数据结构

9 查找

主要内容

- 查找的概念
- 静态查找表
  - 线性查找
  - 折半查找
  - 分块查找
- 动态查找表
  - 二叉查找树、平衡二叉树
  - 哈希查找

查找的概念

- 查找:
  - 在数据集合中寻找满足某种条件的数据元素
  - 查找的基本操作是关键字的比较.
- 关键字
  - 数据元素的部分数据(数据项)或就是数据元素本身
  - 可能唯一标识一个数据元素,也可能多个元素具有同样的关键字

查找的概念

- 静态查找表
  - 数据集合“只读”，即只能对该数据结合进行查询操作
- 动态查找表
  - 数据集合“可写”，即可以对集合元素做插入、删除等操作
- 注：
  - 查找算法和数据存储的结构有关

查找的概念

- 平均查找长度
  - **Average Search Length**
  - 查找就是不断将数据元素的关键字与待查找关键字进行比较，查找算法在查找成功时(统计意义上)平均比较的次数称作平均查找长度

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

  - **Pi**：查找第**i**个数据元素的概率
  - **Ci**：查找该元素的过程中比较的次数

线性查找

- **线性查找**(又称**顺序查找**)
  - 主要用于在线性的数据结构中进行查找

    $(a_1, a_2, a_3, \ldots, a_n)$

- **基本思想**
  - 从线形结构的一端开始，顺序用各数据的关键字与给定值**x**进行比较，直到找到与其值相等的元素，则搜索成功，给出该对象在表中的位置
  - 若整个表都已检测完仍未找到关键字与**x**相等的元素，则搜索失败，给出失败信息

线性查找

i

0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**57**

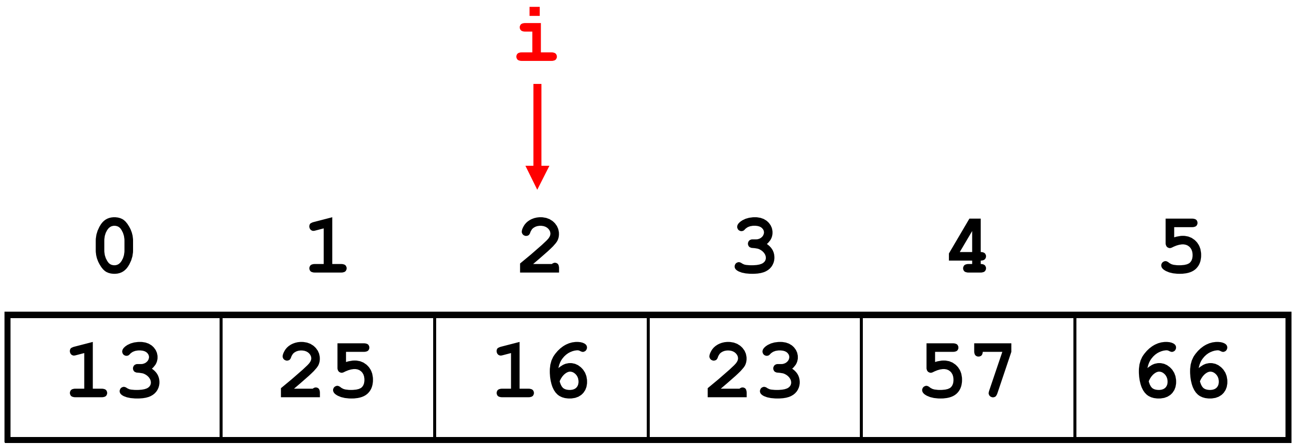
线性查找

i

0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**57**

线性查找



- 欲查找**57**

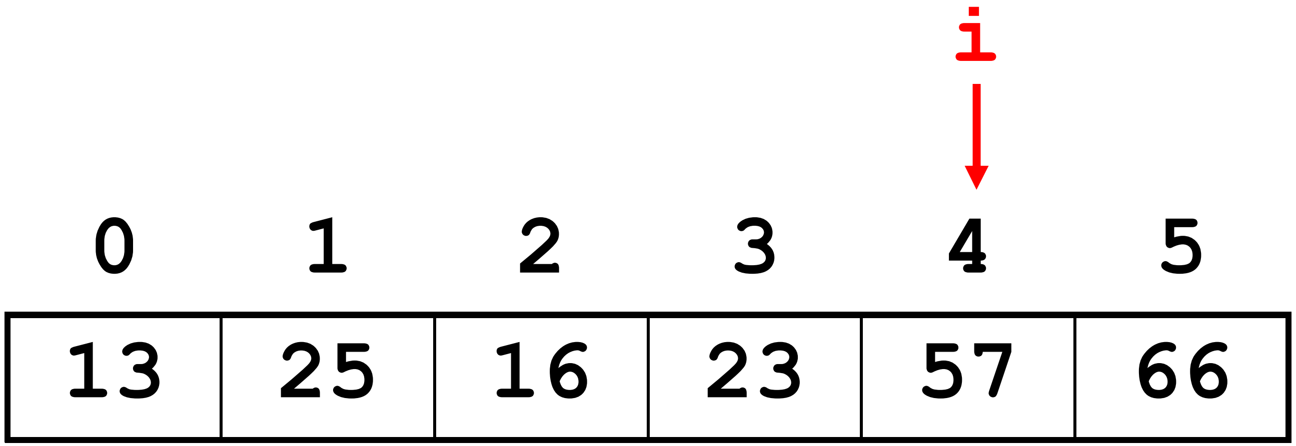
线性查找

i

0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**57**

线性查找



- 欲查找**57**
- 找到，位置在第**4**个单元

线性查找

i

0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**27**

线性查找

	i				
0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**27**

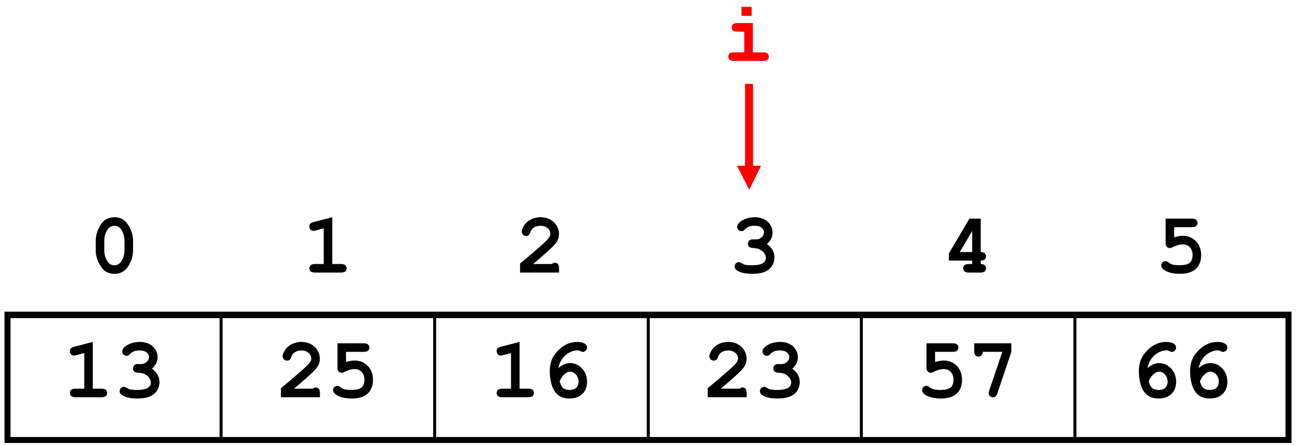
线性查找

i

0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**27**

线性查找



- 欲查找**27**

线性查找

i

0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**27**

线性查找

i

0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**27**

线性查找

i

0	1	2	3	4	5
13	25	16	23	57	66

- 欲查找**27**
- 找不到

线性查找

- 时间复杂度
  - 最好情况：O(1)
    - 第一个就是欲查找值
  - 最差情况：O(n)
    - 欲查找值在最后一个单元
    - 或搜索了所有的数据才得知找不到
  - 平均情况：O(n)
    - 等概率时： $ASL = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$

折半查找

- 折半查找
  - 基于有序表(有序的顺序表)
- 基本思想
  - **middle = n/2**
  - 比较**key** 和 **Data[middle]**
  - 若**key < Data[middle]**：欲查找值在前半段
  - 若**key > Data[middle]**：欲查找值在后半段
  - 若**key = Data[middle]**：查找成功
  - 若搜索区间已缩小到一个数据仍未找到：找不到

折半查找

low					mid						high
0	1	2	3	4	5	6	7	8	9	10	11
5	7	12	25	34	37	43	46	58	80	92	105

- key = 25
- low = 0
- high = 11
- mid = (low + high)/2 = 5

折半查找

low mid high

0	1	2	3	4	5	6	7	8	9	10	11
5	7	12	25	34	37	43	46	58	80	92	105

- **key = 25**
- **key < Data[mid]**
- 搜索范围应缩小为**low ~ mid-1**
- 即**high = mid - 1**

折半查找

low　　mid　　high

0	1	2	3	4	5	6	7	8	9	10	11
5	7	12	25	34	37	43	46	58	80	92	105

- key = 25
- low = 0
- high = 4
- mid = (low + high)/2 = 2

折半查找

low mid high

0	1	2	3	4	5	6	7	8	9	10	11
5	7	12	25	34	37	43	46	58	80	92	105

- **key = 25**
- **key > Data[mid]**
- 搜索范围应缩小为**mid+1 ~ high**
- 即**low = mid + 1**

折半查找

mid

low high

0	1	2	3	4	5	6	7	8	9	10	11
5	7	12	25	34	37	43	46	58	80	92	105

- **key = 25**
- **low = 3**
- **high = 4**
- **mid = (low + high)/2 = 3**

折半查找

mid

low high

0	1	2	3	4	5	6	7	8	9	10	11
5	7	12	25	34	37	43	46	58	80	92	105

- **key = 25**
- **key = Data[middle]**
- 找到
- 位置在第3个单元

折半查找：迭代算法

```
int Search_Bin(SqList ST, KeyType key) {
  int low  = 1, high = ST.Length();
  while(low <= high) {
    int mid = (low + high)/2;
    if( key == ST[mid].key)        //找到
        return mid;
    else if(key < ST[mid].key) //前半段
        high = mid - 1;
    else low = mid + 1;                 //后半段
  }
  return 0;            //如果left>right，说明找不到
}
```

折半查找：递归算法

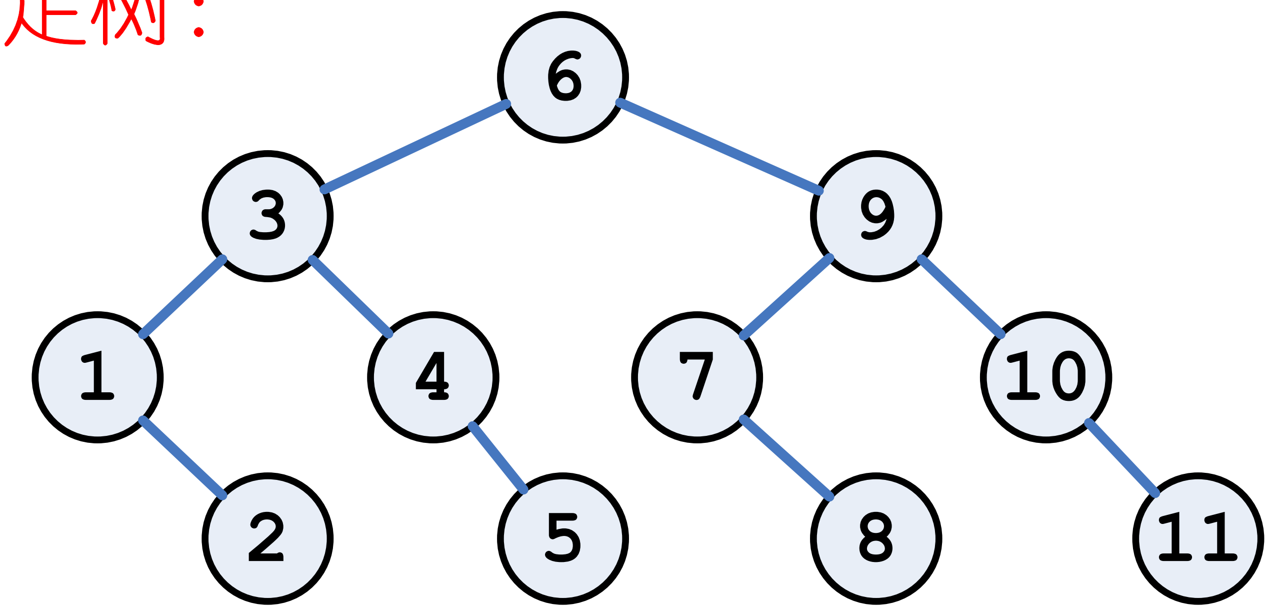
```
int Search_Bin(SqList ST, KeyType key,
                  int low,int high) {
  if(low>high) return -1;
  mid = (low + high)/2;
  if(ST[mid].key == key)
    return mid;
  else if(ST[mid].key < key)
    mid = Search_Bin(ST,key,mid+1,high);
  else
    mid = Search_Bin(ST,key,low,mid-1);
  return mid;
}
```

折半查找

- 性能分析
  - 比较次数 = 折半的次数
  - **n**个元素，最多 $\lfloor \log_2 n \rfloor + 1$ 次折半

$2^k < n < 2^{k+1}$

判定树：

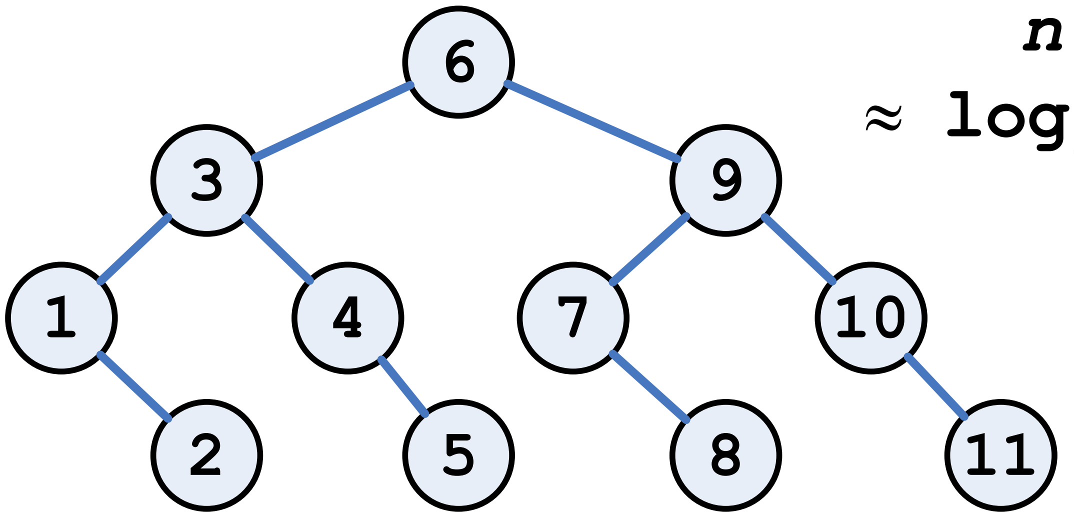


折半查找

- 平均查找长度
  - 第j层有2^{j-1}个元素，每个元素需要比较h次
  - 等概率条件下 $ASL = \sum_{j=1}^{h} \frac{1}{n} j \times 2^{j-1}$

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

$$\approx \log_2(n+1) - 1$$



折半查找

- 总结
  - 顺序查找和折半查找都针对静态查找表
  - 折半查找效率较高
  - 但是
    - 折半查找要求数据有序
    - 并且存储结构必须是顺序存储（链表怎么折半？）

分块查找(索引顺序查找)

- 索引顺序表:顺序表+索引表
  - 顺序表分块有序
  - 索引项:子表最大关键字、子表首指针

  索引表按照关键字有序

索引表

最大关键字	22	49	86
起始地址	1	6	10

22	12	13	20	9	33	26	49	38	60	55	74	51	86	80

分块查找(索引顺序查找)

- 分块查找:

1）先查找索引表，确定数据元素（记录）所在的块（子表）

2）在块（子表）中顺序查找

- 平均查找长度:

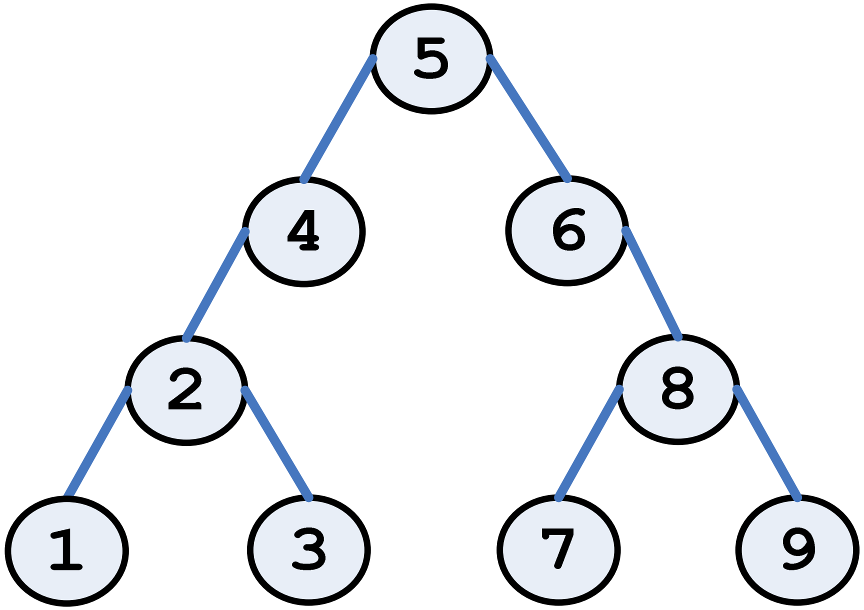
$$ASL_{bs}=ASL_b+ASL_w$$

假设长度为n顺序表被均匀地分成b块,每块有s个记录，则：ASLbs= (b+1)/2 + (s+1)/2

=(n/s+s)/2+1

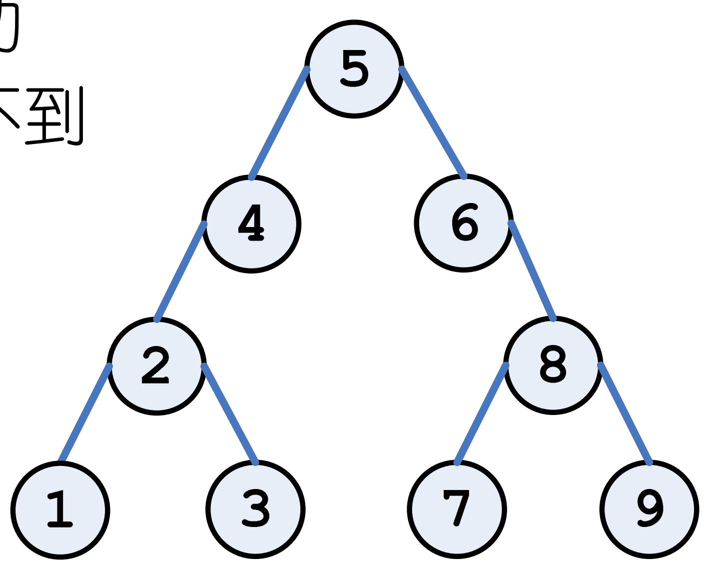
二叉查找树(二叉排序树)

- 二叉查找树
  - 是一棵二叉树，不过
  - 左子树节点的值<根节点的值<右子树节点的值



二叉查找树

- 二叉查找树的查找
  - 从根节点开始查找
  - 比较欲查找值和当前节点的值
    - 若欲查找值更小，则向左子树继续查找
    - 若欲查找值更大，则向右子树继续查找
    - 若相等则查找成功
    - 若走到了底则找不到

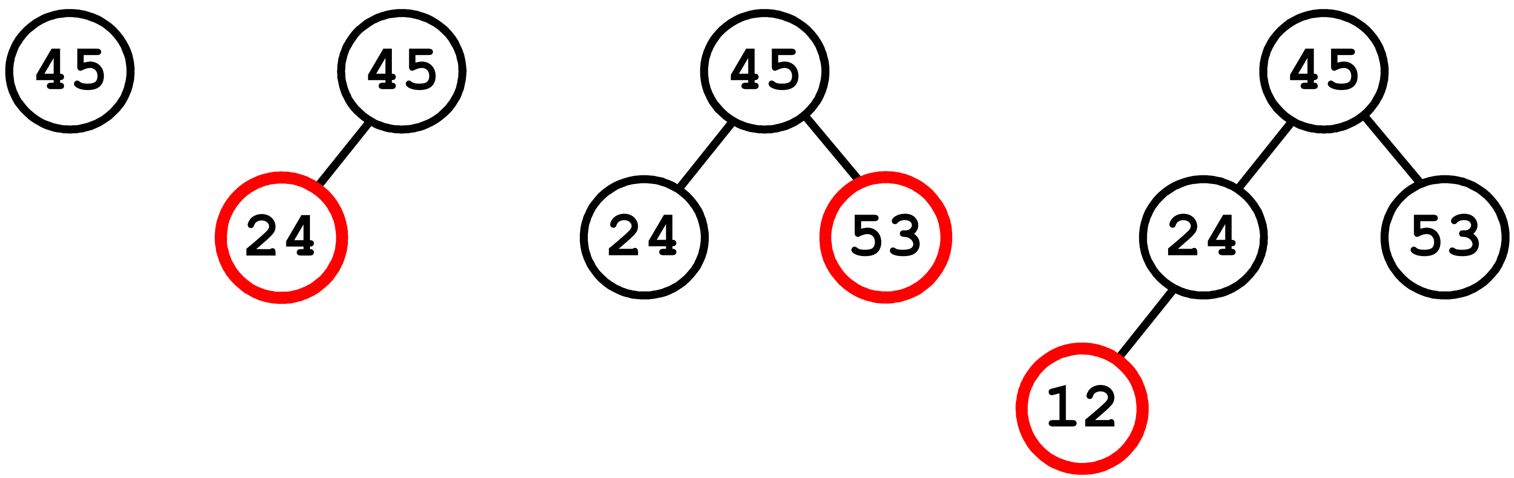


二叉查找树

```
//二叉查找树的查找算法
BiTree SearchBST(BiTNode* T, KeyType key){
    //空树，或者找到，返回树根
    if((!T) || EQ(key, T->data.key))
        return T;
    else
        if(LT(key, T->data.key)) //左子树
            return SearchBST(T->lchild, key);
        else                     //右子树
            return SearchBST(T->rchild, key);
}
```

二叉查找树结点的插入

- 如果是空树
  - 新节点作为树根
- 否则
  - 为这个新节点找到一个父节点



二叉查找树结点的插入

- 首先改写查找算法：

```
BiTNode* SearchBST(BiTNode* T, KeyType key,
         BiTNode* foT,BiTNode* &fop);
```

输入参数：

T （查找树的根结点）

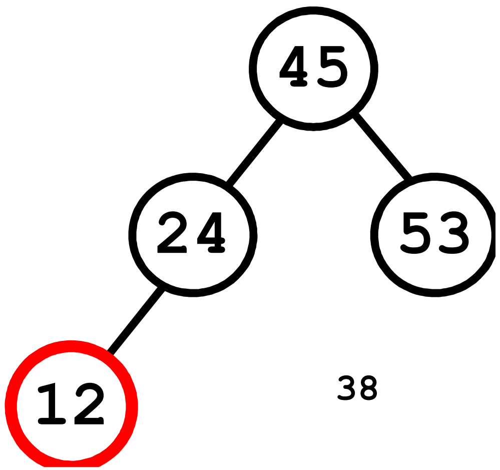
key （查找关键字）

foT （当前查找树根的双亲结点）

输出参数：

返回值表示查找到的结点，返回空值表示未查找到！

fop （返回值结点的双亲结点）



二叉查找树结点的插入

- 首先改写查找算法：

```
BiTNode* SearchBST(BiTNode* T, KeyType key,
         BiTNode* foT,BiTNode* &fop);
```

输入参数：

T （查找树的根结点）　　　　　　　　查找关键字：**12**

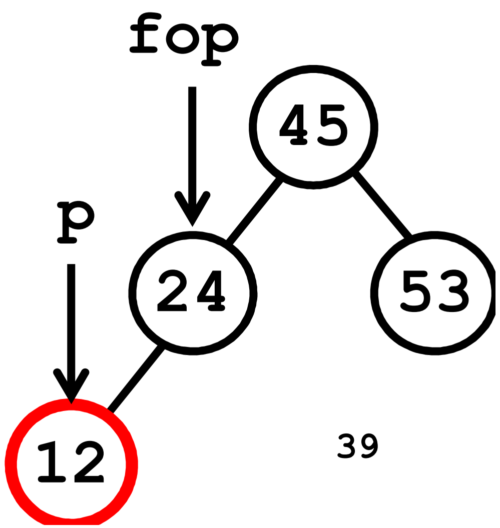
key （查找关键字）

foT （当前查找树根的双亲结点）

输出参数：

返回值表示查找到的结点，返回空值表示未查找到！

fop （返回值结点的双亲结点）



二叉查找树结点的插入

- 首先改写查找算法：

```
BiTNode* SearchBST(BiTNode* T, KeyType key,
         BiTNode* foT,BiTNode* &fop);
```

输入参数：

T （查找树的根结点）

key （查找关键字）

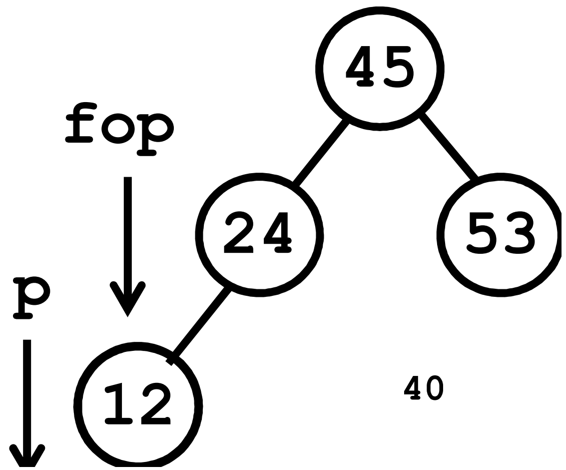
foT （当前查找树根的双亲结点）

查找关键字：**11**

输出参数：

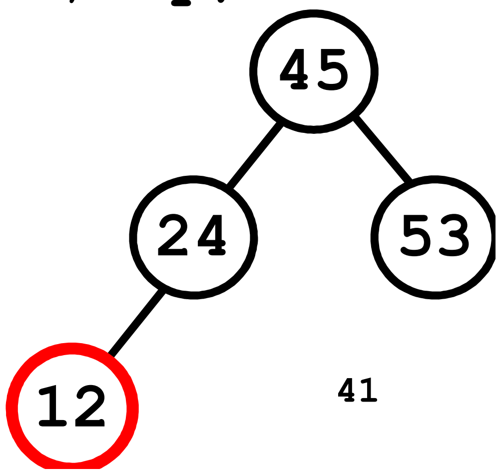
返回值表示查找到的结点，返回空值表示未查找到！

fop （返回值结点的双亲结点）



二叉查找树结点的插入

```
BiTNode* SearchBST(BiTNode* T, KeyType key,
        BiTNode* foT,BiTNode* &fop){
  if(!T|| EQ(key, T->data.key) ) {
      fop = foT; return T;}
  else if (LT(key, T->data.key))
      return SearchBST(T->lchild, key, T,fop);
  else
     return SearchBST(T->rchild,key,T,fop);
}
```



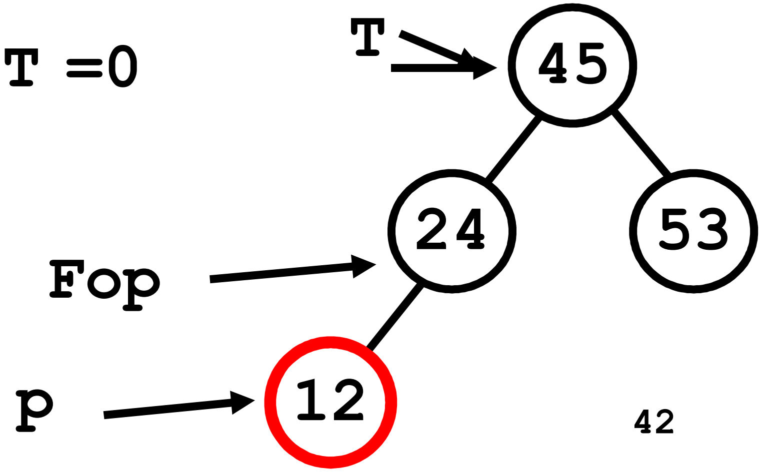
调用：

```
SearchBST(T, 12, NULL, T);
```

改写的查找算法

```
BiTNode* SearchBST(BiTNode* T, KeyType key,
              BiTNode* foT,BiTNode* &fop){
   if(!T|| EQ(key, T->data.key) ) {
        fop = foT; return T;
   }
   else if (LT(key, T->data.key))
      return SearchBST(T->lchild, key, T,fop);
   else
      return SearchBST(T->rchild, key, T,fop);
}
```

foT =0



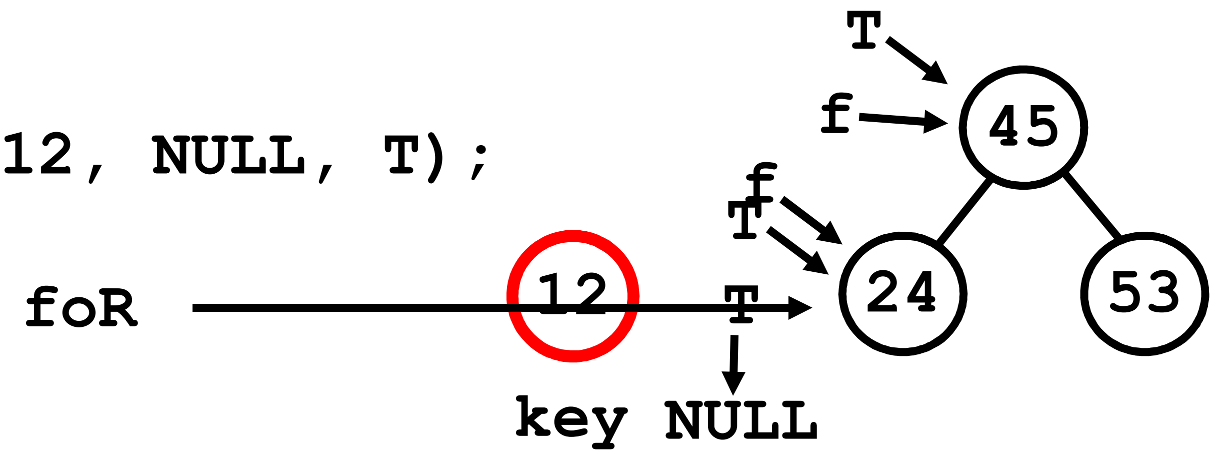
调用：

```
SearchBST(T, 12, NULL, T);
```

```
BiTNode* SearchBST(BiTNode* T, KeyType key,
          BiTNode* f, BiTNode* &fop){
   if(!T|| EQ(key, T->data.key) ) {
       fop = f; return T;
   }
   else if (LT(key, T->data.key))
       return SearchBST(T->lchild, key, T,fop);
   else
       return SearchBST(T->rchild, key, T,fop);
}
```

最初调用：

SearchBST(T, 12, NULL, T);



- 插入算法

```
Status InsertBST(BiTNode* &T, ElemType e){
  if (!SearchBST(T, e.key, NULL, p){
      //生成新结点
      s = (BiTree) malloc (sizeof(BiTNode));
      s->data = e;
      s->lchild = s->rchild = NULL;
      if (!p) T = s;          //原先是空树
      else if (LT(e.key, p->data.key))
          p->lchild = s;      //作为左孩子插入
      else p->rchild = s;     //作为右孩子插入
      return TRUE;
  }
  else return FALSE;          //已存在
}
```

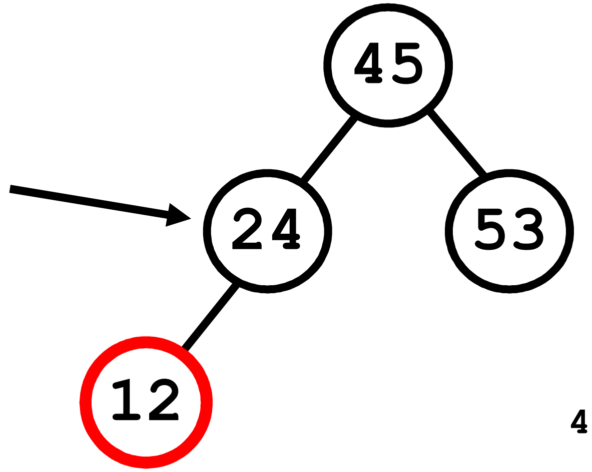
二叉查找树的节点的删除

- 基本思想
  - 找到待删除节点及其父节点

```
BiTNode* SearchBST(BiTNode* T, KeyType key,
      BiTNode* f, BiTNode* &foR);
```

- 剩下来最关键的问题是把该节点删除后，谁来继承该节点的地位？

修改父结点的指针



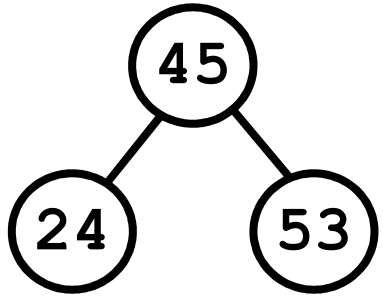
二叉查找树的节点的删除

- 基本思想
  - 找到待删除节点及其父节点

```
BiTNode* SearchBST(BiTNode* T, KeyType key,
        BiTNode* f, BiTNode* &foR);
```

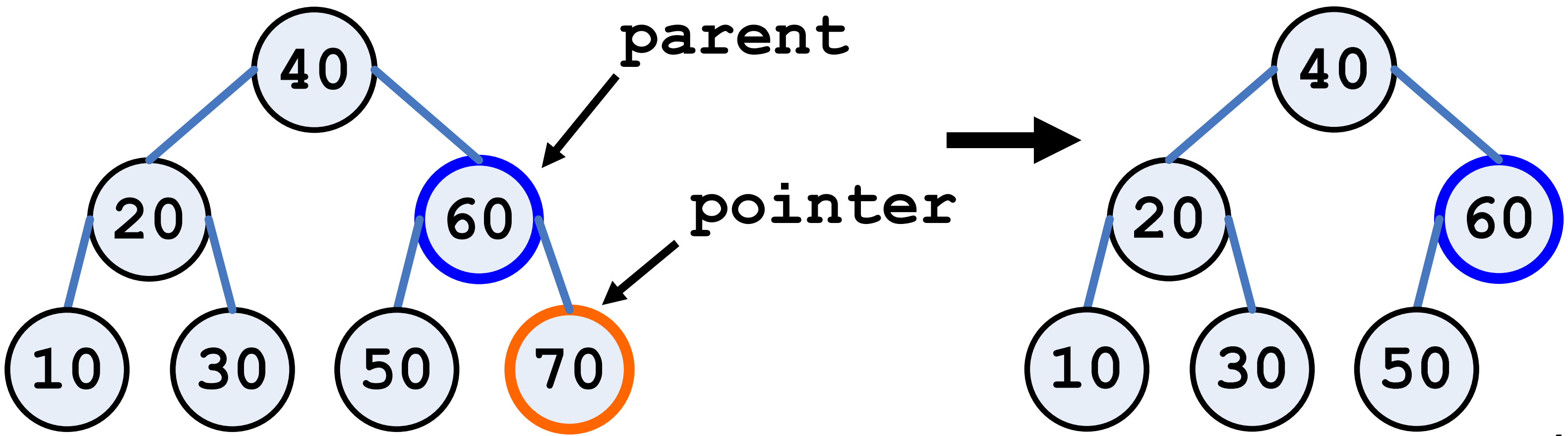
  - 剩下来最关键的问题是把该节点删除后，谁来继承该节点的地位？

修改父结点的指针



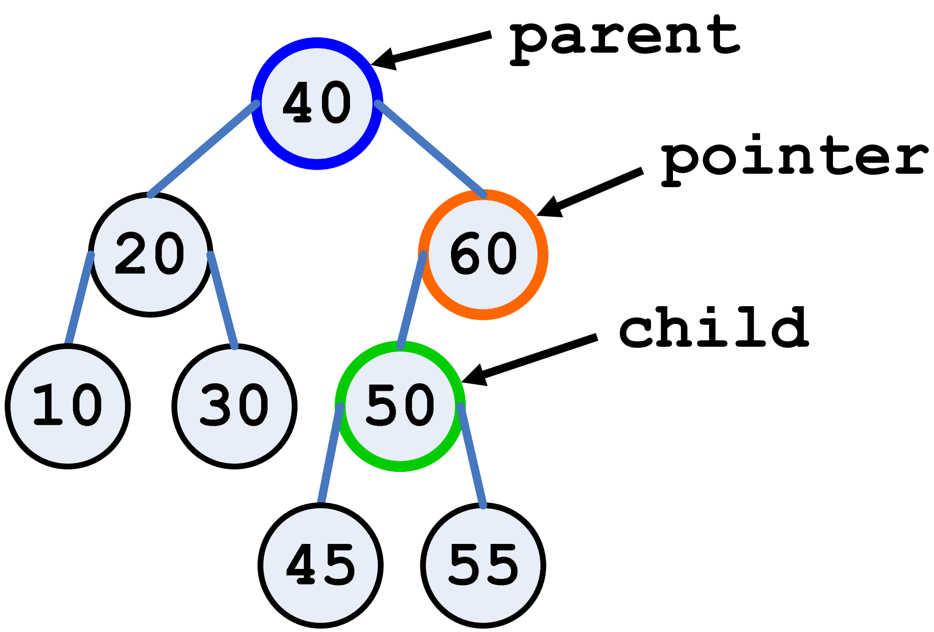
二叉查找树的节点的删除

- 待删除的节点是叶节点
  - 没有“继承人”
  - 若**pointer**是**parent**的左孩子
    - **parent->lchild = NULL**
  - 否则：**parent->rchild = NULL**
  - 最后释放**pointer**的空间



二叉查找树的节点的删除

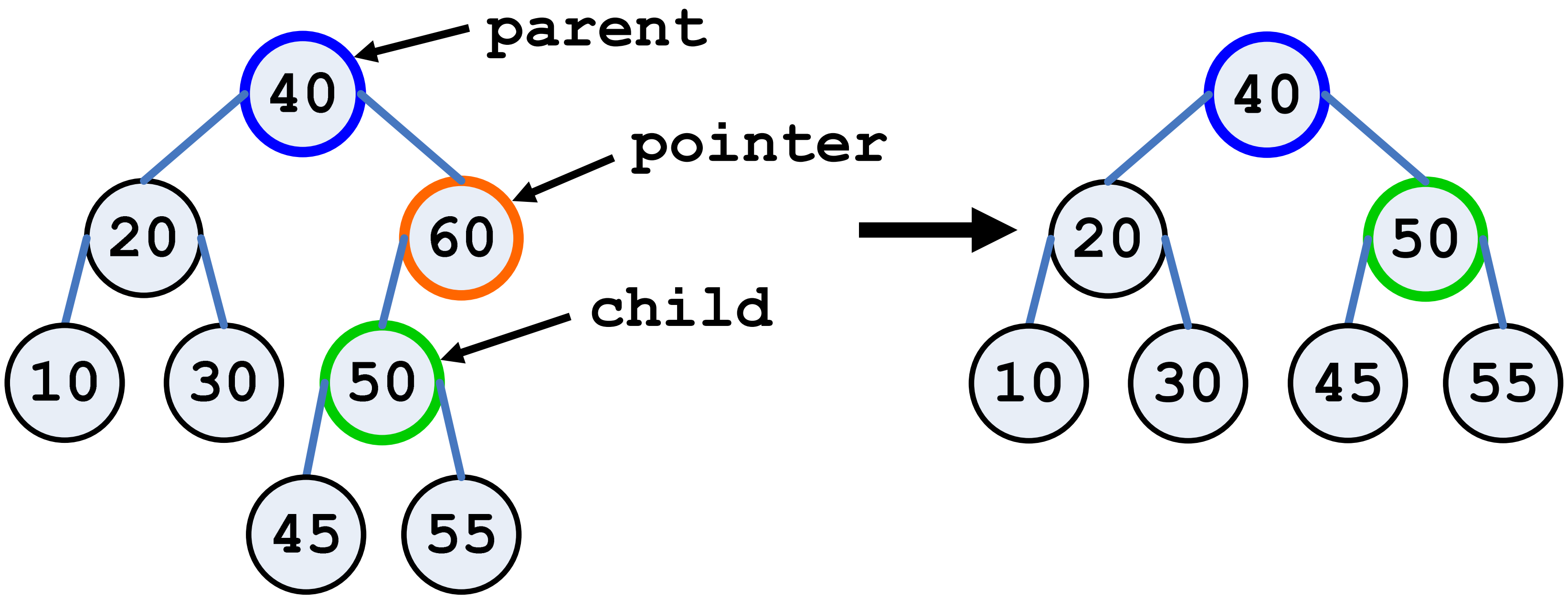
- 待删除的节点只有**1**个孩子



  - 唯一的“继承人”
  - **pointer**的孩子**child**顶替**pointer**的位置

二叉查找树的节点的删除

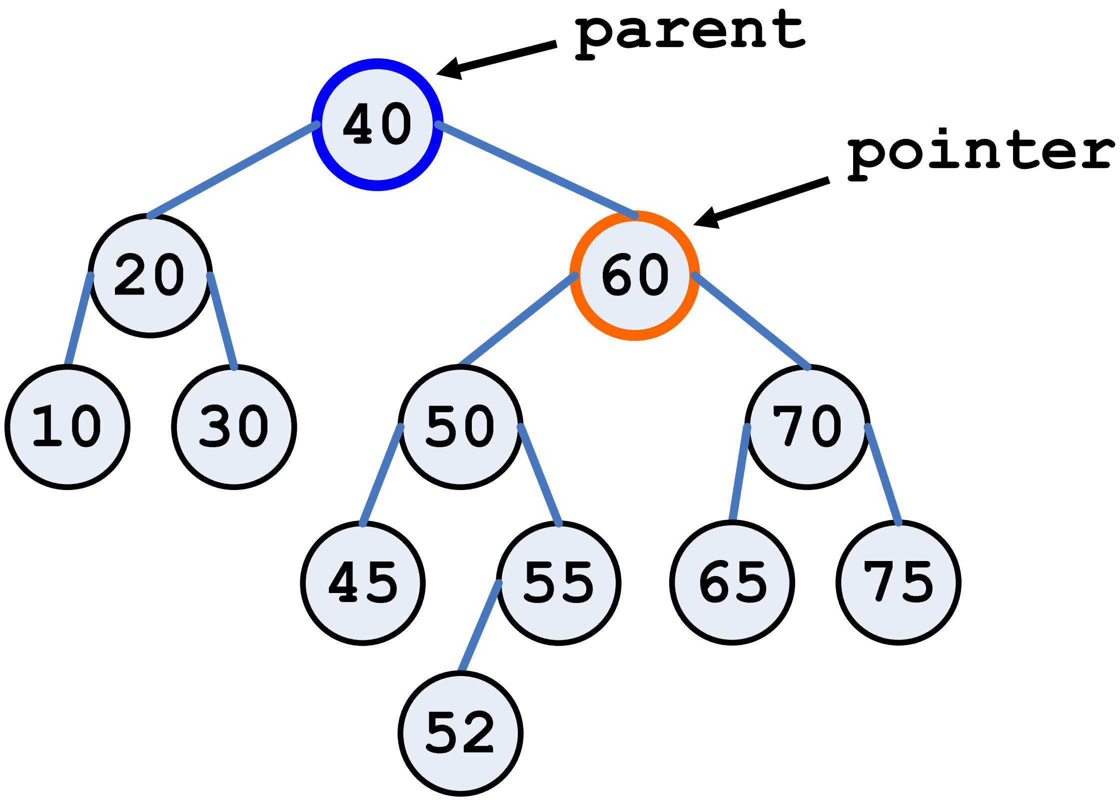
- 待删除的节点只有**1**个孩子



  - 唯一的“继承人”
  - **pointer**的孩子**child**顶替**pointer**的位置

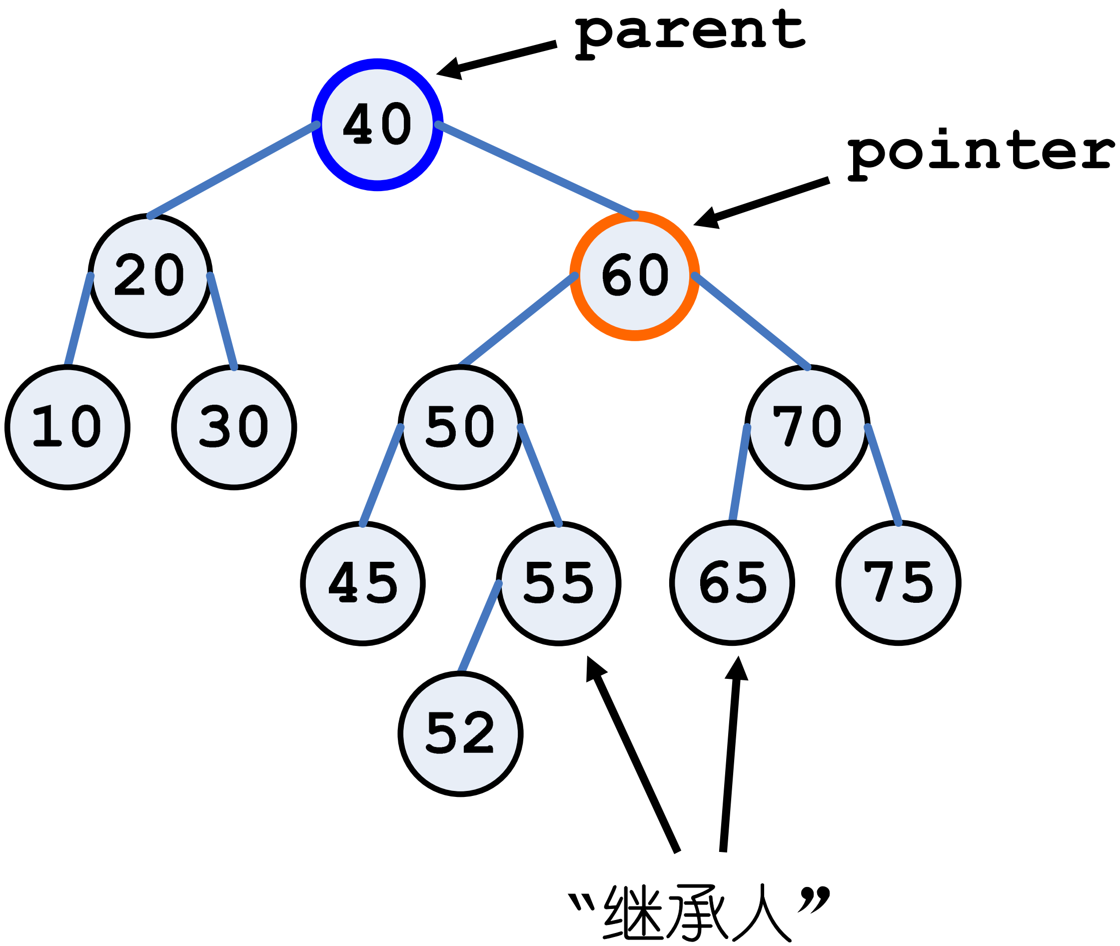
二叉查找树的节点的删除

- 待删除的节点有**2**个孩子



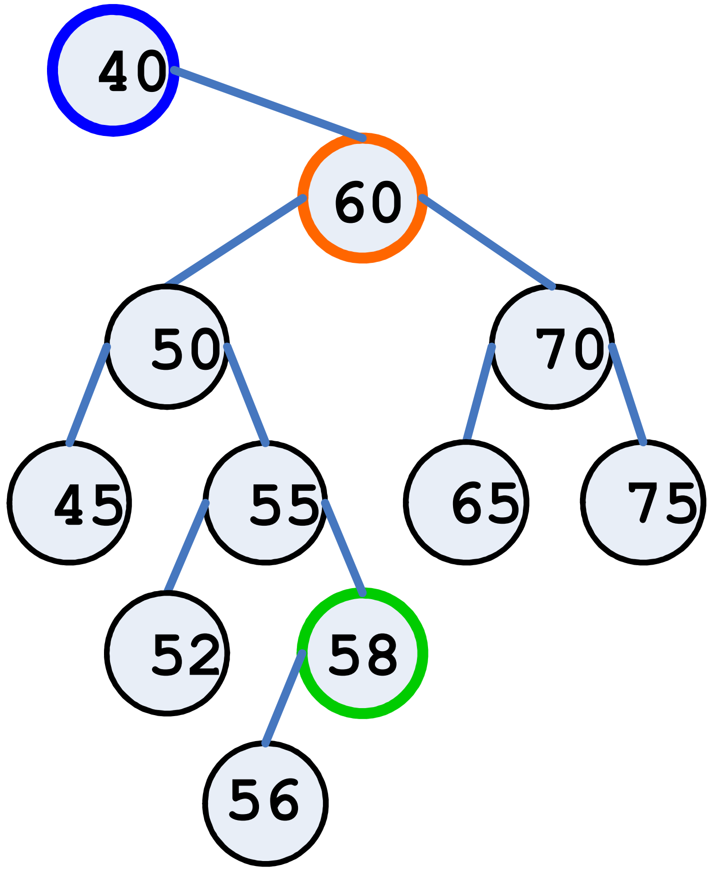
二叉查找树的节点的删除

- 待删除的节点有**2**个孩子



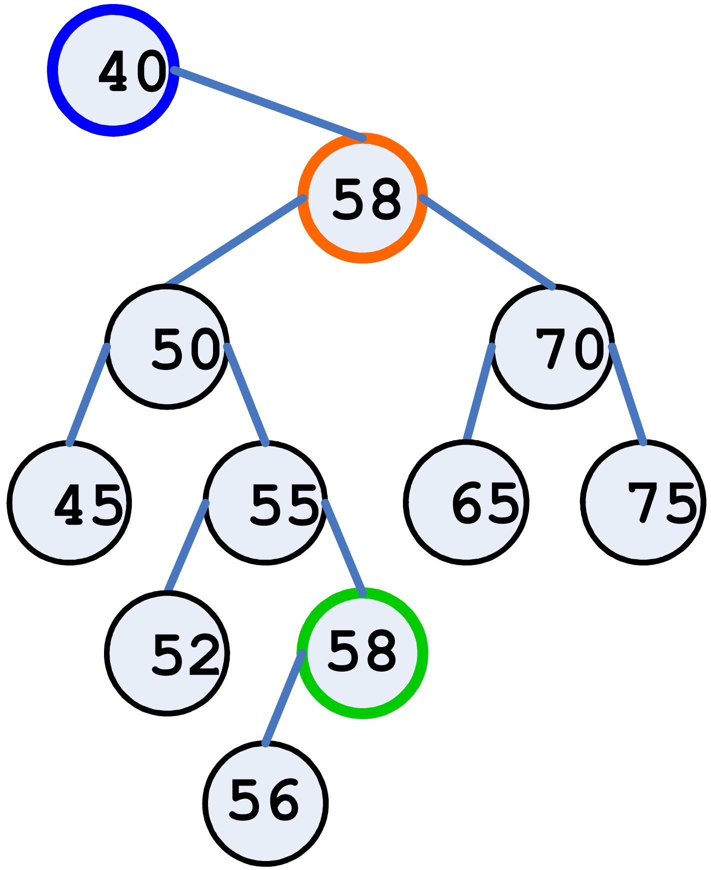
二叉查找树的节点的删除

- “继承人”
  - 左子树中的最右的节点
    - 因为它是左子树中最大的
    - 能够保证比左子树中其余的节点都大；比右子树中的所有节点都小



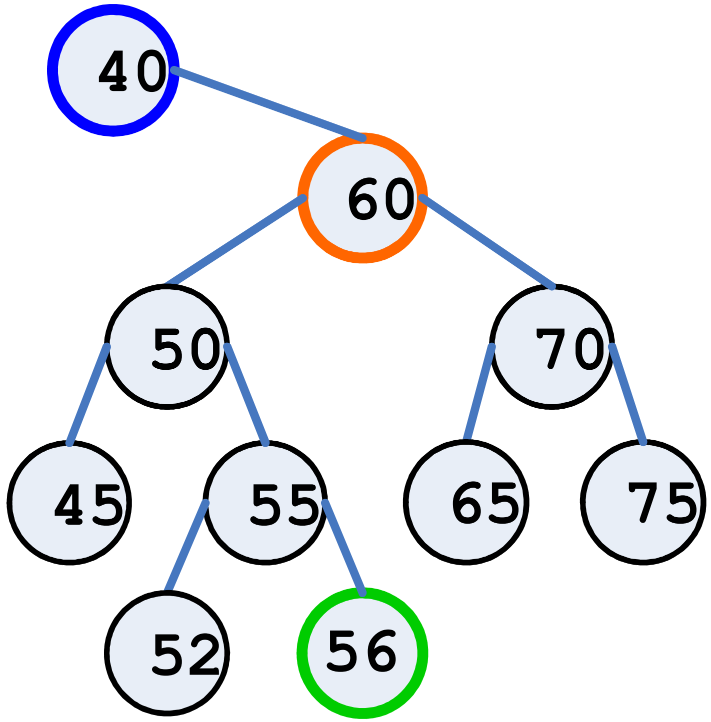
二叉查找树的节点的删除

- “继承人”
  - 左子树中的最右的节点
    - 因为它是左子树中最大的
    - 能够保证比左子树中其余的节点都大；比右子树中的所有节点都小



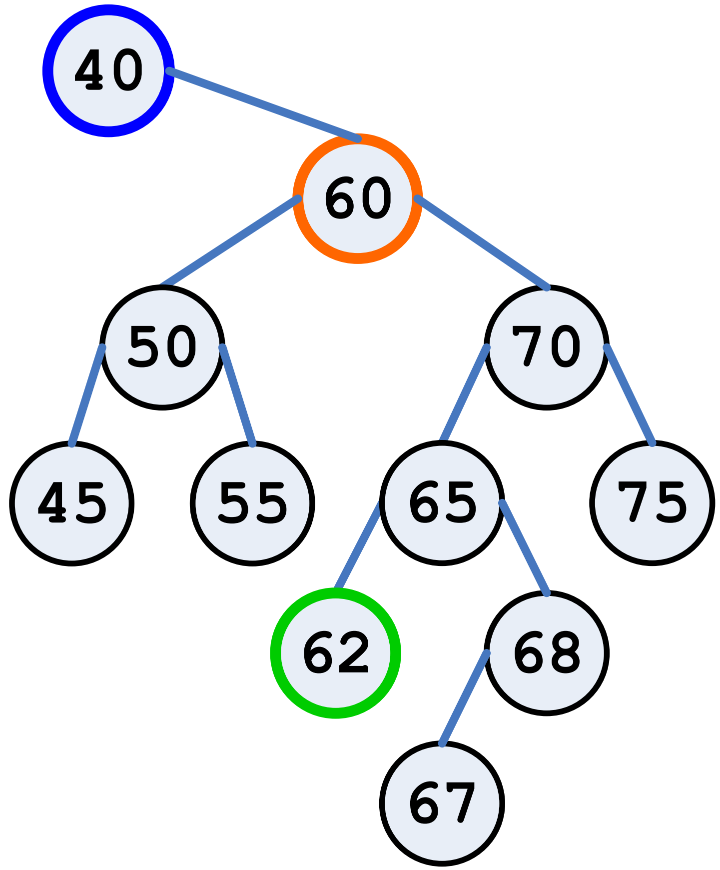
二叉查找树的节点的删除

- “继承人”
  - 左子树中的最右的节点
    - 因为它是左子树中最大的
    - 能够保证比左子树中其余的节点都大；比右子树中的所有节点都小



二叉查找树的节点的删除

- "继承人"
  - 右子树中的最左的节点
    - 因为它是右子树中最小的
    - 能够保证比左子树中的所有节点都大；比右子树中的其它节点都小



二叉查找树的节点的删除

- 算法：

```
bool DeleteBST( BiTNode* &T,
              KeyType key){
   BiTNode *p,*f;
   p = SearchBST(T,key,NULL,f);
   if(!p) return false;
   if( f->lchild==p) Delete(p,f,true);
   else Delete(p,f,false);
}
```

二叉查找树的节点的删除

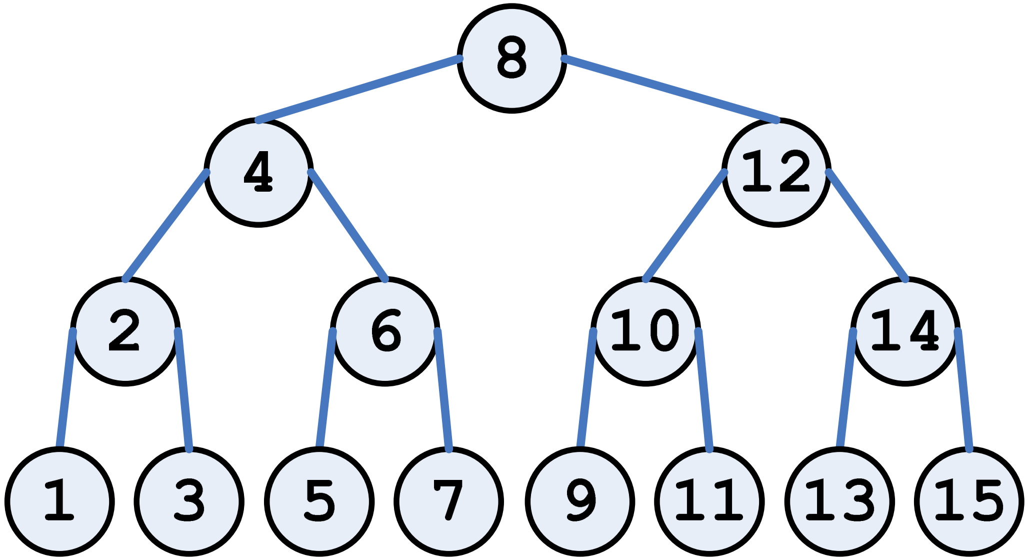
```
void Delete( BiTNode* p, BiTNode* f,
             bool left) {
  if (!p->lchild && !p->rchild) { //p是叶子结点
    if(left) f->lchild = 0;
    else f->rchild = 0; delete p;}
  else if (!p->lchild) { //p只有右孩子
    if(left) f->lchild = p->rchild;
    else f->rchild = p->rchild; delete p;}
  else if (!p->rchild) { //p只有左孩子
    if(left) f->lchild = p->lchild;
    else f->rchild = p->lchild; delete p;}
```

二叉查找树的节点的删除

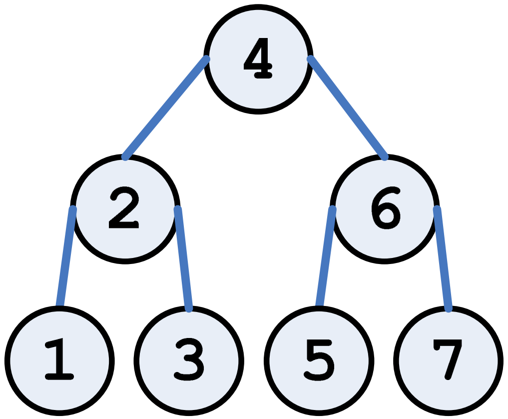
```
else{ //…p的左右孩子都存在
  //找p的左孩子的最右下的结点
   BiTNode* q = p->lchild,*qf = p;
   while( q->rchild ) {
     qf = q;  q = q->rchild; }

   p->data = q->data;  //q的数据顶替p的数据
   if (qf->lchild==q) Delete(q,qf);
   else Delete(q,qf,false);

}
}
```

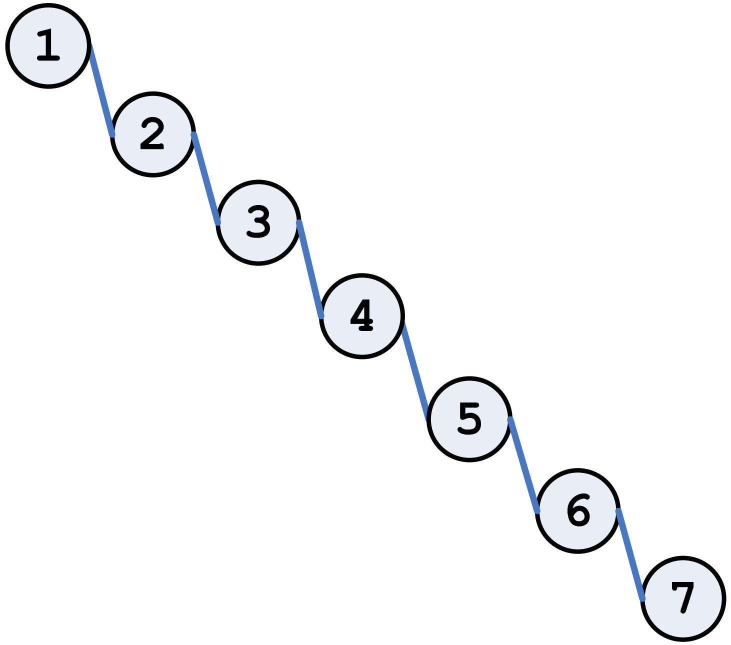
二叉查找树的查找效率

- 查找算法的效率
  - 查找的次数 = 树的高度



  - 因此复杂度=O(h)，问题是h= ？





$$h = \lfloor \log_2 n \rfloor + 1$$

$$ASL = \lfloor \log_2 n \rfloor + 1$$

$$h = n$$

$$ASL = \frac{n + 1}{2}$$

所以二叉搜索树需要平衡

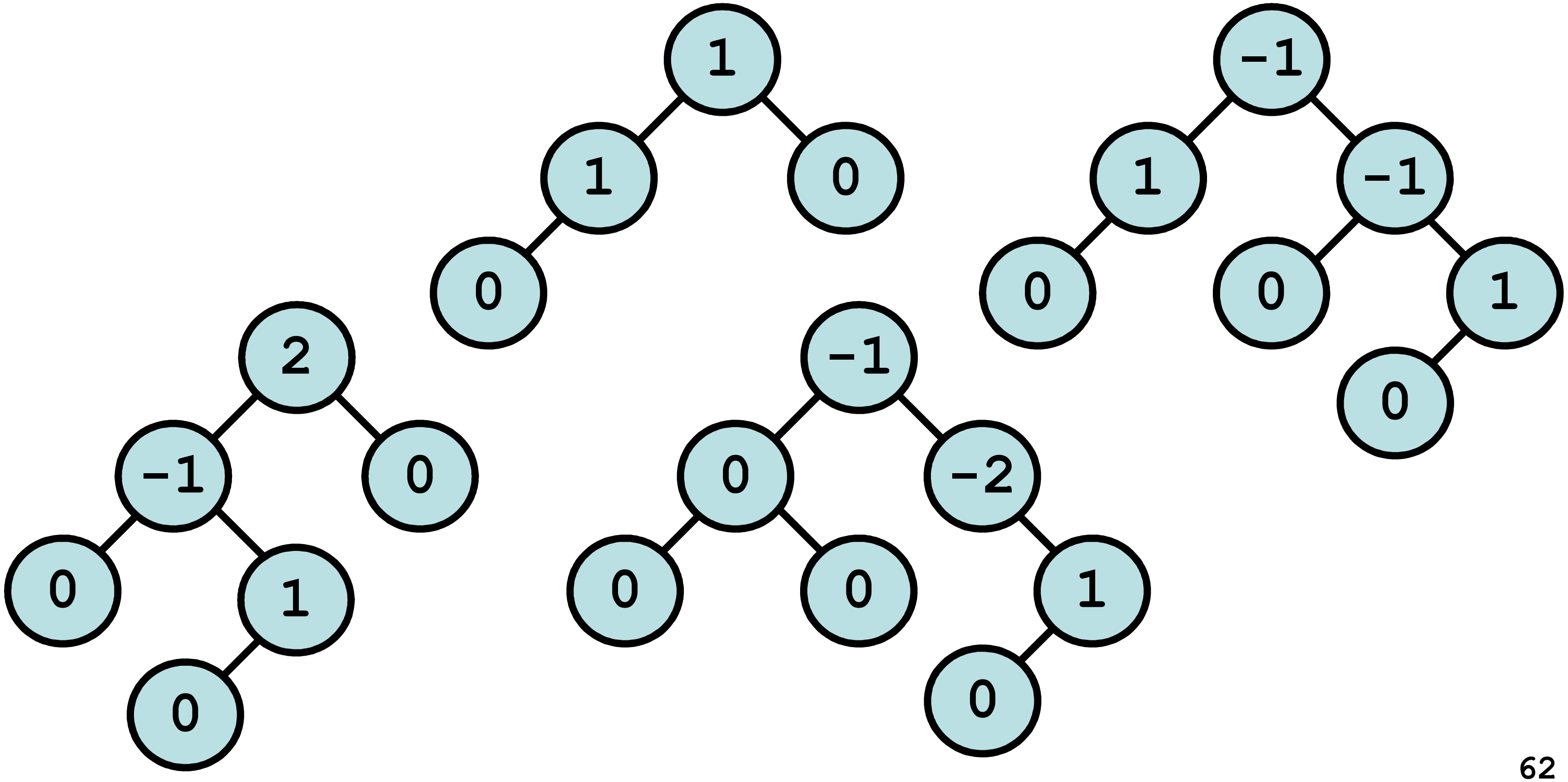
使得查找复杂度可以达到$O(\log_2 n)$

平衡二叉树

- 平衡二叉树
  - 也叫**AVL**树
    - **Adelson-Velskii** 和 **Landis** 发明
  - 或者是空树
  - 或者：
    - 左右子树都是**AVL**树
    - 且左右子树的深度之差不超过**1**

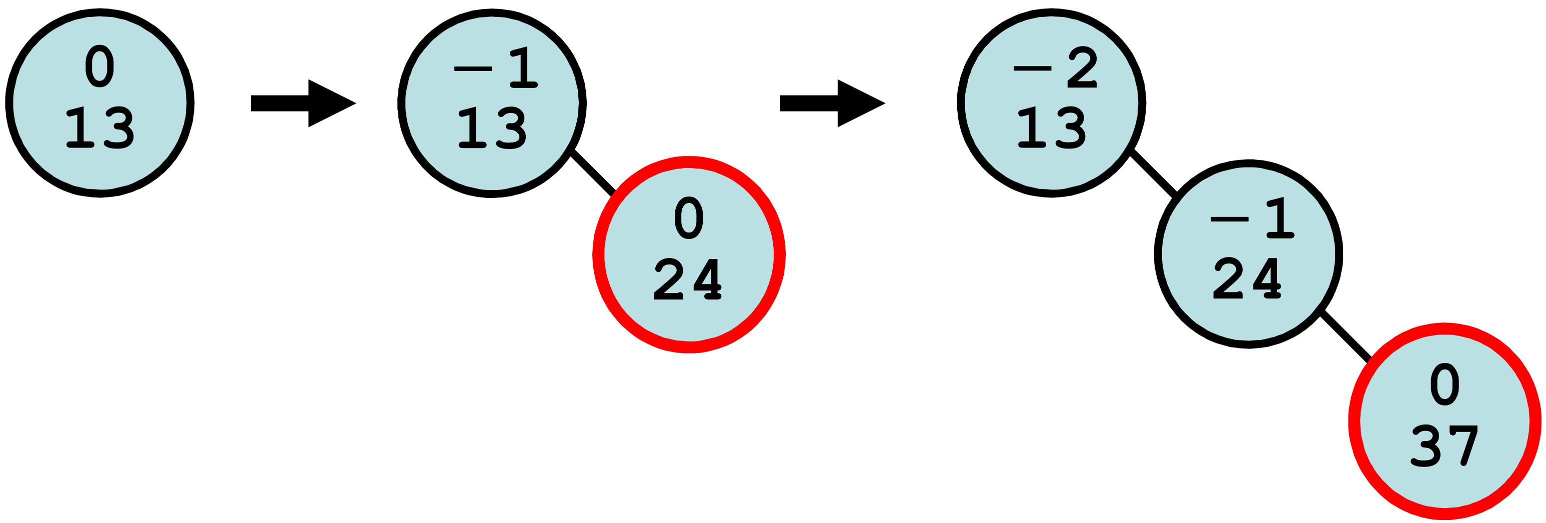
平衡二叉树

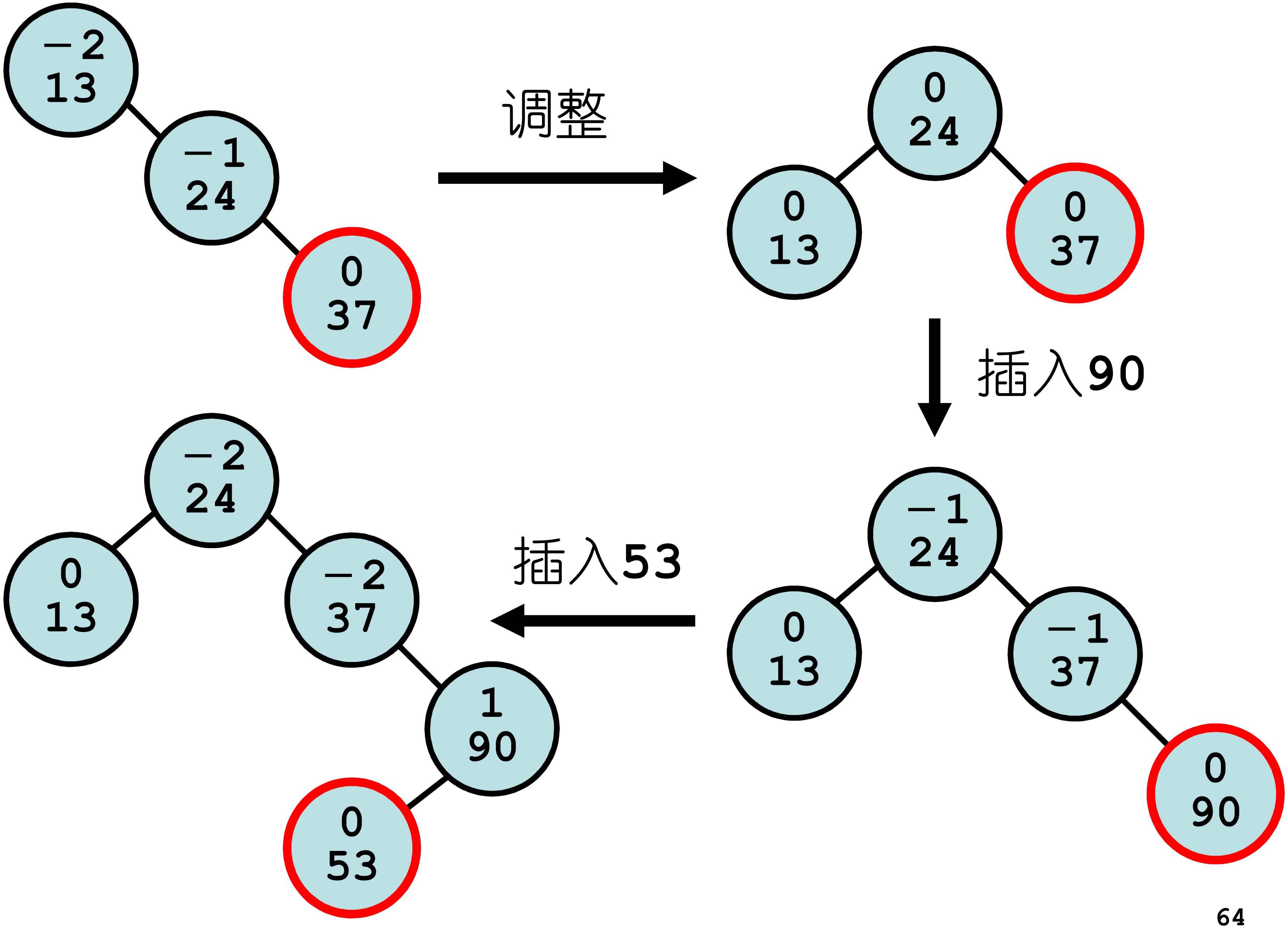
- 平衡因子 **=** 左子树的深度 **-** 右子树的深度
  - **AVL**树上任何结点的平衡因子都 **<= 1**

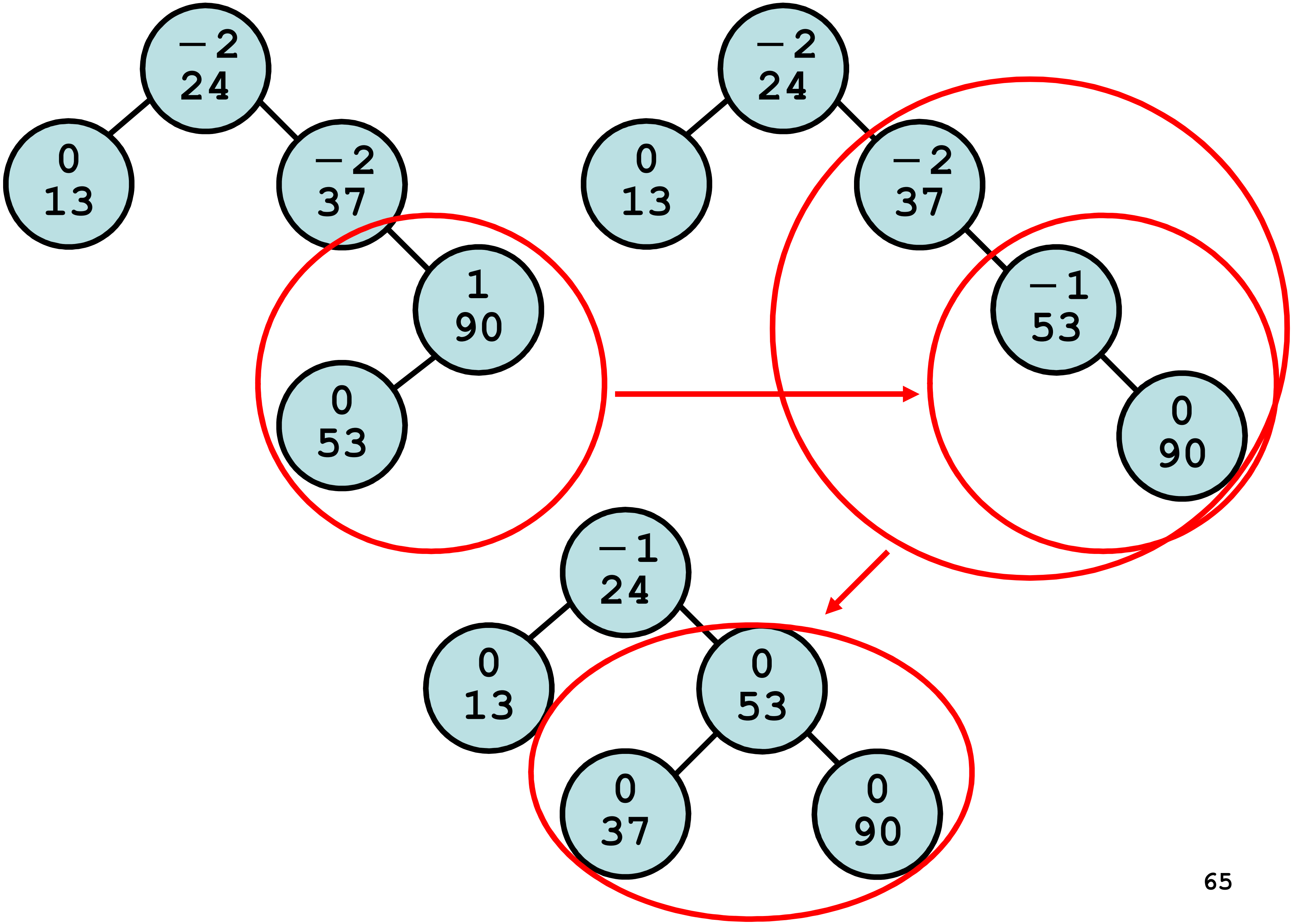


平衡二叉树

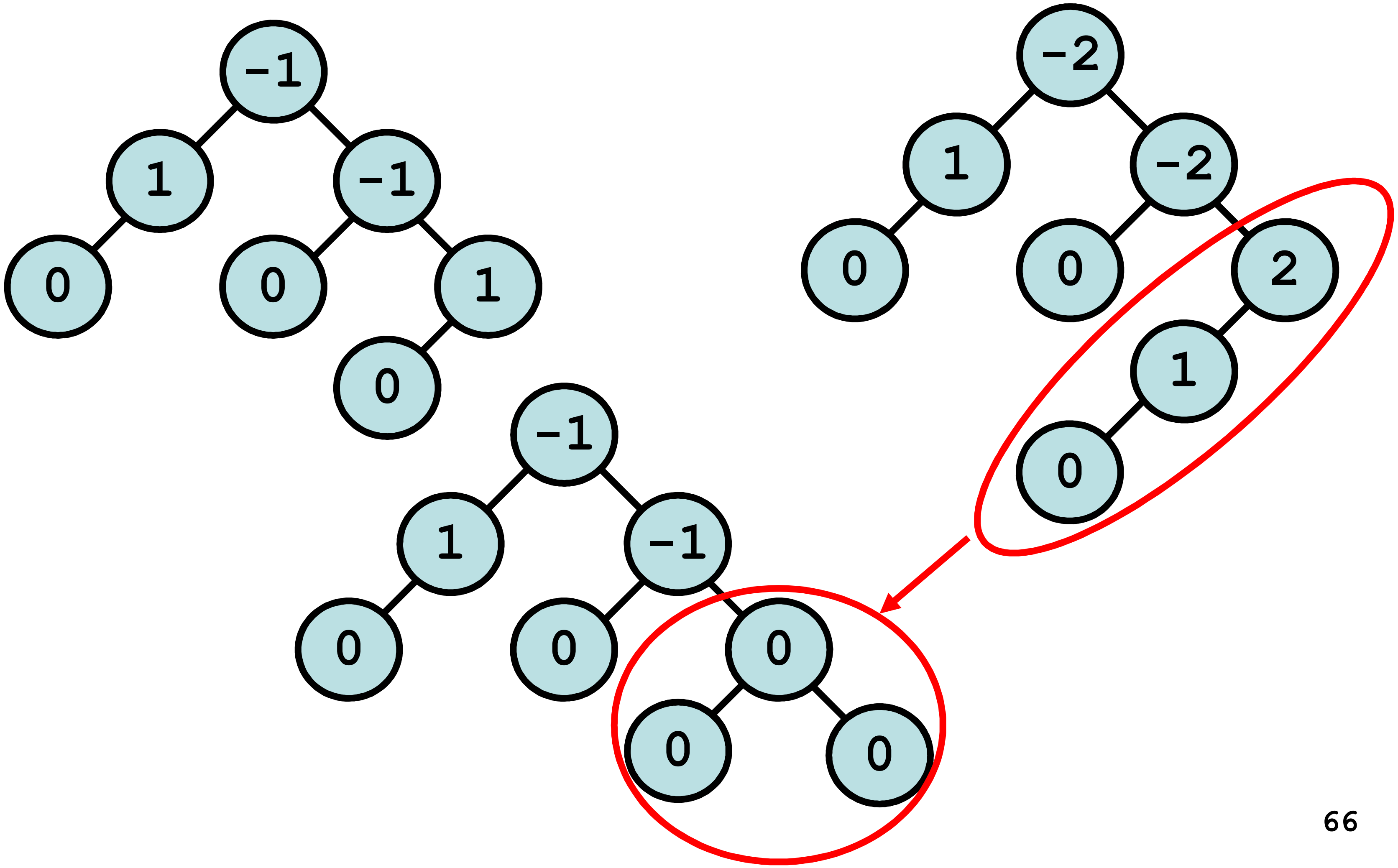
- 平衡二叉树的构造
  - 初始为空树
  - 不断插入结点
  - 如果导致了不平衡，调整之




−2
13
−1
24
0
37
调整
0
24
0
13
0
37
插入90
−1
24
0
13
−1
37
0
90
插入53
−2
24
0
13
−2
37
1
90
0
53

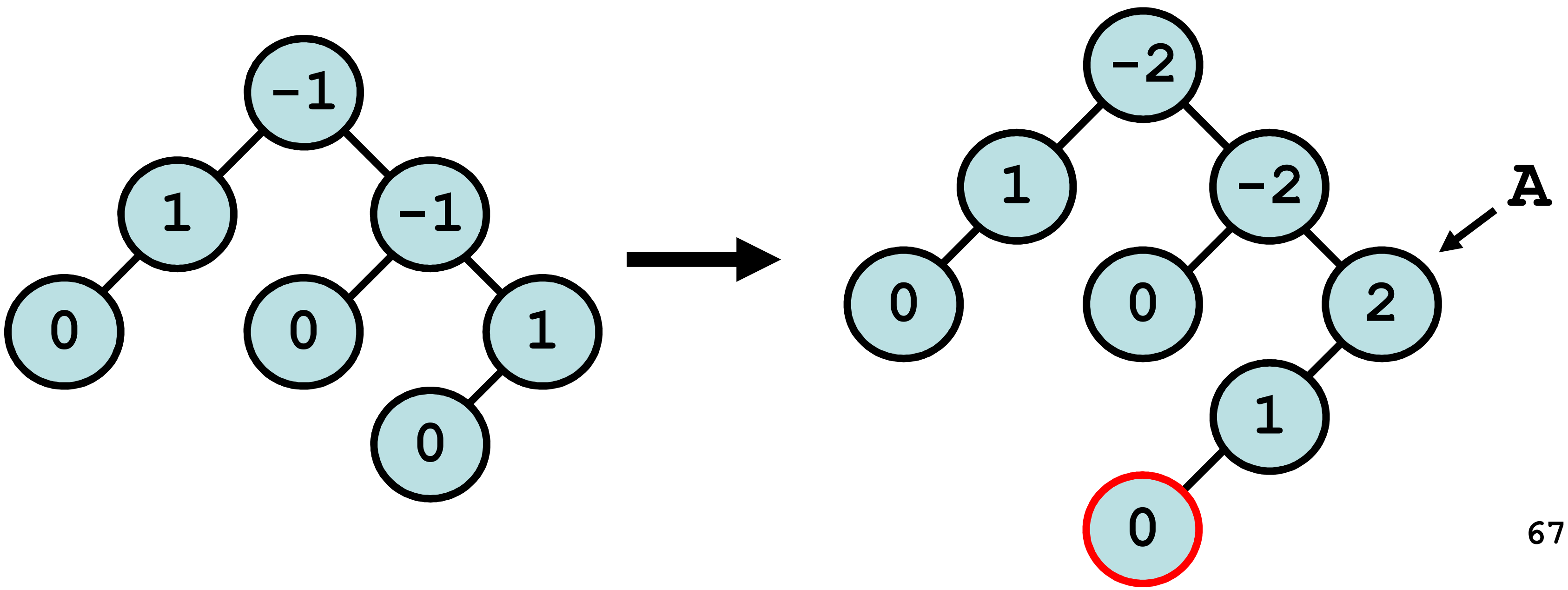

−2
24
0
13
−2
37
1
90
0
53
−2
24
0
13
−2
37
−1
53
0
90
−1
24
0
13
0
53
0
37
0
90

- 局部影响全局
  - 不平衡是从局部开始的
  - 调整也就只需要从局部着手



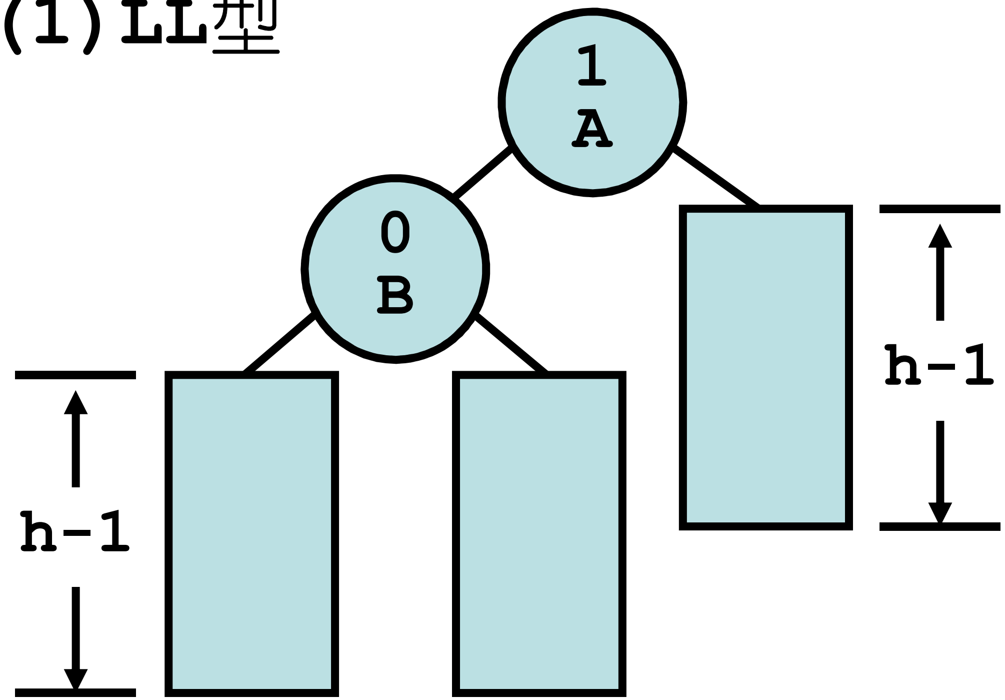
平衡二叉树

- 分析
  - 插入一个结点，树根的平衡因子最多±**1**，所以只需要消除这一层的不平衡即可
  - 设失去平衡的最低的子树根为**A**，“从局部着手”，只需要对以**A**为子树根的子树进行调整



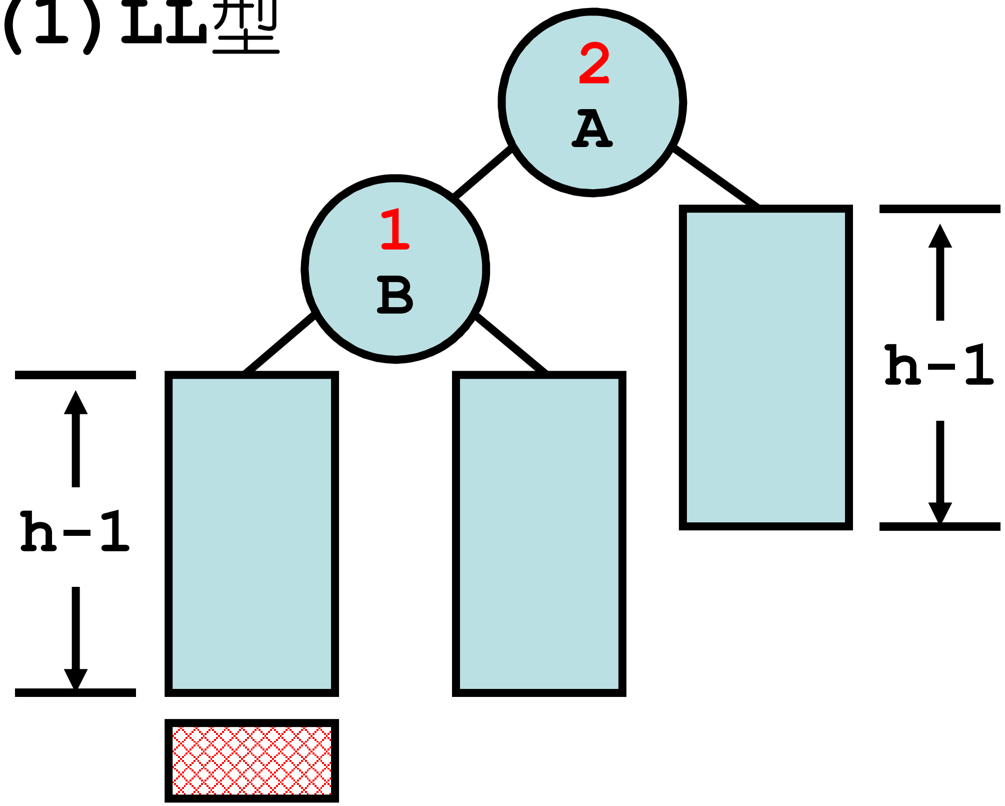
平衡二叉树

- 分情况讨论
  - 以**A**为子树根的子树就这**4**种情况：
    - **(1)LL**型



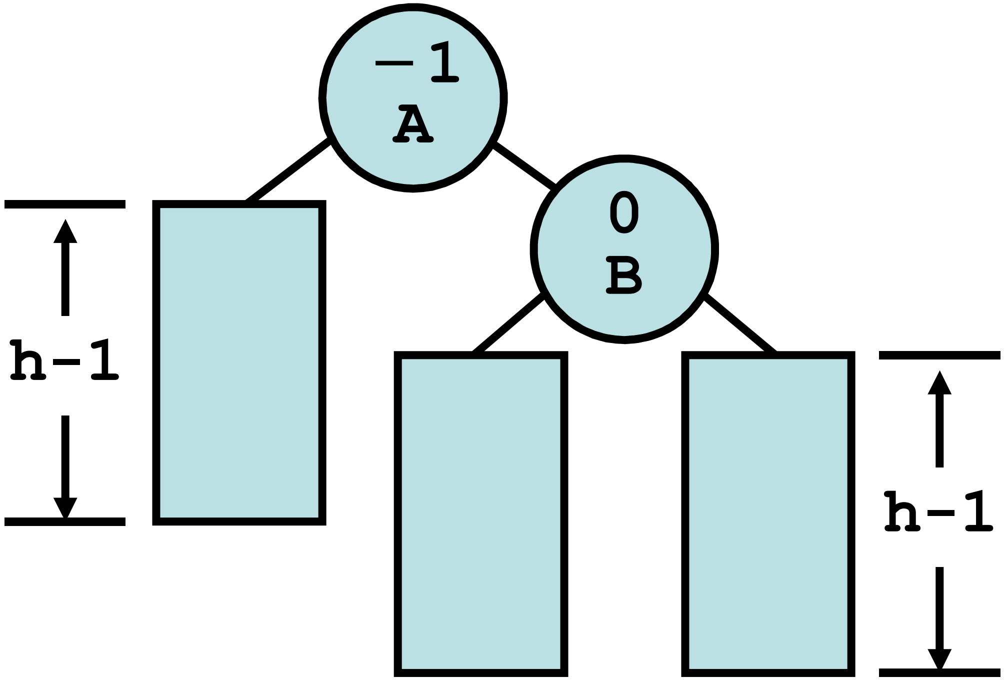
平衡二叉树

- 分情况讨论
  - 以A为子树根的子树就这4种情况：
    - (1) LL型



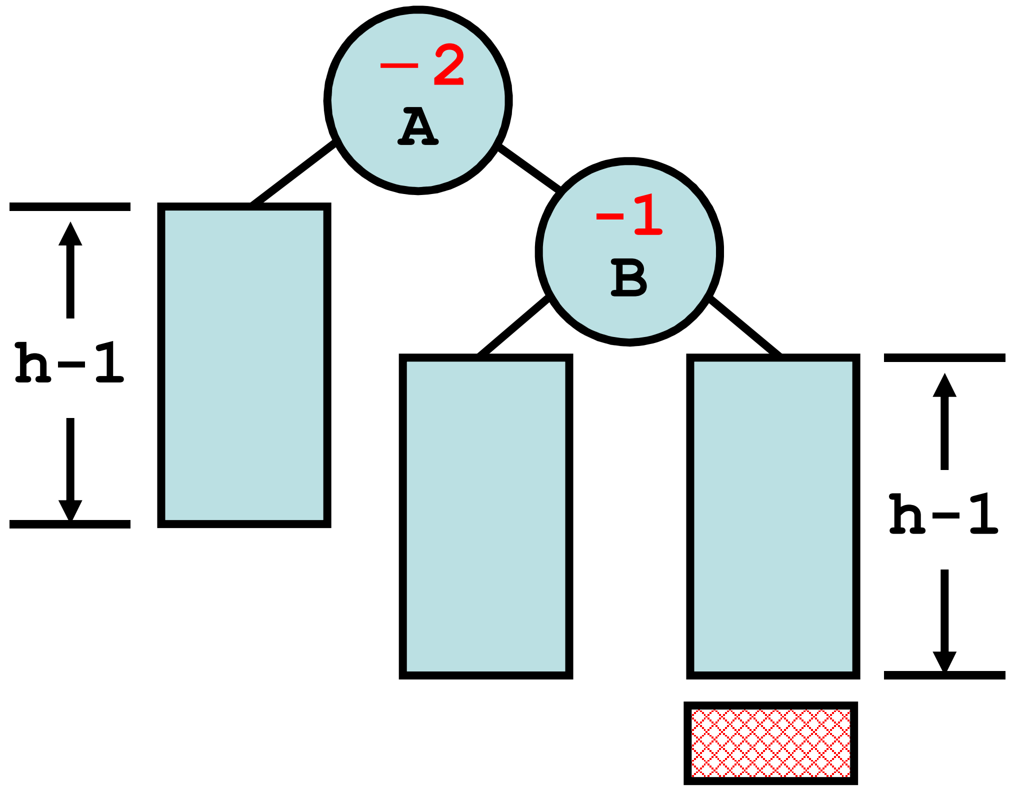
平衡二叉树

- (2) RR型



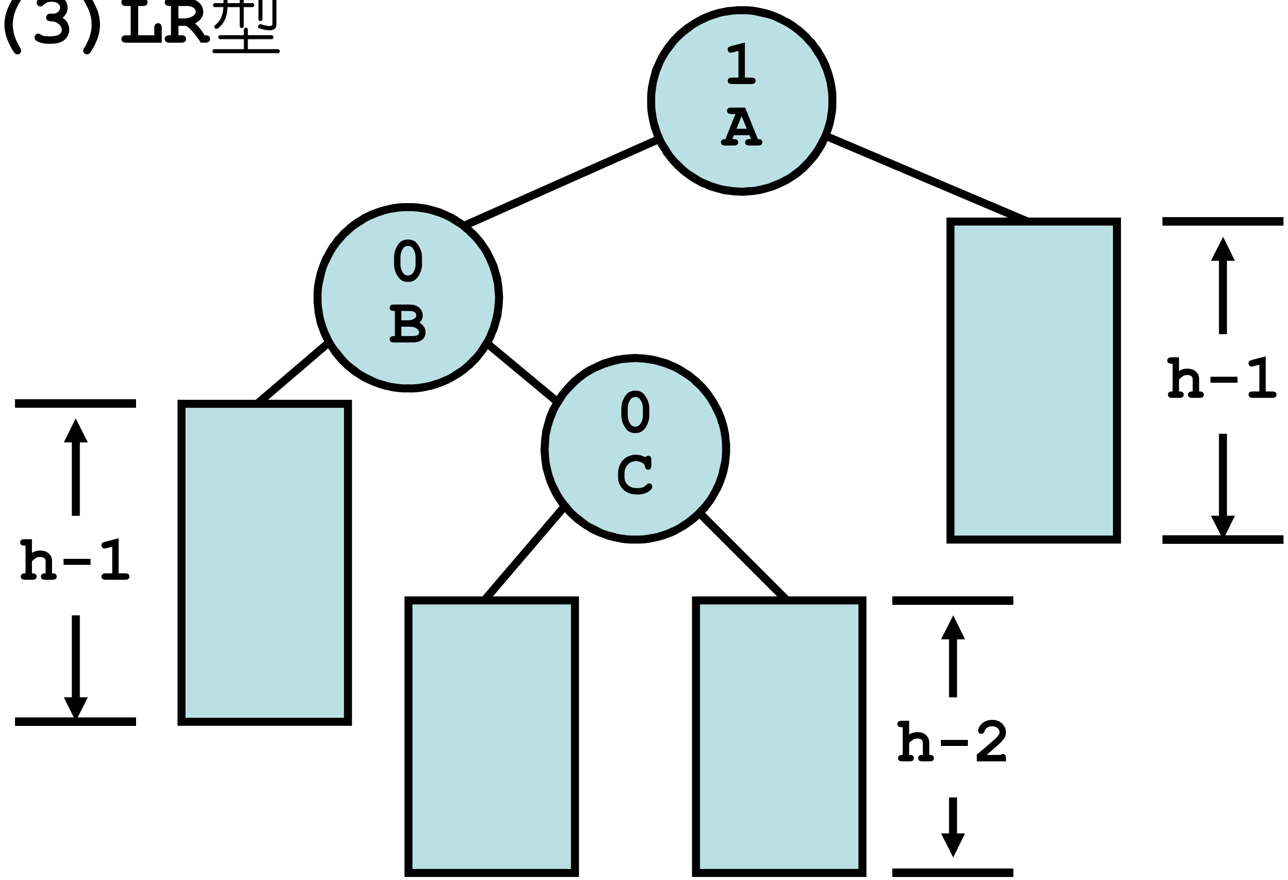
平衡二叉树

- (2)RR型



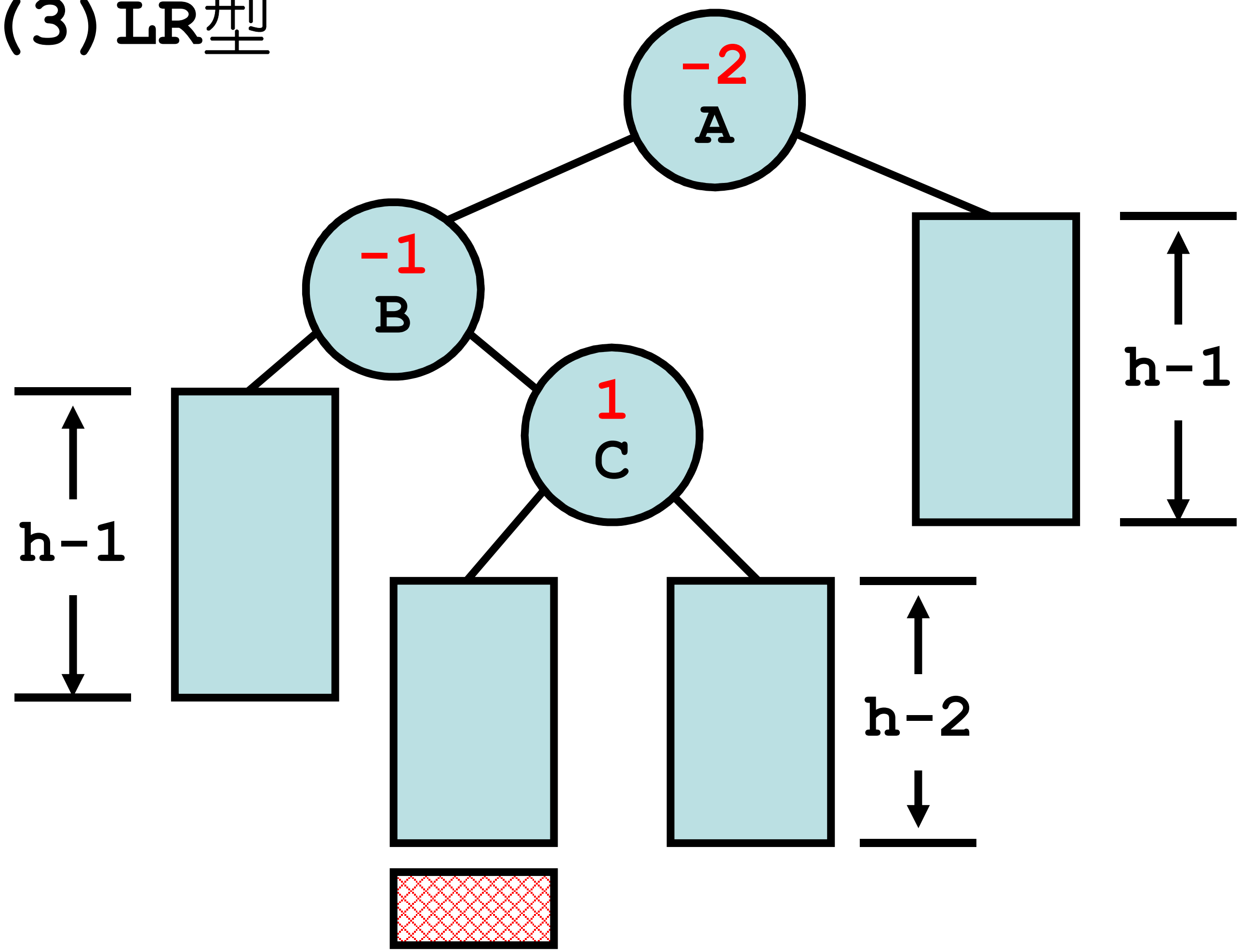
平衡二叉树

- (3) LR型



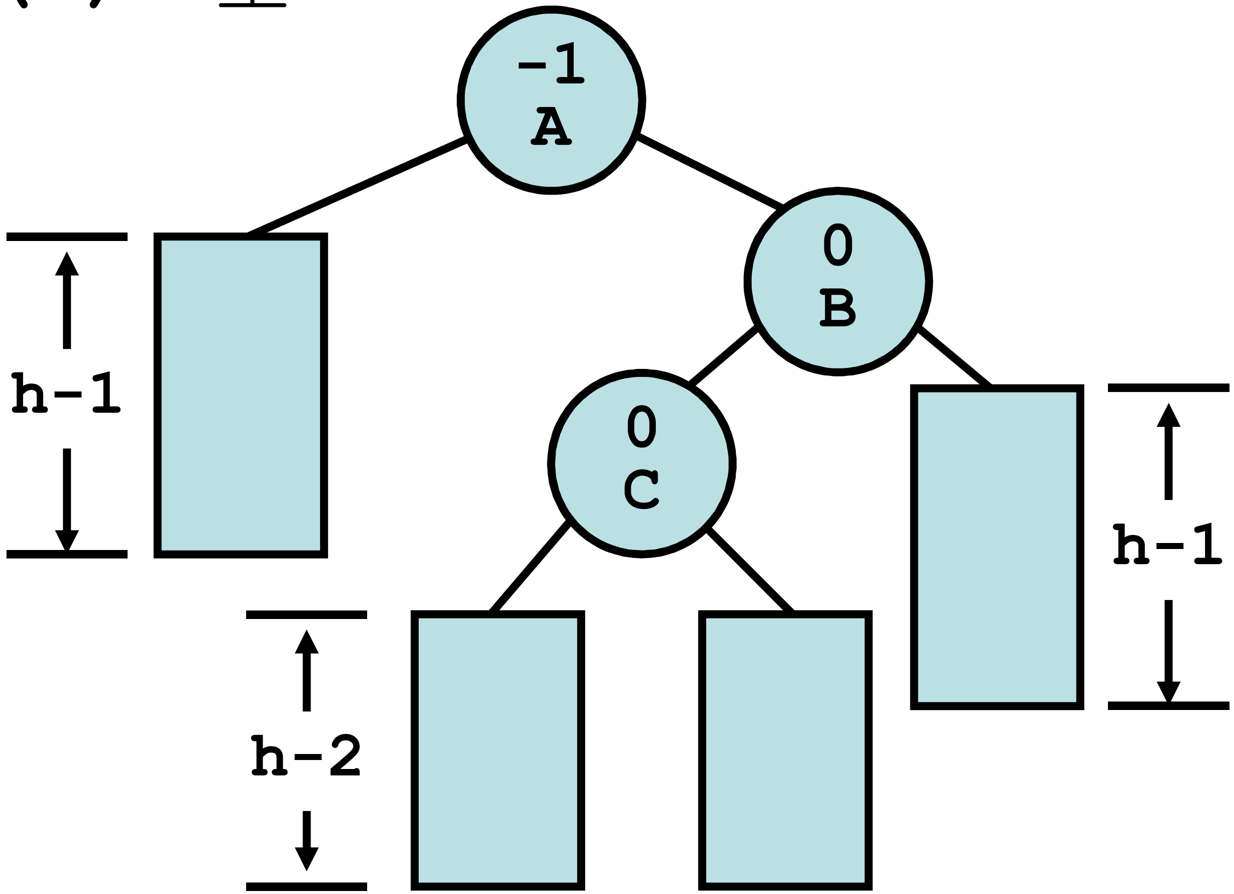
平衡二叉树

- (3) LR型



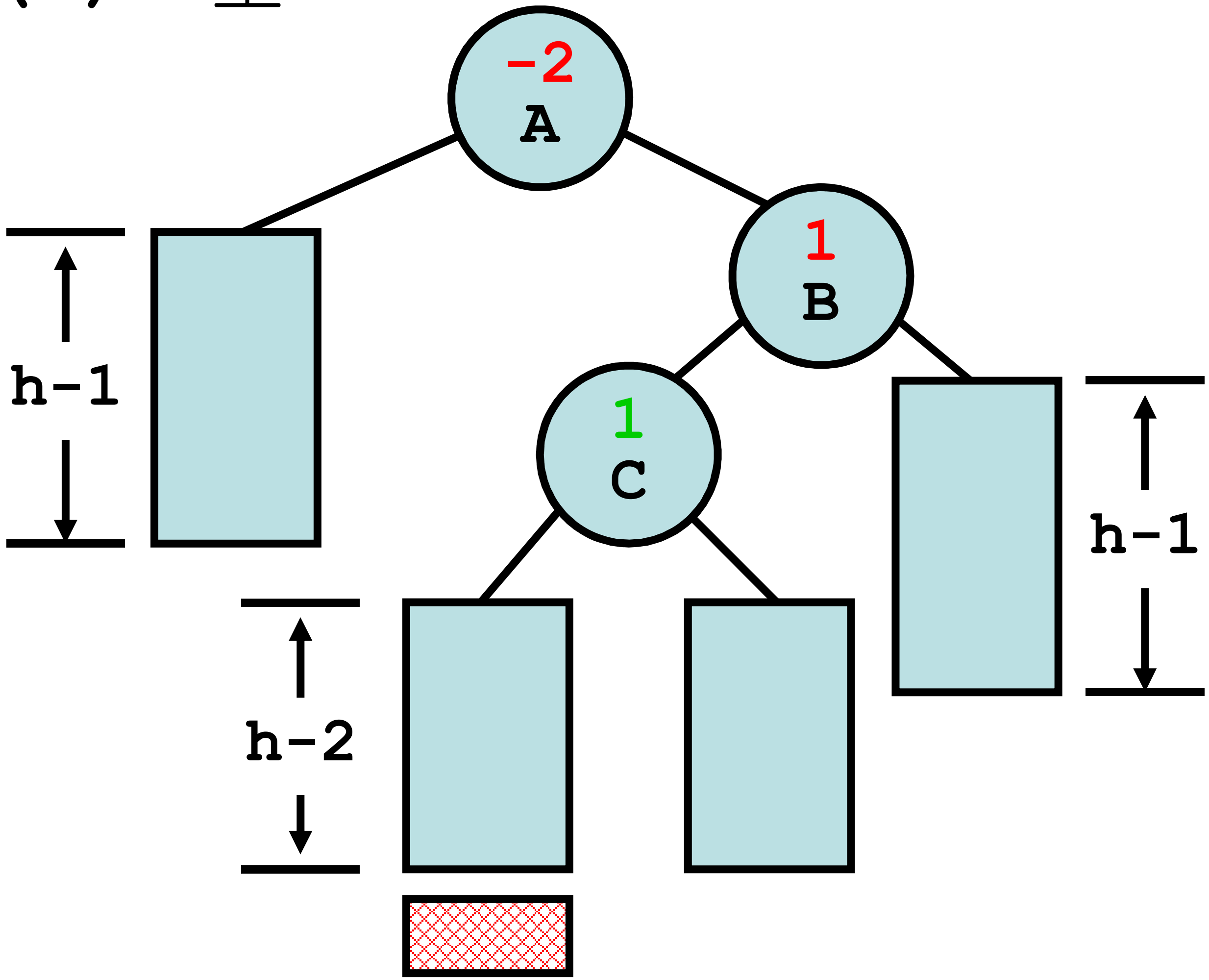
平衡二叉树

- (4) RL型



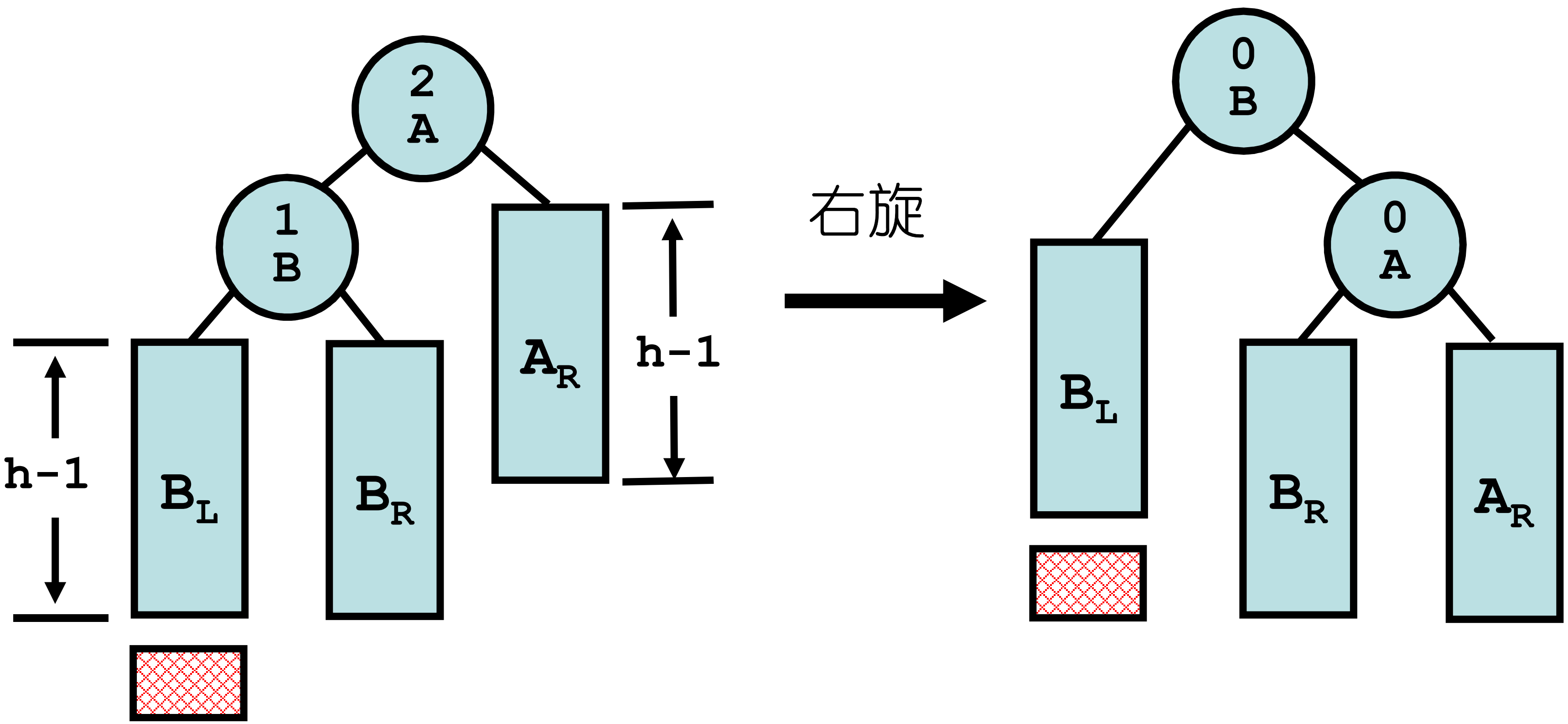
平衡二叉树

- (4) RL型



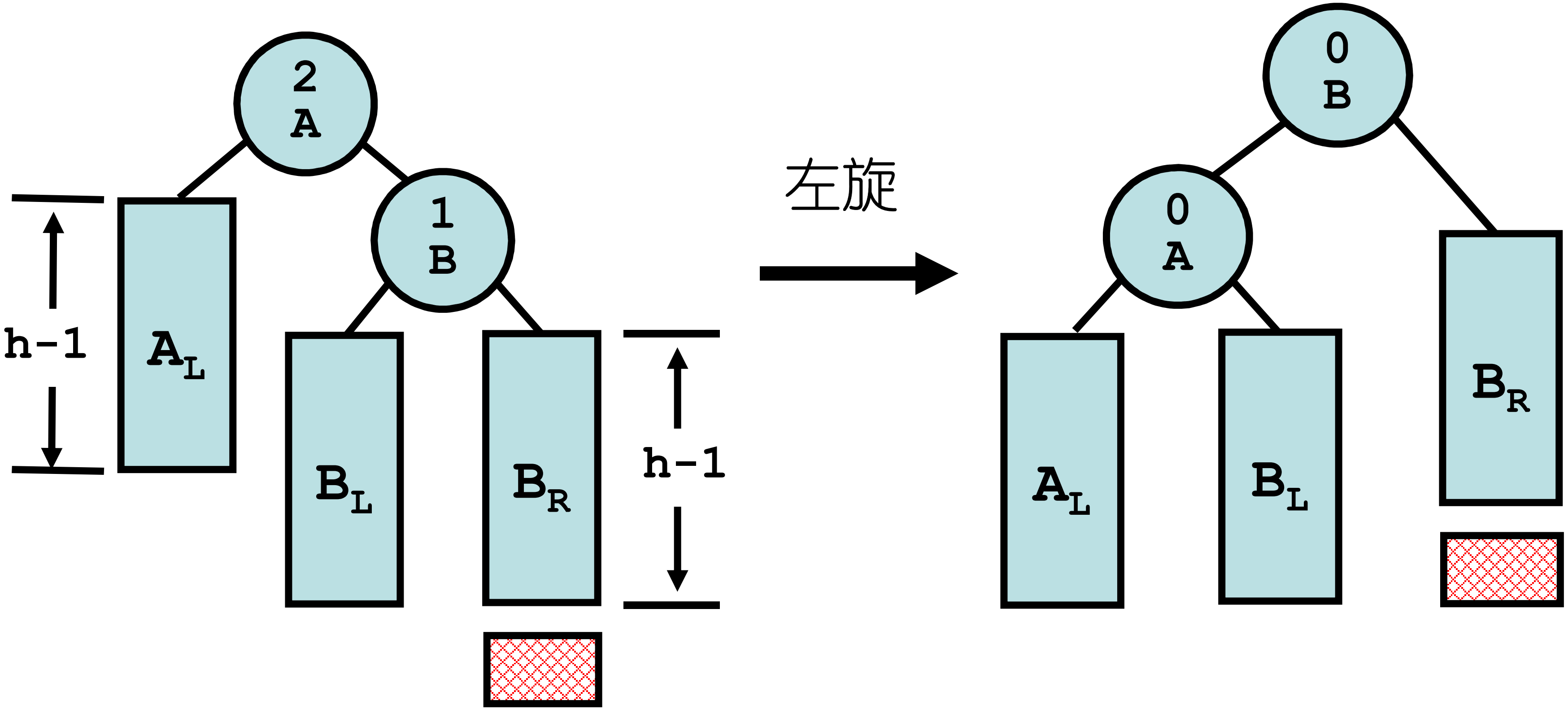
平衡二叉树

- 各个击破
  - (1) LL型



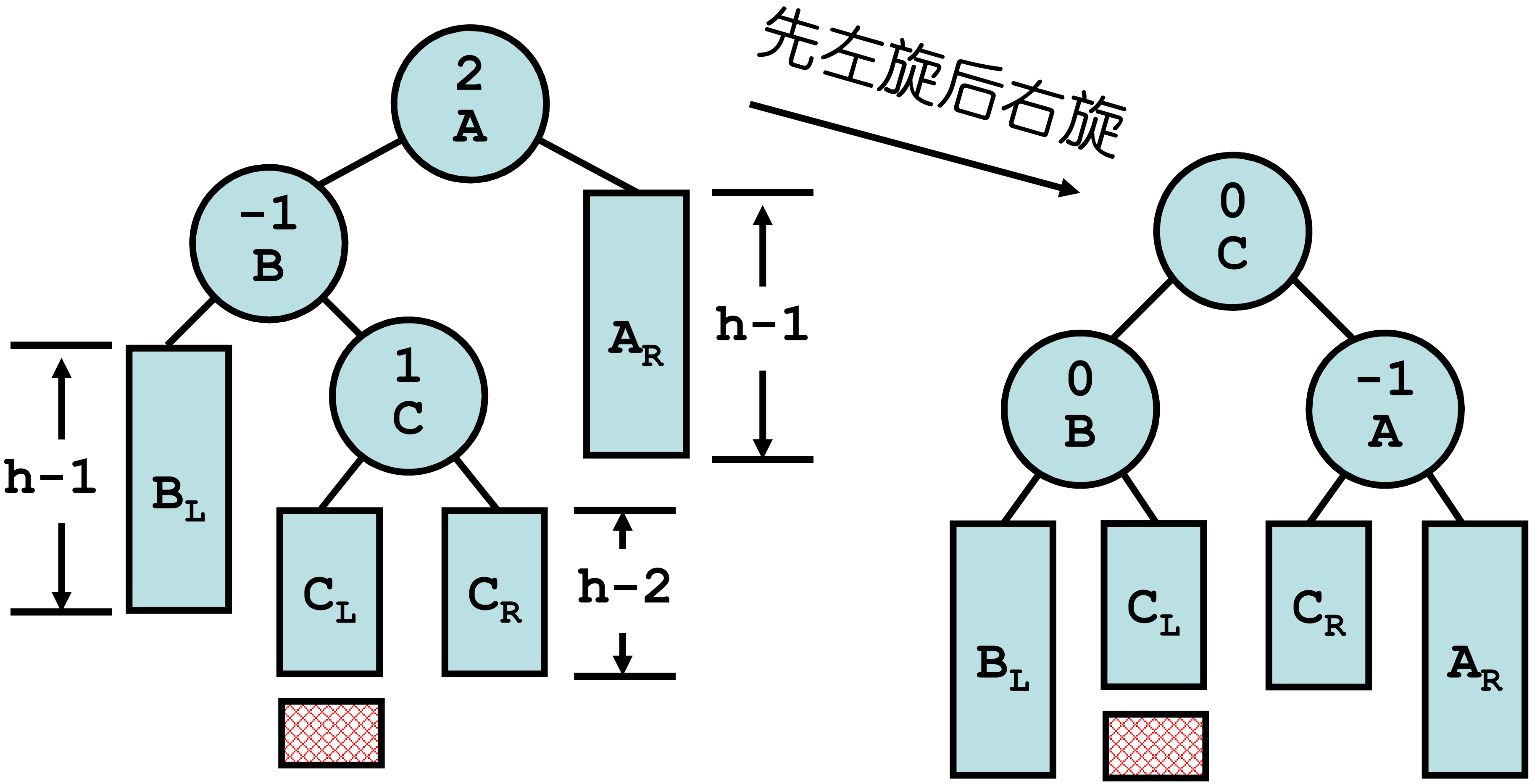
平衡二叉树

– (2) RR型



平衡二叉树

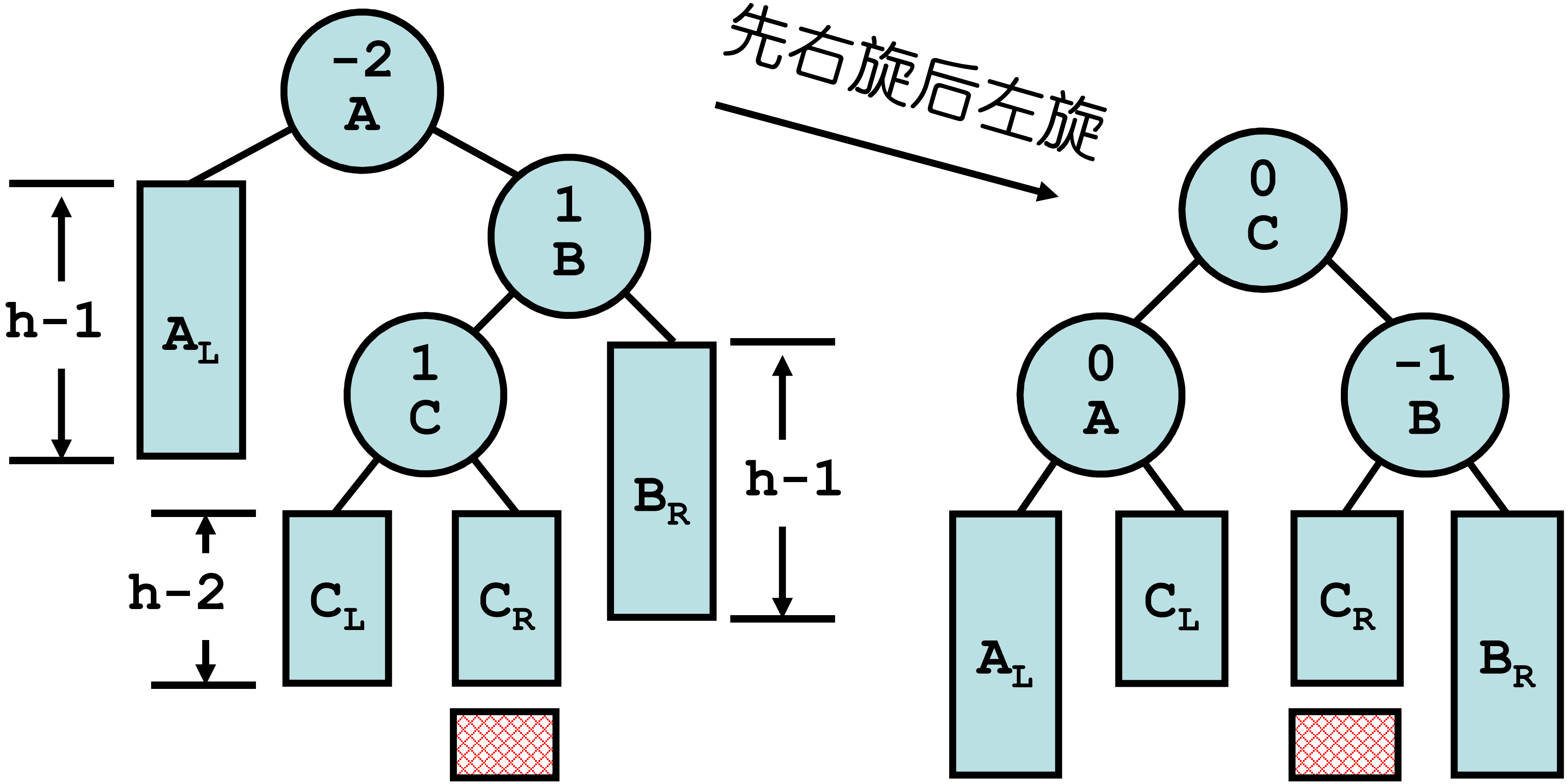
– (3) LR型



先左旋后右旋
2
A
-1
B
1
C
B_L
h-1
C_L
C_R
h-2
A_R
h-1
0
C
0
B
-1
A
B_L
C_L
C_R
A_R

平衡二叉树

– (4) RL型



哈希查找

- 查找算法的效率
  - 主要取决于比较的次数
  - 因此我们希望尽量减少比较的次数
- 哈希查找 (**Hash**)
  - 一种可以最少只比较一次就找到目标的查找方法
  - 又称为散列查找、杂凑查找

哈希查找

- 例
  - 江南大学学生的学号有10位，比如：

$$\underbrace{0\ 3\ 4\ 4}_{\text{院系编号}}\ \underbrace{0\ 4\ 0\ 1}_{\text{班级编号}}\ \underbrace{0\ 1}_{\text{序号}}$$

  - 一共有0~9999999999，10亿个学号
  - 而实际上在校生只有18000
  - 所以学生花名册有18000栏就够了

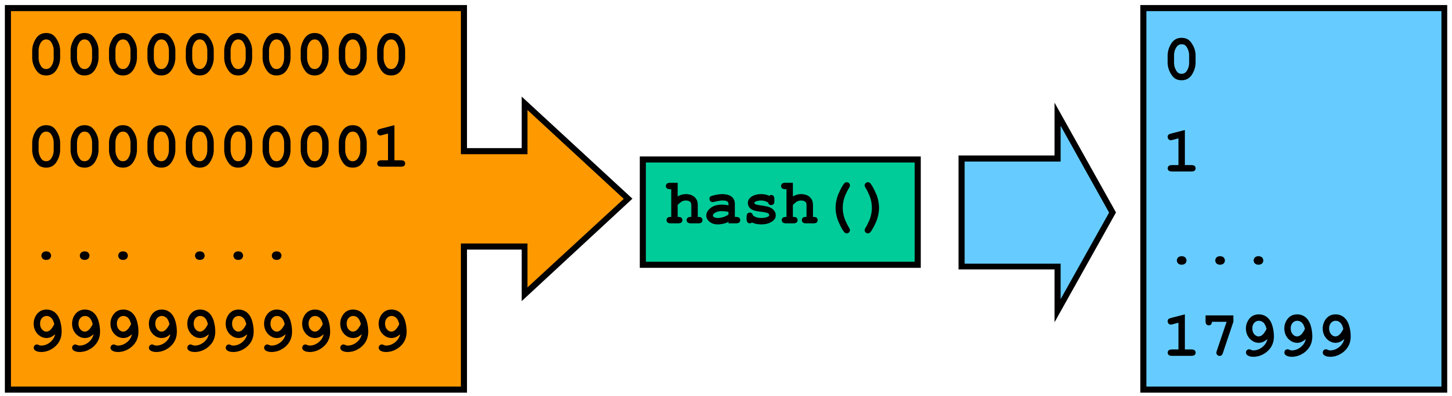
哈希查找

– 设定一个**hash**函数

hash(x) = x % 18000

– 这样**hash**函数值的范围是**0~17999**

– 并且每个关键字映射为唯一的一个函数值



哈希查找

关键字　　Hash函数　　　　　Hash表

0344040101

hash(x)=x%18000

0344030215

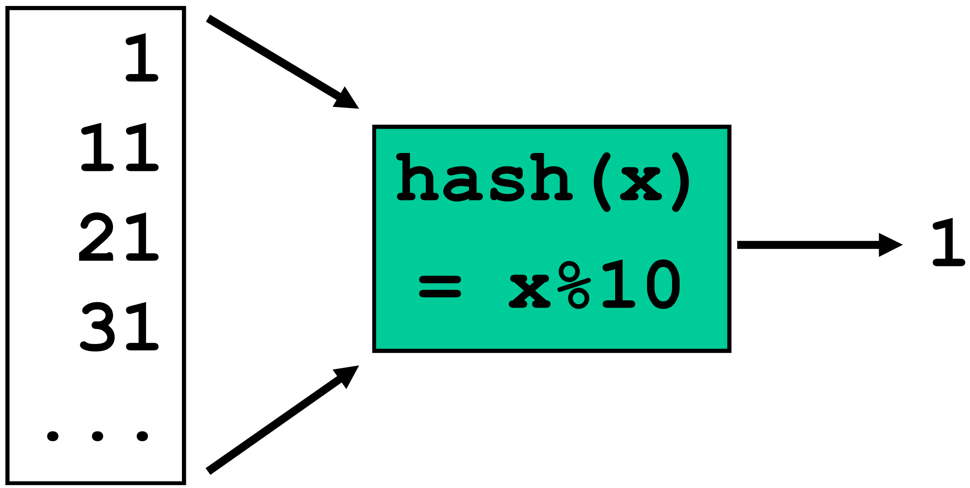
	学号	姓名
0		
1		
…		
6101	0344040101	张三
…		
14215	0344030215	李四
…		
17999		

哈希查找

- **Hash函数**
  - 把关键字映射为存储位置的函数
- **Hash表**
  - 按照此方法构造出来的表或结构
- **Hash查找**
  - 使用**Hash**方法进行查找不必进行多次关键字的比较，搜索速度比较快，可以直接到达或接近具有此关键字的表项的实际存放地址

哈希查找

- 冲突(**Collision**)
  - 散列函数是一个压缩映象函数。关键字集合比**Hash**表地址集合大得多。因此有可能经过**Hash**函数的计算，把不同的关键字映射到同一个**Hash**地址上，这些关键字互为同义词



哈希查找

- **Hash**查找的主要问题
  - 设计**Hash**函数

    计算简单快速、**hash**地址分布比较均匀，避免或尽量减少冲突

  - 研究解决冲突的方案

哈希查找 —— Hash函数

- **Hash函数的设计要求**
  - 定义域 **>=** 全部关键字集合
  - 值域 **<=** 所有的表地址集合

    - 比如学号的范围是**0~9999999999**
    - **Hash**表项的编号是**0~17999**
    - 所以**Hash**函数的定义域应该为**[0,9999999999]**，值域为**[0,17999]**

哈希查找 —— Hash函数

- Hash函数计算出来的地址应能均匀分布在整个地址空间中
  - 若 key 是从关键字集合中随机抽取的一个关键字，Hash函数应能以同等概率取到值域中的每一个值
  - 反之，如果关键字总是更容易映射到某个或某些地址上，称作堆积

哈希查找 —— 常用的Hash函数

- 直接定址法
  - Hash函数取关键字的某个线性函数

```
Hash( key ) = a * key + b
```

  - 一一映射，不会产生冲突
  - 但是，它要求Hash地址空间的大小与关键字集合的大小相同

哈希查找 —— Hash函数

- 余数法
  - 设Hash表中允许地址数为m

```
hash( key ) = key % p
```

  - 对p的要求
    - p <= m，尽量接近m
    - 最好取质数
    - 最好不要接近2的幂

哈希查找 —— Hash函数

- 比如：
  - 有一关键字 key = 962148
  - Hash表大小 m = 25
  - 取质数 p = 23
  - Hash函数：hash(key) = key % p
  - 则Hash地址为：

Hash(962148) = 962148 % 23 = 12

哈希查找 —— Hash函数

- 数值抽取法（数字分析法）
  - 将关键字的某几位数字取出作为地址
  - 比如
    - 关键字为6位数，Hash表地址为3位数
    - 可以取出关键字的第1、2、5位，组成Hash地址
    - 136781->138

哈希查找 —— Hash函数

- 数值抽取法仅适用于事先明确知道表中关键字每一位数值的分布情况，它完全依赖于关键字集合。如果换一个关键字集合，选择哪几位要重新决定
  - 比如如果关键字是电话号码、学号，则前几位就不太适合，因为规律性太强

哈希查找 —— Hash函数

- 平方取中法
  - 将关键字的前几位取出，做平方
  - 再取出平方结果的中间几位作为地址
  - 比如：
    - 325483 => 325^2 = 105625 => 056
  - 此方法在词典处理中使用十分广泛

哈希查找 —— Hash函数

- 折叠法
  - 将关键字拆成位数相等的多段，将这几段叠加起来，相加结果作为**Hash**地址
  - 移位折叠法：各段最后一位对齐相加
    - 比如关键字：**123 456 789 12**
    - **Hash**地址要求3位数

```
   123
   456
   789
+   12
------
  1380   =>  380
```

哈希查找 —— Hash函数

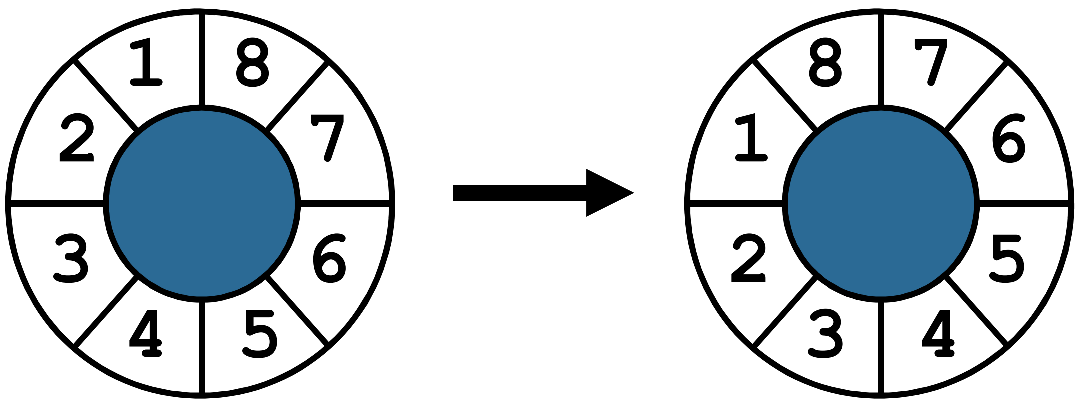
– 边界折叠法：各部分沿各部分的分界来回折叠，然后对齐相加，将相加的结果当做**Hash**地址

- 比如：关键字 = **123 456 789 12**

```
  123
  654
  789
+  21
------
 1587   => 587
```

哈希查找 —— Hash函数

- 旋转法
  - 将关键字中的数字旋转位置
  - 比如：关键字 = **12345678**
  - 把个位数移到最左：**81234567**
  - 此法通常和其它方法结合使用



哈希查找 —— Hash函数

- 伪随机数法
  - 利用伪随机数算法生成**Hash**地址

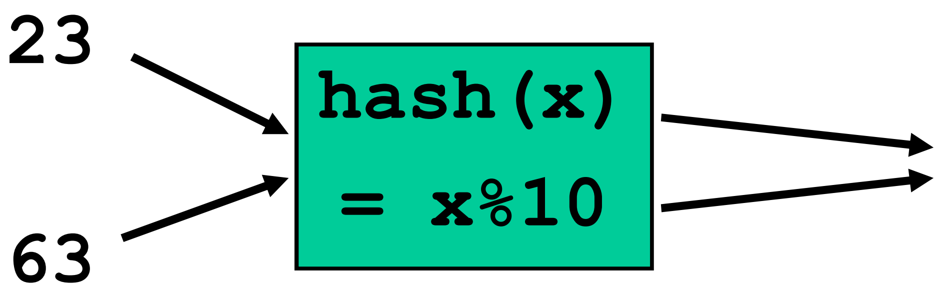
```
Hash(key) = random(key)
```

哈希查找 —— Hash函数

- 总结
  - 应根据实际情况选择：效率要求、关键字长度、Hash表长度、关键字分布情况等
  - 有人曾用统计分析方法对它们进行了模拟分析，结论是中间平方法最“随机”
  - 有时候可以多种方法结合使用，比如
    - `Hash(key) = random(key)%13`

哈希查找 —— 冲突的解决

- 冲突
  - 多个关键字映射到同一个**Hash**表地址



	编号	姓名
0		
1		
2		
3	23	李四
...		
8		
9		

哈希查找 —— 冲突的解决

- 开放定址法
  - 开放定址：如果这个地址冲突，尝试其它地址，即不“封闭”在一个地址上
  - 如果原始地址 $\mathbf{H_0}$ **= H(key)**被占据，在一系列后续地址 $\mathbf{H_i}$ 中去寻找可用空间.

哈希查找 —— 冲突的解决

- 开放定址法
  - “其它”探测地址到底是哪个？
    - 线性探测再散列
    - 二次探测再散列

$$\boxed{H_i = (H(key) + d_i) \% m}$$

    - 再哈希法

$$H_i = Hash_i(key) \quad (i=0,1,\ldots.)$$

哈希查找 —— 冲突的解决

- 开放定址法 之 线性探测
  - 原地址 H_0 发生了冲突
  - 则下一个地址 $H_i = (H_{i-1}+1)\%m$
  - 例如：
    - Hash函数为：H(key) = key % 11
    - 插入关键字key = 60
    - Hash(key) = 60 % 11 = 5，无冲突

0	1	2	3	4	5	6	7	8	9	10
					60					

哈希查找 —— 冲突的解决

- 插入**key = 17**
- **Hash(key) = 17 % 11 = 6**

0	1	2	3	4	5	6	7	8	9	10
					60					

哈希查找 —— 冲突的解决

- 插入**key = 29**
- **Hash(key) = 29 % 11 = 7**

0	1	2	3	4	5	6	7	8	9	10
					60	17				

哈希查找 —— 冲突的解决

- 插入**key = 38**
- **Hash(key) = 38 % 11 = 5**
- 冲突!
- 看看第**6**个单元是否空着？冲突
- 看看第**7**个单元是否空着？冲突
- 看看第**8**个单元是否空着？空闲

0	1	2	3	4	5	6	7	8	9	10
					60	17	29			

哈希查找 —— 冲突的解决

- 插入**key = 51**
- **Hash(key) = 51 % 11 = 8**
- 冲突！ “鸠占雀巢”
- 看看第**9**个单元是否空着？空闲

0	1	2	3	4	5	6	7	8	9	10
					60	17	29	38		

哈希查找 —— 冲突的解决

- 插入**key = 21**
- **Hash(key) = 21 % 11 = 10**

0	1	2	3	4	5	6	7	8	9	10
					60	17	29	38	51	

哈希查找 —— 冲突的解决

- 插入key = 16
- Hash(key) = 16 % 11 = 5
- 冲突!
- 尝试6、7、8、9、10、0

0	1	2	3	4	5	6	7	8	9	10
					60	17	29	38	51	21

哈希查找 —— 冲突的解决

- 线性探索法容易产生“堆积”：
  - 冲突的关键字只好向后寻找可用的空单元
  - 结果又占据了其它关键字的位置
  - 使得冲突更加严重
  - 查找次数增加

哈希查找 —— 冲突的解决

- 开放定址法　之　二次探测
  - 线性探索法容易产生“堆积”：
    - 因为如果当前地址产生了冲突，尝试下一个地址，这样容易造成“连续的一大串”地址冲突，使得查找次数增加
  - 改进方法：
    - 如果当前地址冲突，“下一个”的地址不是紧挨着，而是离远一些，而且冲突次数越多，离得越远

哈希查找 —— 冲突的解决

- 设关键字**key**原本映射为地址$\mathbf{H_0}$

$$\boxed{H_0 = Hash(key)}$$

- 但是$\mathbf{H_0}$发生了冲突
- 则下一次尝试的地址为：

$$H_i = (H_{i-1} \pm i^2) \ \% \ m$$

i为冲突的次数

m为**Hash**表大小

哈希查找 —— 冲突的解决

– 例：
  - 插入**key = 38**
  - **H0 = 38 % 11 = 5**
  - 冲突！
  - **H1 = (H0 + 1^2)%11 = 6**，仍然冲突
  - **H2 = (H1 + 2^2)%11 = 10**

0	1	2	3	4	5	6	7	8	9	10
					60	17	29			

哈希查找 —— 冲突的解决

- 开放定址法　之　再哈希法
  - 若原地址H_0 = $Hash_0(key)$冲突
  - 则下一个地址H_i = $Hash_i(key)$
  - 即换一个哈希函数试试

哈希查找 —— 冲突的解决

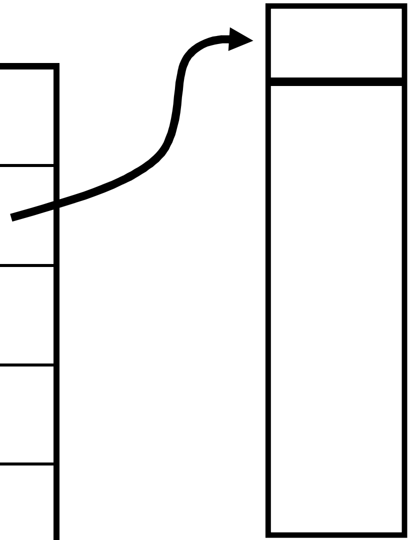
- 桶法

冲突的元素存储在同一地址的桶中.

仍然会产生冲突！
- 开放定址法
- 溢出桶

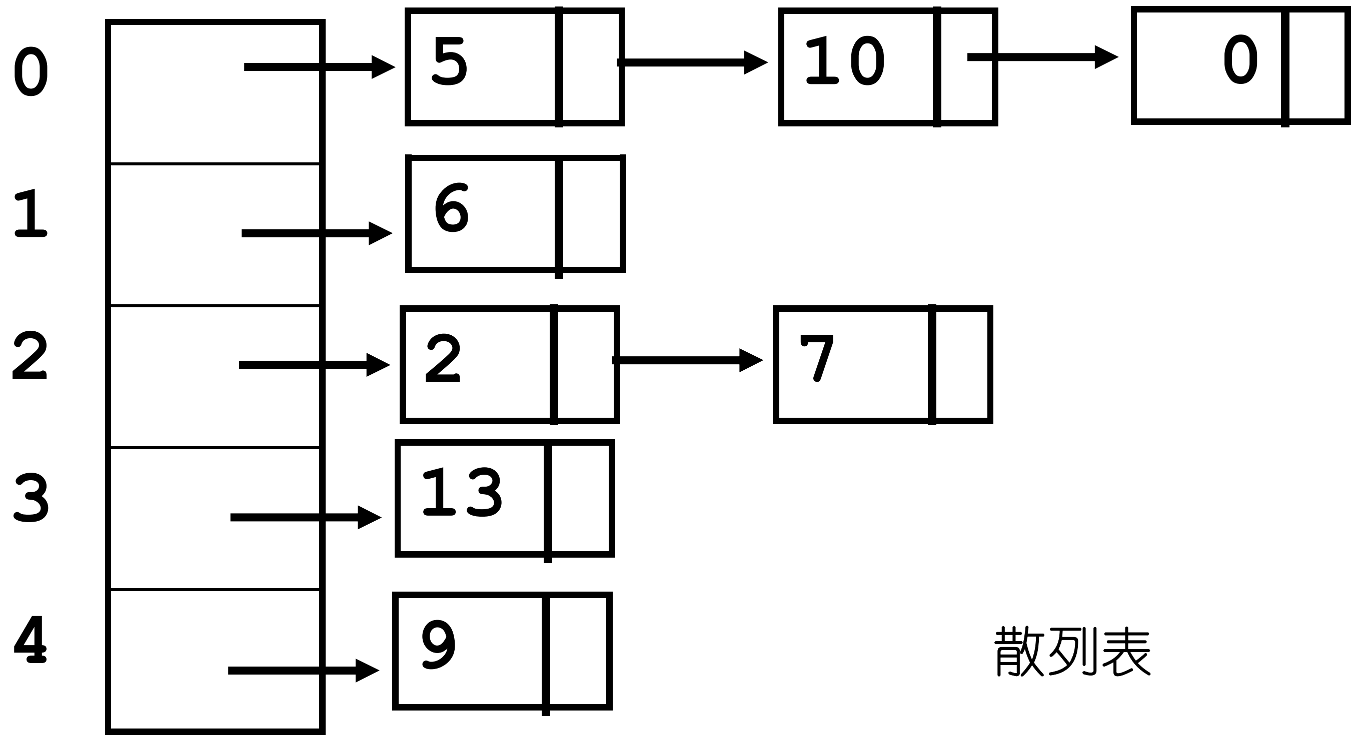
0		
1	A1	B1
2	A2	C2
3		
4	A4	B4
5	A5	

0	A0		
1	A1	B1	
2			
3			
4			



哈希查找 —— 冲突的解决

- 链表法
  - 映射到同一地址的数据存放在链表中
  - 如`Hash(key) = key % 5`



散列表

哈希查找 —— 查表

- 查表
  - 给定关键字**key**
  - 运算**Hash**函数，得到**Hash**地址
  - 若该地址存放的不是该关键字，说明原来出现了冲突，按照冲突解决方法查找下一个可能的地址

哈希查找 —— 查表

- 例
  - **Hash**函数 **H(key) = key % 13**
  - 冲突解决方法：线性探测
  - 查找**84**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84	79	23	11	10			

哈希查找 —— 查表

- 例
  - **Hash**函数 **H(key) = key % 13**
  - 冲突解决方法：线性探测
  - 查找**84**
  - **84 % 13 = 6**，但是第**6**个单元不是**84**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84	79	23	11	10			

哈希查找 —— 查表

- 例
  - **Hash**函数 **H(key) = key % 13**
  - 冲突解决方法：线性探测
  - 查找**84**
  - **84 % 13 = 6**，但是第**6**个单元不是**84**
  - 看看第**7**个单元

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84	79	23	11	10			

哈希查找 —— 查表

- 例
  - **Hash**函数 **H(key) = key % 13**
  - 冲突解决方法：线性探测
  - 查找**84**
  - **84 % 13 = 6**，但是第**6**个单元不是**84**
  - 看看第**7**个单元
  - 看看第**8**个单元

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84	79	23	11	10			

本章小结

- 查找的概念
- 静态查找表
  - 线性查找
  - 折半查找
  - 分块查找
- 动态查找表
  - 二叉查找树、平衡二叉树
  - 哈希查找

作业和思考

- 思考题
  - 习题集P57：
    - 9.19
    - 9.31
    - 9.33