第9章 查找

9.1 查找的概念

9.2 静态查找表

- 顺序表的查找
- 有序表的查找
- 索引顺序表的查找

9.3 动态查找表

- 二叉排序树
- 平衡二叉树
- B-树与B+树
- 键树

9.4 哈希表

9.1 查找的概念

- **查找表** 是由同一类型的数据元素(或记录)构成的集合，由于“集合”中的数据元素之间存在着松散的关系，因此查找表是一种应用灵活的数据结构。
- **对查找表的操作:**
  - 查询某个“特定的”数据元素是否在查找表中；
  - 检索某个“特定的”数据元素的各种属性；
  - 在查找表中插入一个数据元素；
  - 从查找表中删除某个数据元素。

9.1 查找的概念

- 查找表的分类：
  - 静态查找表：
    - 仅作查询或检索操作的查找表。
  - 动态查找表
    - 在查找表**插入**中查找不存在的数据元素，或者从查找表中**删除**已存在的某个数据元素，此类表为动态查找表。
- 关键字
  - 是数据元素（或记录）中某个或某几个数据项的值，用以标识（识别）一个数据元素（或记录）。若此关键字可以识别唯一的一个数据元素（或记录），则称之为“**主关键字**”。若此关键字能识别若干记录，则称之为“**次关键字**”。

9.1 查找的概念

□查找

- 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）。
- 若查找表中存在这样一个记录，则称“**查找成功**”，查找结果：给出整个记录的信息，或指示该记录在查找表中的位置；
- 否则称“**查找不成功**”，查找结果：给出“空记录”或“空指针”。

9.1 查找的概念

□如何进行查找?

- ■查找的方法取决于查找表的结构。
- ■由于查找表中的数据元素之间不存在明显的组织规律，因此不便于查找。
- ■为了提高查找的效率，需要在查找表中的元素之间人为地附加某种确定的关系，换句话说，用另外一种结构来表示查找表。

9.1 查找的概念

- 查找方法评价
  - 查找速度
  - 占用存储空间
  - 算法本身复杂程度
  - **平均查找长度ASL**：为确定记录在表中的位置，需和给定值进行比较的关键字的个数的期望值叫查找算法的ASL。
- 对含有 n 个记录的表，$ASL = \sum_{i=1}^{n} p_i c_i$
  - 其中，p_i 为查找表中第 i 个元素的概率，$\sum_{i=1}^{n} p_i = 1$
  - c_i 为查找表中第 i 个元素所需比较次数。

9.2.1 静态查找表—顺序表的查找

9.2.1 顺序表的查找

以顺序表表示静态查找表。顺序存储结构如下：

```
typedef int KeyType;      // 整型
或
typedef char *KeyType;    // 字符串型
```

```
typedef  struct {
    KeyType key;          // 关键字
    …                     // 其他域
} ElemType;
```

```
typedef  struct {
    // 数据元素存储空间基址，0号单元留空
    ElemType *elem;
    int      length;      // 表的长度
} SSTable;
```

9.2.1 静态查找表—顺序表的查找

- **查找过程：**从表的一端开始逐个进行记录的关键字和给定值的比较。

找64

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

监视哨

i i

比较次数=5

查询元素	比较次数
查询第 n 个数据元素	1
查询第 n - 1 个数据元素	2
······	
查找第 i 个数据元素	n – i + 1
······	
查找第 1 个数据元素	n
查找失败	n + 1

9.2.1 顺序表的查找—算法描述

```
int Search_Seq(SSTable ST, KeyType key) {
    /* 在顺序表ST中顺序查找其关键字等于key的数据元素。若找到，则函数值为该元素在表中的位置，否则为0。*/
    ST.elem[0].key = key;    // 设置哨兵
    for (i = ST.length; ST.elem[i].key != key; --i) ;    // 从后往前找
    return i;  // 找不到时，i 为0

} // Search_Seq
```

```
int Search_Seq(SSTable ST, KeyType key) {



    for (i = 1; i <= ST.length; ++i)
        if (ST.elem[i].key == key)
            return i;
    return 0;
}
```

9.2.1 顺序表的查找—性能分析

- 查找性能分析

$$ASL = \sum_{i=1}^{n} p_i c_i$$

- 对顺序表而言，$c_i = n - i + 1$
- 在等概率查找的情况下，$p_i = \frac{1}{n}$
- **顺序表查找成功时的平均查找长度为：**

$$ASL_{SS} = \frac{1}{n}\sum_{i=1}^{n}(n - i + 1) = \frac{n+1}{2}$$

9.2.1 顺序表的查找—性能分析

- 如果**考虑上查找不成功**的情况，且查找成功与不成功的可能性相同，对每个记录的查找概率也相等，即$p_i=\frac{1}{2n}$。
- 此时顺序查找的平均查找长度为

$$ASL_{SS}=\frac{1}{2n}\sum_{i=1}^{n}(n-i+1)+\frac{1}{2}(n+1)=\frac{3}{4}(n+1)$$

9.2.1 顺序表的查找—性能分析

- 在**不等概率查找**的情况下，ASL在$P_n \geq P_{n-1} \geq \cdots \geq P_1$时取极小值。表中记录按查找概率由小到大重新排列，以提高查找效率。
- 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。**

9.2.2 有序表的查找

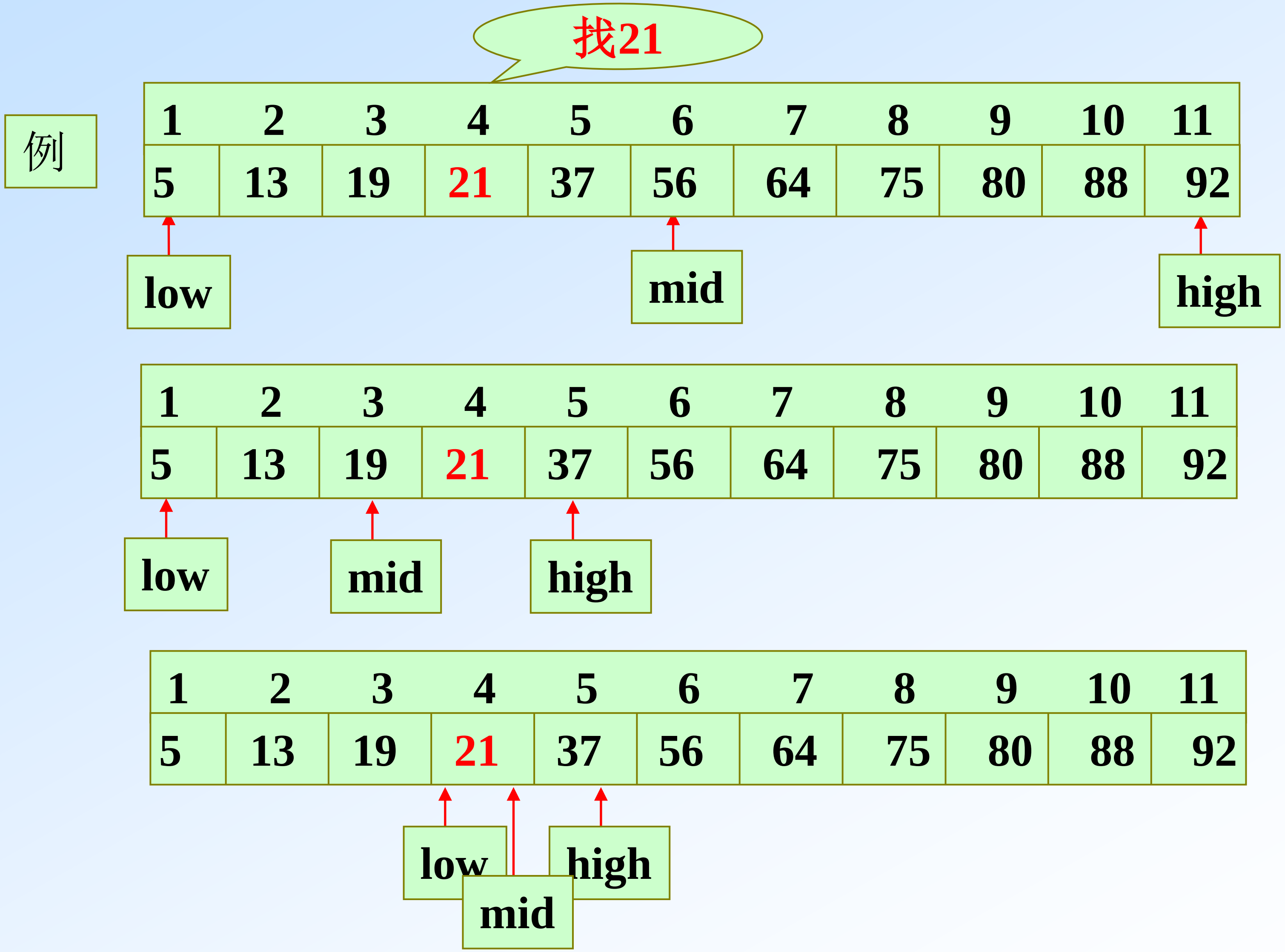
- 顺序表的查找算法简单，但平均查找长度较大，不适用于表长较大的查找表。
- 若以有序表表示静态查找表，则查找过程可以基于“折半”进行。
  - ST.elem[i].key <= ST.elem[i + 1].key, i = 1, 2, …, n – 1

  或
  - ST.elem[i].key >= ST.elem[i + 1].key, i = 1, 2, …, n – 1
- **折半查找**
  - 查找过程：每次将待查记录所在区间缩小一半。
  - 适用条件：采用顺序存储结构的有序表。

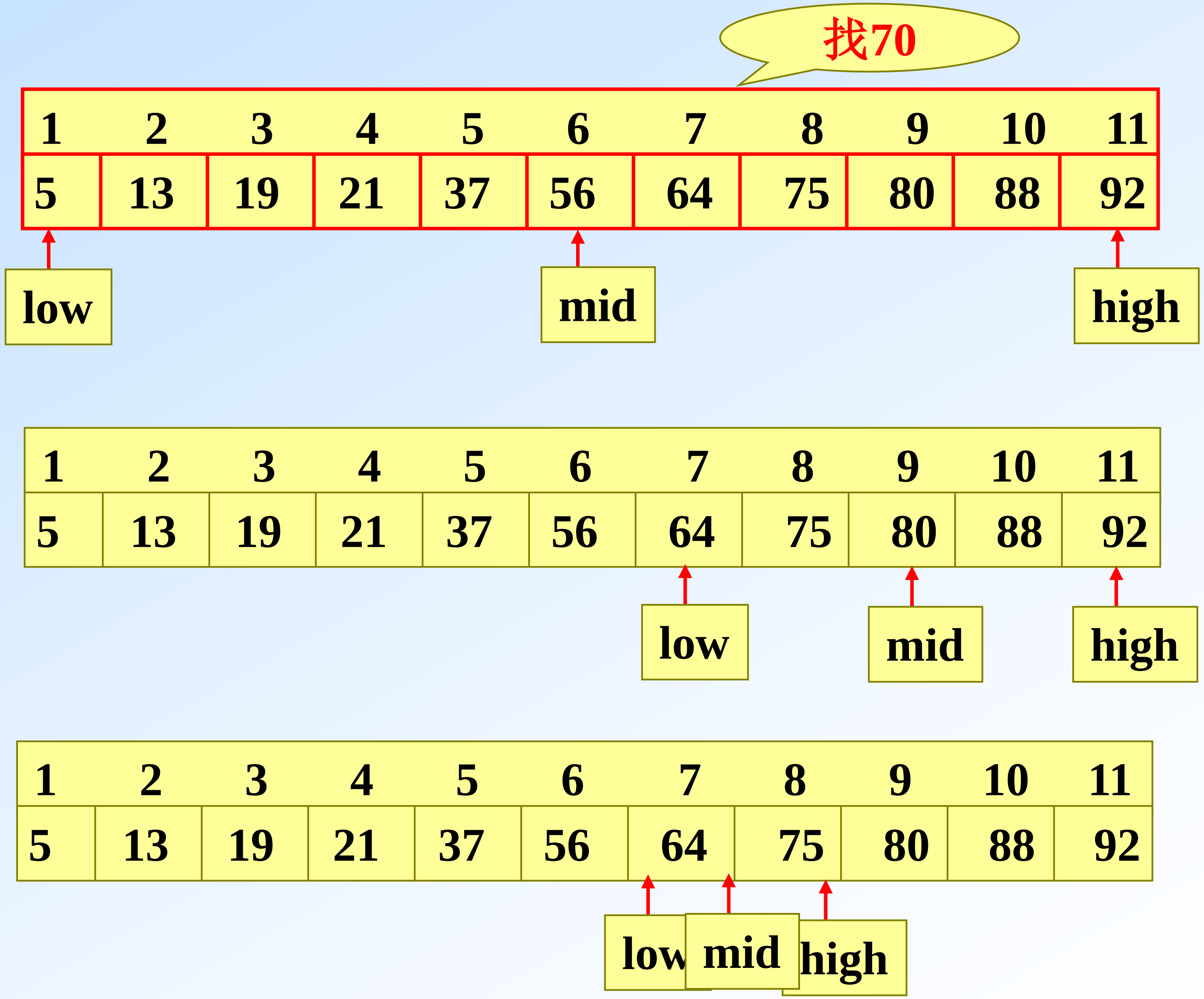
9.2.2 有序表的查找—折半查找算法实现

1. 设表长为n， low、high和mid分别指向待查元素所在区间的下界、上界和中点，key 为给定值。
2. 初始时，令 low = 1, high = n
3. mid = ⌊(low+high)/2⌋
4. 让 key 与 mid 指向的记录比较
   - 若key == r[mid].key，查找成功
   - 若key < r[mid].key，则 high = mid – 1
   - 若key > r[mid].key，则 low = mid + 1
5. 重复步骤3和4，直至low > high时，查找失败。

9.2.2 折半查找—key = 21 的查找过程



9.2.2 折半查找—key = 70 的查找过程



9.2.2 折半查找—key = 70 的查找过程

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

low high
mid

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

high low

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

9.2.2 折半查找算法

```
int Search_Bin ( SSTable ST, KeyType key ) {
   low = 1;  high = ST.length;          // 置区间初值
   while (low <= high) {
      mid = (low + high) / 2;
      if (key == ST.elem[mid].key) return  mid;  // 找到待查元素
      else if (key < ST.elem[mid].key)
         high = mid - 1;                 // 继续在前半区间进行查找
      else low = mid + 1;                // 继续在后半区间进行查找
   }
   return 0;                             // 顺序表中不存在待查元素
}  // end Search_Bin
```

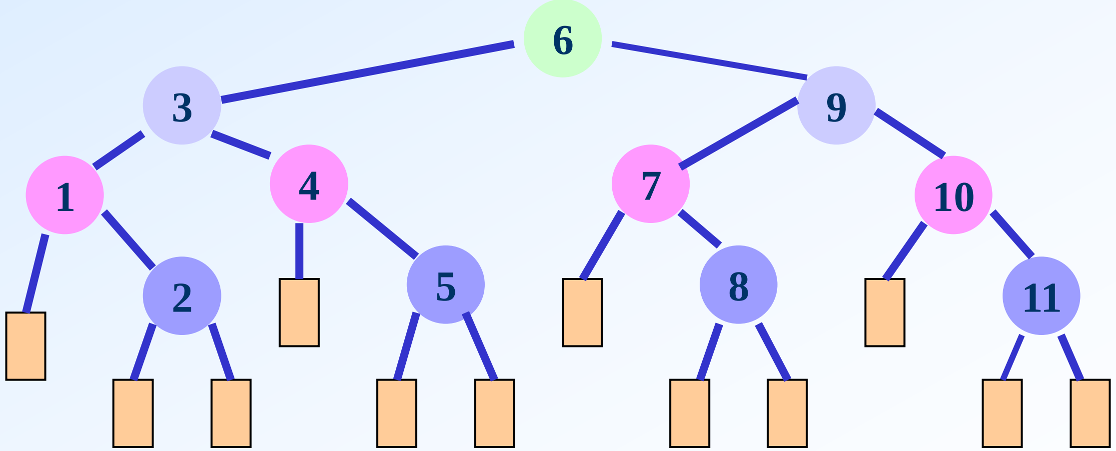
9.2.2 折半查找算法性能分析

- 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$\lfloor \log_2 n \rfloor+1$
- 折半查找法在查找过程中进行的比较次数最多不超过$\lfloor \log_2 n \rfloor+1$

i	1	2	3	4	5	6	7	8	9	10	11
C_i	3	4	2	3	4	1	3	4	2	3	4

有11个元素的表的例子：n=11

ASL=3



9.2.2 折半查找算法性能分析

- 假设 有序表的长度 $n = 2^h - 1$（则 $h = \log_2(n + 1)$），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**

$$ASL_{BS} = \frac{1}{n}\sum_{j=1}^{h} j \times 2^{j-1} = \frac{n+1}{n}\log_2(n+1) - 1 \approx \log_2 n - 1$$

9.2.2 折半查找算法小结

- 折半查找的效率一般比顺序查找高。
- 折半查找适用于有序表，并且以顺序存储结构存储。

	顺序表	有序表
表的特性	可以为无序	有序
存储结构	顺序/链式存储	顺序存储
插删操作	易于进行	需移动元素
ASL的值	$O(n)$	$O(\log_2 n)$

9.2.3 索引顺序表

- 在建立顺序表的同时，建立一个索引项，包括两项：**关键字项和指针项**。索引表按关键字有序，顺序表则为分块有序

顺序表

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	……
17	08	21	19	14	31	33	22	25	40	52	61	78	46	74	……

索引表

21 1	40 6	78 11	……

索引顺序表 = 索引 + 顺序表

9.2.3 索引顺序表

- **索引顺序查找，又称分块查找**
  - 查找过程：将表分成几块，块内无序，块间有序；先确定待查记录所在块，再在块内查找
  - 适用条件：分块有序表
  - 算法实现：
    - 用数组存放待查记录，每个数据元素至少含有关键字域
    - 建立索引表，每个索引表结点含有最大关键字域和指向本块第一个结点的指针

9.2.3 索引顺序表

□索引顺序表查找关键字25

1	6	11	……
21	40	78	……

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	……
17	08	21	19	14	31	33	22	**25**	40	52	61	78	46	74	……

9.2.3 索引顺序表—分块查找算法分析

- $ASL_{BS} = L_B + L_W$
  - 其中，L_B为查找索引表确定所在块的平均查找长度
  - L_W为在块中查找元素的平均查找长度
- 若将表长为n的表平均分为b块，每块含s个记录，并设表中每个记录的查找概率相等，则
  - 用顺序表查找确定所在块：

$$ASL_{BS} = \frac{1}{b}\sum_{j=1}^{b} j + \frac{1}{s}\sum_{i=1}^{s} i = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1$$

  - 用折半查找确定所在块：

$$ASL_{BS} \approx \log_2\left(\frac{n}{s} + 1\right) + \frac{s}{2}$$

9.2 静态查找方法比较

- 从查找性能看，最好情况能达到O($\log_2 n$)，需要求**查找表有序**

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表 无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

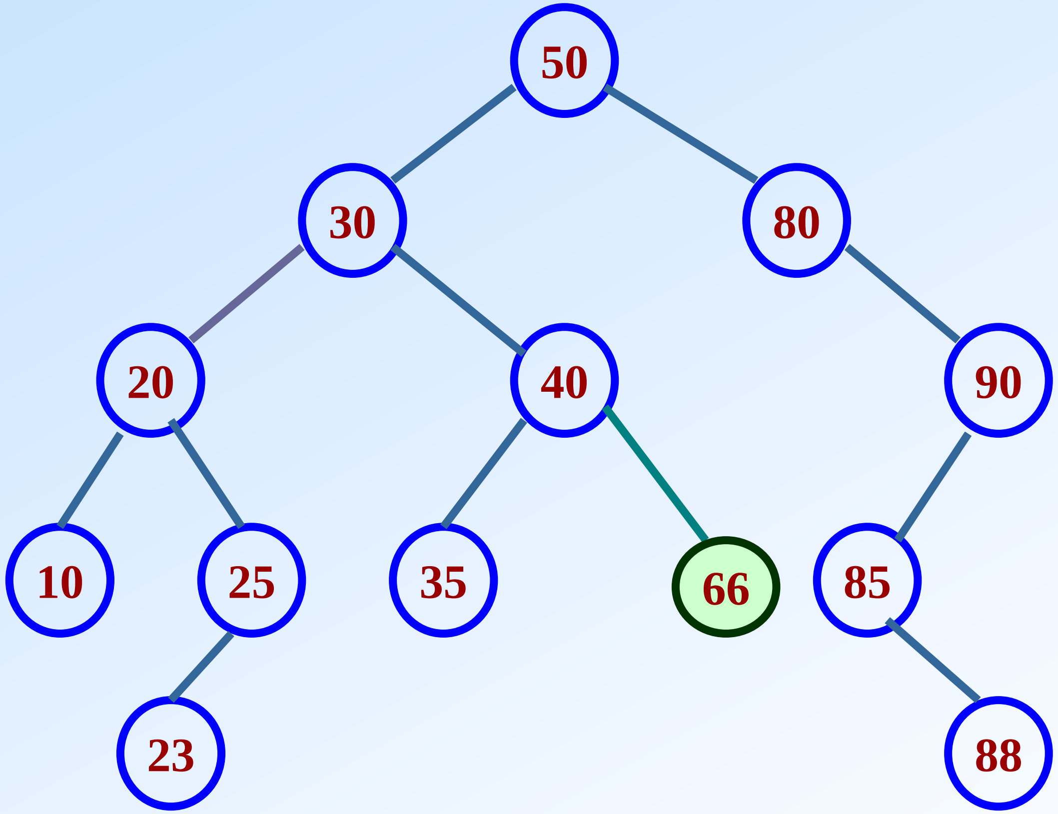
9.3 动态查找表

- **动态查找表特点**：表结构本身是在查找过程中动态生成。
  - 若表中存在其关键字等于给定值 key 的数据元素，表明查找成功；
  - 否则插入关键字等于 key 的数据元素。
- 动态查找表**表示方法**：
  - 二叉排序树
  - 平衡二叉树（AVL树）
  - B-树
  - B+树
  - 键树

9.3.1 二叉排序树

- 二叉排序树又称二叉查找树
- 递归定义
  - 二叉排序树或者是一棵空树；
  - 或者是具有如下特性的二叉树：
    - 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；
    - 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；
    - 它的左、右子树也都分别是二叉排序树。

9.3.1 二叉排序树



不是二叉排序树

9.3.1 二叉排序树

- 存储结构
  - 以二叉链表形式存储

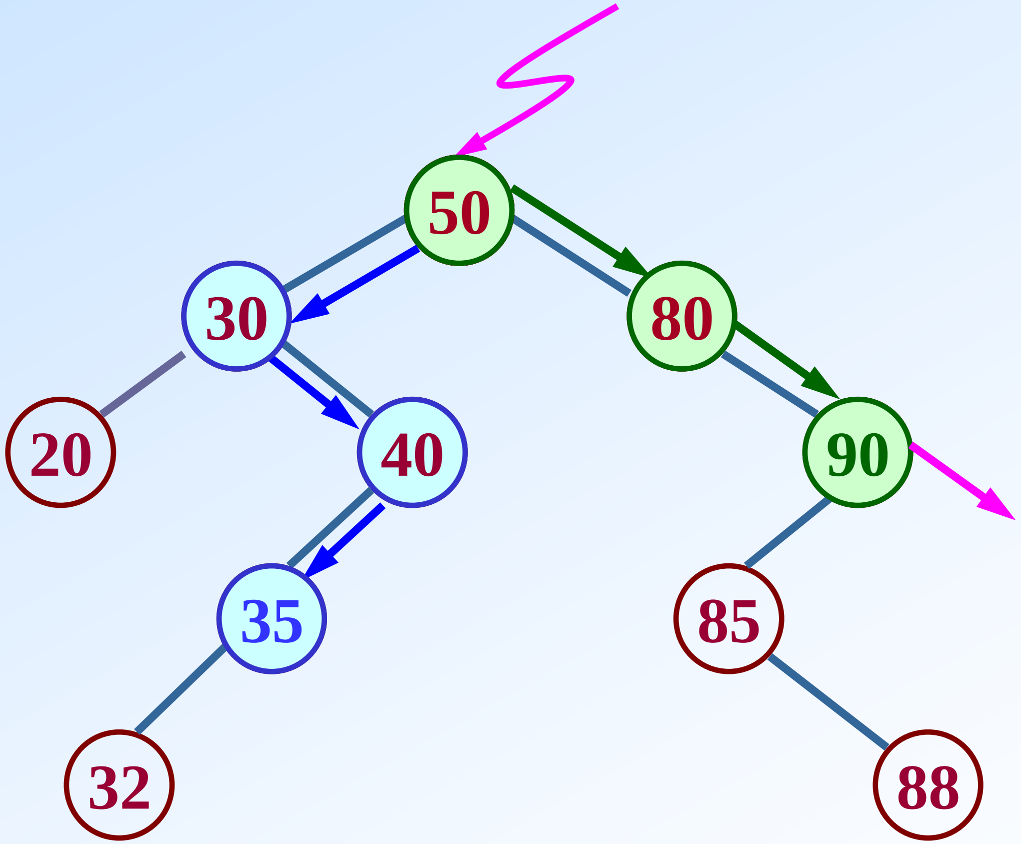
```
typedef struct BiTNode {                    // 结点结构
    TElemType data;
    struct BiTNode  *lchild, *rchild; // 左右指针
} BiTNode, *BiTree;
```

9.3.1 二叉排序树—查找算法

- 若二叉排序树为空，则查找不成功；
- 否则分 3 种情况：
  1. 若给定值等于根结点的关键字，则查找成功；
  2. 若给定值小于根结点的关键字，则继续在左子树上进行查找；
  3. 若给定值大于根结点的关键字，则继续在右子树上进行查找。

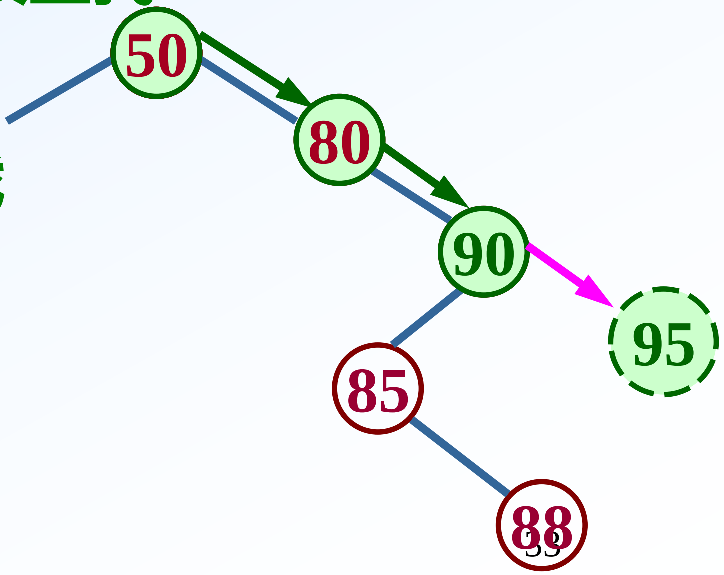
9.3.1 二叉排序树—查找实例

- 在二叉排序树中查找关键字值等于50、35、90、95



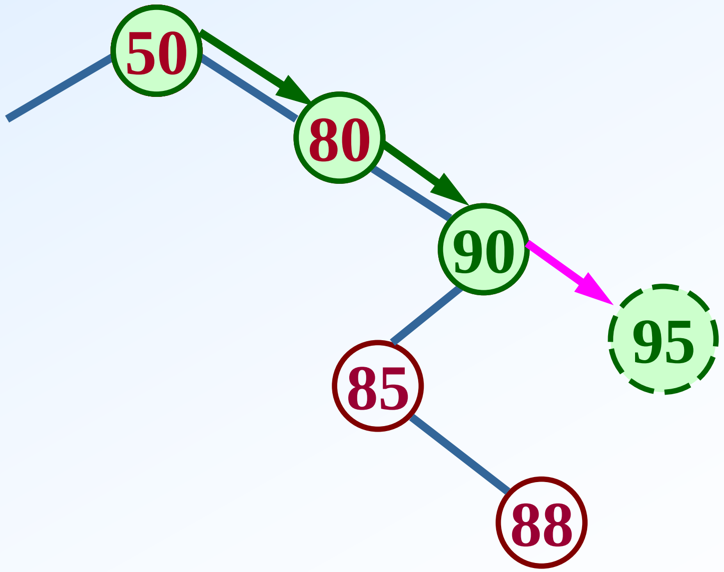
9.3.1 二叉排序树—查找算法

```
BiTree SearchBST (BiTree T, KeyType key) {
   if (!T) return NULL;                    // 查找不成功，返回空
   else if ( key == T->data.key )
      return T;                            // 查找成功，返回树根
   else if ( key < T->data.key )           // 在左子树中继续查找
      return SearchBST (T->lchild, key);
   else                                    // 在右子树中继续查找
      return SearchBST (T->rchild, key);
}
```



9.3.1 二叉排序树—查找算法（改进）

```
Status SearchBST (BiTree T, KeyType key,
                  BiTree f,        // 指向T的双亲
                  BiTree &p ) {    // 记录查找位置
    if (!T) {                      // 查找不成功
        p = f;
        return FALSE;
    }
    else if ( key == T->data.key ) {   // 查找成功
        p = T;
        return TRUE;
    }
    else if ( key < T->data.key ) return SearchBST (T->lchild, key, T, p );
    else   return SearchBST (T->rchild, key, T, p );
}
```

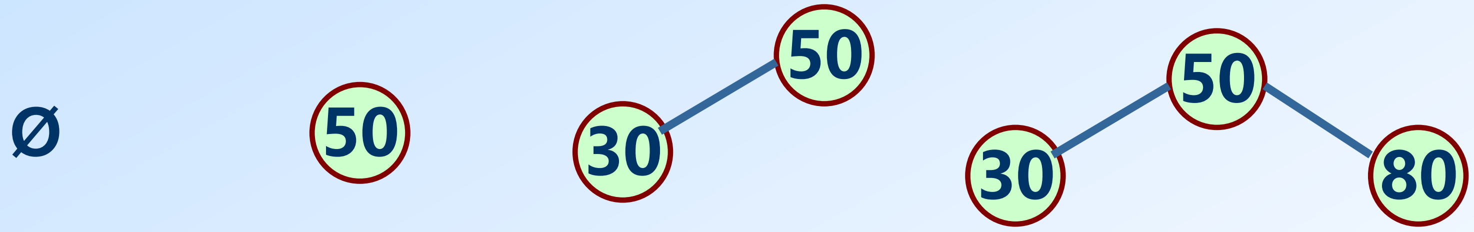


9.3.1 二叉排序树—插入

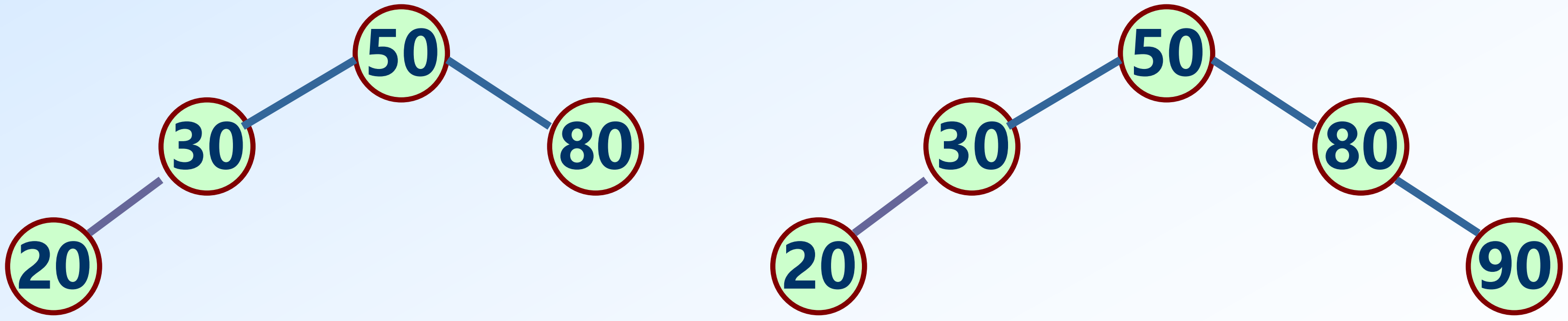
- 当查找不成功时，才进行“插入”操作；
- 在完成插入结点的操作后，仍要保持二叉排序树特性；
- 插入过程
  - 若二叉排序树为空树，则新插入的结点为新的根结点；
  - 否则，新插入的结点必为新的叶子结点，其插入位置由查找过程得到。

9.3.1 二叉排序树—插入实例

- 从空树开始，依次向二叉排序树中插入查找关键字 {50, 30, 80, 20, 90}



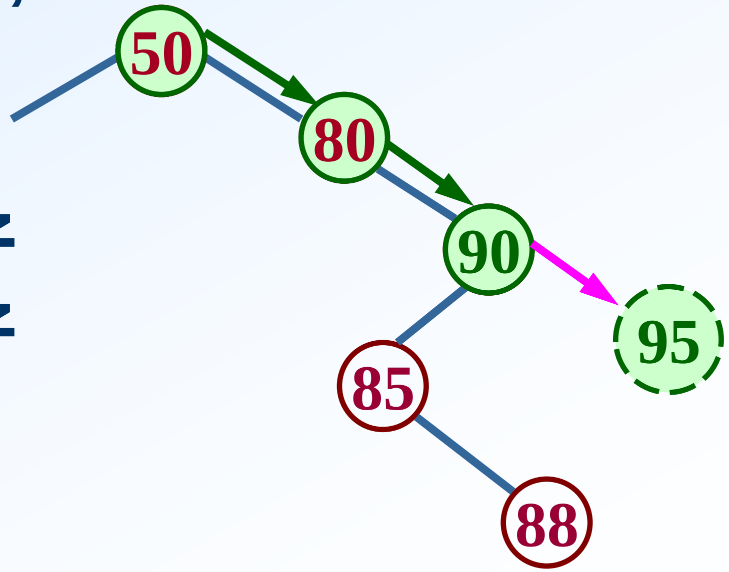
(a)空树　(b)插入50　(c)插入30　(d)插入80



(e)插入20　(f)插入90

9.3.1 二叉排序树—插入算法

```
Status InsertBST (BiTree &T, ElemType e) {
   if (!SearchBST (T, e.key, NULL, p)) {
      s = new BiTNode;             // 为新结点分配空间
      s->data = e;     s->lchild = s->rchild = NULL;
      if  (!p)  T = s;             // 插入 s 为新的根结点
      else if (e.key < p->data.key)
         p->lchild = s;            // 插入 *s 为 *p 的左孩子
      else  p->rchild = s;         // 插入 *s 为 *p 的右孩子
      return TRUE;
   }
   else return FALSE;
}  // end InsertBST
```



9.3.1 二叉排序树—插入算法小结

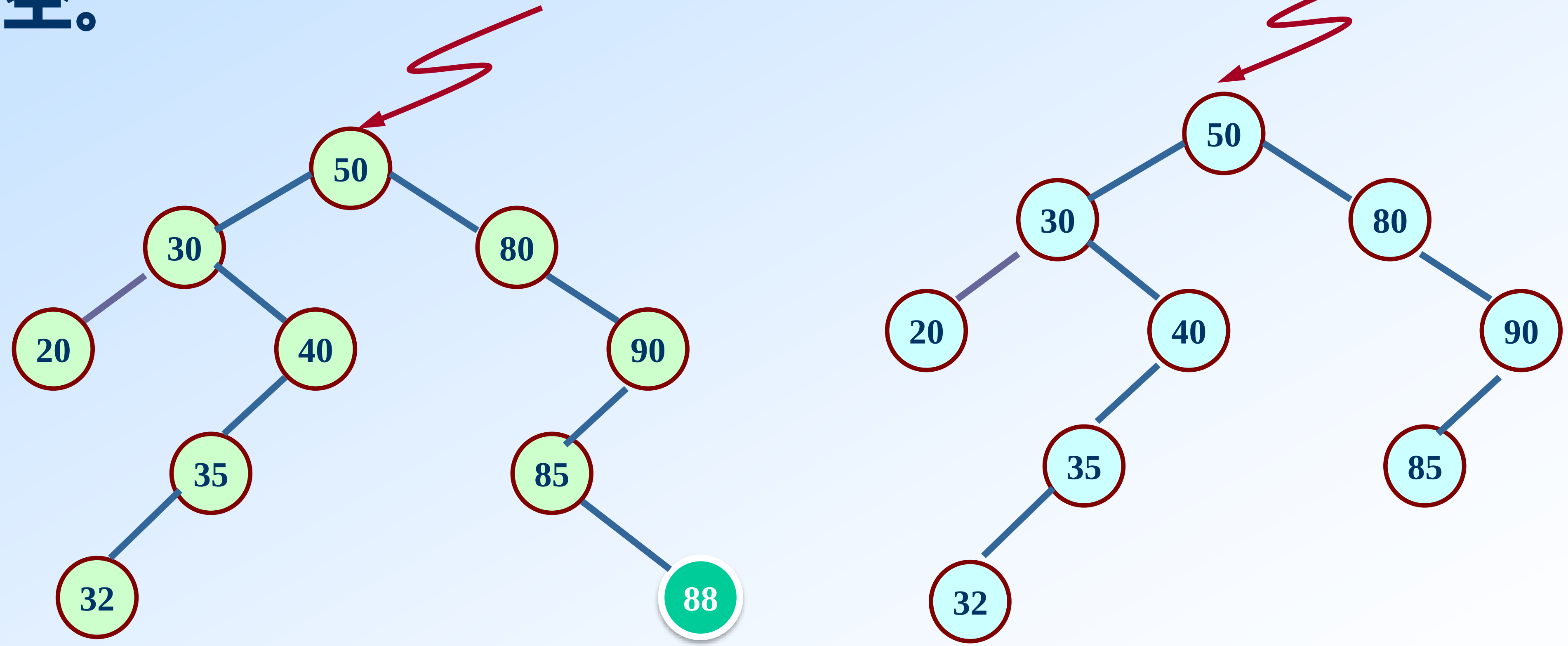
- 查找失败时才进行插入操作；
- 每次插入的新结点都是二叉排序树上的**叶子结点**；
- **插入**时不必移动其他结点，**仅需修改**某个结点的**指针**；
- 一个**无序**序列可通过**构造二叉排序树**而**变成**一个**有序**序列；
- **中序遍历**二叉排序树可得到关键字的**有序序列**。

9.3.1 二叉排序树—删除

- 当查找成功时，才进行“删除”操作。
- 在完成删除结点的操作后，仍要保持二叉排序树特性；
- 删除过程可分3种情况讨论：
  1. 被删除的结点是叶子结点；
  2. 被删除的结点只有左子树或者只有右子树；
  3. 被删除的结点既有左子树，也有右子树。

9.3.1 二叉排序树—删除算法

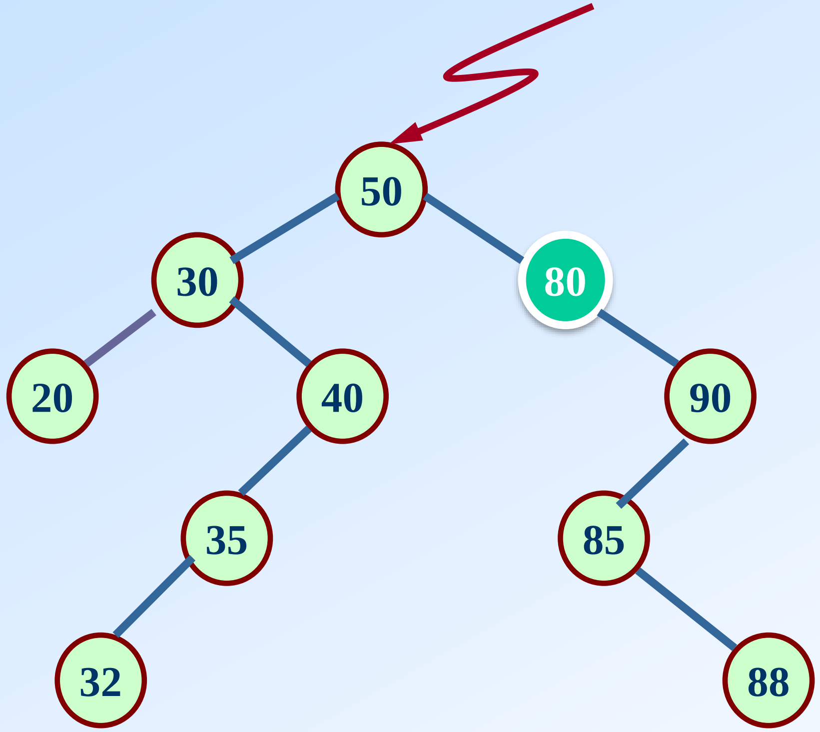
1. 被删除的结点是**叶子结点**，其双亲结点中相应指针域的值改为空。

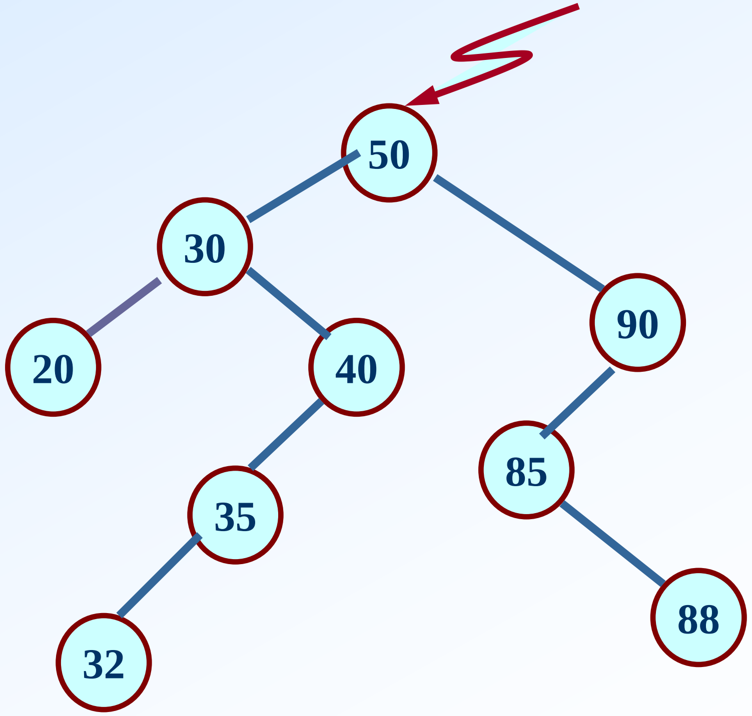


删除 Key = 88 的结点

9.3.1 二叉排序树—删除算法

2. 被删除的结点只有左子树或者只有右子树，其双亲结点的相应指针域的值改为“指向被删除结点的左子树或右子树”。

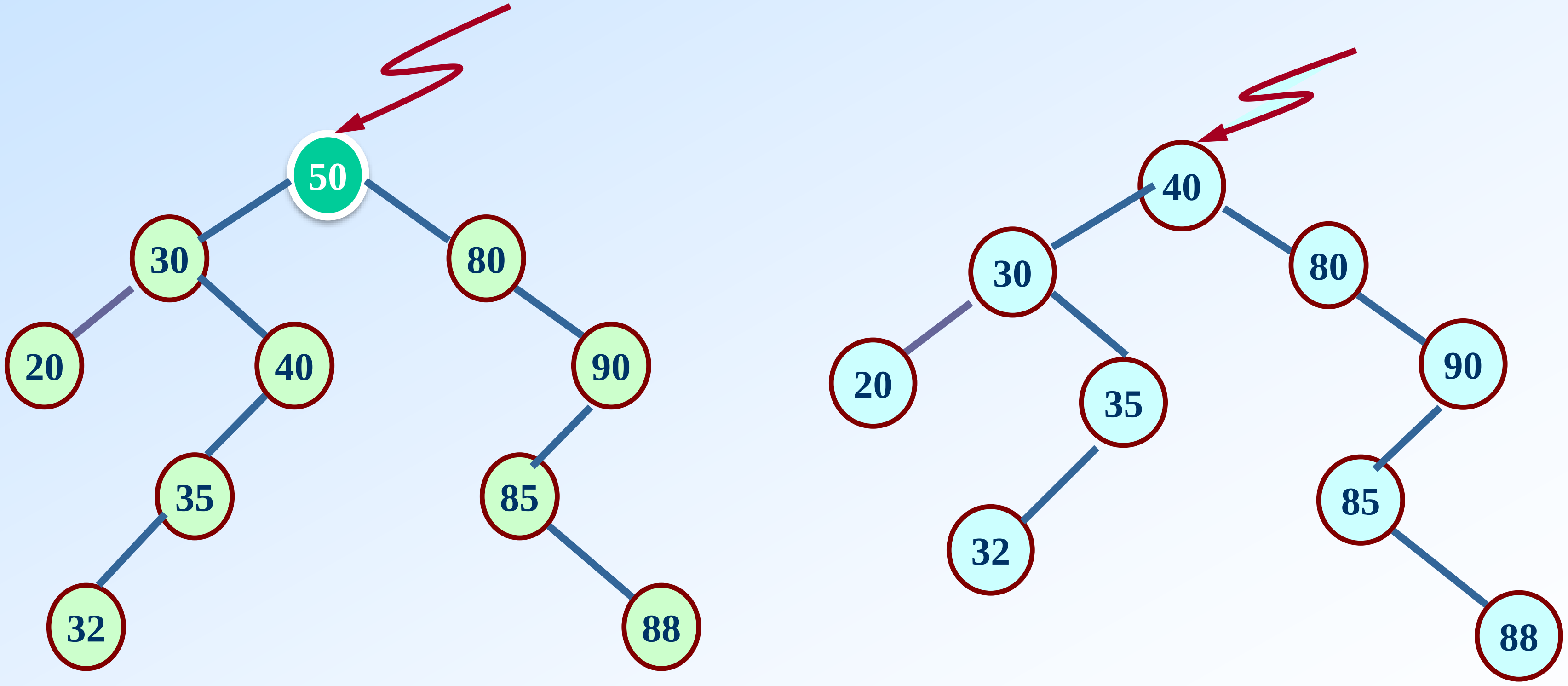




删除 Key = 80 的结点

9.3.1 二叉排序树—删除算法

3. 被删除的结点既有左子树又有右子树，以其前驱替代之，然后再删除该前驱结点。删除前驱结点一定是前2种情况之一。



删除 Key = 50 的结点

9.3.1 二叉排序树—删除算法

```
Status DeleteBST (BiTree &T,  KeyType key, BiTree &f) {
   if (!T) return FALSE;     // 不存在关键字等于 key 的数据元素
   else {
      // 找到关键字等于 key 的数据元素
      if (key == T->data.key) { Delete (T, f);   return TRUE; }
      else if (key < T->data.key)      // 继续在左子树中进行查找
         return DeleteBST (T->lchild, key, T);
      else                             // 继续在右子树中进行查找
         return DeleteBST (T->rchild, key, T);
   }      // end if
}         // DeleteBST
```

9.3.1 二叉排序树—删除算法

- 删除过程的3种情况：
  1. 被删除的结点是叶子结点；
  2. 被删除的结点只有左子树或者只有右子树；
  3. 被删除的结点既有左子树，也有右子树。

- 删除过程的3种情况：
  1. 被删除的结点右子树为空；
  2. 被删除的结点左子树为空；
  3. 被删除的结点既有左子树，也有右子树。

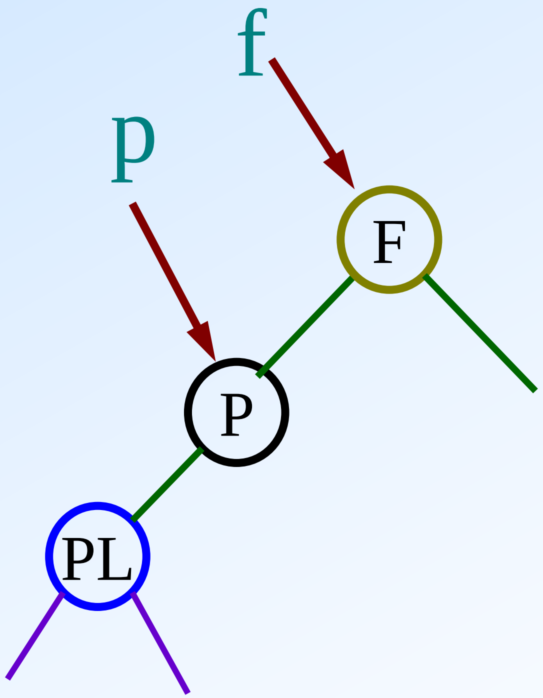
9.3.1 二叉排序树—删除算法

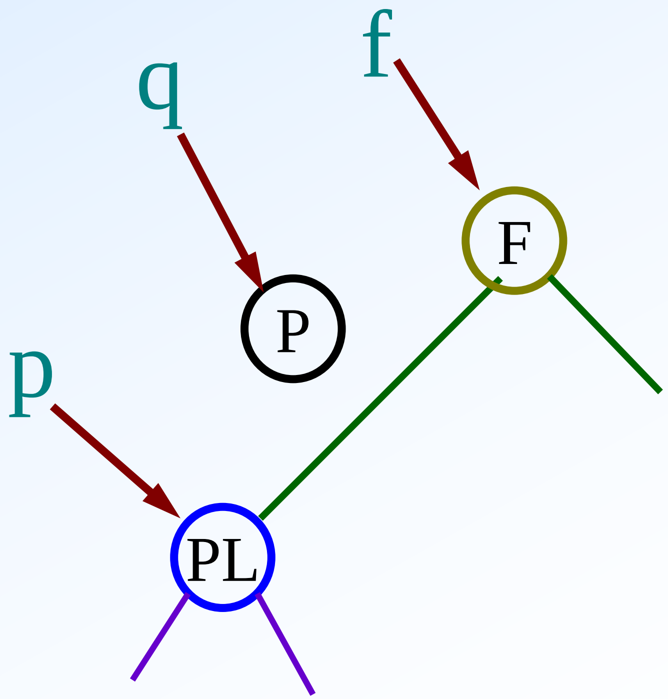
```
/* 删除操作过程Delete */
void Delete (BiTree &p, BiTree f) {
  // 从二叉排序树中删除结点 p，并重接它的左子树或右子树
  //
  if (!p->rchild)  {/* 情况1 */   }
  else if (!p->lchild) {   /* 情况2 */   }
  else {/* 情况3 */   }
} // Delete
```

9.3.1 二叉排序树—删除算法情况1

□情况1：右子树为空树，则只需重接它的左子树

```
q = p;
p = p->lchild;
if (f->lchild == q)
   f->lchild = p;
else if (f->rchild == q)
   f->rchild = p;
delete(q);
```



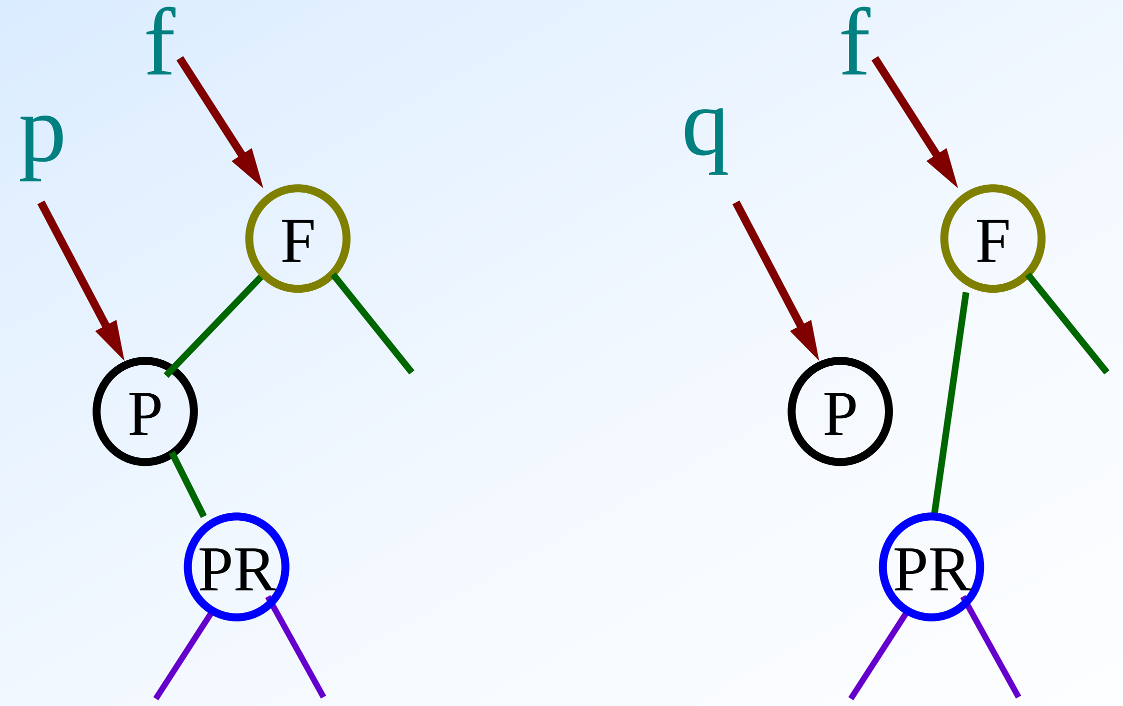


返回

9.3.1 二叉排序树—删除算法情况2

- 情况2：左子树为空树，只需重接它的右子树

```
q = p;
p = p->rchild;
if (f->lchild == q)
   f->lchild = p;
else if (f->rchild == q)
   f->rchild = p;
delete(q);
```

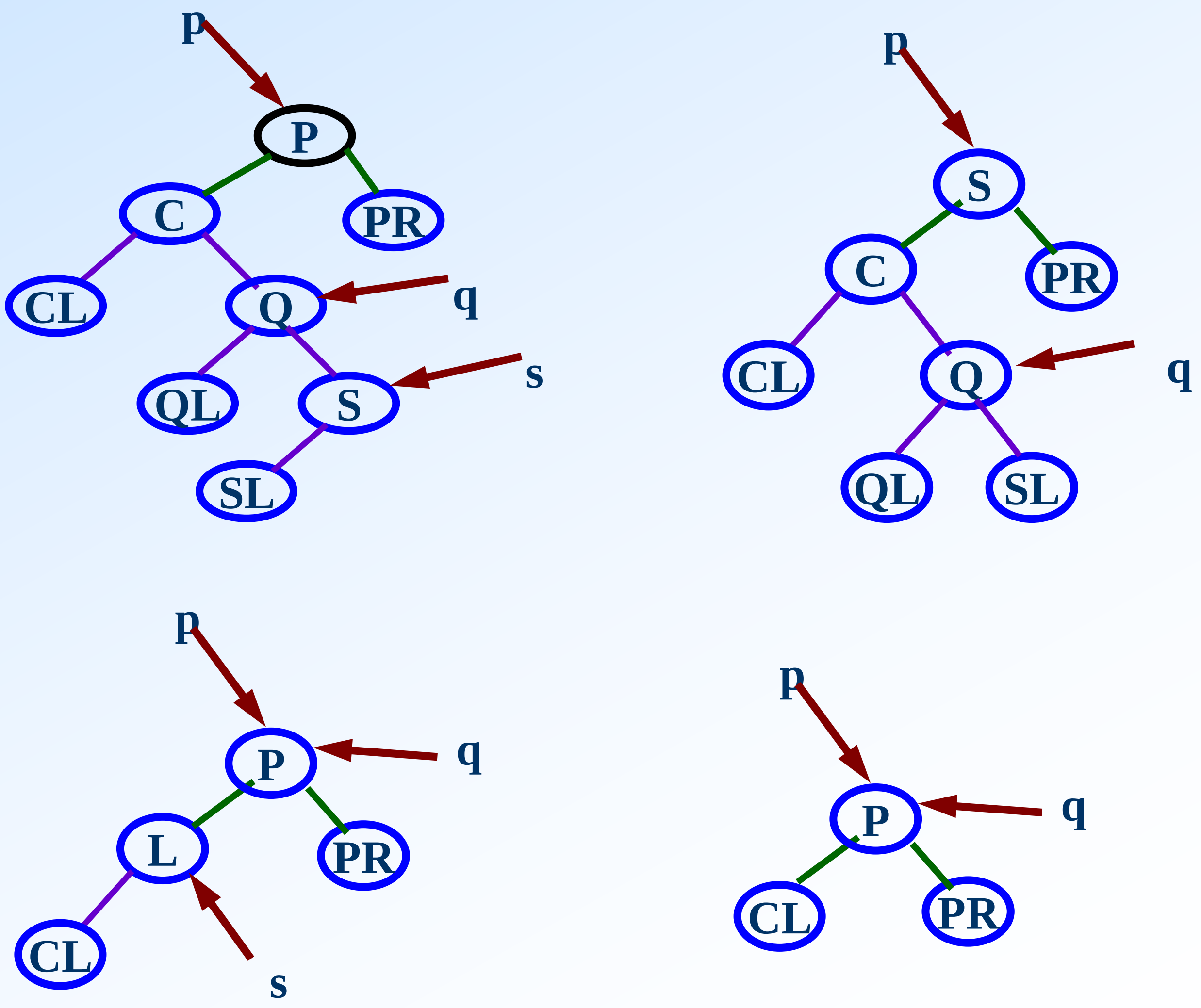


返回

9.3.1 二叉排序树—删除算法情况3

▫ 情况3：左右子树均不空

```
q = p;  s = p->lchild;
while (s->rchild) {
    q = s;    s = s->rchild;
}   // s 指向被删结点的前驱
p->data = s->data;
// 重接*q的左子树
if (q != p )
    q->rchild = s->lchild;
else
    q->lchild = s->lchild;
delete(s);
```

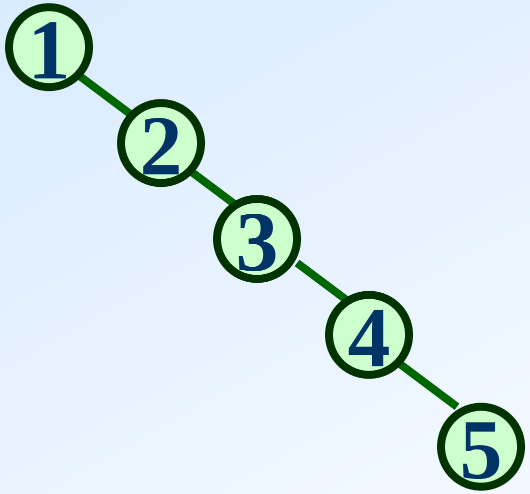


9.3.1 二叉排序树—查找算法性能分析

- 由关键字序列 1，2，3，4，5构造得到的二叉排序树，

ASL =（1+2+3+4+5）/ 5

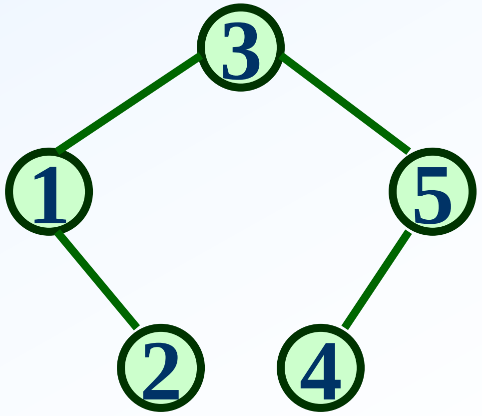
= 3



- 由关键字序列 3，1，2，5，4构造得到的二叉排序树，

ASL =（1+2+3+2+3）/ 5

= 2.2

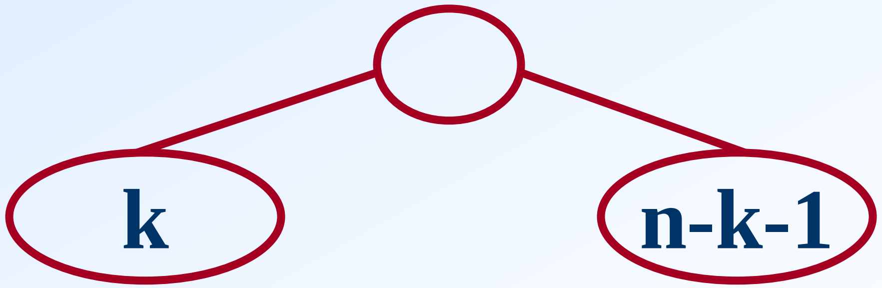


给定关键字的二叉排序树，中序遍历结果相同，然而ASL值不一定相同，甚至差别很大

9.3.1 二叉排序树—查找算法性能分析

- 平均情况
- 不失一般性，假设长度为 n 的序列中有 k 个关键字小于第一个关键字，则必有 n – k – 1 个关键字大于第一个关键字，由它构造的二叉排序树的平均查找长度是 n 和 k 的函数

$$P(n, k) \qquad (0 \le k \le n - 1)$$



9.3.1 二叉排序树—查找算法性能分析

- 假设有 n 个关键字，所有排列的可能性相同，则含 n 个关键字的二叉排序树的平均查找长度

$$ASL = P(n) = \frac{1}{n}\sum_{k=0}^{n-1} P(n,k)$$

- 在等概率查找的情况下，

$$P(n,k) = \sum_{i=1}^{n} p_i C_i = \frac{1}{n}\sum_{i=1}^{n} C_i$$

9.3.1 二叉排序树—查找算法性能分析

$$P(n,k)=\frac{1}{n}\sum_{i=1}^{n}C_i=\frac{1}{n}\left(C_{root}+\sum_{L}C_i+\sum_{R}C_i\right)$$

$$=\frac{1}{n}\left(1+k(P(k)+1)+(n-k-1)(P(n-k-1)+1)\right)$$

$$=1+\frac{1}{n}\left(k\times P(k)+(n-k-1)\times P(n-k-1)\right)$$

9.3.1 二叉排序树—查找算法性能分析

再由 $P(n)=\frac{1}{n}\sum_{k=0}^{n-1}P(n,k)$

$$P(n)=\frac{1}{n}\sum_{k=0}^{n-1}\left(1+\frac{1}{n}\left(k\times P(k)+(n-k-1)\times P(n-k-1)\right)\right)$$

$$=1+\frac{2}{n^2}\sum_{k=1}^{n-1}\left(k\times P(k)\right)$$ 初始条件： $P(1)=1$

可类似于解差分方程，此递归方程有解：

$$P(n)=2\left(1+\frac{1}{n}\right)\sum_{i=1}^{n}\frac{1}{i+1}-1$$

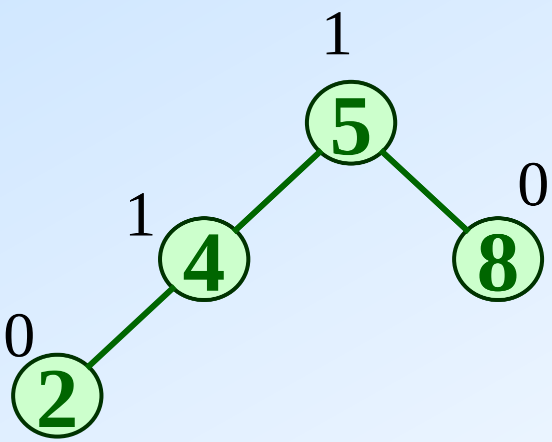
$$P(n)\approx 2\left(1+\frac{1}{n}\right)(\ln n-1)+\frac{2}{n}-1\approx 2\left(1+\frac{1}{n}\right)\ln n$$

9.3.1 二叉排序树—查找算法性能分析

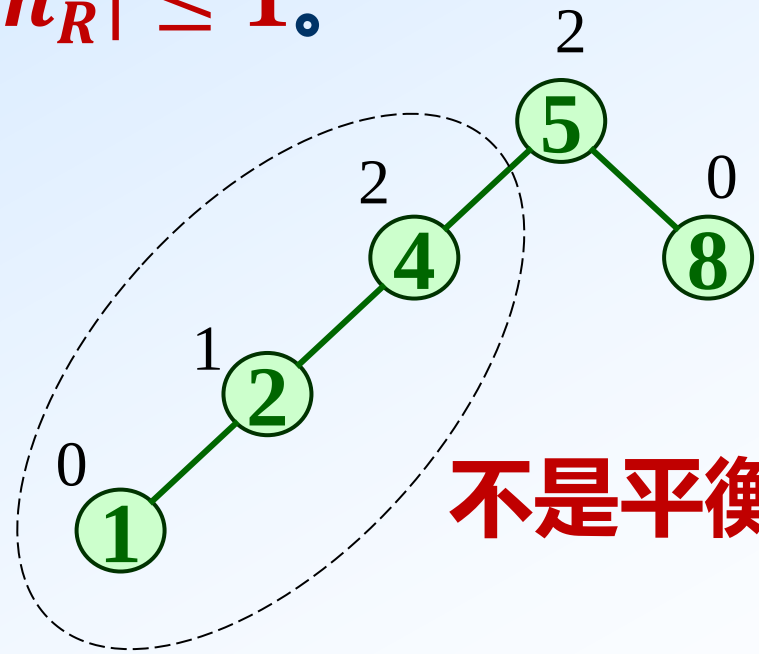
- 二叉排序树的平均查找长度ASL与给定的关键字的总数及顺序相关；
- 给定关键字的数量时，
  - 最差的ASL为单枝树的O(n)；
  - 最优的ASL为O(ln n)；
  - 平均情况下，二叉排序树的ASL为O(ln n)；

9.3.2 平衡二叉树

- 平衡二叉树(AVL树)也是二叉排序树。
- 特点：树中每个结点的左、右子树深度之差（平衡因子BF）的绝对值不大于1，即$|h_L - h_R| \le 1$。



平衡二叉树

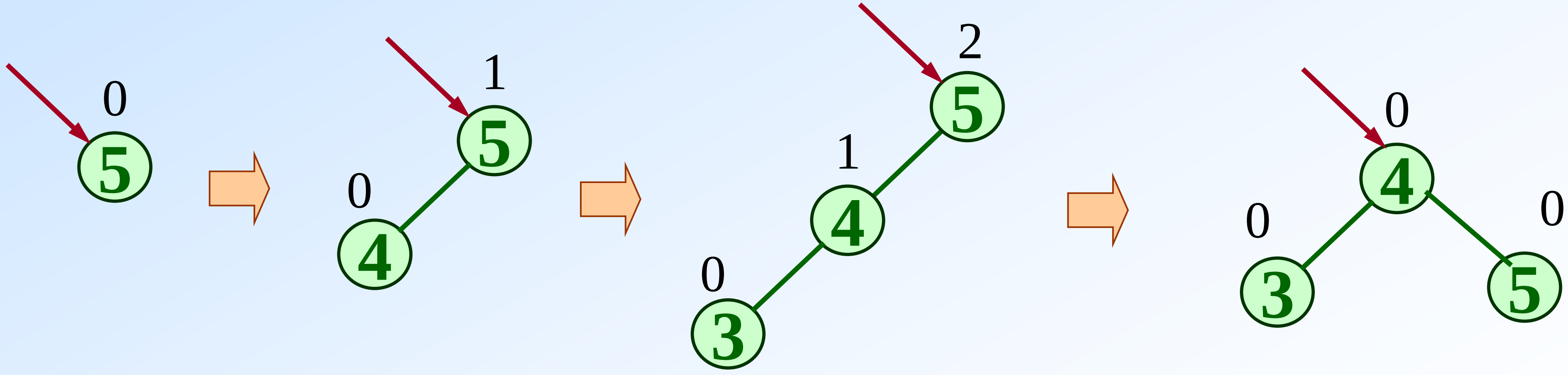


不是平衡二叉树

失去平衡的最小子树

9.3.2 平衡二叉树—构造

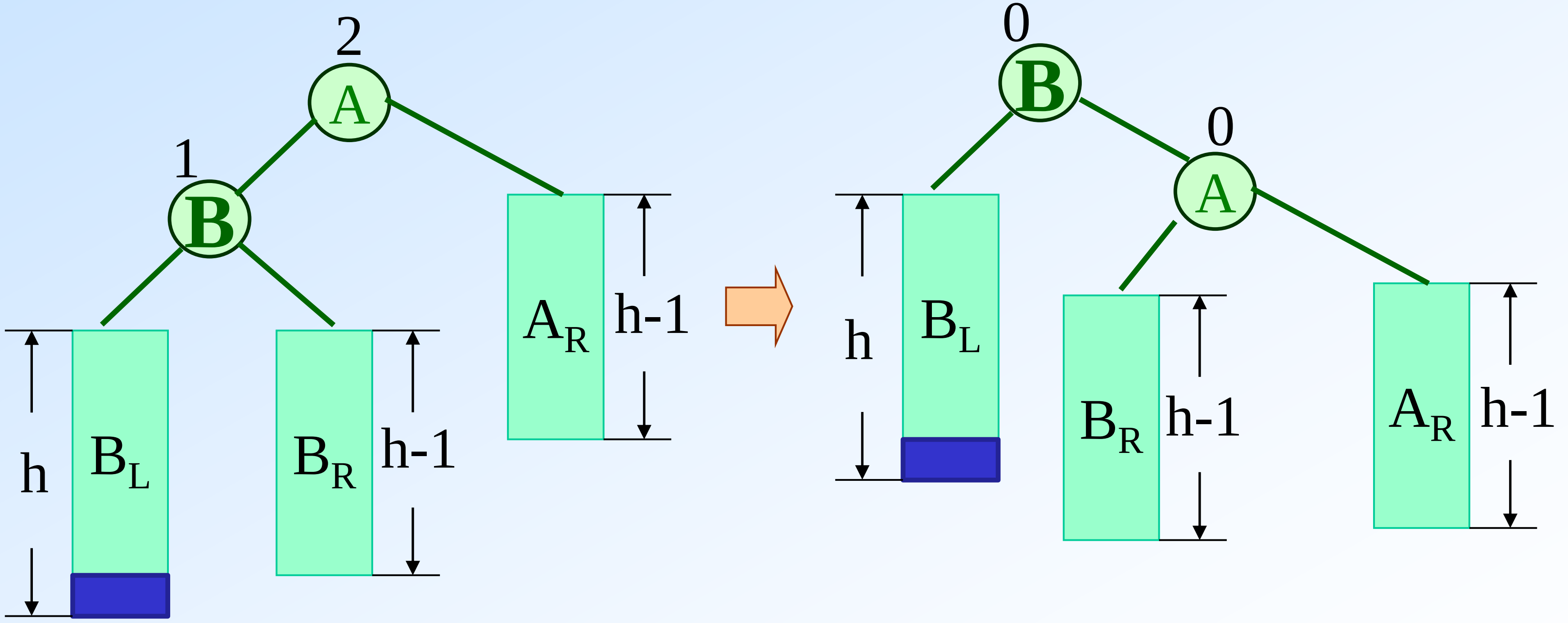
- 在插入过程中，若失去平衡，则需要采用**平衡旋转**技术。
- 依次插入的关键字为**5, 4, 3, 7, 19, 1, 2, 25, 23**
- 可归纳为**LL型，向右旋转**



向右旋转

9.3.2 平衡二叉树—构造

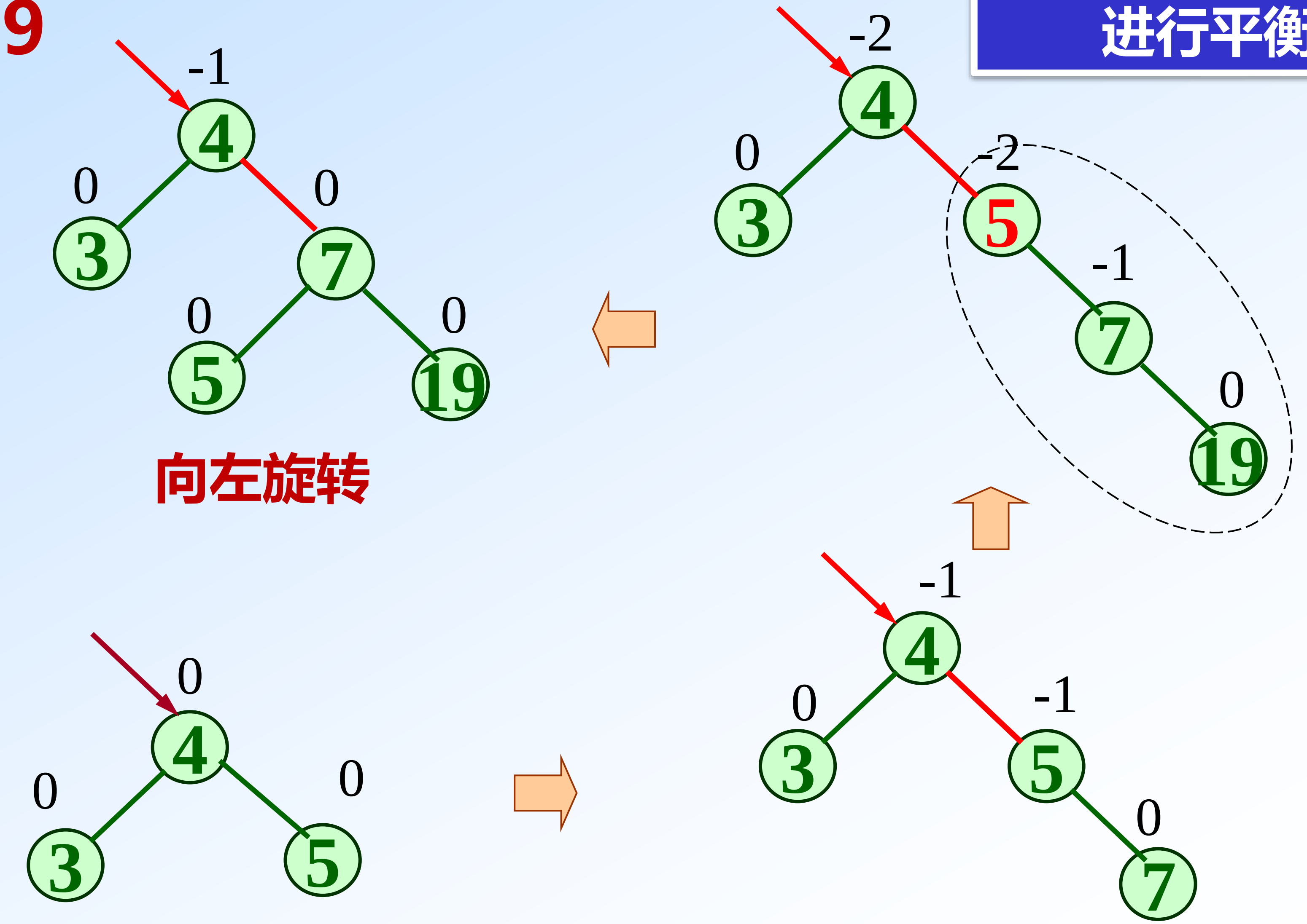
- 可归纳为**LL型**
  - 在单向右旋平衡处理后BF(B)由1变为0，**BF(A)由2变为0**



9.3.2 平衡二叉树—构造

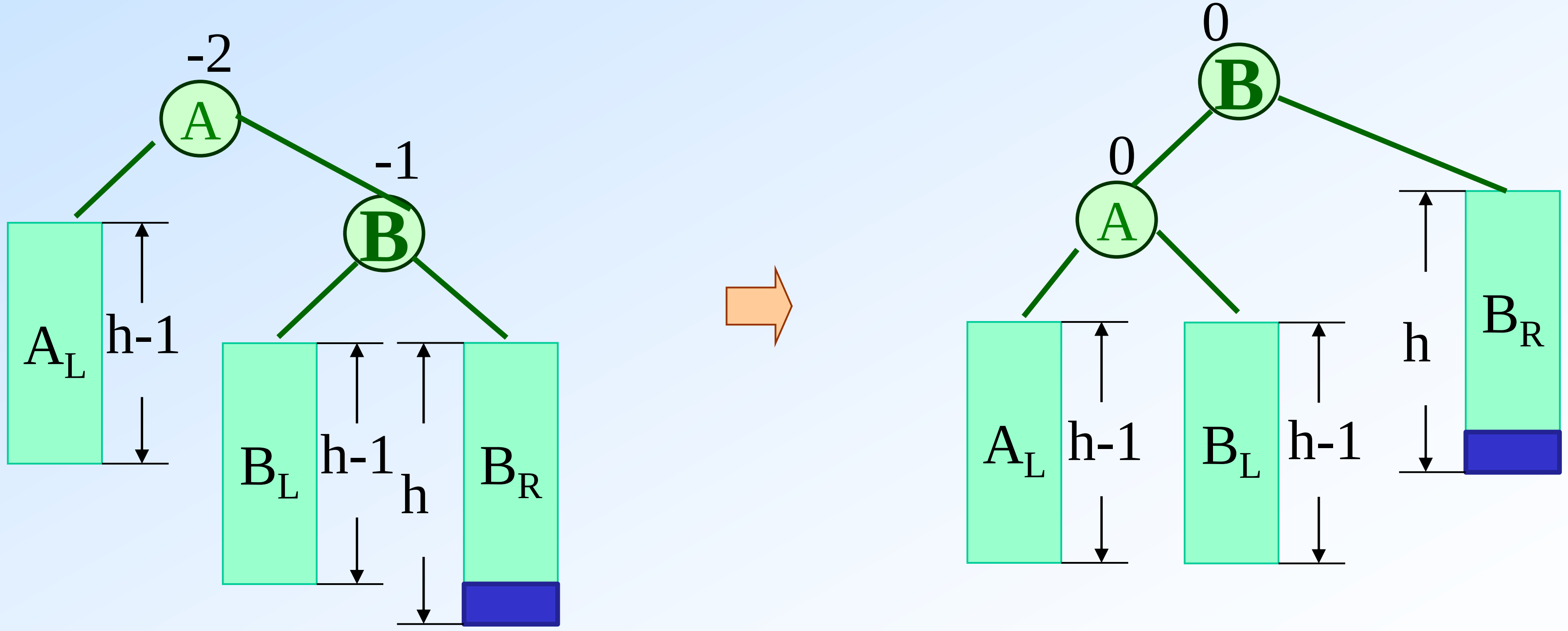
- 继续插入7, 19

对失去平衡的最小子树
进行平衡旋转



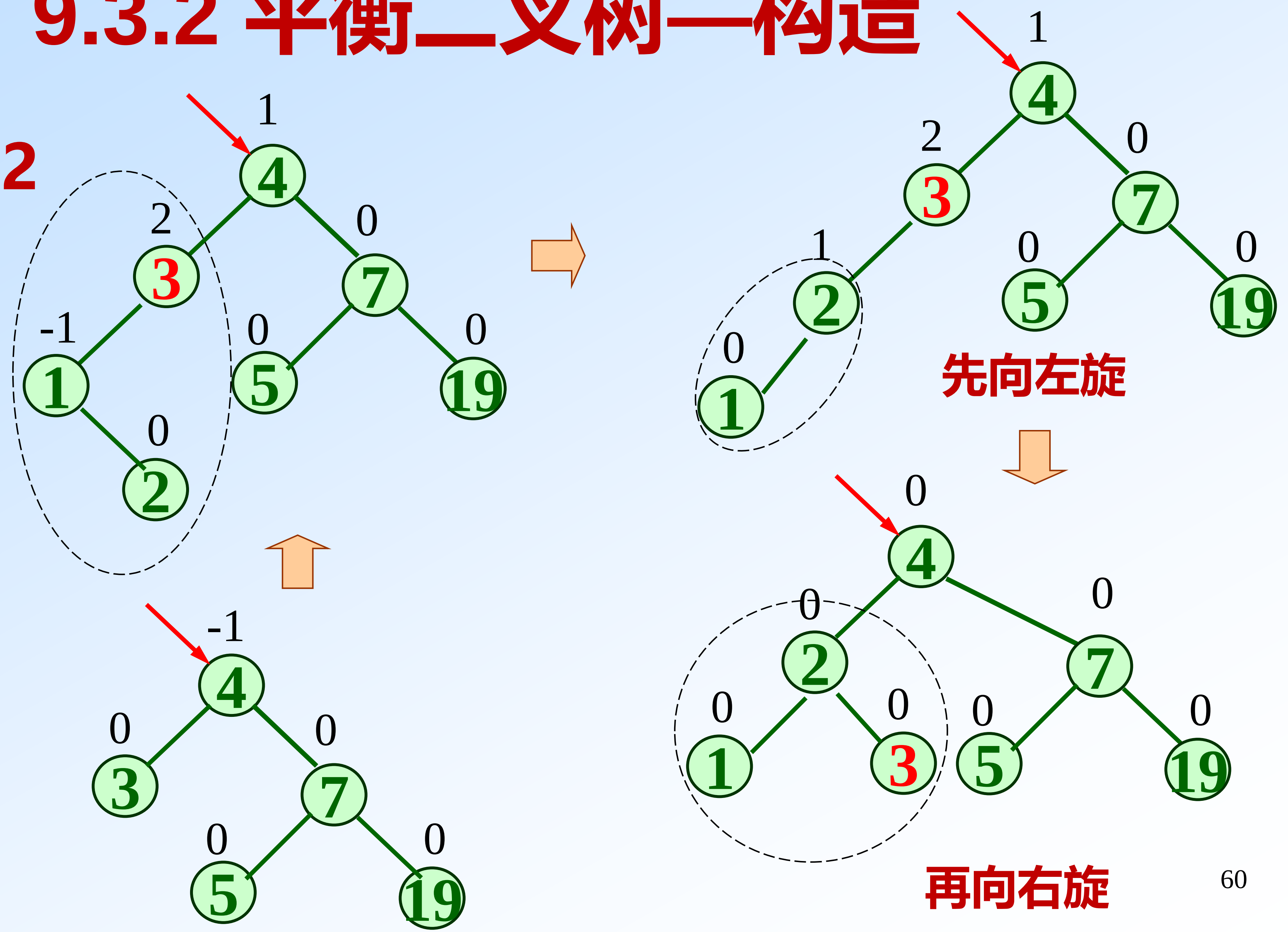
9.3.2 平衡二叉树—构造

- 可归纳为**RR型**
  - 在单向左旋平衡处理后BF(B)由-1变为0，**BF(A)由-2变为0**



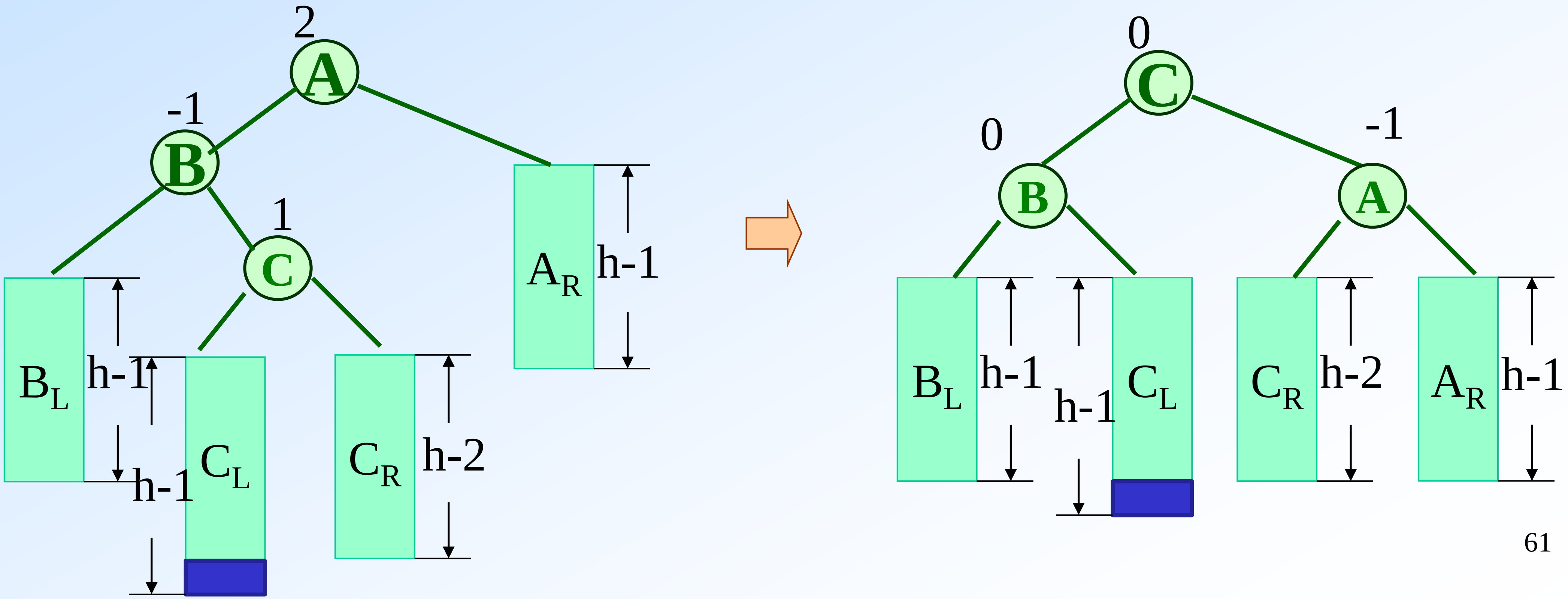
9.3.2 平衡二叉树—构造

- 继续插入1, 2



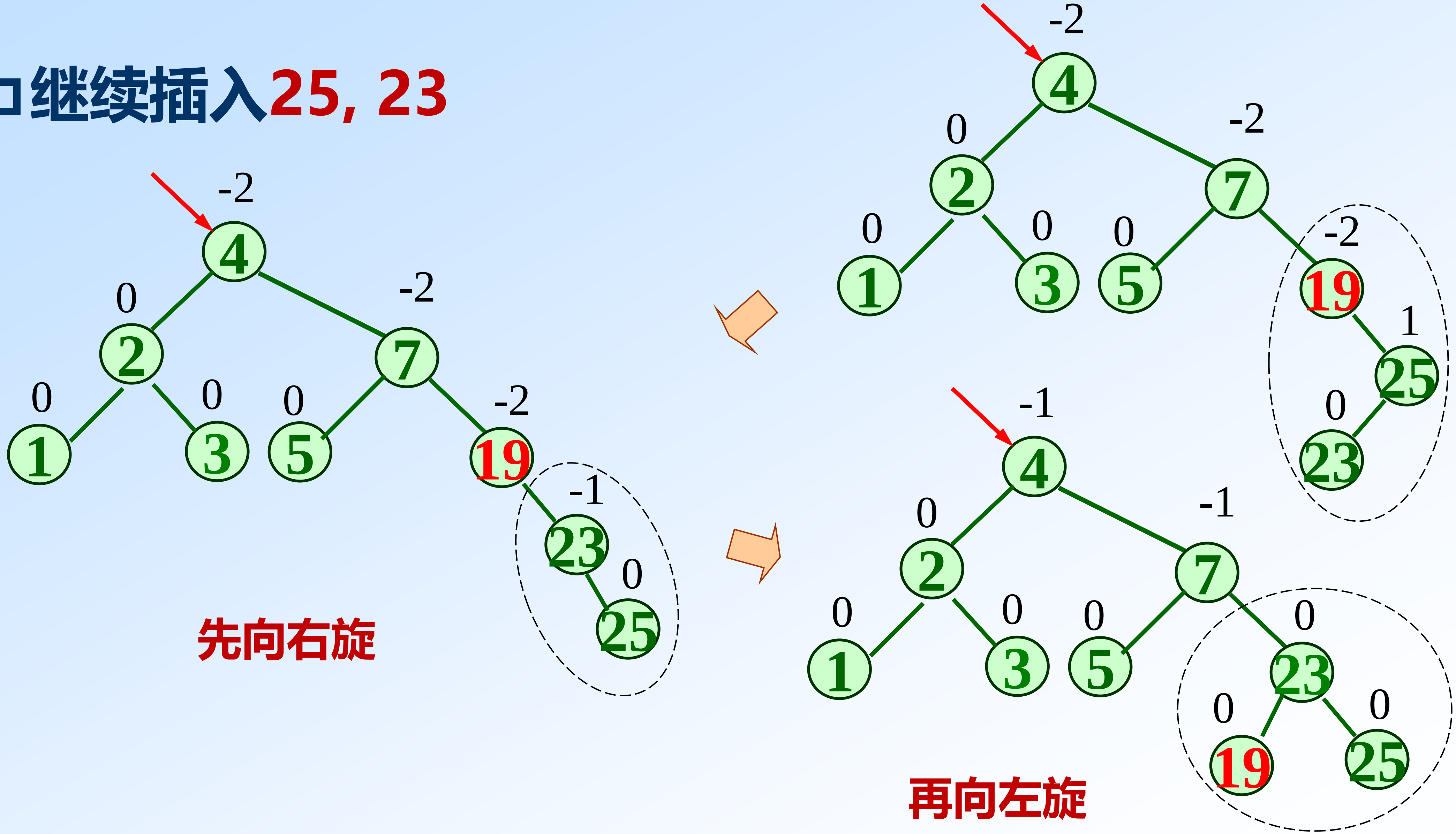
9.3.2 平衡二叉树—构造

- 可归纳为**LR型**
  - 在双向旋转平衡处理后**BF(A)由2变为-1**，BF(B)由-1变为0，BF(C)由1变为0



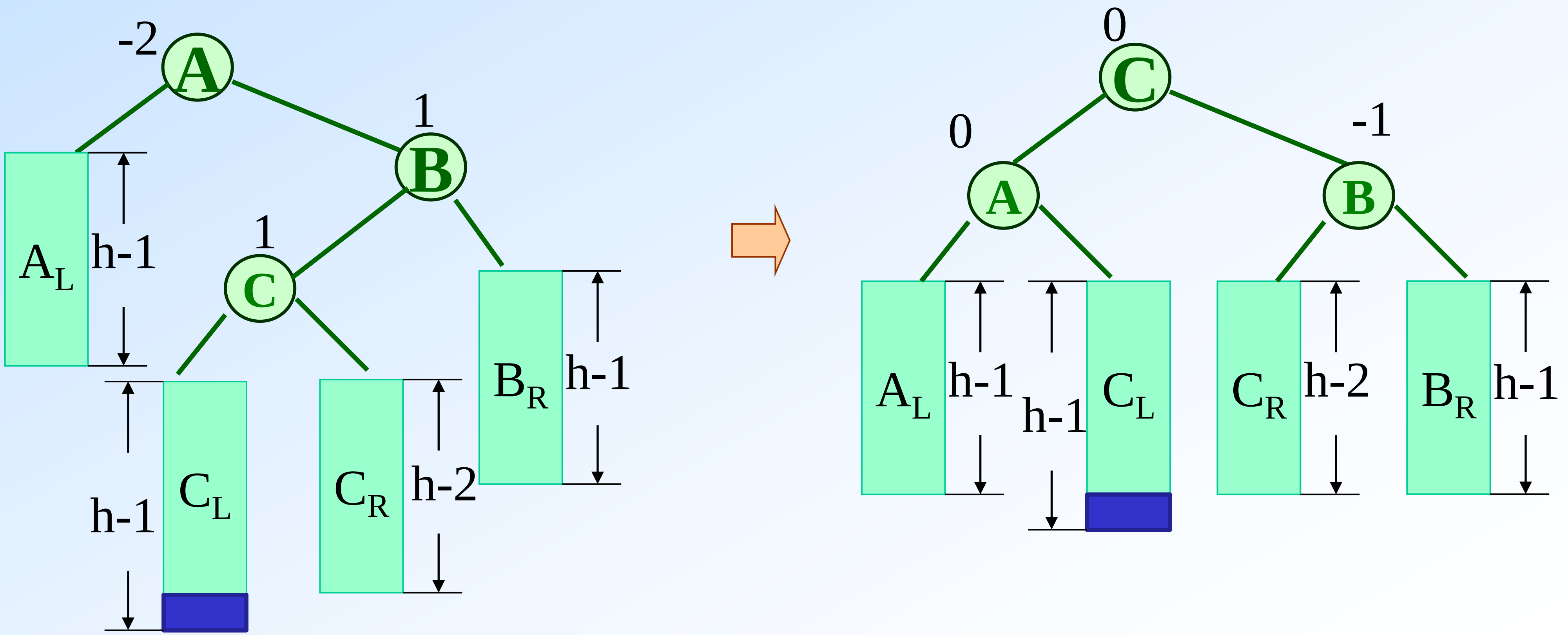
9.3.2 平衡二叉树—构造

- 继续插入25, 23



9.3.2 平衡二叉树—构造

- 可归纳为**RL型**
  - 在双向旋转平衡处理后**BF(A)由-2变为0**，BF(B)由1变为-1，BF(C)由1变为0



9.3.2 构造平衡二叉树—构造小结

- 口诀
  - 左左右，右右左，左右左右，右左右左；
- 对不平衡的最小子树操作；
- 旋转后子树根结点平衡因子为0；
- 旋转后子树深度不变故不影响全树，也不影响插入路径上所有祖先结点的平衡度。

9.3.2 平衡二叉树——存储结构

□平衡二叉树结构定义：

```
typedef struct BSTNode{
    ElemType data;
    int   bf;  // 平衡因子
    struct BSTNode *lchild,*rchild;
}BSTNode,*BSTree;
```

9.3.2 平衡二叉树——插入算法

1. 若是空树，插入结点作为根结点，树深度加1
2. 待插入结点key值等于根结点key值，则不插入
3. 待插入结点key值小于根结点key值，且在左子树中不存在与key值相同的值，则插入在左子树上，左子树深度加1，并且：
   ① 若根结点平衡因子为-1，则改为0，树深度不变
   ② 若根结点平衡因子为0，则改为1，树深度加1
   ③ 若根结点平衡因子为1，且插入后左子的平衡因子为1（左左），则单向右旋；旋转后，新的根结点和右子的平衡因子改为0，树深度不变
   ④ 若根结点平衡因子为1，且插入后左子的平衡因子为-1（左右），则先左旋再右旋；旋转后新的根结点的平衡因子改为0，树深度不变

9.3.2 平衡二叉树——插入算法

4. 待插入结点key值大于根结点key值，且在右子树中不存在与key值相同的值，则插入在右子树上，方法类似第3步：

① 若根结点平衡因子为1，则改为0，树深度不变

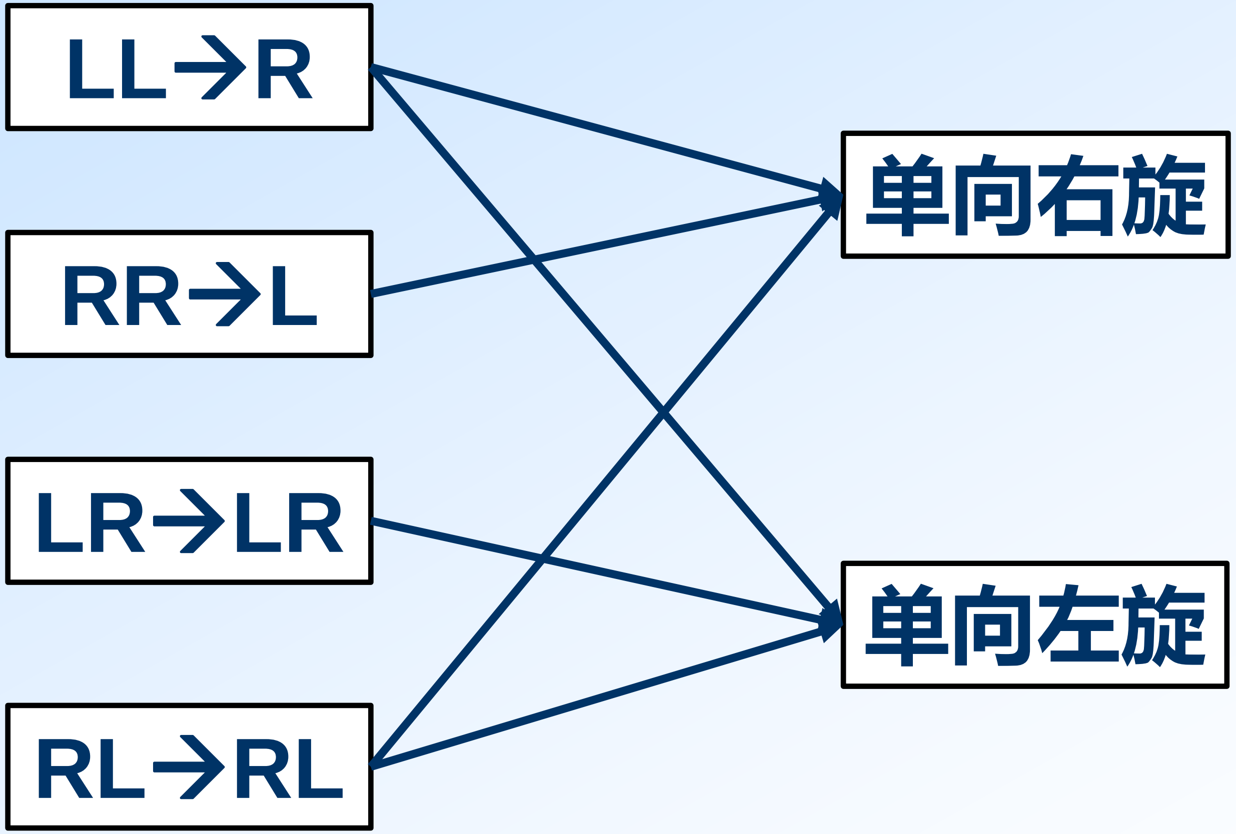
② 若根结点平衡因子为0，则改为-1，树深度加1

③ 若根结点平衡因子为-1，且插入后右子的平衡因子为-1（右右），则单向左旋，旋转后新的根结点和其左子的平衡因子改为0，树深度不变

④ 若根结点平衡因子为-1，且插入后右子的平衡因子为1（右左），则先右旋再左旋，旋转后新的根结点的平衡因子改为0，树深度不变

9.3.2 平衡二叉树——插入算法思想

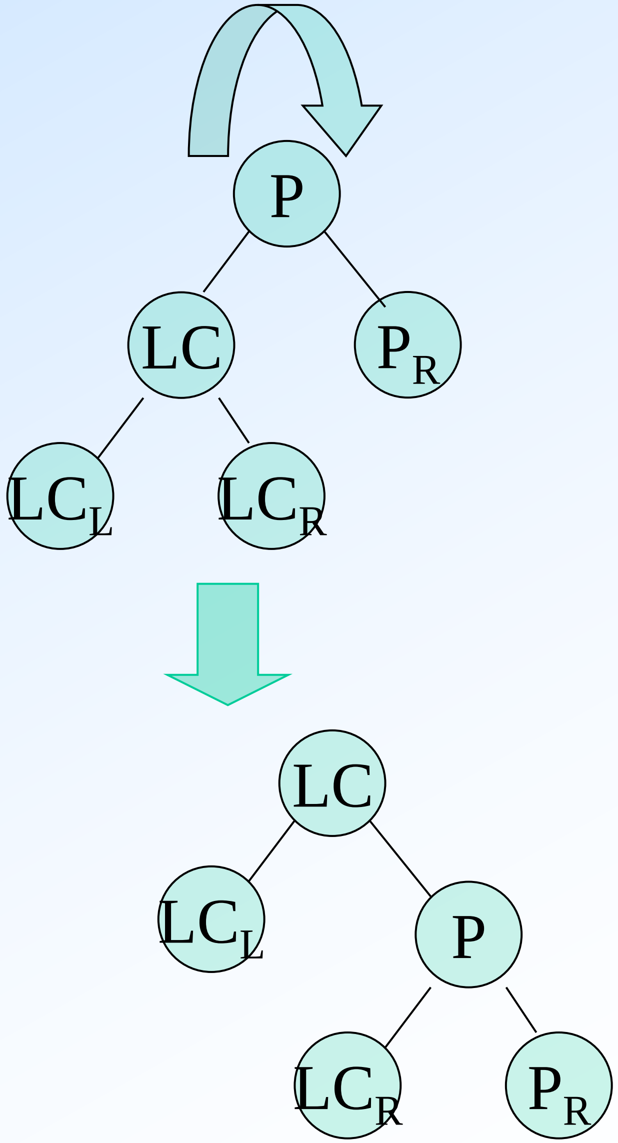
- 自底向上设计思想



9.3.2 平衡二叉树——插入算法

□右旋算法：

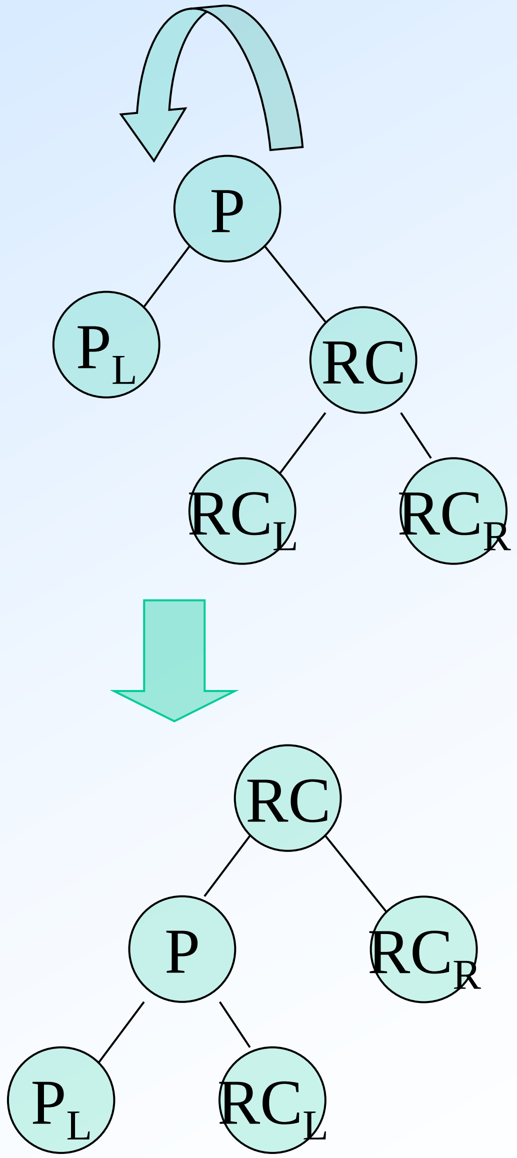
```
Void R_Rotate(BSTree &p){
   lc = p->lchild;
   p->lchild = lc->rchild;
   lc->rchild = p;
   p = lc;
} // R_Rotate
```



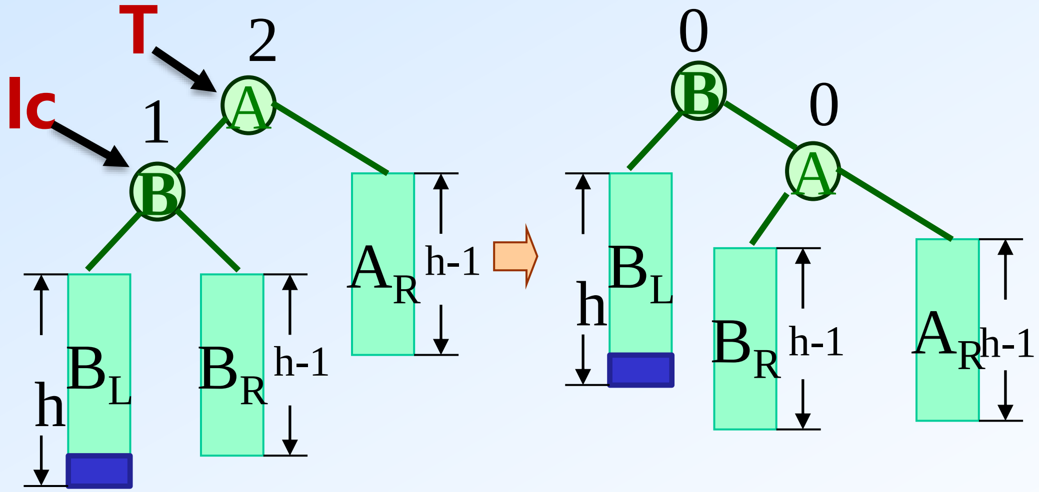
9.3.2 平衡二叉树——插入算法

▫左旋算法：

```
void L_Rotate(BSTree &p){
   rc = p->rchild;
   p->rchild = rc->lchild;
   rc->lchild = p;
   p = rc;
} // L_Rotate
```



9.3.2 平衡二叉树——插入算法



□左平衡旋转：

```
#define LH      +1     // 左高
#define EH       0     // 等高
#define RH      -1     // 右高
void LeftBalance (BSTree &T) {   // T对应图中的树根A
    lc = T->lchild;              // lc对应图中的树根B
    switch (lc->bf){             // 看T的左子(B)，只能是LL或LR型
      case LH:                   // 左左->右旋（LL型）
        T->bf = lc->bf=EH;
        R_Rotate(T);
        break;
```

9.3.2 平衡二叉树——插入算法

```
        case RH:                    // LR型，先左旋后右旋
            rd = lc->rchild;        // 结点C
            switch(rd->bf) {        // 修改BF值
                case LH:  //见图1
                    T->bf = RH; lc->bf=EH; break;
                case EH:  //见图2
                    T->bf = lc->bf = EH; break;
                case RH:  //见图3
                    T->bf=EH; lc->bf=LH; break;
            }  // end switch(rd->bf)
            rd->bf=EH;
            L_Rotate(T->lchild);  R_Rotate(T);
    }  // end switch(lc->bf)
}      // end LeftBalance
```

见图1

见图2

见图3

主程序

9.3.2 平衡二叉树——插入算法

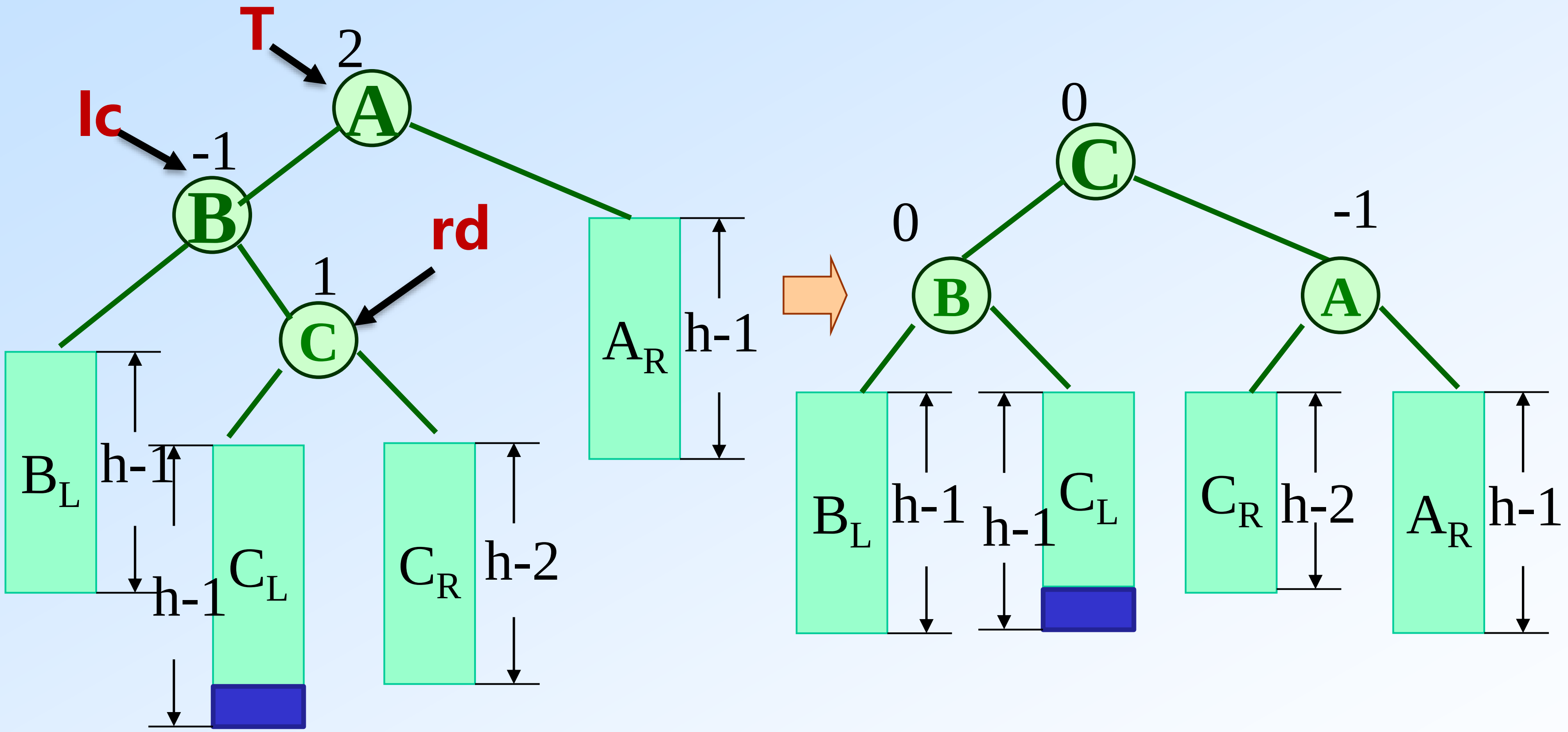


图1 假设插入点在C的左子树上，
且BF(C) = LH

返回

9.3.2 平衡二叉树——插入算法

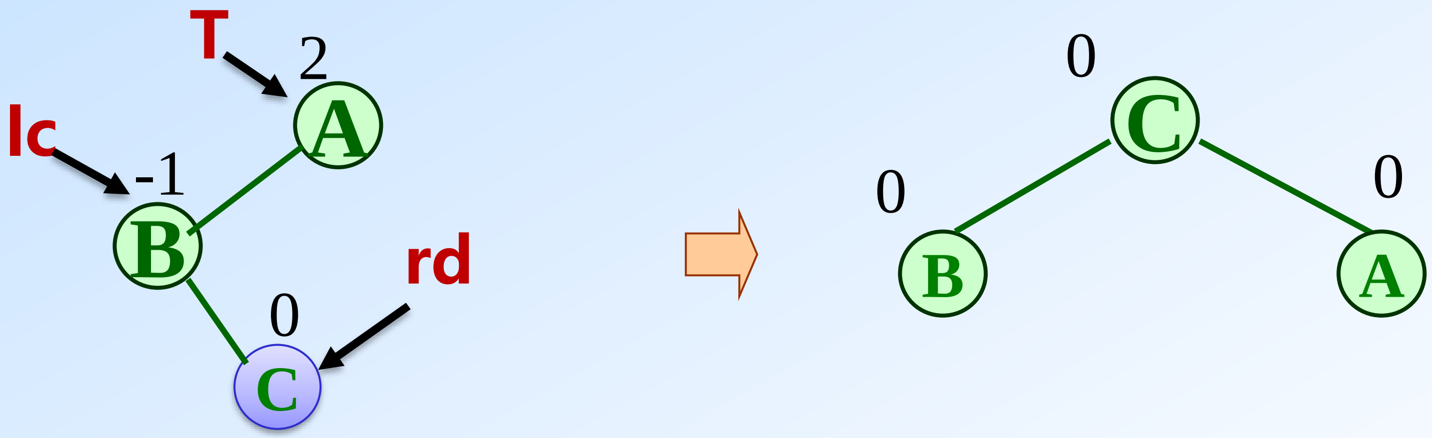


图2 假设插入点在C上，
则BF(C) = EH

返回

9.3.2 平衡二叉树——插入算法

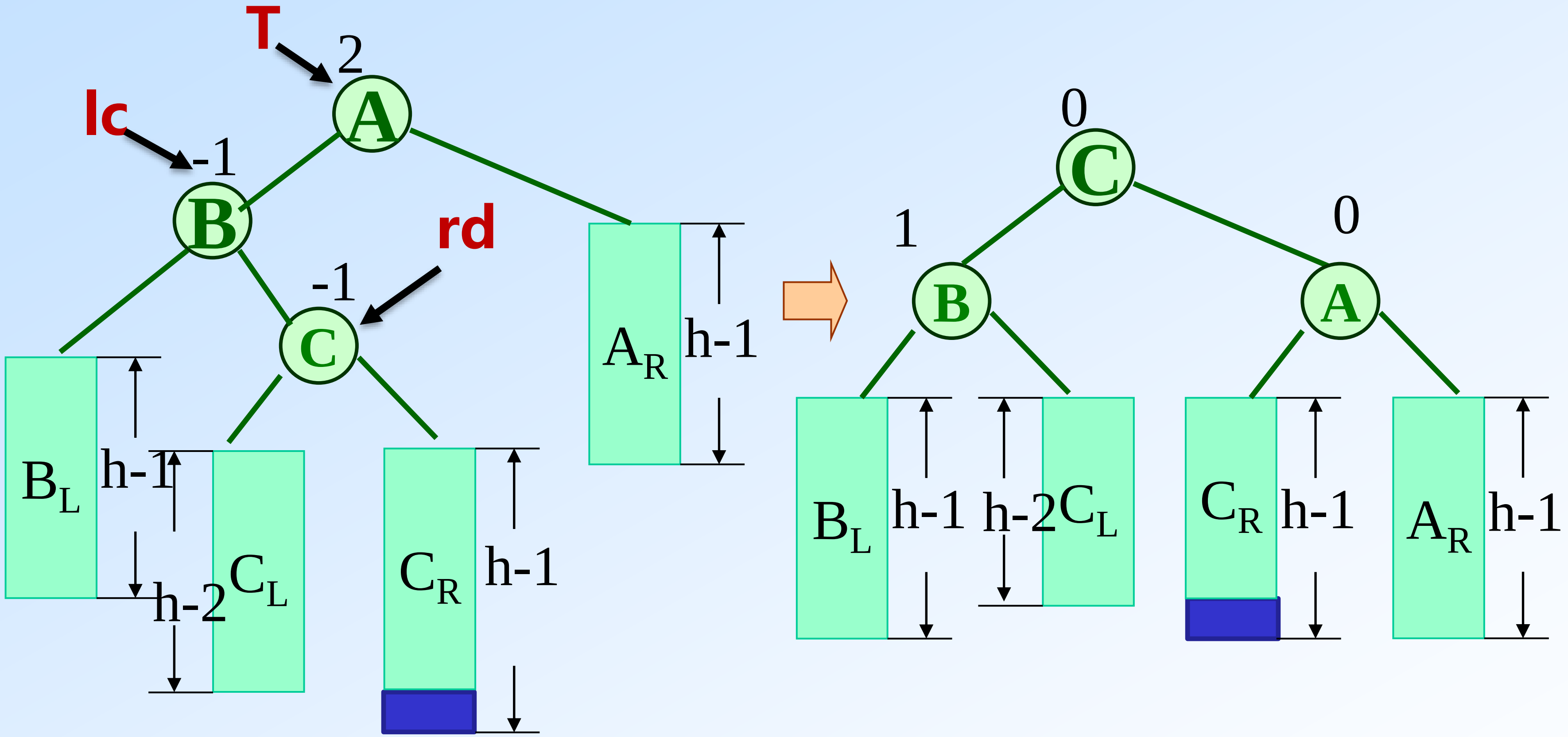


图3 假设插入点在C的左子树上，
且BF(C) = RH

返回

9.3.2 平衡二叉树——插入算法（主程序）

```
Status InsertAVL (BSTree &T, ElemType e,
                  Boolean &taller) {     // taller表示树是否长高
  if(!T) {        // 如果树不存在则创建
    T = (BSTree) malloc (sizeof (BSTNode));
    T->data = e;  T->lchild = T->rchild = NULL;
    T->bf = EH; taller = TRUE;
  }
  else {
    if (e.key == T->data.key) {
      taller = FALSE;   return 0;
    }
```

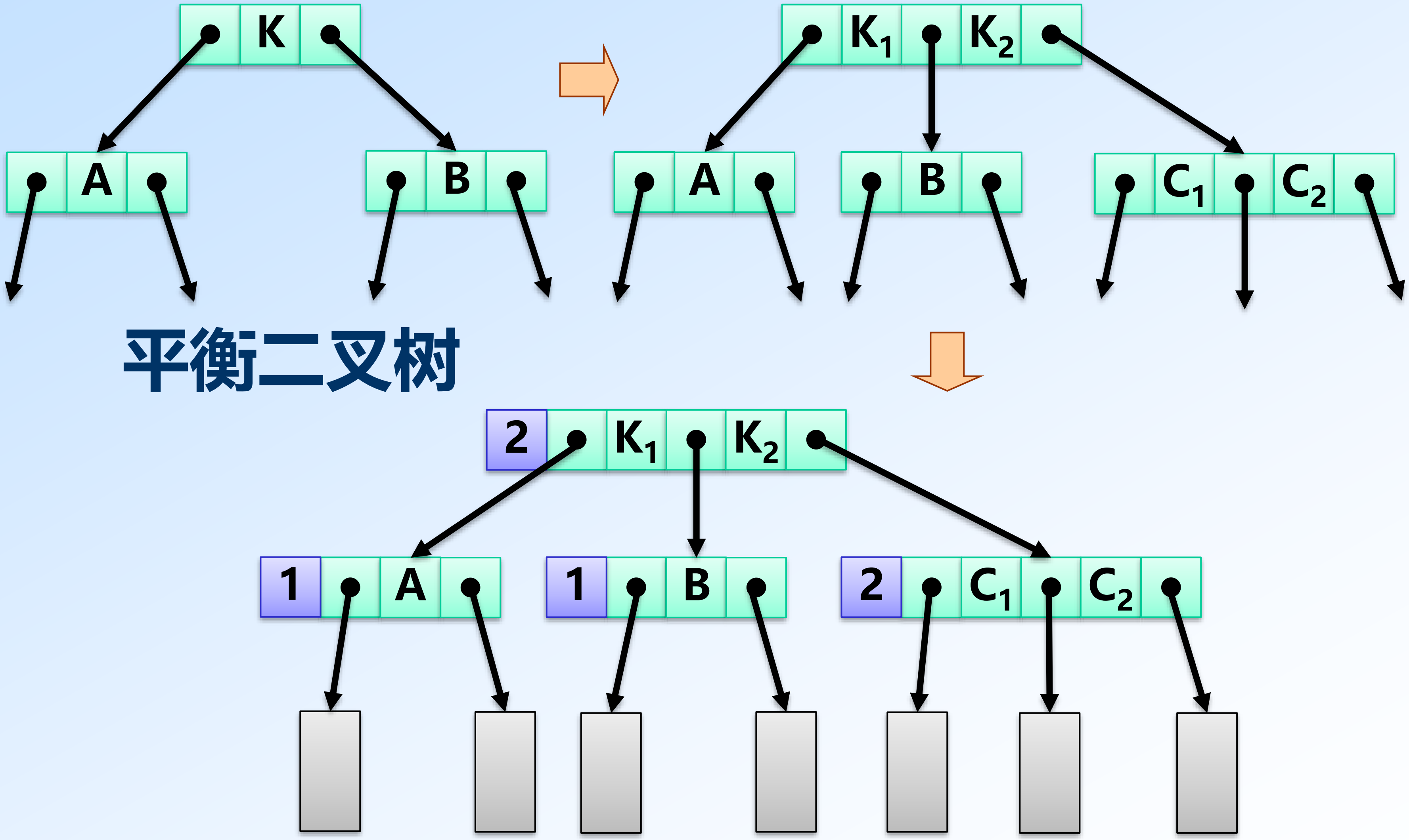
9.3.2 平衡二叉树——插入算法（主程序）

```
if (e.key < T->data.key){   // 要插入在左子树上
  if (!InsertAVL(T->lchild, e, taller)) return 0;
  if (taller)
    switch(T->bf) {             // 检查*T的平衡度
      case LH: LeftBalance(T); taller = FALSE; break;
      case EH: T->bf = LH; taller = TRUE; break;
      case RH: T->bf = EH; taller = FALSE; break;
    } // end switch(T->bf)
}
else {                          // 要插入在右子树上
```

9.3.2 平衡二叉树——插入算法（主程序）

```
    if(!InsertAVL(T->rchild, e, taller)) return 0;
    if(taller)
      switch(T->bf) {
        case LH: T->bf = EH; taller = FALSE; break;
        case EH: T->bf = RH; taller = TRUE; break;
        case RH:      // 右平衡函数没有给出
          RightBalance(T); taller = FALSE; break;
      }  // end switch(T->bf)
    }    // end if < else
  }      // end if (!T) else
  return 1;
}        // end InsertAVL
```

9.3.3 B-树



平衡二叉树

B-树

9.3.3 B-树的定义

- B-树：一种平衡的多路查找树
- 一棵 m 阶的 B-树或为空树，或为满足下列特性的 m 叉树：
  1. 树中每个结点至多有 m 棵子树
  2. 若根结点不是叶子结点，则至少有两棵子树
  3. 除根结点之外的所有非终端结点至少有⌈m/2⌉棵子树

9.3.3 B-树的定义

4. 所有的非终端结点中包含下列信息数据
   $(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$
   - K_i $(i = 1, 2, \ldots, n)$ 为关键字，且 $K_i < K_{i+1}$ $(i = 1, 2, \ldots, n-1)$
   - A_i $(i = 0, 1, \ldots, n)$为指向子树根结点的指针，且指针 A_{i-1} 所指子树中所有结点的关键字均小于 K_i $(i = 1, 2, \ldots, n)$，A_n 所指子树中所有结点的关键字均大于 K_n，n ($\lceil m/2 \rceil - 1 \le n \le m-1$) 为关键字的个数（或 n+1 为子树个数）。
5. 所有的叶子结点都出现在同一层次上，并且不带信息（空指针）。

- 一棵 m 阶B-树每个结点最多有 m 棵子树， m – 1 个关键字，最少有$\lceil m/2 \rceil$棵子树，$\lceil m/2 \rceil - 1$个关键字

9.3.3 B-树——存储结构

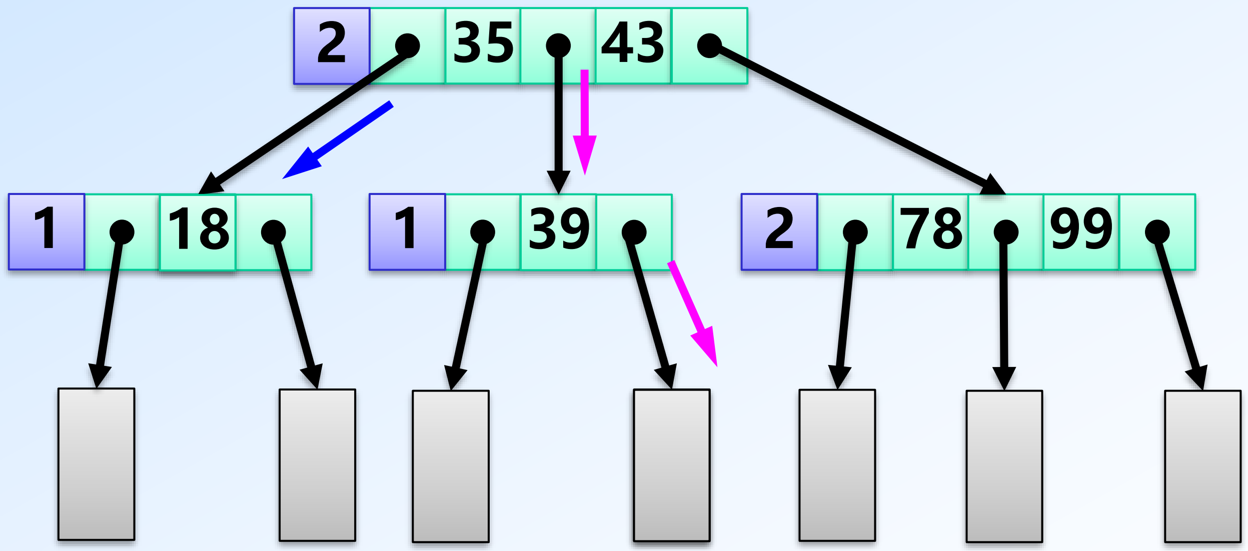
```
#define     m       3
typedef struct BTNode {
    int               keynum;        // 结点中关键字个数
    struct BTNode     *parent;       // 指向双亲结点
    KeyType           K[m+1];        // 关键字向量
    struct BTNode     *ptr[m+1];     // 子树指针向量，指的是A0, A1, …
    Record            *recptr[m+1];  // 记录指针向量
}BTNode, *BTree;

typedef struct{
    BTNode            *pt;           // 指向找到的结点
    int               i;             // 在结点中的关键字序号
    int               tag;           // 返回查找是否成功(0或1)
}Result;                             // 查找结果
```

9.3.3 B-树——查找

□例如：在B-树中查找关键字值等于

18 40



9.3.3 B-树——查找算法

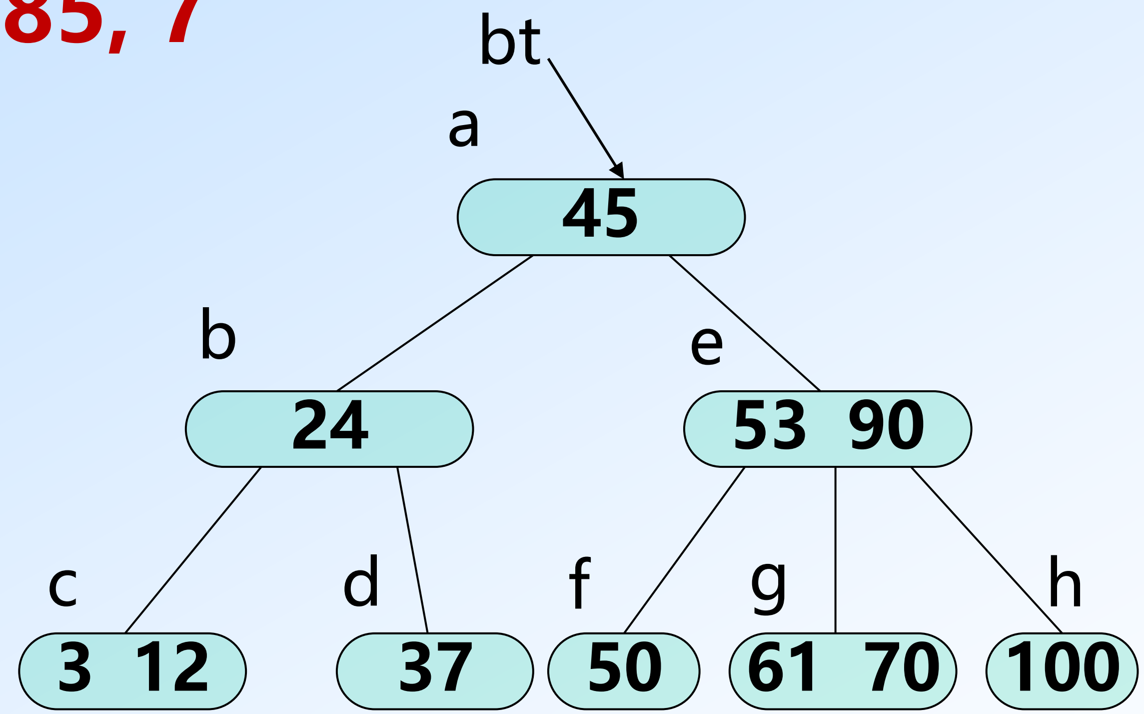
```
Result SearchBTree(BTree T, KeyType key) {
  p = T; q = NULL; found = FALSE; i = 0;
  while (p && !found) {
    // 在 p->K[...] 中查找 key 所在位置 i，使得 p->K[i] <= key < p->K[i+1]
    i = Search (p, key);
    if (i > 0 && p->K[i] == key) found=TRUE;
    else {q = p; p = p->ptr[i];}    // 未在当前结点找到
  }
  if(found) return(p, i, 1);         // 查找成功
  else return(q, i, 0);              // 返回 key 的插入位置
} // end SearchBTree
```

9.3.3 B-树——插入

- B-树 $\lceil m/2 \rceil - 1 \leq$ 结点中的关键字个数 $\leq m - 1$；
- 每次插入一个关键字不是在树中添加一个叶子结点，而是在查找的过程中**找到叶子结点所在层的上一层**；
- 在某个结点中添加一个关键字，若结点的关键字个数**不超过 $m - 1$，则插入完成**；
- **否则**产生结点的**分裂**。

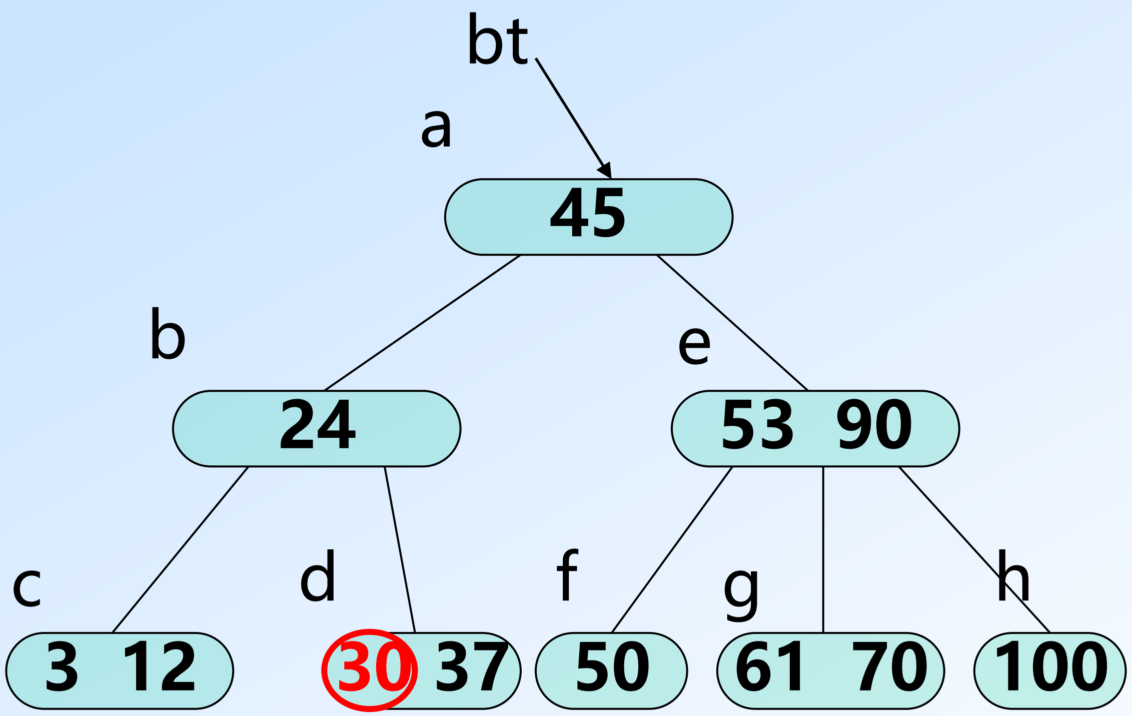
9.3.3 B-树——插入

- 在下图所示的3阶B-树（略去所有叶子结点）中依次插入关键字30, 26, 85, 7



插入关键字30

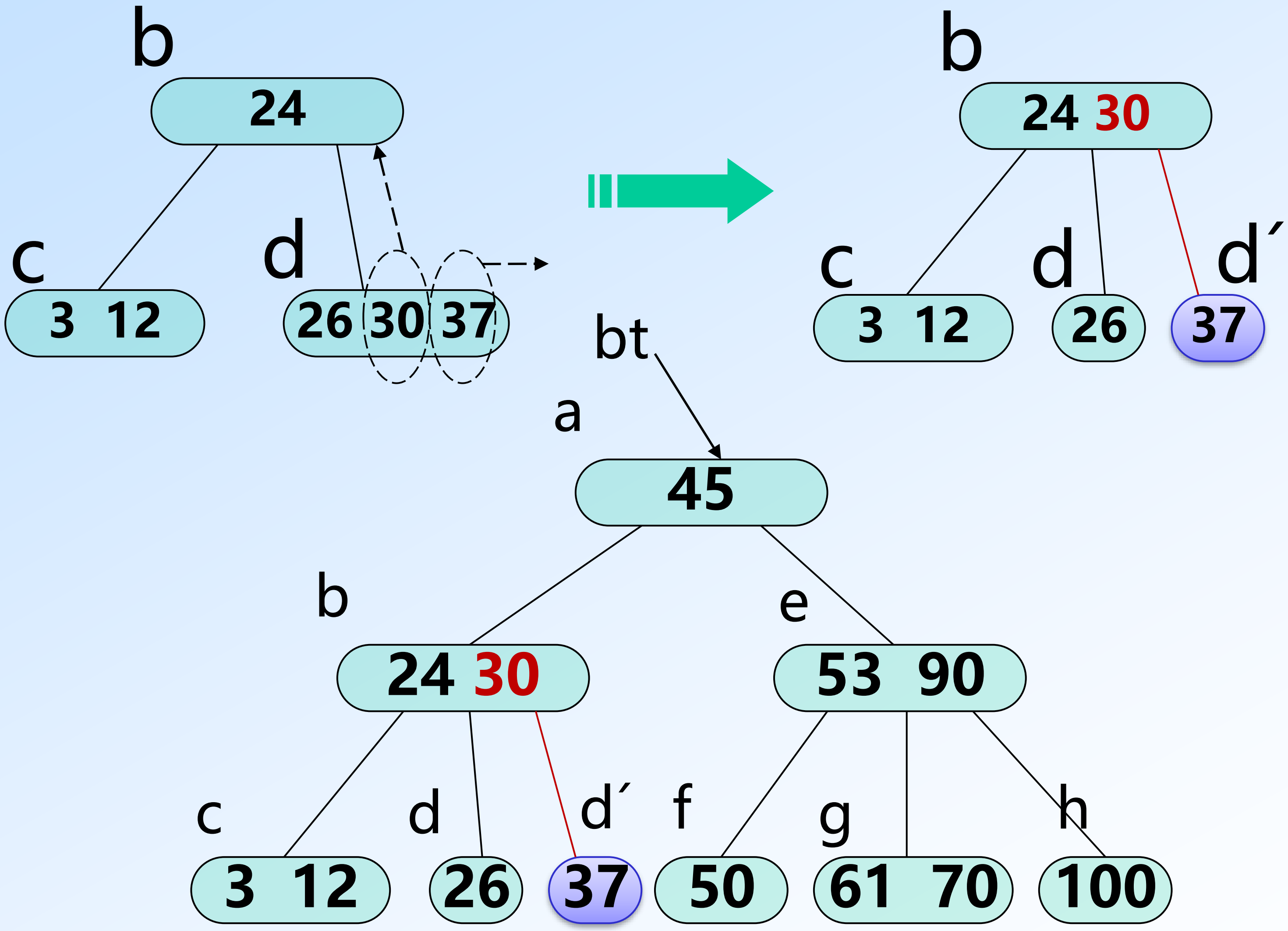
9.3.3 B-树——插入



插入30后的结果

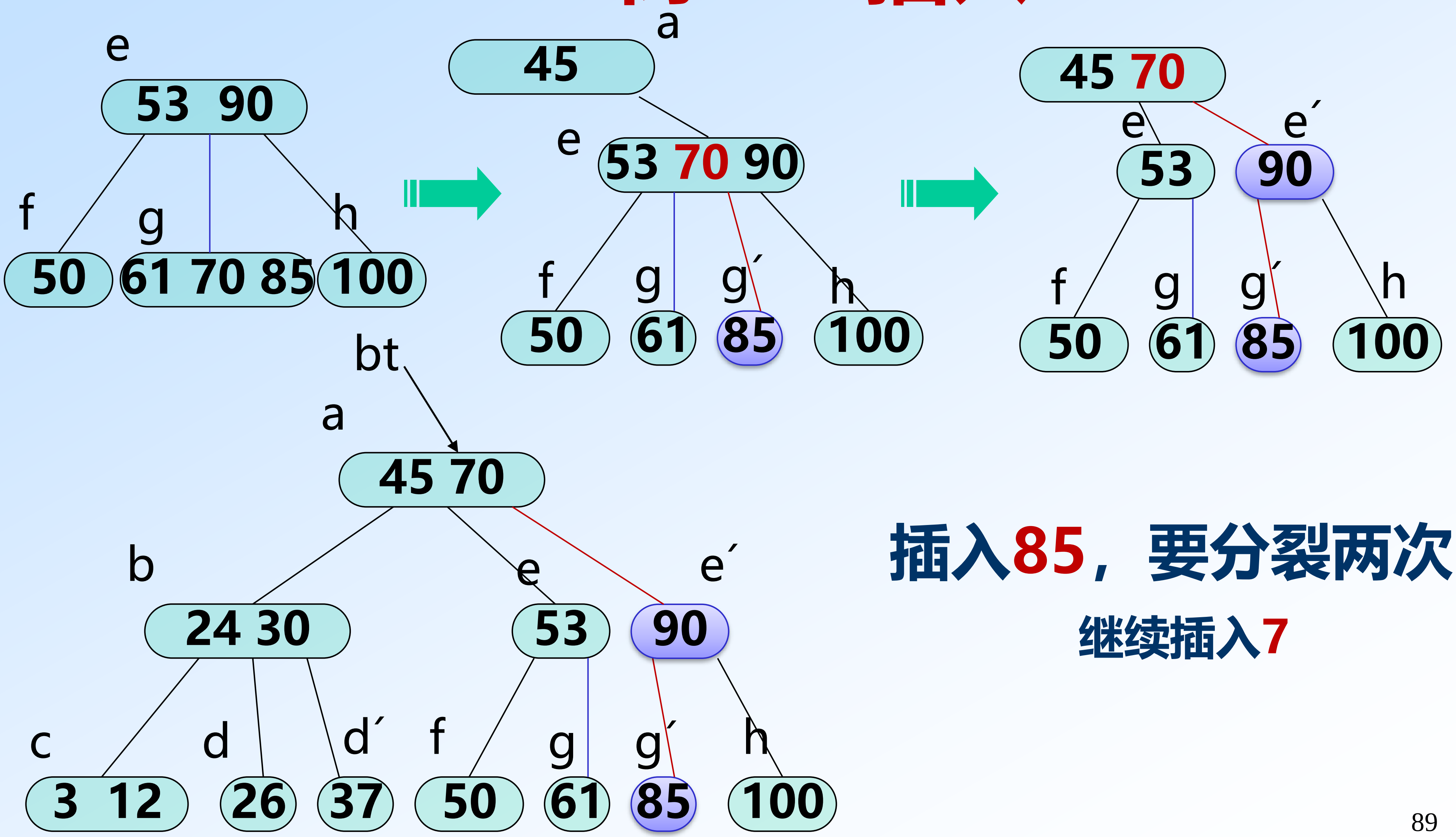
接下来，继续插入26

9.3.3 B-树——插入



插入26 接下来，继续插入85

9.3.3 B-树——插入

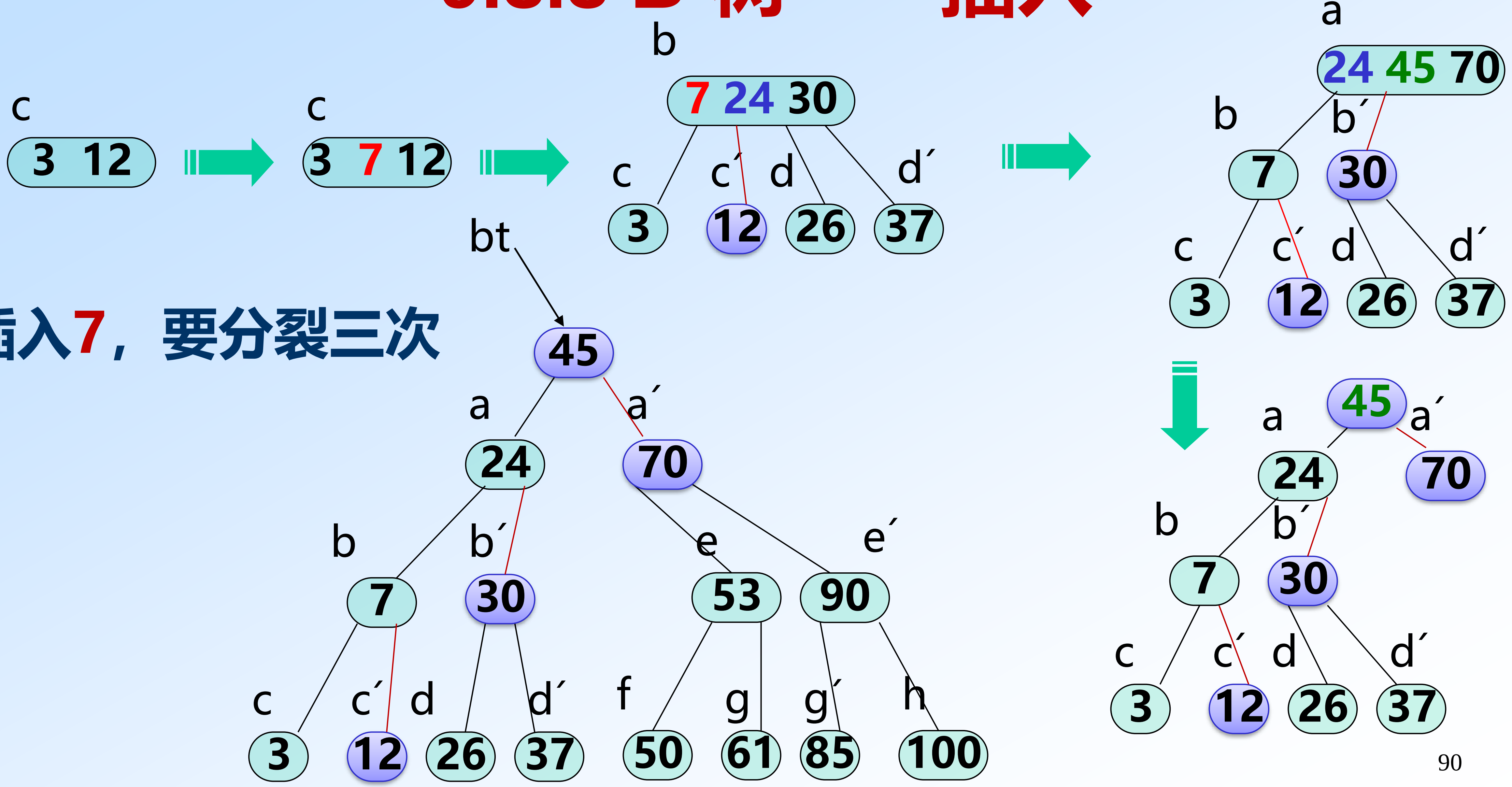


插入85，要分裂两次

继续插入7

9.3.3 B-树——插入

插入7，要分裂三次



9.3.3 B-树——插入小结

- 假设 *p 结点中已有 $m-1$ 个关键字，当插入一个关键字之后，结点中信息为： $m, A_0, (K_1, A_1), (K_2, A_2), \ldots, (K_m, A_m)$，其中 $K_i < K_{i+1}, 1 \le i < m$，先将 *p 分裂成 *p 和 *p' 两个结点
  - *p 信息为 $\lceil m/2 \rceil - 1, A_0, (K_1, A_1), \ldots, (K_{\lceil m/2 \rceil -1}, A_{\lceil m/2 \rceil -1})$
  - *p' 信息为 $m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (K_{\lceil m/2 \rceil +1}, A_{\lceil m/2 \rceil +1}), \ldots, (K_m, A_m)$
- 关键字 $K_{\lceil m/2 \rceil}$ 和其后指针一起插入到 p 的双亲结点中

9.3.3 B-树——插入算法

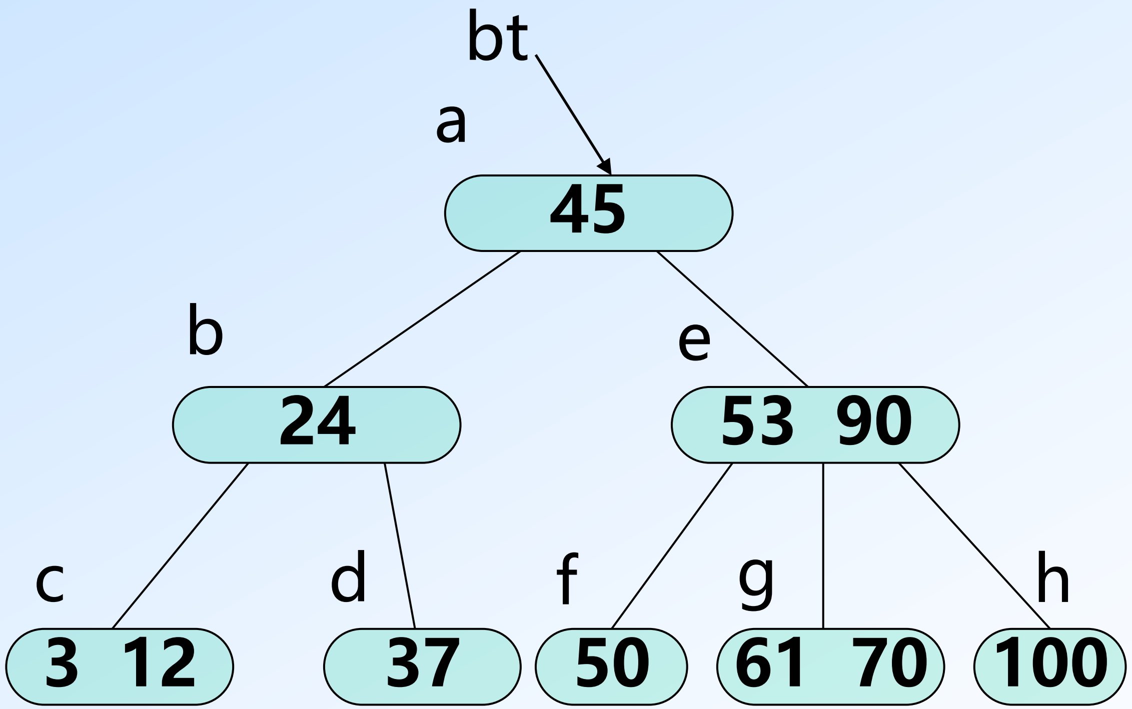
```
Status InsertBTree (BTree &T, KeyType key, BTree q, int i) {
    x = key; ap = NULL; finished = FALSE;
    while (q && !finished) {
        Insert (q, i, x, ap);              // 插入 x 和 ap 到 q->K[i+1] 和 q->ptr[i+1]
        if (q->keynum < m) finished = TRUE; //判断是否要分裂
        else {                              // 进行分裂
            s = ⌈m/2⌉; Split(q, s, ap);    // 在 s 处分裂
            x = q->K[s]; q = q->parent;
            if (q) i = Search(q, x);        // 搜索父结点的插入位置
        }  // end if (q->keynum < m) else
    }  // end while
    if (!finished) NewRoot(T, q, x, ap);        // 空树则创建新根结点
}  // InsertBTree
```

9.3.3 B-树——删除

- 删除结点为**非终端结点**
  - 设关键字为 K_i，则可以用 A_i 指向子树的最小关键字或 A_{i-1} 指向子树的最大关键字替换 K_i，再删去该关键字即可，而该关键字必定在终端结点。
- 删除结点为**终端结点**
  - 若删除后仍满足B-树定义，则删除结束；
  - 否则要进行**合并**结点的操作。

9.3.3 B-树——删除

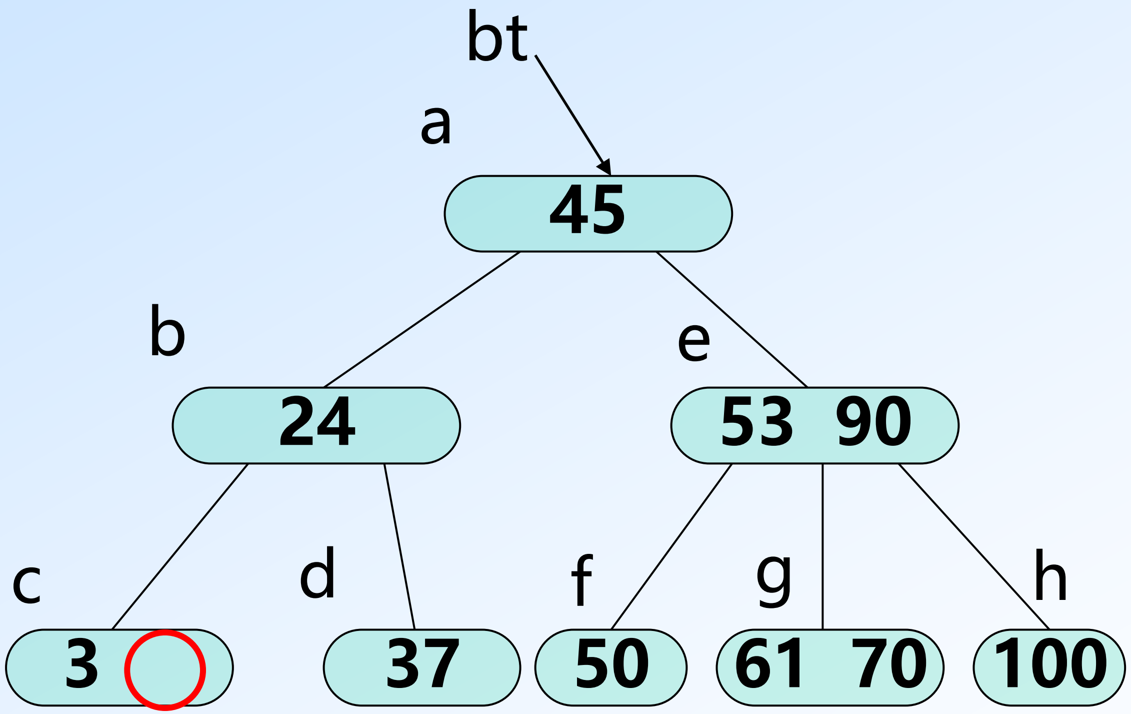
在3阶B-树中依次删除关键字12, 50, 53, 37



3阶B-树

9.3.3 B-树——删除

- 被删关键字12所在结点(c)中的关键字数目不小于⌈m/2⌉，则只须从该结点(c)中删去该关键字12和相应指针，树的其他部分不变。

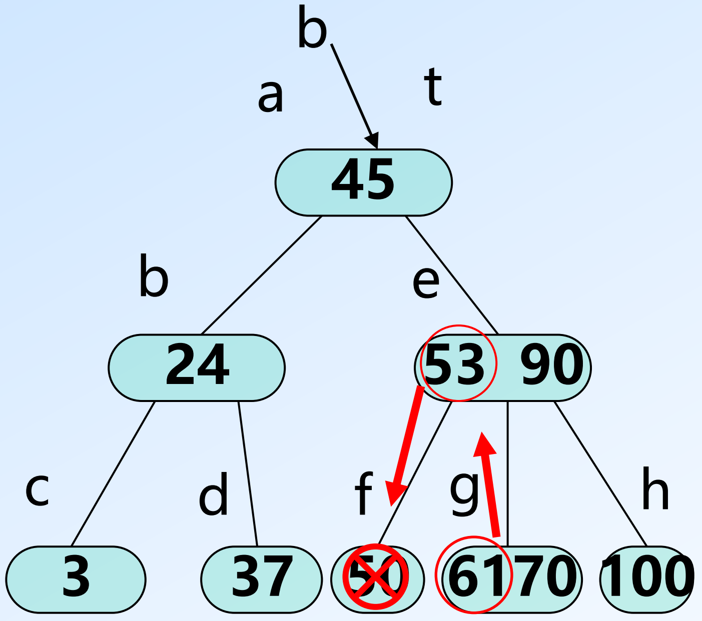


删除关键字12后的结果

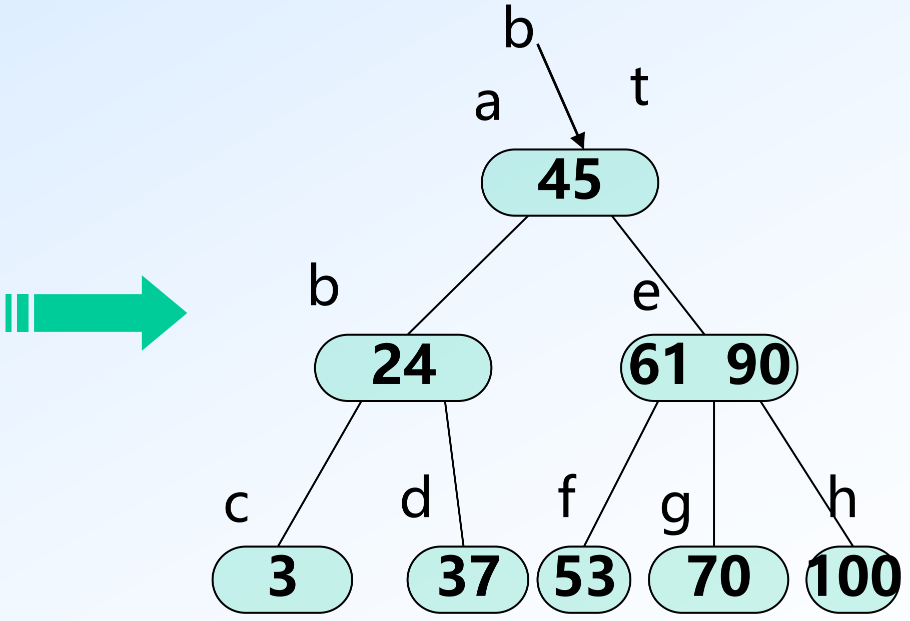
继续删除50

9.3.3 B-树——删除

- 被删关键字50所在结点(f)中的关键字数目等于⌈m/2⌉– 1，其相邻的某个兄弟结点(g) 中的关键字数目大于⌈m/2⌉– 1，则将其兄弟结点(g)中的最小（或最大）关键字61上移至双亲结点中，而将双亲结点中小于（或大于）且紧靠该上移关键字61的关键字53下移至被删关键字50所在结点(f)中。



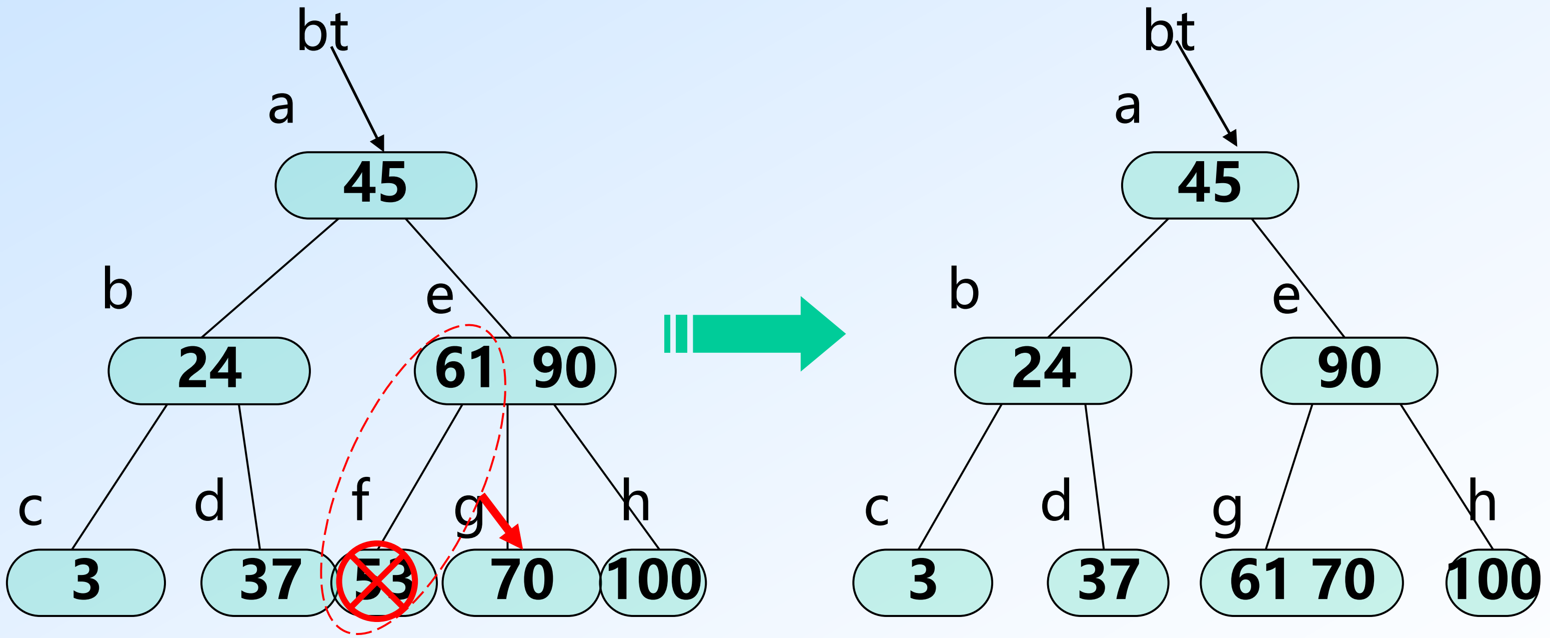
删除关键字50



继续删除53

9.3.3 B-树——删除

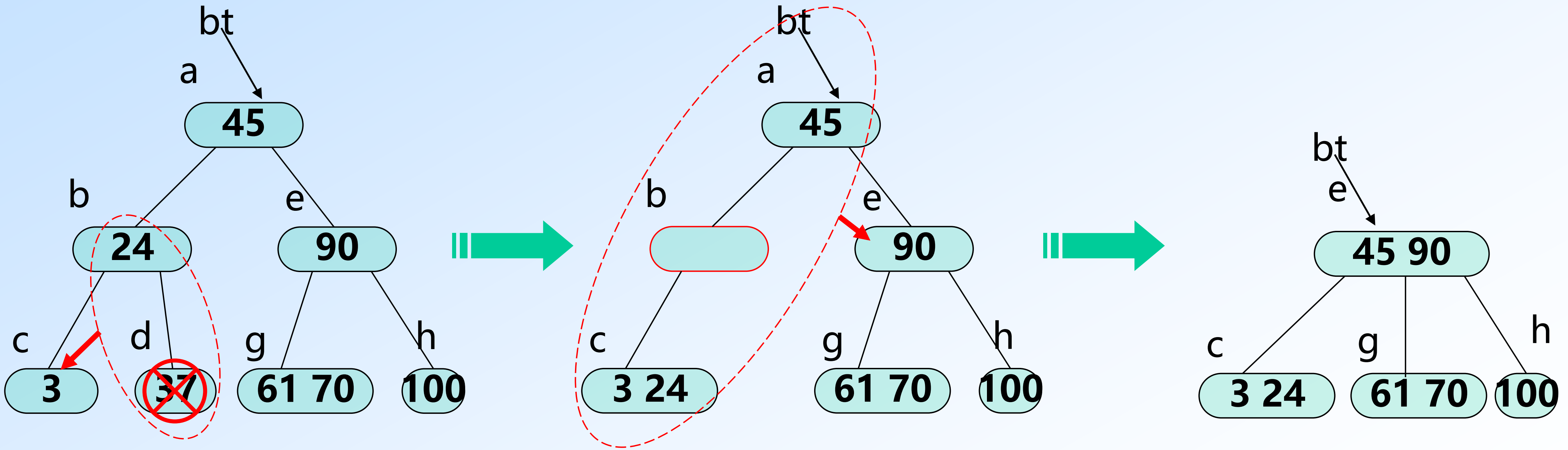
- 被删关键字53所在结点(f)和其相邻的兄弟结点(g)中的关键字数目均等于⌈m/2⌉ – 1，假设该结点有右兄弟(g)，在删去关键字53后，他所在结点(f)中剩余的关键字和指针加上双亲结点(e)中的关键字61一起合并到右兄弟结点(g)中。



删除关键字53　　继续删除37

9.3.3 B-树——删除

- 如果删除后使双亲结点中的关键字数目小于⌈m/2⌉– 1，则依此类推层层向上合并。



删除关键字37

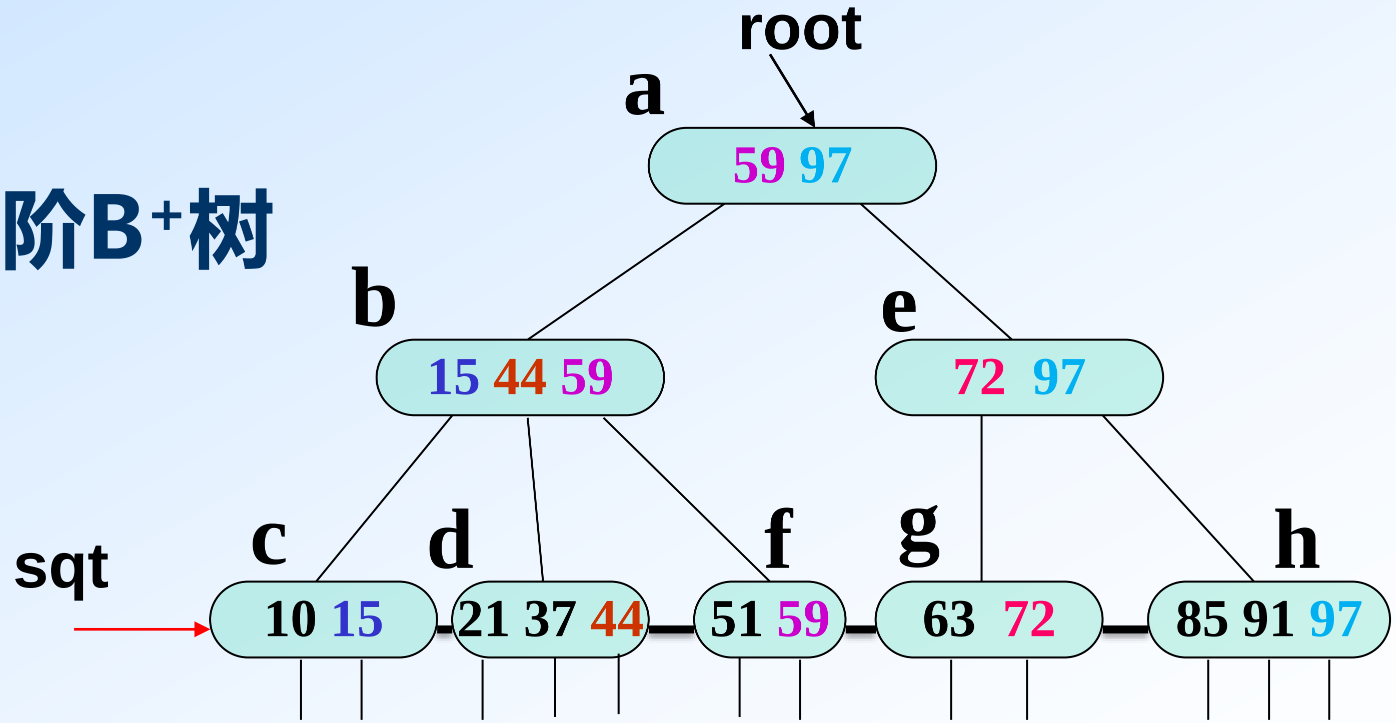
9.3.4 B^+树

- B^+树是B-树的变型，m阶的B^+树和m阶的B-树的区别是：
  - 关键字个数和子树个数一样多;
  - 所有叶子结点中包含全部关键字信息；
  - 叶子结点本身依关键字由小到大顺序链接；
  - 所有非终端结点可看成索引，结点中仅含有其子树中的最大（或最小）关键字。

9.3.4 B⁺树

- 可以**从根结点开始查找**，与B-树查找方式**类似**，不同之处为
  - 从根到叶子结点，即使非终端结点上的关键字等于给定值也不终止
- 还可以从**最小关键字的叶子结点开始**顺序查找。

- 实例：3阶B⁺树



9.3.4 B+树的插入与删除

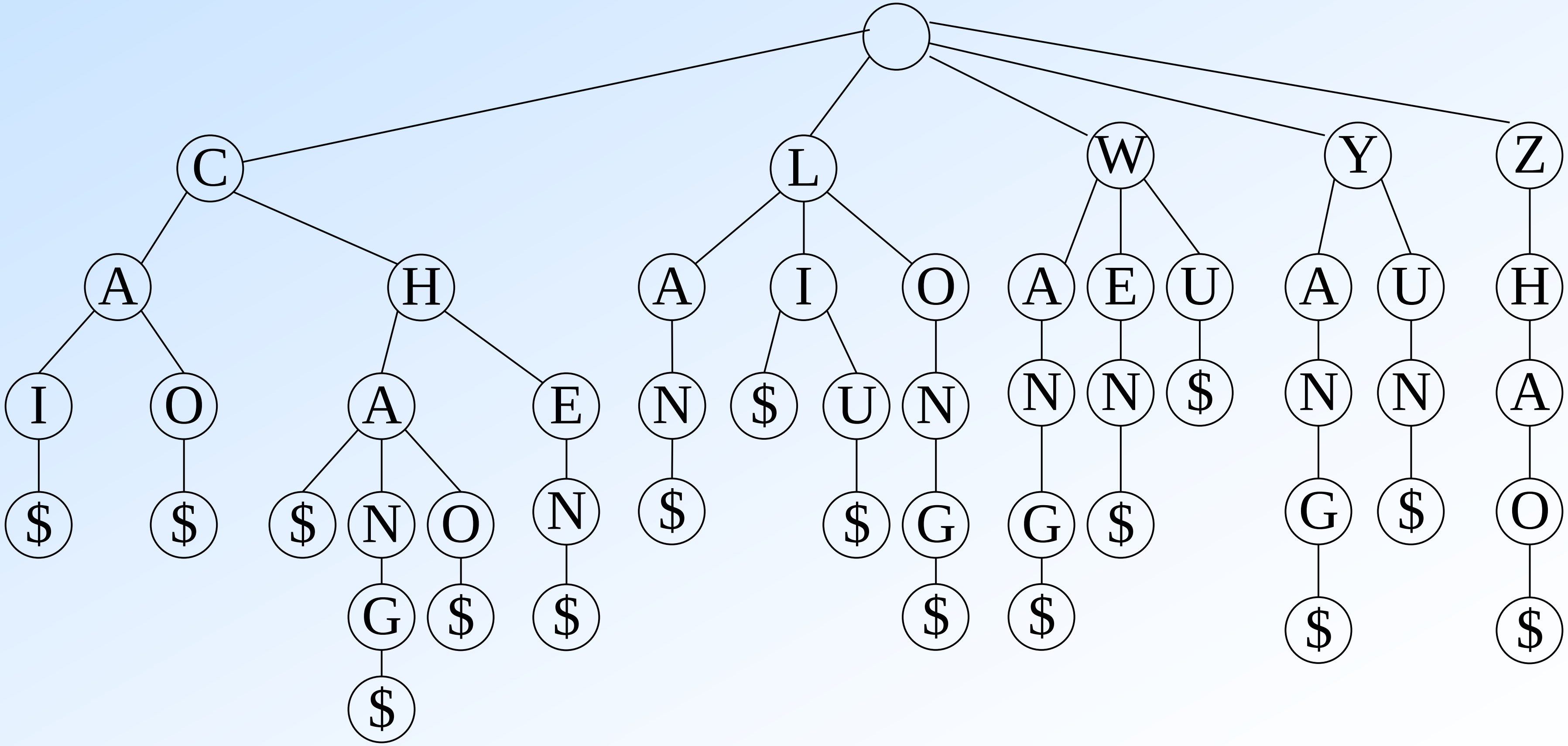
- **插入**基本过程与B-树的插入过程类似，不同之处为：
  - 分裂后的左子结点关键字个数为$\lceil (m+1)/2 \rceil$；
  - 且双亲结点中均包含分裂后两个结点的最大关键字。
- **删除**基本过程与B-树的删除过程类似，不同之处为：
  - 删除仅在叶子结点进行；
  - 最大关键字被删除时，非终端结点中该值可仍保留，作为“分界关键字”；
  - 删除后，关键字个数小于$\lceil m/2 \rceil$时就进行类似B-树的合并操作。

9.3.4 键树

- 键树（数字查找树）
  - 是一棵度≥2的树；
  - 树中每个结点不是包含一个或几个关键字，而是**只含有组成关键字的符号**；
  - 关键字符号：如数值中的每一位，单词中的每个字母。

9.3.4 键树

- 例如关键字集合为{CAI, CAO, LI, LAN, CHA, CHANG, WEN, CHAO, YUN, YANG, LONG, WANG, ZHAO, LIU, WU, CHEN}
- 为了查找和插入方便，可约定键树是有序树，结束符$小于任何字符。



9.4 哈希表

- 哈希表的概念
- 哈希函数的构造方法
  - 直接定址法
  - 数字分析法
  - 平方取中法
  - 折叠法
  - 除留余数法
  - 随机数法
- 处理冲突的方法
  - 开放定址法
  - 再哈希法
  - 链地址法
  - 建立公共溢出区
- 哈希表的查找及其分析

9.4.1 哈希表的概念

- **哈希查找** 又叫散列查找，利用哈希函数进行查找的过程。
  - **基本思想：** 在数据元素的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法。
- **哈希函数** 在数据元素的**关键字**与**存储地址**之间建立的一种对应关系。哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象。
  - 可写成，$addr(a_i) = H(k_i)$；
  - 其中：a_i是表中的一个元素，$addr(a_i)$ 是 a_i 的存储地址，k_i 是 a_i 的关键字。

9.4.1 哈希表的概念

☐ 哈希表

- 根据设定的**哈希函数** H(key) 和所选中的**处理冲突的方法**，将一组关键字映象到一个有限的、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**”。

9.4.1 哈希表的概念

□例　34个地区的各民族人口统计表

编号	地区名	总人口	汉族	回族	…
1	北京				
2	上海				
⋮	⋮				

以编号作关键字，构造哈希函数：H(key)=key
H(1)=1
H(2)=2

以地区名作关键字，取地区名称第一个拼音字母的序号作哈希函数：H(Beijing)=2
H(Shanghai)=19
H(Shandong)=19

9.4.1 哈希表的概念

- 哈希函数只是一种**映象**，所以哈希函数的设定很灵活，只要使任何关键字的哈希函数值都落在表长允许的范围之内即可。
- **冲突**
  - Key1 ≠ key2，但 H(key1) = H(key2) 的现象，称为冲突。
  - 例如 H(Shanghai) = H(Shandong)=19。

9.4.1 哈希表的概念

- 是否可以完全避免哈希冲突?
  - 一般来说，只能**尽量减少冲突而不能完全避免冲突**；
  - **原因**：这是因为通常关键字集合比较大，其元素包括所有可能的关键字，而地址集合的元素仅为哈希表中的地址值。
  - **构造原则**：在定义哈希表时既要定义好哈希函数又要给出处理冲突的方法。

9.4.2 哈希函数构造的方法

- 直接定址法
- 数字分析法
- 平方取中法
- 折叠法
- 除留余数法
- 随机数法

9.4.2 哈希函数构造—直接定址法

- 哈希函数为关键字的**线性函数**
  - H(key) = key 或者 H(key) = a × key + b
  - 其中a和b为常数
- 此法仅适合于：
  - 地址集合的大小 = = 关键字集合的大小

地址	1	2	3	…
年龄	1	2	3	…
人数	…			
…				

按年龄进行人口统计

H(Key) = Key

地址	1	2	3	…
年份	1949	1950	1951	…
人数	…			
…				

按年份进行人口统计

H(Key) = Key - 1948

9.4.2 哈希函数构造—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** $(u_1, u_2, \ldots, u_s)$
- 分析全体关键字，并从中**提取分布均匀的若干位或其组合**作为地址。
- 此法适于**能预先估计**出全体关键字的每一位上各种数字出现的频度。

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

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

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**

9.4.2 哈希函数构造—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”
  - 平方值的中间各位能受到整个关键字各位的影响。
- 此方法适合于
  - 关键字中的每一位都有某些数字重复出现频度很高的现象。

关键字		地址
8 1 3 4 6 5 3 2		6617258**26**8427024
8 1 3 7 2 2 4 2		6621441**76**8106564
8 1 3 8 7 4 2 2		6623912**45**9806084
8 1 3 0 1 3 6 7	⇒	6609912**27**6068689
8 1 3 2 2 8 1 7		6613400**56**4815489
8 1 3 3 8 9 6 7		6616027**55**2627089
8 1 3 6 8 5 3 7		6620838**81**3520369
8 1 4 1 9 3 5 5		6629111**36**8616025

9.4.2 哈希函数构造—折叠法

- 将关键字分割成若干部分，然后取叠加和为哈希地址。
- 两种叠加处理的方法：
  - **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。
- 此法适于关键字的数字位数特别多。

例 关键字为：04 4220 5864，哈希地址位数为4

```
  5864
  4220
    04
------
 10088
```

移位叠加

H(key)=0088

```
 5864
 0224
   04
-----
 6092
```

间界叠加

H(key)=6092

9.4.2 哈希函数构造—除留余数法

- 设定哈希函数为：
  - H(key) = key MOD p (p ≤ m)
  - 其中，m 为表长，p 为不大于 m 的质数或是不含 20 以下质因子的合数
- 为什么要对 p 加限制？ （通过示例说明）
  - 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

9.4.2 哈希函数构造—随机数法

- 设定哈希函数为
  - H(key) = Random(key)
  - 其中，Random 为伪随机函数
- 此法用于对**长度不等**的关键字构造哈希函数。
- **选取哈希函数考虑的因素**
  - 计算哈希函数所需时间
  - 关键字长度
  - 哈希表长度（哈希地址范围）
  - 关键字分布情况
  - 记录的查找频率

9.4.3 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。
- 具体方法
  - 开放定址法
  - 再哈希法
  - 链地址法
  - 建立公共溢出区

9.4.3 处理冲突—开放定址法

- 为产生冲突的地址 H(key) 求得一个地址序列： $H_1, H_2, \ldots, H_s$, $1 \leq s \leq m-1$
  - $H_i = (H(key) + d_i)$ MOD m, $i = 1, 2, \ldots, s$
  - H(key)为哈希函数；m为哈希表长；d_i 为增量序列
- 对增量 d_i 的三种取法
  - 线性探测再散列
  - 二次探测再散列
  - 随机探测再散列

9.4.3 处理冲突—开放定址法

1. 线性探测再散列

问题：二次聚集

- $d_i = c \times i$　最简单的情况　$c = 1$
- 哈希函数：H(key)=key　MOD　11，现有第4个记录，其关键字为38，利用线性探测再散列解决插入冲突

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

H(38) = 38 MOD 11 = 5　冲突

H1 = (5 + 1) MOD 11 = 6　冲突

H2 = (5 + 2) MOD 11 = 7　冲突

H3 = (5 + 3) MOD 11 = 8　不冲突

9.4.3 处理冲突—开放定址法

□ 二次探测再散列

- $d_i = 1^2, -1^2, 2^2, -2^2, \ldots$
- 哈希函数：H(key)=key MOD 11，现有第4个记录，其关键字为38，利用二次探测再散列解决插入冲突

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

H(38) = 38 MOD 11 = 5　　冲突

H1 = $(5 + 1^2)$ MOD 11 = 6　　冲突

H2 = $(5 - 1^2)$ MOD 11 = 4　　不冲突

要求：二次探测时的表长 m 必为形如 4j+3 的素数 (7, 11, 19, 23, ...)

9.4.3 处理冲突—开放定址法

□ 随机探测再散列

- d_i 是一组伪随机数列 或 $d_i = i \times H_2(key)$
- 哈希函数：H(key)=key MOD 11，现有第4个记录，其关键字为38，利用随机探测再散列解决插入冲突

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

H(38) = 38 MOD 11 = 5　　冲突

设伪随机数序列为9，则：

H1 = (5+9) MOD 11=3　　不冲突

要求：随机探测时的表长 m 和 增量 d_i 不能有公因子

9.4.3 处理冲突—开放定址法

- 给定关键字集合构造哈希表 (19, 01, 23, 14, 55, 68, 11, 82, 36)，设定哈希函数 H(key) = key MOD 11 (表长=11)
  - 若采用线性探测再散列处理冲突

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		

  - 若采用二次探测再散列处理冲突

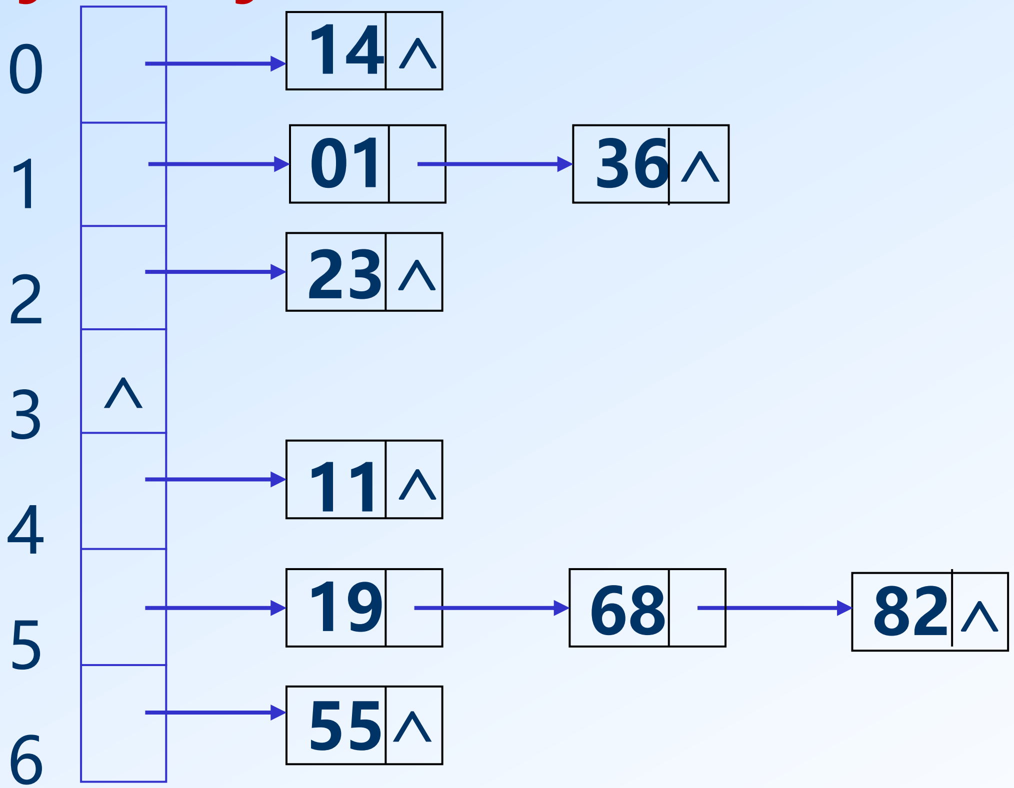
0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11
1	1	2	1	2	1	4		1		3

9.4.3 处理冲突—再哈希法

- **方法**：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：$H_i = Rh_i(key) \quad i=1, 2, ..., K$
  - 其中：Rh_i——不同的哈希函数
- **缺点**：**计算时间增加**

9.4.3 处理冲突—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。
  - 给定关键字 (19, 01, 23, 14, 55, 68, 11, 82, 36)，哈希函数为 H(key)=key MOD 7



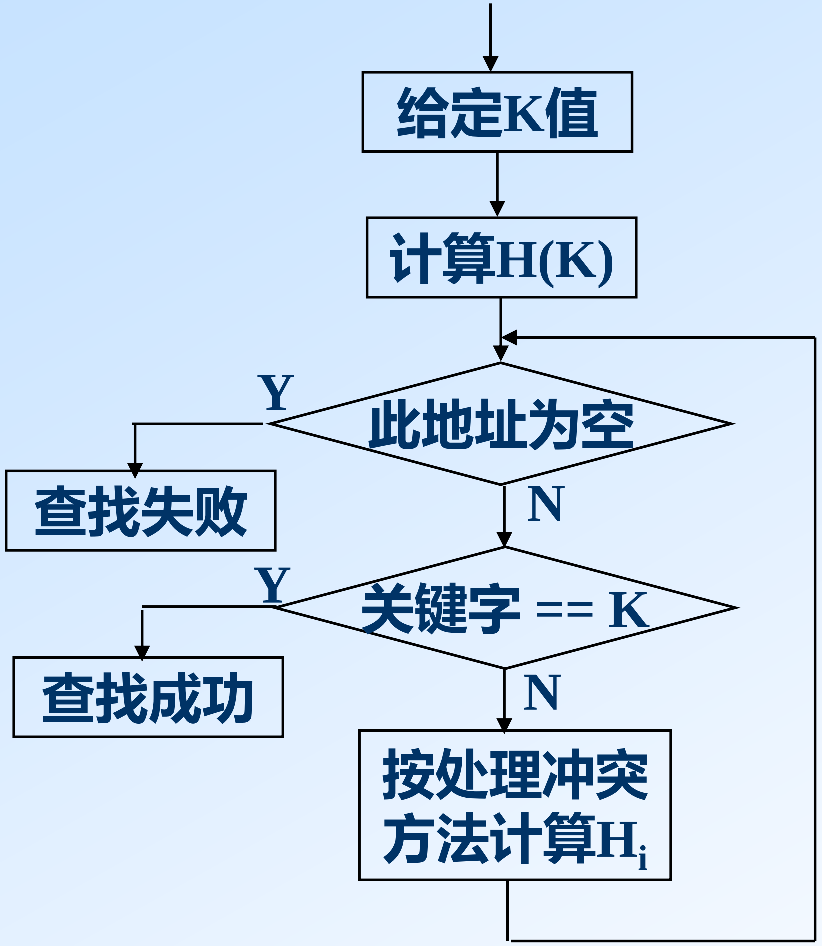
- 优点：非链地址处理冲突哈希表，**删除记录时需填入特殊符号，**而不是直接删除
- 缺点：每次定位到顶点结点后还要进行单链表的查找

9.4.3 处理冲突—公共溢出区

- **建立公共溢出区**
  - 设哈希函数的值域为 [0, m-1]
  - 设向量 HashTable[0..m-1] 为基本表，每个分量存放一个记录
  - 另设立向量 OverTable[0..v] 为溢出表
  - 将所有关键字**冲突的记录都填入溢出表**。

9.4.4 哈希表的查找及其分析

哈希查找过程



对于给定值 K，

计算哈希地址 i = H(K)

1. 若 r[i] == NULL，则查找失败
2. 若 r[i].key == K，则查找成功
3. 否则 “求下一地址 H_i”

直至下面两种情况之一为止

- $r[H_i]$ == NULL（查找失败）
- $r[H_i]$.key == K（查找成功）

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(key) = key MOD 13，哈希表长为m = 16， 用线性探测再散列处理冲突得哈希表

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

给定值K = 84的查找过程为:

1. H(84) = 6，不空且不等于84，冲突
2. H1 = (6+1) MOD 16 = 7，不空且不等于84，冲突
3. H2 = (6+2) MOD 16 = 8，不空且等于84，查找成功，返回记录在表中的序号。

给定值K = 38的查找过程为:

1. H(38) = 12 不空且不等于38，冲突
2. H1 = (12+1)MOD 16 = 13，空记录

表中不存在关键字等于 38 的记录，查找不成功。

9.4.4 哈希表查找的分析

- 从查找过程得知，由于“冲突”的产生，使哈希表查找过程仍然是与关键字比较的过程。
- 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. 哈希表饱和的程度
   - 装载因子 $\alpha = n / m$ 值的大小
   - n—表中填入的记录数，m—表的长度

9.4.4 哈希表查找的分析

1. 选用的哈希函数（可以不考虑该因素）
   - 一般地，选用的哈希函数是“均匀”的，对同一组随机关键字，产生冲突的可能性相同。
2. 选用的处理冲突的方法
   - 相同哈希函数，不同处理冲突方法，ASL不同
   - 线性探测处理冲突时，ASL = 22/9

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		

   - 二次探测处理冲突时，ASL = 16/9

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

9.4.4 哈希表查找的分析

- 哈希表饱和的程度，**装载因子 α**，**则查找成功时ASL为：**
  - 线性探测再散列：$S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$
  - 二次探测再散列：$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$
  - 链地址法：$S_{nc} \approx 1+\frac{\alpha}{2}$
- **哈希表特点：**
  - 哈希表的**ASL是关于 α 的函数**，而不是 n 的函数
  - 用哈希表构造查找表时，选择适当的装填因子 α，可使平均查找长度ASL限定在一个范围内。