



第九章 查找

机电工程与自动化学院 L栋301

任卫红 助理教授

renweihong@hit.edu.cn

<http://faculty.hitsz.edu.cn/renweihong>



9.1 查找的概念

第一节 查找的概念

一、查找表(Search Table)

- 查找表是由同一类型的数据元素(或记录)构成的集合
- 对查找表的操作:
 1. 查询某个“特定的”数据元素是否在查找表中;
 2. 检索某个“特定的”数据元素的各种属性;
 3. 在查找表中插入一个数据元素;
 4. 从查找表中删去某个数据元素

第一节 查找的概念

一、查找表(分类)

- **静态查找表**

仅作**查询和检索**操作的查找表。

- **动态查找表**

在查找过程中同时插入查找表中不存在的数据元素，
或者从查找表中删除已存在的某个数据元素

第一节 查找的概念

二、关键字(Key)

- **关键字**是数据元素（或记录）中某个数据项的值，用以标识（识别）一个数据元素（或记录）
- **主关键字**：可以识别唯一的一个记录的关键字
- **次关键字**：能识别若干记录的关键字

姓名	电话号码
陈海	13612345588
李四锋	13056112345
。 。 。	。 。 。

← 数据元素

← 数据项

第一节 查找的概念

三、查找(Searching)

- **查找**是根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素（或记录）。
 - **查找成功**：在查找表中**查找到**指定的记录
 - **查找不成功**：在查找表中**没有找到**指定记录

第一节 查找的概念

四、衡量查找算法的标准

- 时间复杂度
- 空间复杂度
- 平均查找长度ASL

第一节 查找的概念

五、平均查找长度 (ASL)

- 平均查找长度定义为确定记录在表中的位置所进行的和关键字比较的次数的平均值

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

- n 为查找表的长度，即表中所含元素的个数
- P_i 为查找第 i 个元素的概率 ($\sum P_i = 1$)
- C_i 是查找第 i 个元素时同给定值 K 比较的次数



9.2 静态查找表

第二节 静态查找表

一、顺序查找

- 顺序查找算法是顺序表的查找方法
- 在顺序查找算法中，以**顺序表**或**线性链表**表示静态查找表

第二节 静态查找表

一、顺序查找(算法)

- 顺序查找算法:

1. 从表中**最后一个记录**开始
2. 逐个进行记录的关键字和给定值的比较
3. 若某个记录比较相等, 则查找成功
4. 若直到第1个记录都比较不等, 则查找不成功

第二节 静态查找表

一、顺序查找(算法实现)

```
#define SIZE 100

class SSTable
{
    int    length;
    int    *elem;
public:
    SSTable();    //分配空间, length赋零
    ~SSTable();   //释放空间
    void InitSSTable();    //数据输入, 0空间不用
    int Search(int key);   //查找关键字key
};
```

第二节 静态查找表

一、顺序查找(算法实现)

```
int SSTable::Search(int key)
{
    elem[0] = key;    //哨兵

    //从后向前比较
    //查找成功返回位置, 否则, 返回0
}
//设置“哨兵”的目的是省略对下标越界的检查,
//提高算法执行速度
```

顺序查找举例：

elem

64	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

key=64

Length

elem

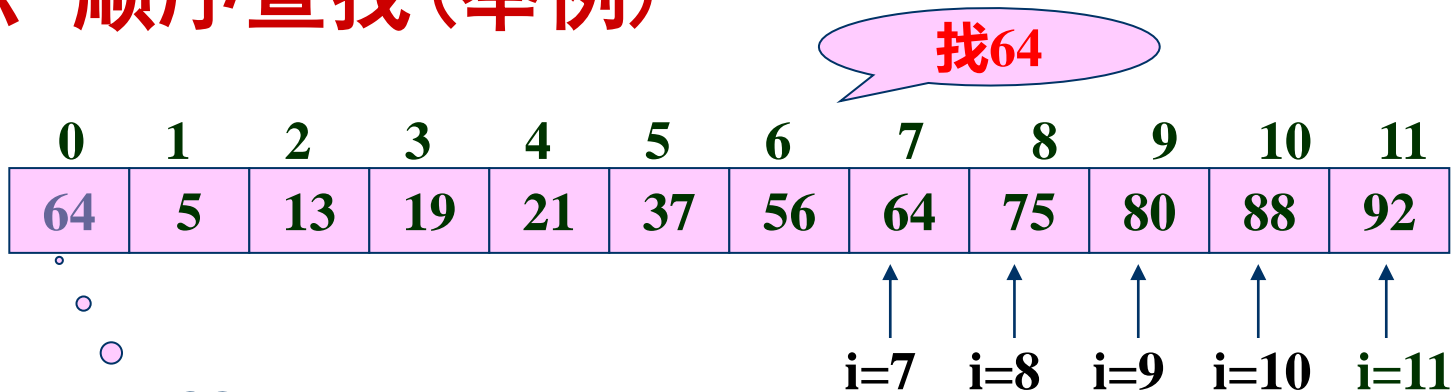
60	21	37	88	19	92	05	64	56	80	75	13	
0	1	2	3	4	5	6	7	8	9	10	11	

key=60

Length

第二节 静态查找表

一、顺序查找(举例)



0	1	2	3	4	5	6	7	8	9	10	11
64	5	13	19	21	37	56	64	75	80	88	92

找64

i=7 i=8 i=9 i=10 i=11

监视哨

比较次数:

查找第n个元素: 1

查找第n-1个元素: 2

.....

查找第1个元素: n

查找第i个元素: $n+1-i$

查找失败: $n+1$

比较次数=5

第二节 静态查找表

一、顺序查找(算法性能分析)

- 对顺序表而言, $C_i = n - i + 1$
- 在等概率查找的情况下, $P_i = 1/n$
- $ASL = n * P_1 + (n-1)P_2 + \dots + 2P_{n-1} + P_n = (n+1)/2$

此为查找成功时, 那么
查找不成功呢?

第二节 静态查找表

一、顺序查找(不等概率)

- 如果被查找的记录概率不等时，取

$$P_n \geq P_{n-1} \geq \dots \geq P_2 \geq P_1$$

- 若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上

第二节 静态查找表

一、顺序查找(特点)

- 优点:

1. 简单
2. 适应面广(对表的结构无任何要求)

- 缺点:

1. 平均查找长度较大
2. 特别是当 n 很大时, 查找效率很低

第二节 静态查找表

二、折半查找

- 折半查找算法是**有序表**的查找方法
- 在折半查找算法中，静态查找表按关键字大小的次序，有序地存放在**顺序表**中
- 折半查找的原理是：
 1. 先确定待查记录所在的范围（前部分或后部分）
 2. 逐步缩小（一半）范围直到找（不）到该记录为止

第二节 静态查找表

二、折半查找(算法)

1. n 个对象从小到大存放在有序顺序表ST中, k 为给定值
2. 设 low 、 $high$ 指向待查元素所在区间的下界、上界, 即 $low=1$, $high=n$
3. 设 mid 指向待区间的中点, 即 $mid=(low+high)/2$
4. 让 k 与 mid 指向的记录比较
若 $k=ST[mid].key$, 查找成功
若 $k<ST[mid].key$, 则 $high=mid-1$ [上半区间]
若 $k>ST[mid].key$, 则 $low=mid+1$ [下半区间]
5. 重复3, 4操作, 直至 $low>high$ 时, 查找失败。

第二节 静态查找表

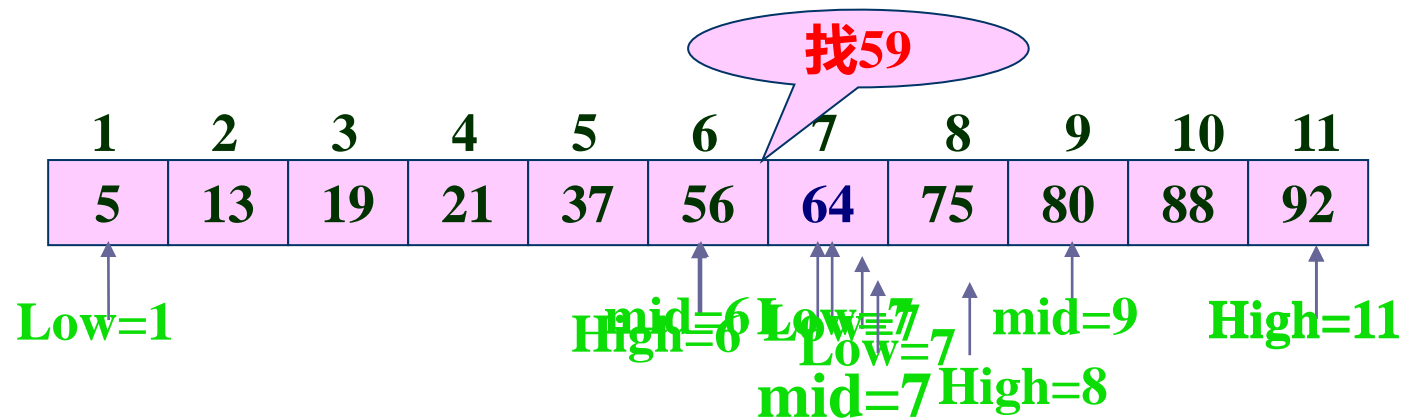
二、折半查找(举例一成功)

找64



第二节 静态查找表

二、折半查找(举例一不成功)



- 当下界low大于上界high时，说明有序表中没有关键字等于Key的元素，查找不成功

第二节 静态查找表

二、折半查找(算法实现)

定义折半查找类所有用到的函数及变量：

```
class BinSeach
{
    int length;
    int *elem;
public:
    BinSeach(); //分配空间, length赋值
    ~BinSeach(); //析构函数
    void InitBinSeach(); //输入数据, 0空间不用
    int BinSearchkey(int key); //二分查找
};
```

第二节 静态查找表

二、折半查找(算法实现)

```
int BinSearch::BinSearchKey(int Key)
{ int Low, Mid, High, Pos=0;
  Low = 1;                                // low指向待查元素所在区间的下界
  High = length;                          // high指向待查元素所在区间的上界
  while (Low <= High) {
    //将key与mid所指向的元素进行比较
    //相等跳出循环
    //不等修改low或high的值缩小比较范围
  }

  return(Pos);                            // 查找成功返回位置, 否则返回0
}
```


练习

试将折半查找的算法改写成递归算法。

```
int search_bin(int *r , int k , int low , int high)
{  if (low>high)
    return (0);          //不成功
  else
  {  mid=(low+high)/2;
    if (k==r[mid])
      return (mid) ; //成功
    else
      if (k<r[mid])
        search_bin(r,k,low,mid-1);
      else
        search_bin(r,k,mid+1,high);
  }
}
```

第二节 静态查找表

二、折半查找(判定树)

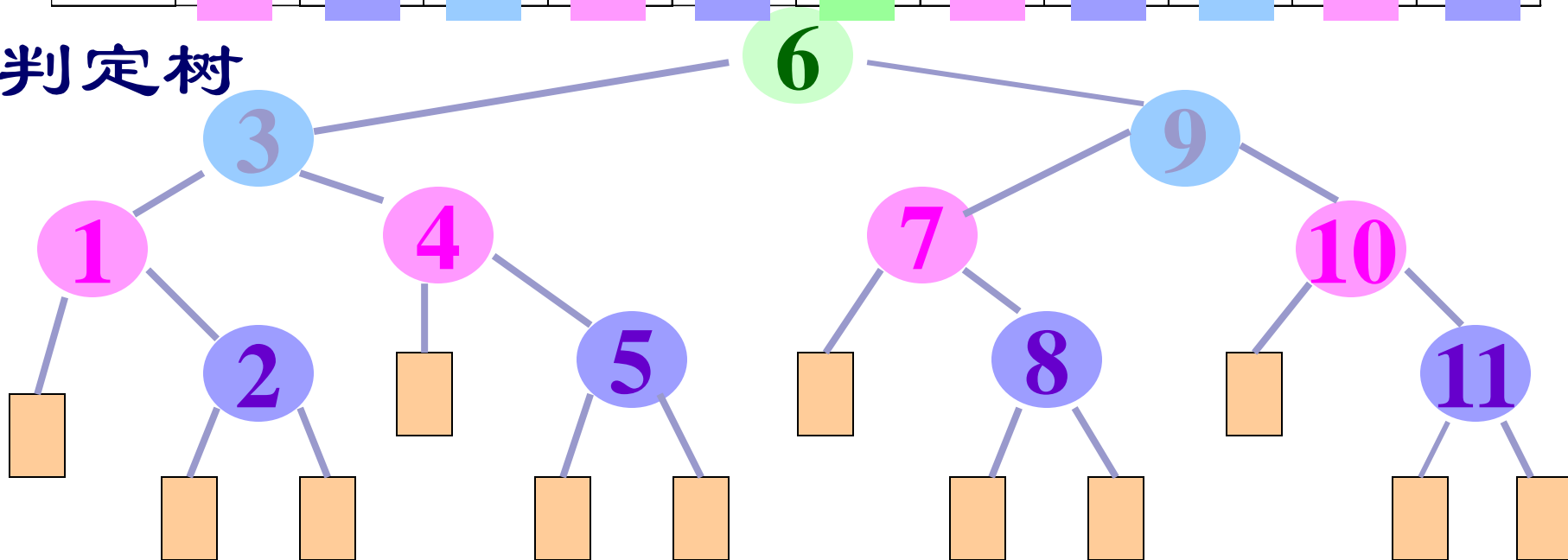
- 判定树：描述查找过程的二叉树。
- 有 n 个结点的判定树的深度为 $\lfloor \log_2 n \rfloor + 1$ （与含有 n 个结点的完全二叉树的深度相同）
- 折半查找法在查找过程中进行的比较次数最多不超过 $\lfloor \log_2 n \rfloor + 1$

分析折半查找的平均查找长度 $ASL_{成功}=3.4$

先看一个具体的情况，假设：n=11

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

判定树



练习

- 假设某有序表查找中有12个元素，请问该查找表查找成功时的平均查找长度为多少？

第二节 静态查找表

二、折半查找(性能分析)

- 设有序表的长度 $n=2^h-1$ （即 $h=\log_2(n+1)$ ），则描述折半查找的判定树是深度为 h 的满二叉树
- 树中层次为1的结点有1个，层次为2的结点有2个，层次为 h 的结点有 2^{h-1} 个
- 假设表中每个记录的查找概率相等，则查找成功时折半查找的平均查找长度

$$ASL_{bs} = \frac{1}{n} \sum_{i=1}^n C_i = \frac{1}{n} \left[\sum_{j=1}^h j \times 2^{j-1} \right] = \frac{n+1}{n} \log_2(n+1) - 1$$

第二节 静态查找表

二、折半查找(特点)

- 折半查找的效率比顺序查找高(特别是在静态查找表的长度很长时)
- 折半查找只能适用于有序表，并且以顺序存储结构存储

第二节 静态查找表

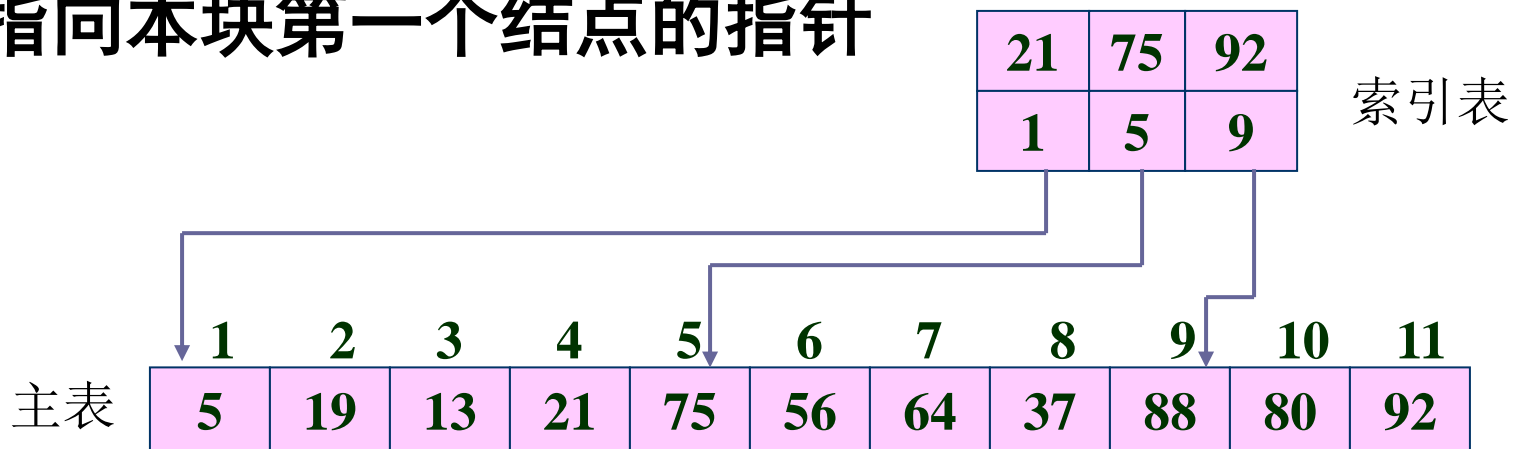
三、分块查找

- 分块查找是一种索引顺序表(分块有序表)查找方法，是折半查找和顺序查找的简单结合
- 索引顺序表(分块有序表)将整个表分成几块，块内无序，块间有序
- 所谓块间有序是指后一块表中所有记录的关键字均大于前一块表中的最大关键字

第二节 静态查找表

三、分块查找(分块有序表)

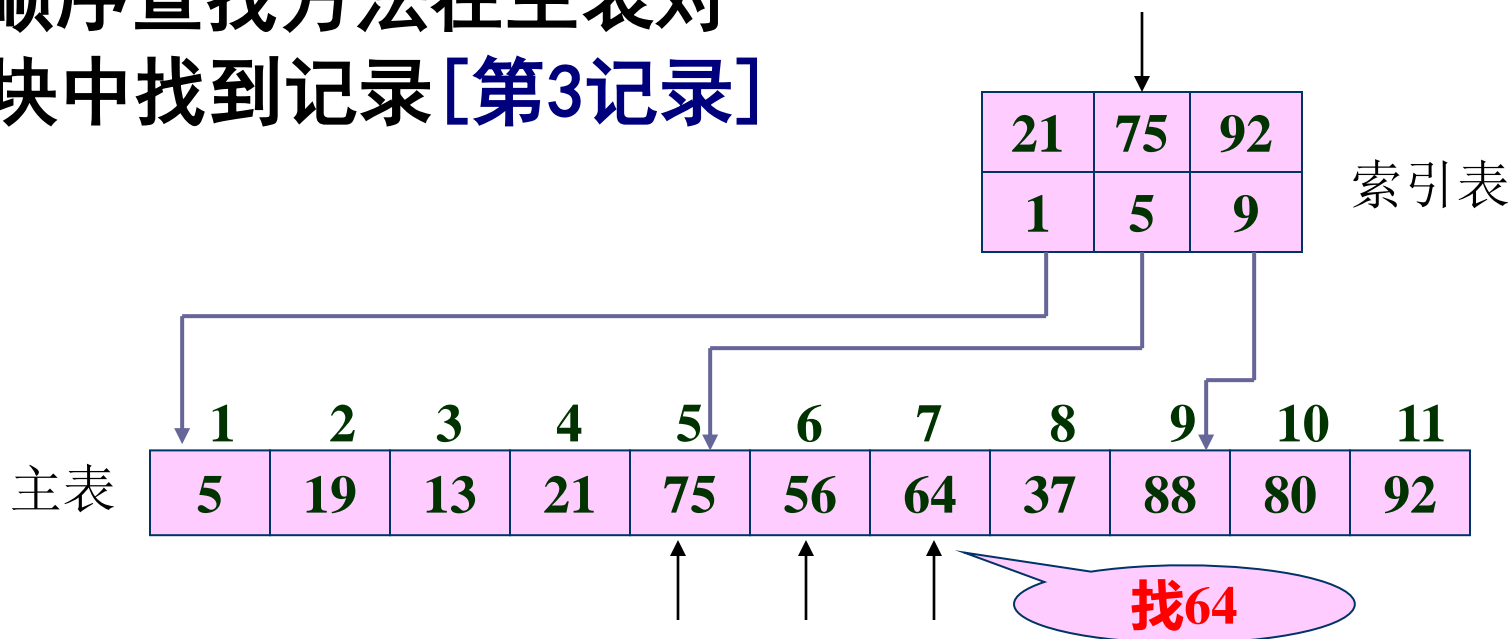
- **主表：**用数组存放待查记录, 每个数据元素至少含有关键字域
- **索引表：**每个结点含有最大关键字域和指向本块第一个结点的指针



第二节 静态查找表

三、分块查找(举例)

- 采用折半查找方法在索引表中找到块[第2块]
- 用顺序查找方法在主表对应块中找到记录[第3记录]



第二节 静态查找表

三、分块查找(性能分析)

- 若将长度为n的表分成b块，每块含s个记录，并设表中每个记录查找概率相等
- 用折半查找方法在索引表中查找索引块，

$$ASL_{\text{块间}} \approx \log_2(n/s+1)$$

- 用顺序查找方法在主表对应块中查找记录，

$$ASL_{\text{块内}} = s/2$$

- $ASL \approx \log_2(n/s+1) + s/2$



9.3 动态查找表

第三节 动态查找表

一、动态查找表

- 表结构本身是在查找过程中动态生成的
- 若表中存在其关键字等于给定值 key 的记录, 表明查找成功;
- 否则插入关键字等于 key 的记录。

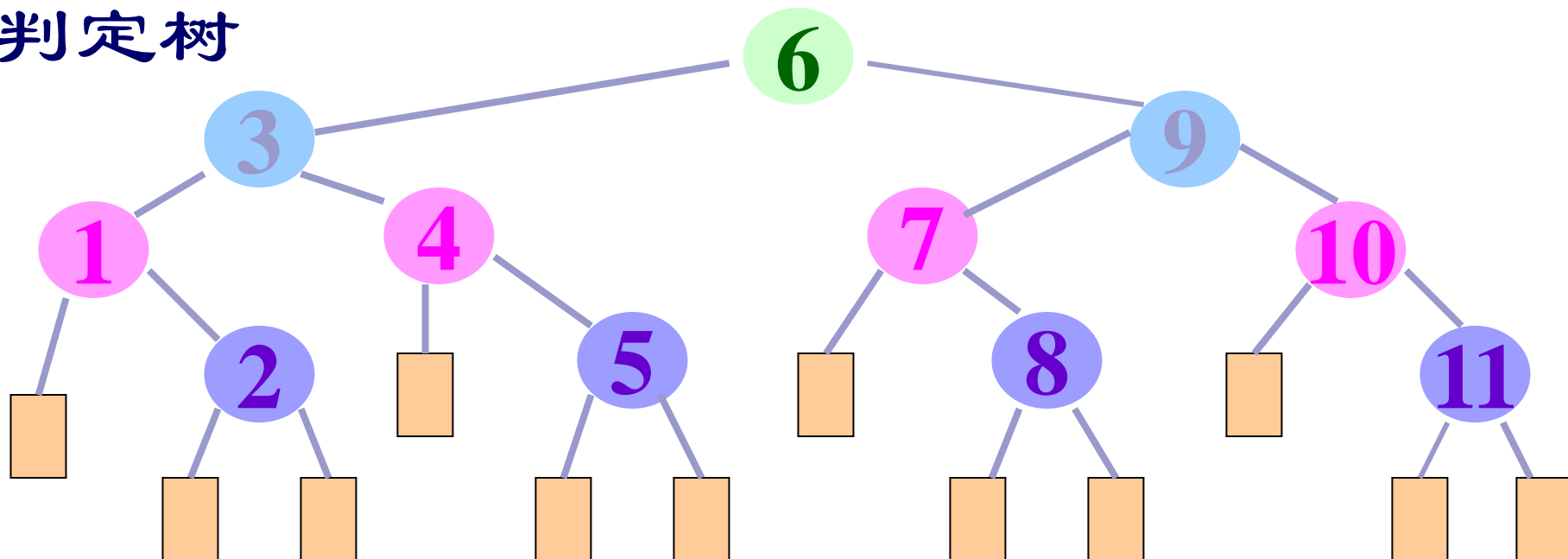
什么样的存储结构利于插入新的关键字呢?

试试把判定树的结点信息从下标改为数据?

先看一个具体的情况, 假设: $n=11$

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

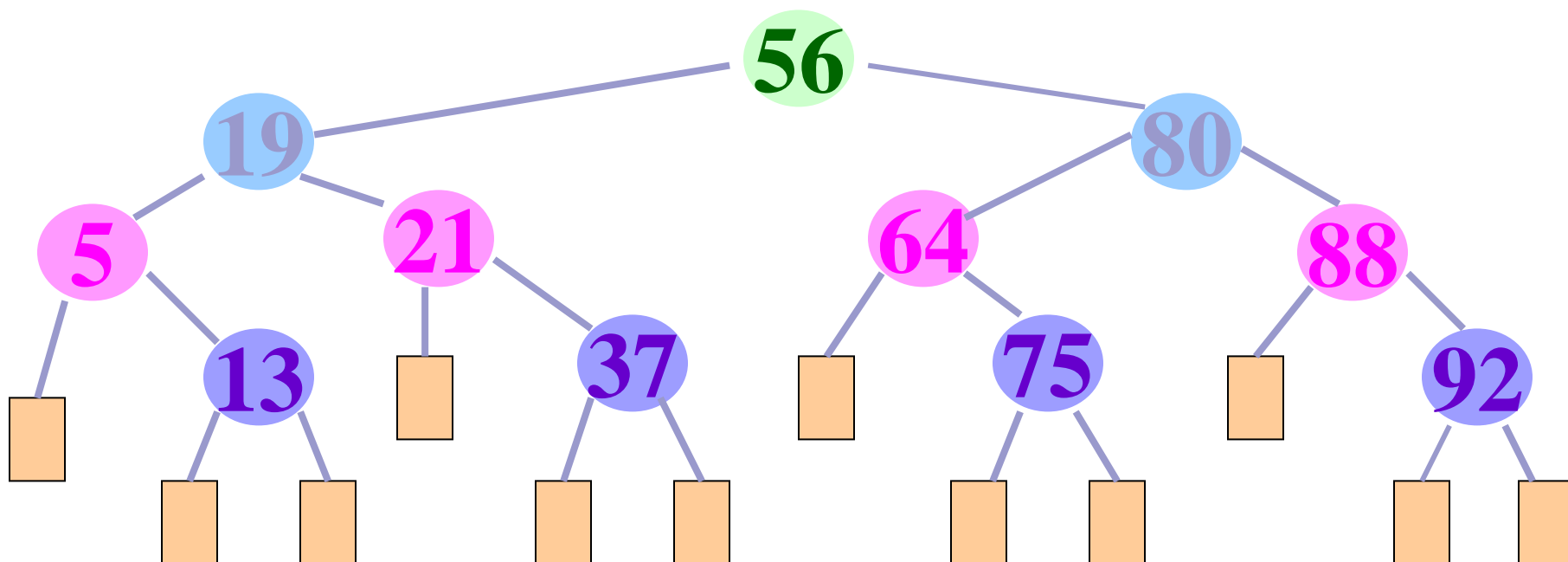
判定树



判定树 → 二叉排序树

结点信息从下标改为数据

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



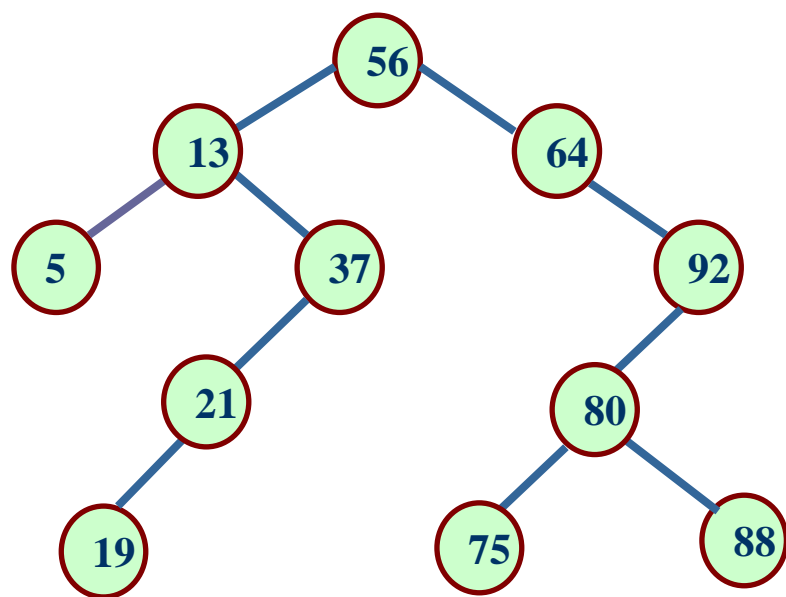
第三节 动态查找表

二、二叉排序树

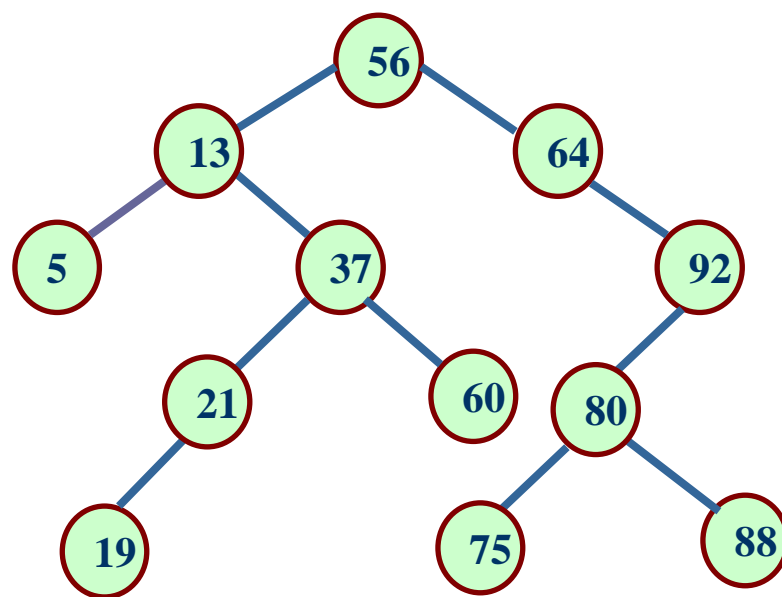
- 空树或者是具有如下特性的二叉树：
 - (1). 若它的左子树不空，则左子树上所有结点的值均小于根结点的值；
 - (2). 若它的右子树不空，则右子树上所有结点的值均大于根结点的值；
 - (3). 它的左、右子树也都分别是二叉排序树。

第三节 动态查找表

二、二叉排序树(举例)



二叉排序树



非二叉排序树

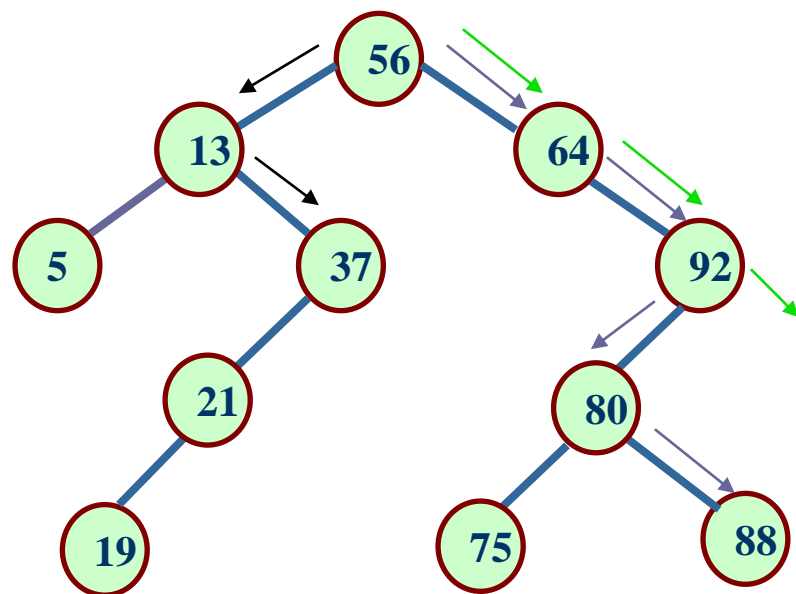
第三节 动态查找表

二、二叉排序树(查找)

- 二叉排序树又称二叉查找树
- 查找算法：

给定值与根结点比较：

1. 若相等，查找成功
2. 若小于，查找左子树
3. 若大于，查找右子树



在二叉排序树中查找关键字值等于37, 88, 94

第三节 动态查找表

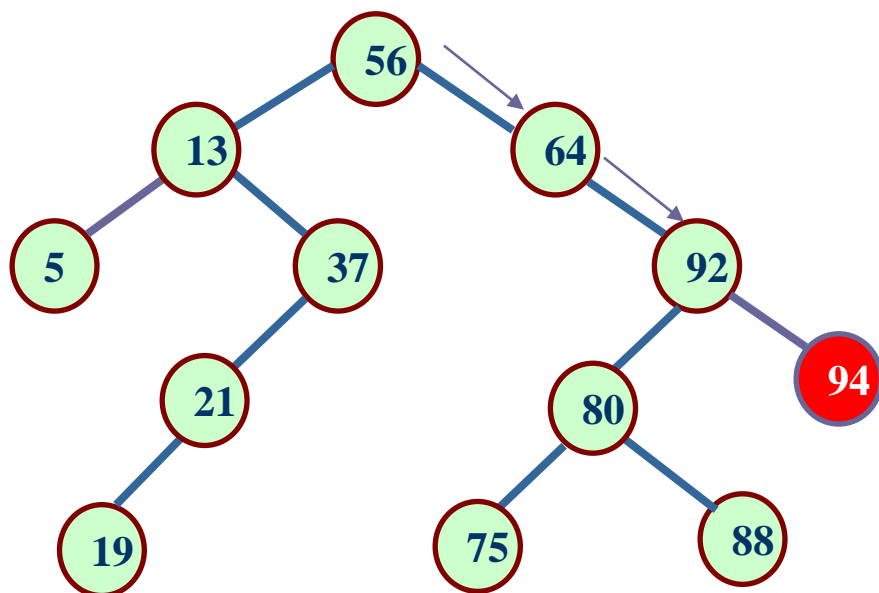
二、二叉排序树(插入)

- 二叉排序树是一种动态树表
- 当树中不存在查找的结点时，作插入操作
- 新插入的结点一定是叶子结点（只需改动一个结点的指针）
- 该叶子结点是查找不成功时路径上访问的最后一个结点左孩子或右孩子（新结点值小于或大于该结点值）

第三节 动态查找表

二、二叉排序树(插入举例)

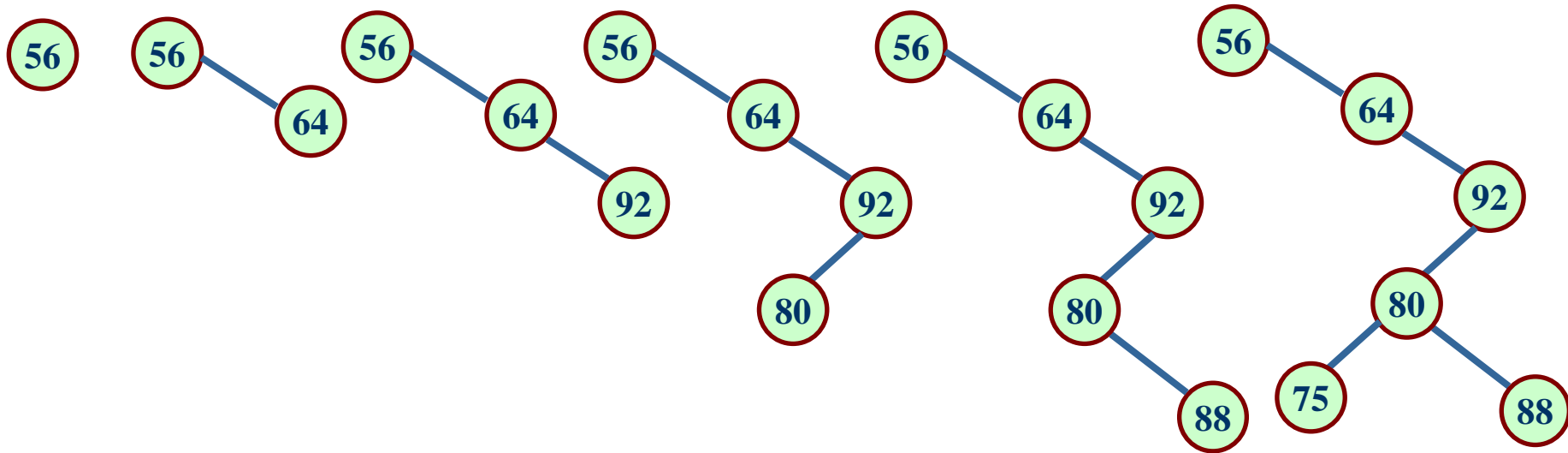
■ 插入结点94



第三节 动态查找表

二、二叉排序树(生成举例)

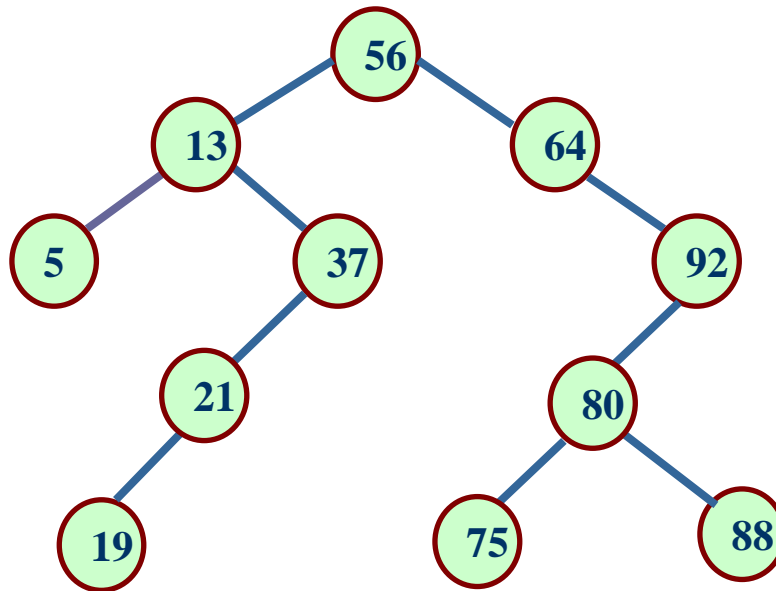
- 画出在初始为空的二叉排序树中依次插入 56, 64, 92, 80, 88, 75 时该树的生长全过程



第三节 动态查找表

二、二叉排序树(中序遍历)

- 中序遍历二叉排序树，可得到一个关键字的有序序列
- 如：5, 13, 19, 21, 37, 56, 64, 75, 80, 88, 92



第三节 动态查找表

二、二叉排序树(删除)

- 删除二叉排序树中的一个结点后，必须保持二叉排序树的特性（左子树的所有结点值小于根结点，右子树的所有结点值大于根结点）
- 也即保持中序遍历后，输出为有序序列

第三节 动态查找表

二、二叉排序树(删除)

- 被删除结点具有以下三种情况：
 1. 是叶子结点
 2. 只有左子树或右子树
 3. 同时有左、右子树

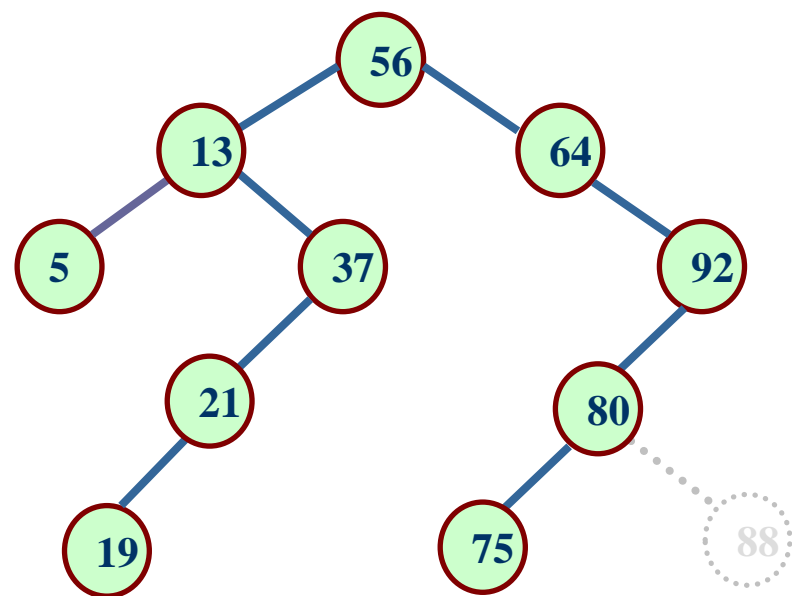
第三节 动态查找表

二、二叉排序树(删除)

1. 被删除结点是叶子结点

- 直接删除结点，并让其父结点指向该结点的指针变为空

删除结点88

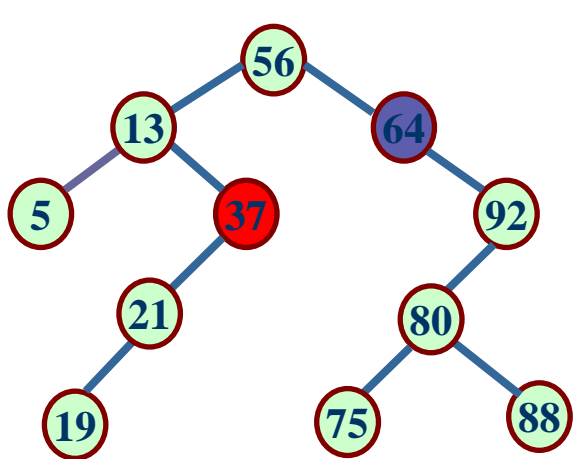


第三节 动态查找表

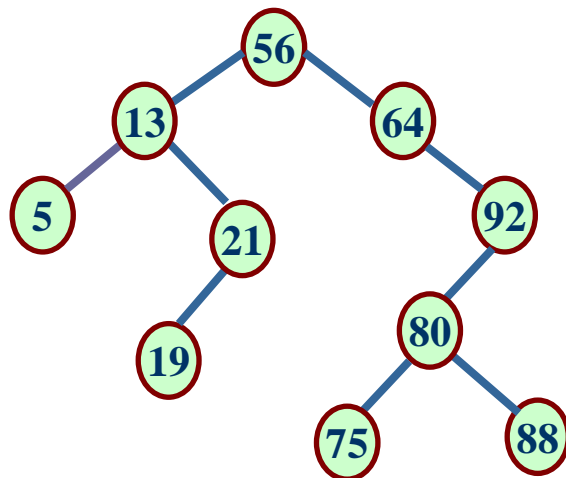
二、二叉排序树(删除)

2. 被删除结点只有左子树或右子树

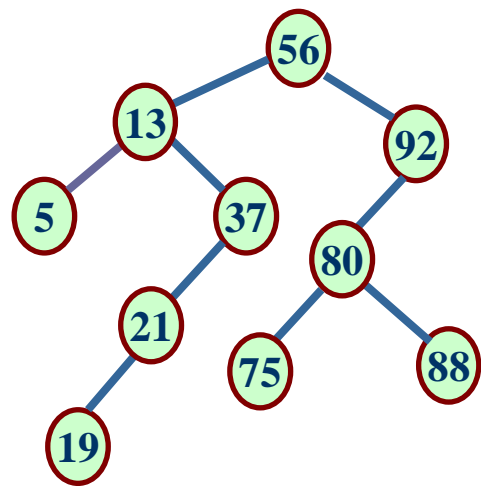
- 删除结点, 让其父结点指向该结点的指针指向其左子树(或右子树), 即用孩子结点替代被删除结点即可



原图



删除结点37(只有左子树)

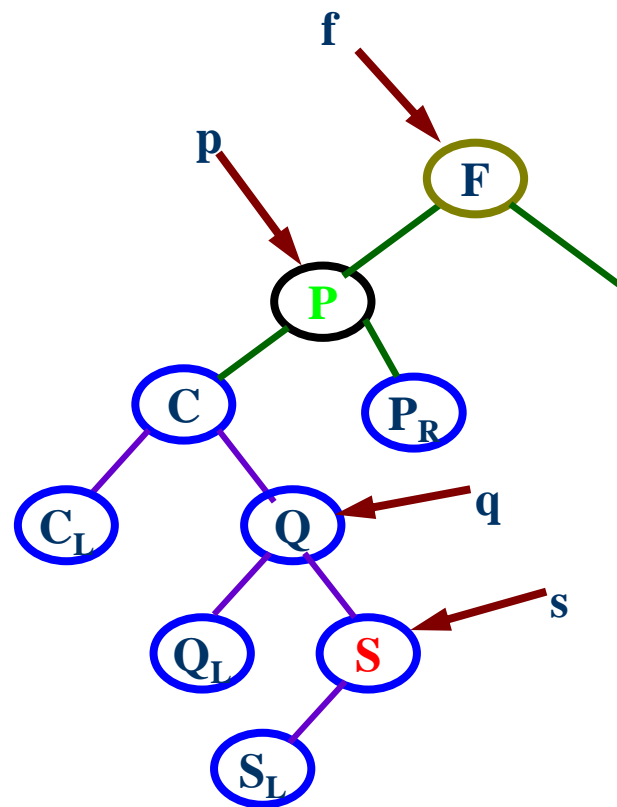


删除结点64(只有右子树)

第三节 动态查找表

二、二叉排序树(删除)

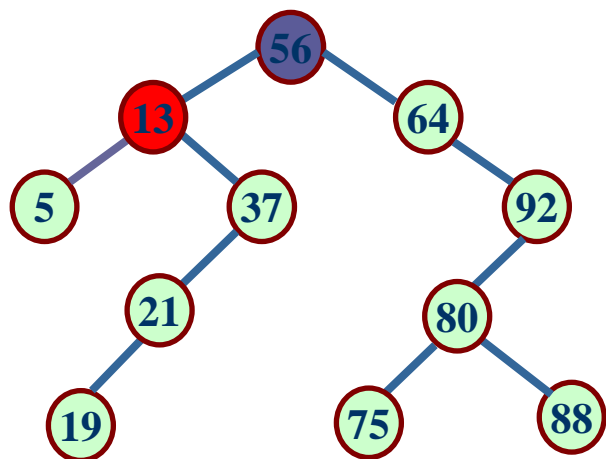
3. 被删除结点 p 既有左子树，又有右子树
- 以中序遍历时的直接前驱 s 替代被删除结点 p ，然后再删除该直接前驱（只可能有左孩子）



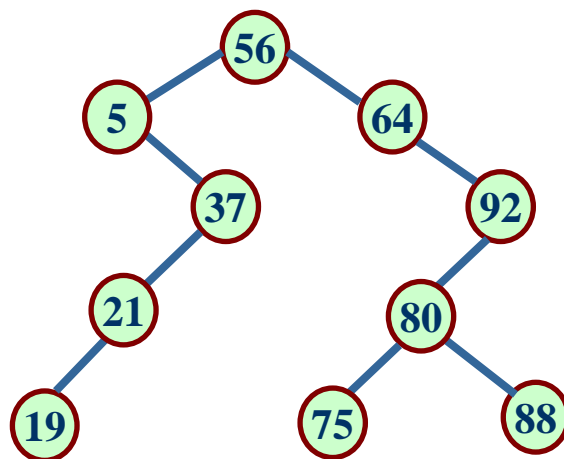
第三节 动态查找表

二、二叉排序树(删除)

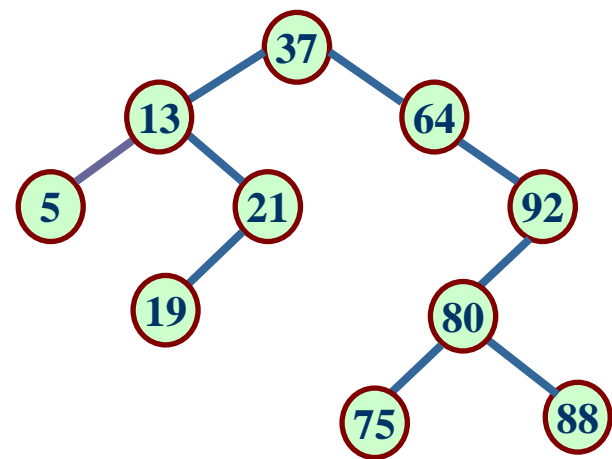
3. 被删除结点既有左子树，又有右子树(举例)



原图



删除结点13



删除结点56

第三节 动态查找表

二、二叉排序树(实现)

```
class BiTNode           //结点定义
{
    private:
        int          data;
        BiTNode      *lChild,*rChild;
    public:
        BiTNode(int e,BiTNode *l=NULL,BiTNode *r=NULL);
        ~BiTNode(){};
        friend class BSTree;
};
```



```
class BSTree
```

```
{
```

```
private:
```

```
    BiTNode    *root;
```

```
    int        count,pos;
```

```
    bool SearchNode(BiTNode *r, int key);
```

```
        //r树查找key,找不到, 返回false; 否则, true
```

```
    void InsertNode(BiTNode *&r, int key); //r树插入key
```

```
    void InShow(BiTNode *r); //中序遍历
```

```
public:
```

```
    BSTree():root(NULL){};
```

```
    ~BSTree();           //释放树结点
```

```
    bool Search(int key); //调用SearchNode查找
```

```
    bool Insert(int key); //查找不成功, 调用InsertNode插入
```

```
    void Show(); //调用InShowBST中序遍历BST
```

```
};
```

第三节 动态查找表

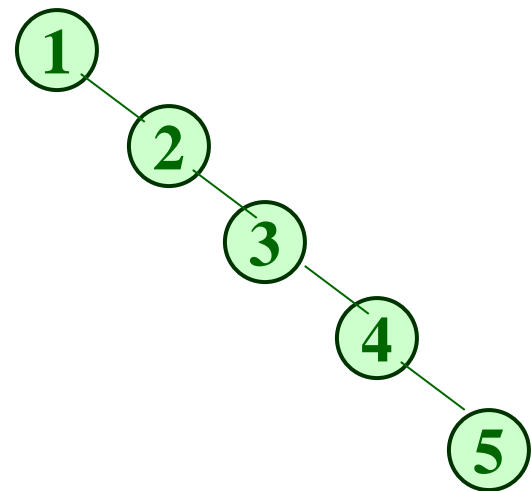
二、二叉排序树(性能分析)

- 对于每一棵特定的二叉排序树，均可按照平均查找长度的定义来求它的 **ASL** 值
- 显然，由值相同的 **n** 个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长度的值不同，甚至可能差别很大。

例如：

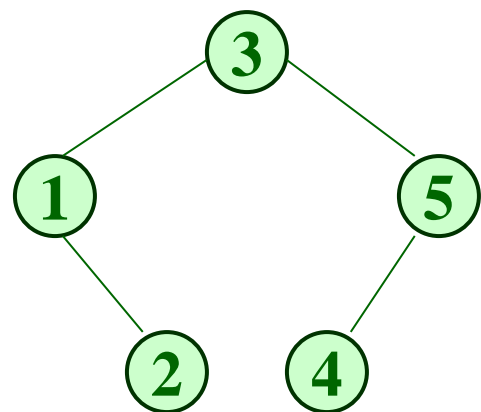
由关键字序列 **1, 2, 3, 4, 5**
构造而得的二叉排序树，

$$\begin{aligned} ASL &= (1+2+3+4+5) / 5 \\ &= 3 \end{aligned}$$



由关键字序列 **3, 1, 2, 5, 4**
构造而得的二叉排序树，

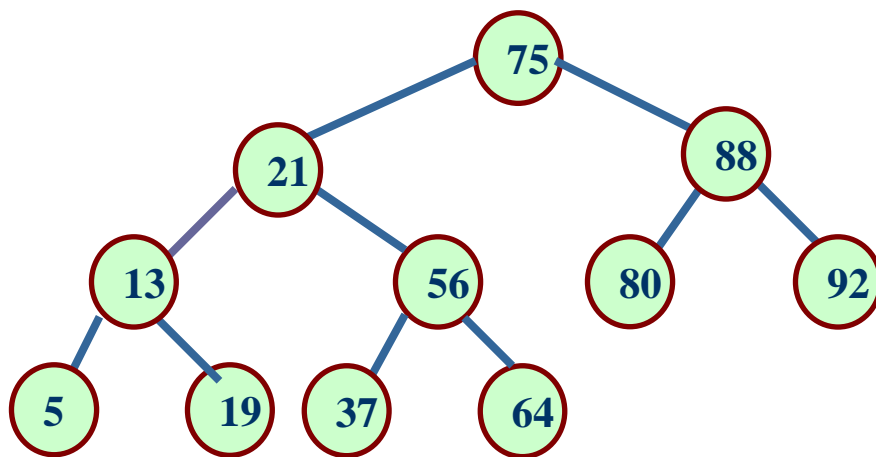
$$\begin{aligned} ASL &= (1+2+3+2+3) / 5 \\ &= 2.2 \end{aligned}$$



第三节 动态查找表

二、二叉排序树(性能分析)

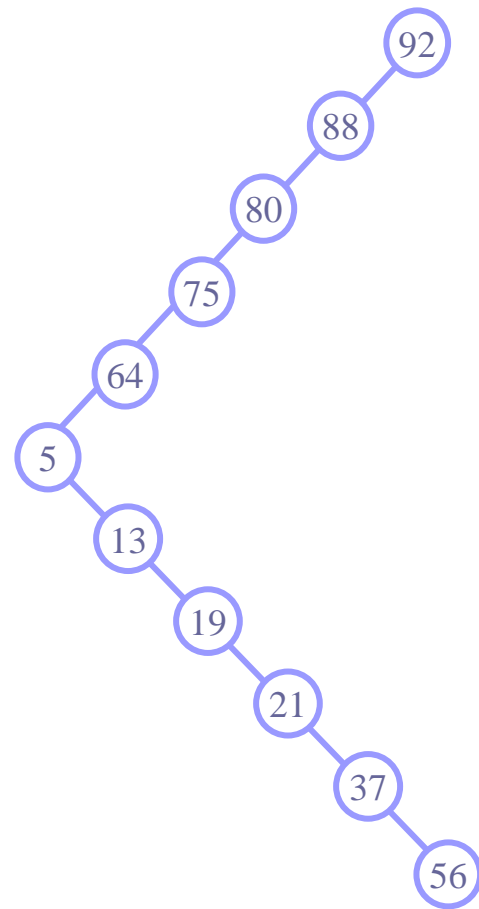
- 在最好的情况下，二叉排序树为一近似完全二叉树时，其查找深度为 $\log_2 n$ 量级，即其时间复杂性为 $O(\log_2 n)$



第三节 动态查找表

二、二叉排序树(性能分析)

- 在最坏的情况下，二叉排序树为近似线性表时(如以升序或降序输入结点时)，其查找深度为 n 量级，即其时间复杂度为 $O(n)$



第三节 动态查找表

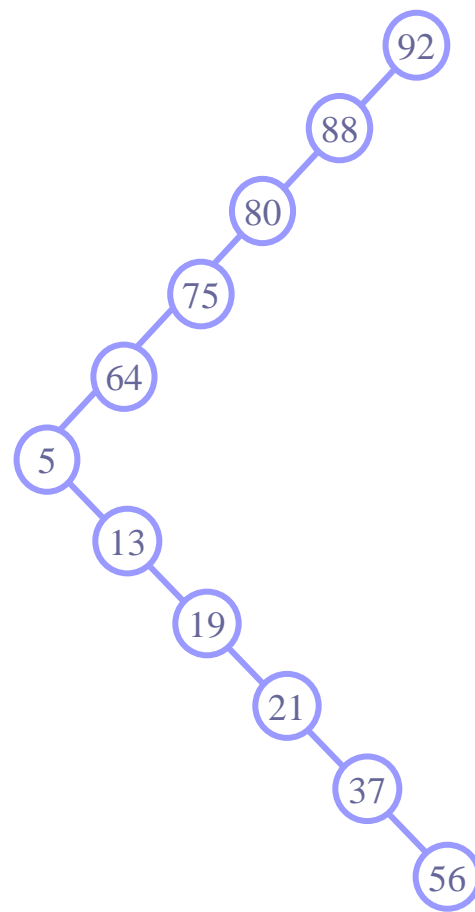
二、二叉排序树(特性)

- 一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列（通过中序遍历）
- 插入新记录时，只需改变一个结点的指针，相当于在有序序列中插入一个记录而不需要移动其它记录
- 二叉排序树既拥有类似于折半查找的特性，又采用了链表作存储结构

第三节 动态查找表

二、二叉排序树(特性)

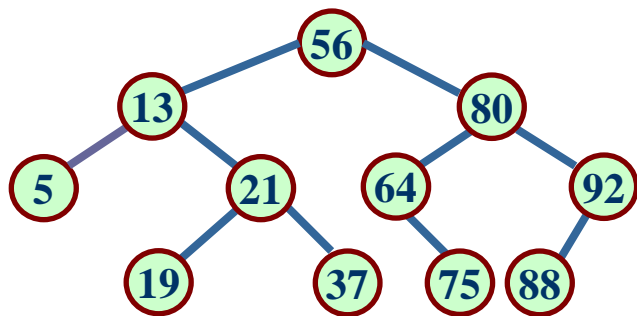
- 但当插入记录的次序不当时(如升序或降序)，则二叉排序树深度很深(11)，增加了查找的时间



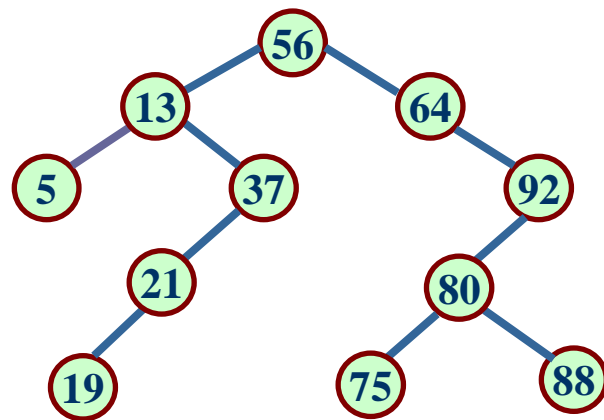
第三节 动态查找表

三、平衡二叉树[AVL] (定义)

- 平衡二叉树是二叉排序(查找)树的另一种形式
- 平衡二叉树又称AVL树 (Adel'sen-Velskii and Landis)
- 其特点为：树中每个结点的左、右子树深度之差的绝对值**不大于1**，即 $|h_L - h_R| \leq 1$



AVL树

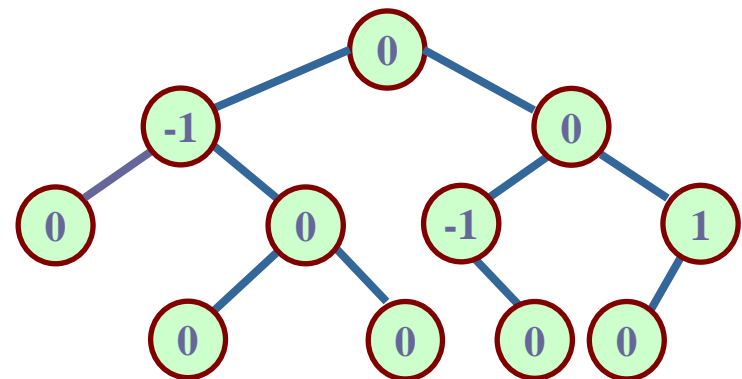
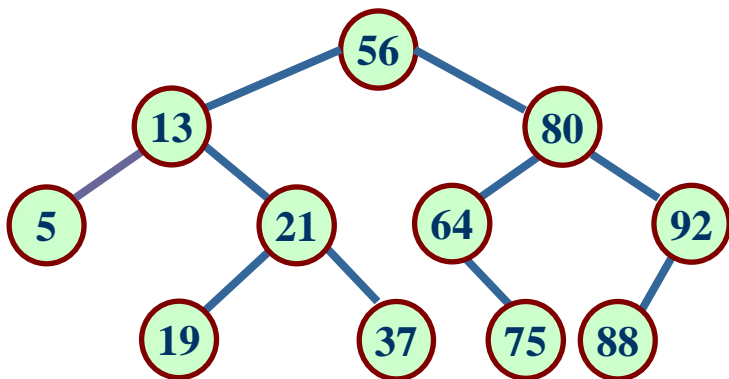


非AVL树

第三节 动态查找表

三、平衡二叉树[AVL] (平衡因子)

- 每个结点附加一个数字, 给出该结点左子树的高度减去右子树的高度所得的高度差, 这个数字即为结点的平衡因子balance
- AVL树任一结点平衡因子只能取 $-1, 0, 1$



第三节 动态查找表

三、平衡二叉树[AVL] (平衡化旋转)

- 如果在一棵平衡的二叉查找树中插入一个新结点，造成了不平衡。此时必须调整树的结构，使之平衡化。
- 平衡化旋转(处理)有两类：
 1. 单向旋转 (单向右旋和单向左旋)
 2. 双向旋转 (先左后右旋转和先右后左旋转)

第三节 动态查找表

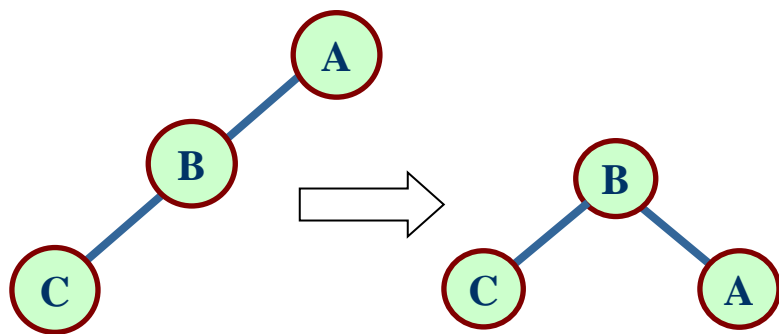
三、平衡二叉树[AVL] (平衡化旋转)

- 每插入一个新结点时，AVL树中相关结点的平衡状态会发生改变。
 1. 在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子。
 2. 如果在某一结点发现高度不平衡，停止回溯。
 3. 从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。

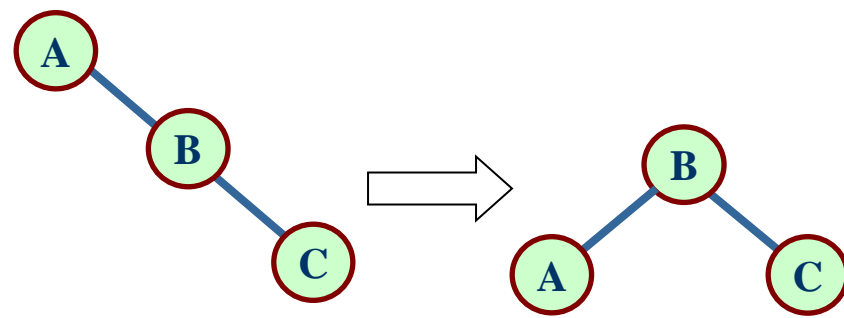
第三节 动态查找表

三、平衡二叉树[AVL] (平衡化单向旋转)

- 如果这三个结点处于一条直线上(“/” LL型或“\” RR型), 则采用单向旋转进行平衡化
- 单向旋转分为单向右旋(“/” 型)和单向左旋(“\” 型)



单向右旋
顺时针

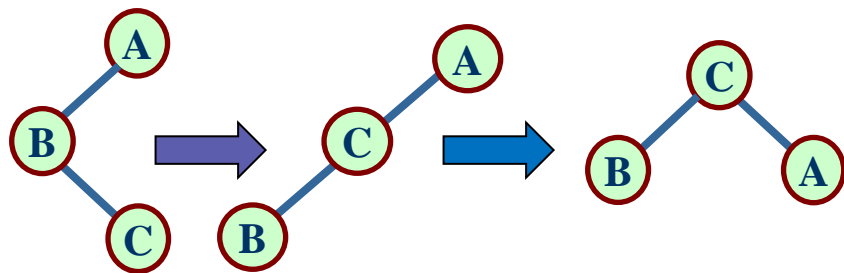


单向左旋
逆时针

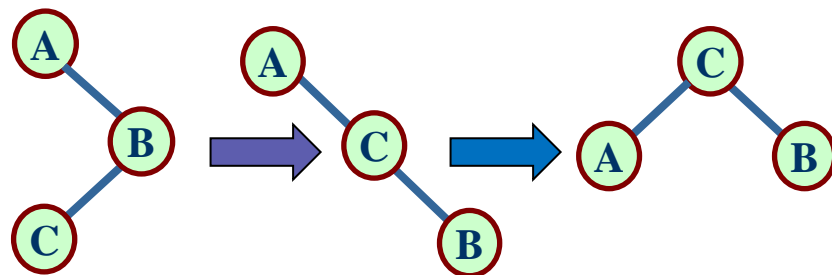
第三节 动态查找表

三、平衡二叉树[AVL] (平衡化双向旋转)

- 如果这三个结点处于一条折线上(“<” LR型或“>” RL型), 则采用双向旋转进行平衡化。
- 双旋转分为先左后右(“<”型)和先右后左(“>”型)



先左逆时针后
右顺时针旋转

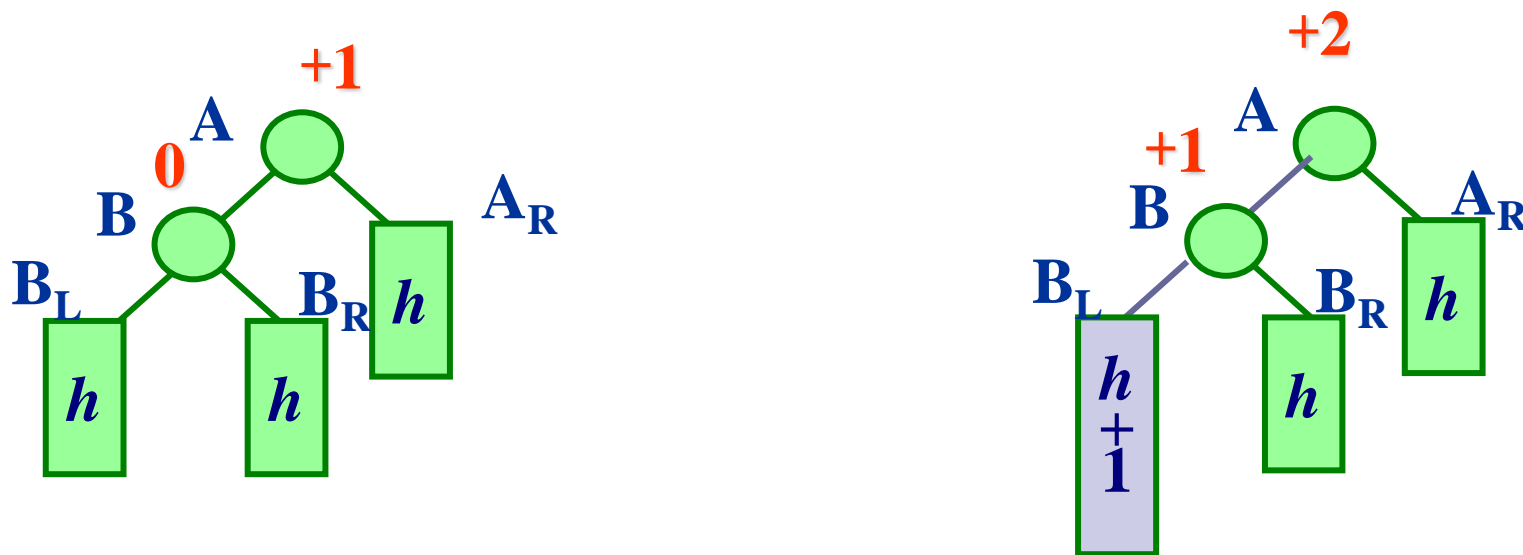


先右顺时针后
左逆时针旋转

第三节 动态查找表

三、平衡二叉树[AVL] (单向右旋)

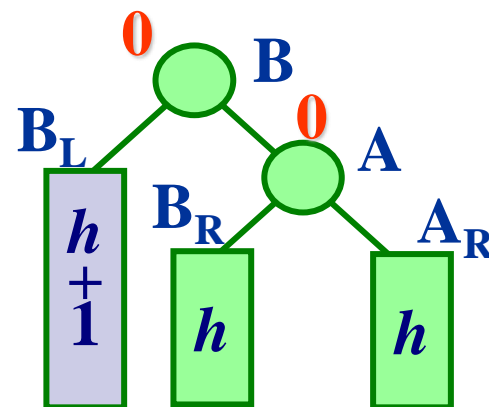
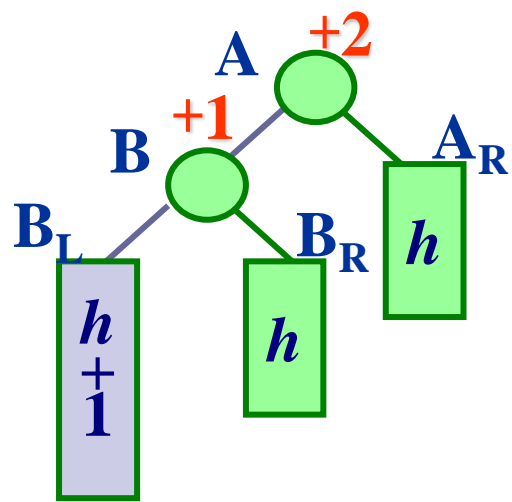
- 在B左子树 B_L 上插入新结点使其高度增1，导致结点A的平衡因子增到 +2，造成不平衡。



第三节 动态查找表

三、平衡二叉树[AVL] (单向右旋)

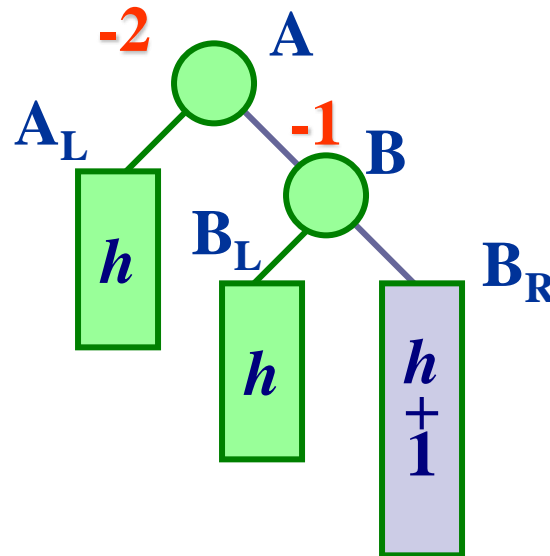
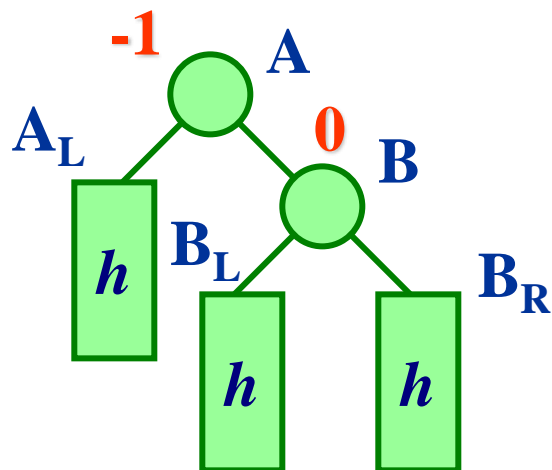
- 为使树恢复平衡，从A沿插入路径连续取3个结点A、B和 B_L (“/”型)
- 以结点B为旋转轴，将结点A顺时针(右)旋转。



第三节 动态查找表

三、平衡二叉树[AVL] (单向左旋)

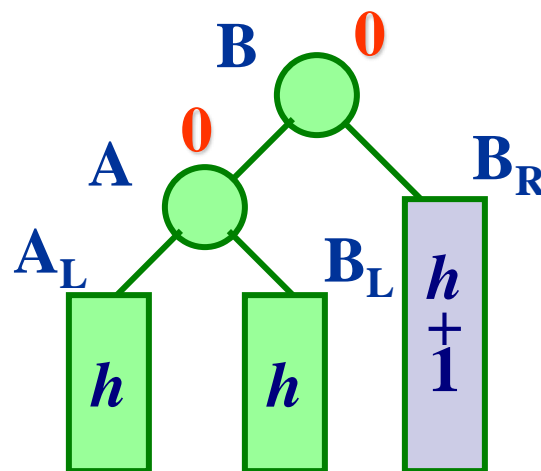
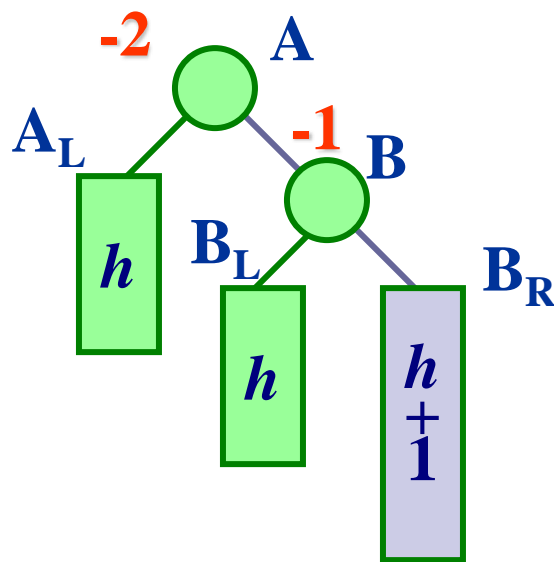
- 在B右子树 B_R 中插入新结点，该子树高度增1导致结点A的平衡因子变成-2，出现不平衡。



第三节 动态查找表

三、平衡二叉树[AVL] (单向左旋)

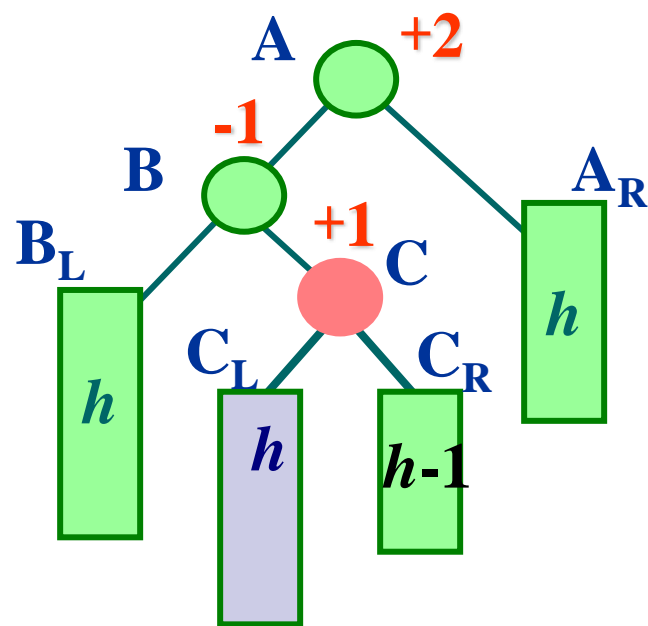
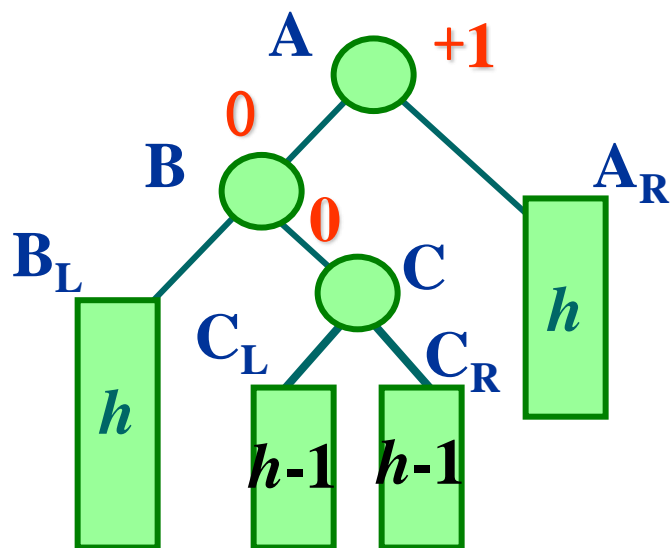
- 沿插入路径检查三个结点A、B和 B_R (“\”型)
- 以结点B为旋转轴，让结点A反时针(左)旋转



第三节 动态查找表

三、平衡二叉树[AVL] (先左后右双向旋转)

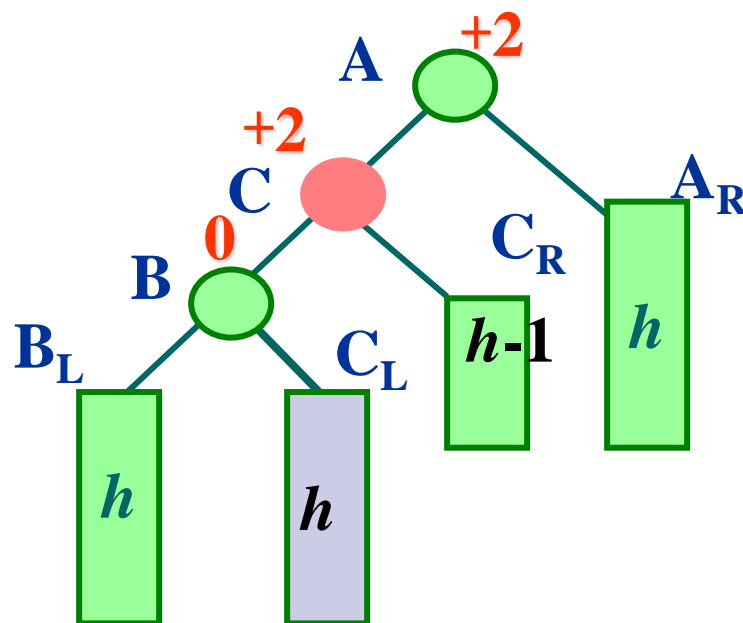
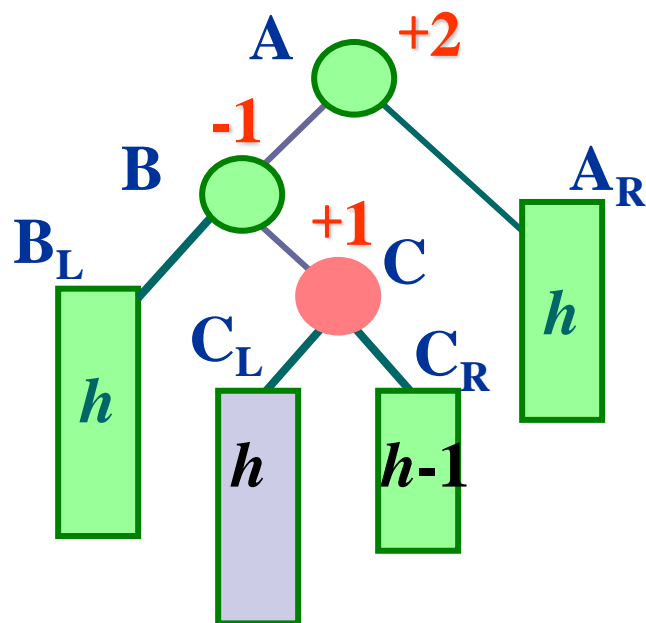
- 在C的子树 C_L 或 C_R 中插入新结点，该子树的高度增1。结点A的平衡因子变为+2，发生了不平衡



第三节 动态查找表

三、平衡二叉树[AVL] (先左后右双向旋转)

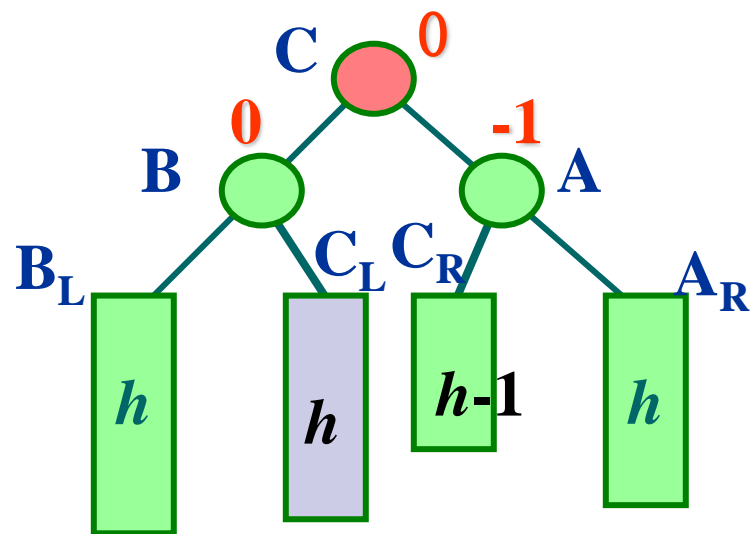
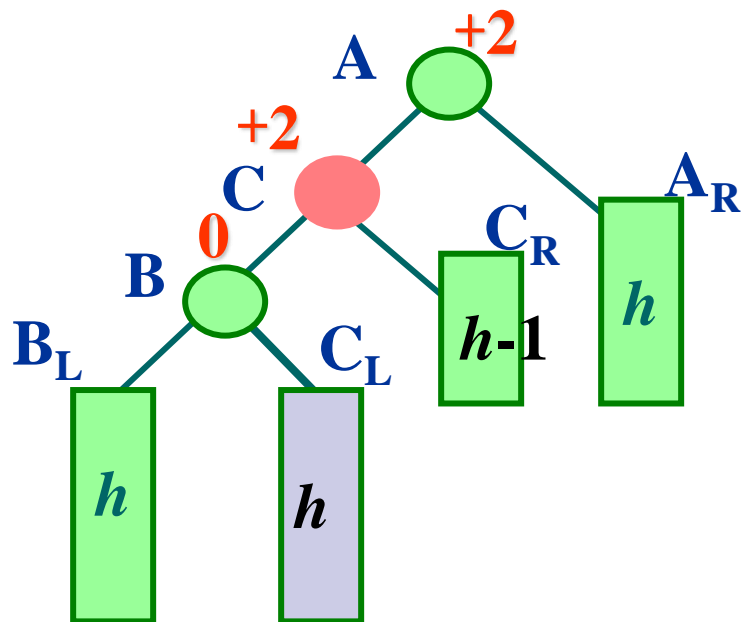
- 从结点A起沿插入路径选取3个结点A、B和C (“<”型)
- 以结点C为旋转轴，做单向左旋



第三节 动态查找表

三、平衡二叉树[AVL] (先左后右双向旋转)

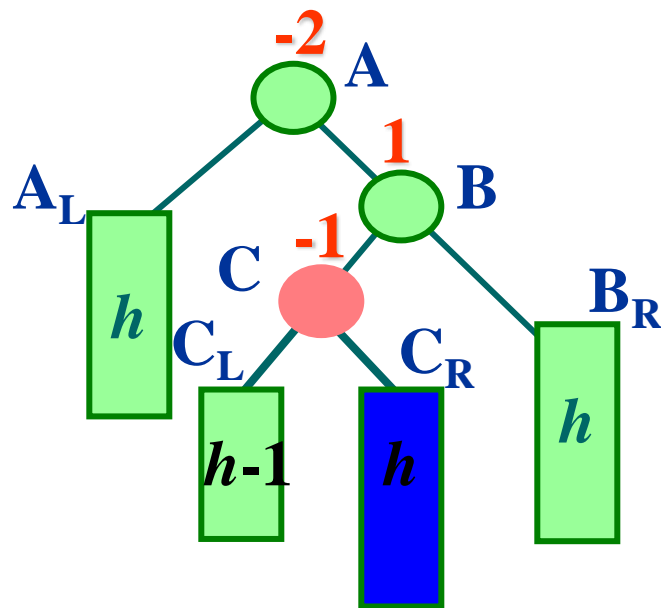
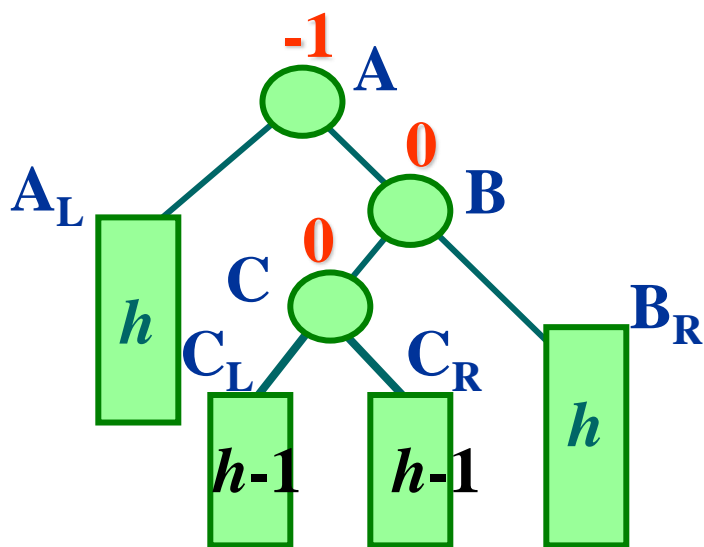
- 再以结点C为旋转轴，做单向右旋



第三节 动态查找表

三、平衡二叉树[AVL] (先右后左双向旋转)

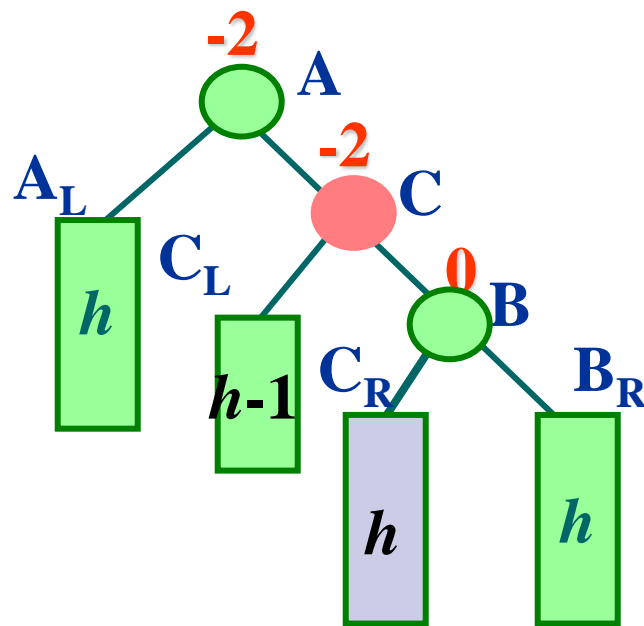
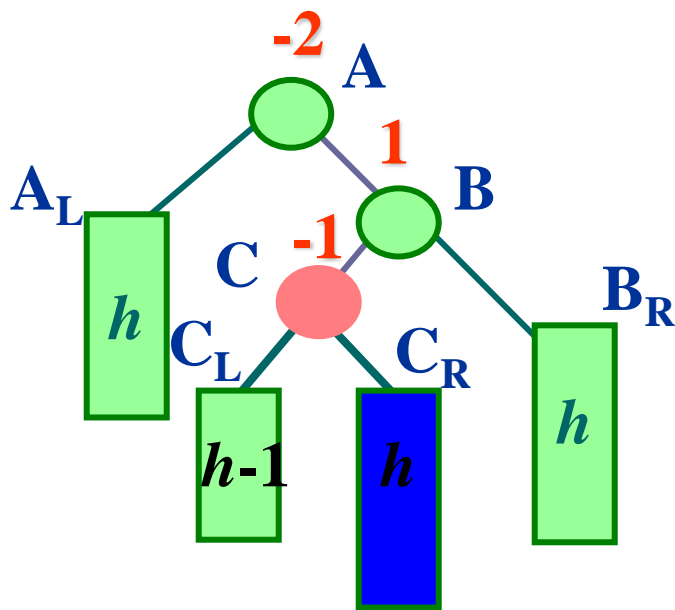
- 在C的子树 C_L 或 C_R 中插入新结点，该子树的高度增1。结点A的平衡因子变为-2，发生了不平衡



第三节 动态查找表

三、平衡二叉树[AVL] (先右后左双向旋转)

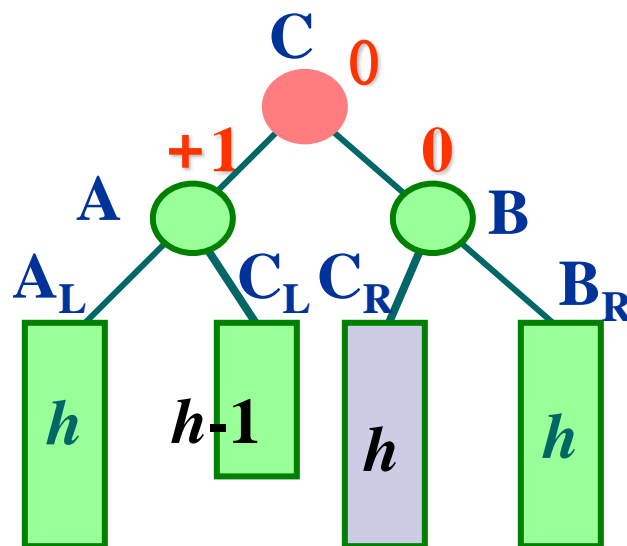
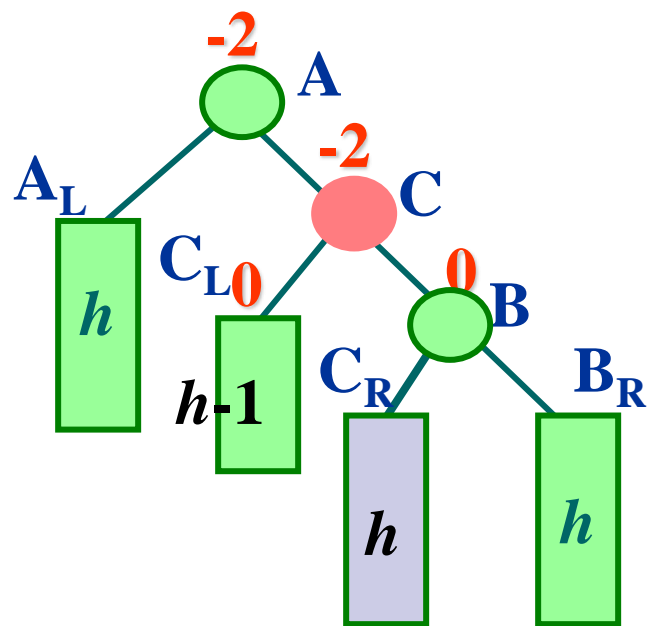
- 从结点A起沿插入路径选取3个结点A、B和C (“>”型)
- 以结点C为旋转轴，作单向右(顺)旋



第三节 动态查找表

三、平衡二叉树[AVL] (先右后左双向旋转)

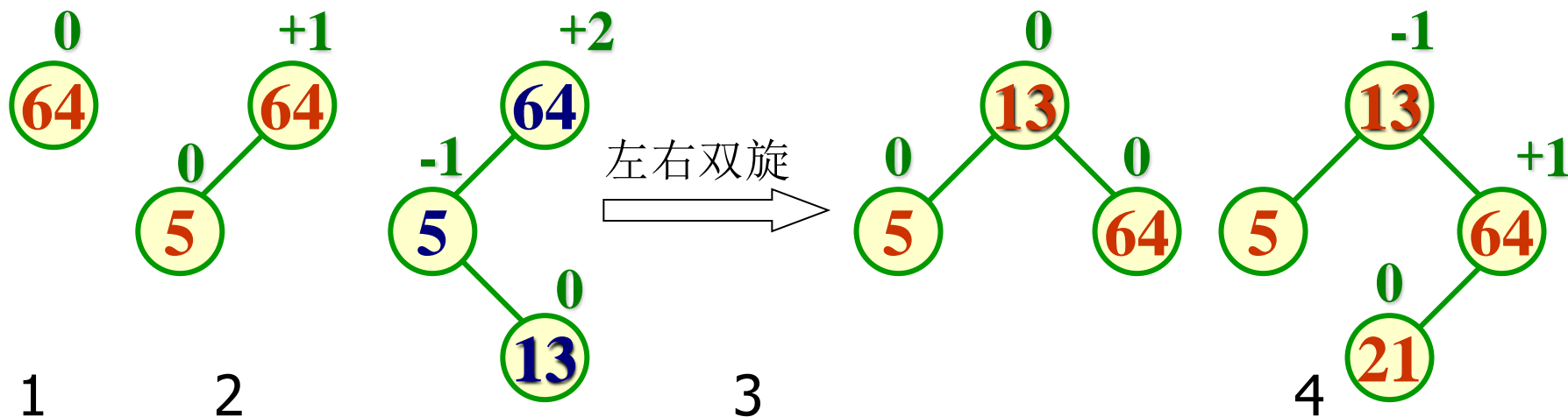
- 再以结点C为旋转轴，作单向左旋



第三节 动态查找表

三、平衡二叉树[AVL] (举例)

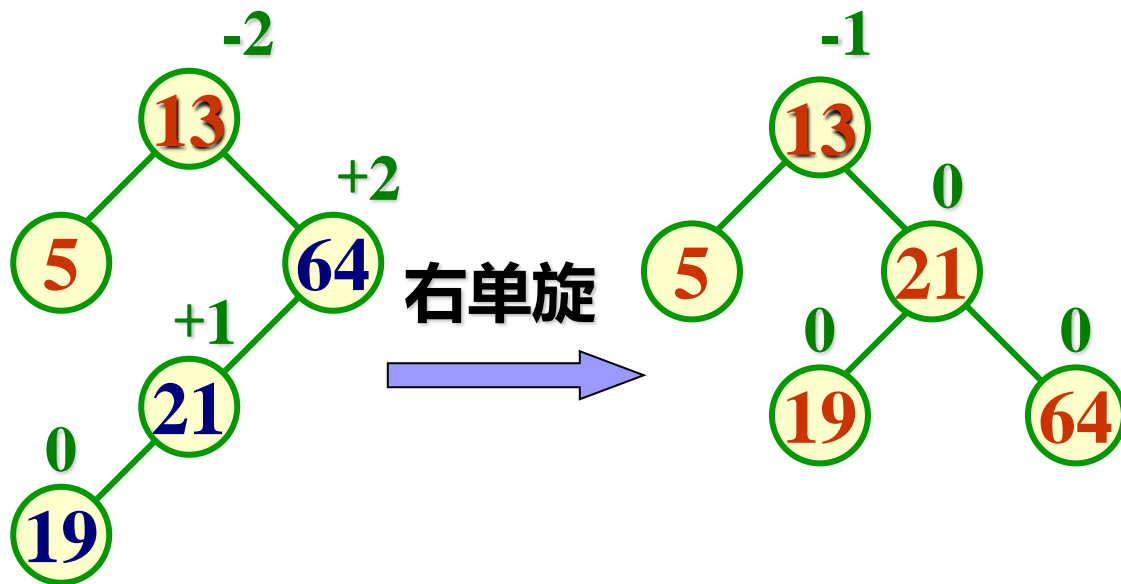
- 画出在初始为空的AVL树中依次插入64, 5, 13, 21, 19, 80, 75, 37, 56时该树的生长过程, 并在有旋转时说出旋转的类型。



第三节 动态查找表

三、平衡二叉树[AVL] (续例)

- 继续插入19, 80, 75, 37, 56时该树的生长过程, 并在有旋转时说出旋转的类型。

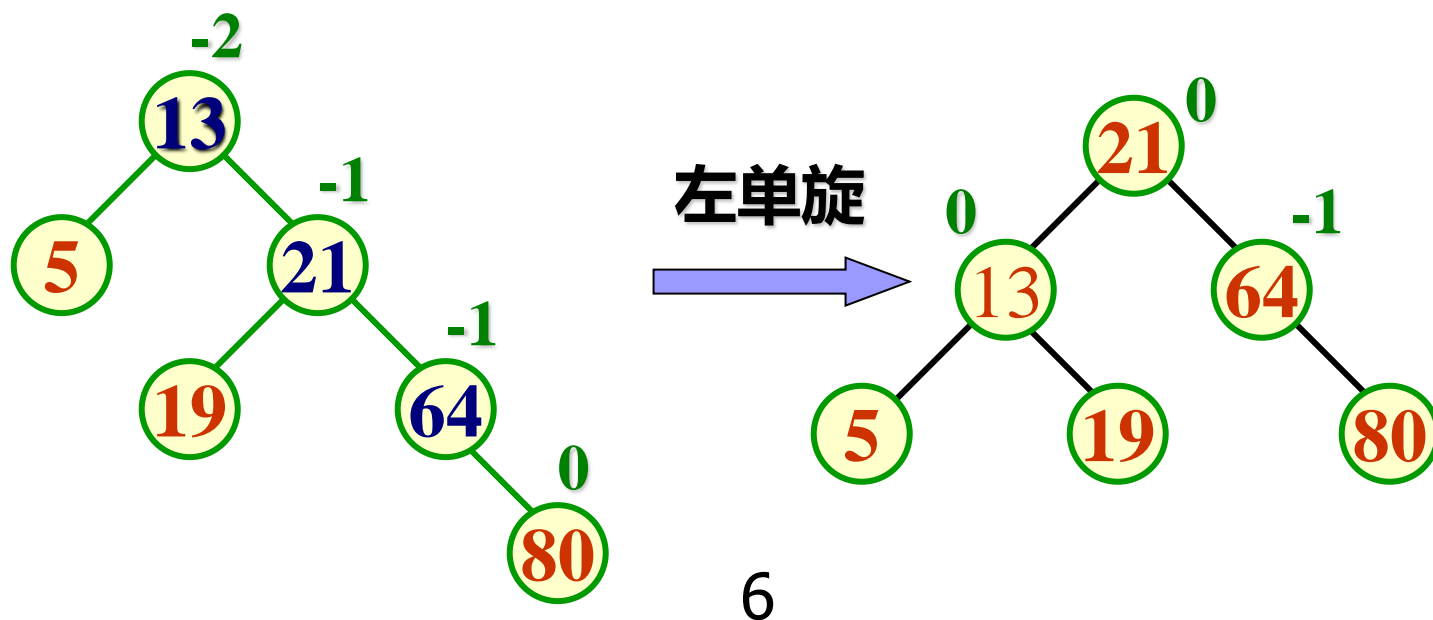


5

第三节 动态查找表

三、平衡二叉树[AVL] (续例)

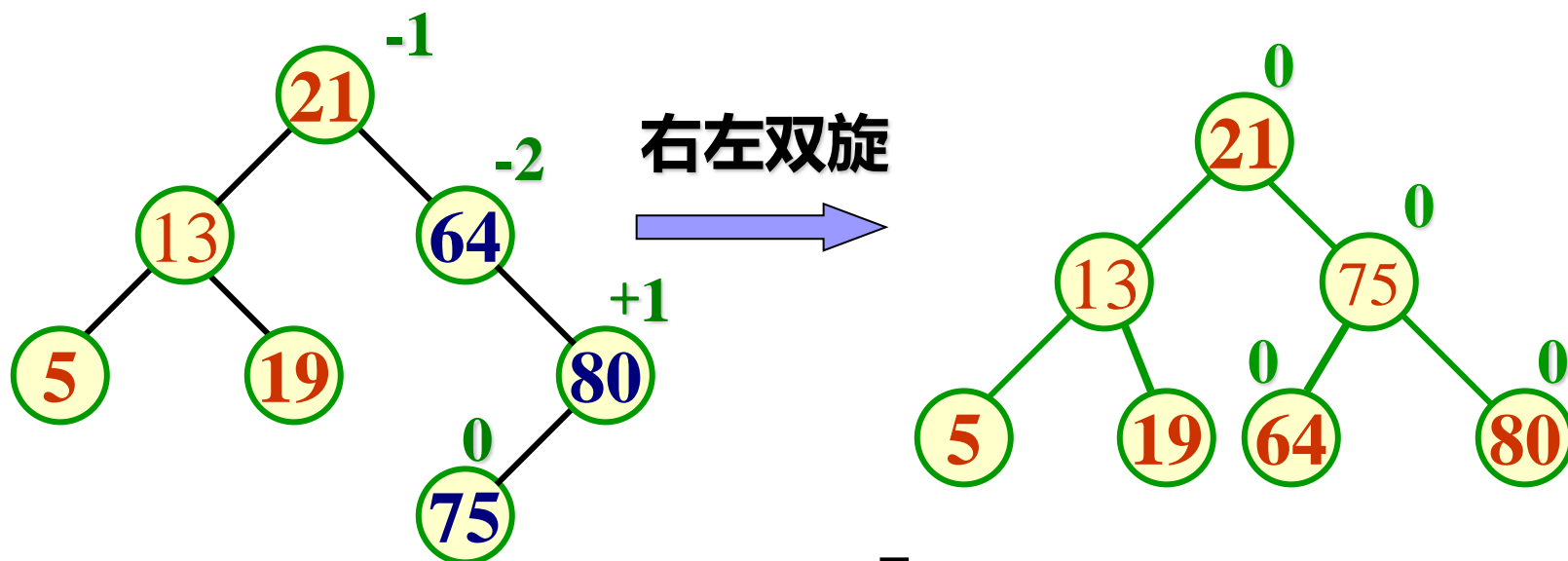
- 继续插入19, 80, 75, 37, 56时该树的生长过程, 并在有旋转时说出旋转的类型。



第三节 动态查找表

三、平衡二叉树[AVL] (续例)

- 继续插入19, 80, 75, 37, 56时该树的生长过程, 并在有旋转时说出旋转的类型。

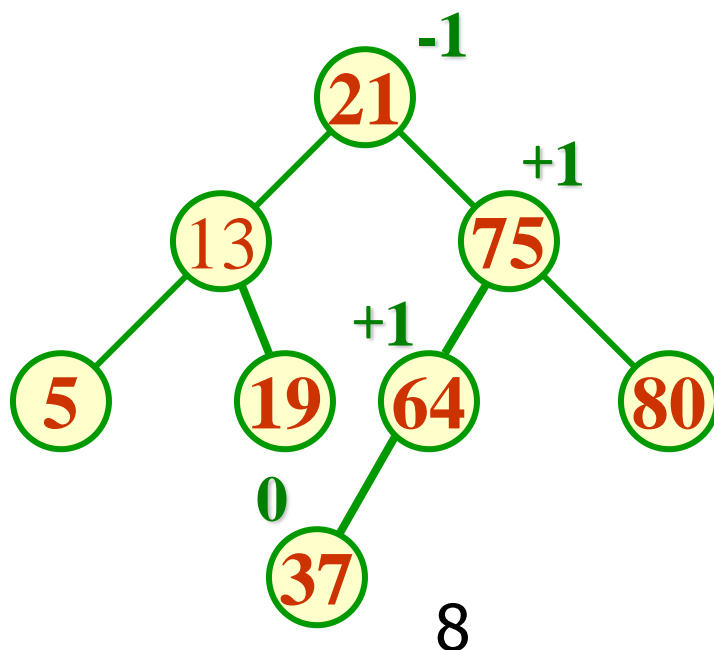


7

第三节 动态查找表

三、平衡二叉树[AVL] (续例)

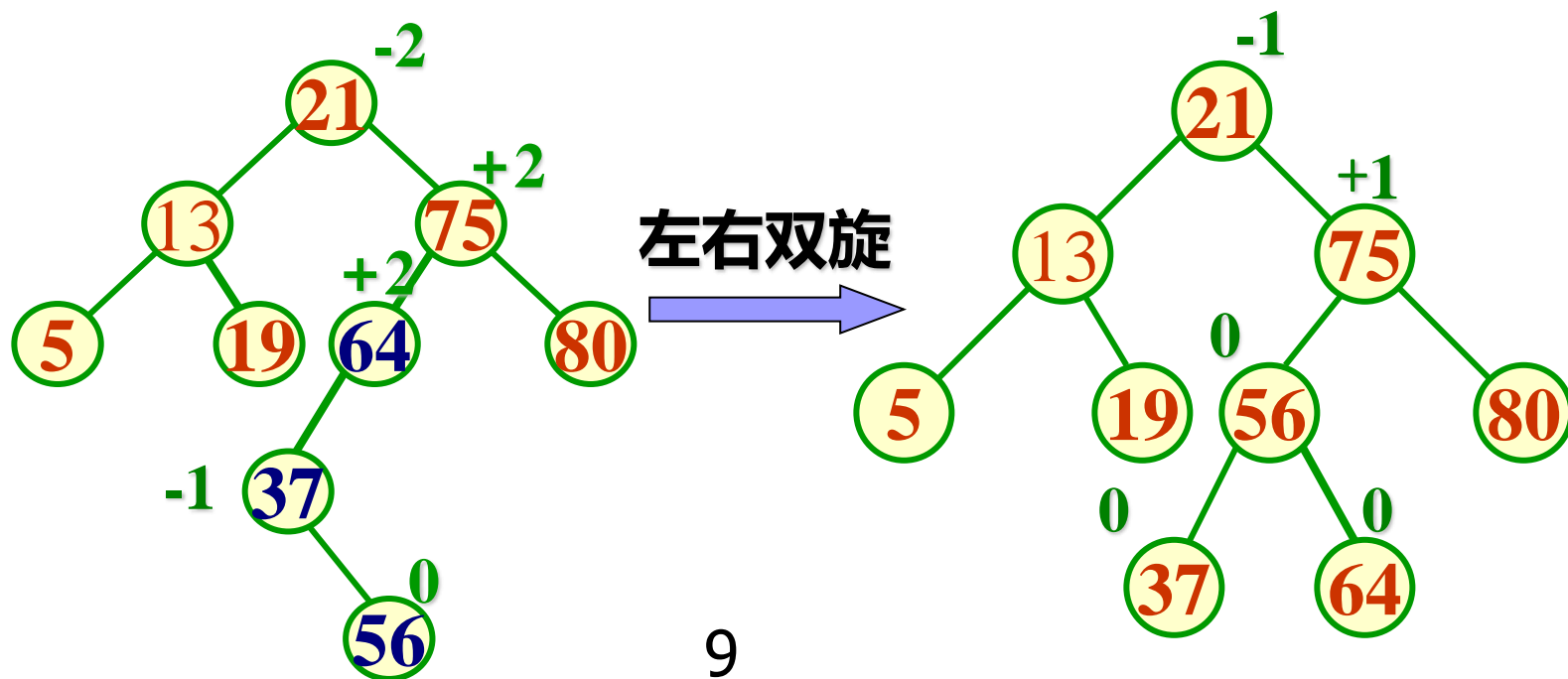
- 继续插入19, 80, 75, 37, 56时该树的生长过程, 并在有旋转时说出旋转的类型。



第三节 动态查找表

三、平衡二叉树[AVL] (续例)

- 继续插入19, 80, 75, 37, 56时该树的生长过程, 并在有旋转时说出旋转的类型。



第三节 动态查找表

三、平衡二叉树(删除)与二叉排序树相同

- 如果被删结点A最多只有一个孩子，那么将结点A从树中删去，并将其双亲指向它的指针指向它的唯一的孩子，并作平衡化处理
- 如果被删结点A没有孩子，则直接删除之，并作平衡化处理
- 如果被删结点A有两个子女，则用该结点的直接前驱S替代被删结点，然后对直接前驱S作删除处理(S只有一个孩子或没有孩子)



9.4 哈希表

第四节 哈希表

一、哈希表(散列表)

- 哈希 (Hash) 表又称散列表
- 散列表是一种直接计算记录存放地址的方法，它在关键码与存储位置之间直接建立了映象。

第四节 哈希表

一、哈希表(函数)

- 哈希函数是从关键字空间到存储地址空间的一种映象
- 哈希函数在记录的关键字与记录的存储地址之间建立起一种对应关系。可写成：

$$\text{addr}(a_i) = H(\text{key}_i)$$

- $H(\cdot)$ 为哈希函数
- key_i 是表中元素 a_i 关键字, $\text{addr}(a_i)$ 是存储地址

第四节 哈希表

一、哈希表(查找)

- 哈希查找也叫散列查找，是利用哈希函数进行查找的过程。
 1. 首先利用哈希函数及记录的关键字计算出记录的存储地址
 2. 然后直接到指定地址进行查找
 3. 不需要经过比较，一次存取就能得到所查元素

第四节 哈希表

一、哈希表(冲突)

- 不同的记录，其关键字通过哈希函数的计算，可能得到相同的地址
- 把不同的记录映射到同一个散列地址上，这种现象称为冲突

第四节 哈希表

一、哈希表(定义)

- 根据设定的**哈希函数** $H(key)$ 和所选中的处理冲突的方法
- 将一组关键字映射到一个有限的、**地址连续的**地址集(区间) 上
- 并以关键字在地址集中的“象”作为相应记录在表中的存储位置
- 如此构造所得的查找表称之为 **“哈希表”**

第四节 哈希表

二、哈希函数(均匀性)

- 哈希函数实现的一般是从一个**大的集合**（部分元素，空间位置上一般不连续）**到一个小的集合**（空间连续）的映射
- 一个**好的**哈希函数，对于记录中的任何关键字，将其映射到地址集合中任何一个地址的**概率应该是相等的**
- 即**关键字**经过哈希函数得到一个“**随机的地址**”

第四节 哈希表

二、哈希函数(要求)

- 哈希函数应是简单的，能在较短的时间内计算出结果。
- 哈希函数的定义域尽可能包括需要存储的全部关键字，如果散列表允许有 m 个地址时，其值域必须在 0 到 $m-1$ 之间。
- 散列函数计算出来的地址应能均匀分布在整个地址空间中

第四节 哈希表

二、哈希函数(直接定址法)

- 直接定址法中，哈希函数取关键字的线性函数

$$H(\text{key}) = a \times \text{key} + b$$

其中a和b为常数

第四节 哈希表

二、哈希函数(直接定址法—举例)

■ $H(\text{key}) = \text{key} - 2019131000$

06	2019131006	邓煦	男	信息工程学院
11	2019131011	张国明	男	信息工程学院
17	2019131017	刘金棠	男	信息工程学院
22	2019131022	陈俊东	男	信息工程学院
25	2019131025	邱益林	男	信息工程学院
31	2019131031	陈明亮	男	信息工程学院
32	2019131032	郭宁	男	信息工程学院

第四节 哈希表

二、哈希函数(直接定址法—特性)

- 直接定址法仅适合于地址集合的大小与关键字集合的大小相等的情况
- 当 $a=1$ 时, $H(key)=key$, 即用关键字作地址
- 在实际应用中能使用这种哈希函数的情况很少

第四节 哈希表

二、哈希函数(数字分析法)

- 假设关键字集合中的每个关键字都是由 s 位数字组成 (u_1, u_2, \dots, u_s)
- 分析关键字集中的全体
- 从中提取分布均匀的若干位或它们的组合作为地址。

第四节 哈希表

二、哈希函数(数字分析法一举例)

- 有80个记录，关键字为8位十进制数，哈希地址为2位十进制数

分析：①只取8
②只取1
③只取3、4
⑧只取2、7、5
④⑤⑥⑦数字分布近乎随机

所以：取④⑤⑥⑦任意两位或两位与另两位的叠加作哈希地址

①	②	③	④	⑤	⑥	⑦	⑧
			⋮				
8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5

第四节 哈希表

二、哈希函数(数字分析法—特性)

- 数字分析法仅适用于事先明确知道表中所有关键码每一位数值的分布情况
- 数字分析法完全依赖于关键码集合。
- 如果换一个关键码集合，选择哪几位要重新决定。

第四节 哈希表

二、哈希函数(平方取中法)

- 以关键字的平方值的中间几位作为存储地址。
- 求“关键字的平方值” 的目的是“扩大差别”
- 同时平方值的中间各位又能受到整个关键字中各位的影响。

第四节 哈希表

二、哈希函数(平方取中法一举例)

- 此方法在词典处理中使用十分广泛。
- 它先计算构成关键码的标识符的内码的平方，然后按照散列表的大小取中间的若干位作为散列地址。

第四节 哈希表

二、哈希函数(平方取中法一举例)

- 标识符的八进制内码表示及其平方值

标识符	内码	内码的平方	散列地址
ABC	010203	0104 <u>101</u> 209	101
BCD	020304	0412 <u>252</u> 416	252
CDE	030405	0924 <u>464</u> 025	464
DEF	040506	1640 <u>736</u> 036	736

第四节 哈希表

二、哈希函数(平方取中法一特性)

- 平方取中法是比较常用的构造哈希函数的方法
- 适合于关键字中的每一位都有某些数字重复出现且频率很高的情况
- 中间所取的位数，由哈希表长决定

第四节 哈希表

二、哈希函数(折叠法)

- 将关键字分割成位数相同的若干部分(最后部分的位数可以不同)，然后取它们的叠加和(舍去进位)为哈希地址。
- 移位叠加:将分割后的几部分低位对齐相加
- 间界叠加:从一端沿分割界来回折送，然后对齐相加

第四节 哈希表

二、哈希函数(折叠法一举例)

- 关键字为：0442205864，哈希地址位数为4

$$\begin{array}{r} 5\ 8\ 6\ 4 \\ 4\ 2\ 2\ 0 \\ \underline{0\ 4} \\ 1\ 0\ 0\ 8\ 8 \end{array}$$

$$H(\text{key})=0088$$

移位叠加

$$\begin{array}{r} 5\ 8\ 6\ 4 \\ 0\ 2\ 2\ 4 \\ \underline{0\ 4} \\ 6\ 0\ 9\ 2 \end{array}$$

$$H(\text{key})=6092$$

间界叠加

第四节 哈希表

二、哈希函数(折叠法一特性)

- 折叠法适合于关键字的数字位数特别多，而且每一位上数字分布大致均匀的情况

第四节 哈希表

二、哈希函数(除留余数法)

- 取关键字被某个不大于哈希表长 m 的数 p 除后所得余数为哈希地址

$$H(\text{key}) = \text{key} \text{ MOD } p \quad (p \leq m)$$

- m 为表长
- p 为不大于 m 的素数或是不含20以下的质因子

第四节 哈希表

二、哈希函数(除留余数法— p 值)

- 给定一组关键字为：12, 39, 18, 24, 33, 21，若取 $p=9$ ，则他们对应的哈希函数值将为：

3, 3, 0, 6, 6, 3

- 可见，若 p 中含质因子3，则所有含质因子3的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能。

第四节 哈希表

二、哈希函数(除留余数法一特性)

- 除留余数法是一种最简单、最常用的构造哈希函数的方法
- 不仅可以对关键字直接取模(MOD)，也可在折叠、平方取中等运算之后取模

第四节 哈希表

三、处理冲突的方法

- “处理冲突” 的实际含义是：为产生冲突的地址寻找下一个哈希地址。
- 处理冲突的方法主要有三种：
 1. 开放定址法
 2. 再哈希法
 3. 链地址法

第四节 哈希表

三、处理冲突的方法(开放定址法)

- 为产生冲突的地址 $H(\text{key})$ 求得一个地址序列:

$$H_0, H_1, H_2, \dots, H_s, 1 \leq s \leq m-1$$

$$H_i = [H(\text{key}) + d_i] \text{ MOD } m \quad i=1, 2, \dots, s$$

- $H(\text{key})$ 为哈希函数
- m 为哈希表长

第四节 哈希表

三、处理冲突的方法(开放定址法—线性探测)

- 当 d_i 取 $1, 2, 3, \dots, m-1$ 时, 称这种开放定址法为**线性探测再散列**
- **举例:** 给定关键字集合 $\{19, 01, 23, 14, 55, 68, 11, 82, 36\}$, 设定哈希函数 $H(\text{key}) = \text{key} \text{ MOD } 11$ (表长=11)

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1		

第四节 哈希表

三、处理冲突的方法(开放定址法—二次探测)

- 当 d_i 取 $= 1^2, -1^2, 2^2, -2^2, 3^2, \dots$ 时, 称这种开放定址法为**二次探测再散列**
- **举例:** 给定关键字集合 {19, 01, 23, 14, 55, 68, 11, 82, 36}, 设定哈希函数 $H(\text{key}) = \text{key} \text{ MOD } 11$ (表长=11)

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11
1	1	2	1	2	1	4		1		3

第四节 哈希表

三、处理冲突的方法(开放定址法—特性)

- **优点：**线性探测再散列只要哈希表中有空位置，总能找到一个不发生冲突的地址
- **二次探测再散列：** $m=4j+3$ 的素数时总能找到。
- **缺点：**易产生“二次聚集”，即在处理同义词的冲突过程中，又添加了非同义词的冲突(如82)，对查找不利

第四节 哈希表

三、处理冲突的方法(再哈希法)

- 构造若干个哈希函数，当发生冲突时，计算下一个哈希地址，直到冲突不再发生，即：

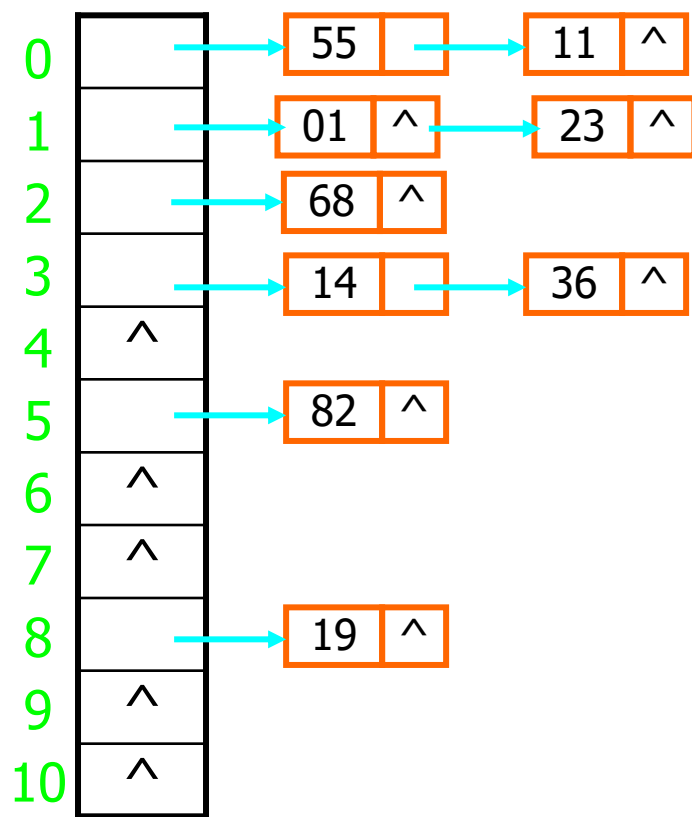
$$H_i = Rh_i(\text{key}) \quad i=1, 2, \dots, k$$

- Rh_i —不同的哈希函数
- 特点：不易产生聚集，但增加计算时间

第四节 哈希表

三、处理冲突的方法(链地址法)

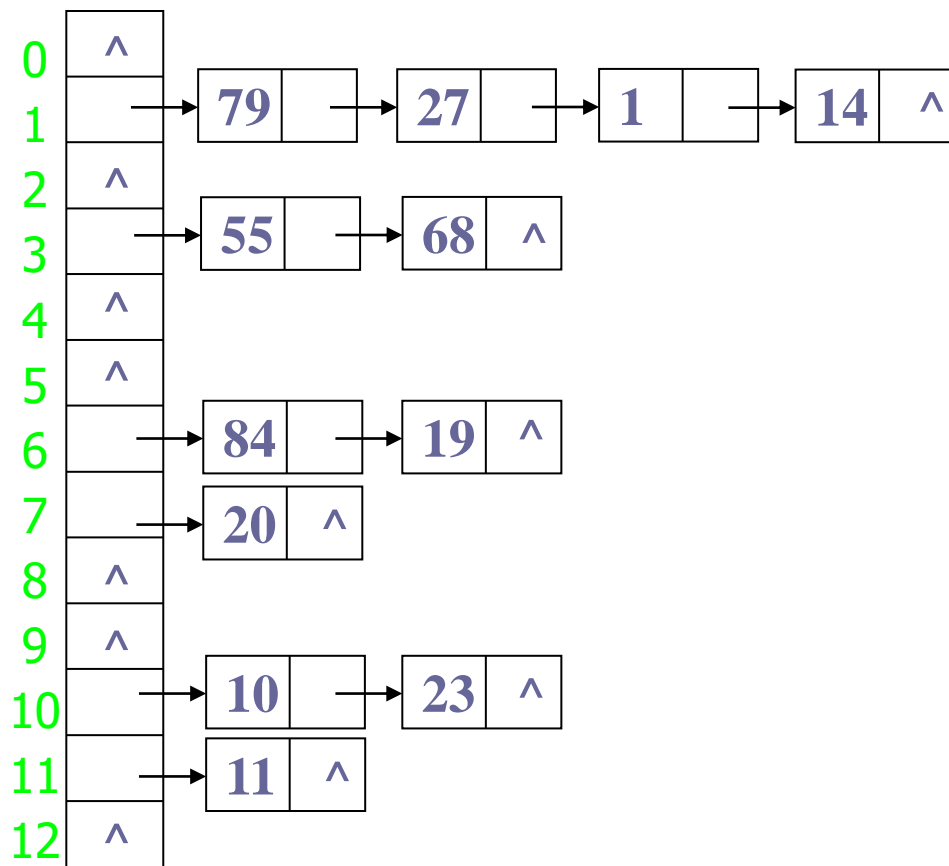
- 将所有哈希地址相同的记录都链接在同一链表中
- 举例：给定关键字集合 {19, 01, 23, 14, 55, 68, 11, 82, 36}，设定哈希函数 $H(\text{key}) = \text{key} \text{ MOD } 11$ (表长=11) [表后插入]



第四节 哈希表

三、处理冲突的方法(链地址法一举例)

- 例：已知一组关键字
(19, 14, 23, 1, 68, 20, 84
, 27, 55, 11, 10, 79) 哈希
函数为： $H(\text{key}) = \text{key}$
 $\text{MOD } 13$, 用链地址法处
理冲突[表头插入]



第四节 哈希表

四、哈希表的实现

- 假设哈希函数为关键字求模运算，哈希表用拉链法解决冲突，其结构可以定义如下：

```
#define LEN 32
```

```
struct node {  
    int data;  
    struct node *next; };
```

```
struct node *HashTab[LEN];
```

第四节 哈希表

四、哈希表的实现—哈希函数hash()

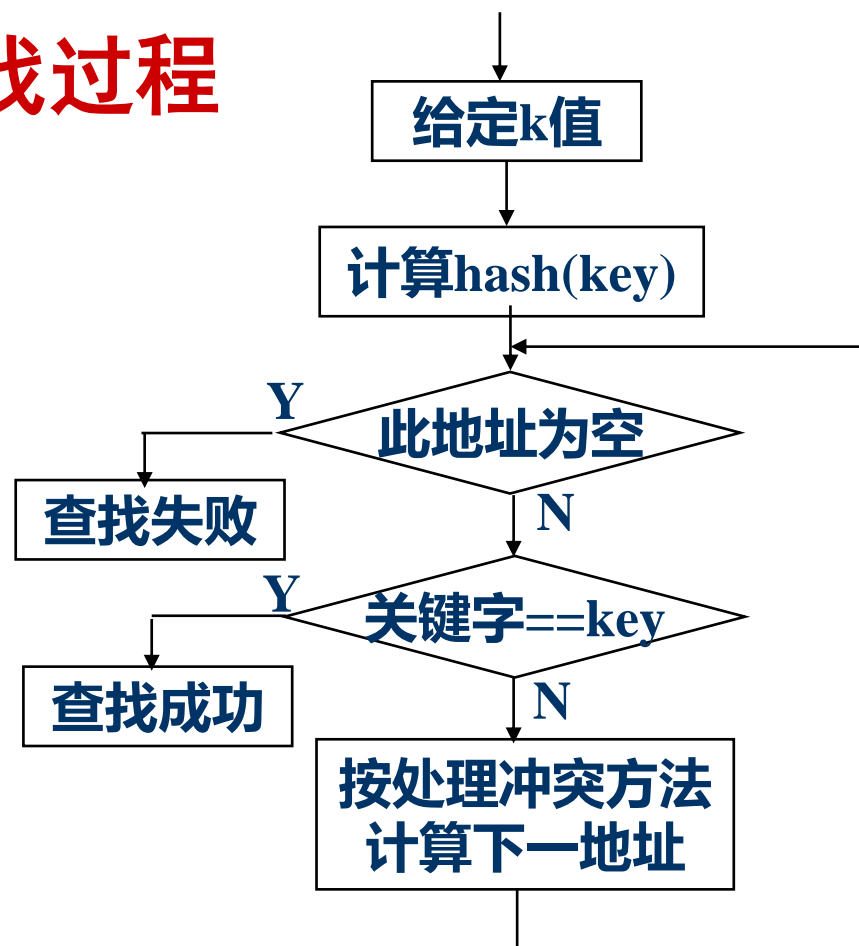
- 返回值为哈希表地址

```
int hash(int key) {  
    retAddr = key MOD LEN;  
    return(retAddr);  
}
```

第四节 哈希表

四、哈希表的实现—查找过程

- 对于给定值key，计算哈希地址 $p = \text{hash}(\text{Key})$
- 若 $\text{HashTab}[p] = \text{NULL}$ ，则查找不成功
- 若 $\text{HashTab}[p] \rightarrow \text{data} = \text{key}$ ，则查找成功
- 否则 “求下一地址”，再进行比较



第四节 哈希表

四、哈希表的实现—查找函数search()

- 若找到key，返回其结点指针；否则将其插入表中再返回其结点指针[表头插入]

```
node *search(int key) {  
    p = hash(key); q = HashTab[p];  
    while (q != NULL) {  
        if (q->data == key) return(q);  
        q = q->next;}  
    q = new node;  
    q->data = key; q->next = HashTab[p];  
    HashTab[p] = q;  
    return(q);}
```

第四节 哈希表

四、哈希表的实现—删除函数remove()

- 若key在表中，删除并返回1；否则仅返回0。

```
int remove(int key) {  
    p = hash(key);  
    r = HashTab[p]; q=r;  
    while (q != NULL) {  
        if (q->data == key) {  
            if (q== HashTab[p]) HashTab[p]=q->next;  
            else r->next=q->next;  
            delete(q); return(1);}  
        r = q; q = q->next;}  
    return(0);}
```

第四节 哈希表

五、哈希查找的性能分析

- 虽然哈希表在关键字与记录的存储位置之间建立了直接映象，但由于“冲突”的产生，使得哈希表的**查找过程**仍然是一个**给定值和关键字进行比较**的过程。
- 因此，**仍需以平均查找长度 (ASL)**作为衡量哈希表的查找效率的量度。

第四节 哈希表

五、哈希查找的性能分析(线性探测再散列举例)

- 假设每个关键字的查找概率相同，则：

$$\begin{aligned} ASL &= (1/9) (1 \times 4 + 2 \times 2 + 3 \times 1 + 5 \times 1 + 6 \times 1) \\ &= 22/9 \end{aligned}$$

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		
1	1	2	1	3	6	2	5	1		

第四节 哈希表

五、哈希查找的性能分析(二次探测再散列举例)

- 假设每个关键字的查找概率相同，则：

$$\begin{aligned} \text{ASL} &= (1/9) (1 \times 5 + 2 \times 2 + 3 \times 1 + 4 \times 1) \\ &= 16/9 \end{aligned}$$

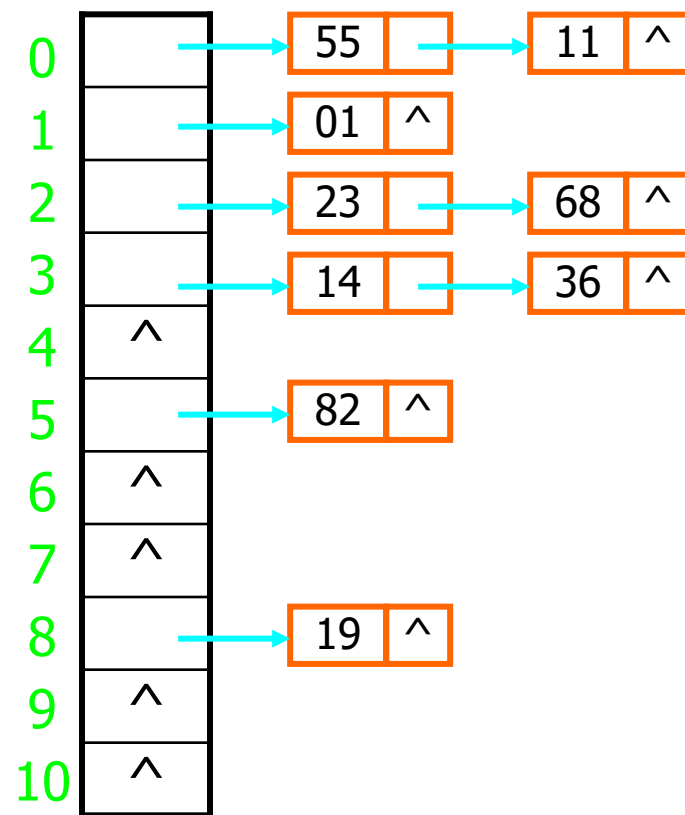
0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11
1	1	2	1	2	1	4		1		3

第四节 哈希表

五、哈希查找的性能分析(链地址法举例)

- 假设每个关键字的查找概率相同，则：

$$\begin{aligned} ASL &= (1/9) (1 \times 6 + 2 \times 3) \\ &= 12/9 \end{aligned}$$



第四节 哈希表

五、哈希查找的性能分析

- 决定哈希表查找的ASL的因素：

1. 选用的哈希函数
2. 选用的处理冲突的方法
3. 哈希表的装填因子

第四节 哈希表

五、哈希查找的性能分析(装填因子)

- 哈希表的装填因子是哈希表中填入的记录数与哈希表的长度的比值，即：

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

- 装填因子 α 标志哈希表的装满程度

第四节 哈希表

五、哈希查找的性能分析(装填因子)

直观来看：

- 装填因子 α 越小，发生冲突的可能性就越小
- 装填因子 α 越大，发生冲突的可能性就越大

第四节 哈希表

五、哈希查找的性能分析(平均查找长度ASL)

- 线性探测再散列的哈希表查找成功时:
 $ASL \approx (1/2) (1 + 1/(1-\alpha))$
- 二次探测再散列的哈希表查找成功时:
 $ASL \approx -(1/\alpha) \ln(1-\alpha)$
- 链地址法处理冲突的哈希表查找成功时:
 $ASL \approx (1 + \alpha/2)$

练习

1. 画出在初始为空的AVL树中依次插入2, 1, 3, 5, 8, 4, 7, 6时该树的生长全过程，并在有“旋转”时说出“旋转”的类型。
2. 画出在初始为空的AVL树中依次插入b, a, c, e, h, d, g, f时该树的生长全过程，并在有“旋转”时说出“旋转”的类型。

作业

作业1: 对以下序列进行二分查找，请画出判定树，并计算查找成功时的ASL。1, 2, 3, 4, 5, 6, 7, 8,

作业2: 请按照下列序列画出二叉排序树的生长过程。2, 1, 3, 5, 8, 4, 7, 6, 并计算查找成功时的ASL.

作业3（练习）: 画出在初始为空的AVL树中依次插入2, 1, 3, 5, 8, 4, 7, 6时该树的生长全过程，并在有“旋转”时说出“旋转”的类型，及调整方法。

作业

作业4 哈希表依次插入10, 5, 8, 23, 19, 30, 25, 26, 假设表长为13, 用开放地址线性探测解决冲突。

- (1) 写哈希函数;
- (2) 画出插入各关键字后的哈希表。
- (3) 求查找成功时的ASL.

作业5 哈希表依次插入10, 5, 8, 23, 19, 30, 25, 26, 用拉链法解决冲突（**表后插入**）。

- (1) 写哈希函数;
- (2) 画出插入各关键字后的哈希表。
- (3) 求查找成功时的ASL.