点：delete s

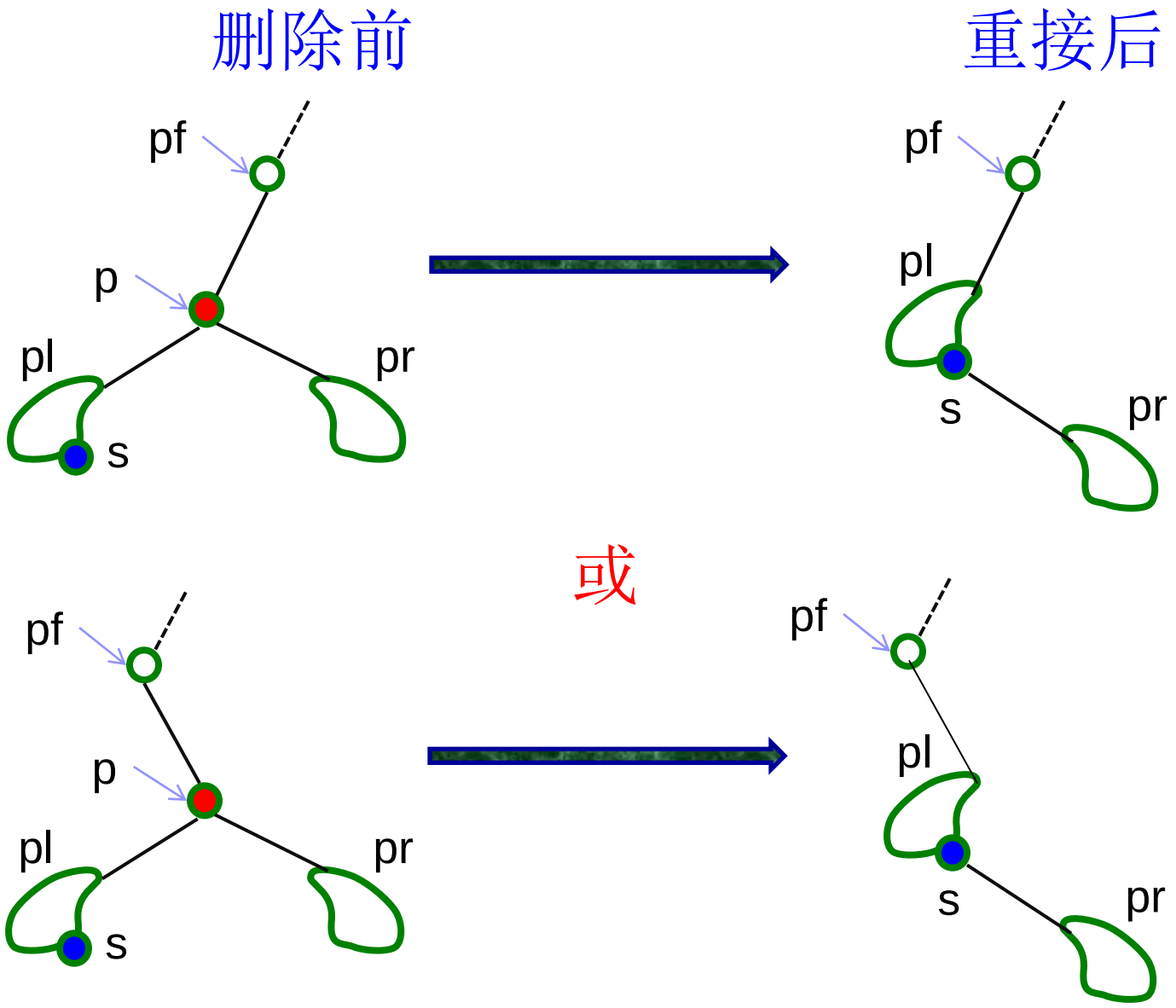
- 直接后继顶替完整代码：

```
{       sf=p;                   s=p->rChild;
        while(s->lChild)
        {       sf=s;
                s=s->lChild;                }
        p->data=s->data;
        if(sf==p)    //s为p（sf）右子树的根结点。
                sf->rChild=s->rChild;
        else
                sf->lChild=s->rChild;
        delete s;
  }
```

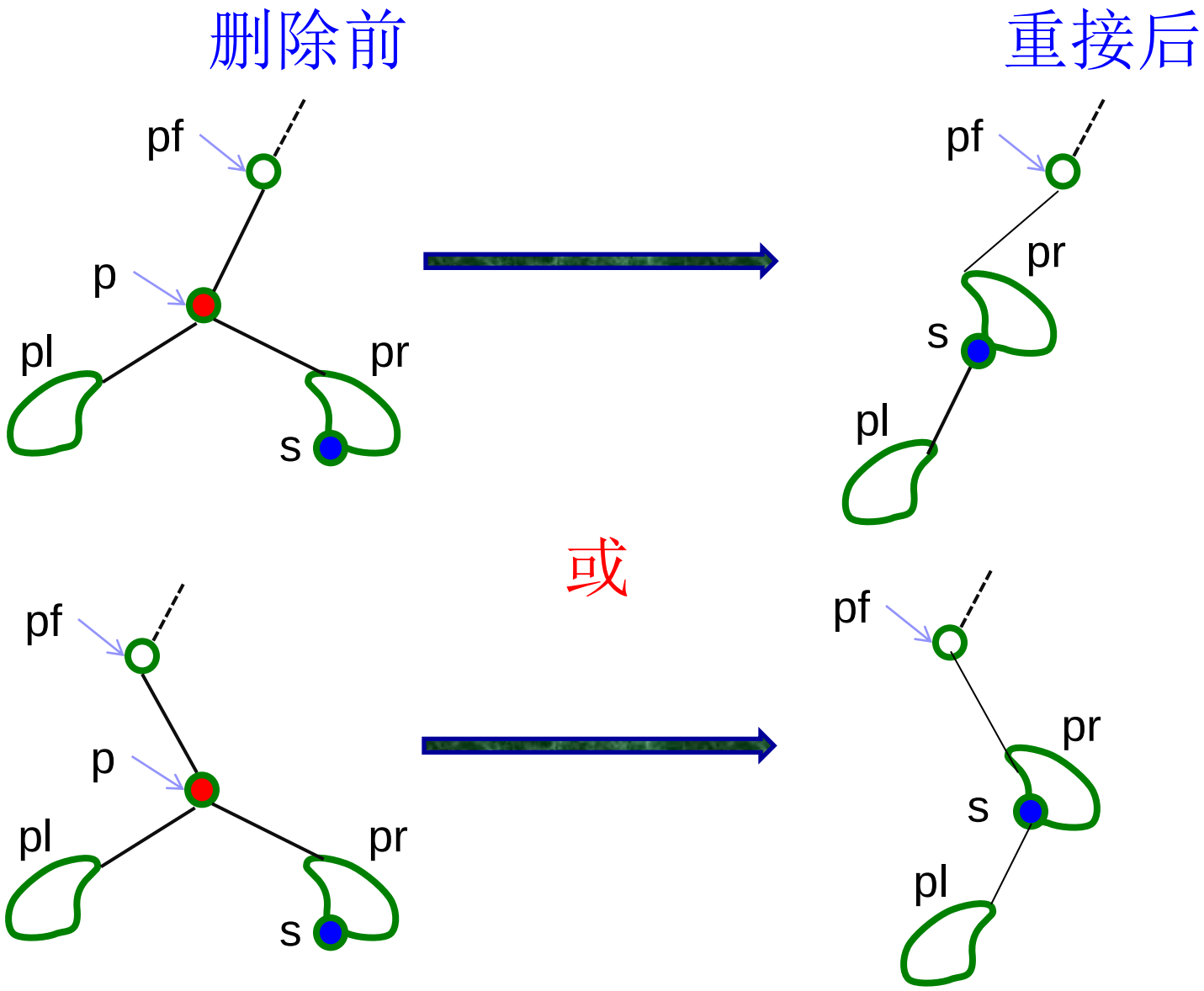
- 顶替法的优点：
  - 算法简单，容易理解；
  - 顶替后不会增加树的高度。

# （2）重接法

- 重接方法一：
  - 将p的左子树重接为pf的子树（左子树或右子树，视p是pf的左右孩子而定）；
  - p的右子树重接为p的直接前驱结点s的右子树（s原没有右子树）。


删除前
重接后
pf
p
pl
pr
s
pf
pl
s
pr
或
pf
p
pl
pr
s
pf
pl
s
pr

- 重接方法二：
  - 将p的右子树重接为pf的子树（左子树或右子树，视p是pf的左右孩子而定）；
  - p的左子树重接为p的直接后继结点s的左子树（s原没有左子树）


删除前
重接后
pf
p
pl
pr
s
或

- 重接法一代码：（重接p左子树）

```
{    s=p->lChild;
     while(s->rChild)     //搜索p的直接前驱
     {        s=s->rChild;        }
     if(pf==p)   //删除根结点，p左孩子成新的根结点
              T=p->lChild;
     else if(pf->lChild==p) //p的左子树重接为pf的左子树
              pf->lChild=p->lChild;
     else                          //p的左子树重接为pf的右子树
              pf->rChild=p->lChild;
      //p的右子树重接为p的直接前趋s的右子树
     s->rChild=p->rChild;
     delete p;
}
```

- 重接方法二的算法请大家自行分析给出。

- 重接法的缺点：
  - 重接直接前趋（直接后继）子树后，会改变树的高度，影响二叉排序树的搜索性能。

单选题 1分

设置

二分查找只能应用于有序顺序表。

A 错

B 对

提交

# 7.3.2 平衡二叉树

## 1. 平衡二叉树的定义

- 平衡二叉树（**Balance Binary Tree**），又称**AVL**树，是一棵二叉排序树，或者为空，或者满足以下条件：

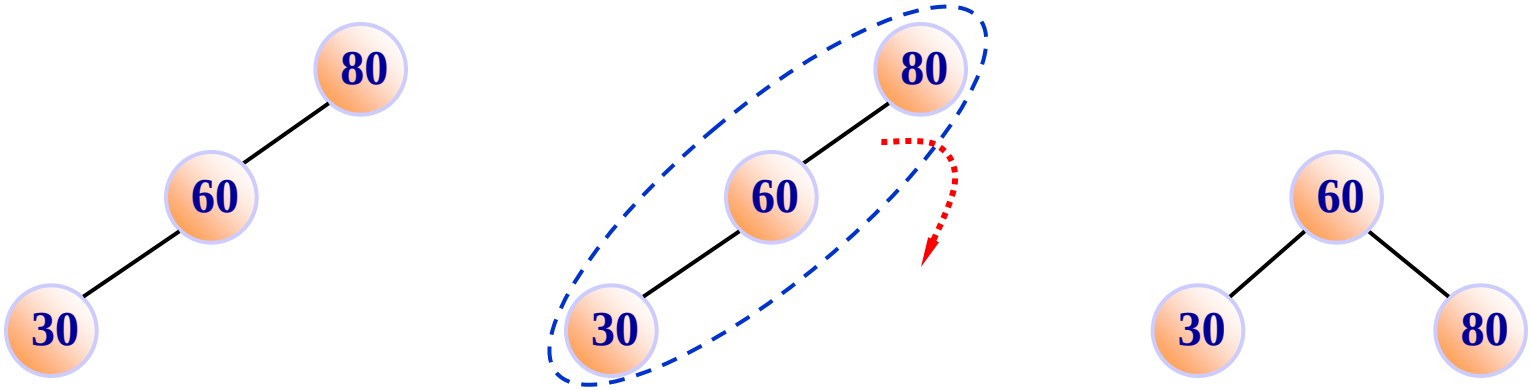
① 左、右子树高度差的绝对值不大于**1**；

② 左、右子树都是平衡二叉树。

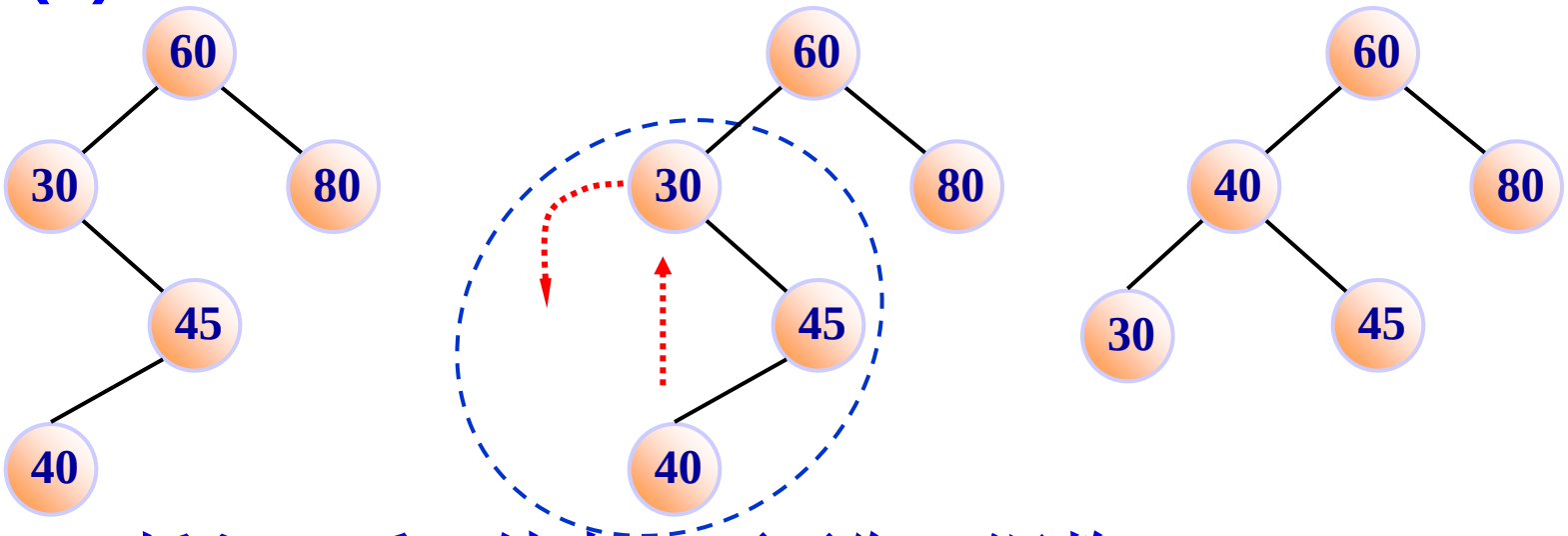
- 平衡因子（**Balance Factor--BF**）：
  - 左子树的高度减去右子树的高度。（或反之）
  - 显然，在平衡二叉树中，每个结点的平衡因子的值为 **-1**，**0** 或 **1**。
- 平衡二叉树可以确保平均查找长度较小。

## 2. 平衡二叉树的平衡化调整

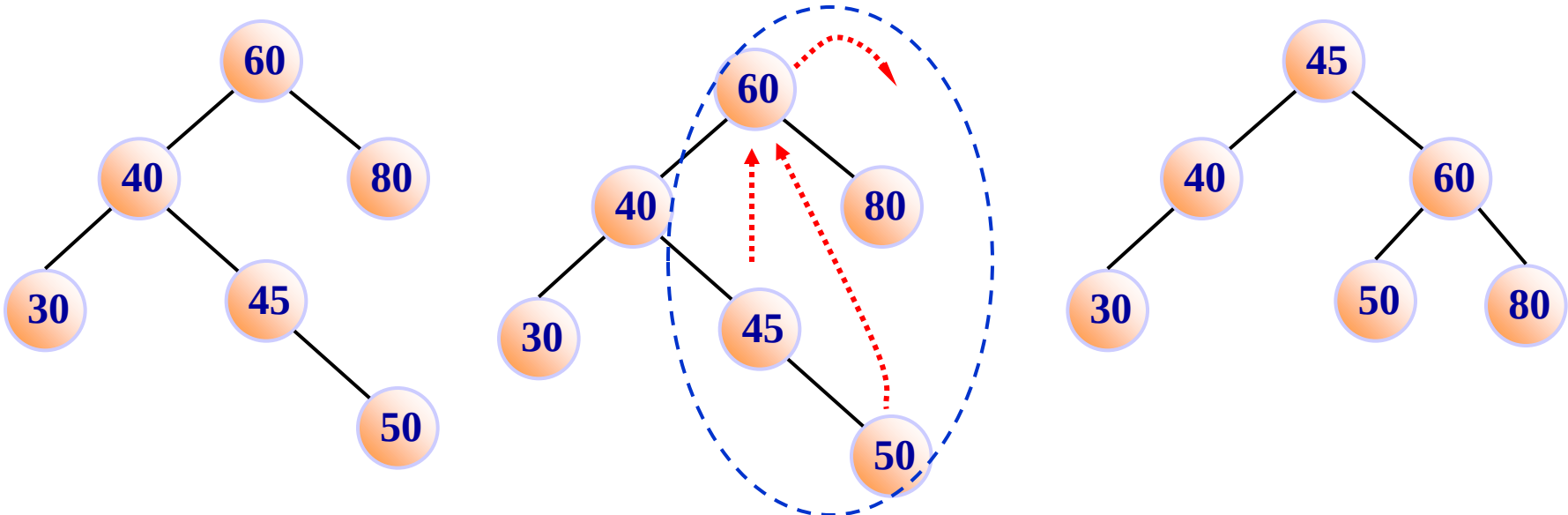
- 如何构造平衡二叉树？
- 基本思想：按照二叉排序树创建的方法依次插入结点，当插入结点出现不平衡时，及时进行调整，使之保持平衡。保证中序有序。
- 下面先以一个实例对有关调整操作产生一个感性认识，然后再介绍这种调整方法。假定插入序列为：80、60、30、45、40、50、85、90。
- (a) 依次插入80、60、30后，结点80不平衡，调整：以结点60为根结点，使之平衡，如图(a)。



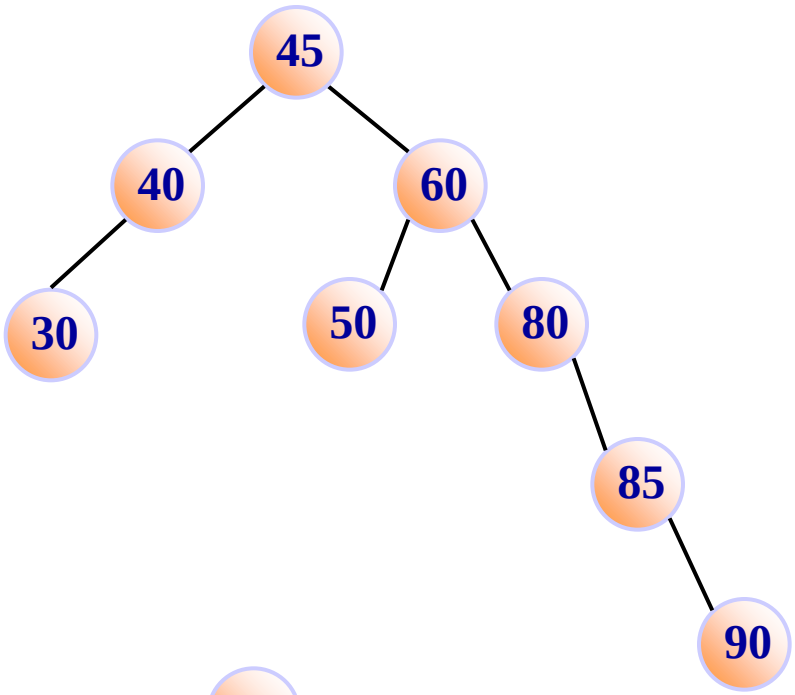
- (b) 依次插入45、40后，结点30不平衡，调整：

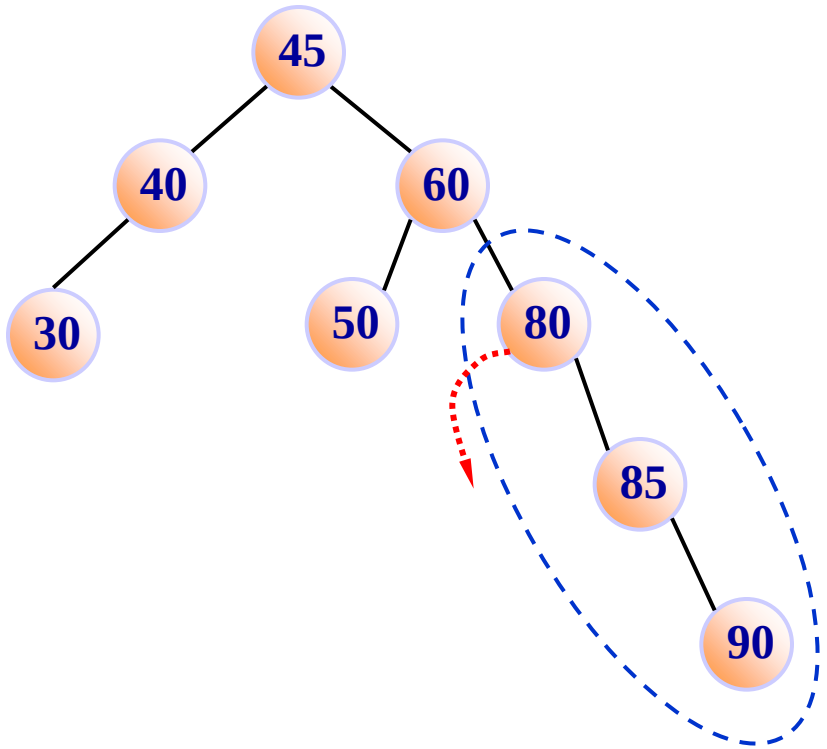


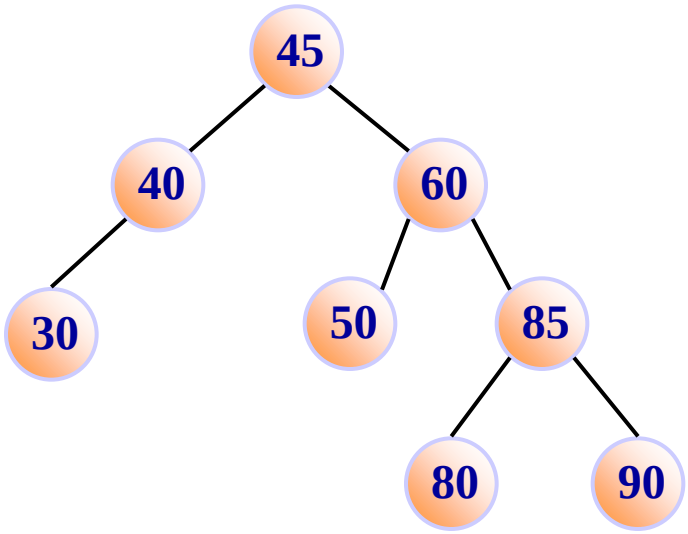

- (c) 插入50后，结点60不平衡，调整：



## (d) 插入85、90后，结点80不平衡，调整



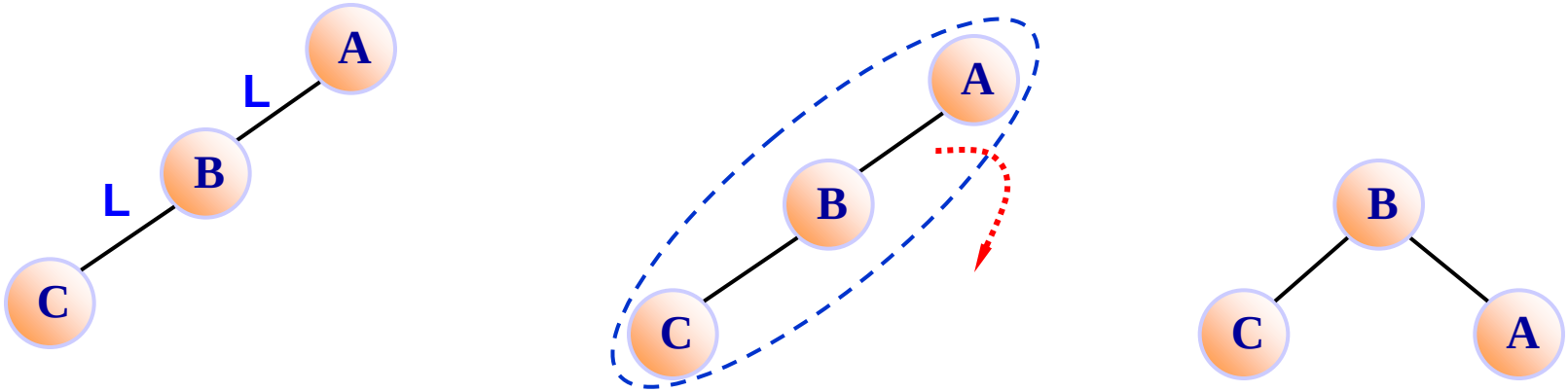



- 从实例可见平衡二叉树的平衡调整的基本思想是在平衡的二叉排序树上插入结点，当出现不平衡时及时调整，以保持二叉排序树的平衡性质。
- 最低不平衡结点
  - 离插入点（删除点）最近的祖先结点，且bf绝对值大于1，不妨设为A。
  - 调整只要在以A为根结点的子树上进行，
  - A称为最低不平衡结点。
- 调整操作的方法
  - 根据新插入结点与最低不平衡结点的位置关系分为LL型、RR型、LR型和RL型4种类型分别处理。
  - 各种调整方法如下：

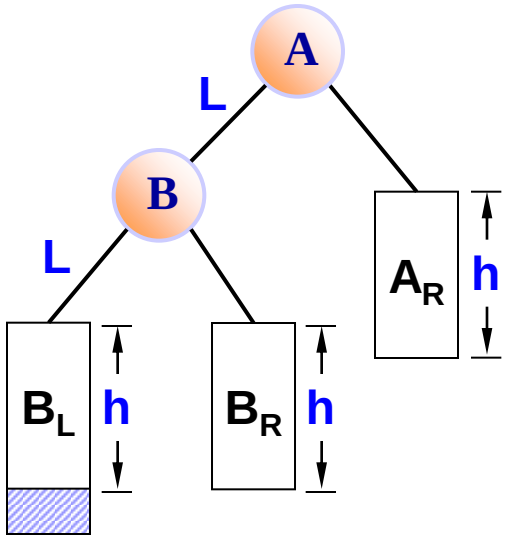
## (1) LL型调整

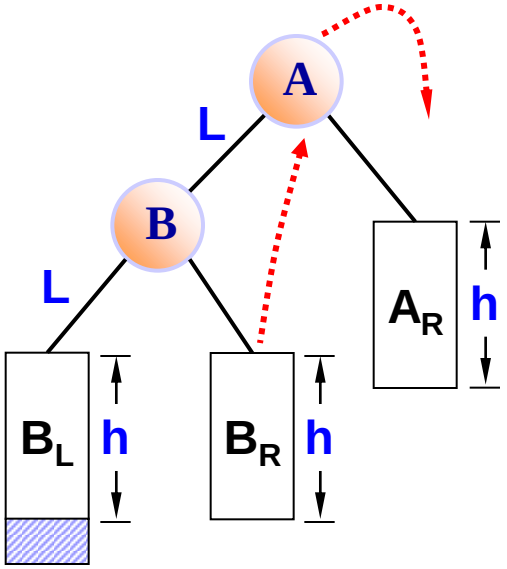
- A为最低不平衡结点，且插入前A->bf=1；
- 在A的左孩子（L）的左子树（L）上插入新结点，使A->bf=2，失去平衡，需调整。
- 如下两图所示。

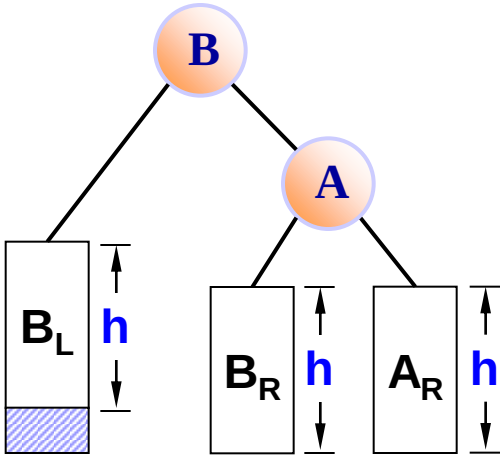
- 最简单LL型调整



# 一般LL型调整



- 针对这种情况的调整方法包括如下3部分：

① B的右子树调为A的左子树，即：

A->lChild=B->rChild;

② A调为B的右孩子，即：

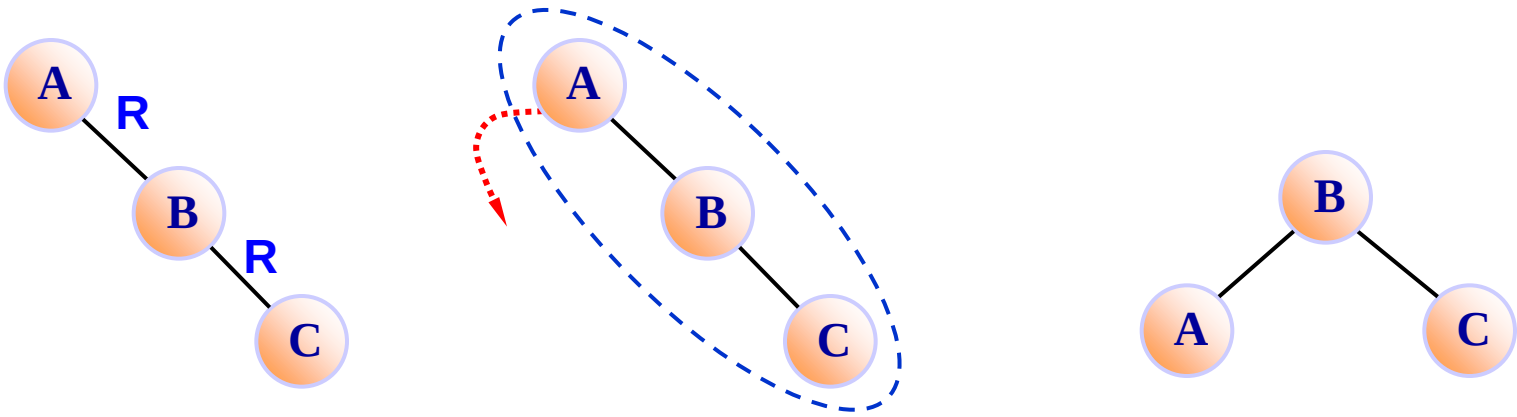
B->rChild=A；

③ B成为调整后的子树根结点。

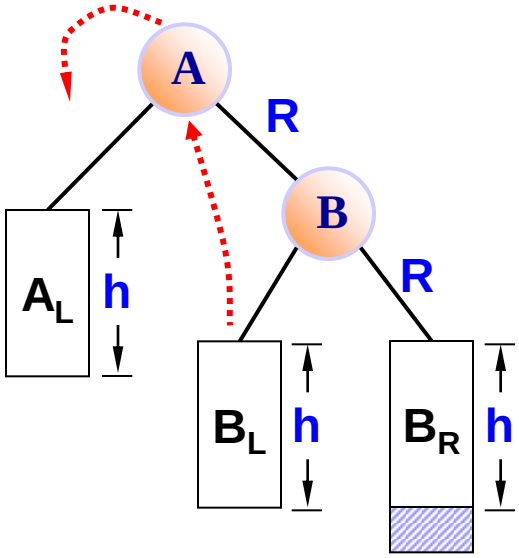
- 调整后的平衡因子：
  - A->bf=0; B->bf=0;
- 检查一下调整前后各个结点和子树的大小关系（中序序列）可知，调整操作不仅调整了各子树的高度以保持平衡，而且还保持了二叉排序树的特性，即中序序列有序的不变性。

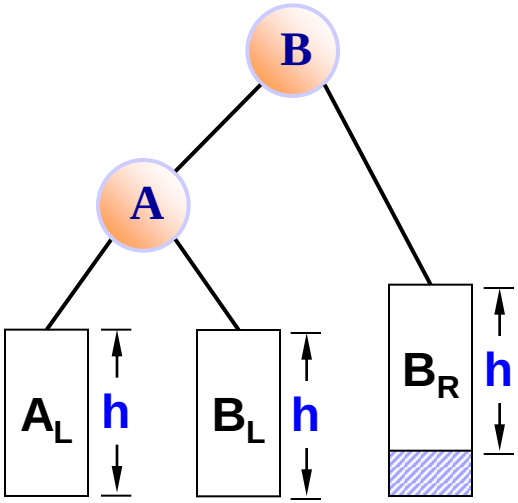
## (2) RR型调整

- A为最低不平衡结点，且A->bf=-1；
- 在A的右孩子（R）的右子树（R）上插入新结点，使A->bf=-2，失去平衡，需调整。
- 最简单RR型调整



# 一般RR型调整



- 针对这种情况的调整方法包括如下3个部分:

  ① B的左子树调为A的右子树，即：

  A->rChild=B->lChild；

  ② A调为B的左孩子，即：

  B->lChild=A；

  ③ B成为调整后的子树根结点。

- 调整后的平衡因子：

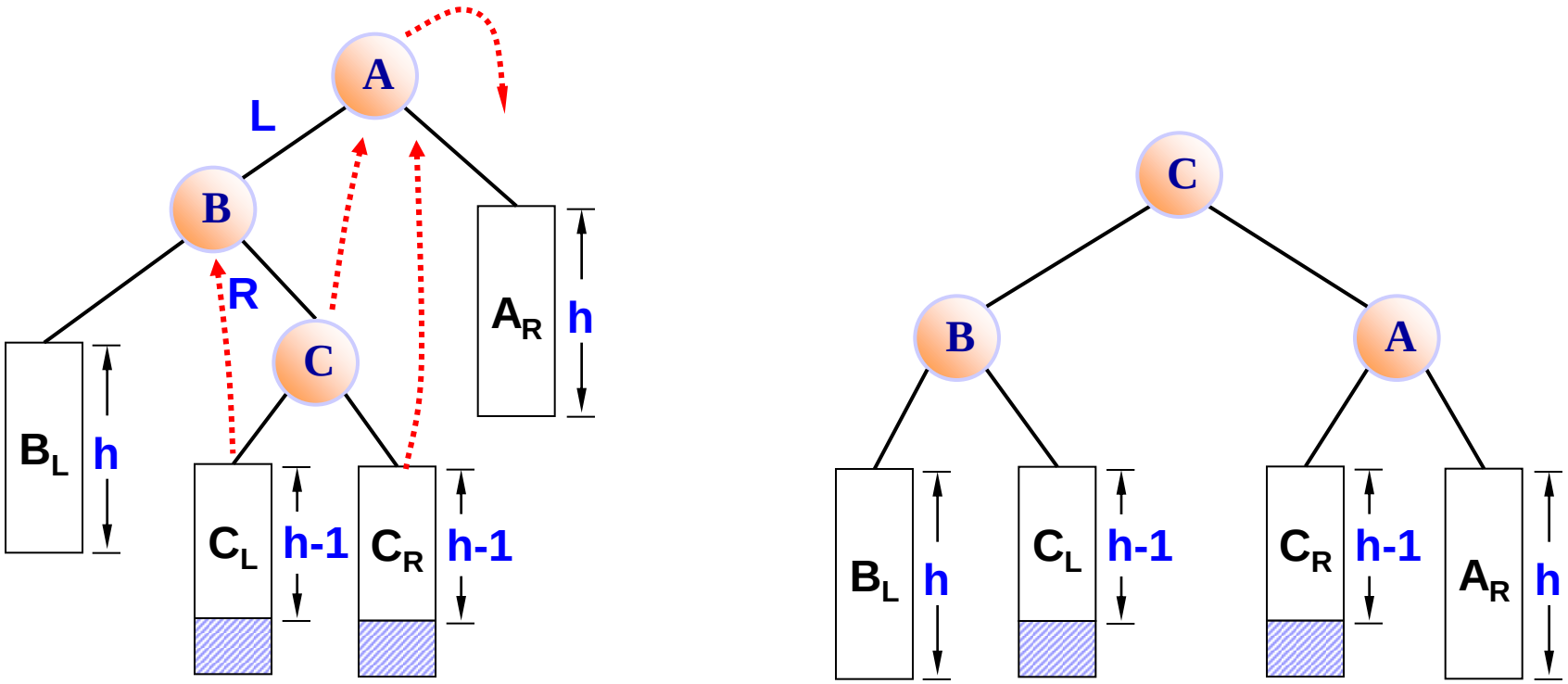
  ☞ A->bf=0; B->bf=0;

## (3) LR型调整

- A为最低不平衡结点，且A->bf=1；
- 在A的左孩子（L）的右子树（R）中插入新结点，使A->bf=2，失去平衡，需调整。

- 最简单的LR型调整

# ☞一般LR型调整



- 针对这种情况的调整方法包括如下3个部分：

① C的左子树调为B的右子树，即：

B->rChild=C->lChild;

C的右子树调为A的左子树，即：

A->lChild=C->rChild;

② B调为C的左子树，即：

C->lChild=B;

A调为C的右子树，即：

C->rChild=A;

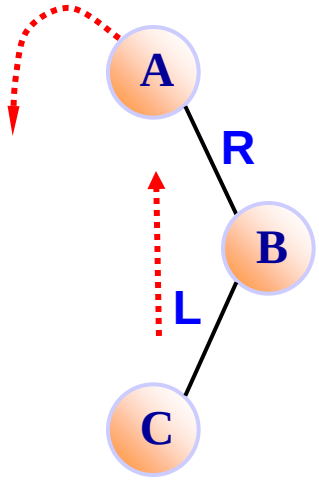
③ C成为调整后的子树根结点；

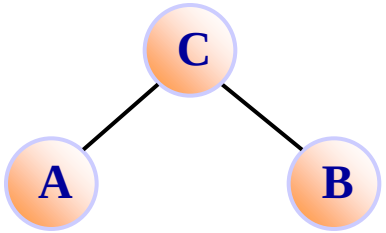
- 调整后的平衡因子：
  - 分为3种情况
  - ① 新结点插入到C的左子树上，使C->bf=1，调整后
    - A->bf=-1; B->bf=0; C->bf=0;
  - ② 新结点插入到C的右子树上，使C->bf=-1，调整后
    - A->bf=0; B->bf=1; C->bf=0;
  - ③ 新结点即是C自己，则C->bf=0（对应最简单的LR型），调整后
    - A->bf=0; B->bf=0; C->bf=0;

## (4) RL型调整

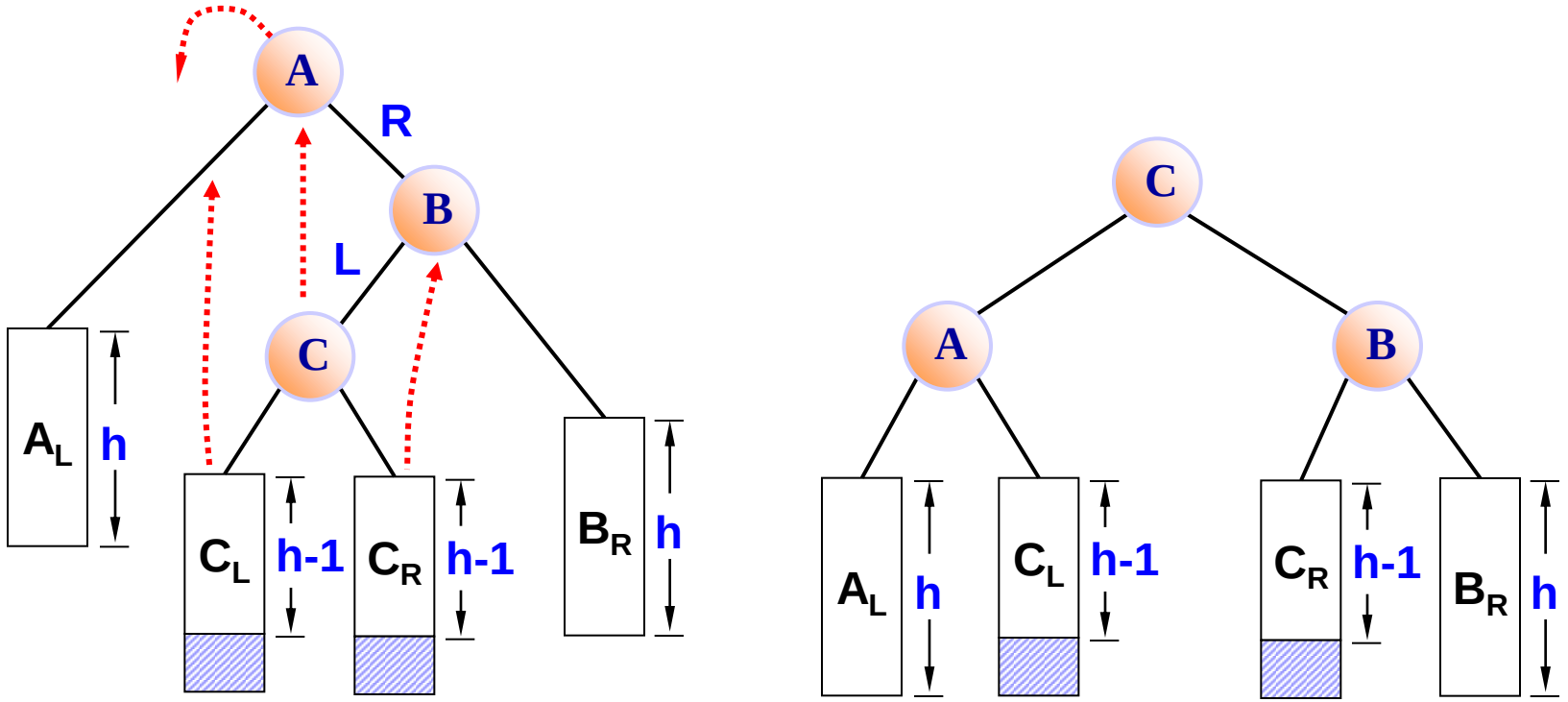
- A为最低不平衡结点，且A->bf=-1，
- 在A的右孩子（R）的左子树（L）中插入新结点，使A->bf=-2，失去平衡需调整 。

- 最简单的RL型调整



# 一般RL型调整



- 针对这种情况的调整方法包括如下3个部分：

① C的左子树调为A的右子树，即：

A->rChild=C->lChild;

C的右子树调为B的左子树，即：

B->lChild=C->rChild;

② A调为C的左孩子，即：

C->lChild=A;

B调为C的右孩子，即：

C->rChild=B;

③ C成为调整后的子树根结点。

- 调整后的平衡因子：
  - 分为3种情况
  - ①新结点插入到C的左子树上，使C->bf=1，调整后
    - A->bf=0; B->bf=-1; C->bf=0;
  - ②新结点插入到C的右子树上，使C->bf=-1，调整后
    - A->bf=1; B->bf=0; C->bf=0;
  - ③新结点即是C自己，则C->bf=0（对应最简单的RL型），调整后
    - A->bf=0; B->bf=0; C->bf=0;

- 平衡二叉排序树的高度为$\log_2 n$数量级
- 平衡二叉树插入、删除、查找等基本操作的时间复杂度为：$O(\log_2 n)$

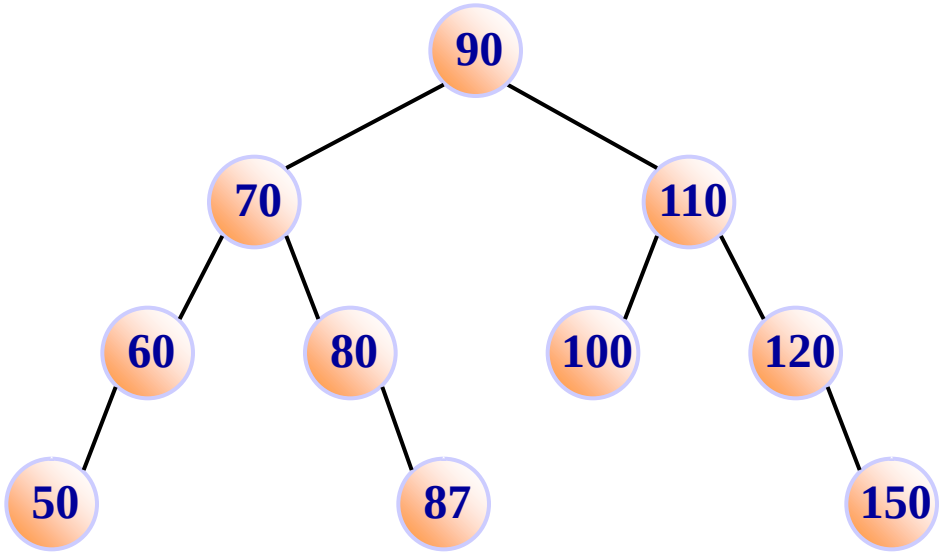
- 调整前后，中序序列不变。

## 【练习题】

1. 依次输入下列数据构造一棵平衡二叉树：

100，90，80，60，70，50，120，110，150，87。

【解】



2. 依次输入下列数据构造一棵平衡二叉树：

2，7，8，5，4，6，1

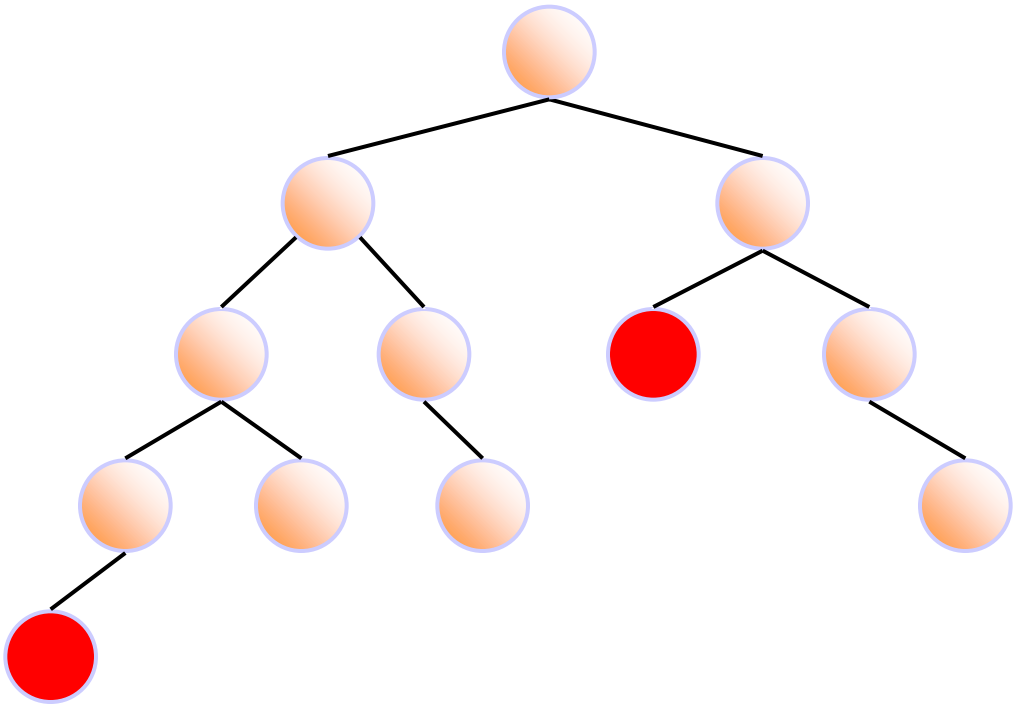
3. 至少要用7个数作为输入序列，构造一棵平衡二叉树，构造过程中同时包含4种调整方法。请给出这样的序列。

4. 在构造平衡二叉树的过程中，若A的平衡因子变为-2，A的左孩子的平衡因子为0，右孩子的平衡因子为1，则进行何种调整？

5. 一棵二叉树，已知每个结点的平衡因子bf，求二叉树的高度？（不一定平衡二叉树）

6. 一棵平衡二叉树中的叶子结点高度之差能不能超过2？

答：正确。



## 7. 求高度为n的平衡二叉树至少需要的结点个数。

【解】

☞ 设高度为n的平衡二叉树至少需要的结点个数为f(n)，则：

① 高度为0的平衡二叉树的至少需要0个结点，则f(0)=0;

② 高度为1的平衡二叉树的至少需要1个结点，则f(1)=1;

③ 高度为2的平衡二叉树的至少需要2个结点，则 f(2)=f(0)+f(1)+1=2;

④ 高度为3的平衡二叉树,当左右子树高度绝对差为1时，所需结数最少，则f(3)=f(1)+f(2)+1;

⑤ 则根据归纳法得，当n≥2时f(n)=f(n-1)+f(n-2)+1

所以：

$$f(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ f(n-1)+f(n-2)+1 & n\geq 2\end{cases}$$

单选题 1分

设置

链式存储结构一定优于顺序存储结构。

A 错

B 对

提交

# 7.3.3 B树

- 前面介绍的几种查找方法均适用于存储在计算机内存中的较小文件，统称内查找法。
- 当文件很大不能全部放入内存，用内查找法时需要反复进行内、外存交换（I/O），非常费时。
- 1970年，R.Bayer和E.Mccreight提出了B-树结构，适合外查找。
- B-树是适合外查找的平衡多叉树。
- 磁盘目录管理、数据库中的索引机制等多采用B-树结构。

## 1. B树的定义

一棵 m 阶B树，或为空树，或为满足以下条件的m叉树：

① 树中每个结点至多m-1个关键字，m棵子树；

② 若根结点不是叶子结点，则至少有1个关键字，2棵子树；

③ 除根结点之外，所有非终端结点（所有分支结点），至少有⌈m/2⌉-1个关键字，⌈m/2⌉棵子树；

④ 所有叶子结点都出现在同一层次上，并且不带信息，通常称为失败结点；

- 失败结点并不存在，指向这些结点的指针为空。
- 引入失败结点是为了便于分析B树的查找性能。

- B树结点结构

| n | $P_0$ | $K_1$ | $P_1$ | $K_2$ | $P_2$ | … | $P_{n-1}$ | $K_n$ | $P_n$ |
|---|---|---|---|---|---|---|---|---|---|

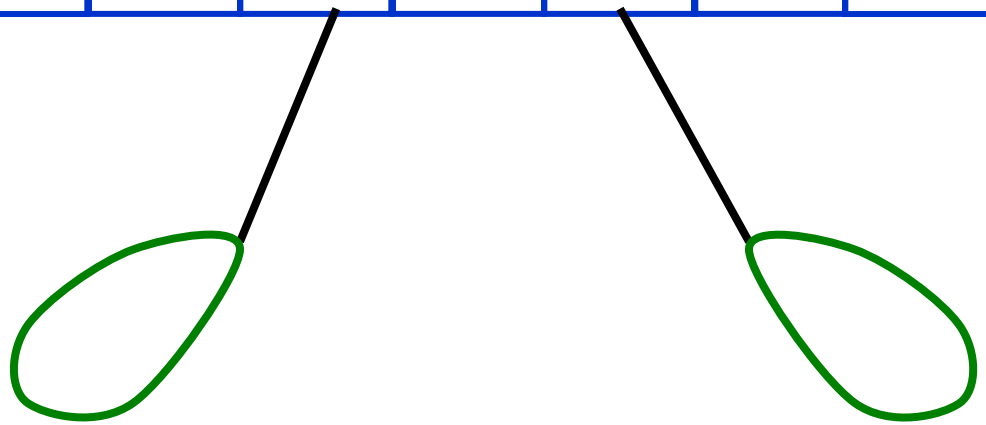
- 其中：
  - $K_i$ ( i=1,…,n )为关键字，且 $k_i < K_{i+1}$ ( i=1,…,n-1)，即每个结点的关键字是递增序列。
  - $P_i$ ( i=0,…,n )为指向子树根结点的指针，且$P_{i-1}$ 指针所指子树中的所有结点的关键字均小于$K_i$ ( i=1,…,n )。

    $P_i$ 所指子树中所有结点的关键字均大于$K_i$。
  - n (⌈m/2⌉<=n<=m-1 ) 为关键字个数（或n+1为子树棵数）。

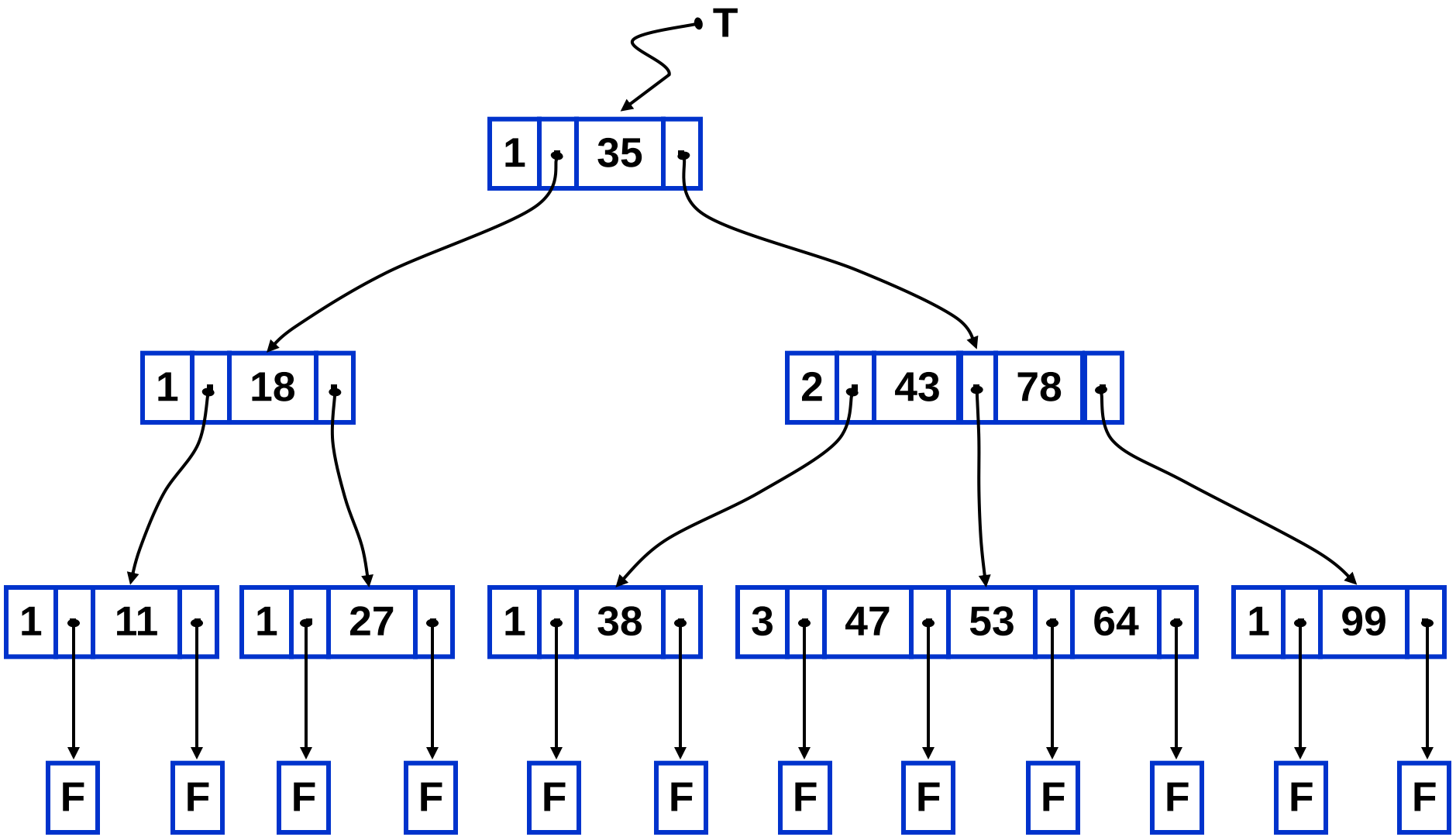
| n | $P_0$ | $K_1$ | $P_1$ | $K_2$ | $P_2$ | … | $P_{n-1}$ | $K_n$ | $P_n$ |
|---|---|---|---|---|---|---|---|---|---|

- 除了叶子结点，其它结点的指针数比关键字数多1。
- 对任一关键字$K_i$ 而言，$P_{i-1}$ 相当于指向其“左子树”，$P_i$ 相当于指向其“右子树”。

| n | $P_0$ | $K_1$ | … | $P_{i-1}$ | $k_i$ | $P_i$ | … | $K_n$ | $P_n$ |
|---|---|---|---|---|---|---|---|---|---|



- 例：一棵4阶B树



- **B树的特点**
  - ① 平衡—所有叶子结点都在同一层次，体现平衡性。
  - ② 有序--树中每个结点中的关键字都是有序的，且关键字$K_i$“左子树”中的关键字均小于$K_i$，而其“右子树”中的关键字均大于$K_i$，体现有序性。
  - ③ 多路—除叶子结点外，有的结点1个关键字，2棵子树；有的2个关键字，3棵子树；而4阶B_树最多3个关键字，4棵子树，体现多路性。

- **增加双亲指针**
  - ☞ 增加双亲指针指向其双亲结点，如图：

| parent | n | $P_0$ | $K_1$ | $P_1$ | $K_2$ | $P_2$ | … | $P_{n-1}$ | $K_n$ | $P_n$ |
|---|---|---|---|---|---|---|---|---|---|---|

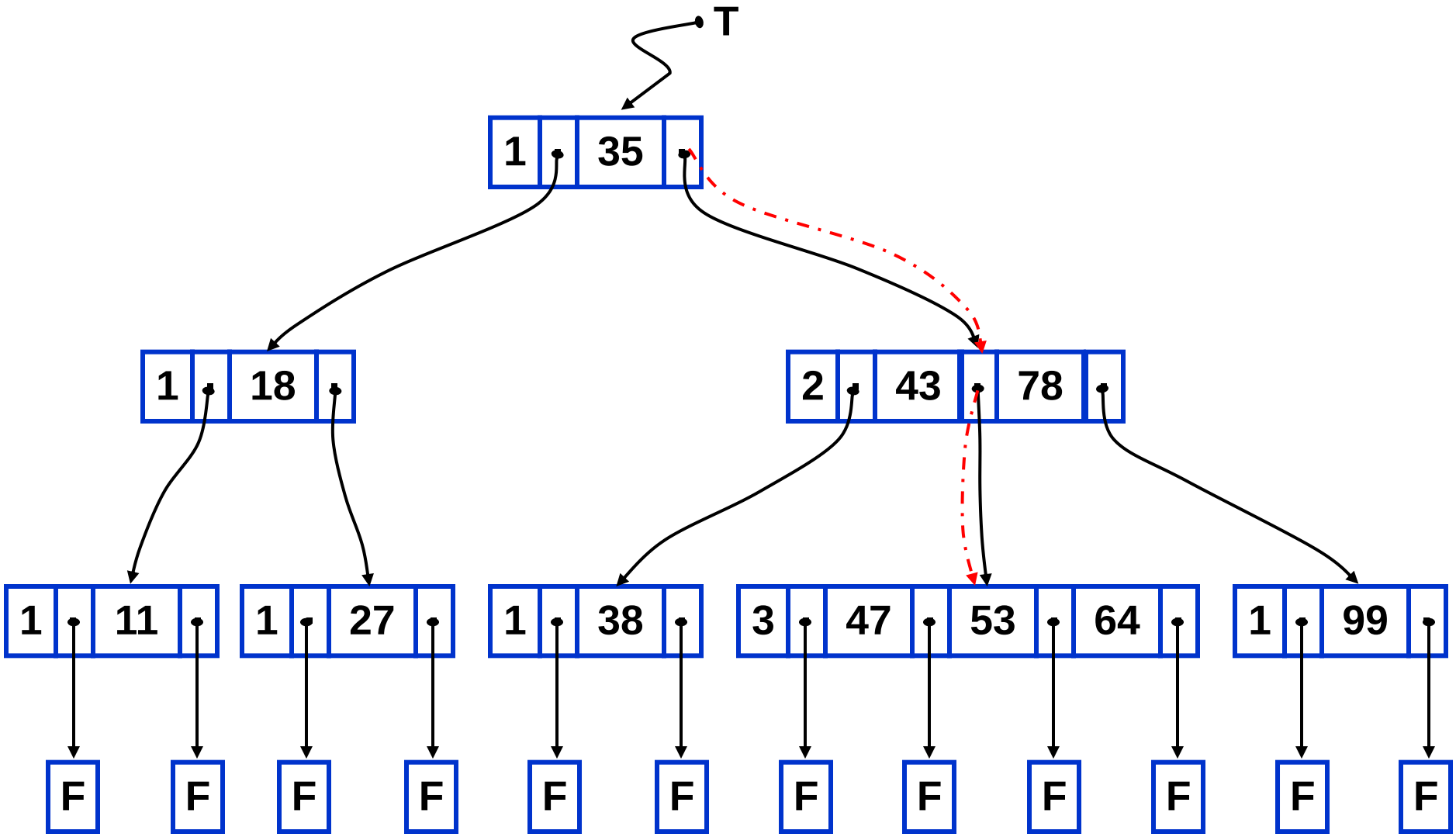
## 2. B树的存储

```
#define m 4   //暂设为4阶B树
typedef struct BTNode
{
  int keyNum;   //结点中关键字个数，即结点大小
  struct BTNode* parent;  //双亲指针
  KeyType key[m+1];  //保存关键字，0单元不用
  struct BTNode* ptr[m+1]; //子树指针，
  Record *recPtr[m+1];    //记录（元素）指针，0单
   元不用
}BTNode, *BTree;  //B_树结点和指针类型
```

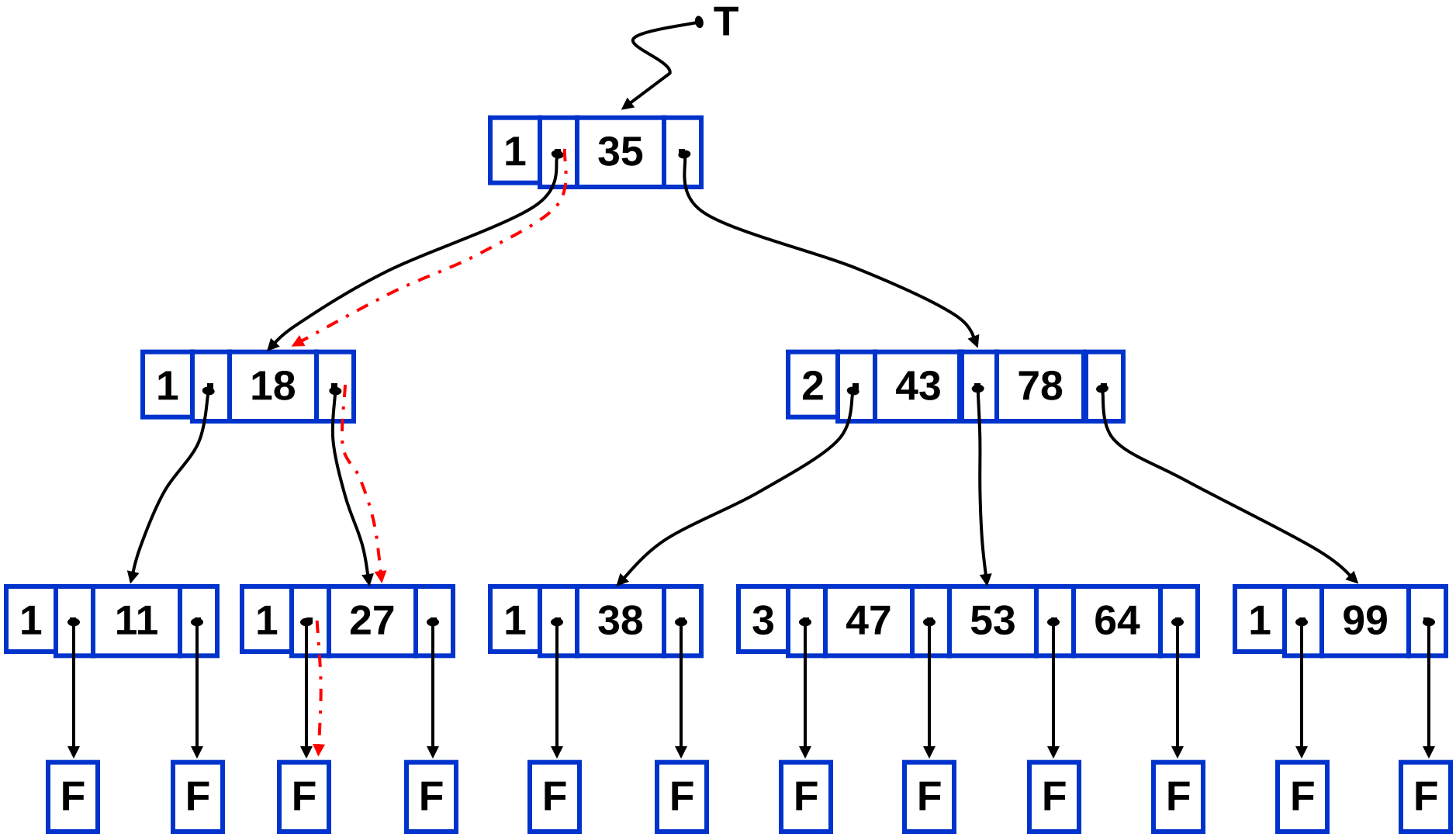
## 2. B树的查找

- 由定义可知B树查找和二叉排序树的查找类似。
- B树查找是顺着指针查找结点和在结点中进行关键字查找交叉进行的过程。

- 查找成功—例，下图中查找关键字47。



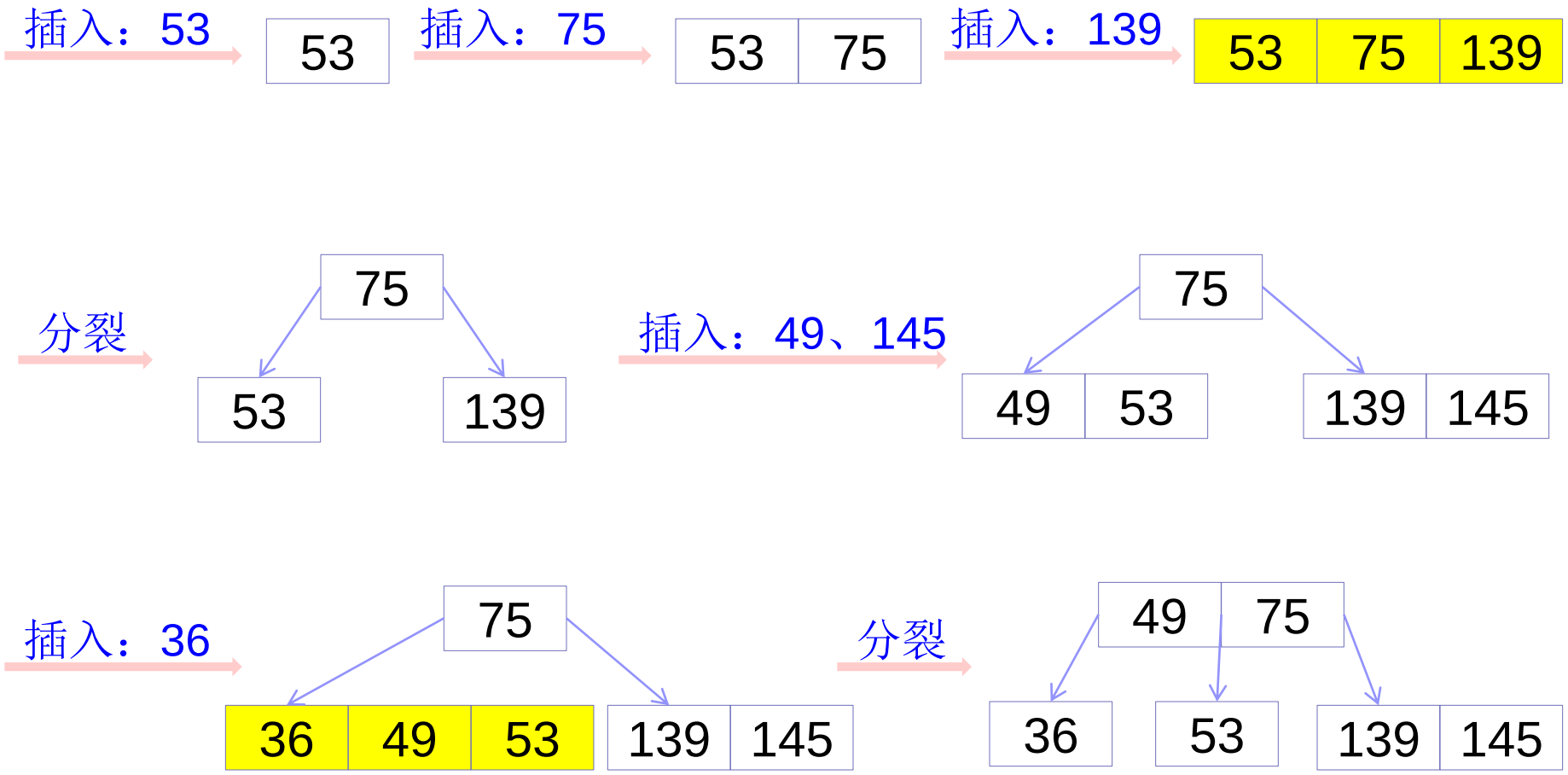
- 查找失败—例，下图中查找关键字23。



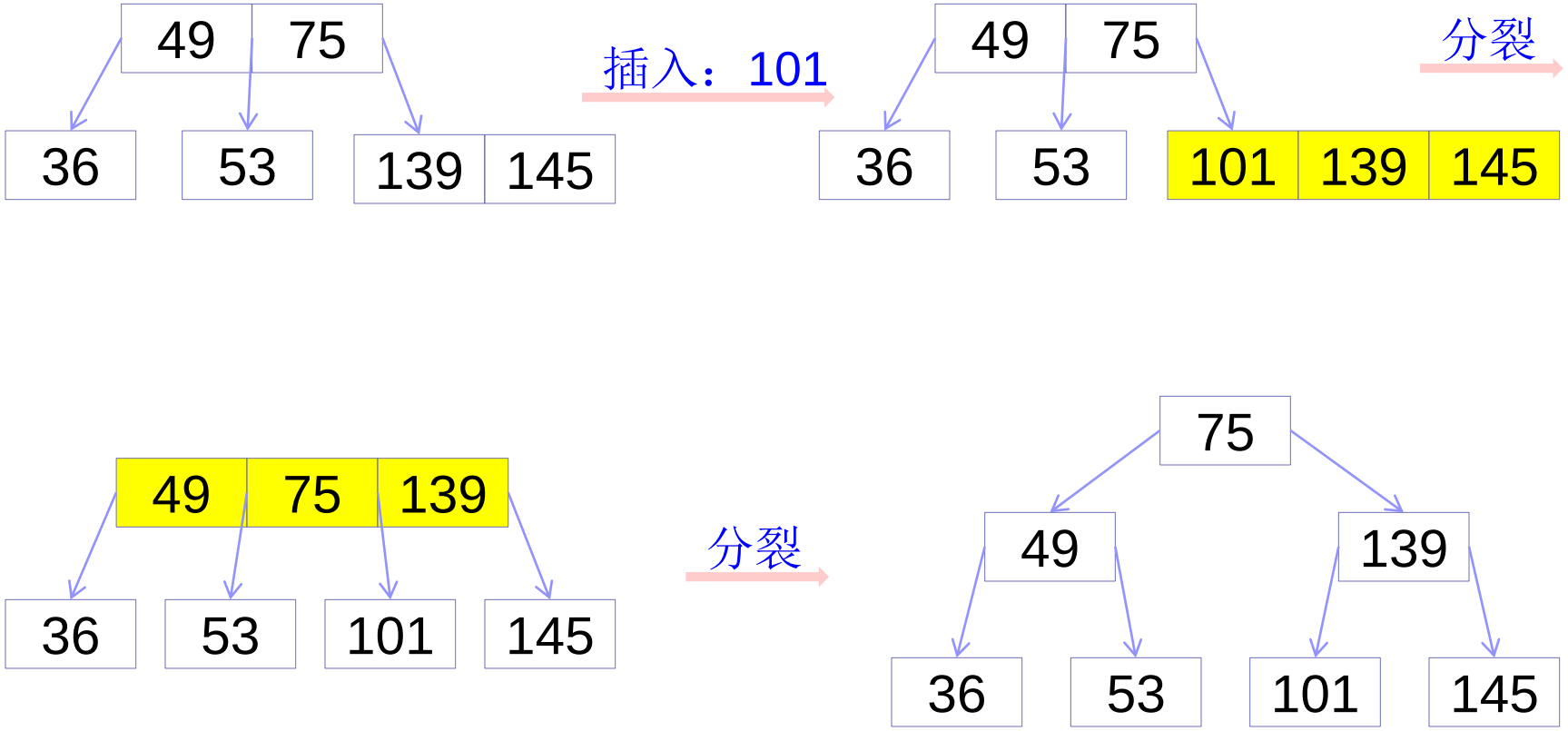
## 3. B树插入

- 按照有序性要求，找到最后一层合适位置插入；
- 此结点关键字数>=m时，进行“分裂”操作；分裂操作一直往上进行，直到产生新的根结点。

# B树插入操作示例（3阶B树）

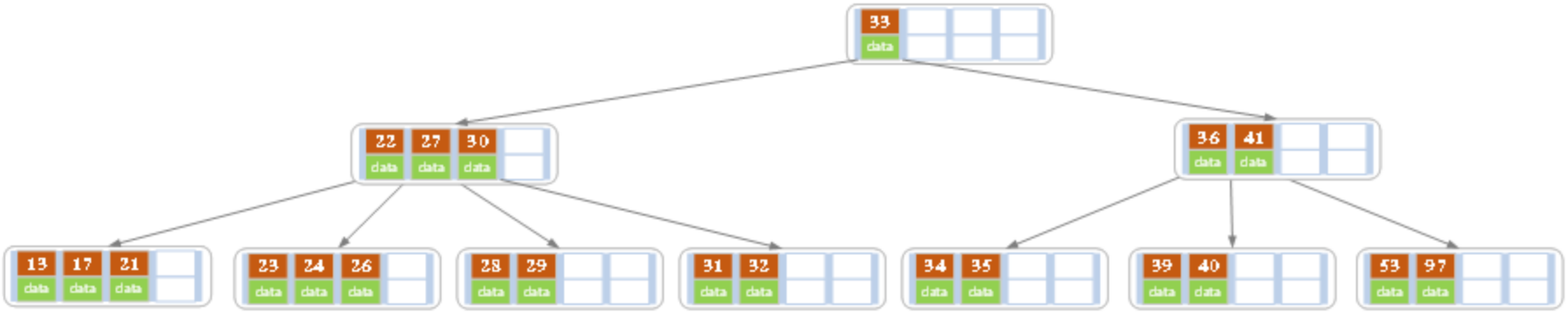


# B树插入操作示例（3阶B树）



## 练习：5阶B树

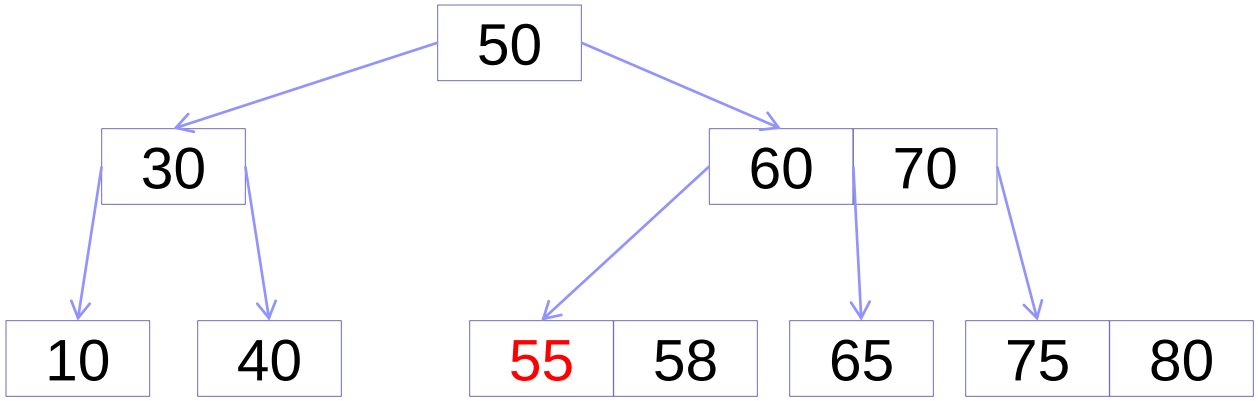
- 插入： 39、22、97、41、53、13、21、40、30、27、 33、36、35、34、24、29、26、17、28、31、32



## 4. B树删除

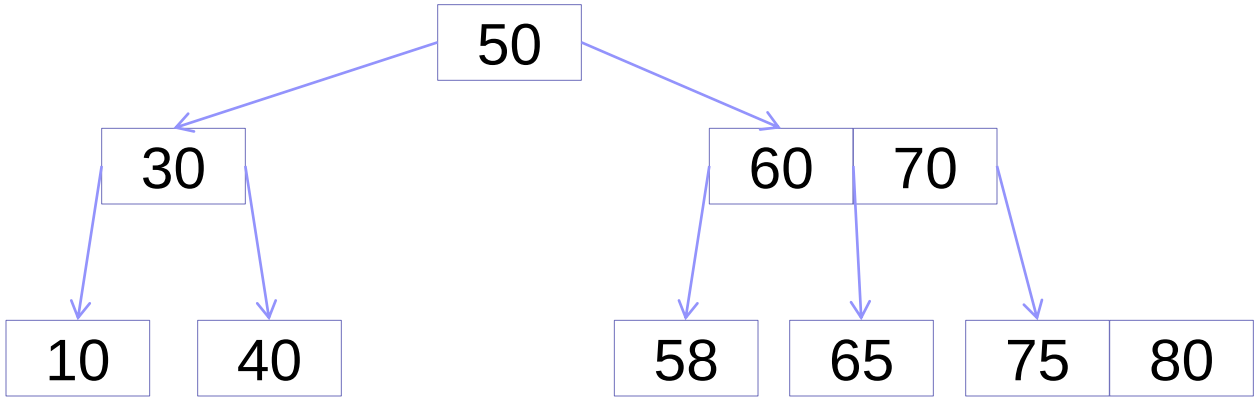
- 找到目标关键字位置；
- 找左子树最后一层的直接前驱（或找右子树上的直接后继），替代要删除关键字；
- 删除左后一层的直接前驱（直接后继）；
- 当此结点关键字数<⌈m/2⌉-1时，通过“借用”和“合并”，确保满足定义。合并一直往上进行，直到根结点。

# B树删除操作示例（3阶B树）

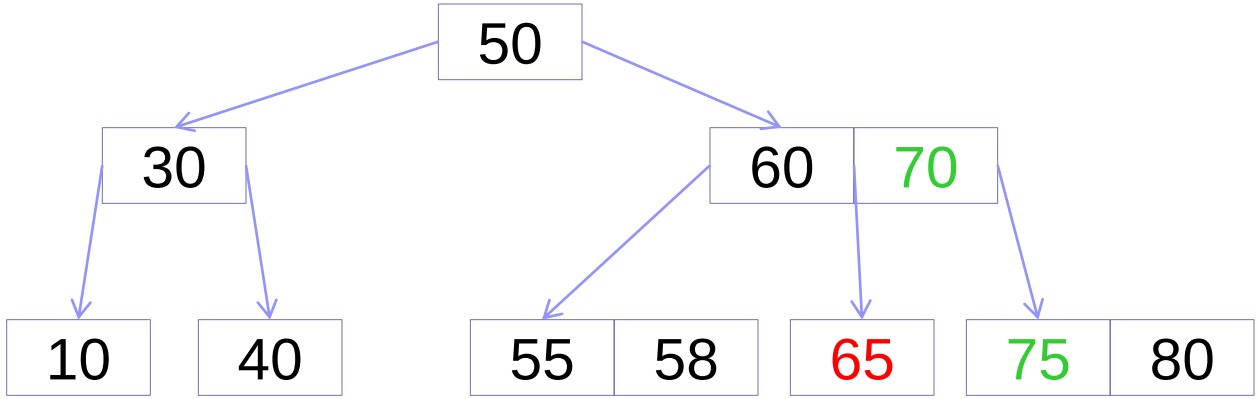



删除：55

简单删除

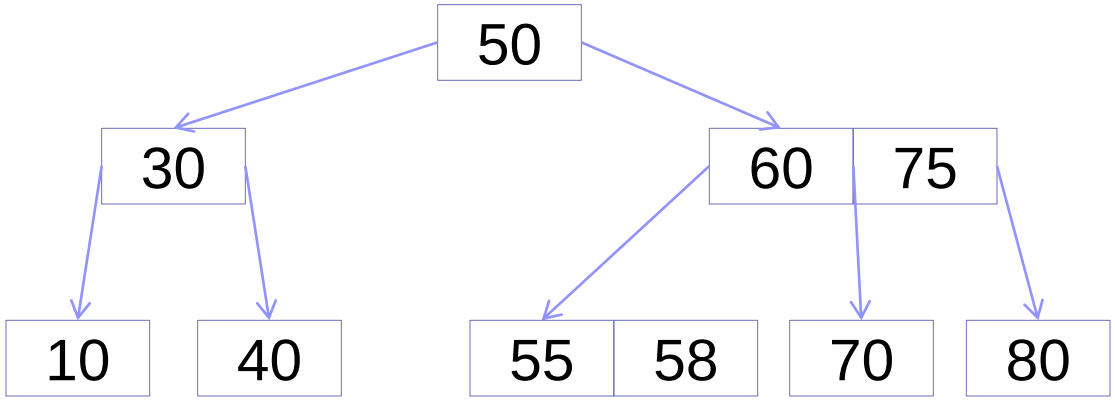


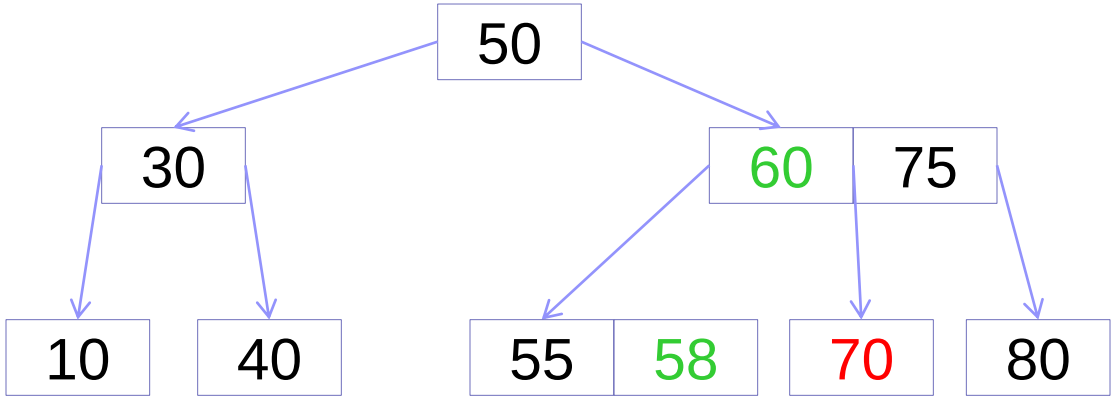
# B树删除操作示例（3阶B树）--恢复原树




删除：65

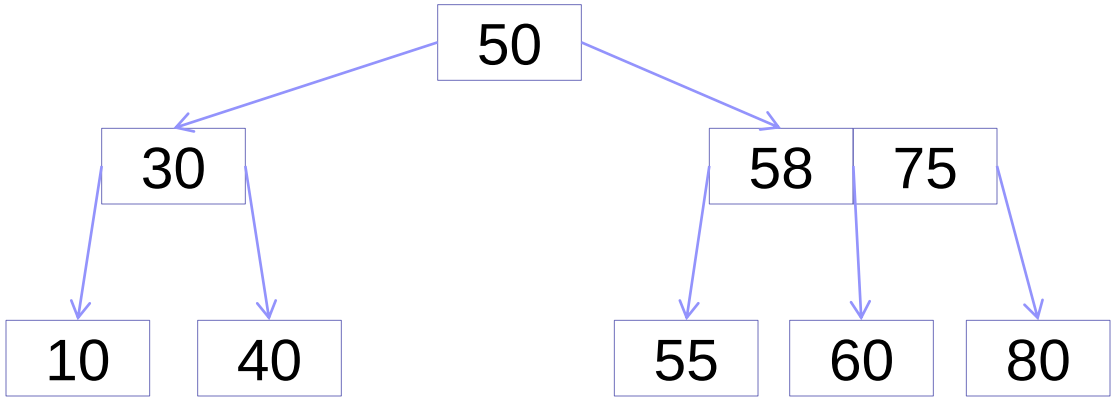
右借关键字

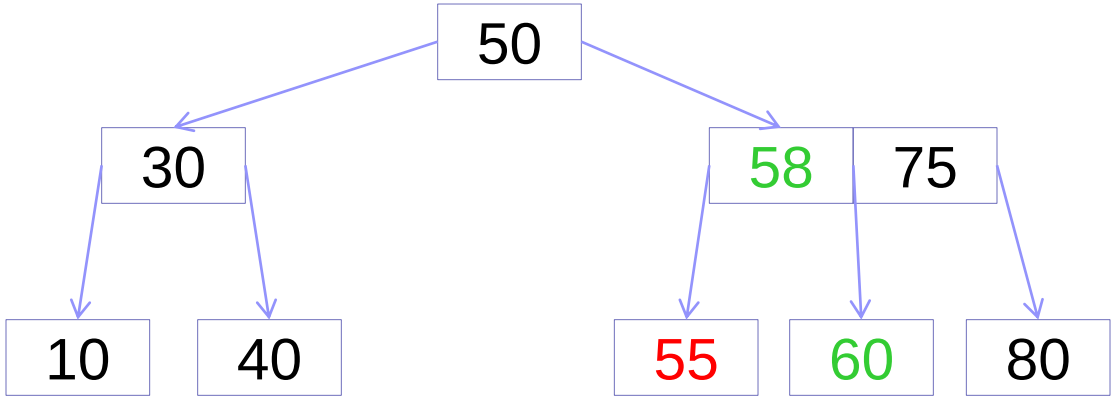





删除：70

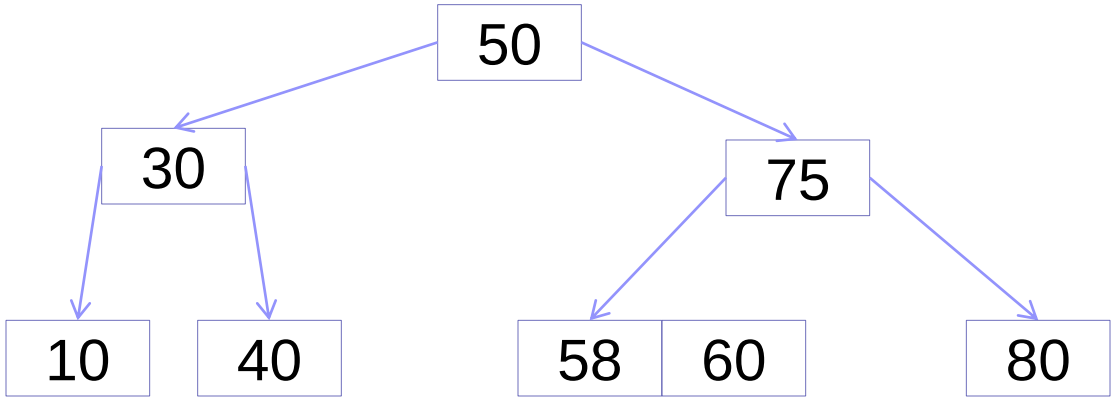
左借关键字

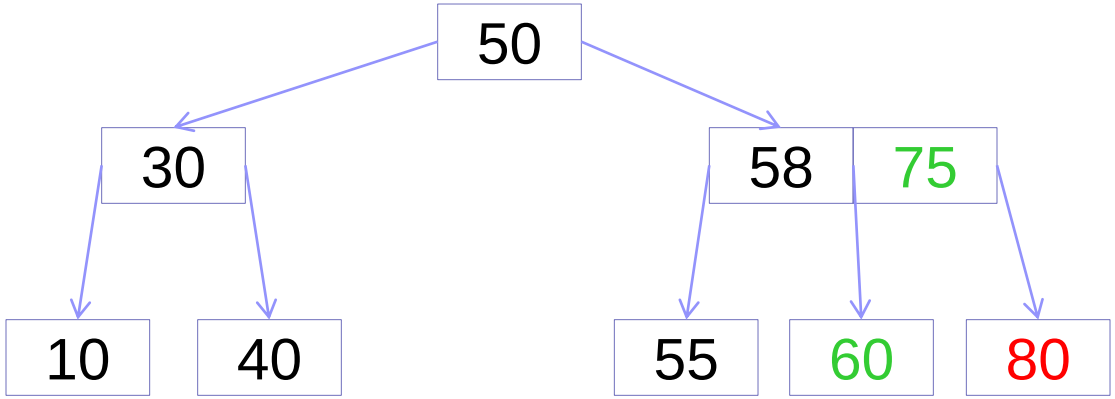





删除：55

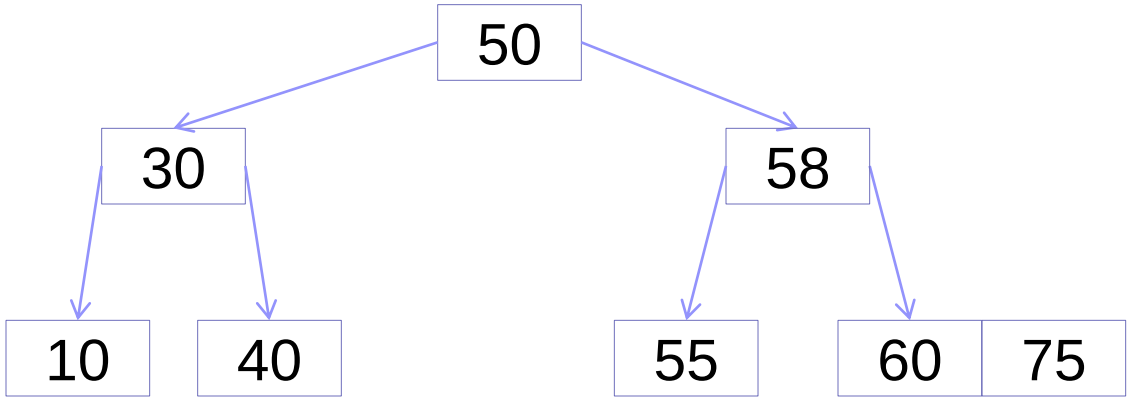
合并结点



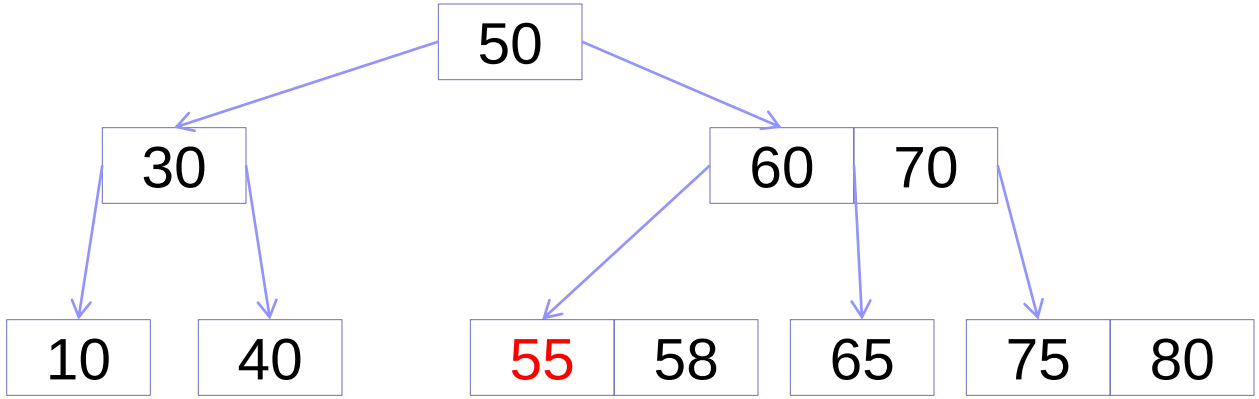



删除：80

合并结点



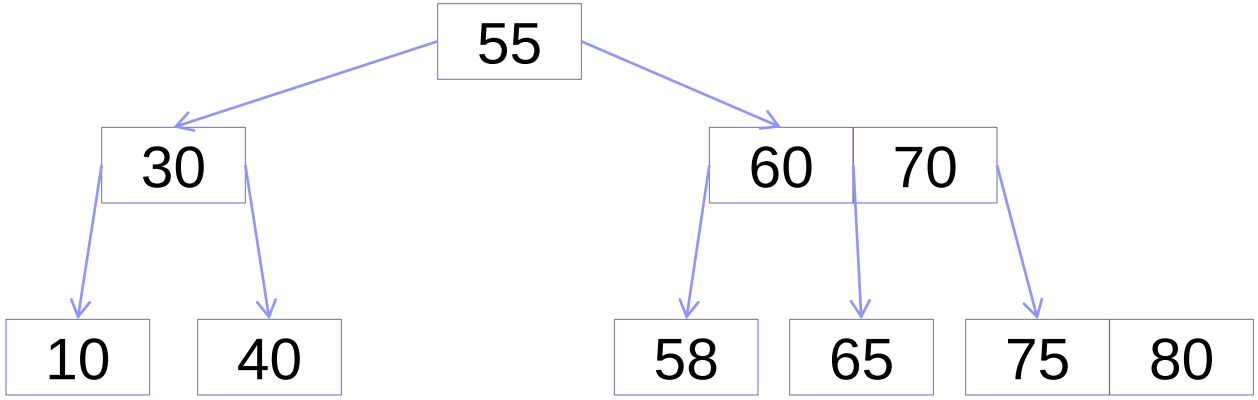
# B树删除操作示例（3阶B树）--恢复原树

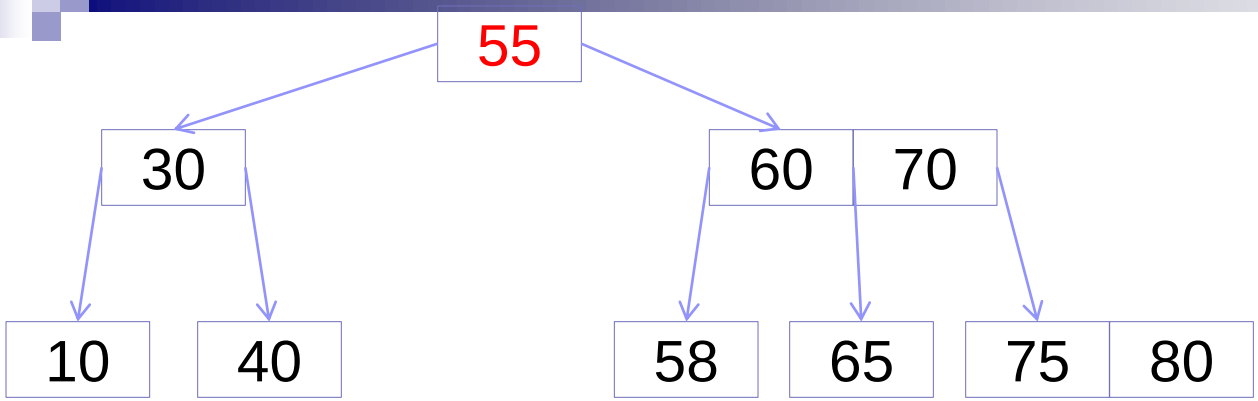



删除：50

后继顶替

删除后继

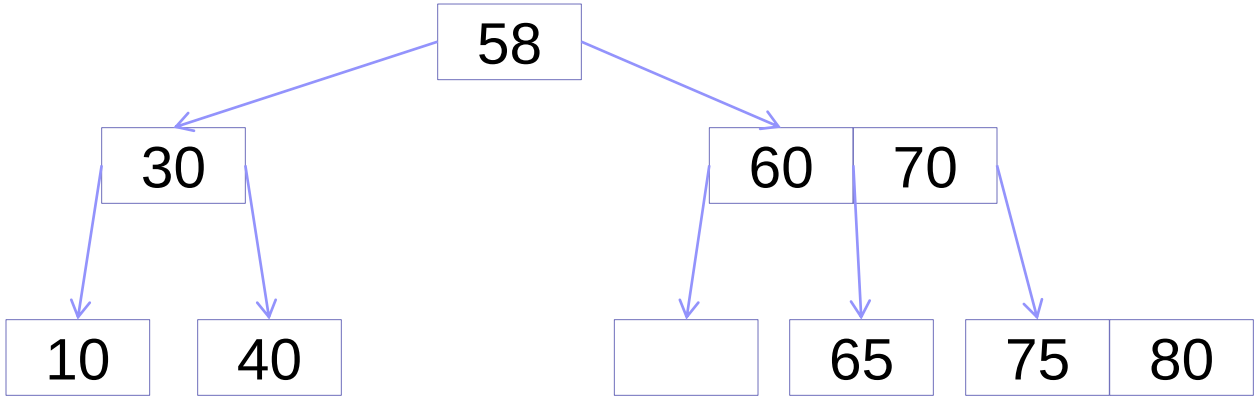




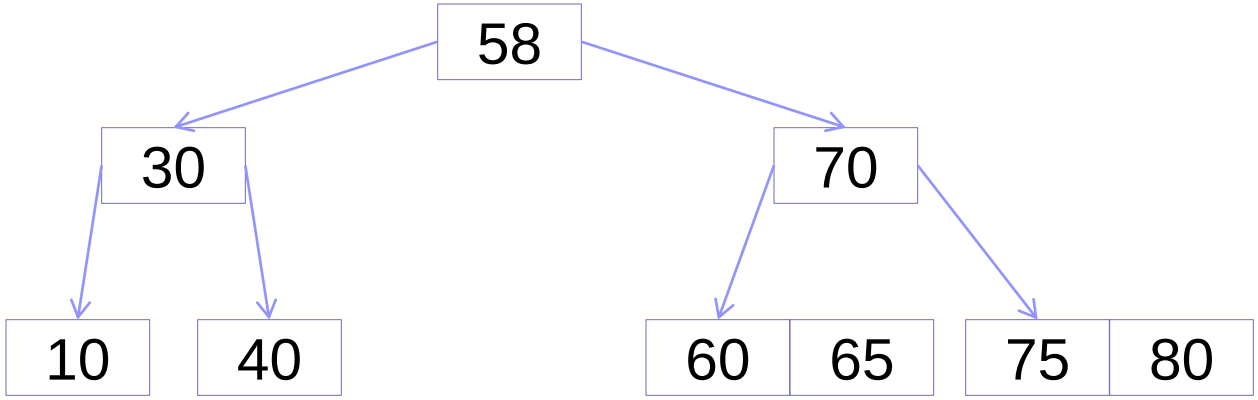

删除：55

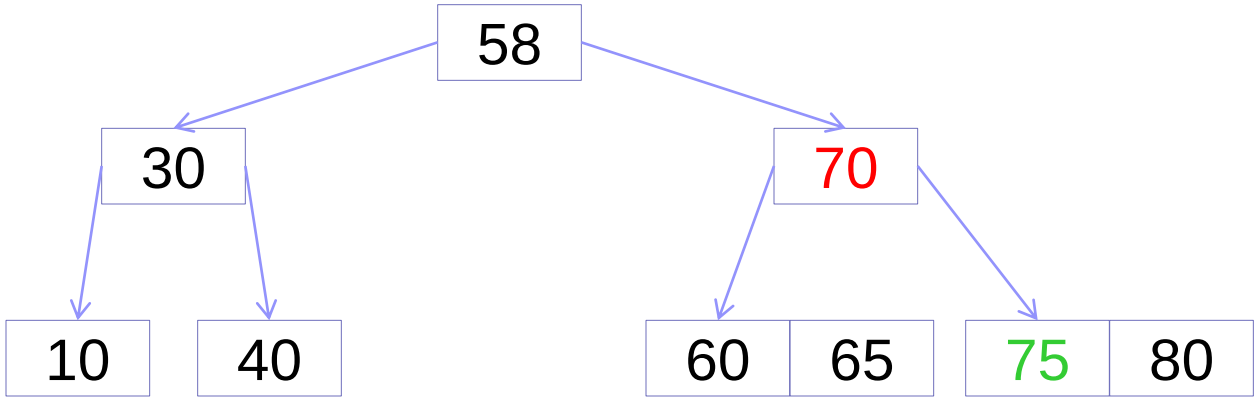
后继顶替

删除后继




合并结点

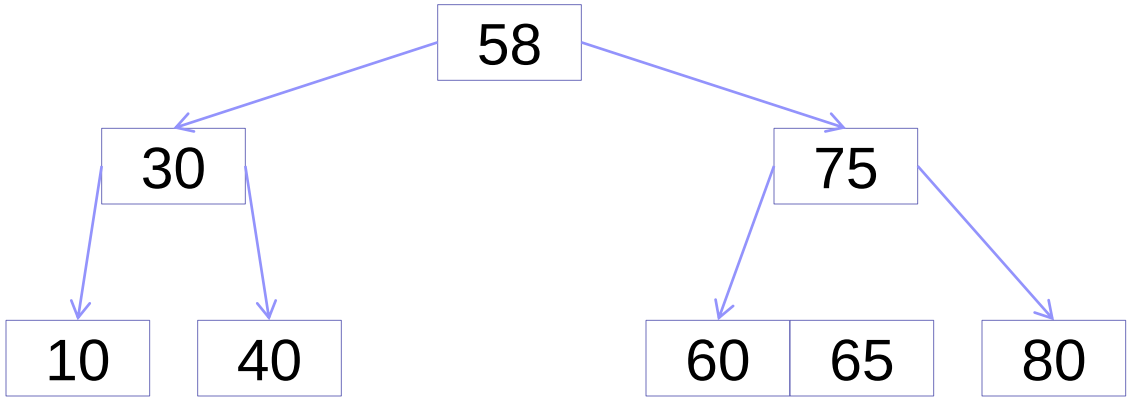


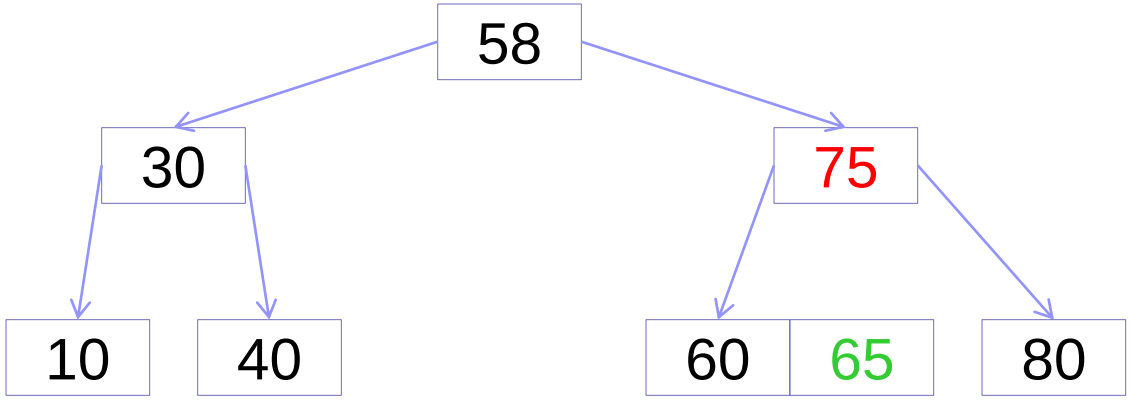



删除：70

后继顶替

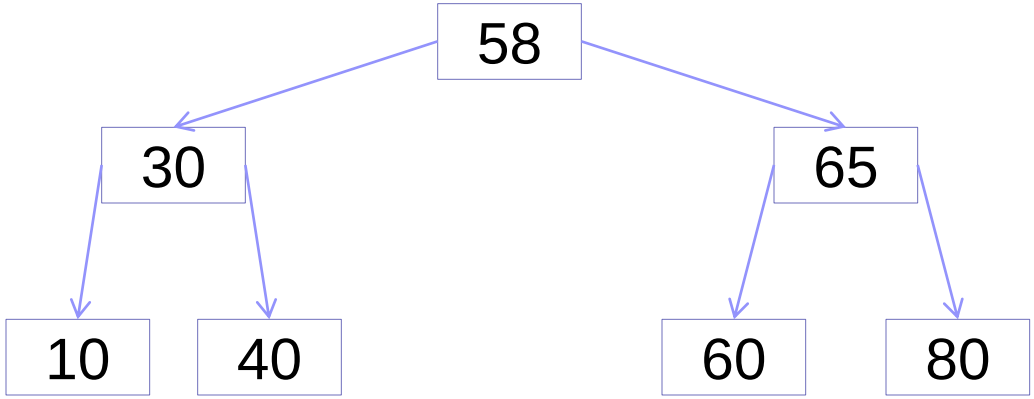
删除后继






删除：75

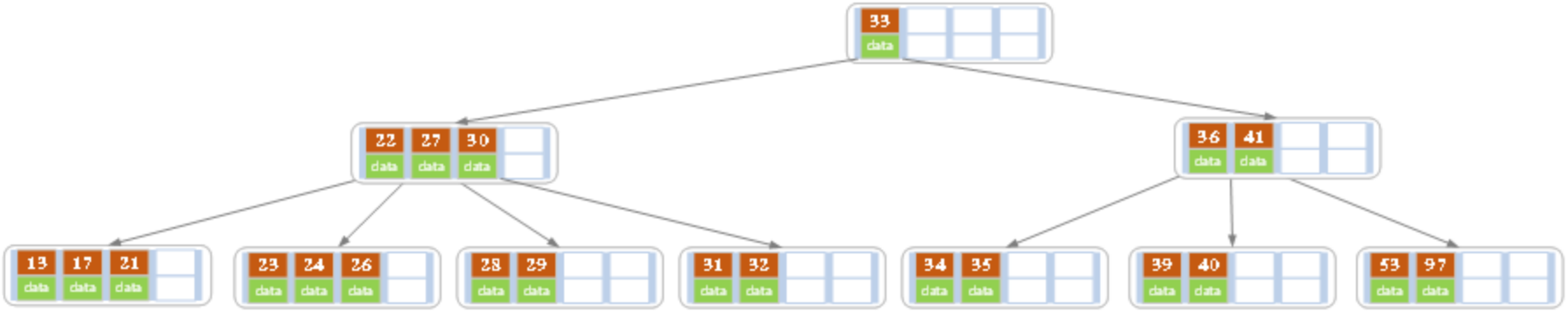
前驱顶替
删除前驱

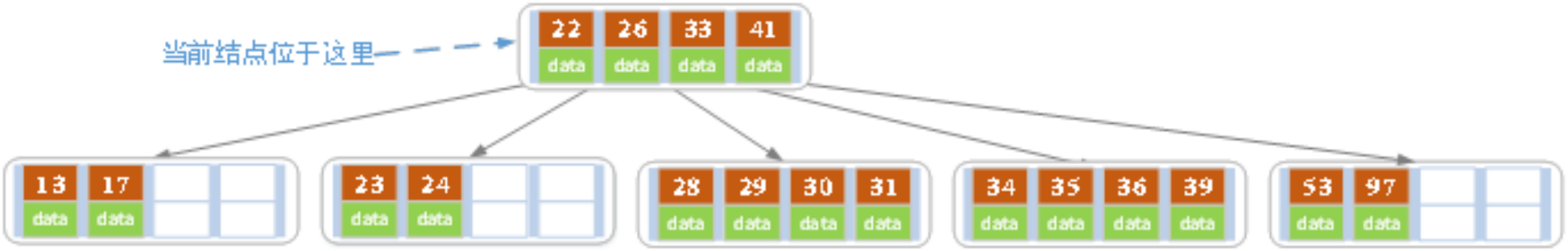



思考问题：

继续删除呢？

- 练习：在前面创建的5阶B树上
  - 删除： 21、27、32、40、



- 其它树表
  - B+树
  - B-树
  - 红黑树
  - 2-3树
  - 线段树
  - trie树
  - …

人生，那七笑，最美：

1.被人误解的时候能微微一笑，是一种素养；

2.受委屈的时候能坦然一笑，是一种大度；

3.吃亏的时候能开心一笑，是一种豁达；

4.无奈的时候能达观一笑，是一种境界；

5.危难的时候能泰然一笑，是一种大气；

6.被轻蔑的时候能平静一笑，是一种自信；

7.失恋的时候能轻轻一笑，是一种洒脱。

# 7.4 散列表的查找

- 前面两类表的查找均是基于比较运算来实现的，即通过比较元素的值来确定下一次查找的位置。

- 然而，在现实中，有很多查找是直接通过计算来实现的，即对给定的关键字**key**，用一个函数**H(key)**来计算出该关键字所标识元素的地址。

- 散列表的应用
  - 查找、搜索、摘要、数字签名、信息安全、网络等

# 7.4.1 散列表的基本概念

## 1. 散列表（哈希表）--Hash Table

- 给定关键字key，用一个函数H(key)计算出元素地址，由此而得散列表（Hash表，哈希表）。
- 其中，函数H(key)称为散列函数，
- 此函数值称为散列地址。

- 著名的散列函数

  MD5，SHA等

## 2. 冲突和同义词

- 对不同的关键字，key1和key2，即key1≠key2，但出现H(key1)=H(key2)的情况，称这种现象为冲突（Collision，碰撞）。
- 此时，称，key1，key2为同义词。

## 3. 装填因子

- 元素占表空间的比例。
- 一般选择在0.7-0.85之间。

- 针对散列表主要要解决如下问题
  - 如何构造好的散列函数
  - 如何处理冲突

# 7.4.2 构造散列函数的基本方法

## 1. 直接定址法：

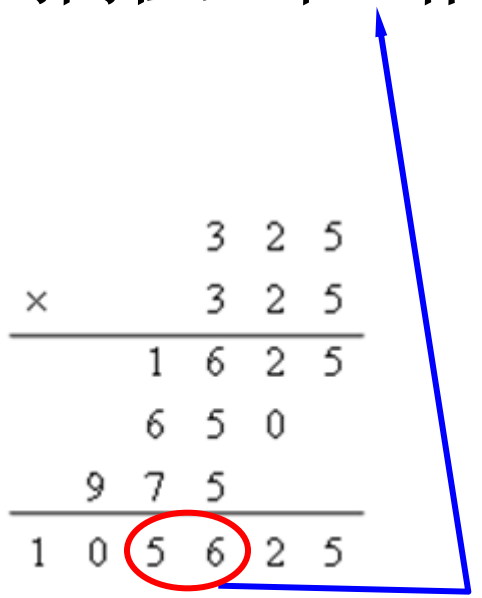
- **H(key)= key 或者 H(k)= a*key+b （a，b为任意正整数）**

## 2. 除留余数法：

- **H(key)= key % p**
- **其中，p≤m，m为数组规模（表长度），p为不大于m的整数。P最好为小于m的最大质数。例，m=100，p=97。**
- 除留余数法是简单、常用的一种构造方法。
- **p取素数（质数）可减少冲突的发生。**

## 3. 平方取中法：

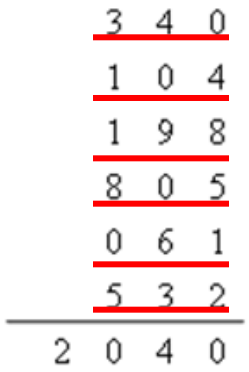
- 例：325在平方后取105625中间两位，即56作为它的散列地址。

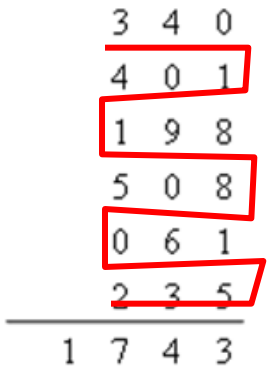


## 4. 折叠法：

- 如关键字为身份证号码：34010419880506153 2
- 假定表长度为1000，散列地址在0到999之间即可。
- 我们可对身份证号码，按3位一组进行分组：340 104 198 805 061 532。
- 最后一组不足3位可补0；
- 将每组数值相加，和作为散列地址。如果叠加和超出3为，可舍去进位，或通过摸运算处理。
- 有多种处理方式。

## 5. 数值分析法



# 7.4.3 冲突处理方法

- 一个冲突实例
  - 设散列表地址范围为0—9，散列函数 H(key)=key%7，采用线性探测法将下列数据依次插入下表中： 23,34,56,12,8,14,35,25

14？ 需处理冲突！

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 56 | 8 | 23 | | | 12 | 34 | | | |
| 1 | 1 | 1 | | | 1 | 1 | | | |

搜索次数

- 处理冲突方法：
  ① 开放地址法　$H_i(key)=(H(key)+d_i)\ \%\ m$，$i=1,2,\ldots,q\quad q<=m$
  ② 拉链法
  ③ 再散列法
  $\rightarrow H(key)\rightarrow H_1(key)\rightarrow H_2(key)\rightarrow \ldots\ldots \rightarrow H_i(key)$

## 1. 开放定址法

- $H_i(key)=(H(key) + d_i) \% m$，i=1,2,…,q，q<=m

## (1) 线性探测法：

- $H_i(key)=(H(key)+i)\%m$，m为表长度。
  $d_i=i=1,2,3,\ldots,m-1$
- 即H(key)冲突，从此地址开始逐个往后搜寻空位置，搜到最后仍未找到空位，再循环从表头搜索空位。
- 找到空位，将元素放入。
- 如果表满，找不到空位要做溢出处理。

【例7.1】前例散列表地址范围为0—9，散列函数H(key)=key%7，采用线性探测法将下列数据依次插入下表中：23,34,56,12,8,14,35,25

【解】

在表的下方给出了找到目标位置的搜索次数，这也是各元素查找长度（次数）。

所以，ASL=(1+1+1+4+5+1+1+4)/8=18/8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 56 | 8 | 23 | 14 | 35 | 12 | 34 | 25 | | |
| 1 | 1 | 1 | 4 | 5 | 1 | 1 | 4 | | |

搜索次数

## (2) 二次探测法

- $H_i(k) = ( H(k) \pm i^2 ) \% m$
- $d_i = \pm i^2 = \pm 1^2, \pm 2^2, \ldots, \pm q^2$（$q <= m/2$）
- 在原计算位置H(k)两边按平方数增长方式探测空位置。

## (3) 伪随机数法

- $d_i$=伪随机数序列
- 找到一个空位可能要多次产生伪随机数。

## 堆积（二次聚集）

- 探测找到的空位占据了正常H(key)计算的位置，即关键字不同，H(key)也不同，本来没有冲突，因为前面处理冲突而形成的新的冲突。
- 【例】散列表地址范围为0—9，散列函数H(K)=K%7，采用线性探测法插入：7,14,21,28,1,2,3
- 本来14和1是没有冲突的，但前面处理冲突时，14占据了1这个位置，这就是“堆积”现象。同样，在放置2、3时也有这个问题。

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 14 | 21 | 28 | | | | | | |

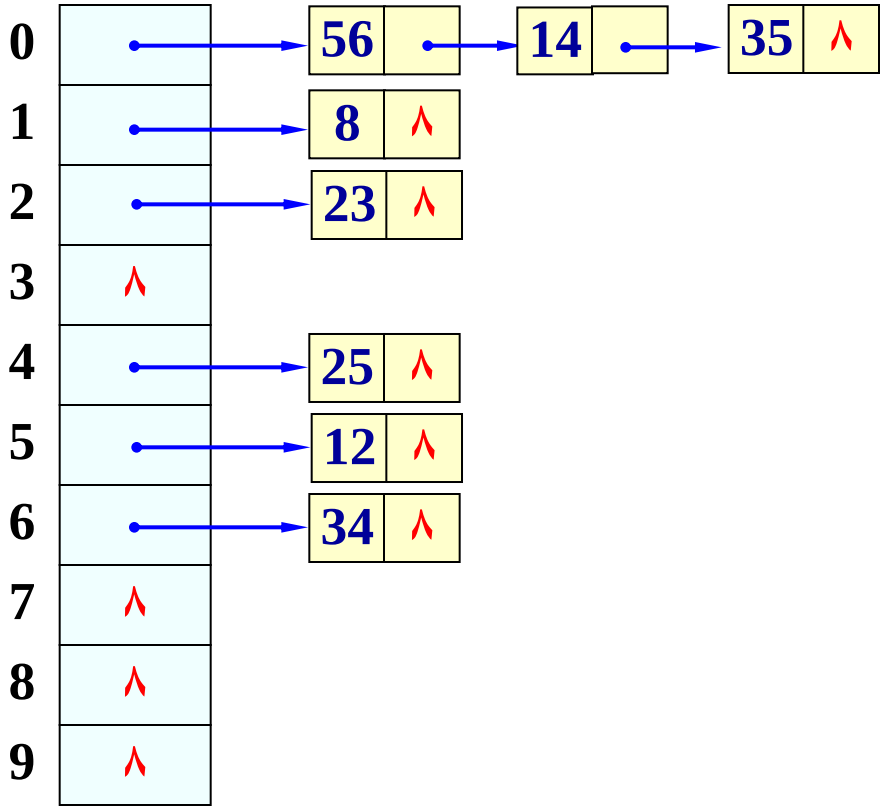
- 三种开放地址法比较
  - 线性探测法，
    - 优点：只要表未满，总能找到一个不冲突的空位。
    - 缺点：会产生“堆积”。
  - 二次探测和伪随机探测，
    - 优点：可较好避免“堆积”。
    - 缺点：不能保证找到不冲突的空位置。

## 2. 拉链法

- 也叫链地址法、链表法。
- 将具有相同散列地址（同义词、冲突）的记录放在一个单链表中，并将这些链表的头指针放到一个数组中。
- 类似图的邻接表结构，或树的孩子链表结构。
- 具体实现：根据关键字计算散列地址，按散列地址将关键字存入相应的链表。
  - 一个散列地址对应1条链表。

【例7.2】散列函数H(key)=key%7，采用拉链法将下列数据依次插入散列表中　23，34，56，12，8，14，35，25

【解】各链表第一个元素查找长度为1，第二个为2，依次类推，则ASL=(1*6+2*1+3*1)/8=11/8



# 7.4.4 散列表的查找

- 在散列表中查找元素的过程和构造的过程基本一致：

- **对给定关键字key，由散列函数H计算出该元素的地址H(key)。**
  - 若表中该位置为空，则查找失败。
  - 否则，比较关键字，若相等，查找成功；否则根据构造表时所采用的冲突处理方法找下一个地址，直至找到关键字等于key的元素（成功）或者找到空位置（失败）为止。
- **一般在用链地址法构造的表中进行查找，比在用线性探测法构造的表中进行查找，查找长度要小 — 冲突、堆叠**

## 【算法分析】

- 因为冲突，查找时仍需做关键字比较，仍以ASL作为查找性能的指标。

- **ASL取决于三个因素**
  - 散列函数
  - 处理冲突的方法
  - 装填因子
    - **α=表中装入的记录数/散列表的长度**
    - **α越大，装入的记录数越多，发生冲突的概率增加，查找次数增加。**

单选题 1分

设置

假设图G有n个顶点，e条边，采用邻接表存储，则DFS的时间复杂度为（　　）。

A. O(n)　　B. $O(n^2)$

C. O(n+e)　　D. $O(\log_2 n)$

A O(n)

B $O(n^2)$

C O(n+e)

D $O(\log_2 n)$

提交

单选题 1分

设置

一棵二叉树，中序遍历序与后续遍历序相反，则此二叉树（ ）。

A 每个分支结点只有左孩子。

B 每个分支结点只有右孩子。

C 每个分支结点只有一个孩子。

D 每个分支结点都有两个孩子。

提交

Thank you ！