


10-查找

C生万物 ● 大道至简 ● 鲍鱼科技+v(15339278619)

1、目标

 本节是一个重要的专题，也是拉开分数的地方，掌握好本章的内容是能够考高分的


掌握数组顺序查找：顺序查找、折半查找

掌握树形结构查找：BST、AVL、RB、B、B+ (重点)

掌握字典查找：hashTable

一定要会分析平均查找长度（这是考试的重点和难点）




 在查找中，有一个非常重要的概念：**平均查找长度 ASL - Age Search Length**，这是衡量查找算法效率的重要指标。

一次查找长度：一次查找过程中关键字的比较次数

平均查找长度：所有查找过程中进行关键字的比较次数的平均值


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

 n 是表的长度， p_i 是查找第 i 个数据元素的概率，如果概率相等，则 $p_i=1/n$ ， c_i 是找到第 i 个数据元素所需比较的次数。

一、连续空间查找

1、顺序查找

- 一般线性表的顺序查找

 • 一般线性表的顺序查找：即为按照顺序一一比较，实现简单，效率不高，要求会计算查找成功或不成功的平均长度

对于有 n 个元素的表，给定值 key 与表中第 i 个元素相等，即定位第 i 个元素时，需进行 $n-i+1$ 次关键字的比较，即 $C_i=n-i+1$ 。查找成功时，顺序查找的平均长度为


$$ASL_{成功} = \sum_{i=1}^n P_i (n-i+1)$$

当每个元素的查找概率相等，即 $P_i = 1/n$ 时，有

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

查找不成功时，与表中各关键字的比较次数显然是 $n+1$ 次，即 $ASL_{不成功} = n+1$ 。

- 有序表的顺序查找

 有序表的顺序查找：可以利用好有序的特性，降低查找长度，从而提高查找效率

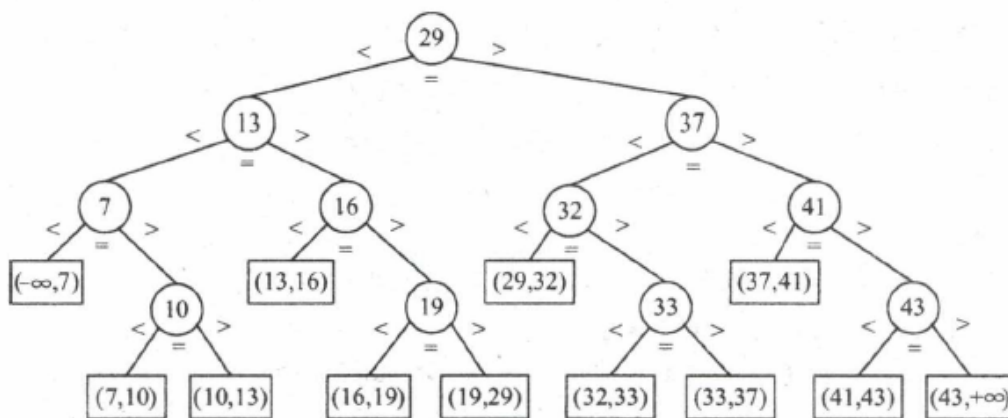


图 7.2 描述折半查找过程的判定树

由上述分析可知，用折半查找法查找到给定值的比较次数最多不会超过树的高度。在等概率查找时，查找成功的平均查找长度为

$$ASL = \frac{1}{n} \sum_{i=1}^n l_i = \frac{1}{n} (1 \times 1 + 2 \times 2 + \dots + h \times 2^{h-1}) = \frac{n+1}{n} \log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

式中， h 是树的高度，并且元素个数为 n 时树高 $h = \lceil \log_2(n+1) \rceil$ 。所以，折半查找的时间复杂度为 $O(\log_2 n)$ ，平均情况下比顺序查找的效率 high。

在图 7.2 所示的判定树中，在等概率情况下，查找成功（圆形结点）的 $ASL = (1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 4) / 11 = 3$ ，查找不成功（方形结点）的 $ASL = (3 \times 4 + 4 \times 8) / 12 = 11/3$ 。

📌 对于有序顺序查找和二分查找，记住一个求解 ASL 的口诀：

【查找成功看顶点，查找失败去看边】

二、二叉树形查找

1、二叉排序树 BST

📌 BST 概念：

二叉排序树(也称二叉查找树)或者是一棵空树，或者是具有下列特性的二叉树：

- 1、若左子树非空，左子树上所有结点的值均小于根结点的值。
- 2、若右子树非空，则右子树上所有结点的值均大于根结点的值。
- 3、左、右子树也分别是一棵二叉排序树。

📌 为什么叫二叉排序树：

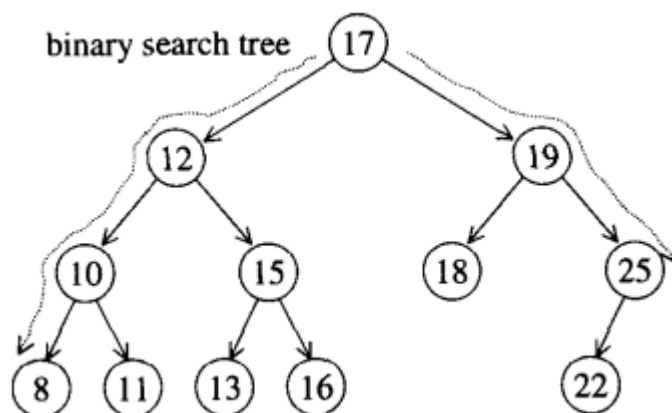
因为对二叉树进行中序遍历，将得到从小到大的排序顺序

📌 二叉排序树的重要性：

- 1、BST是所有平衡树的基础，尤其是AVL和RB都是在BST的基础上加以平衡条件限制实现的。
- 2、二叉搜索树最大的功劳在于：**规定了节点的位置**，因此针对BST可以有效实现查找、插入、删除，让树形结构可以进行动态调整

📌 二叉搜索树的操作：

- 1、构造BST
- 2、查找节点：值小往左走，值大往右走，相等查找成功，到达叶子节点则失败
- 3、插入节点：先查找插入位置，**永远在叶子节点的位置插入**
- 4、删除节点：永远将真正要删除的节点转换为**最多只有一棵子树**的节点



• BST的ADT

```
1 #define T int
2 typedef struct BSTNode
3 {
4     T data;
5     struct BSTNode *leftChild;
6     struct BSTNode *rightChild;
7 }BSTNode;
8
9 typedef BSTNode* BST;
10 //在BST中插入值x
11 void InsertBST(BST *t, T x);
12 //删除关键值为key的节点
13 void RemoveBST(BST *t, T key);
14 //求BST中最小值
15 T Min(BST t);
```


```

16 //求BST中最大值
17 T      Max(BST t);
18 //对BST进行中序排序
19 void    InOrder(BST t);
20 //在BST中查找key
21 BSTNode* Search(BST t, T key);
22 //释放BST树
23 void    DestroyBST(BST *t);

```

2、高度平衡二叉树 AVL

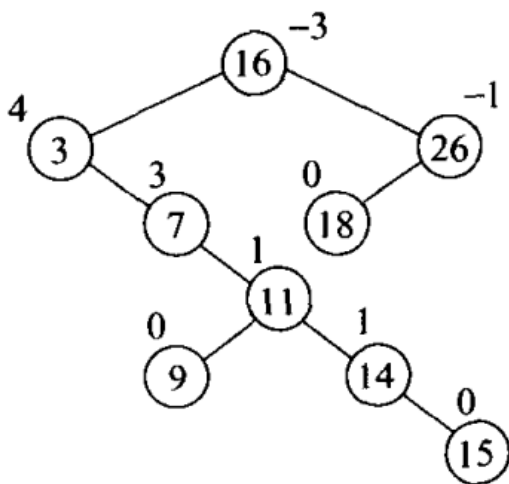
- AVL概念

 首先AVL树必须是BST树

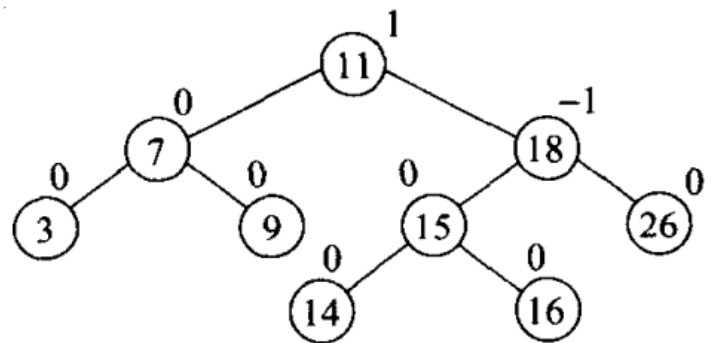
AVL的平衡条件为：任意节点的左右子树的高度差不能超过1，及 $|bf| \leq 1$

平衡因子bf的值：一般使用右树高度减去左树高度


高度差为平衡因子bf：值只能为-1，0，1




(a) 高度不平衡的二叉搜索树

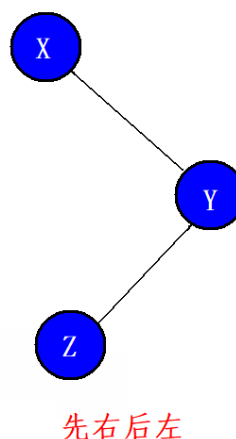
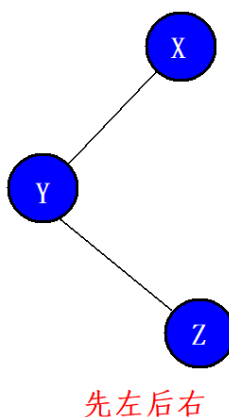
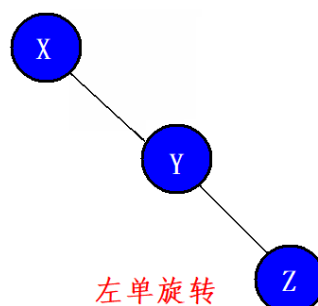
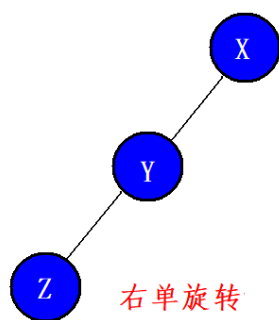


(b) 高度平衡的二叉搜索树


-  1、AVL树最重要的就是要保持平衡，如果树形结构不平衡，则需要调整平衡
- 2、能影响AVL树平衡的因素只有两个，**插入节点**和**删除节点**
- 3、如果插入节点或者删除节点导致AVL不平衡，则需要调整平衡，调整的手段为**旋转节点**
- 4、旋转分为：**单旋转**（左单旋转、右单旋转）、**双旋转**（先左后右、先右后左）

- AVL旋转情形


 AVL树所有的不平衡方式最终只有以下四种，所有的不平衡调整只需从不平衡的节点向下数两层进行调整即可




- AVL旋转举例

 将序列{16, 3, 7, 11, 9, 26, 18, 14, 15}，构造一棵AVL树

- AVL插入节点的过程

-  1、插入节点之前，必须保证AVL树是平衡的
- 2、查找新插节点的位置，新插节点为叶子节点， $bf=0$
- 3、修改父节点的平衡因子 bf
- 4、考察父节点的平衡因子，有三种情况：
 - 4.1 父节点 $bf=0$ ，结束调整
 - 4.2 父节点 $|bf|=1$ ，向根的方向回溯
 - 4.3 父节点 $|bf|=2$ ，发生不平衡，判断平衡因子的符号，进行旋转调整

- AVL删除节点

-  1、查找到需要删除的节点

2、删除节点并修改父节点的平衡因子bf

3、考察父节点的平衡因子，有三种情况：

3.1 父节点 $|bf|=1$, 结束调整

3.2 父节点 $|bf|=0$, 向根的方向回溯


3.3 父节点 $|bf|=2$, 发生不平衡，此时令q指向父节点的较高子树，情况又分为三种：

3.3.1 q的bf=0, 执行一个单旋转恢复平衡,结束调整

3.3.2 q的bf与父节点bf符号相同，执行一个单旋转恢复平衡，树高度降低，需要继续向上回溯

3.3.3 q的bf与父节点bf符号相反，执行一个双旋转恢复平衡，树高度降低，需要继续向上回溯

- AVL代码实现

 单独对AVL代码实现进行讲解


也可以看我的B站录屏讲解：

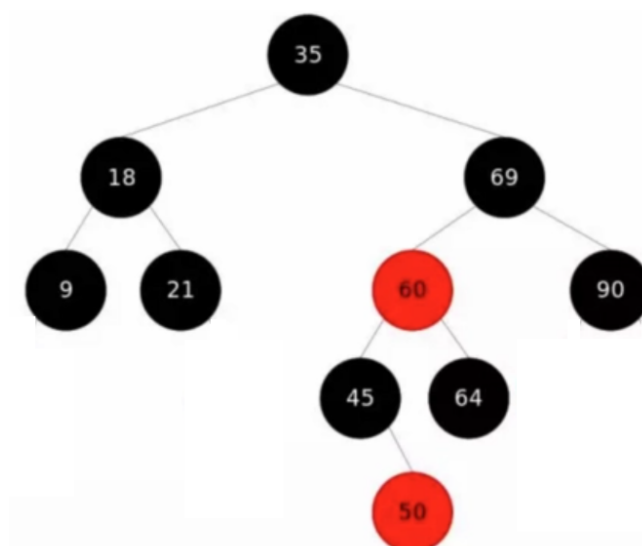
<https://www.bilibili.com/video/BV1bM411u7Ki?>


[p=62&vd_source=e1686c82128572ae175af1318c920cde](https://www.bilibili.com/video/BV1bM411u7Ki?p=62&vd_source=e1686c82128572ae175af1318c920cde)

3、红黑树 RBtree

- 红黑树概念

 AVL树是绝对平衡，红黑树的平衡条件相对较弱



 红黑树需要具备的性质：

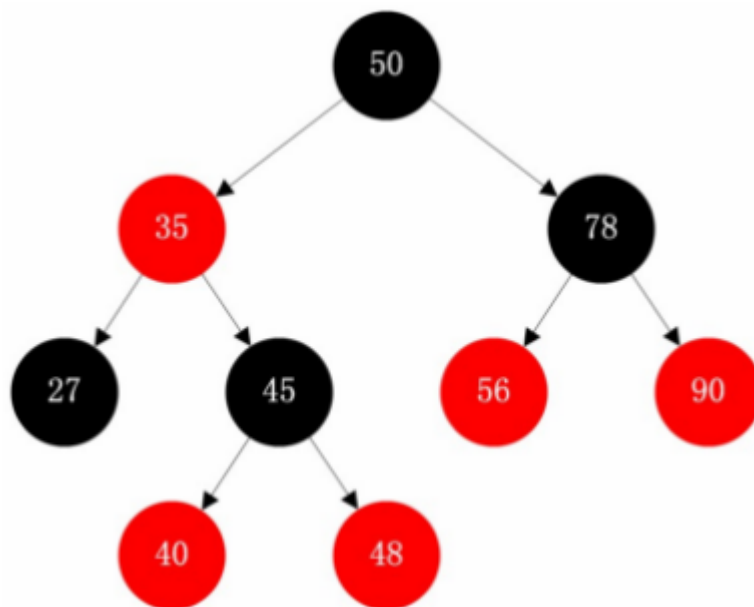
- 1、结点不是红色就是黑色
- 2、根结点必须是黑色
- 3、相邻的两个结点不能是同时为红色
- 4、从任意一结点到叶子结点的路径，黑色结点的个数必须相同
- 5、空指针或虚拟的外部结点为黑色

记住下面的口诀，一招搞定红黑树的概念：

【一头一脚黑，黑同红不连】

- 红黑树的性质

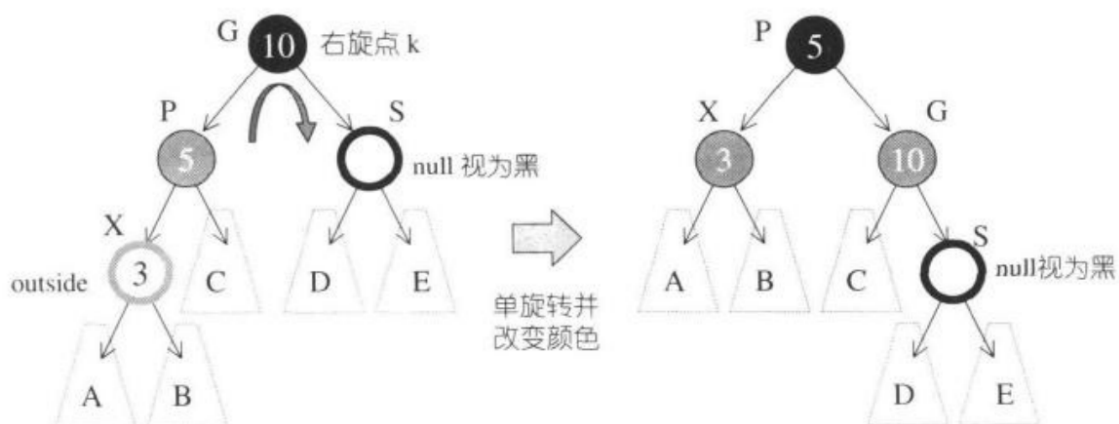
- 📌 1、根据性质4，红黑树插入节点以红色节点进行插入
- 2、从根到叶结点的最长路径不大于最短路径的2倍
- 3、有n个内部结点的红黑树的高度 $h \leq 2\log_2(n+1)$



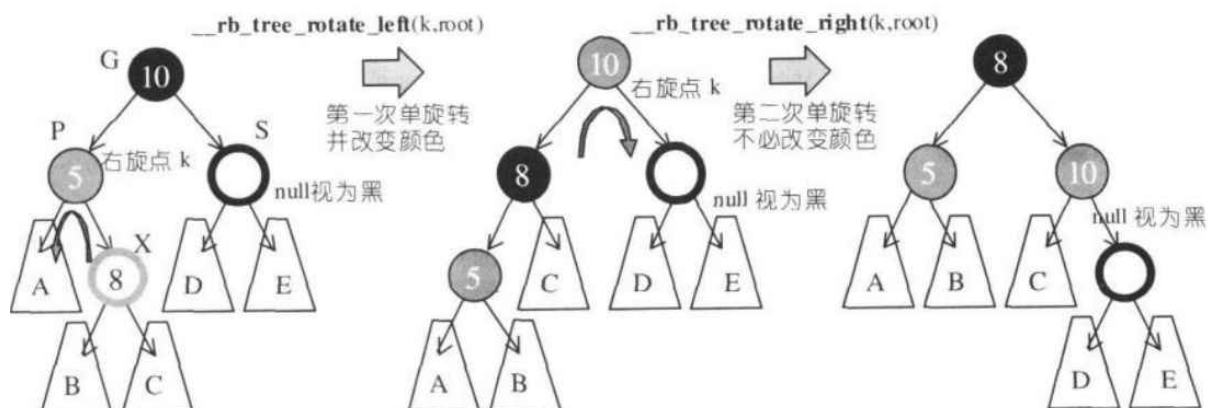
- 红黑树的平衡调整

📌 所有红黑树的不平衡情况只会有以下四种情况：

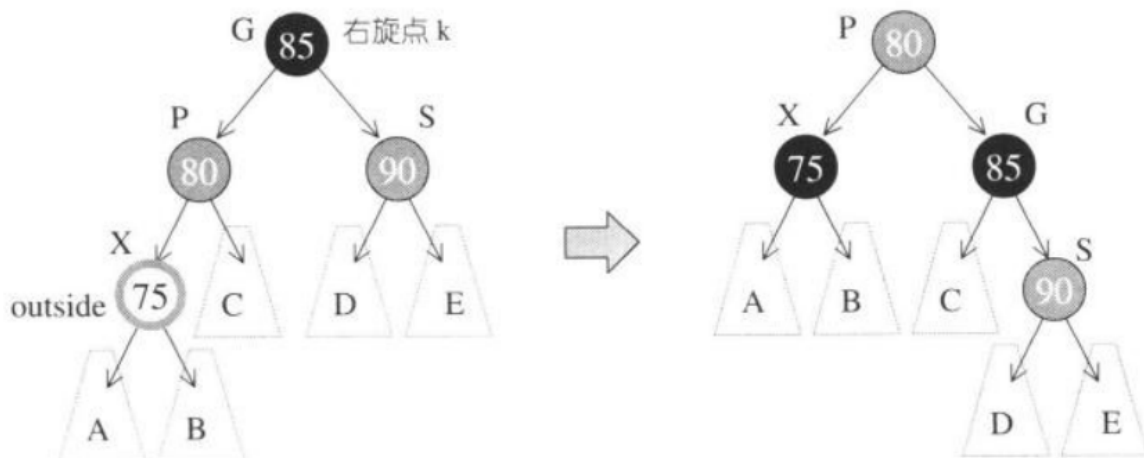
- 状况 1: S 为黑且 X 为外侧插入。对此情况，我们先对 P,G 做一次单旋转再更改 P,G 颜色，即可重新满足红黑树的规则 3。见图 5-15a。



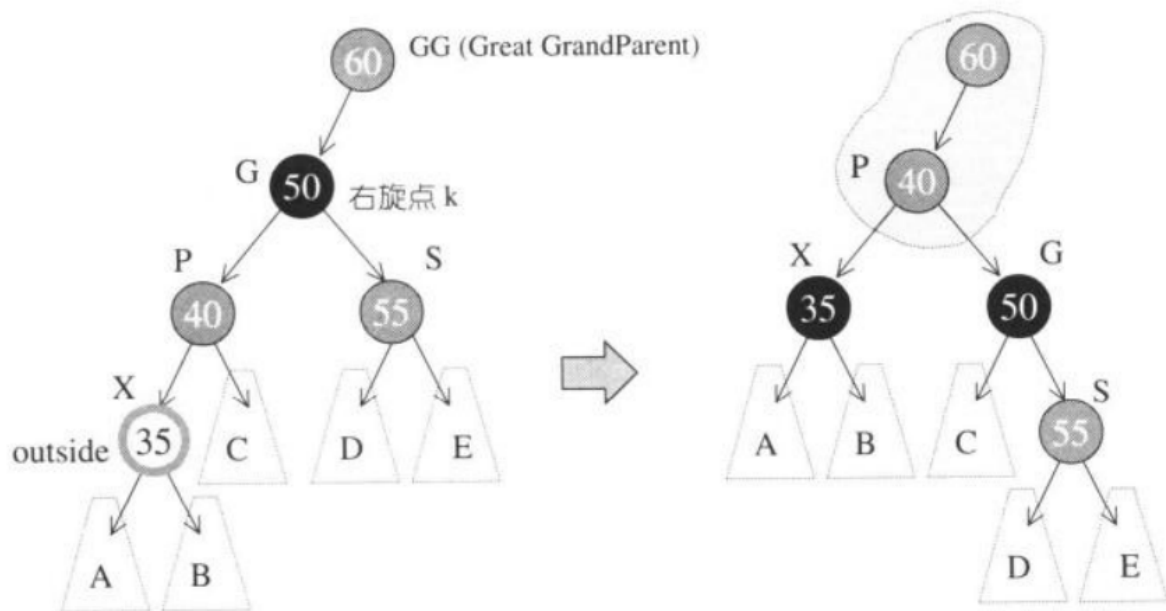
- 状况 2: S 为黑且 X 为内侧插入。对此情况，我们必须先对 P,X 做一次单旋转并更改 G,X 颜色，再将结果对 G 做一次单旋转，即可再次满足红黑树规则 3。见图 5-15b。



- 状况 3: S 为红且 X 为外侧插入。对此情况，先对 P 和 G 做一次单旋转，并改变 X 的颜色。此时如果 GG 为黑，一切搞定，如图 5-15c。但如果 GG 为红，则问题就比较大了，唔…见状况 4。



- 状况 4: S 为红且 X 为外侧插入。对此情况, 先对 P 和 G 做一次单旋转, 并改变 X 的颜色。此时如果 GG 亦为红, 还得持续往上做, 直到不再有父子连续为红的情况。

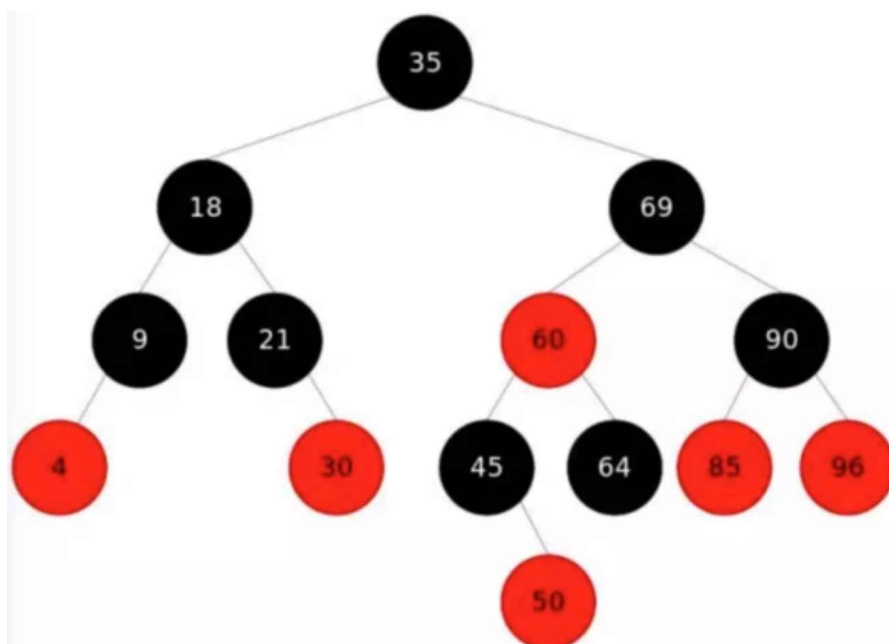



- 红黑树的构造举例

📌 将序列{16, 3, 7, 11, 9, 26, 18, 14, 15}, 构造一棵RB树


- 红黑树的插入删除


📌 有一个口诀: **插入看叔伯, 删除看兄弟**




 红黑树的插入如果出现不平衡，可以按照以上的四种状况进行平衡调整即可

 红黑树的删除调整比插入更为复杂，需要单独讨论：

 删除红黑树中一个结点，删除的结点是其子结点状态和颜色的组合，子结点的状态有三种：无子结点、只有一个子结点、有两个子结点，颜色有红色、黑色两种，所以共有6种组合。

 **组合1：被删结点无子结点，且被删结点为红色**

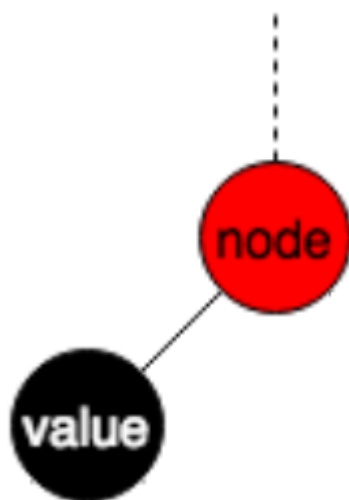
此时直接将结点删除即可，不破坏任何红黑树的性质。


 **组合2：被删结点无子结点，且被删结点为黑色**

处理方法略微复杂，稍后再议。

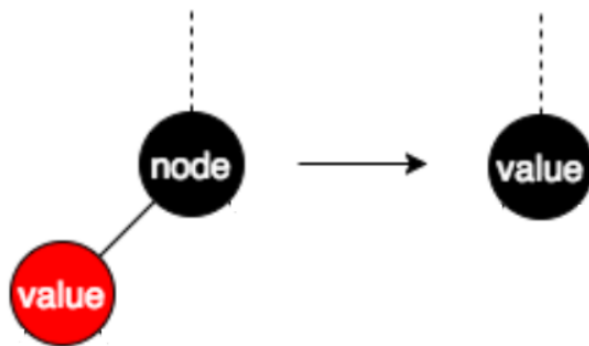
 **组合3：被删结点有一个子结点，且被删结点为红色**

这种组合是不存在的，如图，如果被删除的红色结点node有一个左子女结点value，那么在删除之前node一定会有一个黑色的右子女结点，否则将违背从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色结点的性质。



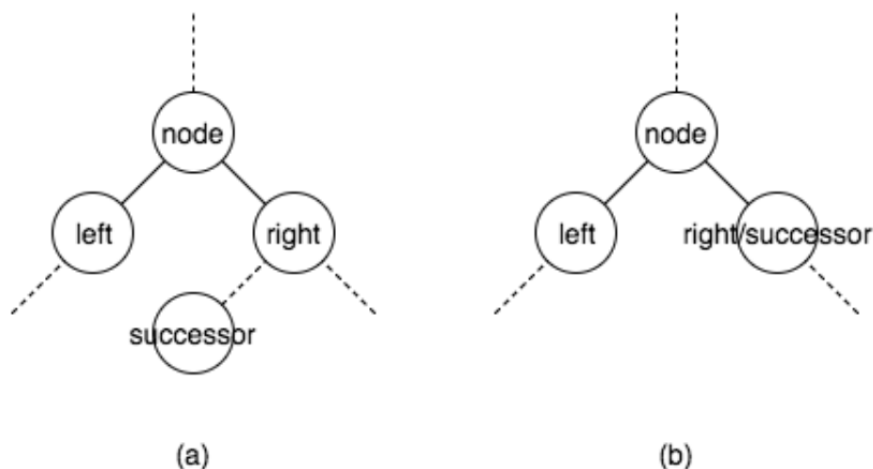
 **组合4：被删结点有一个子结点，且被删结点为黑色**

这种组合下，被删结点node的另一个子结点value必然为红色，此时直接将node删掉，用value代替node的位置，并将value着黑即可。



📌 组合5&6：被删结点有两个子结点，且被删结点为黑色或红色

当被删结点node有两个子结点时，先要找到这个被删结点的后继结点successor，然后用successor代替node的位置，此时相当于successor被删。因为node有两个子结点，所以successor必然在node的右子树中，必然是下图两种形态中的一种。



📌 若是(a)的情形，用successor代替node后，相当于successor被删，若successor为红色，则变成了组合1；若successor为黑色，则变成了组合2。

若是(b)的情形，用successor代替node后，相当于successor被删，若successor为红色，则变成了组合1；若successor为黑色，则变成了组合2或4。

📌 综上所述：

若被删结点是组合1或组合4的状态，很容易处理；被删结点不可能是组合3的状态；被删结点是组合5&6的状态，将变成组合1或组合2或组合4。所以，删除结点的6种情形实际上最终只有3种。

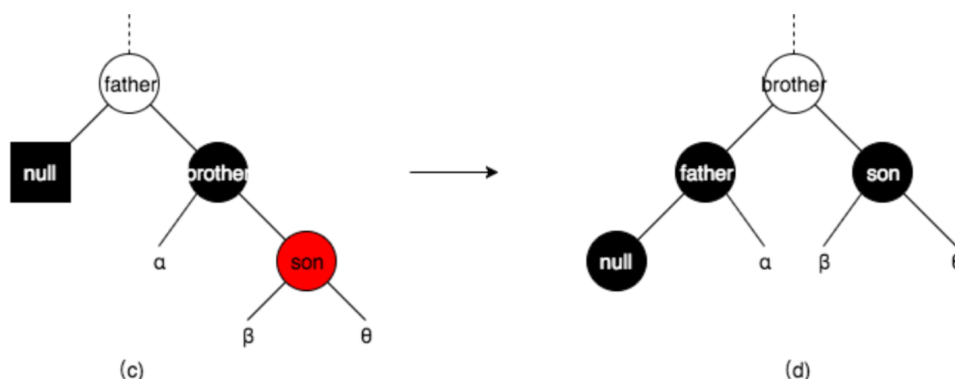
📌 再议组合2：被删结点无子结点，且被删结点为黑色

因为删除黑色结点会破坏红黑树的性质5，所以为了不破坏性质5，在替代结点上额外增加一个黑色，这样不违背性质5而只违背性质1，每个结点或是黑色或是红色。此时将额外的黑色移除，则完成删除操作。



情形一：

brother为黑色，且brother有一个与其方向一致的红色子结点son，所谓方向一致，是指brother为father的左子结点，son也为brother的左子结点；或者brother为father的右子结点，son也为brother的右子结点。

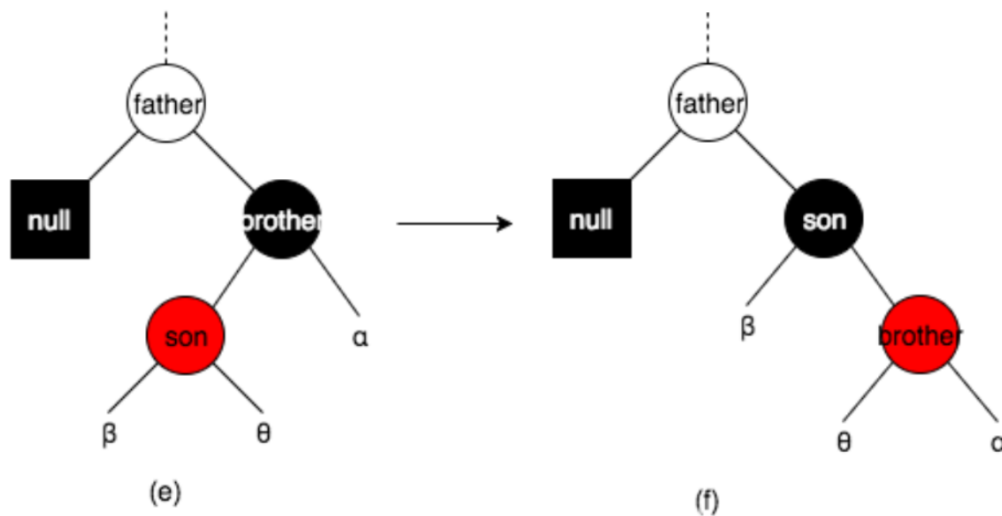


图(c)中，白色代表随便是黑或是红，方形结点除了存储自身黑色外，还额外存储一个黑色。将brother和father旋转，并重新上色后，变成了图(d)，方形结点额外存储的黑色转移到了father，且不违背任何红黑树的性质，删除操作完成。

图(c)中的情形颠倒过来，也是一样的操作。

情形二

brother为黑色，且brother有一个与其方向不一致的红色子结点son，但右子树有无红色节点无所谓

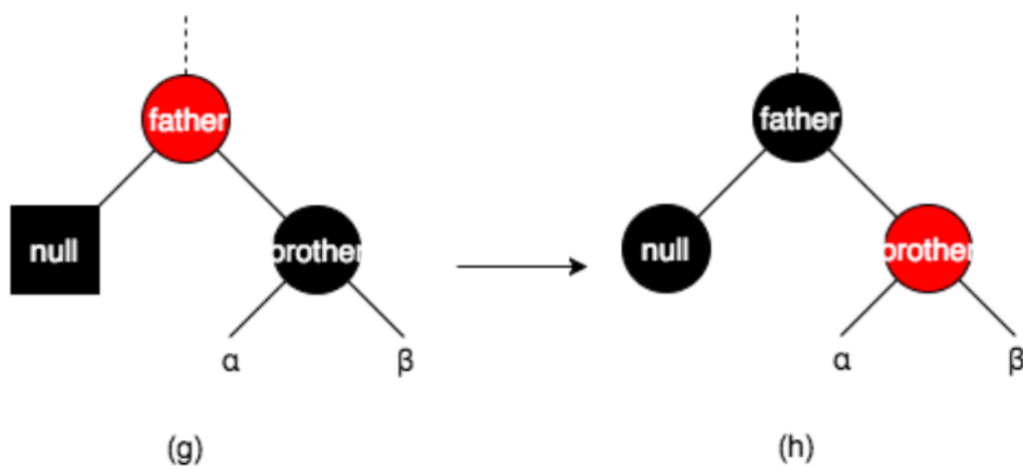


图(e)中，将son和brother旋转，重新上色后，变成了图(f)，情形一。

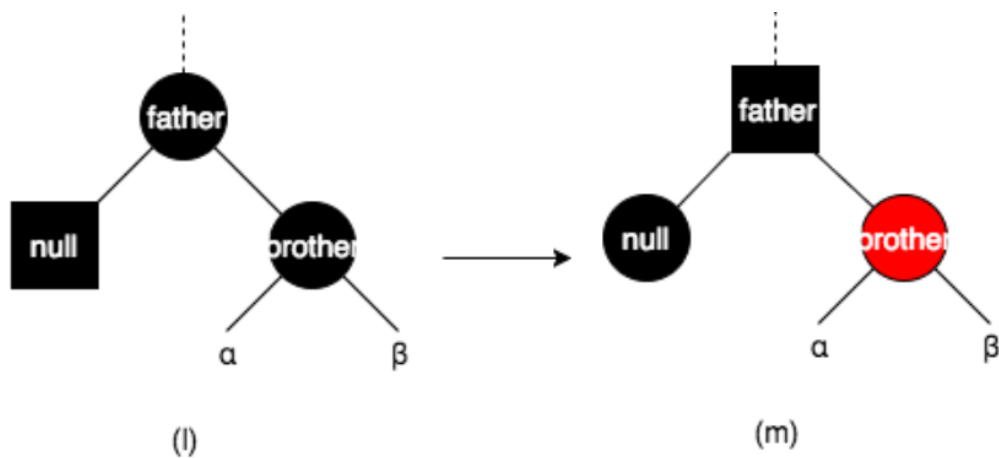
图(e)中的情形颠倒过来，也是一样的操作。

情形三

brother为黑色，且brother无红色子结点，此时若father为红，则重新着色，在继续向上回溯。如图下图(g)和(h)。

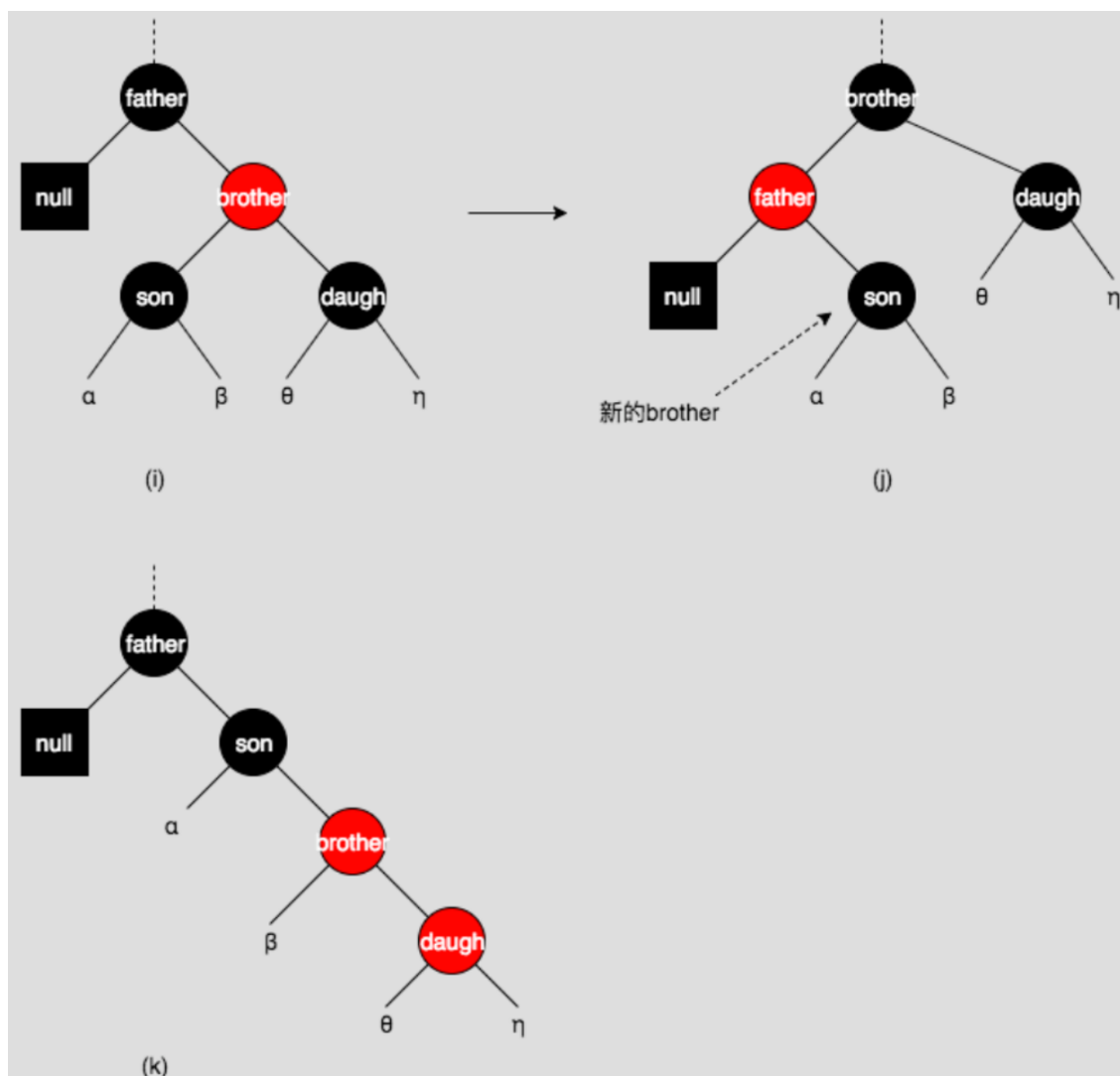


此时若father为黑，则重新着色，将额外的黑色存到father，将father作为新的结点进行情形判断(不删除)，遇到情形一、情形二，则进行相应的调整，完成删除操作；如果没有，则结点一直上移，直到根结点存储额外的黑色，此时将该额外的黑色移除，即完成了删除操作。



情形四

brother为红色，则father必为黑色。



图(i)中，将brother和father旋转，重新上色后，变成了图(j)，新的brother变成了黑色，这样就成了情形一、二、三中的一种。

如果将son和brother旋转，无论怎么重新上色，都会破坏红黑树的性质4或5，例如图(k)。图(i)中的情形颠倒过来，也是一样的操作。

• 红黑树代码实现

📌 单独对红黑树的代码实现进行讲解

三、m叉树形查找

📌 为什么会有m路搜索树？

人以“高、瘦”为美 - 高瘦对计算来说，查找效率不高
计算机偏钟爱“矮、胖” - 树形高度低，查询效率高

1、m路搜索树

动态 m 路搜索树是指系统运行过程中可以动态调整的能保持较高搜索效率的最多 m 路的搜索树,是 1972 年由 R. Bayer 和 E. McCreight 提出来的。动态 m 路搜索树的一般定义为：一棵 m 路搜索树，它或者是一棵空树，或者是满足如下性质的树。

① 根结点最多有 m 棵子树，并具有如下的结构：

$$n, P_0, (K_1, P_1), (K_2, P_2), \dots, (K_n, P_n)$$

其中, P_i 是指向子树的指针, $0 \leq i \leq n < m$; K_i 是关键码, $1 \leq i \leq n < m$ 。

② $K_i < K_{i+1}, 1 \leq i < n$ 。

③ 在子树 P_i 中所有的关键码都小于 K_{i+1} , 且大于 $K_i, 0 < i < n$ 。

④ 在子树 P_n 中所有的关键码都大于 K_n , 而子树 P_0 中的所有关键码都小于 K_1 。

⑤ 子树 P_i 也是 m 叉搜索树, $0 \leq i \leq n$ 。

例如,图 10.29 是一棵 3 路搜索树,它有 9 个关键码。每个结点最多有 3 棵子树,因而最多有 2 个关键码。但最少有 0 棵子树,有 1 个关键码。结点 a 的格式为 2, b , $(20, c)$, $(40, d)$, 结点 b 上所有关键码均小于 20, 结点 c 上所有关键码都介于 20~40 之间, 结点 d 上所有关键码都大于 40; 结点 c 的格式为 2, 0, $(25, 0)$, $(30, e)$, 结点 e 中所有的关键码均大于 30。

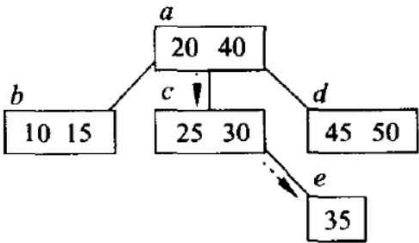


图 10.29 一棵 3 路搜索树的示例

📌 当 m 为 2 时, m 路搜索树就变成了二叉搜索树, 对比二叉搜索树, 就好理解 m 路搜索树

2、B 树

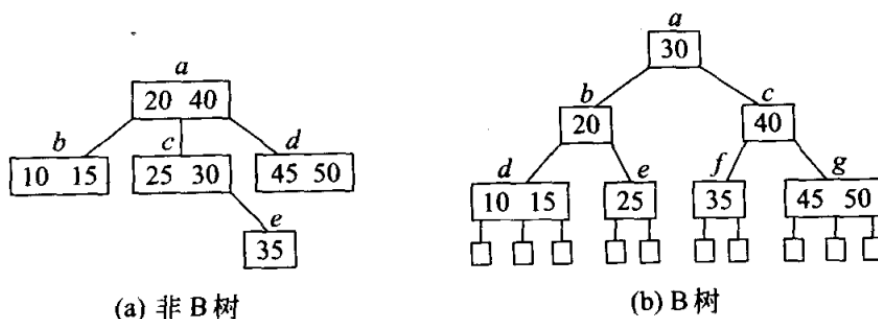
📌 所谓B树，就是平衡树的意思，B为Balance的简写

📌 • B树概念

一棵 m 阶 B 树 (balanced tree of order m) 是一棵平衡的 m 路搜索树，它或者是空树，或者是满足下列性质的树：

- ① 根结点至少有两个子女。
- ② 除根结点以外的所有结点 (不包括失败结点) 至少有 $\lceil m/2 \rceil$ 个子女。
- ③ 所有的失败结点都位于同一层。

注意，有的教科书上写成 B-树或者 B_树，不要误解成“B 减树”，“-”、“_”只是连字符。
图 10.30(a) 不是 B 树，图 10.30(b) 是 B 树，它的所有失败结点都在同一层上。



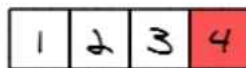
📌 • B树的构造举例

- 插入数据 - 分裂结点
- 删除数据 - 合并结点

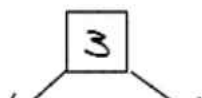
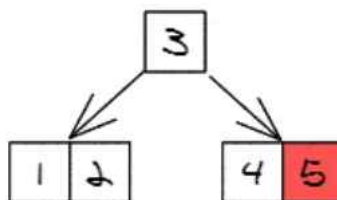
• 插入过程

下图是一个5阶B树，我们通过顺序插入1到17，来观察节点的分裂过程。

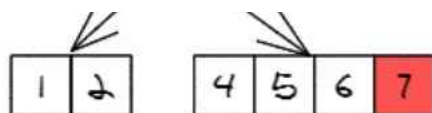
陆续插入1、2、3、4



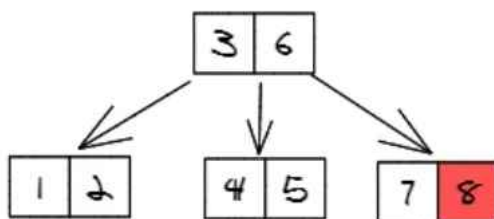
当插入5时，进行分裂：
将中数3上提至父节点，
1、2为左子节点，
4、5为右子节点



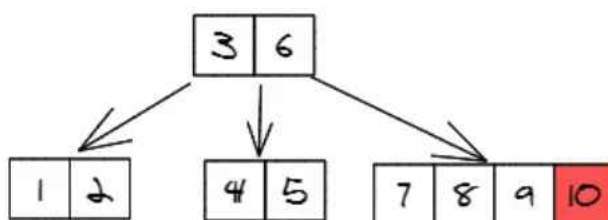
继续插入6、7



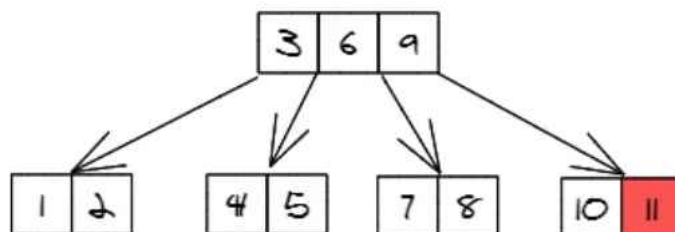
当插入8时，进行分裂：
将中数6上提至父节点，
4、5为左子节点，
7、8为右子节点



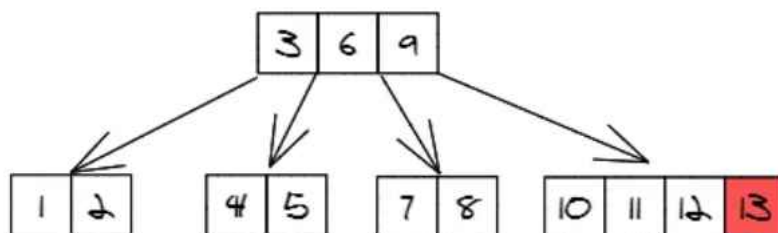
继续插入9、10



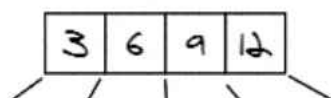
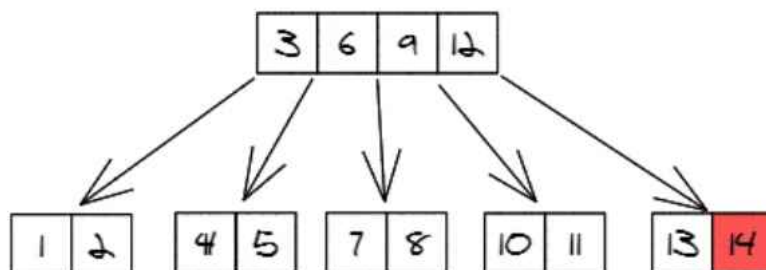
当插入11时，进行分裂：
将中数9上提至父节点，
7、8为左子节点，
10、11为右子节点



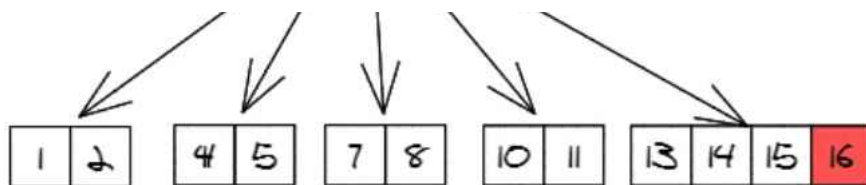
继续插入12、13



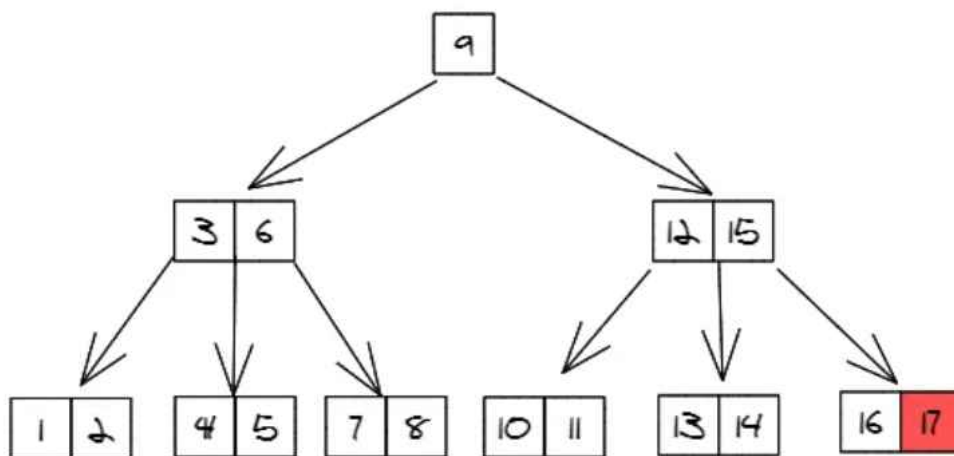
当插入14时，进行分裂：
将中数12上提至父节点，
10、11为左子节点，
13、14为右子节点



继续插入15、16



关键的一步来了，
当插入17时，进行分裂：
将中数15上提至父节点，
13、14为左子节点，
16、17为右子节点。



但将17上提父节点后，
其父节点也需要分裂：
将中数9上提至父节点，
3、6为左子节点，
13、15为右子节点。

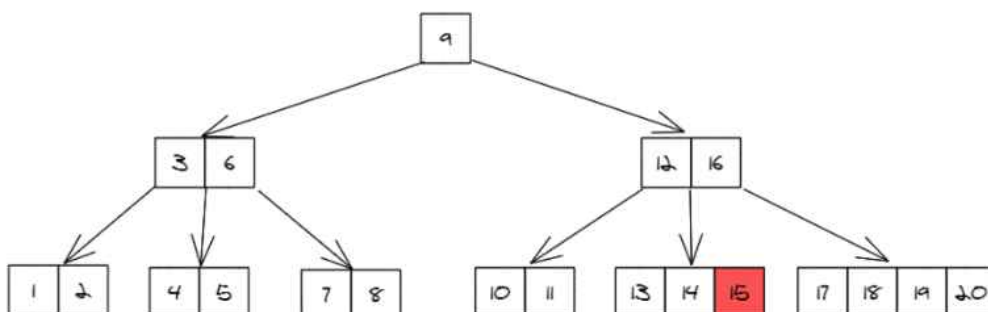
删除过程

B树的删除就复杂了许多，可分为下面几种情况：

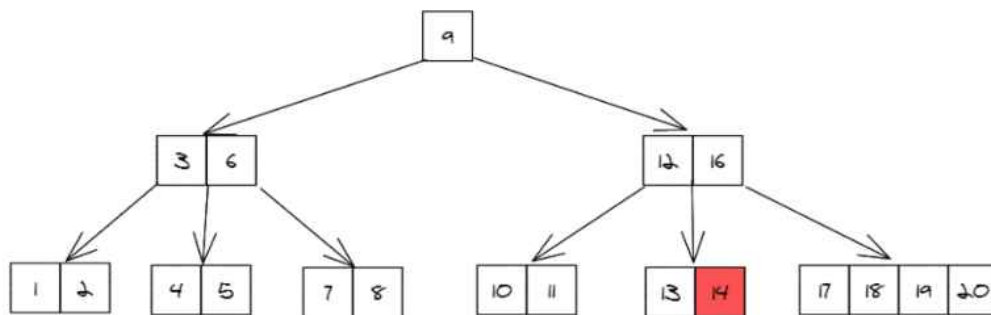
- 删除叶子节点中的元素 (1) 搜索要删除的元素 (2) 如果它在叶子节点上，直接将其删除 (3) 如果删除后产生了下溢出（键数小于最小值），则向其兄弟节点借元素。即将其父节点元素下移至当前节点，将兄弟节点中元素上移至父节点（若是左节点，上移最大元素；若是右节点，上移最小元素） (4) 若兄弟节点也达到下限，则合并兄弟节点与分割键。
- 删除内部节点中的元素 (1) 内部节点中元素为其左右子节点的分割值，需要从左子节点最大元素或右子节点最小元素中选出一个新的分割符。被选中的分割符从原子节点中移除，作为新的分隔值替换掉被删除的元素。 (2) 上一步中，若左右子节点元素数均达到下限，则合并左右子节点。 (3) 若删除元素后，其中节点元素数小于下限，则继续合并。

下图是一个5阶B树，我们通过删除15、14、17、5四个键，来观察删除过程（基本涵盖所有情况）。

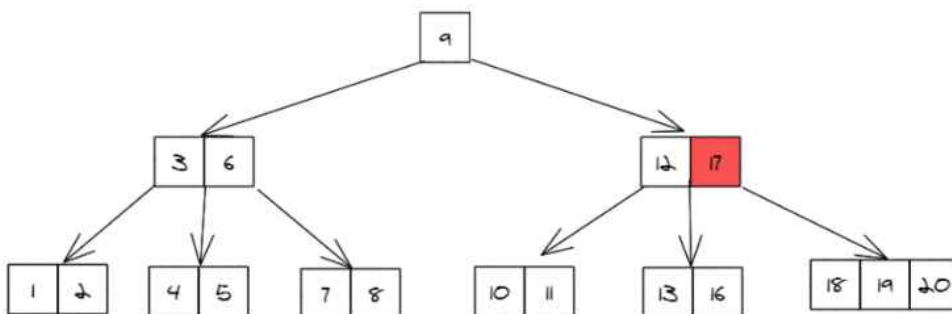
删除叶子节点中键15，
其节点中的元素数未小于2，
直接删除



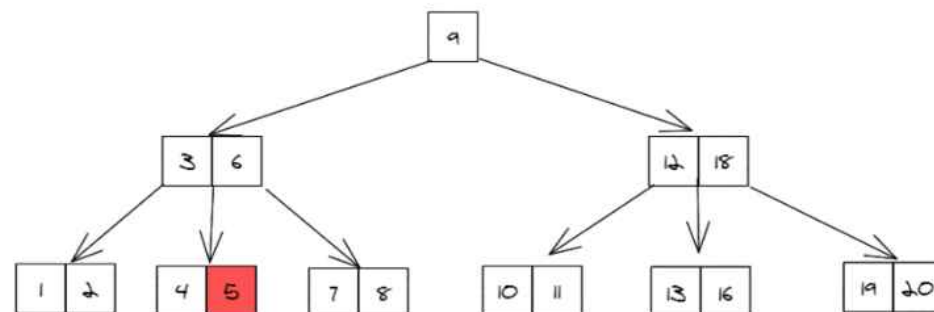
删除叶子节点中键14，
其节点中的元素数小于2，
则向元素多的右兄弟节点借，
将16下移，将17上移



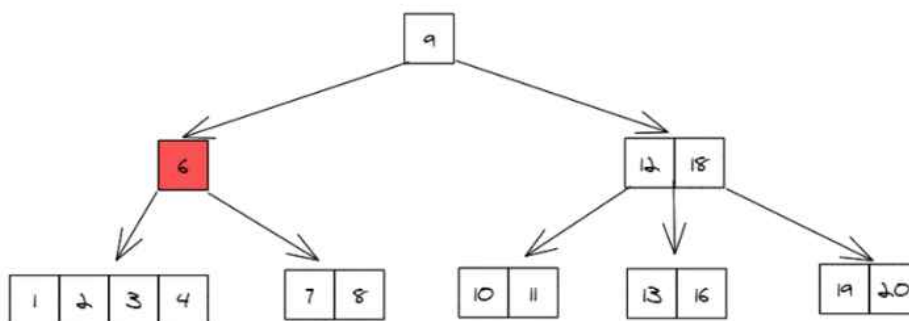
删除内部节点中键17，
需要将元素多的右子节点
中元素最小值18上移



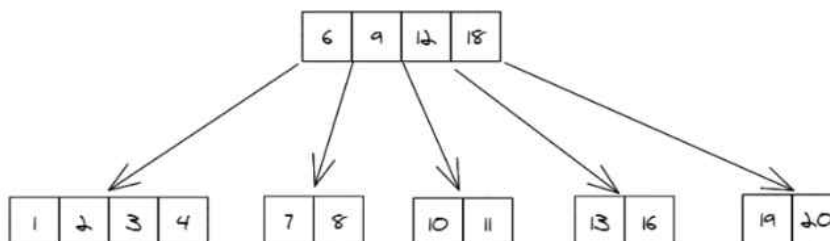
删除叶子节点中键5，
发现其左右兄弟节点都无法借，
需要与左兄弟节点、分隔值合并



上一步合并后，发现6所在节点
元素数小于2，则再合并



最终效果



3、B+树



• B+树的概念

B⁺树是1974年由Wedekind提出来的,它可以看作是B树的一种变形,在实现文件索引结构方面比B树使用得更普遍。

1. B⁺树的定义

一棵 m 阶B⁺树是B树的特殊情形,它与B树的不同之处在于:

(1) 所有关键码都存放在叶结点中,上层的非叶结点的关键码是其子树中最小(或最大)关键码的复写。

(2) 叶结点包含了全部关键码及指向相应数据记录存放地址的指针,且叶结点本身按关键码从小到大顺序链接。

关于每个非叶结点的结构,有两种方式处理。如果按下层结点“最小关键码复写”原则,则树中每个非叶结点中有 m 棵子树必有 $m-1$ 个关键码,关键码 $k_i (1 \leq i < m)$ 是指针 P_i 所指子树中最小的关键码,见图10.38。

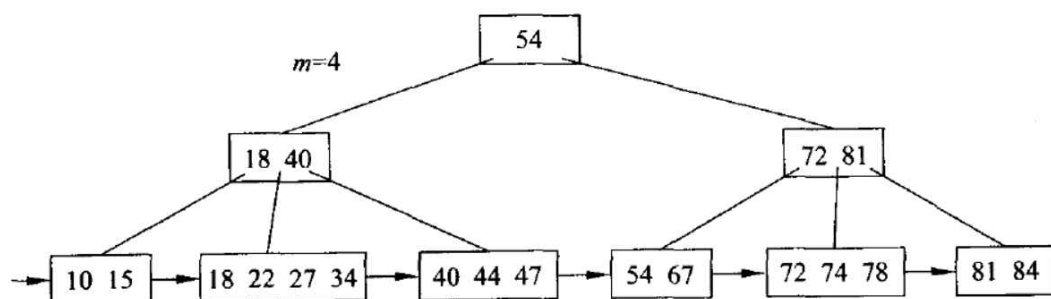


图 10.38 按“最小关键码复写”原则组织的一棵4阶B⁺树

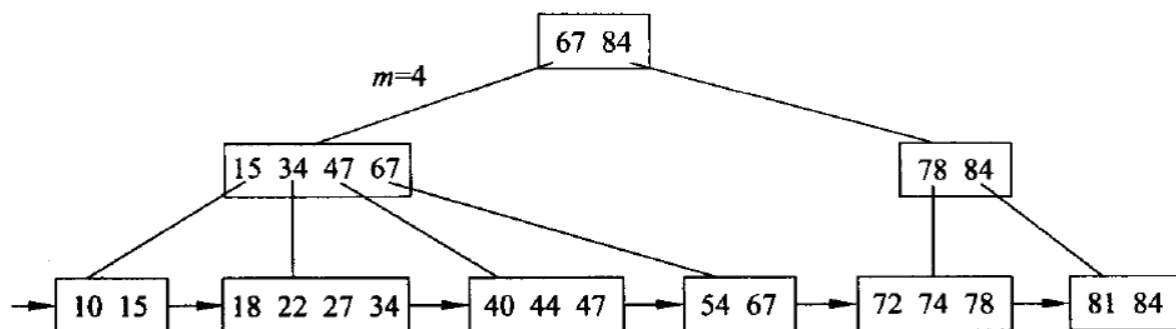


图 10.39 按“最大关键码复写”原则组织一棵4阶B⁺树

如果按下层结点“最大关键码复写”原则，则每个非叶结点中有 m 棵子树必有 m 个关键码。为讨论问题方便起见，本书按“最大关键码复写”原则定义 B^+ 树。

一棵 m 阶 B^+ 树的结构定义如下：

- (1) 每个结点最多有 m 棵子树(子结点)；
 - (2) 根结点最少有两个子树，除根结点外，其他结点至少有 $\lceil m/2 \rceil$ 个子树；
 - (3) 所有叶结点在同一层，按从小到大的顺序存放全部关键码，各个叶结点顺序链接；
 - (4) 有 n 个子树的结点有 n 个关键码；
 - (5) 所有非叶结点可以看成是叶结点的索引，结点中关键码 K_i 与指向子树的指针 P_i 构成对子树(即下一层索引块)的索引项 (K_i, P_i) ， K_i 是子树中最大的关键码。
- 叶结点中存放的是对实际数据记录的索引，每个索引项 (K_i, P_i) 给出数据记录的关键码及实际存储地址。

• B+树的构造过程举例

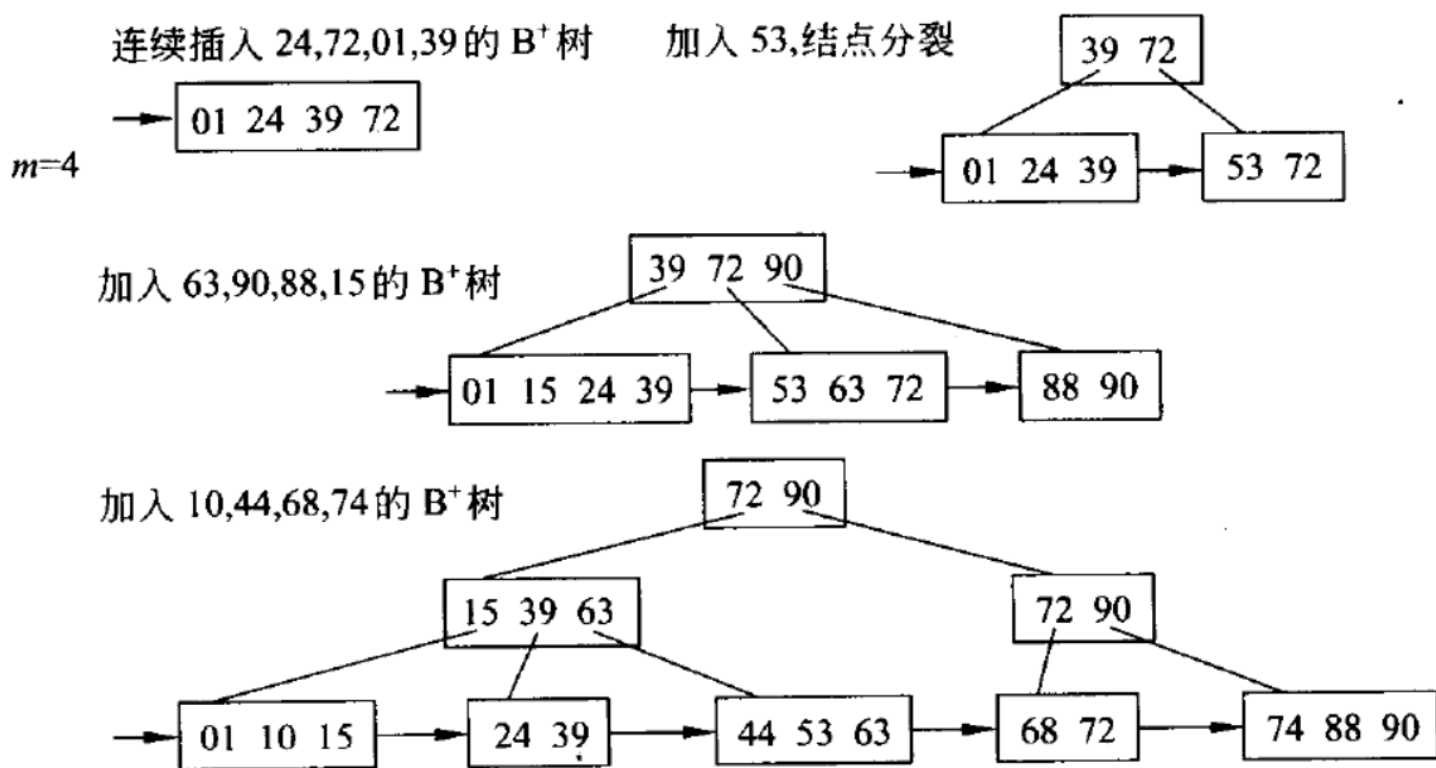



图 10.40 B^+ 树插入的示例

四、散列查找 - HashTable

1、哈希表概念

通过哈希函数，直接对关键字进行映射访问的表就称为**哈希表**

2、hashTable需要解决的两个问题

-  1、映射函数 --- hash函数 (除留余数法)
- 2、冲突解决 --- (链地址法)

3、常用哈希函数

要求了解每一种哈希函数的适用场景

1、直接定址法 - 适合关键字的分布基本连续的情况，若关键字分布不连续，空位较多，则会造成存储空间的浪费。


2、除留余数法 - 最简单、最常用的方法，可以把数据控制在一定的范围内

3、数字分析法 - 适合于已知的关键字集合，若更换了关键字，则需要重新构造新的散列函数

4、平方取中法 - 适用于关键字的每位取值都不够均匀或均小于散列地址所需的位数。

采用何种构造散列函数的方法取决于关键字集合的情况，但最终的**目标是尽量降低产生冲突的可能性**

4、解决冲突的方式

 **闭散列（开放定址法）** - 所有的数据都是在封闭空间内进行定位存储，不增加新的存储空间，需要手动掌握各种不同处理冲突的方式

1、线性探测 - 思路简单，但容易产生元素堆积

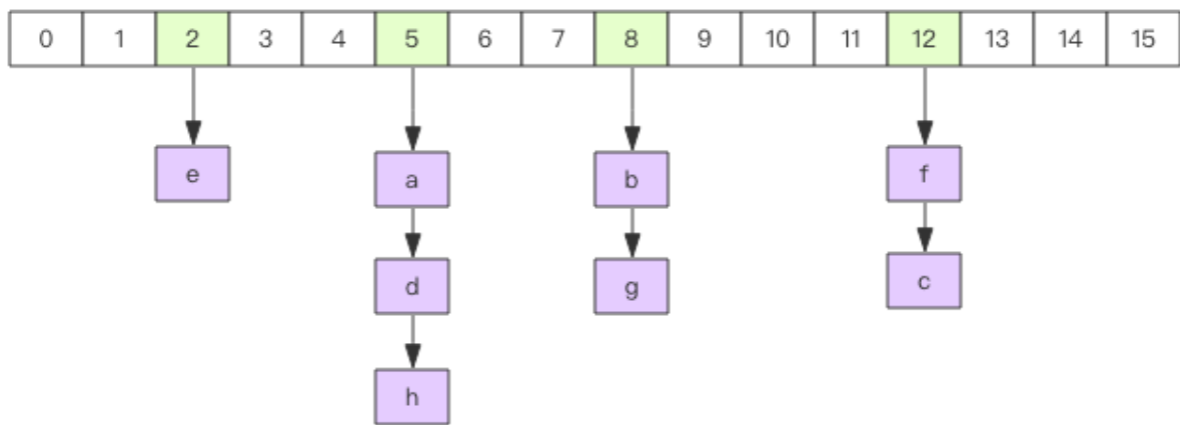
2、平方探测法 - 可以有效解决元素堆积问题，但只能探测一半空间，空间利用率不高

3、双散列法 - 是利用伪随机探测的思路的方法，跳跃式的探测，失败概率不好计算

4、伪随机数法 - 利用一个伪随机产生器，进行随机探测

开散列（拉链法） - 数据冲突时，申请新的空间进行数据的存储

顺序表+链表



5、HashTable的代码实现



B站视频具体讲解：

HashTable代码实现：

https://www.bilibili.com/video/BV1bM411u7Ki?p=68&vd_source=e1686c82128572ae175af1318c920cde

HashTable溢出桶代码实现：

https://www.bilibili.com/video/BV1bM411u7Ki?p=69&vd_source=e1686c82128572ae175af1318c920cde

6、HashTable的查找及性能分析



虽然散列表在关键字与记录的存储位置之间建立了直接映像，但由于“冲突”的产生，使得散列表的查找过程仍然是一个**给定关键之和关键字进行比较的过程**。因此，仍需要以**平均查找长度**作为衡量散列表的查找效率的度量。



搜索成功的平均搜索长度：指搜索到表中已有元素的平均探查次数，它是找到表中各个已有元素的探查次数的平均值

搜索不成功的平均搜索长度：指在表中搜索不到待查的元素，但找到插入位置的平均的探查次数，它是表中所有可能散列到的位置上要插入新元素时为找到空桶的探查次数的平均值



散列表的查找效率取决于三个因素：**散列函数、处理冲突的方法和装载因子**

其中装载因子的计算为：

$$\alpha = \frac{\text{表中记录数 } n}{\text{散列表长度 } m}$$

✚ 散列表的平均查找长度依赖于散列表的装载因子,而不直接依赖于n或m,直观地看,装载因子越大,表示装填的记录越“满”,发生冲突的可能性越大,反之发生冲突的可能性越小。

7、哈希表ASL举例

• 线性探测

关键码为 37, 25, 14, 36, 49, 68, 57, 11

散列表为HT[12],表的大小 $m=12$

散列函数是: $\text{Hash}(x) = x \% 11$

$\text{Hash}(37)=4$ $\text{Hash}(25)=3$ $\text{Hash}(14)=3$ $\text{Hash}(36)=3$

$\text{Hash}(49)=5$ $\text{Hash}(68)=2$ $\text{Hash}(57)=2$ $\text{Hash}(11)=0$

0	1	2	3	4	5	6	7	8	9	10	11
11		68	25	37	14	36	49	57			

$$\text{ASL}_{\text{succ}} = \frac{1}{8} \sum_{i=1}^8 C_i = \frac{1}{8} (1 + 1 + 3 + 4 + 3 + 1 + 7 + 1) = \frac{21}{8}$$

$$\text{ASL}_{\text{unsucc}} = \frac{2 + 1 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 1}{11} = \frac{40}{11}$$

• 二次探测

关键码为 37, 25, 14, 36, 49, 68, 57, 11

散列表为HT[19],表的大小 $m=19$

散列函数是: $\text{Hash}(x) = x \% 19$

$$H_i = (H_0 + i^2) \% m, \quad H_i = (H_0 - i^2) \% m, \quad i = 1, 2, 3, \dots, (m-1)/2$$

$\text{Hash}(37)=18$ $\text{Hash}(25)=6$ $\text{Hash}(14)=14$ $\text{Hash}(36)=17$

$\text{Hash}(49)=11$ $\text{Hash}(68)=11$ $\text{Hash}(57)=0$ $\text{Hash}(11)=11$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
57						25				11	49	68		14			36	37

$$\text{ASL}_{\text{succ}} = \frac{1}{8} \sum_{i=1}^8 C_i = \frac{1}{8} (1 + 1 + 1 + 1 + 1 + 2 + 1 + 3) = \frac{11}{8}$$

$$\text{ASL}_{\text{unsucc}} = \frac{1}{19} (2 + 1 + 1 + 1 + 1 + 1 + 2 + 1 + 1 + 1 + 3 + 4 + 2 + 1 + 2 + 1 + 1 + 3 + 4) = \frac{33}{19}$$

• 拉链法

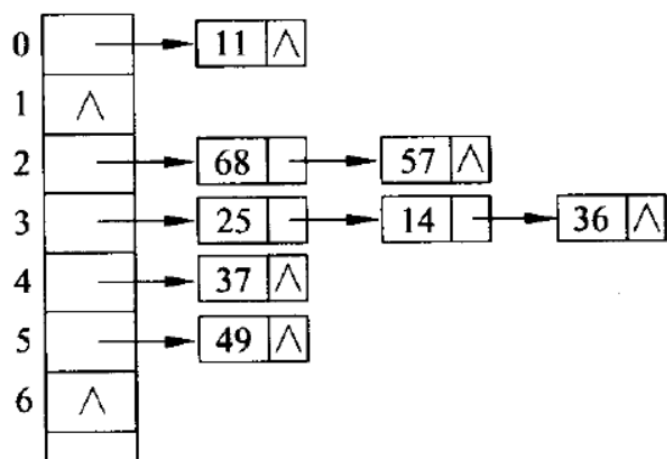
关键码为 37, 25, 14, 36, 49, 68, 57, 11

散列表为 HT[12], 表的大小 $m=12$

散列函数是: $\text{Hash}(x) = x \% 11$

$\text{Hash}(37)=4$ $\text{Hash}(25)=3$ $\text{Hash}(14)=3$ $\text{Hash}(36)=3$

$\text{Hash}(49)=5$ $\text{Hash}(68)=2$ $\text{Hash}(57)=2$ $\text{Hash}(11)=0$



$$ASL_{\text{succ}} = \frac{1}{8}(1 + 1 + 2 + 1 + 2 + 3 + 1 + 1) = \frac{12}{8}$$

$$ASL_{\text{unsucc}} = \frac{1}{11}(2 + 1 + 3 + 4 + 2 + 2 + 1 + 1 + 1 + 1 + 1) = \frac{19}{11}$$