

9 查找

董洪伟

<http://hwdong.com>

主要内容

- 查找的概念
- 线性查找表
 - 线性查找
 - 折半查找
 - 索引查找（分块查找）
- 二叉查找树（平衡二叉树）
 - 二叉查找树
 - 平衡二叉树
- 哈希查找（散列查找）

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

- 关键字

- 相当于排序中的排序码
- 数据元素中某个数据项的值
- 可以标识一个数据元素
- 关键字可以相同，即不一定唯一标识这个元素

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平均比较的次数称作平均查找长度

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

- P_i ：查找第*i*个数据元素的概率

- C_i ：查找该元素的过程中比较的次数

查找的概念

- 线性查找表

- 数据集合是一个线性表

- 二叉查找树

- 数据集合是一个二叉树

- 哈希(散列)表

- 根据哈希函数(关键字映射到存储地址)存储和查找。有点类似根据下标到地址。

线性查找


- 线性查找

- 又称为顺序查找
- 主要用于在线性的数据结构中进行查找

- 基本思想

- 设若表中有 N 个对象，则从表的一端开始，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



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

- 欲查找57


线性查找



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

- 欲查找57


线性查找



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

- 欲查找57


线性查找



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

- 欲查找57


线性查找



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

- 欲查找57
- 找到，位置在第4个单元


线性查找



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

- 欲查找27


线性查找



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

- 欲查找27


线性查找



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

- 欲查找27


线性查找



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

- 欲查找27

线性查找



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

- 欲查找27

线性查找

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

- 欲查找27

线性查找

0	1	2	3	4	5	i ↓
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

Diagram illustrating a binary search process on an array. The array contains 12 elements, indexed 0 to 11. The current search range is from index 3 (low) to index 4 (high). The middle element (mid) is at index 3, which contains the value 25. Red arrows indicate the current search range and the middle element.

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

Diagram illustrating a binary search process on an array. The array contains 12 elements, indexed 0 to 11. The current search range is from index 3 to index 4. The middle element (mid) is at index 3.

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：算法

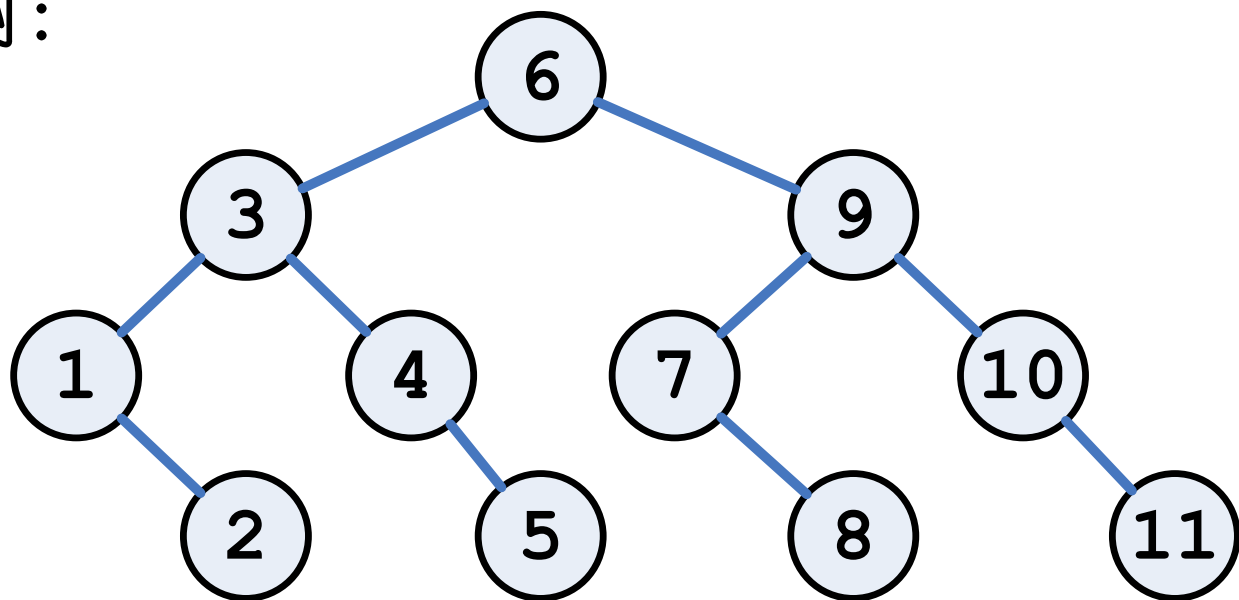
```
typedef struct{
    ElemType data;
    int length, listsize;
} SqList;
```

[illegible]

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
 - 判定树:



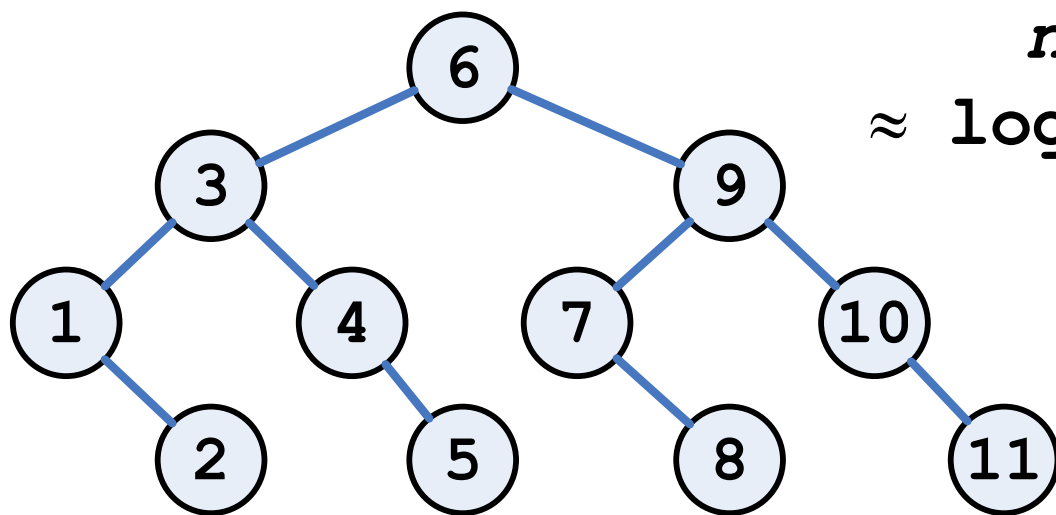
折半查找

• 平均查找长度

– 第 h 层的元素有 2^{h-1} 个，找到它需要比较 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
起始地址

22	49	86
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:

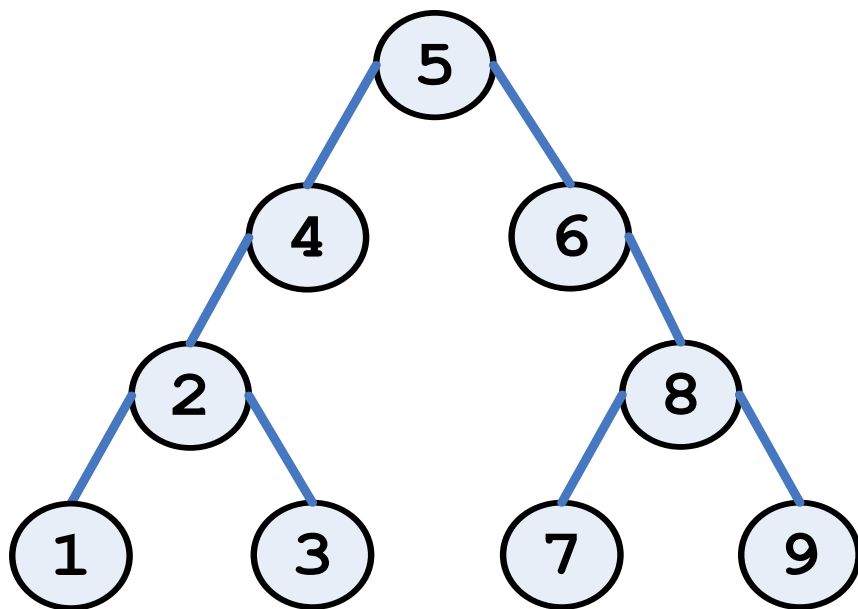
$$ASL_{bs} = ASL_b + ASL_w$$

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: $ASL_{bs} = (b+1)/2 + (s+1)/2$
 $= (n/s + s)/2 + 1$

二叉查找树

- 二叉查找树

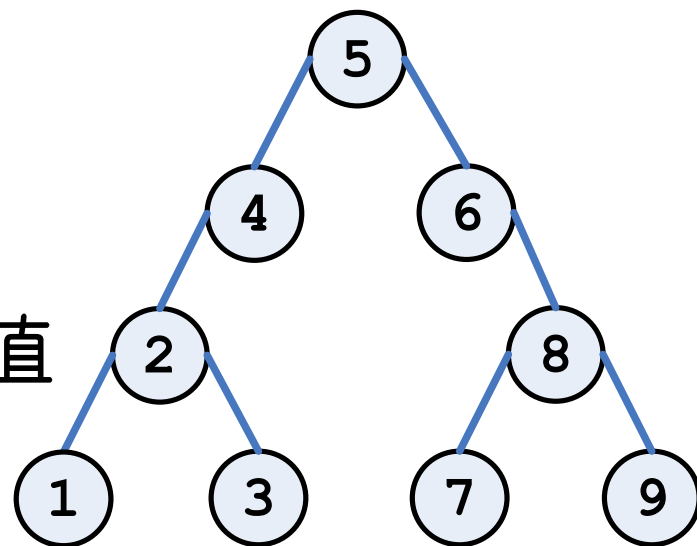
- 又称二叉排序树
- 是一棵二叉树，不过
- 左子树节点的值 < 根节点的值 < 右子树节点的值



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
 - 若当前节点为空，则未找到
 - 若相等则查找成功
 - 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



二叉查找树

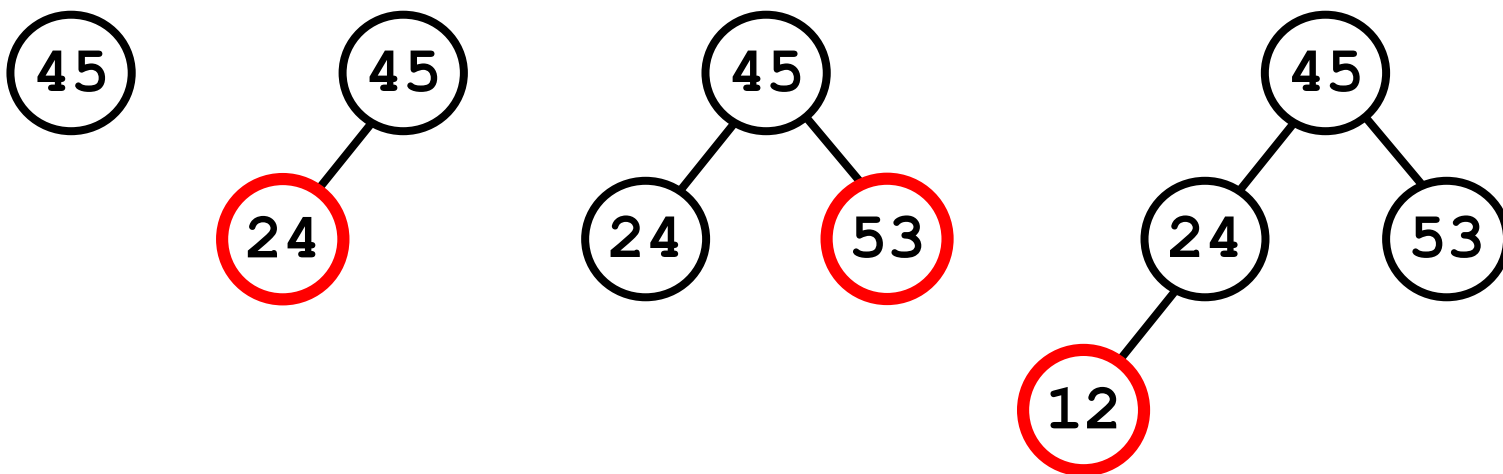
```
typedef struct binode{  
    ElemType data;  
    struct binode *lchild,*rchild;  
} BiTNode;
```

//二叉查找树的查找算法

```
BiTNode* SearchBST(BiTNode* T, ElemType e){  
    if (!T) return 0; //递归出口  
  
    if (T->data == e) return T; //处理根  
    if (e < T->data) //左子树  
        return SearchBST(T->lchild, e);  
    else //右子树  
        return SearchBST(T->rchild, e);  
}
```

二叉查找树结点的插入

- 如果是空树，新结点作为树根
- 如果等于当前结点，则返回
- 如果小于当前结点，则在其左子树上插入
- 如果大于当前结点，则在其右子树上插入



二叉查找树的插入

```
bool InsertBST(BiTNode* &T, ElemType e){
    if(T==0) {
        T = (BiTNode *)malloc(sizeof(BiTNode));
        if(T){ T->data = e;
                T->lchild = 0;    T->rchild = 0; }
        return true;
    }
    if (T->data == e)    return false;
    else if (e<T->data)    //左子树
        InsertBST(T->lchild, e);
    else if (e>T->data)    //右子树
        InsertBST(T->rchild, e);
    return true;
}
```

```

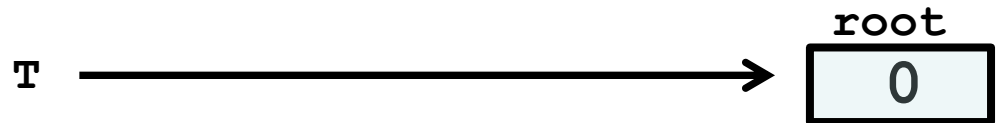
bool InsertBST(BiTNode* &T, ElemType e){
    if(T==0) {
        T = (BiTNode *)malloc(sizeof(BiTNode));
        if(T){ T->data = e;
                T->lchild = 0;    T->rchild = 0; }
        return true;
    }
    //if (T->data == e) return false;
    if (e<T->data)           //左子树
        InsertBST(T->lchild, e);
    else if (e>T->data)       //右子树
        InsertBST(T->rchild, e);
    return true;
}

```

```

void main(){
    BiTNode* root = 0;
    ElemType e;
    e = 30;InsertBST(root, e);
    e = 20;InsertBST(root, e);
    e = 38;InsertBST(root, e);
    e = 22;InsertBST(root, e);
    e = 19;InsertBST(root, e);
}

```



```

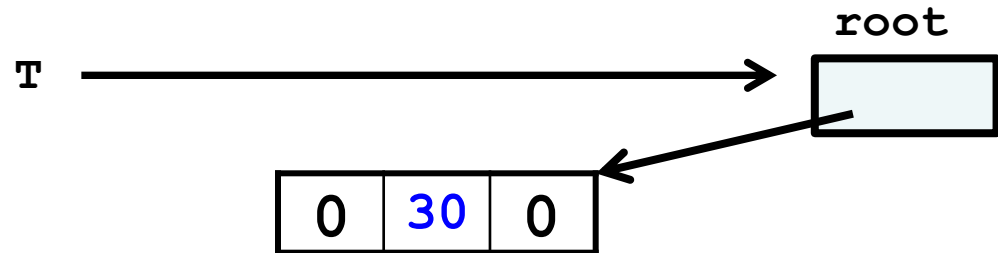
bool InsertBST(BiTNode* &T, ElemType e){
    if(T==0) {
        T = (BiTNode *)malloc(sizeof(BiTNode));
        if(T){ T->data = e;
                T->lchild = 0;    T->rchild = 0; }
        return true;
    }
    //if (T->data == e) return false;
    if (e<T->data)           //左子树
        InsertBST(T->lchild, e);
    else if (e>T->data)       //右子树
        InsertBST(T->rchild, e);
    return true;
}

```

```

void main(){
    BiTNode* root = 0;
    ElemType e;
    e = 30;InsertBST(root, e);
    e = 20;InsertBST(root, e);
    e = 38;InsertBST(root, e);
    e = 22;InsertBST(root, e);
    e = 19;InsertBST(root, e);
}

```



```

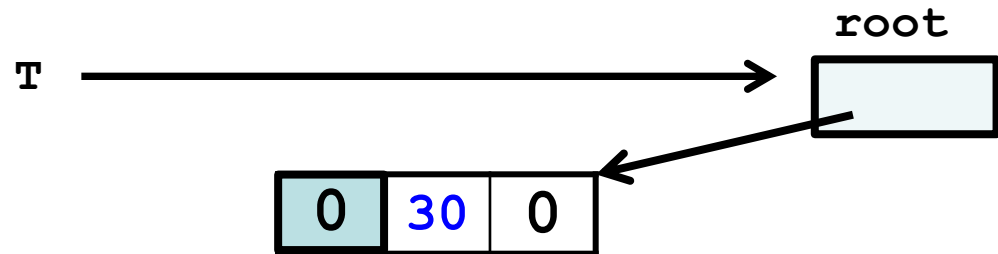
bool InsertBST(BiTNode* &T, ElemType e){
    if(T==0) {
        T = (BiTNode *)malloc(sizeof(BiTNode));
        if(T){ T->data = e;
                T->lchild = 0;    T->rchild = 0; }
        return false;
    }
    //if (T->data == e) return false;
    if (e<T->data)           //左子树
        InsertBST(T->lchild, e);
    else if (e>T->data)       //右子树
        InsertBST(T->rchild, e);
    return true;
}

```

```

void main(){
    BiTNode* root = 0;
    ElemType e;
    e = 30;InsertBST(root, e);
    e = 20;InsertBST(root, e);
    e = 38;InsertBST(root, e);
    e = 22;InsertBST(root, e);
    e = 19;InsertBST(root, e);
}

```




```

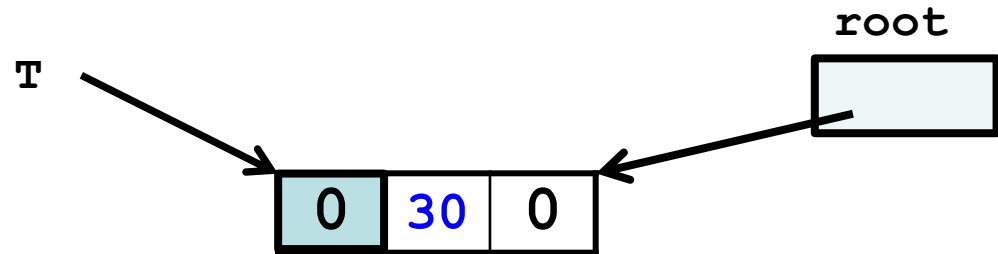
bool InsertBST(BiTNode* &T, ElemType e){
    if(T==0) {
        T = (BiTNode *)malloc(sizeof(BiTNode));
        if(T){ T->data = e;
                T->lchild = 0;    T->rchild = 0; }
        return true;
    }
    //if (T->data == e) return false;
    if (e<T->data)          //左子树
        InsertBST(T->lchild, e);
    else if (e>T->data)      //右子树
        InsertBST(T->rchild, e);
    return true;
}

```

```

void main(){
    BiTNode* root = 0;
    ElemType e;
    e = 30;InsertBST(root, e);
    e = 20;InsertBST(root, e);
    e = 38;InsertBST(root, e);
    e = 22;InsertBST(root, e);
    e = 19;InsertBST(root, e);
}

```



```

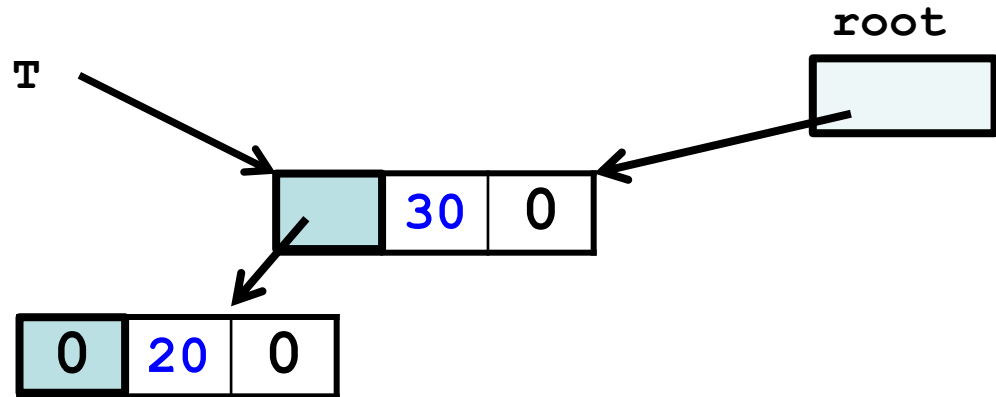
bool InsertBST(BiTNode* &T, ElemType e){
    if(T==0) {
        T = (BiTNode *)malloc(sizeof(BiTNode));
        if(T){ T->data = e;
                T->lchild = 0;    T->rchild = 0; }
        return true;
    }
    //if (T->data == e) return false;
    if (e<T->data)           //左子树
        InsertBST(T->lchild, e);
    else if (e>T->data)       //右子树
        InsertBST(T->rchild, e);
    return true;
}

```

```

void main(){
    BiTNode* root = 0;
    ElemType e;
    e = 30;InsertBST(root, e);
    e = 20;InsertBST(root, e);
    e = 38;InsertBST(root, e);
    e = 22;InsertBST(root, e);
    e = 19;InsertBST(root, e);
}

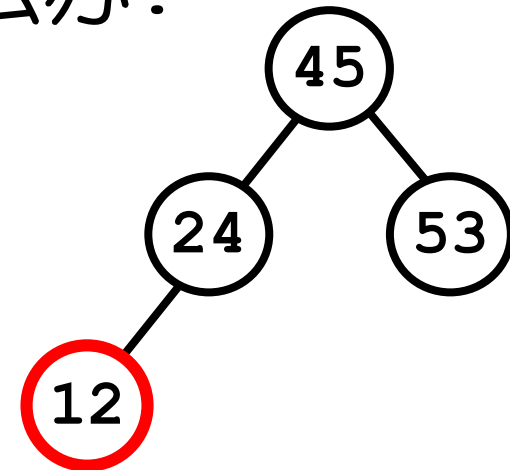
```



二叉查找树的结点的删除

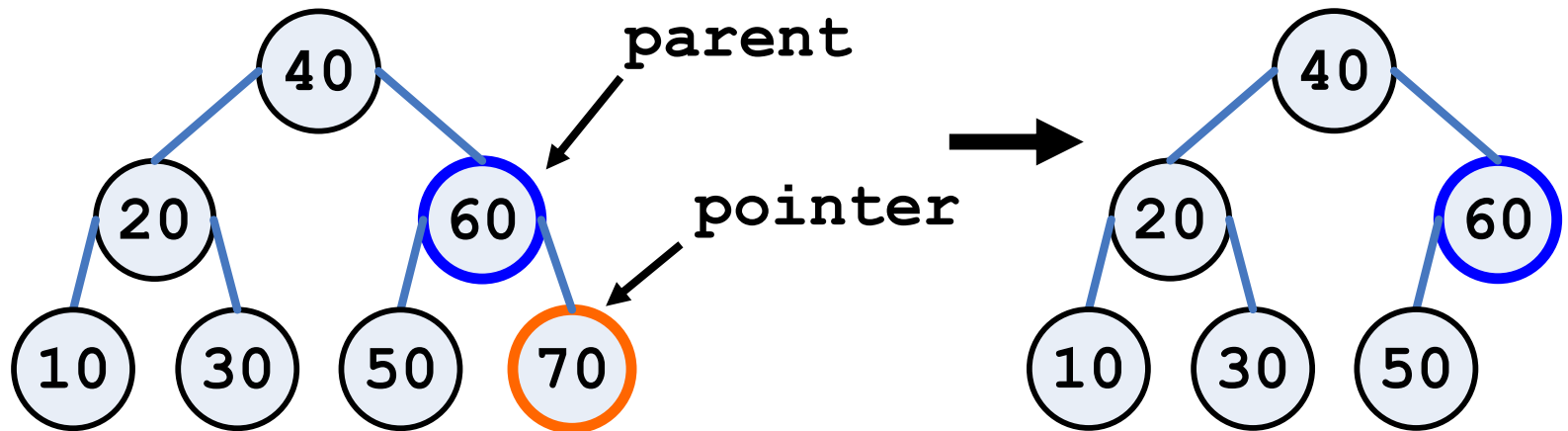
- 基本思想

- 找到待删除结点， `pointer`
- 找到待删除结点的父结点， `parent`
- 该结点的子孙怎么办？即谁来继承它？
- 该结点的父结点指针怎么办？



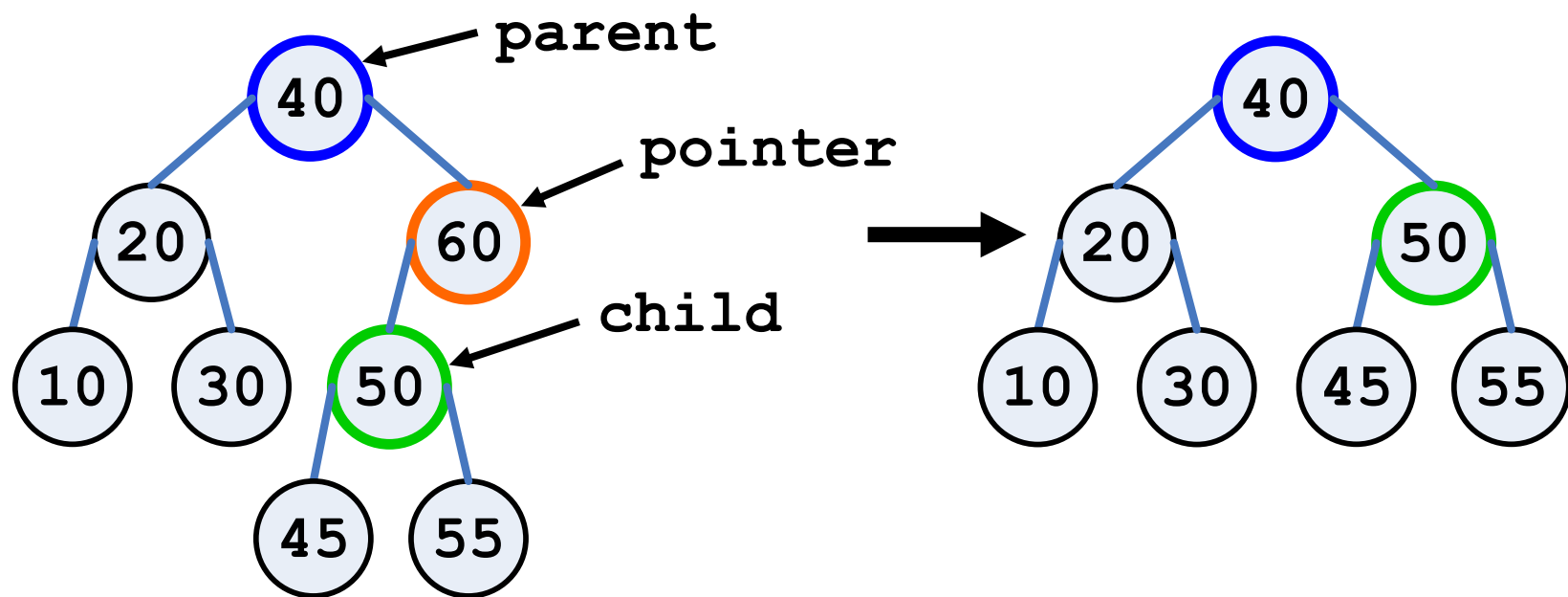
二叉查找树的结点的删除

- 待删除的结点是叶结点
 - 若 `pointer` 是 `parent` 的左孩子
 - `parent->lchild = NULL`
 - 否则: `parent->rchild = NULL`
 - 最后释放 `pointer` 的空间



二叉查找树的结点的删除

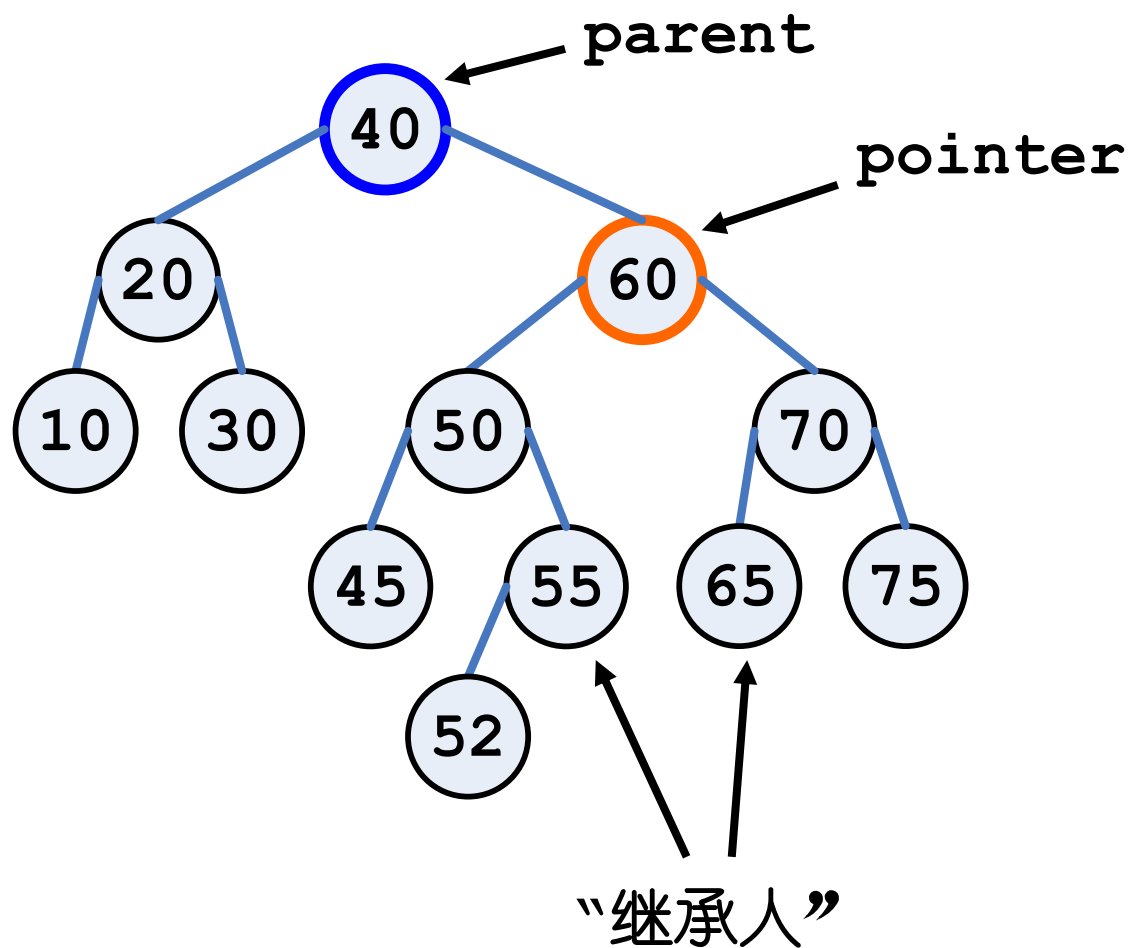
- 待删除的结点只有1个孩子



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

二叉查找树的结点的删除

- 待删除的结点有2个孩子

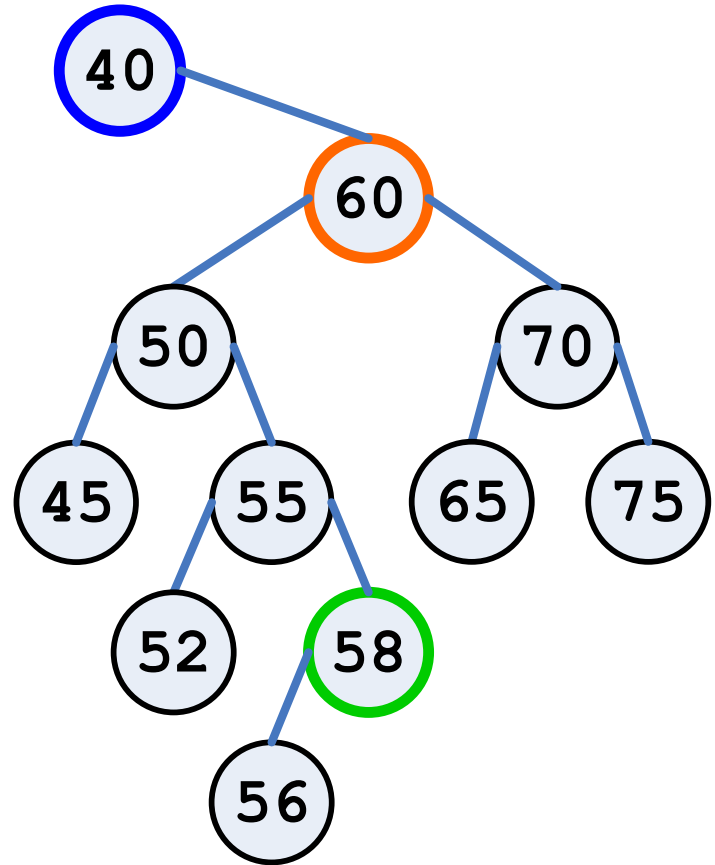


二叉查找树的结点的删除

- “继承人”

- 左子树中的最右的结点

- 因为它是左子树中最大的
 - 能够保证比左子树中其余的结点都大；比右子树中的所有结点都小

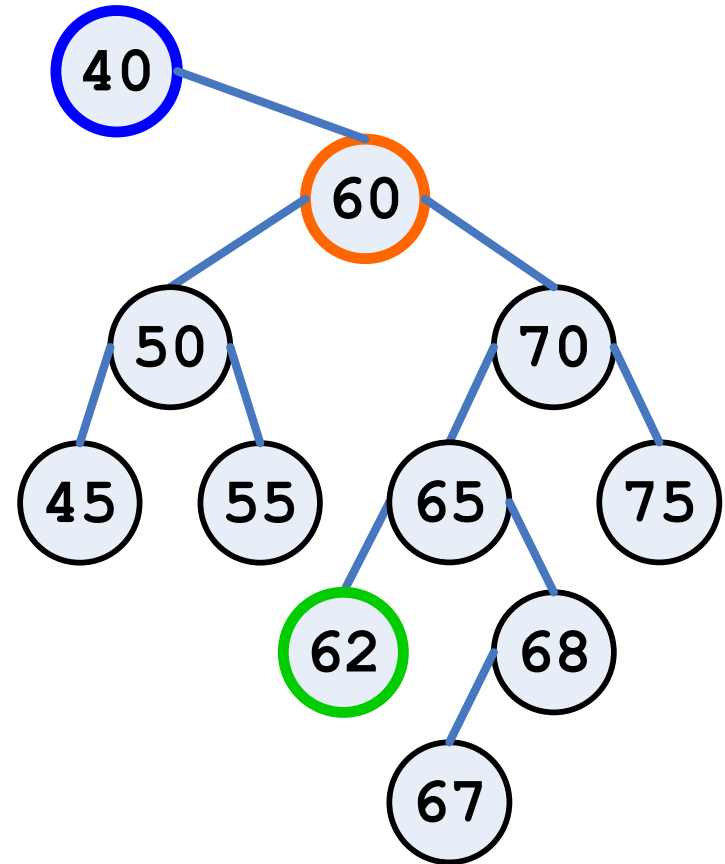


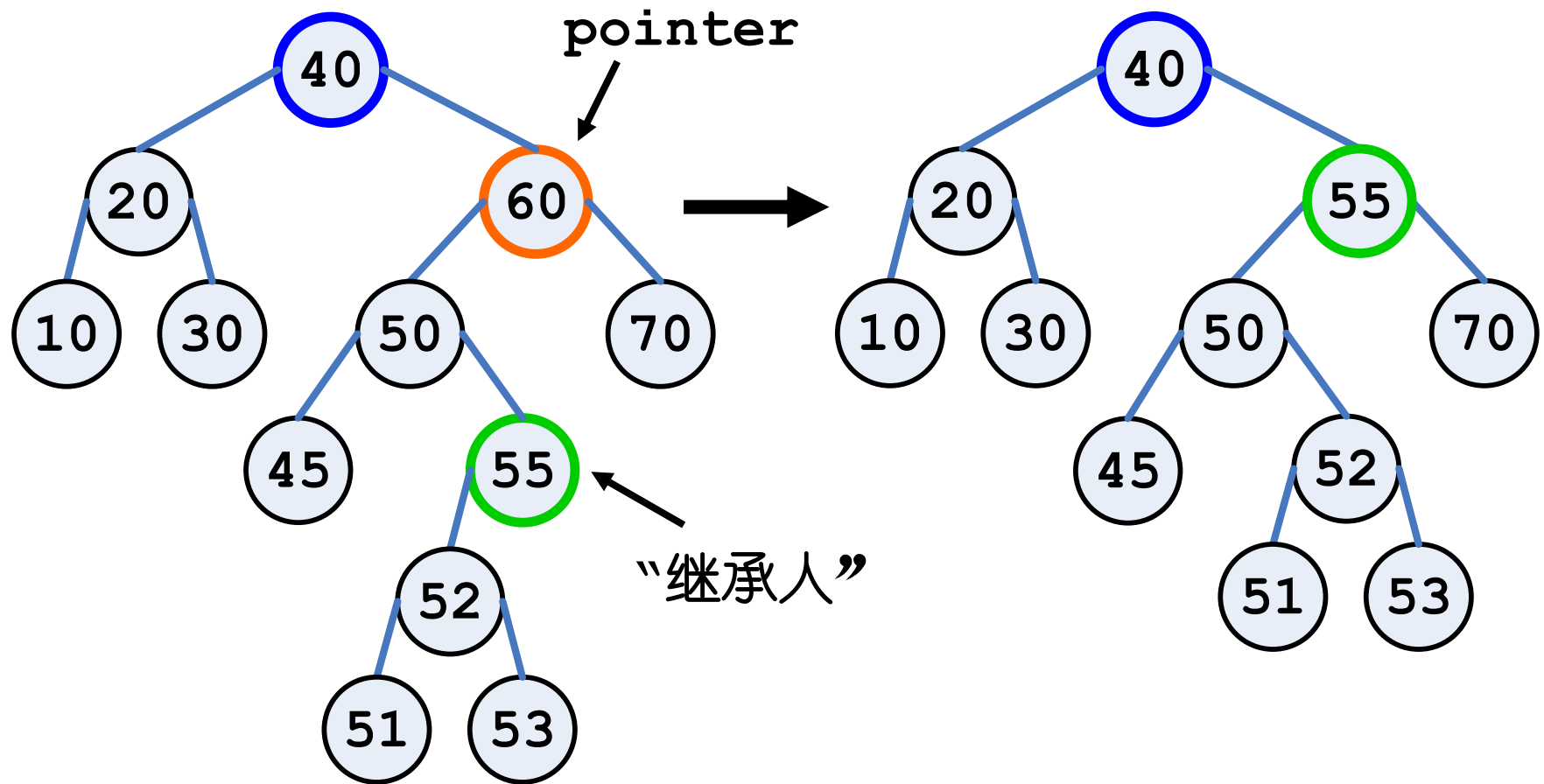
二叉查找树的结点的删除

- “继承人”

- 右子树中的最左的结点

- 因为它是右子树中最小的
 - 能够保证比左子树中的所有结点都大；比右子树中的其它结点都小





- “继承人” 顶替被删除结点
- 问题是：谁又来顶替“继承人”的位置呢？
 - “继承人” 最多只有一个孩子，否则它就不会是最左/最右的了

二叉查找树的结点的删除

- 新的查找算法：

//找到待删除的结点的双亲结点

```
BiTNode* SearchBST( BiTNode* &T, ElemType e,
                    BiTNode* &parent){
    if (!T) return 0; //递归出口

    if (T->data == e) return T;           //处理根
    parent = T;
    if (e < T->data)                       //左子树
        return SearchBST(T->lchild, e, parent);
    else                                   //右子树
        return SearchBST(T->rchild, e, parent);
}
```

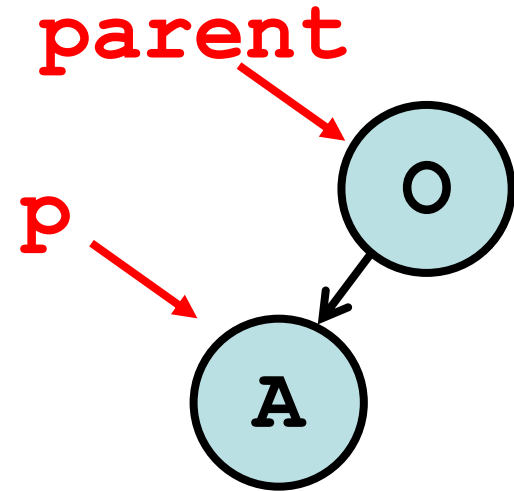
二叉查找树的结点的删除

```
bool DeleteBST( BiTNode* &root, ElemType e) {  
    if(!root) return false;  
    BiTNode *T = root, *parent = 0;  
    BiTNode* p = SearchBST(T,e,parent) ;  
    if(p)  
        Delete( p, parent);  
    return true;  
}
```

```

void Delete( BiTNode* &p, BiTNode* parent) {
    BiTNode* q ; if(!p) return;
    if (!p->lchild && !p->rchild ) {
        if(parent){
            if( parent->lchild==p) parent->lchild=0;
            else parent->rchild=0;
        }
        free(p) ;p = 0; return;
    }
    else if ( !p->rchild) {
        /*...*/
    }
    else if (!p->lchild){
        /*...*/
    }
    else {          /*...*/
        return true;
    }
}

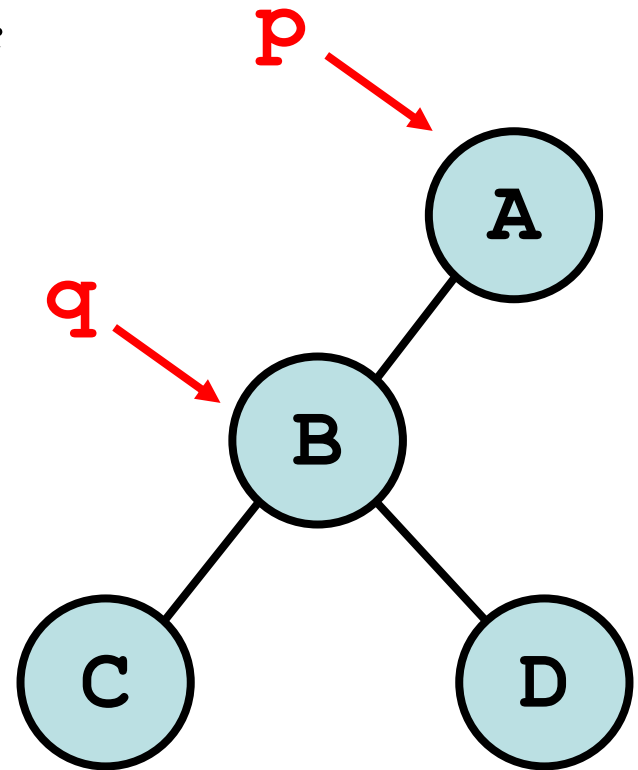
```



```

void Delete( BiTNode* &p, BiTNode* parent) {
    BiTNode* q ; if(!p) return;
    if (!p->lchild && !p->rchild ) { /*...*/ }
    else if (!p->rchild) {
        q = p->lchild;
        if(q){ p->data = q->data;
            p->lchild = q->lchild;
            p->rchild = q->rchild;
            free(q) ;
        }
    }
    else if (!p->lchild) {
        /*...*/
    }
    else {
        /*...*/
    }
    return;
}

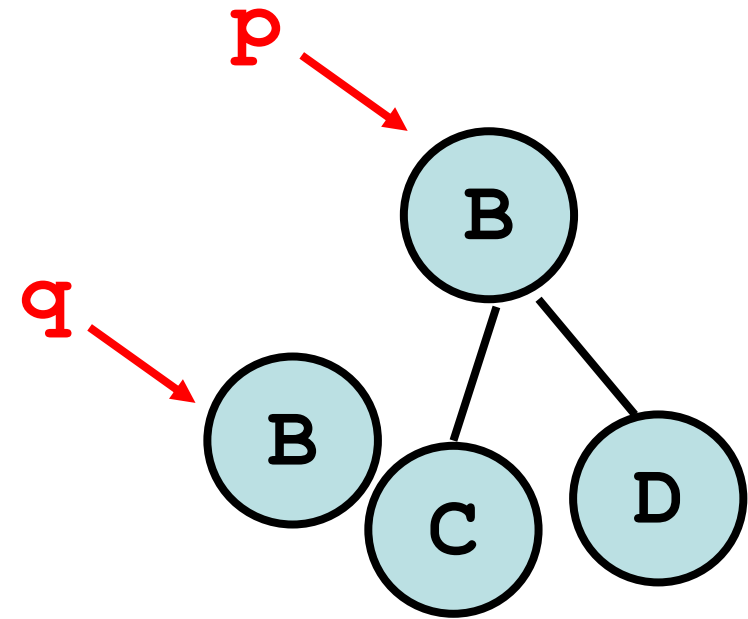
```



```

void Delete( BiTNode* &p, BiTNode* parent) {
    BiTNode* q ; if(!p) return;
    if (!p->lchild && !p->rchild ) { /*...*/ }
    else if (!p->rchild) {
        q = p->lchild;
        if(q){ p->data = q->data;
            p->lchild = q->lchild;
            p->rchild = q->rchild;
            free(q) ;
        }
    }
    else if (!p->lchild) {
        /*...*/
    }
    else {          /*...*/      }
    return;
}

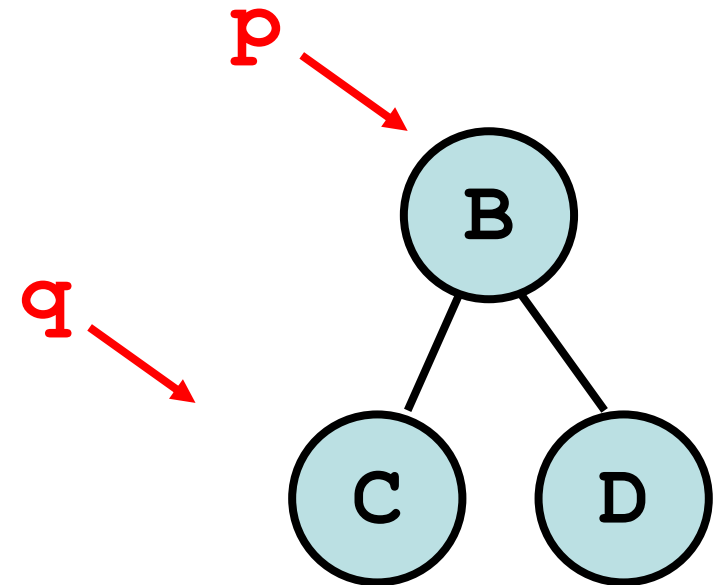
```



```

void Delete( BiTNode* &p, BiTNode* parent) {
    BiTNode* q ; if(!p) return;
    if (!p->lchild && !p->rchild ) { /*...*/ }
    else if (!p->rchild) {
        q = p->lchild;
        if(q){ p->data = q->data;
            p->lchild = q->lchild;
            p->rchild = q->rchild;
            free(q) ;
        }
    }
    else if (!p->lchild) {
        /*...*/
    }
    else {          /*...*/      }
    return;
}

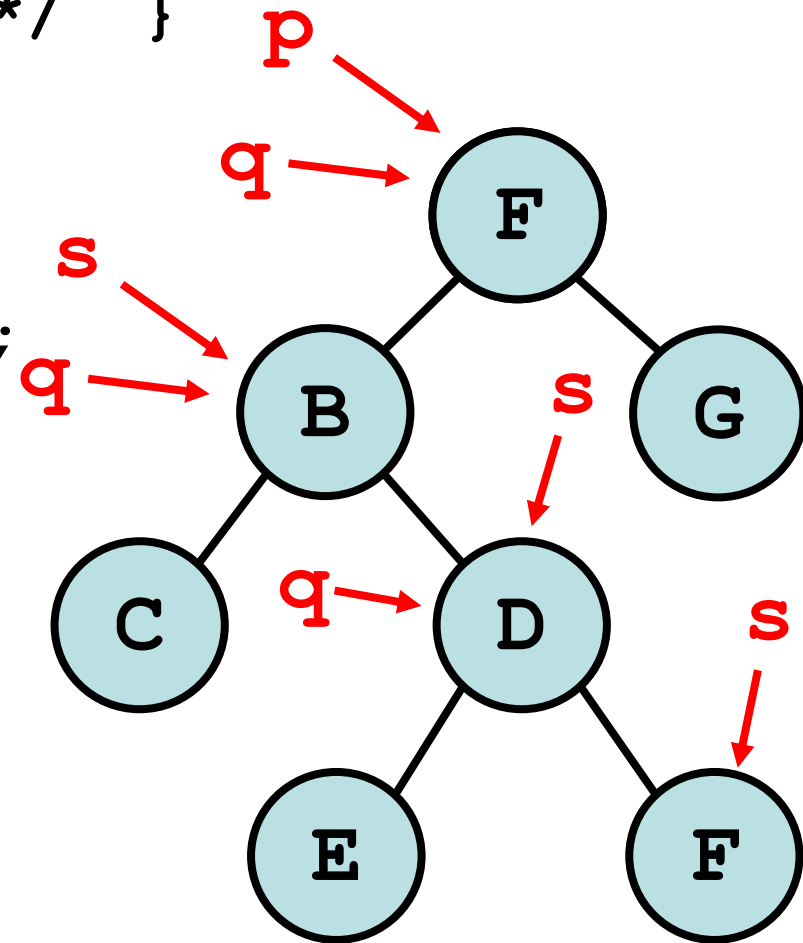
```



```

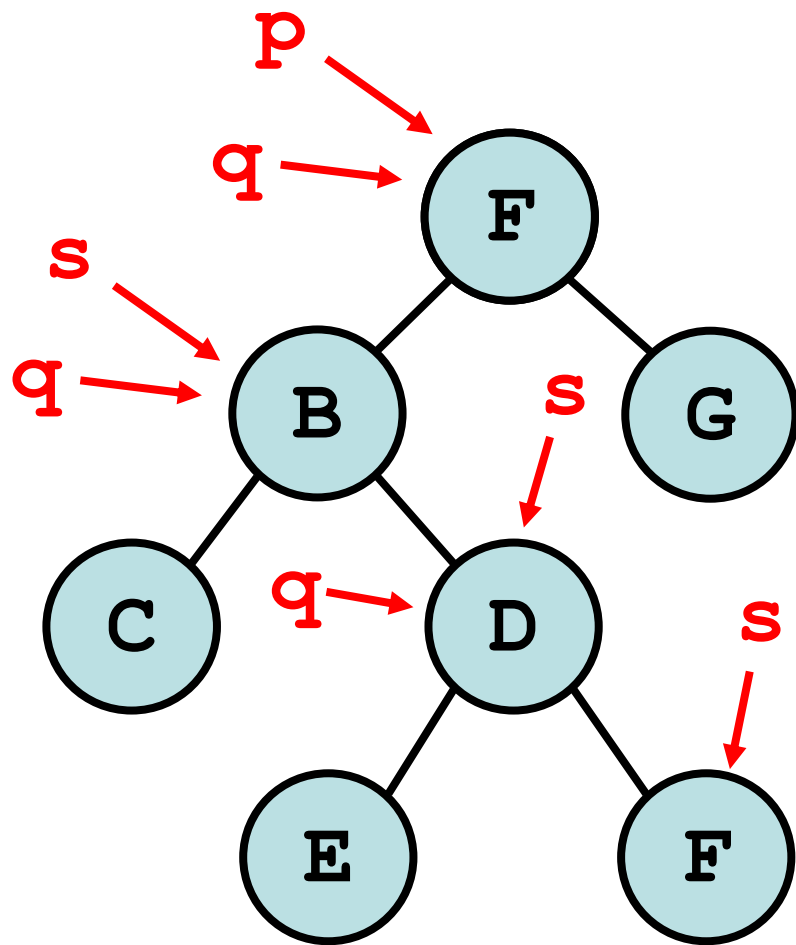
void Delete( BiTNode* &p, BiTNode* parent) {
    BiTNode* q ; if(!p) return;
    if (!p->lchild && !p->rchild ) { /*...*/ }
    else if (!p->rchild) { /*...*/ }
    else if (!p->lchild) { /*...*/ }
    else {
        q = p; s = p->lchild;
        while (s->rchild) {
            q = s; s = s->rchild;
        }
        p->data = s->data;
        if (q != p)
            q->rchild=s->lchild;
        else
            q->lchild=s->lchild;
        free(s) ;
    }
    return;
}

```



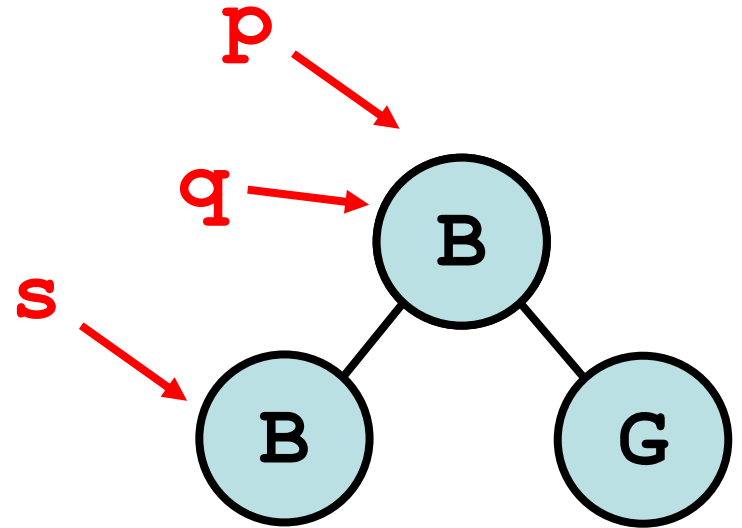
二叉查找树的结点的删除

```
q = p;  
s = p->lchild;  
while (s->rchild)  
{  
    q = s;  
    s = s->rchild;  
}  
p->data = s->data;
```



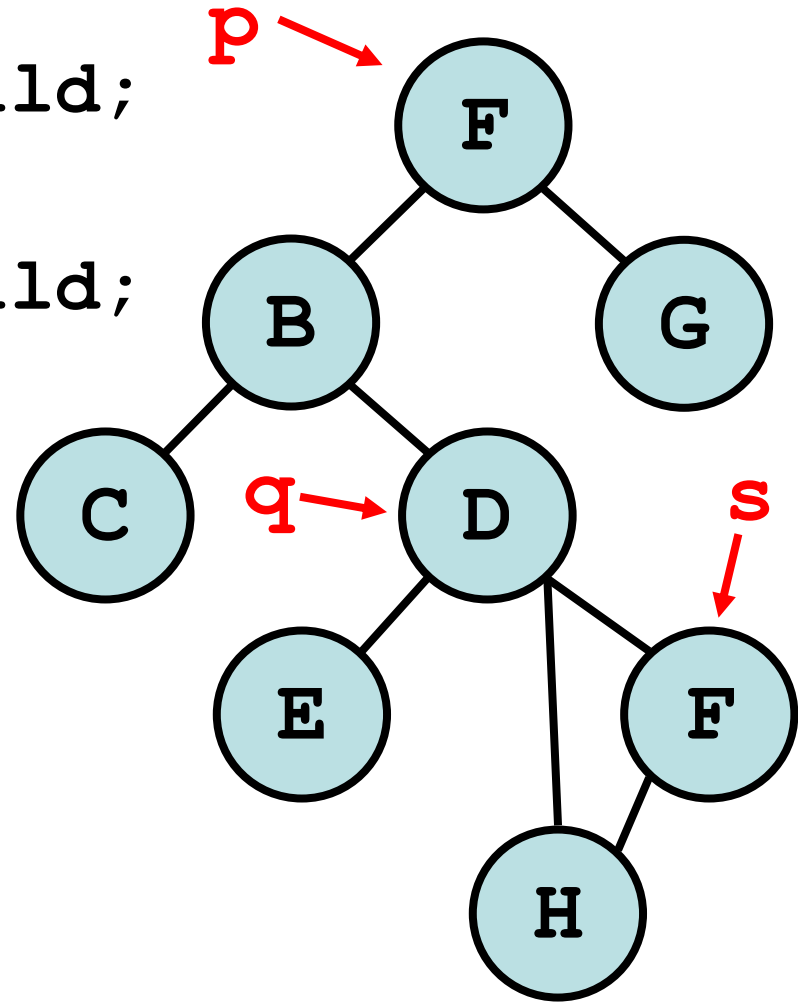
二叉查找树的结点的删除

```
q = p;  
s = p->lchild;  
while (s->rchild)  
{  
    q = s;  
    s = s->rchild  
}  
p->data = s->data;
```



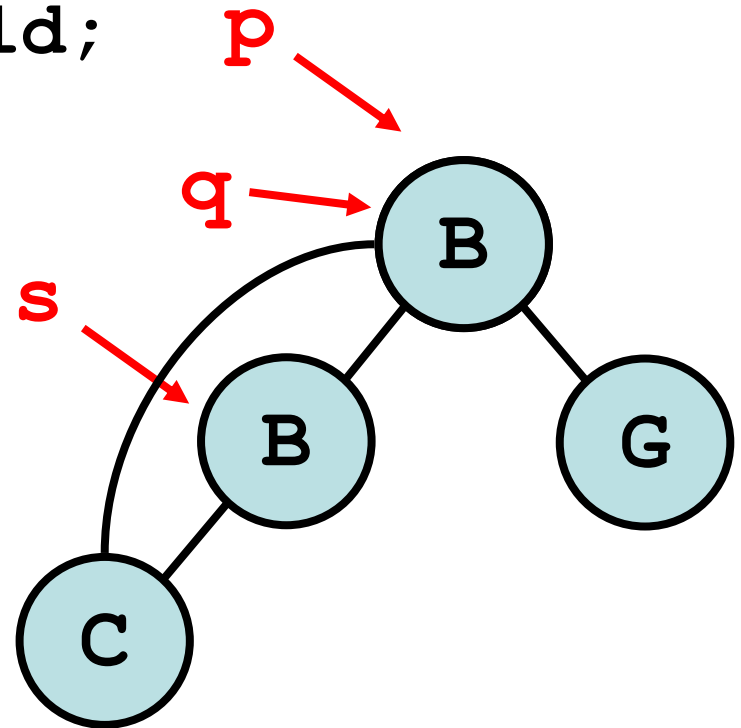
二叉查找树的结点的删除

```
if (q != p)
    q->rchild = s->lchild;
else
    q->lchild = s->lchild;
free(s);
```



二叉查找树的结点的删除

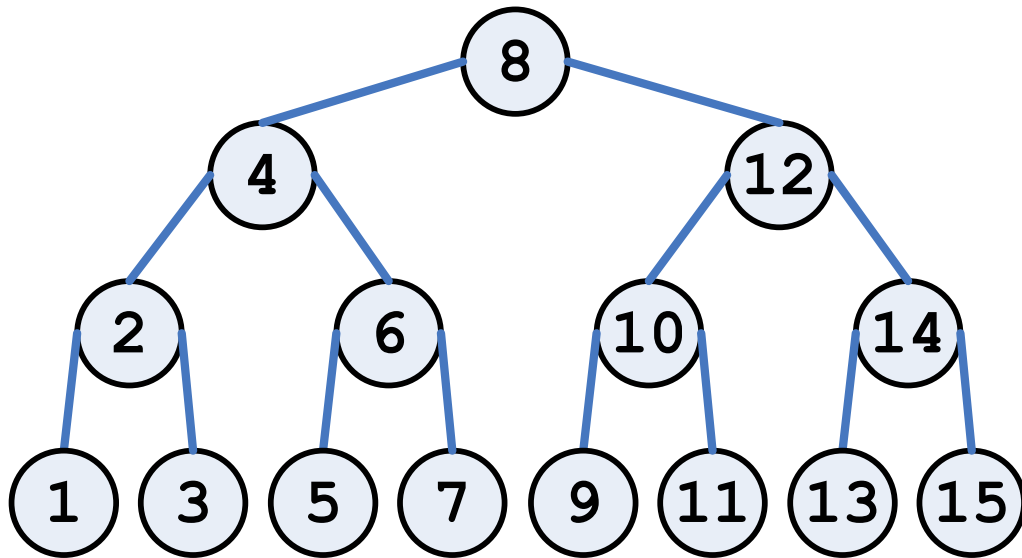
```
if (q != p)
    q->rchild = s->lchild;
else
    q->lchild = s->lchild;
free(s);
```



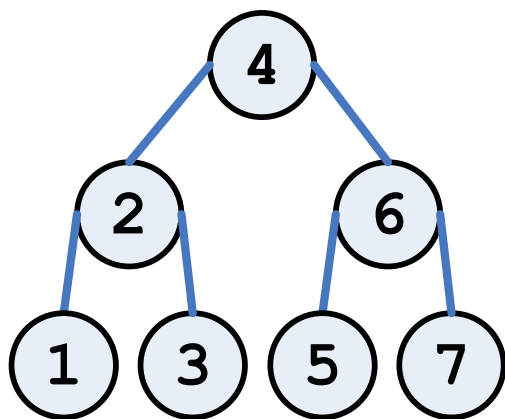
二叉查找

- 查找算法的效率

- 查找的次数 = 树的高度

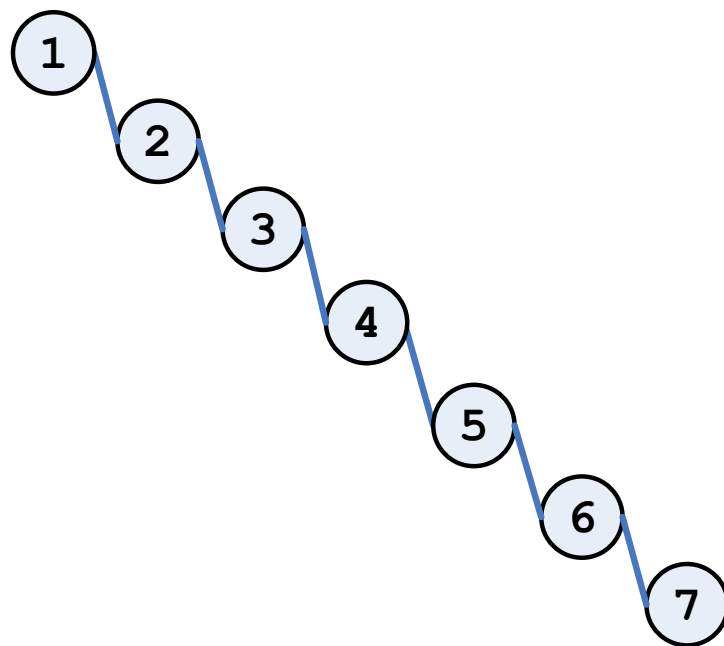


- 因此复杂度= $O(h)$ ，问题是 $h=?$



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

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



$$h = n$$

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

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

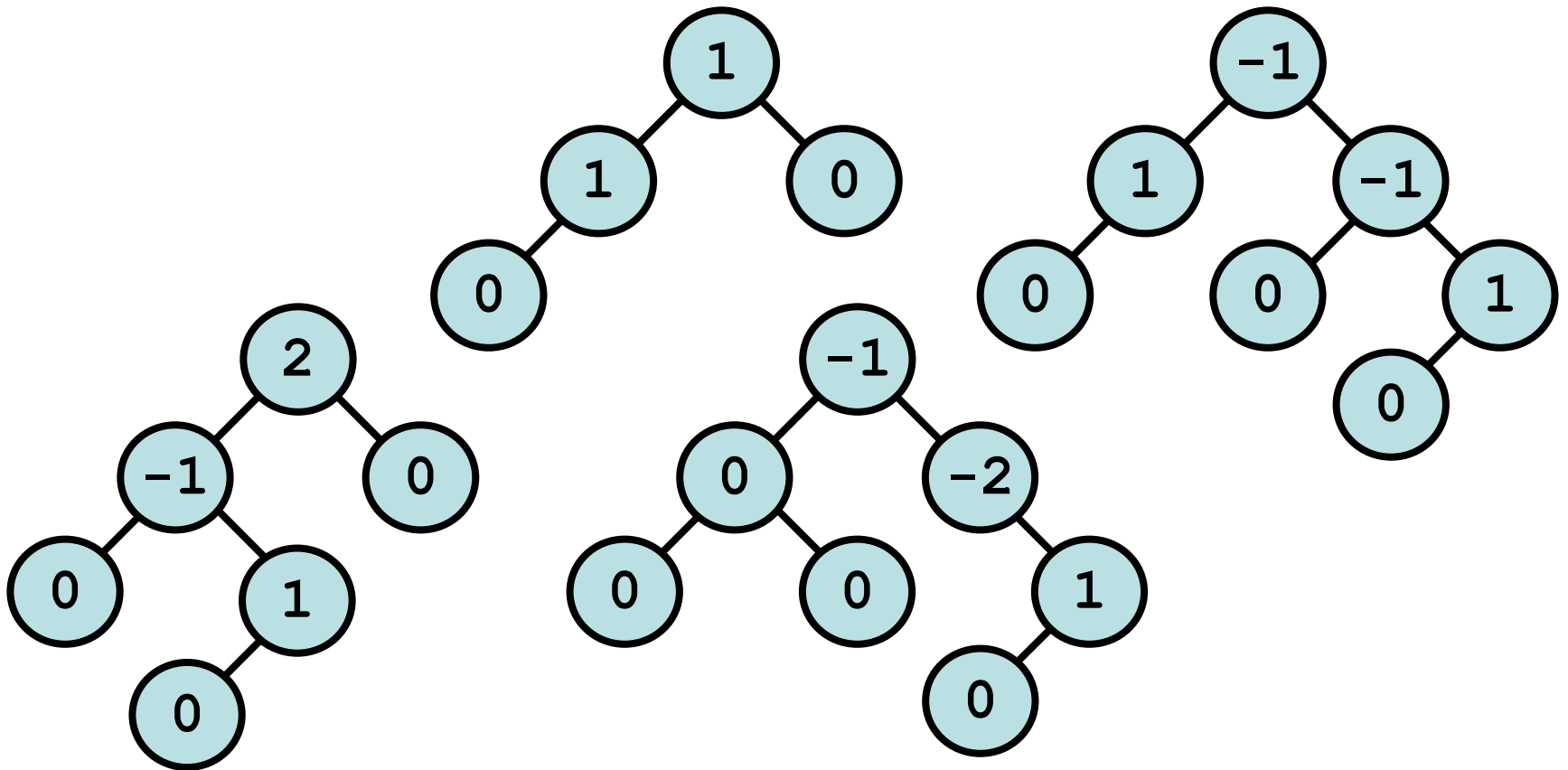
- 或者：

- 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

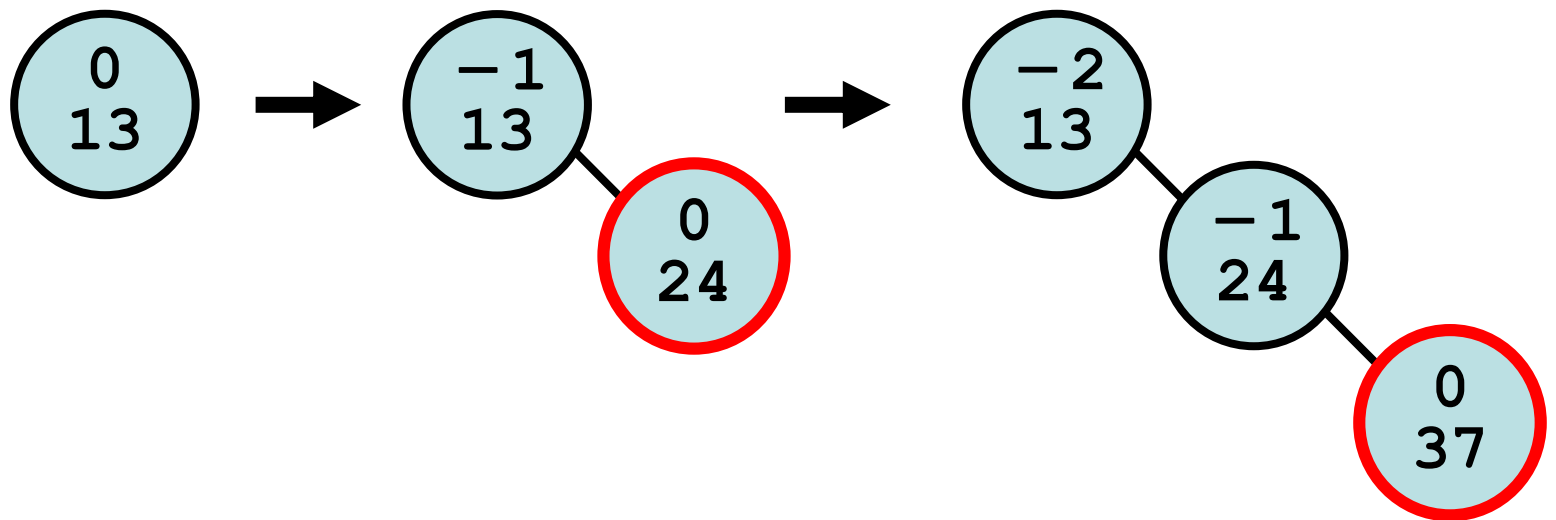
- **平衡因子** = 左子树的深度 - 右子树的深度
 - AVL树上任何结点的平衡因子都 ≤ 1

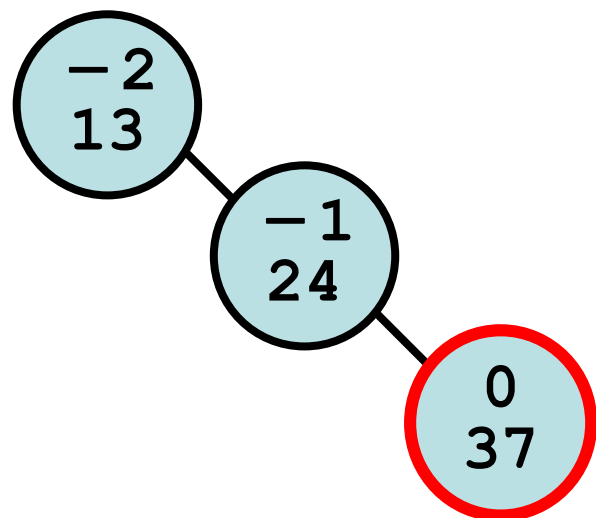


平衡二叉树

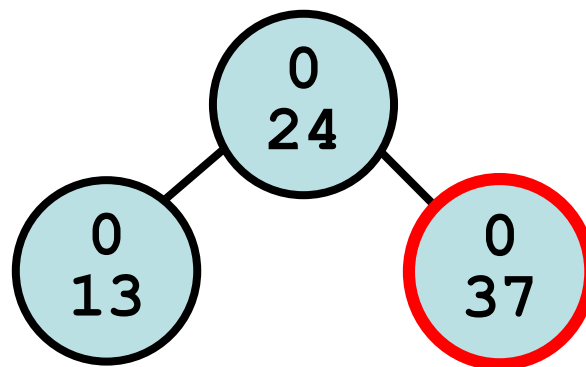
- 平衡二叉树的构造

- 初始为空树
- 不断插入结点
- 如果导致了不平衡，调整之

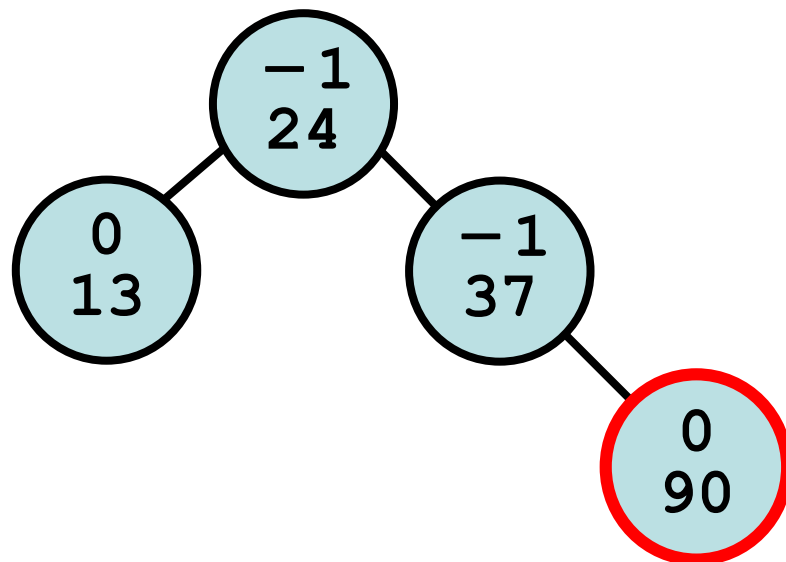




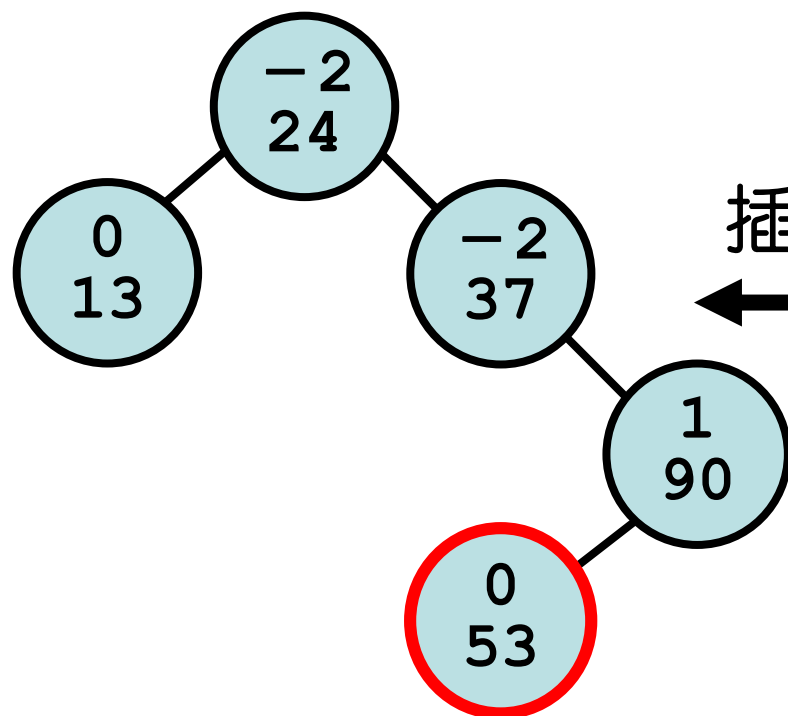
调整

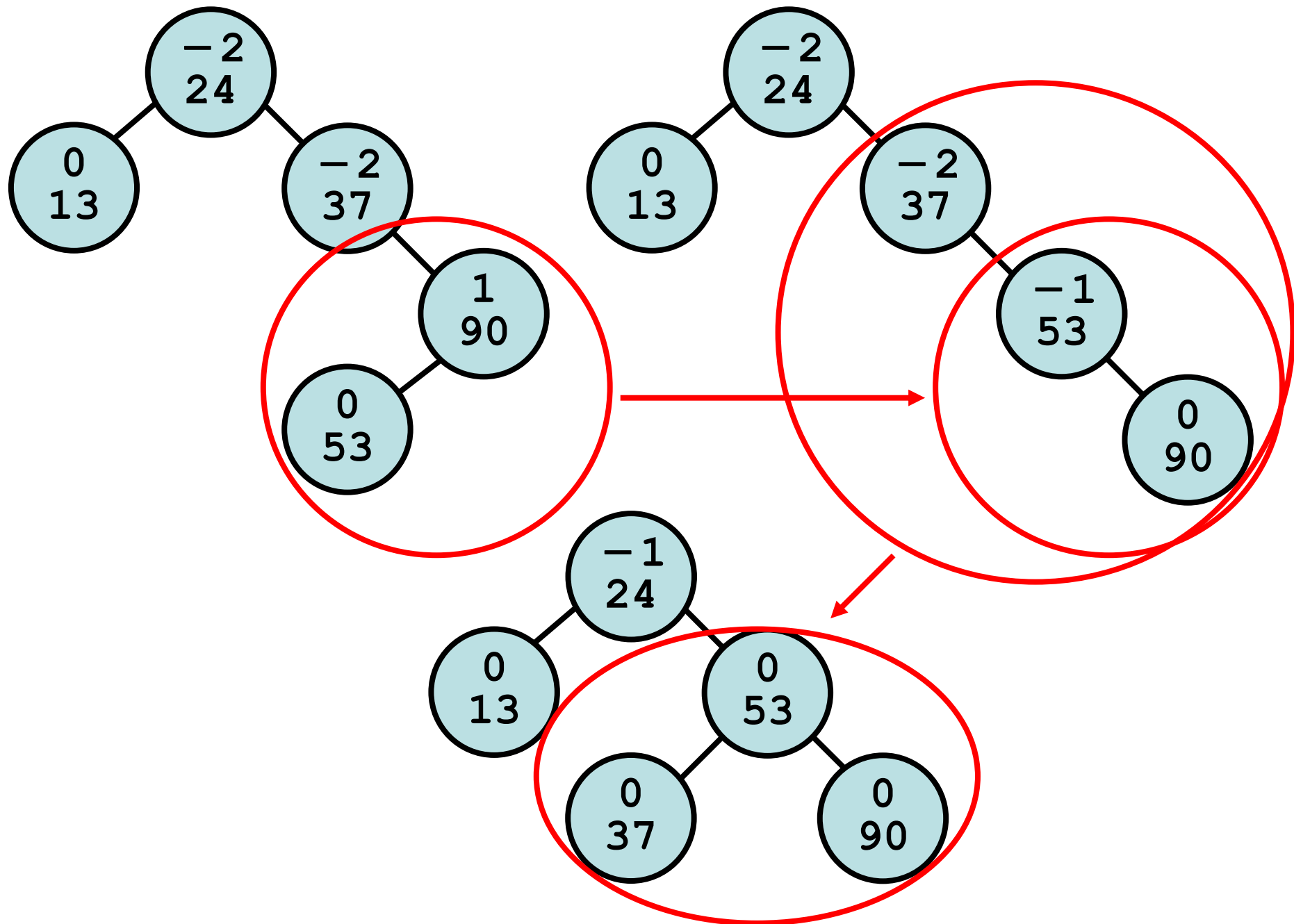


插入90



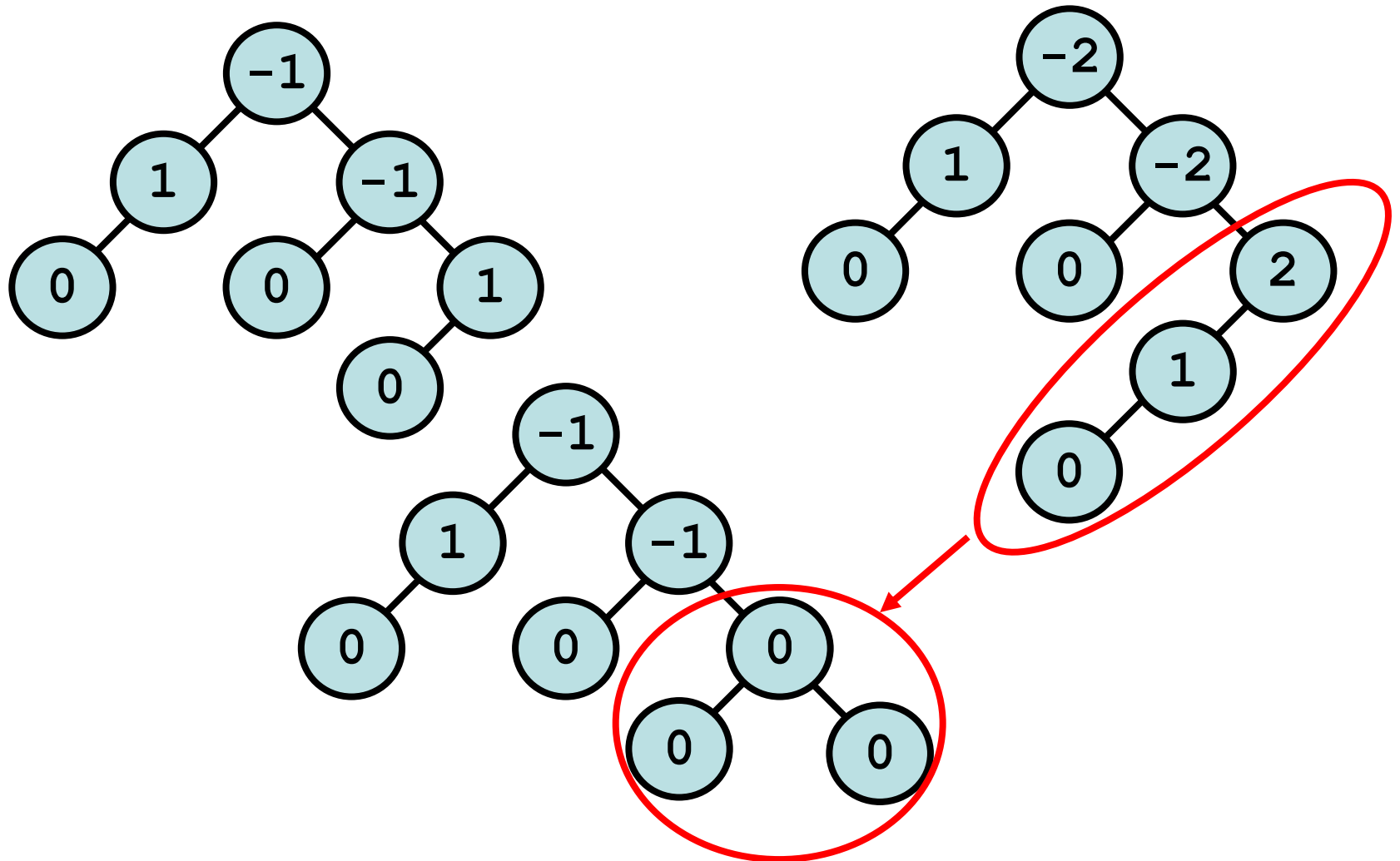
插入53





• 局部影响全局

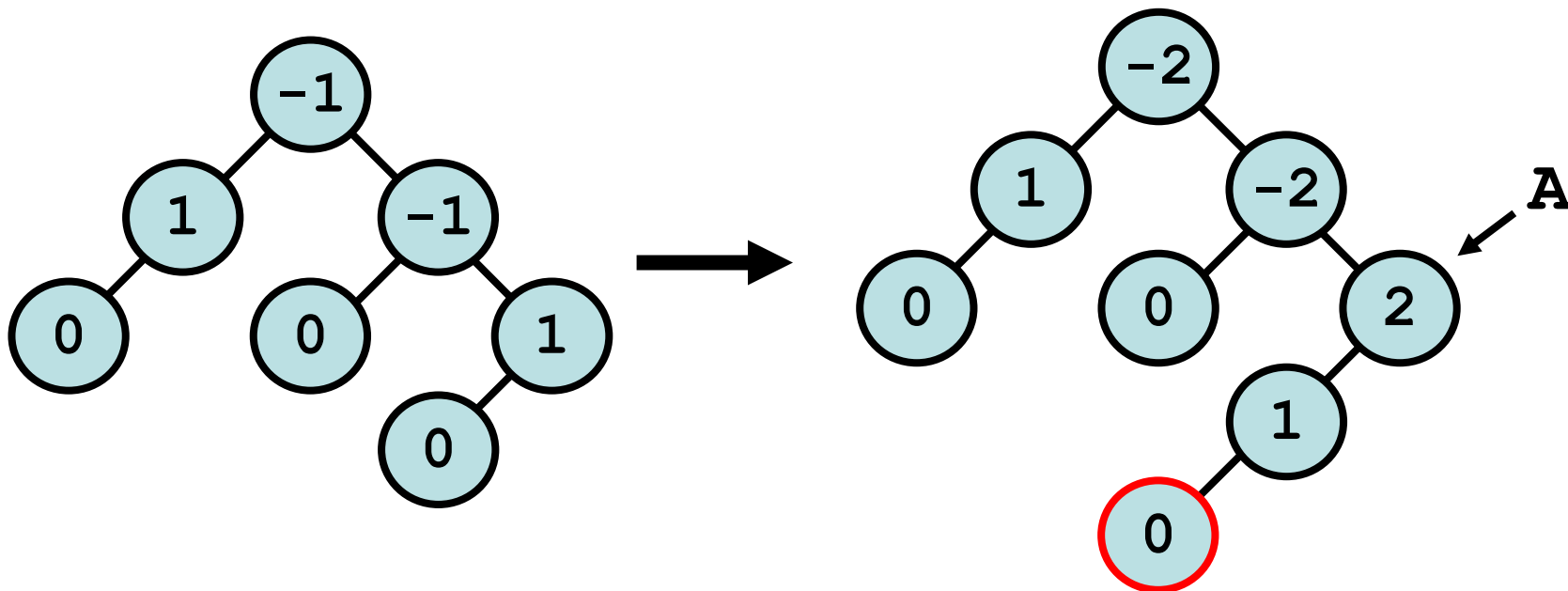
- 不平衡是从局部开始的
- 调整也就只需要从局部着手



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

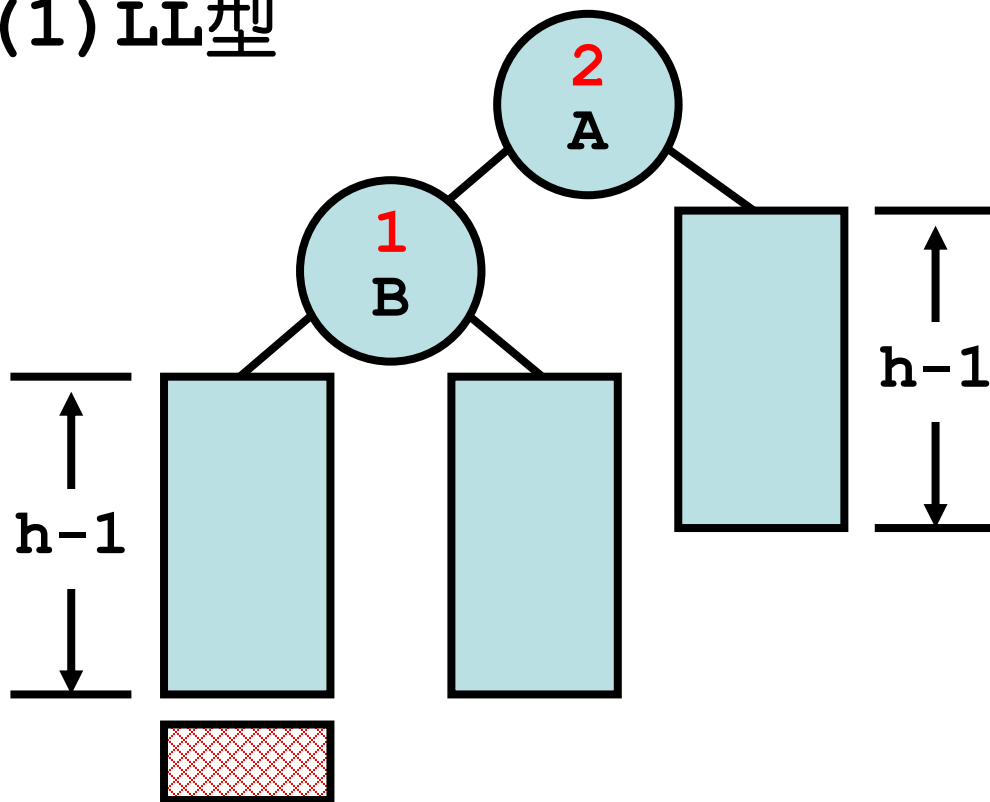


平衡二叉树

• 分情况讨论

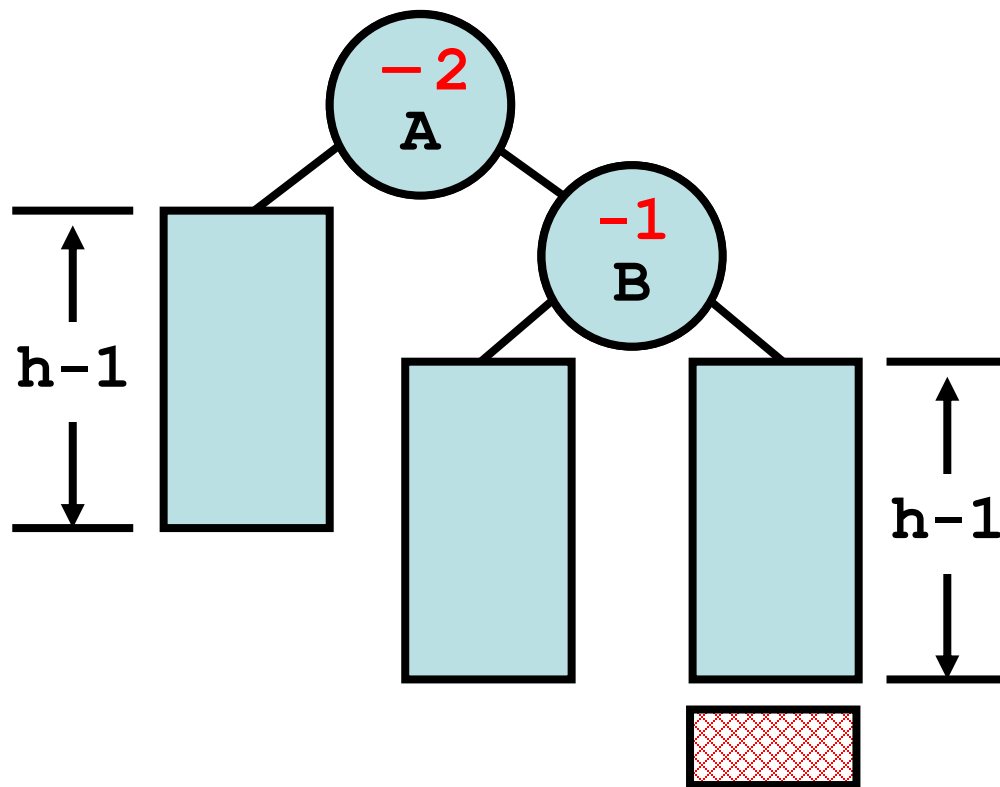
– 以A为子树根的子树就这4种情况：

• (1) LL型



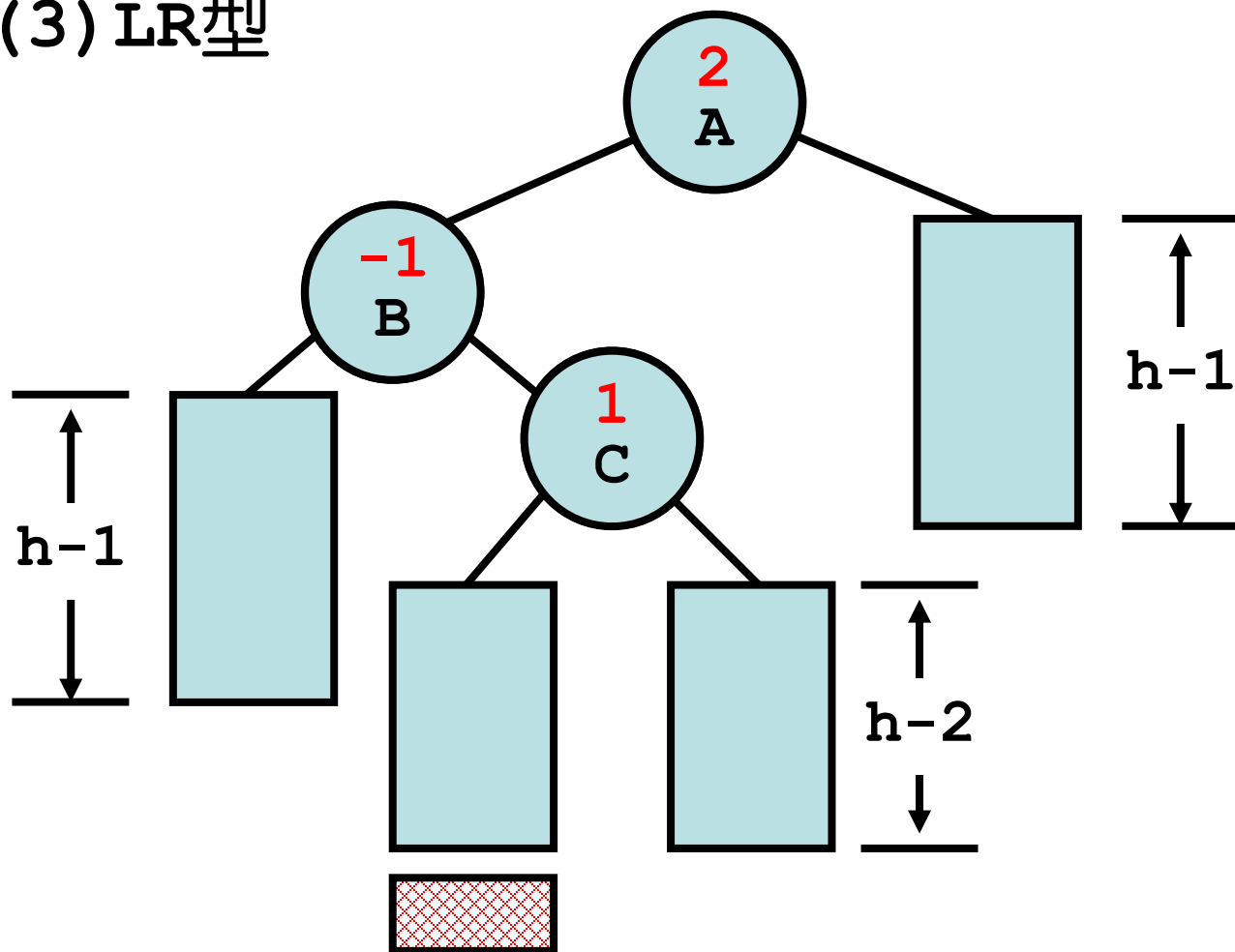
平衡二叉树

- (2) RR型



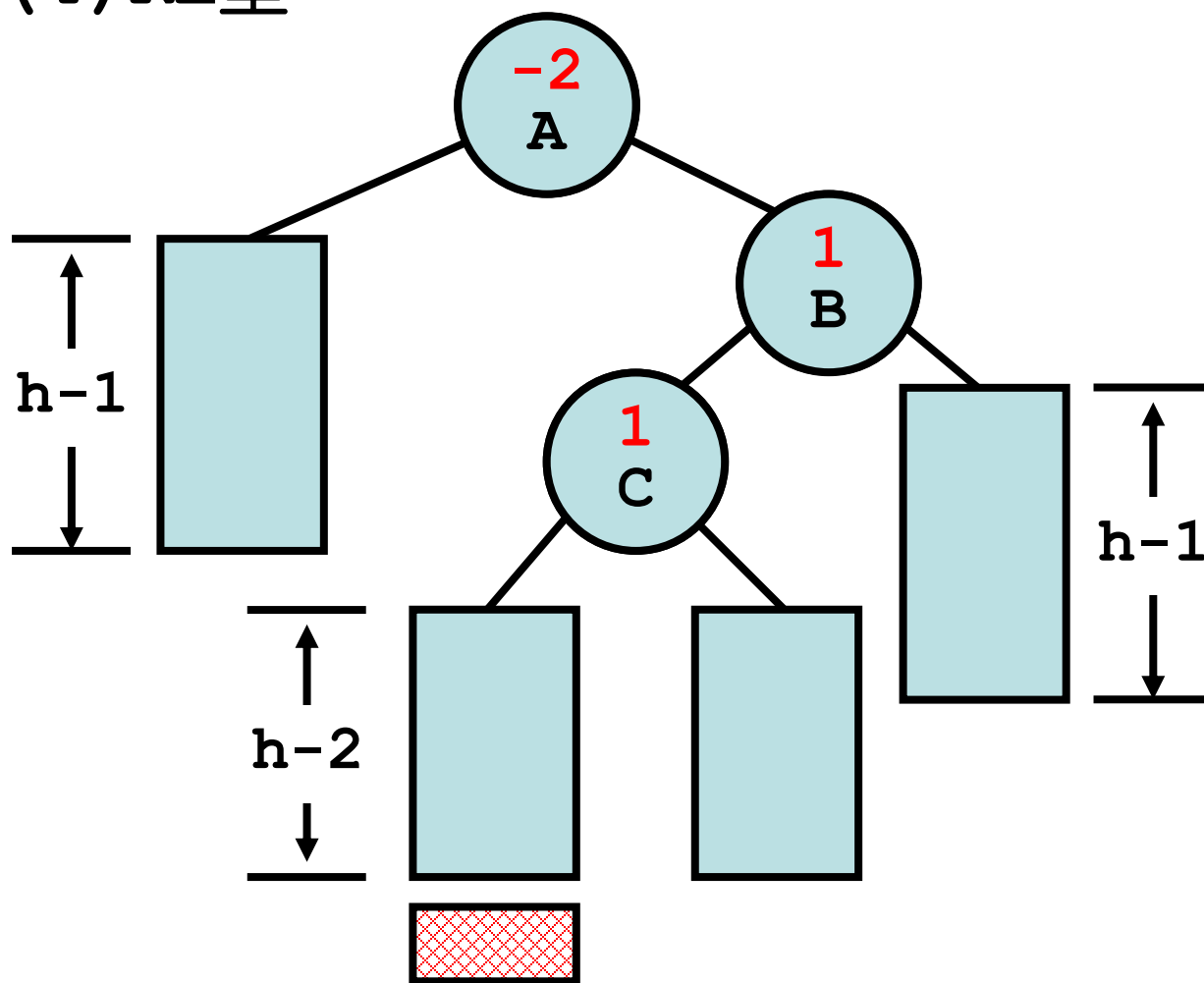
平衡二叉树

- (3) LR型



平衡二叉树

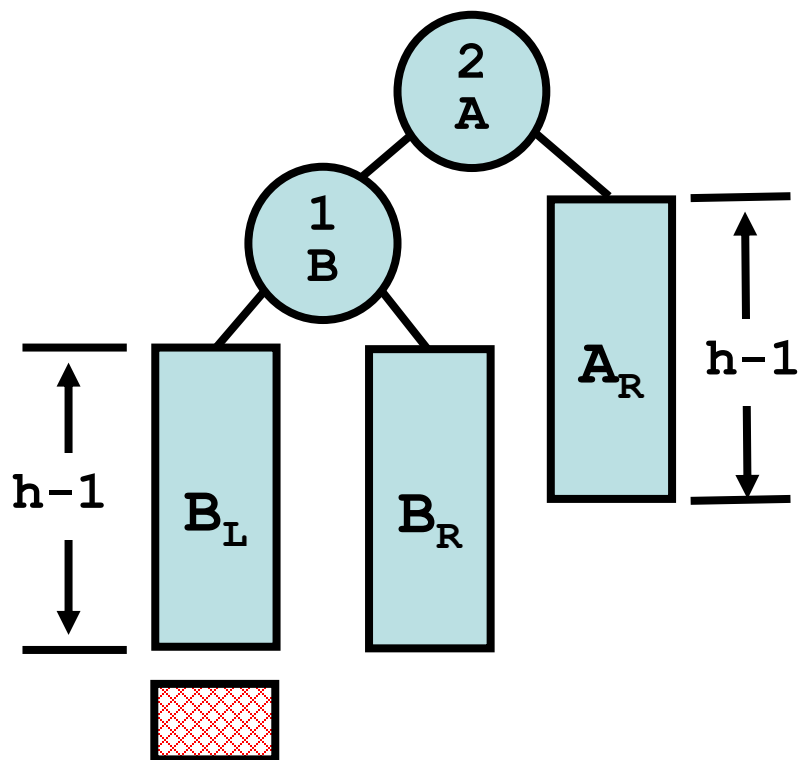
- (4) RL型



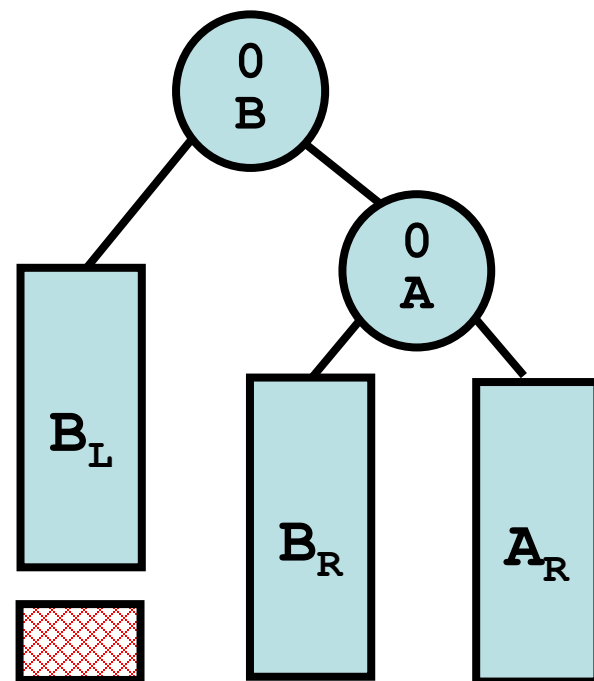
平衡二叉树

- 各个击破

- (1) LL型

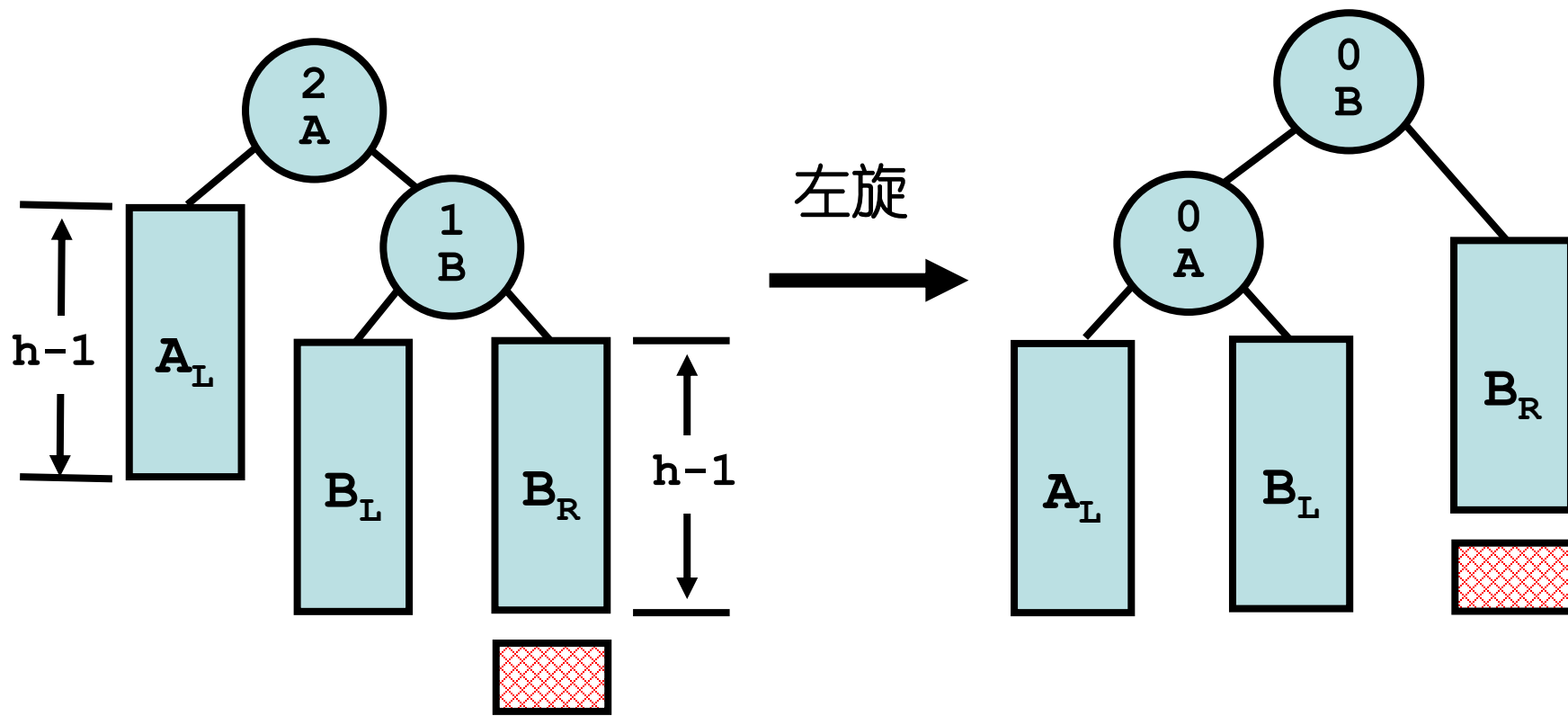


右旋



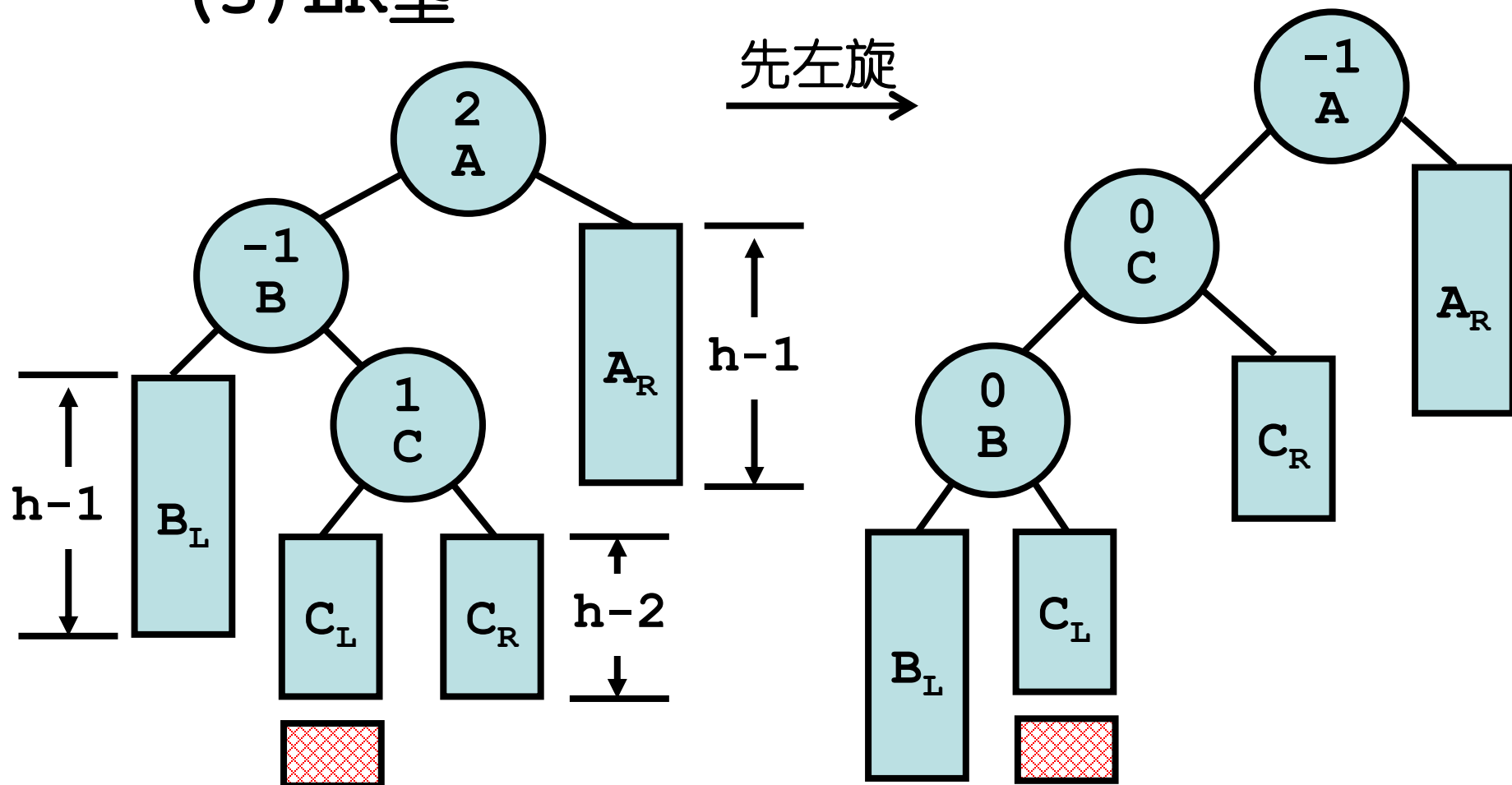
平衡二叉树

— (2) RR型



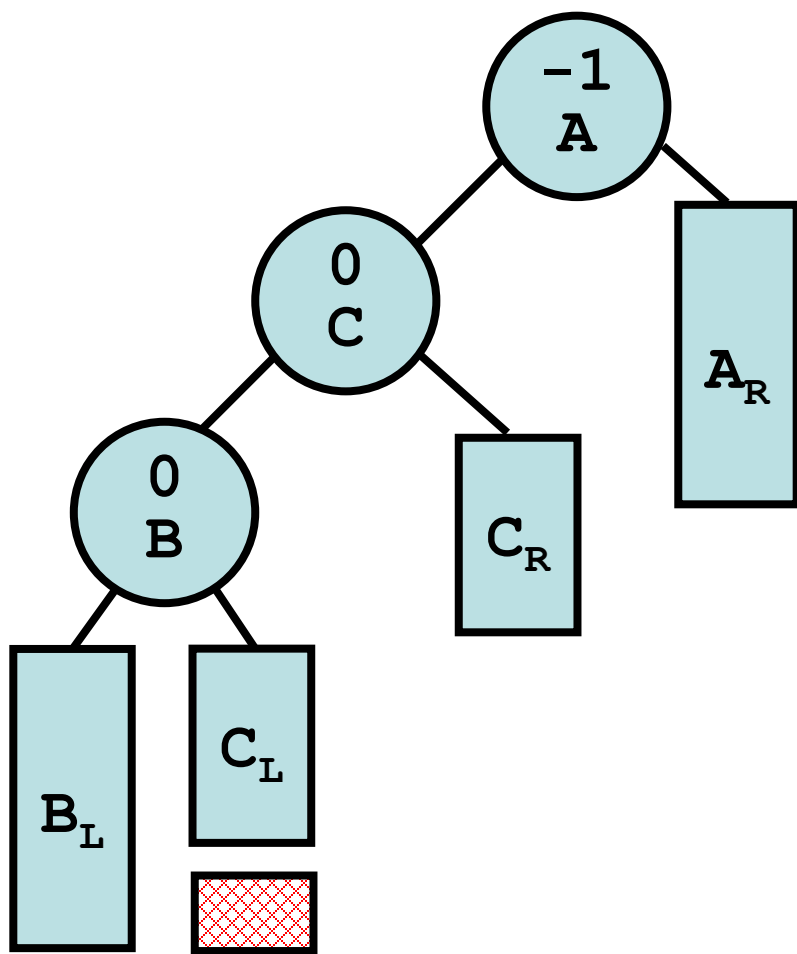
平衡二叉树

— (3) LR型

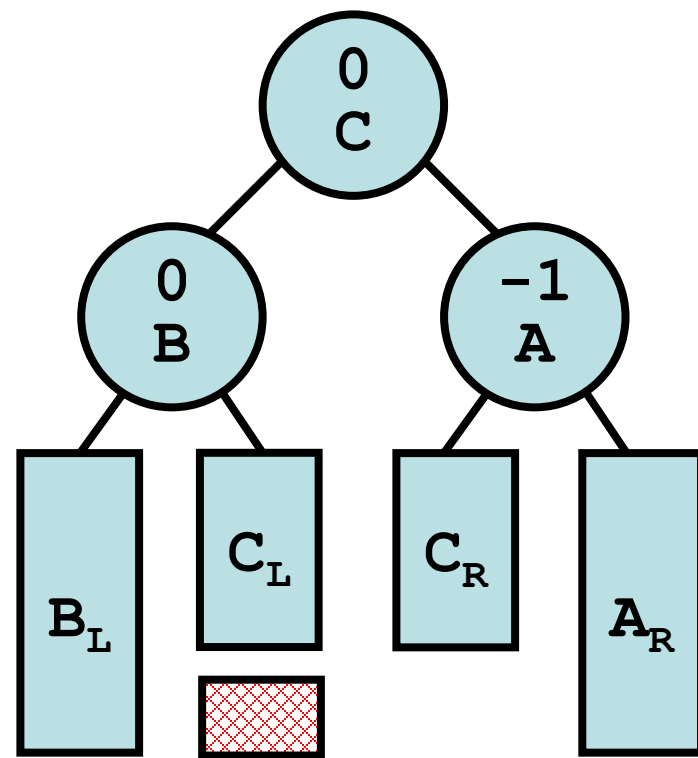


平衡二叉树

— (3) LR型

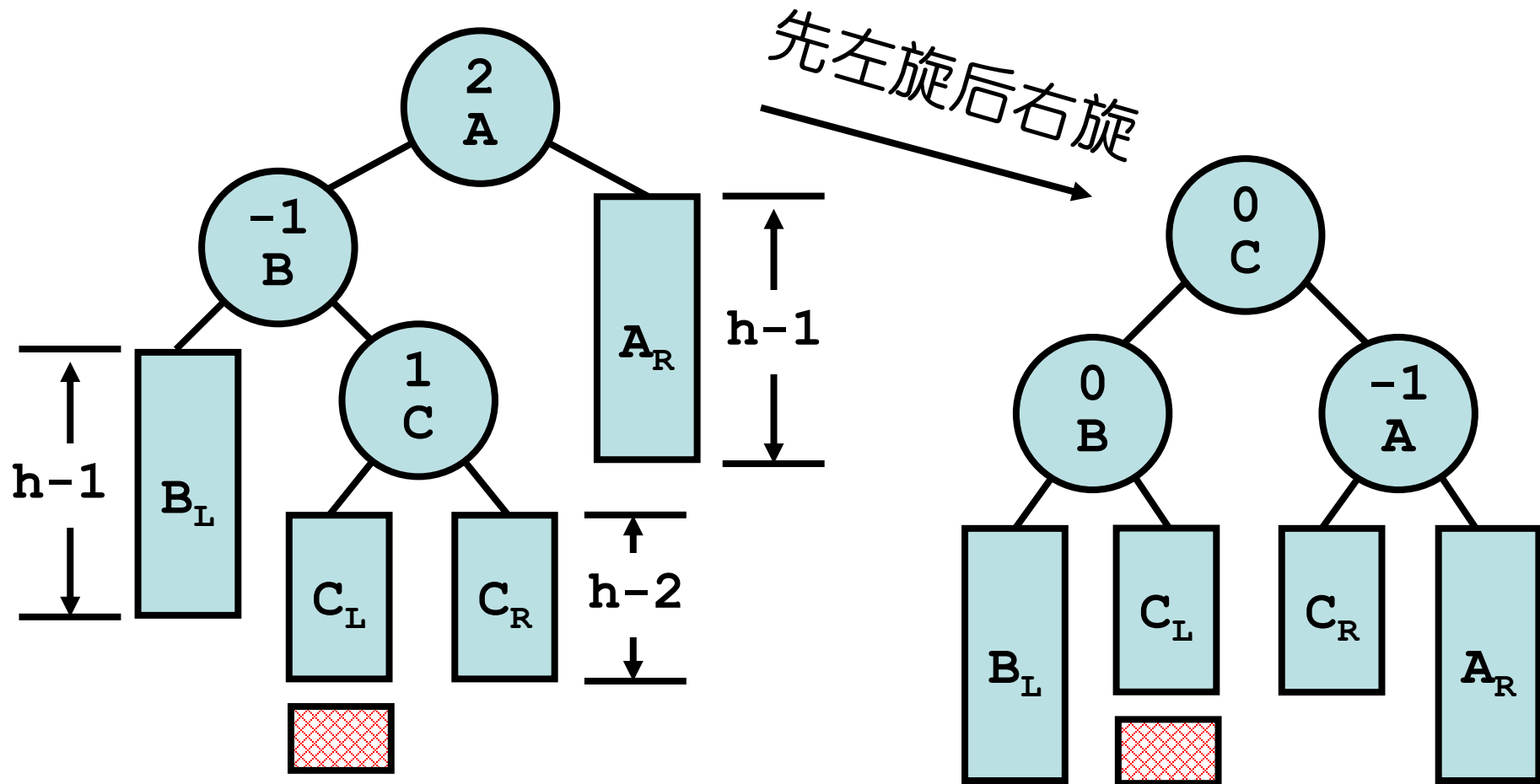


后右旋



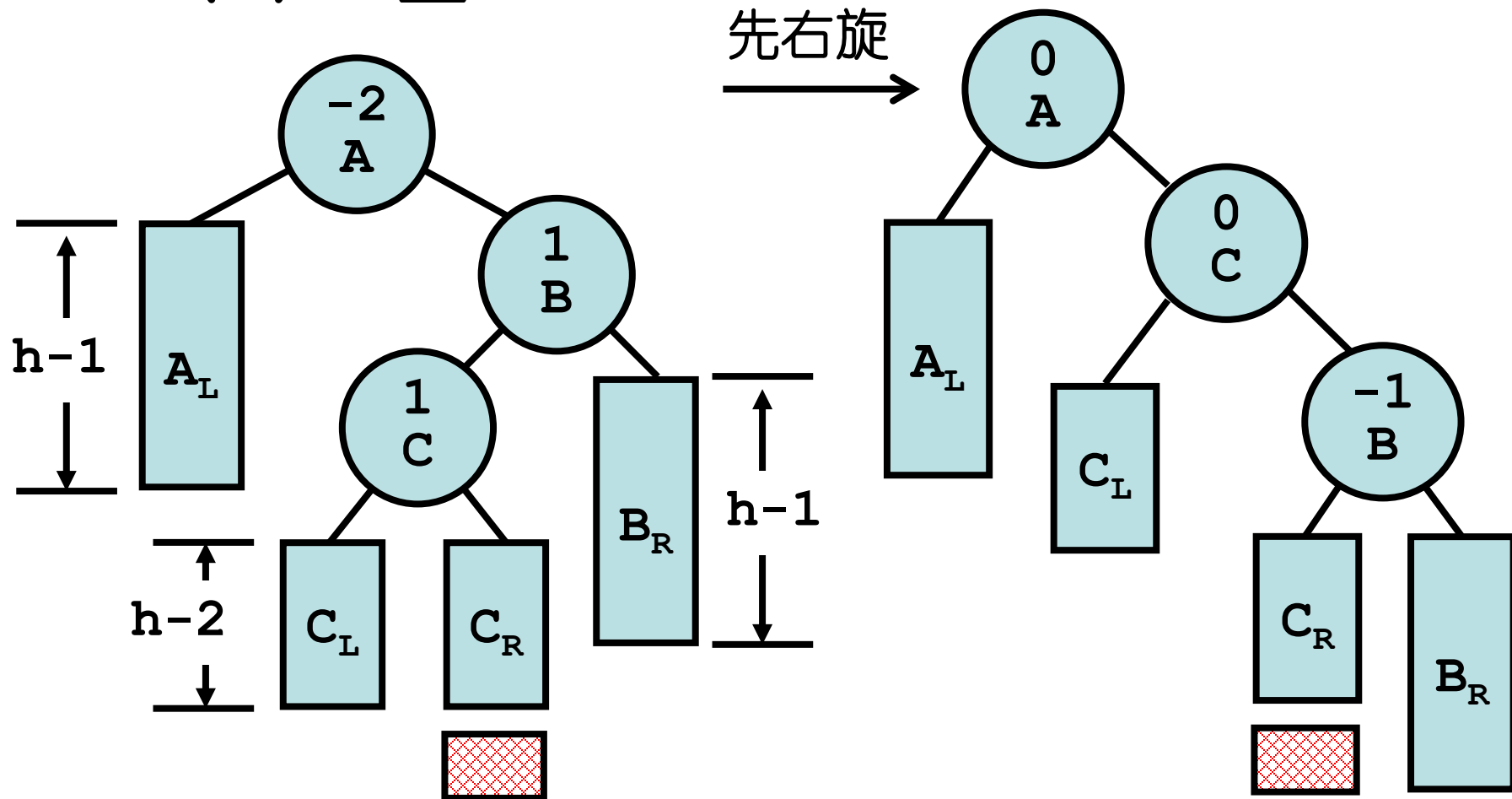
平衡二叉树

— (3) LR型



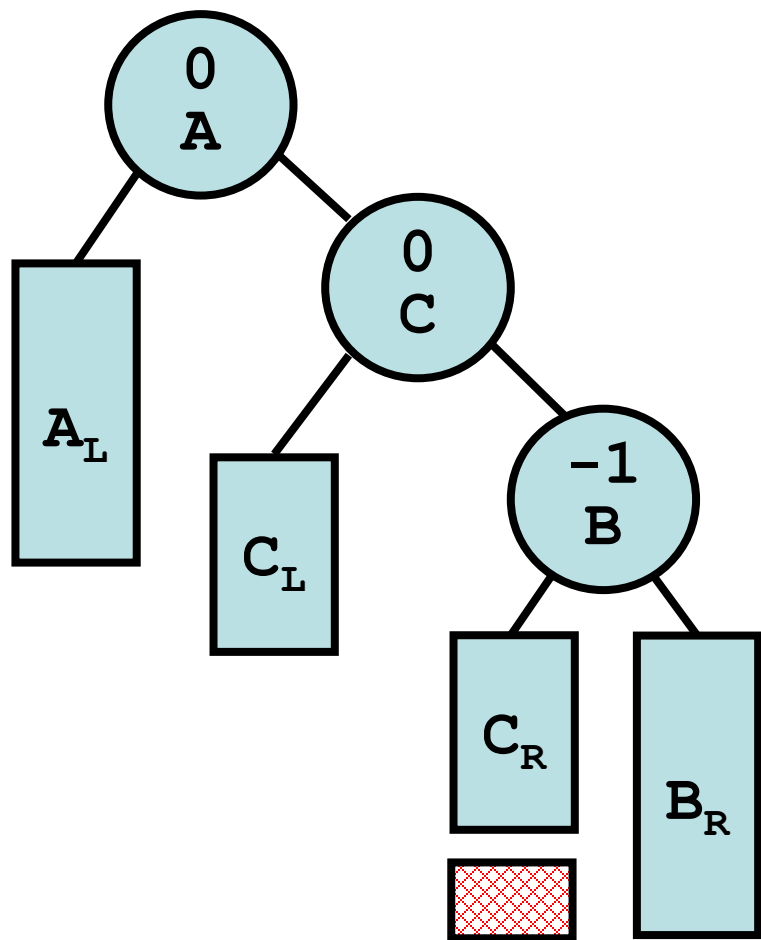
平衡二叉树

— (4) RL型

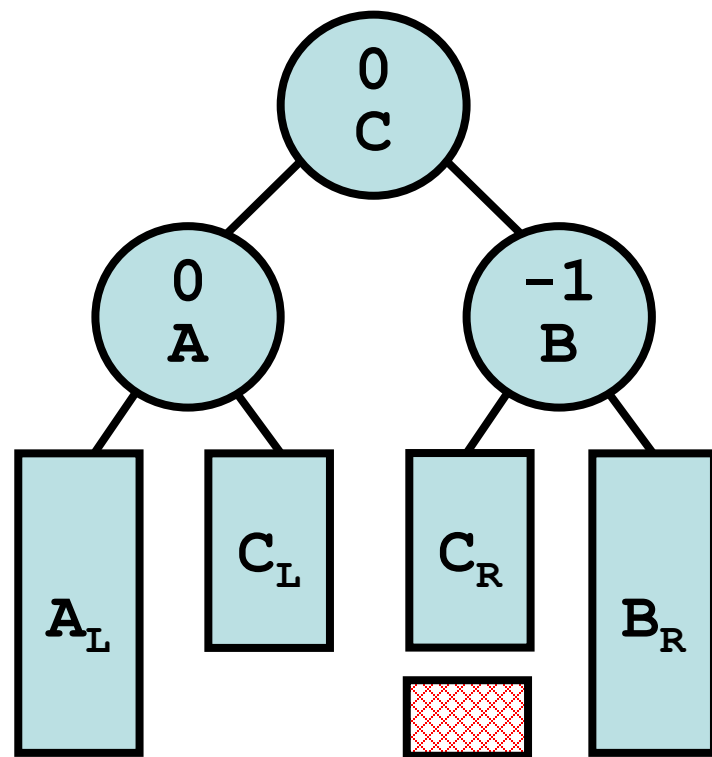


平衡二叉树

— (4) RL型

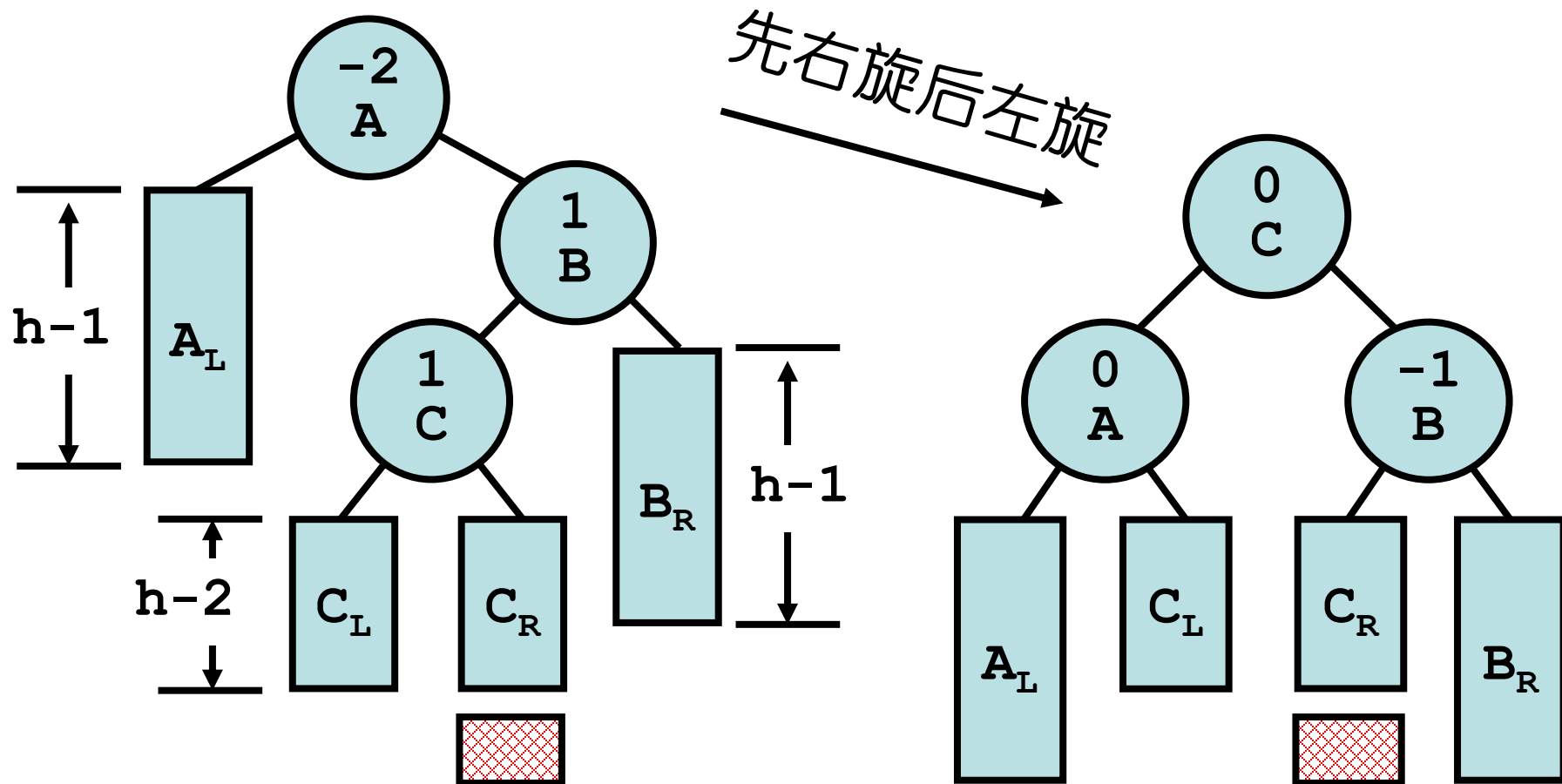


后左旋
→



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

- 某大学学生的学号有10位，比如：

0	3	4	4	0	4	0	1	0	1
									
院系编号				班级编号			序号		

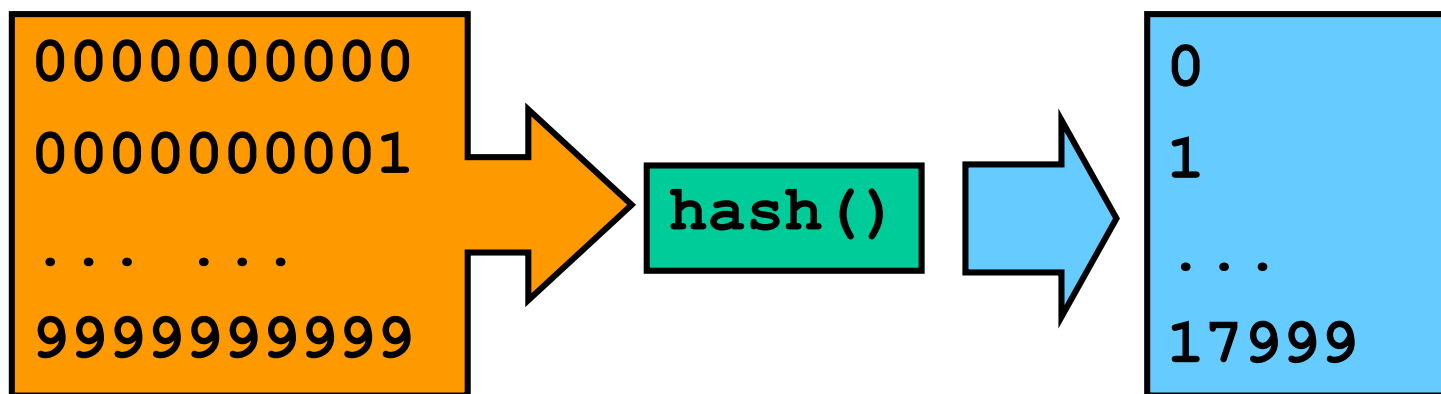
- 一共有0~9999999999，10亿个学号
- 而实际上在校生只有18000
- 所以学生花名册有18000栏就够了

哈希查找

- 设定一个hash函数

$$\text{hash}(x) = x \% 18000$$

- 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

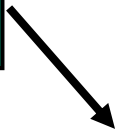
0344040101



$$\text{hash}(x) = x \% 18000$$



0344030215



0

1

...

6101

...

14215

...

17999

学号	姓名
0344040101	张三
0344030215	李四

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

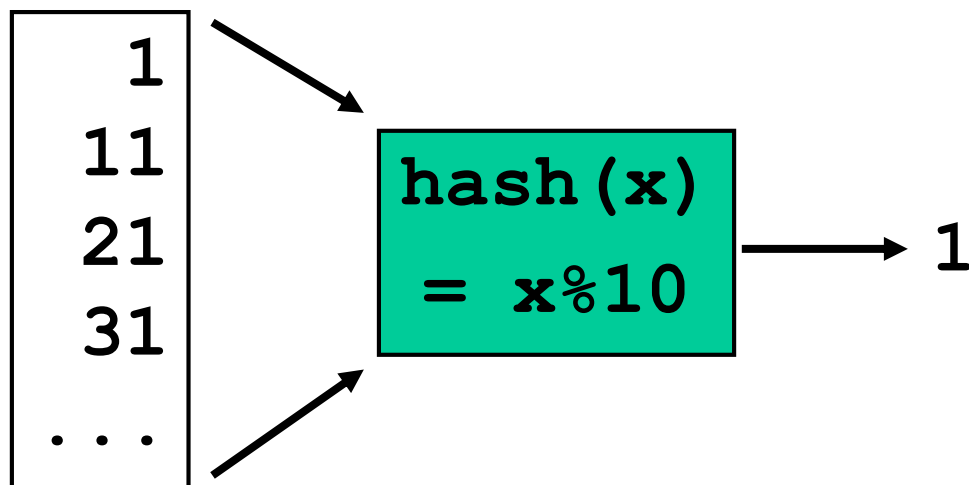
- **Hash查找**

- 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像函数。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

- 对于给定的一个关键字集合选择一个计算简单且地址分布比较均匀的Hash函数，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

- 简单，能很快计算出函数值
- 定义域 \geq 全部关键字集合
- 值域 \leq 所有的表地址集合
 - 比如学号的范围是0~999999999999
 - Hash表项的编号是0~17999
 - 所以Hash函数的定义域应该为
[0, 999999999999], 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

$$\text{Hash}(\text{key}) = a * \text{key} + b$$

- 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

$$\text{hash}(\text{key}) = \text{key} \% p$$

- 对p的要求

- $p \leq m$, 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 $\text{key} = 962148$
- Hash表大小 $m = 25$
- 取质数 $p = 23$
- Hash函数： $\text{hash}(\text{key}) = \text{key} \% p$
- 则Hash地址为：

$$\text{Hash}(962148) = 962148 \% 23 = 12$$

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：

$$\bullet \underline{325}483 \Rightarrow 325^2 = 1\underline{056}25 \Rightarrow 056$$

- 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

$$\begin{array}{r} 123 \\ 456 \\ 789 \\ + 12 \\ \hline 1380 \end{array} \Rightarrow 380$$

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

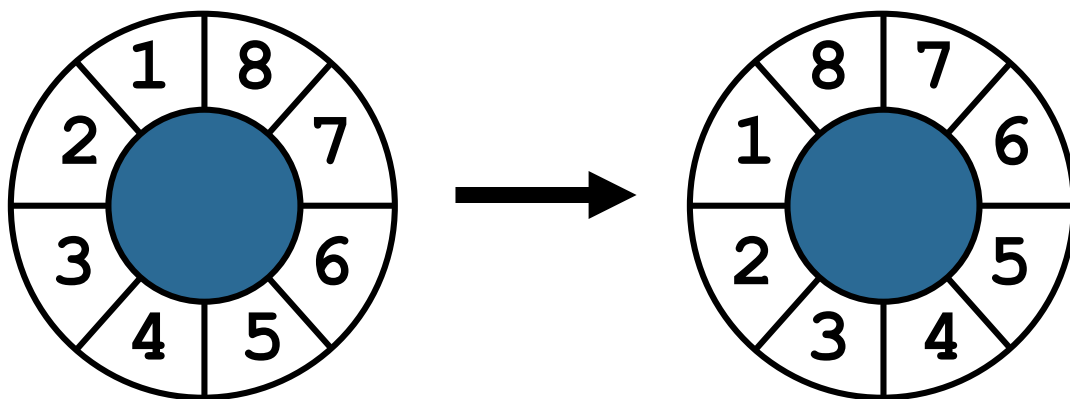
• 比如：关键字 = 123 456 789 12

$$\begin{array}{r} 123 \\ 654 \\ 789 \\ + 21 \\ \hline 1587 \end{array} \Rightarrow 587$$

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

$$\text{Hash}(\text{key}) = \text{random}(\text{key})$$

哈希查找 — Hash函数

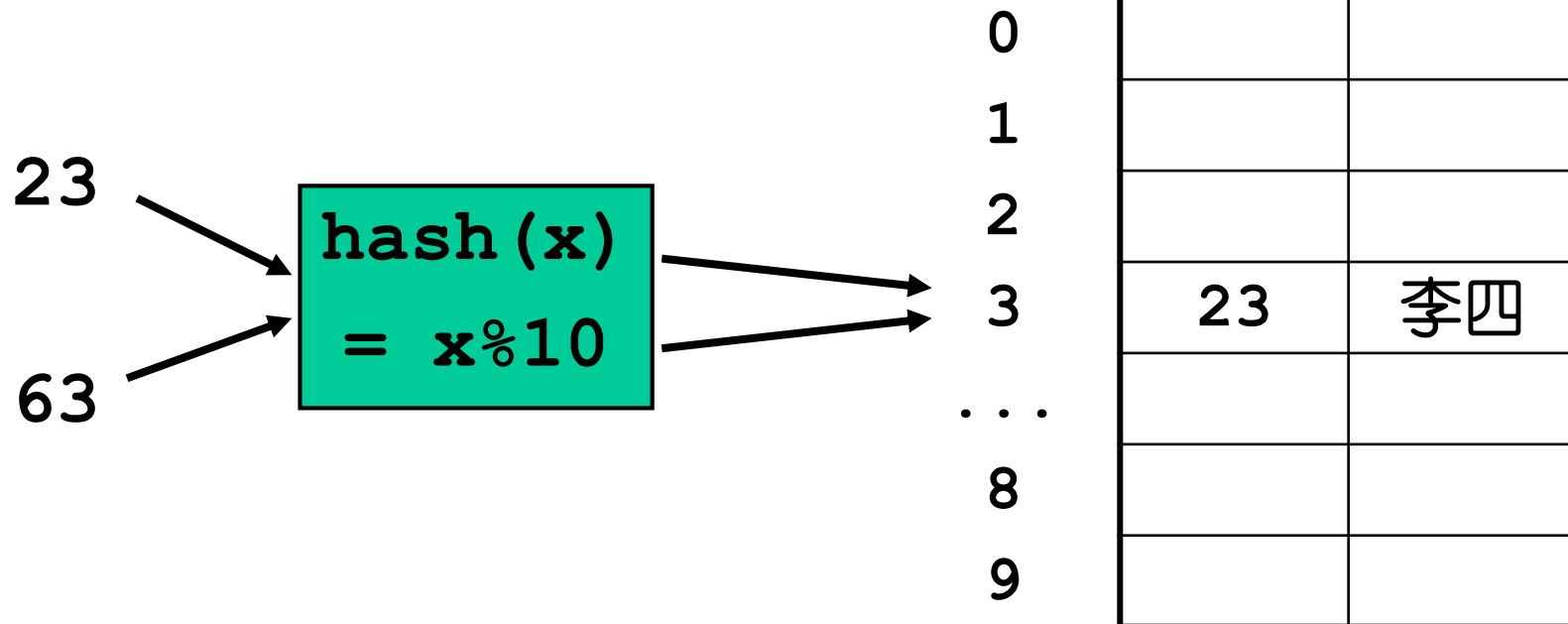
• 总结

- 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
 - $\text{Hash}(\text{key}) = \text{random}(\text{key}) \% 13$

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



哈希查找 — 冲突的解决

- 线性开放地址法

- 开放地址：如果这个地址冲突，尝试其它地址，即不“封闭”在一个地址上
- “其它”地址到底是哪个？
 - 线性探测再散列
 - 二次探测再散列
 - 伪随机探测再散列

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

哈希查找 — 冲突的解决

• 线性开放地址法 之 线性探测再散列

- 原地址 H_0 发生了冲突
- 则第 i 次冲突的探测地址

$$H_i = (H_0 + i) \% m$$

- 例如：

- Hash函数为： $H(\text{key}) = \text{key} \% 11$

- 插入关键字 $\text{key} = 60$

- $\text{Hash}(\text{key}) = 60 \% 11 = 5$ ，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 $\text{key} = 17$

– $\text{Hash}(\text{key}) = 17 \% 11 = 6$

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

哈希查找 — 冲突的解决

– 插入 $\text{key} = 29$

– $\text{Hash}(\text{key}) = 29 \% 11 = 7$

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

哈希查找 — 冲突的解决

- 插入 $\text{key} = 38$
- $\text{Hash}(\text{key}) = 38 \% 11 = 5$
- 冲突!
- 看看第6 ($5+1$) 个单元是否空着? 冲突
- 看看第7 ($5+2$) 个单元是否空着? 冲突
- 看看第8 ($5+3$) 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 $\text{key} = 51$
- $\text{Hash}(\text{key}) = 51 \% 11 = 8$
- 冲突! “鸠占雀巢”
- 看看第9 ($8+1$) 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 $\text{key} = 21$
- $\text{Hash}(\text{key}) = 21 \% 11 = 10$

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 $\text{key} = 16$
- $\text{Hash}(\text{key}) = 16 \% 11 = 5$
- 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

- 线性开放地址法 之 二次探测再散列

- 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 H_0

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

- 但是 H_0 发生了冲突
- 则下一次尝试的地址为：

$$H_i = (H_0 \pm i^2) \% m$$

i 为冲突的次数

m 为Hash表大小

$$m = 4k+3, \quad k=0, 1, 2, \dots$$

哈希查找 — 冲突的解决

— 例：

- 插入 **key** = 38
- $H_0 = 38 \% 11 = 5$
- 冲突！
- $H_1 = (H_0 + 1^2) \% 11 = 6$ ，仍然冲突
- $H_2 = (H_0 + 2^2) \% 11 = 9$

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

哈希查找 — 冲突的解决

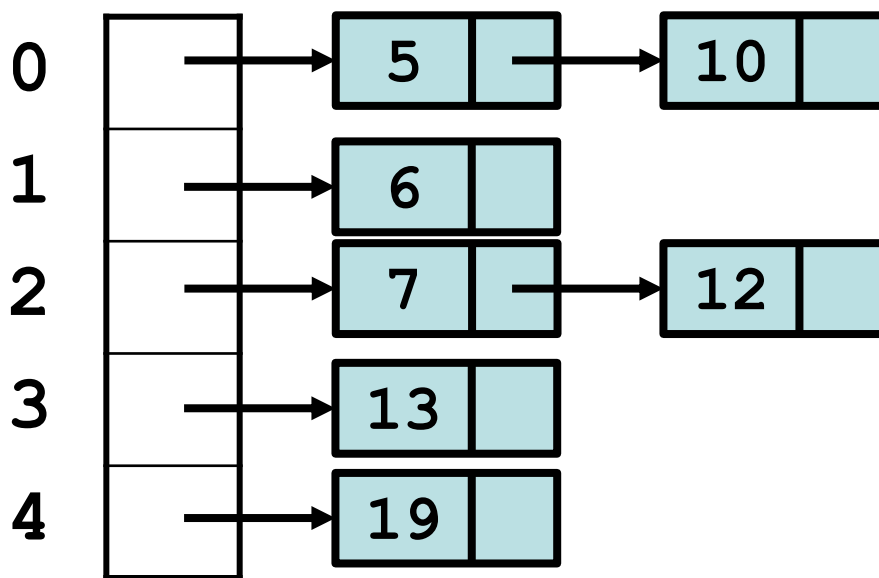
- 再哈希法

- 若原地址 $H_0 = \text{Hash}_0(\text{key})$ 冲突
- 则下一个地址 $H_i = \text{Hash}_i(\text{key})$
- 即换一个哈希函数试试

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 $\text{Hash}(\text{key}) = \text{key} \% 5$



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 $H(\text{key}) = \text{key} \% 13$
- 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：平均查找长度ASL
- 线性查找表
 - 线性查找：顺序表或链表 $O(n)$
 - 折半查找：有序表 $O(\log n)$
 - 分块查找：索引顺序表 $O(s+n/s)$
- 二叉查找树
 - 二叉查找树、平衡二叉树 $O(\log n)$
- 散列表（哈希表）
 - 哈希查找（哈希函数、冲突解决方法（开放、链地址）） $O(\text{冲突次数})$

作业和参考

- 作业

- 习题集P57:

- 9.19

- 9.31

- 9.33

- 参考

- <https://gist.github.com/tonious/1377667>

- <http://pumpkinprogrammer.com/2014/06/21/c-tutorial-intro-to-hash-tables/>

附：链式Hash表

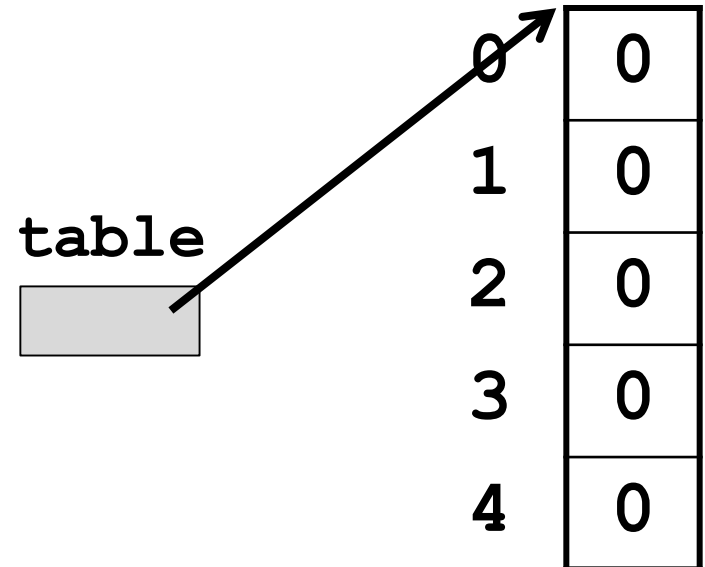
```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 119
typedef int KeyType;
typedef struct{
    KeyType key;
    double score;
}ElemType;
```

```
typedef struct h_lnode{
    ElemType data;
    struct h_lnode *next;
}HLNode;
```

```
typedef struct{
    // int *array = (int *) malloc(100 * sizeof(int))
    HLNode* *table; // HLNode* table[100]
    int size;
}HashTable;
```

附：链式Hash表

```
bool InitHashTable( HashTable &hashT){  
    hashT.table = (HLNode* *)  
                    malloc(SIZE*sizeof(HLNode*)) ;  
    if (!hashT.table) return false;  
  
    hashT.size = SIZE;  
    for (int i = 0; i < hashT.size; i++)  
        hashT.table[i] = 0;  
    return true;  
}
```



附：链式Hash表

```
int Hash(KeyType key){ return key%SIZE;}

bool InsertHashTable( HashTable &hashT,
                      KeyType key, ElemType e){
    HLNNode* s = (HLNNode* )malloc(sizeof(HLNNode));
    s->data = e;

    int hash_address = Hash(key);

    //新结点插入在hashT.table[hash_address]所指示的链表的最前面
    s->next = hashT.table[hash_address];
    hashT.table[hash_address] = s;
    return true;
}
```

附：链式Hash表

```
bool FindHashTable( HashTable &hashT,  
                    KeyType key, ElemType &e) {  
    int hash_address = Hash(key);  
  
    HLNode* p = hashT.table[hash_address];  
    while (p) {  
        if (p->data.key == key) {  
            e = p->data; return true;  
        }  
        p = p->next;  
    }  
    return false;  
}
```

附：链式Hash表

```
void HashTable_test(){
    HashTable hT; ElemType e; KeyType key;
    InitHashTable(hT);
    e.key = 2134; e.score = 10.5; InsertHashTable(hT,e.key, e);
    e.key = 34; e.score = 20.5; InsertHashTable(hT, e.key, e);
    e.key = 25; e.score = 30.5; InsertHashTable(hT, e.key, e);

    key = 20;
    if (FindHashTable(hT, key, e)) printf("%f\n", e.score);
    else printf("can't find the key%d\n", key);
    key = 25;
    if (FindHashTable(hT, key, e)) printf("%f\n", e.score);
    else printf("can't find the key%d\n", key);
}
```