

算法设计与分析

主讲人：权丽君

Email: *ljquan@suda.edu.cn*

苏州大学 计算机学院



第7讲 数据结构

内容提要：

- 基本数据结构
- 散列表
- 二叉搜索树
- 红黑树



数据结构

□ 基本的数据结构

(1)集合结构:

- a. 该结构的数据元素间的关系是“属于同一个集合”。
- b. 特征（数学上）：确定性；互异性；无序性。

(2)线性结构:

- a. 该结构的数据元素之间存在着一对一的关系。
- b. 常用的线性结构有：线性表，栈，队列，双队列，数组，串。

(3)树型结构:

- a. 该结构的数据元素之间存在着一对多的关系。
- b. 树形结构是一层次的嵌套结构。一个树形结构的外层和内层有相似的结构，所以这种结构多可以递归的表示。

(4)图形结构:

该结构的数据元素之间存在着多对多的关系，也称网状结构。



动态集合

- 动态集合：算法操作的集合在能增大、缩小或发生其他变化。
- 动态集合的元素：由一个对象表示。
 - ✓ 关键字：对象属性为标识；
 - ✓ 卫星数据：这些数据与其他对象属性一起移动；
 - ✓ 指针：指向对象的指针，可以对属性进行检查和操作。
 - ✓ Note: 动态集合以其关键字来自于某个全序集为前提。
- 动态集合的操作
 - ✓ 查询操作：简单返回有关集合信息
 - ✓ 修改操作：改变集合的操作。



标准操作

SEARCH(S, k): 一个查询操作，给定一个集合 S 和关键字 k ，返回指向 S 中某个元素的指针 x ，使得 $x.key = k$ ；如果 S 中没有这样的元素，则返回 NIL。

INSERT(S, x): 一个修改操作，将由 x 指向的元素加入到集合 S 中。通常假定元素 x 中集合 S 所需要的每个属性都已经被初始化好了。

DELETE(S, x): 一个修改操作，给定指针 x 指向集合 S 中的一个元素，从 S 中删除 x 。(注意，这个操作取一个指向元素 x 的指针作为输入，而不是一个关键字的值。)

MINIMUM(S): 一个查询操作，在全序集 S 上返回一个指向 S 中具有最小关键字元素的指针。

MAXIMUM(S): 一个查询操作，在全序集 S 上返回一个指向 S 中具有最大关键字元素的指针。

SUCCESSOR(S, x): 一个查询操作，给定关键字属于全序集 S 的一个元素 x ，返回 S 中比 x 大的下一个元素的指针；如果 x 为最大元素，则返回 NIL。

PREDECESSOR(S, x): 一个查询操作，给定关键字属于全序集 S 的一个元素 x ，返回 S 中比 x 小的前一个元素的指针；如果 x 为最小元素，则返回 NIL。

推广案例：

按序枚举集合（ n 个元素）中的所有数据：调用一次 **MAXIMUM**，然后再调用 $(n-1)$ **PREDECESSOR**

实现动态集合的几种数据结构：栈、队列、链表、根数（基础的）
散列表、二叉搜索树、红黑树

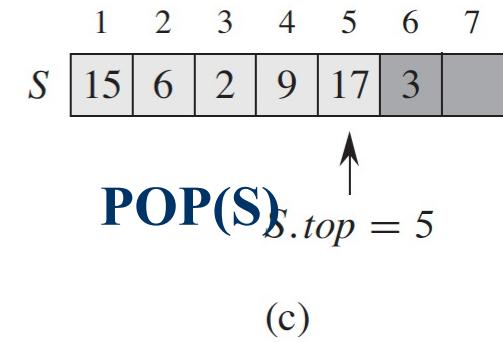
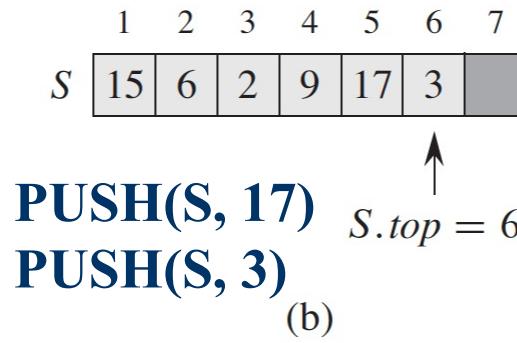
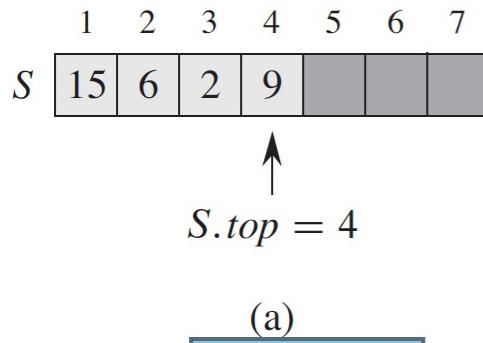


第10章 基本数据结构

□ 栈

- ✓ 动态集合。
- ✓ 被删除的是最近插入的元素：实现的是一种后进先出（LIFO）策略。
- ✓ 栈的基本操作：
 - INSERT操作称为压入（PUSH），
 - 无参数的DELETE操作称为弹出（POP）。

当 $S.top=0$ ，栈是空的。





栈的几种操作只需分别用几行代码来实现

STACK-EMPTY(S)

```
1 if  $S.top == 0$ 
2     return TRUE
3 else return FALSE
```

PUSH(S, x)

```
1  $S.top = S.top + 1$ 
2  $S[S.top] = x$ 
```

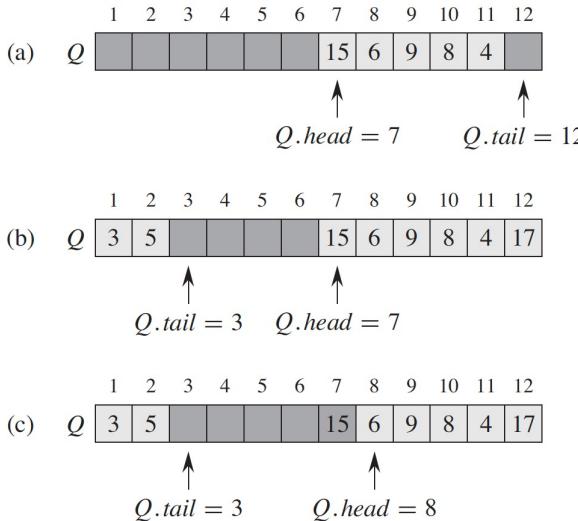
POP(S)

```
1 if STACK-EMPTY( $S$ )
2     error "underflow"
3 else  $S.top = S.top - 1$ 
4     return  $S[S.top + 1]$ 
```



□ 队列

- ✓ 动态集合。
- ✓ 被删除的是最久插入的元素：队列实现的是一种先进先出（FIFO）策略。
- ✓ 栈的基本操作：
 - INSERT操作称为入队（ENQUEUE），
 - 无参数的DELETE操作称为出队（DEQUEUE）。



(a) $Q=[Q.head, Q.head+1, \dots, Q.tail-1]$

(b) ENQUEUE(Q, 17)

ENQUEUE(Q, 3)

ENQUEUE(Q, 5)

Note: 可循环（环序）

(c) DEQUEUE(Q)

队列为空: $Q.head=Q.tail$

初始化时: $Q.head=Q.tail=1$

队列为满: $Q.head=Q.tail+1$



队列的操作（忽略下溢，上溢）

ENQUEUE(Q, x)

```
1  $Q[Q.tail] = x$ 
2 if  $Q.tail == Q.length$ 
3      $Q.tail = 1$ 
4 else  $Q.tail = Q.tail + 1$ 
```

DEQUEUE(Q)

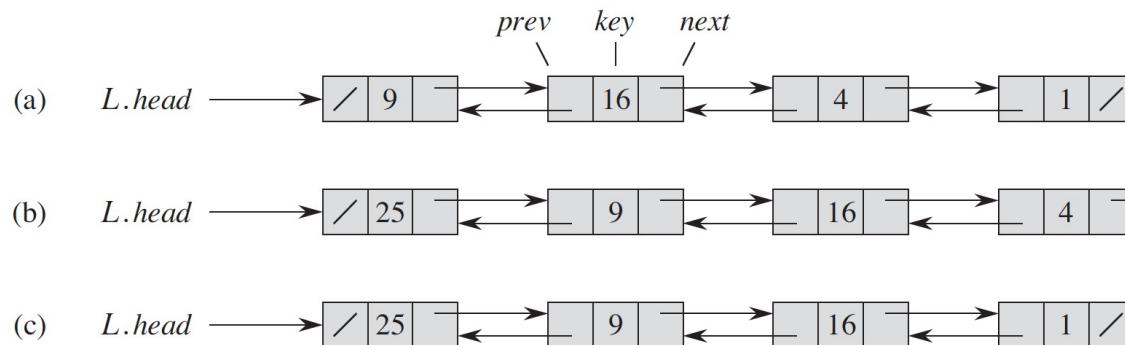
```
1  $x = Q[Q.head]$ 
2 if  $Q.head == Q.length$ 
3      $Q.head = 1$ 
4 else  $Q.head = Q.head + 1$ 
5 return  $x$ 
```

下溢：从空队列删除一个元素
上溢：向满队列插入一个元素



链表

- 链表：是一种这样的数据结构，其中的各对象按线性顺序排列。
- 与数组不同的是，链表的顺序是由各个对象里的指针决定的。
- 形式多样：单链接；双链接；排序的（关键字）；未排序的；循环链表
- 链表主要包括搜索、插入和删除操作。



- (a) 设 x 为链表的一个元素
 - 1, $x.\text{prev}$ 指向它的前驱元素
 - 2, $x.\text{next}$ 指向它的后继元素
 - 3, $x.\text{key}$ 是关键字
 - 4, 头部: $x.\text{prev}=\text{NIL}$
 - 5, 尾部: $x.\text{next}=\text{NIL}$

(b) **LIST-INSERT(L,X):**

$x.\text{key}=25$

(c) **LIST-DELETE(L,X):**

$x.\text{key}=4$
Soochow University



链表的操作

LIST-SEARCH(L, k)

```
1  $x = L.\text{head}$ 
2 while  $x \neq \text{NIL}$  and  $x.\text{key} \neq k$ 
3      $x = x.\text{next}$ 
4 return  $x$ 
```

LIST-INSERT(L, x)

```
1  $x.\text{next} = L.\text{head}$ 
2 if  $L.\text{head} \neq \text{NIL}$ 
3      $L.\text{head}.\text{prev} = x$ 
4  $L.\text{head} = x$ 
5  $x.\text{prev} = \text{NIL}$ 
```

LIST-SEARCH

线性搜索方法

查找链表L中第一个关键字为k的元素
最坏运行时间为 $\theta(n)$

LIST-DELETE(L, x)

```
1 if  $x.\text{prev} \neq \text{NIL}$ 
2      $x.\text{prev}.\text{next} = x.\text{next}$ 
3 else  $L.\text{head} = x.\text{next}$ 
4 if  $x.\text{next} \neq \text{NIL}$ 
5      $x.\text{next}.\text{prev} = x.\text{prev}$ 
```

LIST-DELETE

通过修改一些指针，将x”删除出”该链接
的运行时间为 $O(1)$ ；但如果要删除具有
给定关键字的元素最坏运行时间为 $\theta(n)$

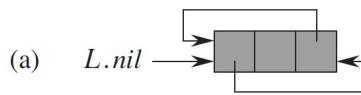
LIST-INSERT

给定设置好关键字key的元素x
连接到链表的头部
最坏运行时间为 $O(1)$

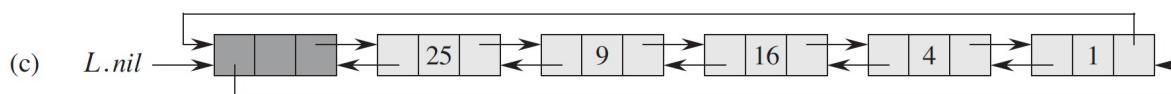
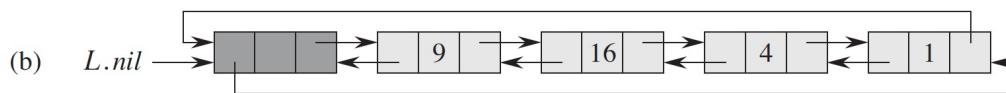


哨兵是一个哑对象，其作用是简化边界条件的处理

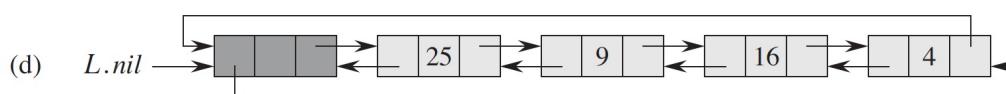
- a) L.nil位于表头和表位之间；
L.nil.next指向表头；
L.nil.prev指向表尾；



- b) 表尾的next属性和表头的prev指向L.nil。



c) LIST-INSERT' (L, x) ,
 $x.key=25$



d) LIST-DELETE' (L, x) ,
 $x.key=1$



有哨兵的链表操作

LIST-SEARCH'(L, k)

```
1  $x = L.nil.next$ 
2 while  $x \neq L.nil$  and  $x.key \neq k$ 
3      $x = x.next$ 
4 return  $x$ 
```

哨兵基本不能降低数据结构相关操作的渐近时间界，但可以降低常数因子

◦

LIST-DELETE'(L, x)

```
1  $x.prev.next = x.next$ 
2  $x.next.prev = x.prev$ 
```

LIST-INSERT'(L, x)

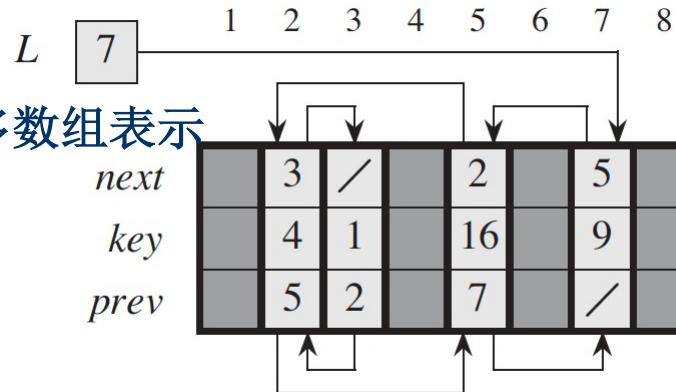
```
1  $x.next = L.nil.next$ 
2  $L.nil.next.prev = x$ 
3  $L.nil.next = x$ 
4  $x.prev = L.nil$ 
```



指针和对象的实现

在没有显示的指针数据类型的情况下实现链式数据结构的两种方法

1



1, 使用多个数组存储下标索引，使用索引代替指针。

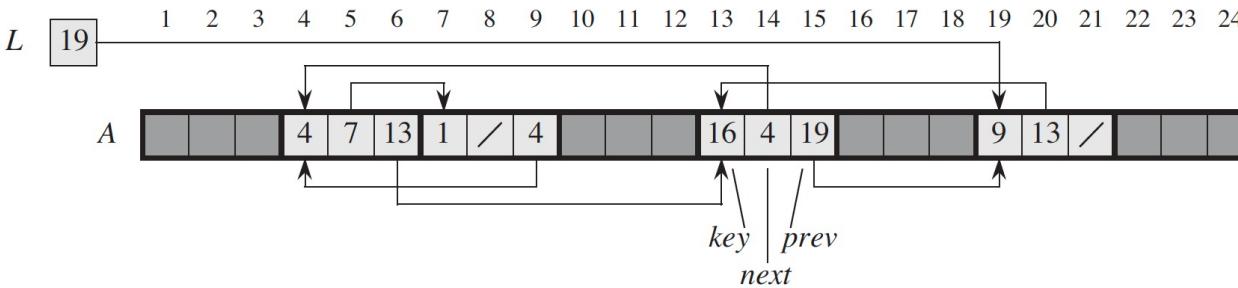
2, 每一列数组项表示一个单一的对象。

3, 变量L存放表头元素的下标。

4, NIL常用不能代表数组中任何实际位置的整数 (-1) 来表示

2

对象的单数组表示



1, 使用一个数组，key、
next、prev的偏移量分别为
0, 1, 2。指向某个对象的
指针就是该对象内第一个
元素下标。

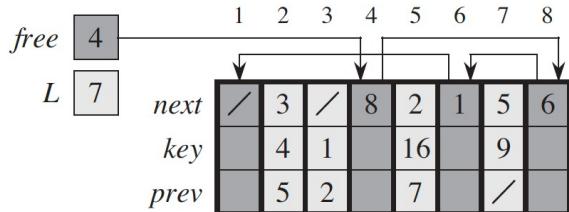


对象的分配与释放

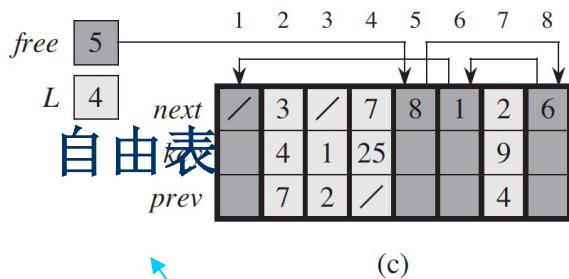
- 对链表表示中尚未利用的对象空间进行管理，使其能够被分配。
- 垃圾收集器就是负责确定哪些对象是未使用的。
 - ✓ 假设多数组表示法中的各数组长度为m，且在某一时候该动态集合含有 $n \leq m$ 个元素，余下的 $m-n$ 个对象是自由的。
 - ✓ 自由对象保存在一个单链表中，称为自由表。
 - 自由表中只使用next数组
 - 自由表的头保存在全局变量free中。



使用数组实现指针，还需要有自由表（free list）的配合（类似于栈），通过，ALLOCATE-OBJECT()过程分配一个对象，通过FREE-OBJECT(x)过程释放对象。

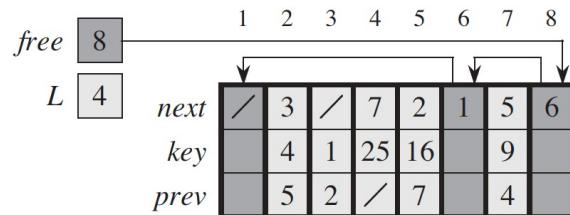


(a)



(c)

执行LIST-DELETE
(L, 5),
然后调用FREE-
OBJECT (5),
对象5称为新自由表
表头



(b)

ALLOCATE-OBJECT()

```
1 if free == NIL  
2     error "out of space"  
3 else x = free  
4     free = x.next  
5 return x
```

调用ALLOCATE-OBJECT () 返回下标4，
将key[4]设为25，再调用LIST-INSERT (L, 4)
) 处理

FREE-OBJECT(x)

1 $x.next = free$

$$2 \text{ } free = x$$

案例：多个链表公用一个自由表



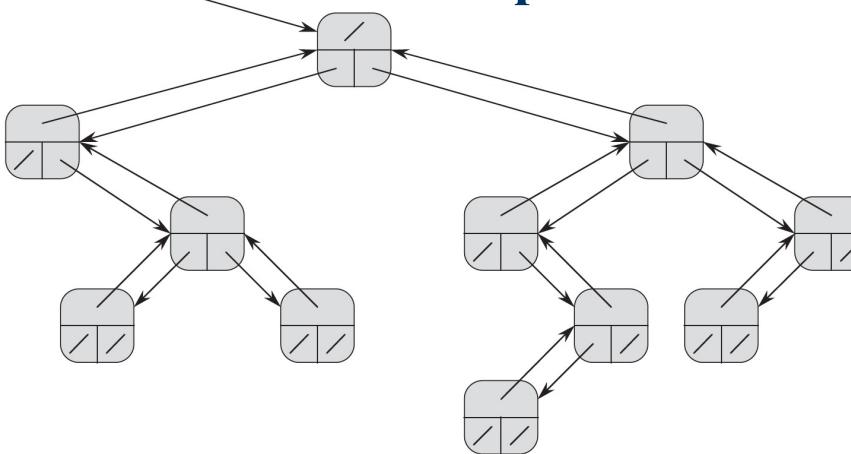


有根树的表示

每个节点都含有一个关键字，指向其它节点的指针p, left and right

1 二叉树

T.root: x.p=Nil

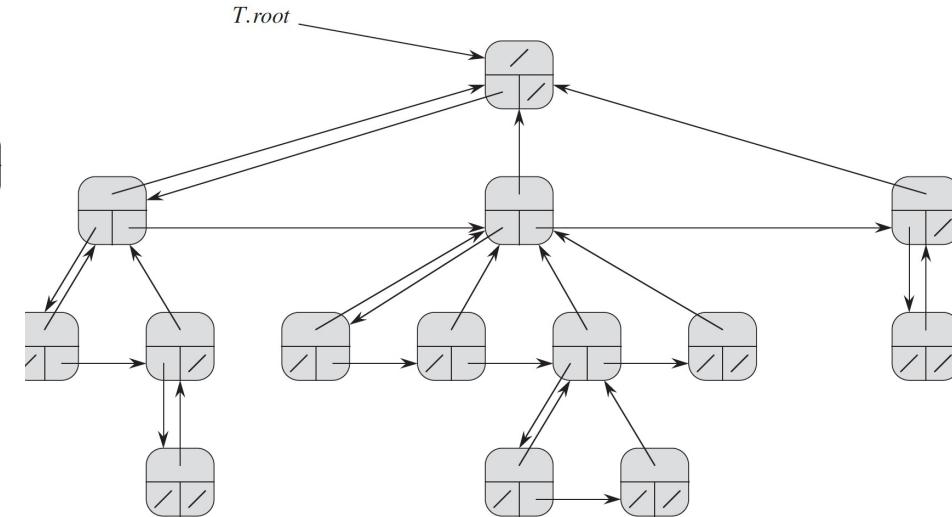


没有左孩子: x.left=Nil

没有右孩子: x.right=Nil

2 分支无限制的有根树

每个节点的孩子数至多有常数k个，则可以表示为 $child_1, child_2, \dots, child_k$ 。当节点无限时，该方法无效：不知道预先分配多少属性。



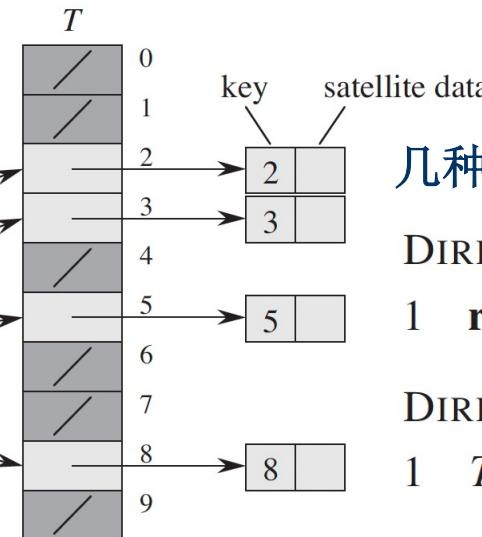
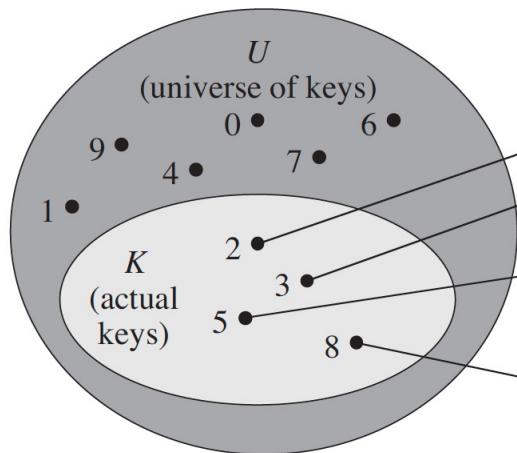
左孩子右兄弟表示法:

x.left-child: 指向结点的最左边孩子结点
x.right-sibling: 指向结点右测相邻的结点



第11章 散列表

- 许多应用都需要动态集合结构，它至少需要支持Insert, search和delete字典操作。散列表（hash table）是实现字典操作的一种有效 的数据结构
- 在介绍散列表之前，我们先介绍直接寻址表。
 - ✓ 当关键字的全域U（关键字的范围）比较小时，直接寻址是一种简单而有 效的技术。我们假设某应用要用到一个动态集合，其中每个元素的关键字 都是取自于全域 $U = \{0, 1, \dots, m-1\}$ ，其中m不是一个很大的数。另外， 假设每个元素的关键字都不同。
 - ✓ 为表示动态集合，我们用一个数组，或称为直接寻址表（direct-address table），记为 $T[0 \sim m-1]$ ，其中每一个位置（槽）对应全域U中的一个关 键字，对应规则是，槽 k 指向集合中关键字为 k 的元素，如果集合中没有关键 字为 k 的元素，则 $T[k] = NIL$ 。



几种字典操作实现起来非常简单：

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] = \text{NIL}$

直接寻址的缺点

1) 如果全域U很大，那么表T 将要申请一段非常长的空间，很可能失败；

2) 对于全域较大，但是元素却十分稀疏的情况，使用这种存储方式将浪费大量的存储空间。

上述的每一个操作的时间均为O(1)时间。

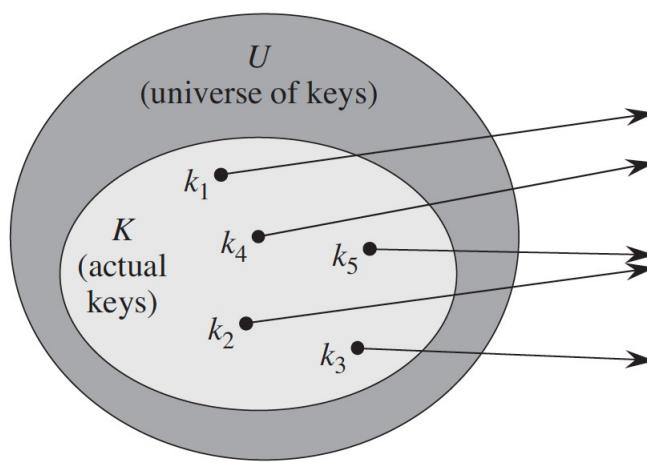


散列表

- 为了克服直接寻址技术的缺点，而又保持其快速字典操作的优势，我们可以利用散列函数（**hash function**）来计算关键字k所在的位置

$$h: U \rightarrow \{0, 1, 2, \dots, m-1\}$$

- 简单的讲，散列函数(k)的作用是将范围较大的关键字映射到一个范围较小的集合中。
- 这时我们可以说，一个具有关键字k的元素被散列到槽h(k)上，或者说h(k)是关键字k的散列值。



这时会产生一个问题：两个关键字可能映射到同一槽中（我们称之为冲突（**collision**）），并且不管你如何优化 $h(k)$ 函数，这种情况都会发生（因为 $|U|>m$ ）。

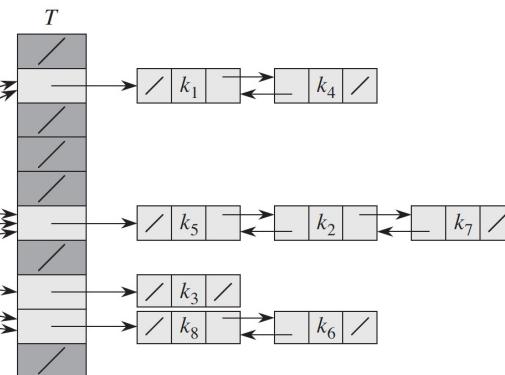
因此我们现在面临两个问题：
一）是遇到冲突时如何解决；
二）是要找出一个的函数 $h(k)$ 能够尽量的减少冲突；

解决冲突：
链表法
开放寻址法



通过链表法解决冲突

解决办法就是，我们把同时散列到同一槽中的元素以链表的形式“串联”起来，而该槽中保存的是指向该链表的指针。如下图所示：



采用该解决办法后，我们可以通过如下的操作方式来进行字典操作：

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

1) 插入操作，很明显时间为O(1)

2) 查找操作的最坏情况运行时间与表的长度成正比

3) 删除操作：

a) 如果链表 $T[h(k)]$ 是双链表，花费的时间为O(1)；

b) 如果链表 $T[h(k)]$ 是单链表，则花费的时间和查找操作的渐进运行时间相同。

在表 $T[h(x.key)]$ 中找到 x ，然后通过更改 x 的前驱元素的next的属性（找前驱需要查找操作）。**Soochow University**



链接法散列的分析

查找运行时间：

首先，我们假定任何一个给定元素都等可能地散列在散列表T的任何一个槽位中，且与其他元素被散列在T的那个位置无关。我们称这个假设为简单均匀散列（simple uniform hashing）。

不失一般性，我们设散列表T的m个槽位散列了n个元素，则平均每个槽位散列了 $\alpha = n/m$ 个元素，我们称 α 为T的装载因子。我们记位于槽位j的链表为 $T[j]$ ($j=1, 2, \dots, m-1$)，而 n_j 表示链表 $T[j]$ 的长度，于是有

$$n = n_0 + n_1 + \dots + n_{m-1},$$

且期望值 $E[n_j] = \alpha = n / m$ 。

现在我们分查找成功和查找不成功两种情况讨论。

① 查找不成功

在查找不成功的情况下，我们需要遍历链表 $T[j]$ 的每一个元素，而链表 $T[j]$ 的期望长度是 α ，因此需要时间 $O(\alpha)$ ，加上索引到 $T(j)$ 的时间 $O(1)$ ，总时间为 $\Theta(1 + \alpha)$ 。

定理 11.1 在简单均匀散列的假设下，对于用链接法解决冲突的散列表，一次不成功查找的平均时间为 $\Theta(1 + \alpha)$ 。



② 查找成功

在查找成功的情况下，我们无法准确知道遍历到链表T[j]的何处停止，因此我们只能讨论平均情况。

1) 我们设 x_i 是散列表T的第i个元素（假设我们按插入顺序对散列表T中的n个元素进行了1~n的编号）， k_i 表示 $x_i.key$ ，其中 $i = 1, 2, \dots, n$ 。

2) 再定义随机变量 $X_{ij} = I\{h(k_i) = h(k_j)\}$ ，即：

$$X_{ij} = \begin{cases} 0, & h(k_i) \neq h(k_j) \\ 1, & h(k_i) = h(k_j) \end{cases}$$

□ 3) 在简单均匀散列的假设下有：

- ✓ $P\{h(k_i) = h(k_j)\} = 1/m$,
- ✓ $E[X_{ij}] = 1/m$

□ 则所需检查的元素的数目的期望是：

因为新的元素都是在表头插入的，所以出现在x之前的元素都是在x之后插入的。

对元素x的一次成功查找中，所检查的元素树就是x所在的链表中x前面的元素多1

$$\begin{aligned} E\left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{ij}\right)\right] &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{ij}]\right) \quad (\text{by linearity of expectation}) \\ &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m}\right) \\ &= 1 + \frac{1}{nm} \sum_{i=1}^n (n-i) \\ &= 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i\right) \\ &= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2}\right) \quad (\text{by equation (A.1)}) \\ &= 1 + \frac{n-1}{2m} \\ &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}. \end{aligned}$$

加上索引到T(j)的时间O(1)

因此，一次成功的检查所需要的时间是 $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$ 。

综合上面的分析，在平均下，全部的字典操作都可以在O(1)时间下完成（槽数至少是元素个数成正比， $n=O(m)$ ，所以 $\alpha=n/m=O(m)/m=O(1)$ ）。



散列函数

- 一个好的散列函数应（近似地）满足简单均匀散列：
 - ✓ 每个关键字都等可能的被散列到各个槽位，并与其他关键字散列到哪一个槽位无关。
 - ✓ 我们一般无法检验这一条件是否成立。
- 下面给出两种基本的构造散列函数的方法：
 - ✓ (1) 除法散列法：

除法散列法的做法很简单，就是让关键字k去除以一个数m，取余数，这样就将k映射到m个槽位中的某一个，即散列函数是：

$$h(k) = k \bmod m ,$$

由于只做一次除法运算，该方法的速度是非常快的。但应当注意的是，我们在选取m的值时，应当避免一些选取一些值。

例如，m不应是2的整数幂，因为如果 $m = 2^p$ ，则 $h(k)$ 就是k的p个最低位数字。除非我们已经知道各种最低p位的排列是等可能的，否则我们最好慎重的选择m（如练习11.3-3）。而一个不太接近2的整数幂的素数，往往是较好的选择。



□ (2) 乘法散列法

□ 该方法包含两个步骤：

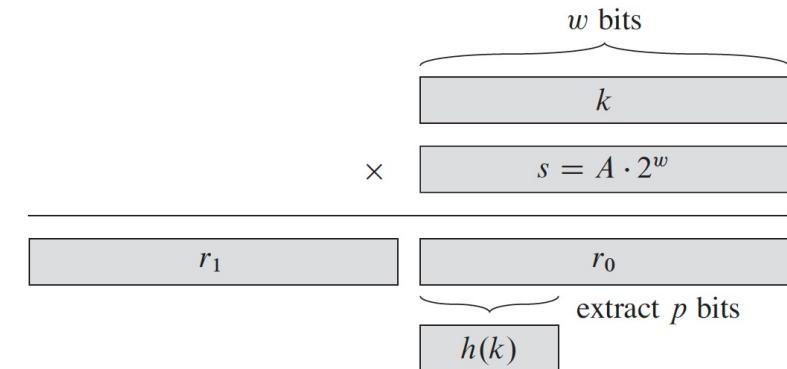
- ✓ 第一步：用关键字 k 乘以 A ($0 < A < 1$)，并提取 kA 的小数部分；
- ✓ 第二步：用 m 乘以这个值，在向下取整，即散列函数是：
 ✓
$$h(k) = [m(kA \bmod 1)],$$

 ✓ 这里 “ $kA \bmod 1$ ” 的是取 kA 小数部分的意思，即 $kA - \lfloor kA \rfloor$ 。

□ 乘法散列法的一个优点是，一般我们对 m 的选择不是特别的关键，一般选择它为 2 的整数幂即可 (2^p)。

假设某计算机的字长为 w 位，而 k 正好可用一个单字表示。限制 A 为形如 $s/2^w$ 的一个分数，其中 s 是一个取自 $0 < s < 2^w$ 的整数。参见图 11-4，先用 w 位整数 $s = A \cdot 2^w$ 乘上 k ，其结果是一个 $2w$ 位的值 $r_1 2^w + r_0$ ，这里 r_1 为乘积的高位字， r_0 为乘积的低位字。所求的 p 位散列值中，包含了 r_0 的 p 个最高有效位。

□ 虽然这个方法对任意的 A 都适用，但 Knuth 认为， $A \approx (\sqrt{5} - 1)/2 = 0.618033988\dots$ 是一个比较理想的值。



散列的乘法方法。关键字 k 的 w 位表示乘上 $s = A \cdot 2^w$ 的 w 位值。在乘积的低 w 位中， p 个最高位构成了所需的散列值 $h(k)$

作为一个例子，假设 $k=123\,456$, $p=14$, $m=2^{14}=16\,384$, 且 $w=32$ 。依据 Knuth 的建议，取 A 为形如 $s/2^{32}$ 的分数，它与 $(\sqrt{5}-1)/2$ 最为接近，于是 $A=2\,654\,435\,769/2^{32}$ 。那么， $k \times s = 327\,706\,022\,297\,664 = (76\,300 \times 2^{32}) + 17\,612\,864$ ，从而有 $r_1=76\,300$ 和 $r_0=17\,612\,864$ 。 r_0 的 14 个最高有效位产生了散列值 $h(k)=67$ 。

例子



通过开放寻址解决冲突

- 在开放寻址法（open addressing）中，所有的元素存放在散列表里。也就是说，每个表项或包含动态集合的一个元素，或包含NIL。
- 当查找某个元素时，要系统地检查所有的表项，直到找到所需的元素，或者最终查明该元素不在表中。
- 该方法导致的一个结果就是转载因子alpha不会超过1。
- 开放寻址法的好处就在于它不用指针，而是计算出要存取的槽序列。
 - ✓ 为了使用开放寻址法插入一个元素，需要连续地检查散列表，或成为探查（probe），直到找到一个空槽来放置待插入的元素为止。检查的顺序依赖于待插入的关键字。
 - ✓ 为了确定要探查哪些槽，我们将散列函数加以扩充，使之包含探查号（从0开始）以作为第二个输入参数。这样，散列函数就变成：

$$h: U^* \{0\ 1\ 2\ ..\ m-1\} \rightarrow \{0\ 1\ 2\ ... m-1\}$$

- ✓ 对每一个关键字k，使用开放寻址法的探查序列

$$\langle h(k,0), h(k,1), ..., h(k,m-1) \rangle$$



插入散列表

HASH-INSERT(T, k)

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == \text{NIL}$ 
5           $T[j] = k$ 
6          return  $j$ 
7      else  $i = i + 1$ 
8  until  $i == m$ 
9  error "hash table overflow"
```

搜索散列表

HASH-SEARCH(T, k)

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == k$ 
5          return  $j$ 
6       $i = i + 1$ 
7  until  $T[j] == \text{NIL}$  or  $i == m$ 
8  return NIL
```

假设散列表 T 中的元素为无卫星数据的关键字；
关键字 k 等同于包含关键字 k 的元素。
每个槽或包含一个关键字，或包含NIL（空的）

有三种主要的探查方法：
1) 线性探查；
2) 二次探查；
3) 双重散列。



线性探查

- 给定一个普通的散列函数
 - ✓ $h': U \rightarrow \{0, 1, \dots, m-1\}$, 称之为辅助散列函数 (auxiliary hash)
- 线性探查方法采用的散列函数为:
$$h(k, i) = (h'(k) + i) \bmod m, i = 0, 1, \dots, m-1$$
- 给定一个关键字 k , 探查过程为:
 - ✓ 首先探查 $T[h'(k)]$,
 - ✓ 再探查槽 $T[h'(k)+1]$, 以此类推, 直至槽 $T[m-1]$.
 - ✓ 然后, 又绕到 $T[0], T[1], \dots$, 直到最后探查到 $T[h'(k)-1]$ 。
- 在线性探查方法中, 初始探查位置决定了整个序列, 故只有 m 种不同的探查序列。
- 它存在一个问题, 称为一次群集 (primary cluster)。
 - ✓ 随着连续被占用的槽不断增加, 平均查找时间也随之不断增加。
 - ✓ 群集现象很容易出现, 这是因为当一个空槽前有 i 个满的槽时, 该空槽为下一个将被占用的概率是 $(i+1)/m$ 。
 - ✓ 连续被占用的槽就会越来越长, 因而平均查找时间也会越来越大。



二次探查

- 二次探查 (quadratic probing) 采用如下形式的散列函数：

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m ,$$

- ✓ 其中 h' 是一个辅助散列函数， c_1 和 c_2 为正的辅助常数。
- 初始的探查位置为 $T[h'(k)]$,后续的探查位置要加上一个偏移量，该偏移量以二次的方式依赖于探查序号 i 。
- 这种探查方法的效果要比线性探查好得多。
- 此外，如果两个关键字的初始探查位置相同，那么它们的探查序列也是相同的，这是因为 $h(k_1, 0) = h(k_2, 0)$ 蕴含着 $h(k_1, i) = h(k_2, i)$ 。这一性质可能导致一种轻度的群集，称为二次群集 (secondary cluster)。
- 像在线性探查中一样，初始探查位置决定了整个序列，这样也仅有 m 个不同的探查序列被用到。



双重散列

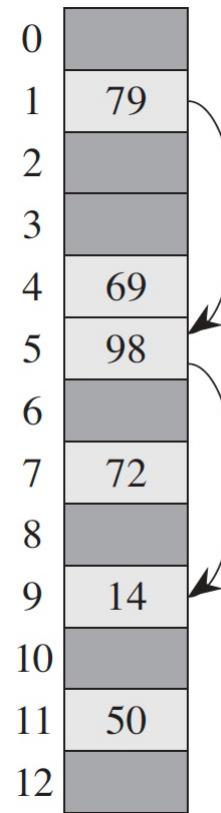
- 双重散列（double hashing）是用于开放寻址法的最好方法之一，因为它所产生的排列具有随机选择排列的许多特性。双重散列采用如下形式的散列函数：

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m ,$$

- ✓ 其中 h_1 和 h_2 均为辅助散列函数。
- ✓ 初始探查位置为 $T[h_1(k)]$ ，后续的探查位置是前一个位置加上偏移量 $h_2(k)$ 模 m 。
- 为了查找整个散列表，要求所有的 $h_2(k)$ 和 m 互质。
 - ✓ 因为如果 $h_2(k)$ 和 m 的最大公约数是 d ，
 - ✓ $h_2(k) \bmod m$ 和 m 的最大公约数也是 d （辗转相除算法的由来），
 - ✓ 则 $t \in [0, m-1]$, $t * h_2(k) \bmod m$ 可以整除 d ，
 - ✓ 因此，该双重散列只能遍历散列表中 $1/d$ 个元素，因此当 $d=1$ 时，即 $h_2(k)$ 和 m 互质，则可以遍历整个散列表。



案例



$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m ,$$

$$h_1(k) = k \bmod m ,$$

$$h_2(k) = 1 + (k \bmod m') ,$$

双重散列法的插入。此处，散列表的大小为 13， $h_1(k) = k \bmod 13$ ， $h_2(k) = 1 + (k \bmod 11)$ 。因为 $14 \equiv 1 \pmod{13}$ ，且 $14 \equiv 3 \pmod{11}$ ，故在探查了槽 1 和槽 5，并发现它们被占用后，关键字 14 被插入到槽 9 中



开放寻址散列的分析

在均匀散列的假设下，用开放寻址法来进行散列时探查的期望次数：

定理 11.6 给定一个装载因子为 $\alpha = n/m < 1$ 的开放寻址散列表，并假设是均匀散列的，则对于一次不成功的查找，其期望的探查次数至多为 $1/(1-\alpha)$ 。

1

一次不成功
查询时的探
查次数

证明 在一次不成功的查找中，除了最后一次探查，每一次探查都要检查一个被占用但并不包含所求关键字的槽，最后检查的槽是空的。先定义随机变量 X 为一次不成功查找的探查次数，再定义事件 $A_i (i=1, 2, \dots)$ 为第 i 次探查且探查到的是一个已经被占用的槽。那么，事件 $\{X \geq i\}$ 即为事件 $A_1 \cap A_2 \cap \dots \cap A_{i-1}$ 的交集。下面通过给出 $\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$ 的界来得到 $\Pr\{X \geq i\}$ 的界。根据练习 C. 2-5，有

$$\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\} = \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \cdots \\ \Pr\{A_{i-1} | A_1 \cap A_2 \cap \dots \cap A_{i-2}\}$$

由于有 n 个元素和 m 个槽，所以 $\Pr\{A_1\} = n/m$ 。对于 $j > 1$ ，在前 $j-1$ 次探查到的都是已占用槽的前提下，第 j 次探查且探查到的仍是已占用槽的概率是 $(n-j+1)/(m-j+1)$ 。这是因为要在 $(m-(j-1))$ 个未探查的槽中，查找余下的 $(n-(j-1))$ 个元素中的某一个。由均匀散列的假设知，这一概率为这两个量的比值。注意到 $n < m$ ，对于所有 $j (0 \leq j < m)$ ，就有 $(n-j)/(m-j) \leq n/m$ 。于是，对所有 $i (1 \leq i \leq m)$ ，有

$$\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$



当随机变量 X 可在自然数集 $\mathbf{N}=\{0, 1, 2, \dots\}$ 中取值时，有一个很好的期望计算公式：

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} i \cdot \Pr\{X = i\} = \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \end{aligned} \quad (\text{C. 25})$$

现在，再利用公式(C. 25)来得出探查期望数的界：

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha} \quad \blacksquare$$

$1/(1-\alpha)=1+\alpha+\alpha^2+\alpha^3+\cdots$ 的这个界有一个直观的解释。无论如何，总要进行第一次探查。第一次探查发现的是一个已占用的槽时，必须要进行第二次探查，进行第二次探查的概率大约为 α 。前两次探查所发现的槽均是已占用时，需要进行第三次探查，进行第三次探查的概率大约为 α^2 ，等等。

如果 α 是一个常数，由定理 11.6 可知，一次不成功查找的运行时间为 $O(1)$ 。例如，如果散列表一半是满的，一次不成功查找的平均探查数至多是 $1/(1-0.5)=2$ 。如果散列表是 90% 满的，则平均探查数至多为 $1/(1-0.9)=10$ 。

根据定理 11.6，几乎直接可以得到 HASH-INSERT 过程的性能。



2

推论 11.7 假设采用的是均匀散列，平均情况下，向一个装载因子为 α 的开放寻址散列表中插入一个元素至多需要做 $1/(1-\alpha)$ 次探查。

证明 只有当表中有空槽时，才可以插入新元素，故 $\alpha < 1$ 。插入一个关键字要先做一次不成功的查找，然后将该关键字置入第一个遇到的空槽中。所以，期望的探查次数至多为 $1/(1-\alpha)$ 。■

对于一次成功的查找，需要稍做一些工作来得到探查的期望次数。

定理 11.8 对于一个装载因子为 $\alpha < 1$ 的开放寻址散列表，一次成功查找中的探查期望数至多为

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

假设采用均匀散列，且表中的每个关键字被查找的可能性是相同的。

证明 查找关键字 k 的探查序列与插入关键字为 k 的元素的探查序列是相同的。根据推论 11.7，如果 k 是第 $(i+1)$ 个被插入表中的关键字，则对 k 的一次查找中，探查的期望次数至多为 $1/(1-i/m) = m/(m-i)$ 。对散列表中所有 n 个关键字求平均，则得到一次成功查找的探查期望次数为：

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha} \sum_{k=m-n+1}^m \frac{1}{k} \leq \frac{1}{\alpha} \int_{m-n}^m (1/x) dx \quad (\text{由不等式(A.12)}) \\ &= \frac{1}{\alpha} \ln \frac{m}{m-n} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

如果散列表是半满的，则一次成功的查找中，探查的期望数小于 1.387。如果散列表为 90% 满的，则探查的期望数小于 2.559。

插入一个元素需要探查的次数

3

一次成功查找的探查次数



第12章 二叉搜索树

1) 什么是二叉搜索树

a) 二叉搜索树是以一棵二叉树来组织的：

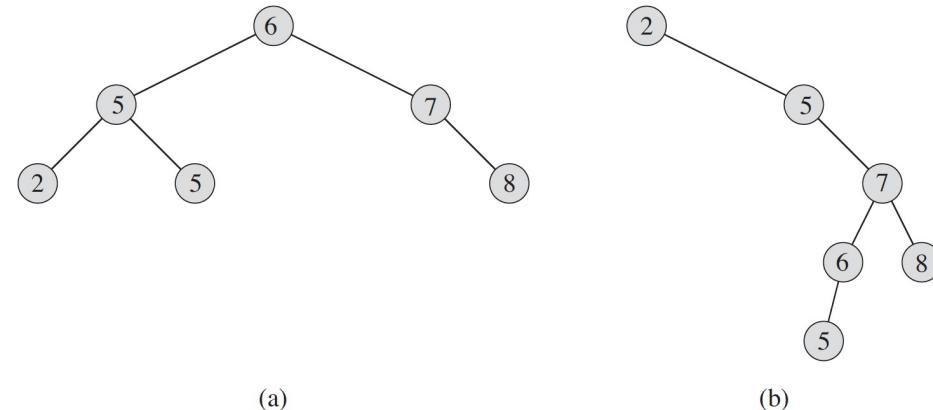
这样的一棵树可以使用一个链表数据结构来表示，其中的每一个节点是一个对象。除了key和卫星数据之外，每个节点还包含属性left（左孩子）、right（右孩子）、和p（双亲）（若不存在，则值为NIL）。

b) 二叉搜索树的性质：

设x为二叉树中的任一结点，那么对于x左子树中的任一结点y都有 $\text{key}[y] \leq \text{key}[x]$ ，x右子树中的任一结点y都有 $\text{key}[x] \leq \text{key}[y]$ 。

设二叉树高度为h

树高不同

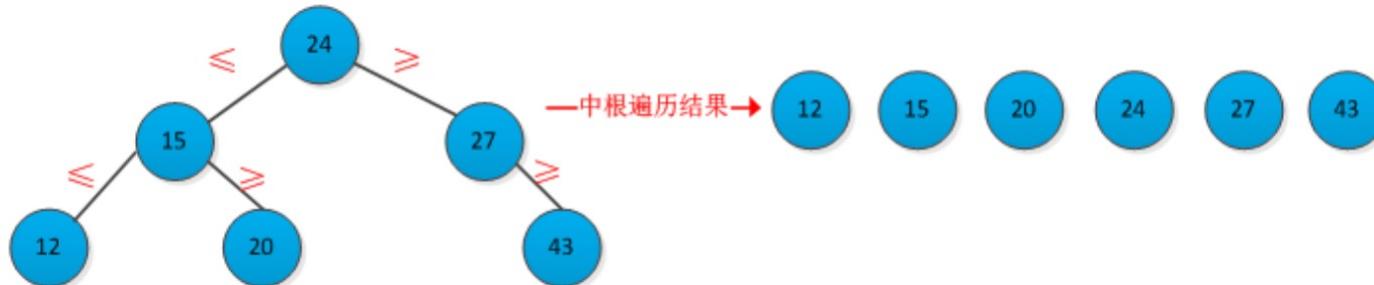




□ 二叉搜索树性质允许我们通过一个简单的递归算法来按序输出二叉搜索树中的所有关键字

- ✓ 前序遍历：先遍历根再遍历左右子树，简称根-左-右。
- ✓ 中序遍历：先遍历左子树再遍历根再遍历右子树，简称左-根-右。
- ✓ 后序遍历：先遍历左右子树再遍历根，简称左-右-根。

□ 采用中序遍历一棵二叉查找树，可以得到树中关键字由小到大的序列。





反应了执行该函数的一个时间上界，
不包含递归调用所花时间

测试是否为空树

中序遍历代码：

INORDER-TREE-WALK(x)

```
1 if  $x \neq \text{NIL}$ 
2   INORDER-TREE-WALK( $x.\text{left}$ )
3   print  $x.\text{key}$ 
4   INORDER-TREE-WALK( $x.\text{right}$ )
```

定理12.1：如果 x 是一棵有 n 个结点子树的根，那么调用**Inorder-tree-walk(x)**，需要 $\theta(n)$ 时间。

证明：

1) 要访问这个子树的全部结点，所以 $T(n) = \Omega(n)$

2) 设 x 结点左子树有 k 个结点且右子树上有 $n-k-1$ 个结点，则：

$T(n) \leq T(k) + T(n-k-1) + d$, $T(0) = c$
其中， c, d 为常数。

可以使用替换法证明 $T(n) \leq (c+d)n + c$ ，得到 $T(n) = O(n)$ 。

$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c, \end{aligned}$$



查询二叉搜索树

- 这一小节涉及二叉搜索树的操作诸如：
 - ✓ SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR。
- 查找
 - ✓ 输入：一个指向根节点的指针x和待查找的关键字k；
 - ✓ 输出：为指向关键字为k的节点的指针（若存在。否则输出NIL）。

查询13:

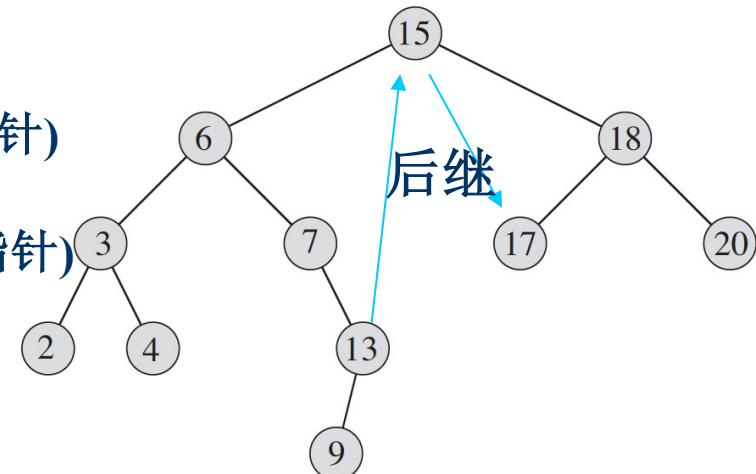
15->6->7->13

查询最小2:

15->6->3->2(沿着left指针)

查询最大20:

15->18->20(沿着right指针)



TREE-SEARCH(x, k)

```
1 if  $x == \text{NIL}$  or  $k == x.\text{key}$ 
2   return  $x$ 
3 if  $k < x.\text{key}$ 
4   return TREE-SEARCH( $x.\text{left}, k$ )
5 else return TREE-SEARCH( $x.\text{right}, k$ )
```

TREE-SEARCH的运行时间为O(h)



查询：迭代代替递归

ITERATIVE-TREE-SEARCH(x, k)

```
1 while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2   if  $k < x.\text{key}$ 
3      $x = x.\text{left}$ 
4   else  $x = x.\text{right}$ 
5 return  $x$ 
```

最小关键字元素

TREE-MINIMUM(x)

```
1 while  $x.\text{left} \neq \text{NIL}$ 
2    $x = x.\text{left}$ 
3 return  $x$ 
```

最大关键字元素

TREE-MAXIMUM(x)

```
1 while  $x.\text{right} \neq \text{NIL}$ 
2    $x = x.\text{right}$ 
3 return  $x$ 
```

运行时间为 $O(h)$



后继和前驱

1) 后继: 如果所有的关键字互不相同, 则一个节点 x 的后继是大于 $x.key$ 的最小关键字的节点。

2) 前驱: 如果所有的关键字互不相同, 则一个节点 x 的前驱是小于 $x.key$ 的最大关键字的节点。

如果联系二叉搜索树的性质:

节点的key值总是大于它的左节点(如果存在)的key值并且小于它的右节点(如果存在)的key值。

那么我们容易推知:

1) 后继

- a) 如果一个节点有右子树, 它后继即是它的右子树的最“左”端的节点, 也即右子树的最左节点,
- b) 否则它的后继位于它的某个父或祖父点。简单的从 x 开始沿树而上, 直到遇到这样一个结点: 这个结点是它双亲的左孩子, 他的父亲就是后继节点

2) 前驱

- a) 而一个节点如果有左子树, 它的前驱即是它的左子树的最“右”端的节点, 也即左子树的最大节点,
- b) 否则它的前驱位于它的某个父或者祖父节点。简单的从 x 开始沿树而上, 直到遇到这样一个结点: 这个结点是它双亲的右孩子, 他的父亲节点是前驱节点

TREE-SUCCESSOR(x)

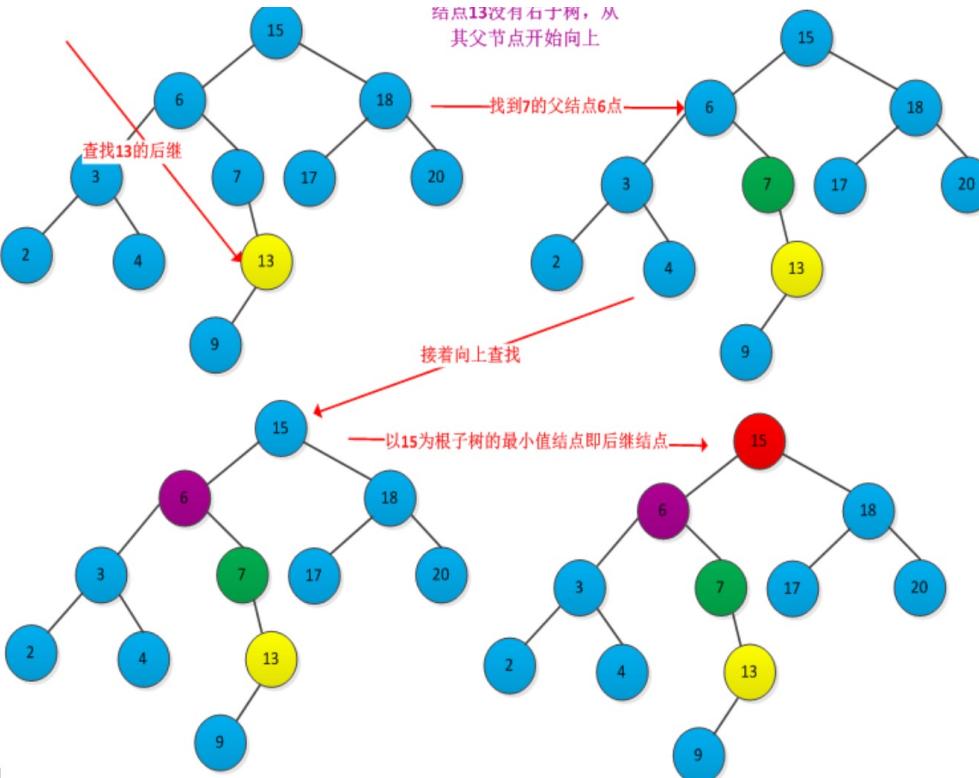
```
1  if  $x.right \neq NIL$ 
2      return TREE-MINIMUM( $x.right$ )
3   $y = x.p$ 
4  while  $y \neq NIL$  and  $x == y.right$ 
5       $x = y$ 
6       $y = y.p$ 
7  return  $y$ 
```

该过程或者遵从一条简单路径沿树向上,
或者遵从一条简单路径沿树向下

TREE-SUCCESSOR和TREE-PREDECESSOR的运行时间为O(h)



案例：



总的来说：在一棵高度为 h 的二叉搜索树上，集合操作SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR都可以在 $O(h)$ 时间内完成。



插入和删除

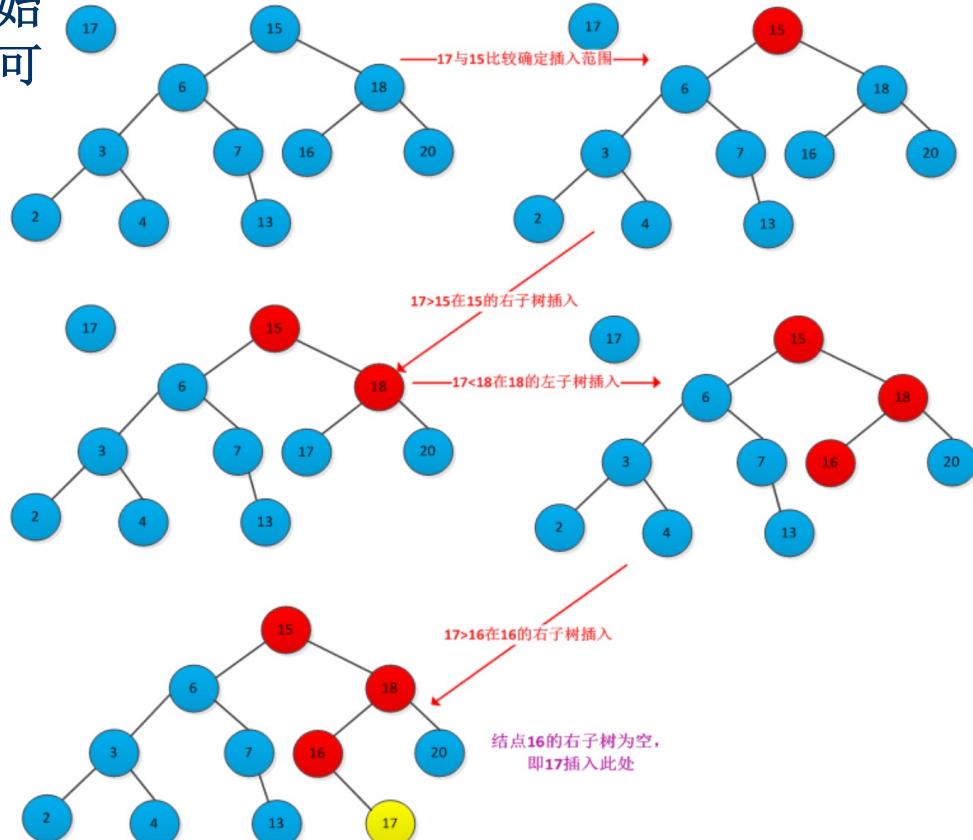
1

插入

插入结点的位置对应着查找过程中查找不成功时候的结点位置，因此需要从根结点开始查找带插入结点位置，找到位置后插入即可。下图所示插入结点过程

TREE-INSERT(T, z)

```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$            // tree  $T$  was empty
11  elseif  $z.\text{key} < y.\text{key}$ 
12       $y.\text{left} = z$ 
13  else  $y.\text{right} = z$ 
```



TREE-INSERT的运行时间为O(h)
Soochow University



2 删除

从二叉查找树中删除给定的结点z，分三种情况讨论：

① 被删除节点没有孩子。只需要修改其父节点，用NIL去替换自己。

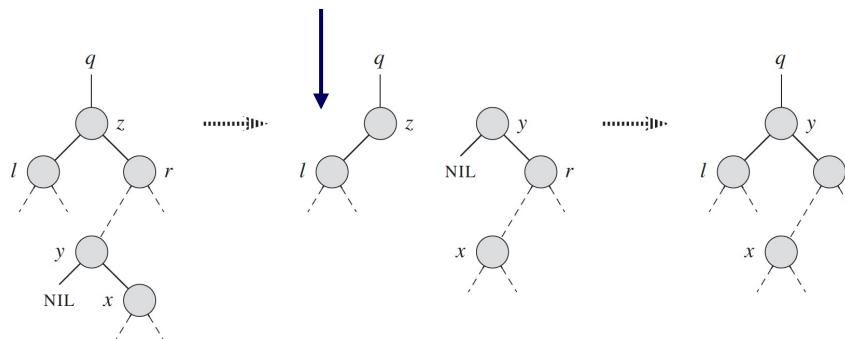
② 被删除节点有一个孩子。也需要修改其父节点，用这个孩子去替换自己。

③ 被删除节点有两个孩子。那么先找z的后继y（一定在z的右子树中），并让y占据树中z的位置。

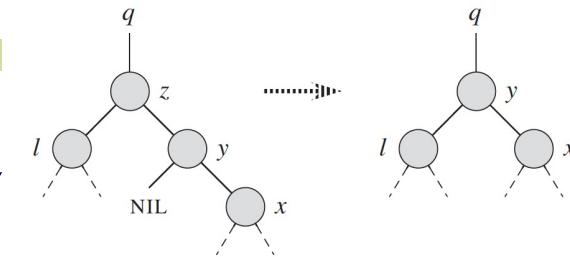
z的原来的右子树部分成为y的新的右子树，并且z的左子树成为y的左子树。

a) 如果z的后继y就是z的右孩子（即y没有左孩子），直接用y代替z，并保留y的右子树，如右上图所示：

b) 如果z的后继y不是z的右孩子，先用y的右孩子替换y，再用y替换z。如下图所示：



该过程能够实现让子树v代替子树u



TREE-DELETE(T, z)

```

if  $z.left == \text{NIL}$ 
    TRANSPLANT( $T, z, z.right$ )
elseif  $z.right == \text{NIL}$ 
    TRANSPLANT( $T, z, z.left$ )
else
     $y = \text{TREE-MINIMUM}(z.right)$ 
    if  $y.p \neq z$ 
        TRANSPLANT( $T, y, y.right$ )
         $y.right = z.right$ 
         $y.right.p = y$ 
    TRANSPLANT( $T, z, y$ )
     $y.left = z.left$ 
     $y.left.p = y$ 

```

12

分析该算法，我们发除了TREE-MINIMUM外，其他操作均花费常量时间。因此删除操作将花费 $O(h)$ 时间

◦



随机构建二叉搜索树

我们先给出随机构建二叉搜索树的定义：

1) 向一棵空树随机的插入n个关键字而得到的树。

(这里随机的意思是n个关键字的n!种排列都可能的出现。)

2) 一棵有n个不同关键字的随机构建二叉搜索树的期望高度为O(lgn)

↓ 证明

先定义三个随机变量Xn, Yn, Rn

1) Xn表示一棵有n个不同关键字的随机构建二叉搜索树的高度；

2) Yn=2^Xn表示二叉搜索树的指数高度 (exponential height)；

3) Rn表示当在n个不同的关键字中选择一个作为树根时，该关键字在这n个关键字集合中的秩 (rank) (即Rn表示这些关键字排好序后这个关键字应占据的位置)：

这样如果Rn = i, 那么表示根的左子树有i-1个元素，右子树有n-i个元素，此时有：

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i})$$

我们再定义一个指示器随机变量Zn,i, Zn,i = I{Rn = i}。因为Rn对于集合 {1, 2, ..., n} 中的任一元素都是等可能的，因此有：

$$p\{R_n = i\} = 1/n, (i=1, 2, \dots, n)$$

$$E(Z_{n,i}) = 1/n$$

由于Zn,i只等于1或0, Yn = 2 · max(Yi-1, Yn-i)

$$Y_n = \sum_{i=1}^n Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i})) .$$

$$\begin{aligned} E[Y_n] &= E\left[\sum_{i=1}^n Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\ &= \sum_{i=1}^n E[Z_{n,i} (2 \cdot \max(Y_{i-1}, Y_{n-i}))] \quad (\text{by linearity of expectation}) \\ &= \sum_{i=1}^n E[Z_{n,i}] E[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad (\text{by independence}) \\ &= \sum_{i=1}^n \frac{1}{n} \cdot E[2 \cdot \max(Y_{i-1}, Y_{n-i})] \quad (\text{by equation (12.1)}) \\ &= \frac{2}{n} \sum_{i=1}^n E[\max(Y_{i-1}, Y_{n-i})] \quad (\text{by equation (C.22)}) \\ &\leq \frac{2}{n} \sum_{i=1}^n (E[Y_{i-1}] + E[Y_{n-i}]) \quad (\text{by Exercise C.3-4}) . \end{aligned}$$



在上式最后的和式中， $Y[0], Y[1], \dots, Y[n-1]$ 都会出现两次（ $E[Y_{i-1}]$ 和 $E[Y_{n-i}]$ 都会出现），因此：

$$E[Y_n] \leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i].$$

上式是一个递归式，我们可以猜测：

$$E[Y_n] \leq \frac{1}{4} \binom{n+3}{3}.$$

上式可以用数学归纳法可证明：

对于边界：

$f(x) = 2^x$ 为凸函数，由Jensen不等式，我们可得进一步可得

$$0 = Y_0 = E[Y_0] \leq (1/4) \binom{3}{3} = 1/4;$$

$$1 = Y_1 = E[Y_1] \leq (1/4) \binom{1+3}{3}$$

如下可得：

$$2^{E[X_n]} \leq \frac{1}{4} \binom{n+3}{3} = \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} = \frac{n^3 + 6n^2 + 11n + 6}{24}$$

两边取对数，得到 $E[X_n] = O(\lg n)$ 。

$$\begin{aligned} E[Y_n] &\leq \frac{4}{n} \sum_{i=0}^{n-1} E[Y_i] \\ &\leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4} \binom{i+3}{3} \quad (\text{by the inductive hypothesis}) \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \binom{i+3}{3} \\ &= \frac{1}{n} \binom{n+3}{4} \quad (\text{by equation (12.3)}) \\ &= \frac{1}{n} \cdot \frac{(n+3)!}{4! (n-1)!} \\ &= \frac{1}{4} \cdot \frac{(n+3)!}{3! n!} \\ &= \frac{1}{4} \binom{n+3}{3}. \end{aligned}$$

$$2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n]$$



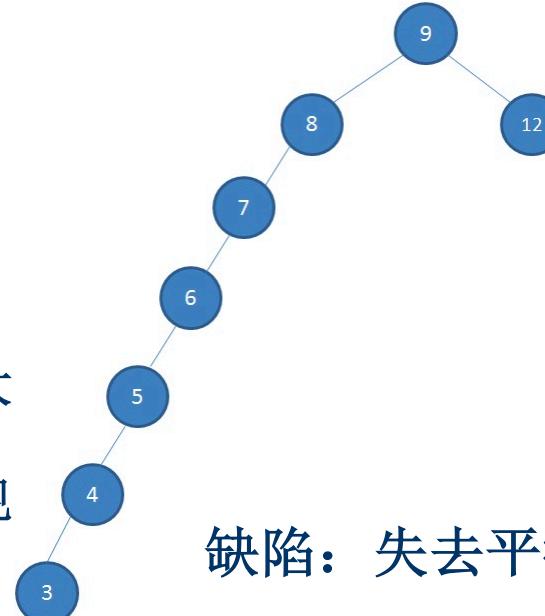
第13章 红黑树

□ 回顾二叉搜索树

- ✓ 性质：设x为二叉树中的任一结点，那么对于x左子树中的任一结点y都有 $\text{key}[y] \leq \text{key}[x]$,x右子树中的任一结点y都有 $\text{key}[x] \leq \text{key}[y]$ 。
- ✓ 算法时间：设二叉树高度为h，
 - 静态操作:查找结点、找前驱、后继等时间为O(h)
 - 动态操作：插入和删除结点等时间为O(h)
- ✓ 对于插入操作，新结点总是插入在某个叶结点位置
- ✓ 对于删除操作需依被删结点z的情况而定：
 - ①z无左右孩子
 - ②z只有一个孩子
 - ③z的左右孩子均存在

“平衡”的大致意义是：没有任何一个节点过深（深度过大）：

不同的平衡条件，造就出不同的效率表现，以及不同的实现复杂度。



缺陷：失去平衡

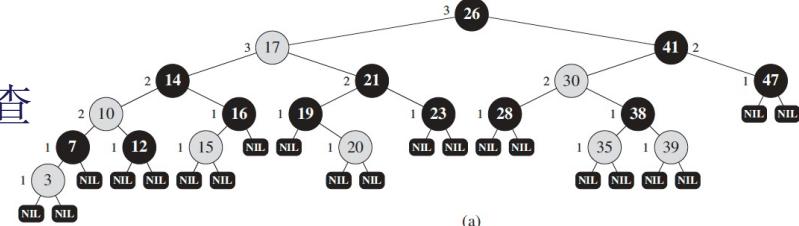
红黑树可实现出平衡二叉搜索树：

确保没有一条路径会比其他路径长出俩倍，因而是接近平衡的。

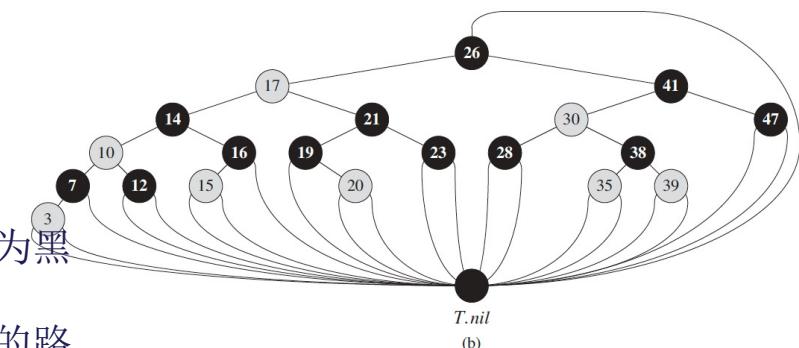


红黑树的特性

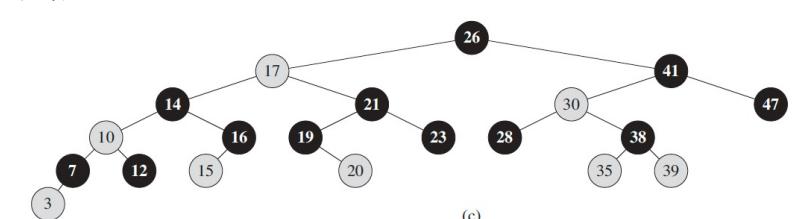
- 1) 红黑树是一棵自平衡的二叉搜索树，所以二叉查找树的所有性质红黑树都具有，
- 此外还有自身五个特性：
 - ✓ ①每个结点要么是黑色要么是红色
 - ✓ ②树根结点的颜色为黑色
 - ✓ ③叶结点(nil)为黑色
 - ✓ ④如果某个结点为红色，则它的左、右孩子结点均为黑色
 - ✓ ⑤对树中任一结点，所有从该结点出发到其叶结点的路径中均包含相同数目的黑色结点
- 外部结点(external node):不包含实际关键字
- 内部结点(internal node):包含实际关键字
- 红黑树结点的形式为：



(a)



(b)



(c)

P	Left	Right	Key	color
---	------	-------	-----	-------



黑高度

□ 定义:

- ✓ 黑高 $bh(x)$ 表示从 x 结点出发(不包含 x 结点)到其叶结点路径上的黑色结点个数。
- ✓ 关于 $bh(x)$ 有两点需要说明:

1) 根据红黑树的"特性(5)"，即从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点"可知，从节点 x 出发到达的所有叶节点具有相同数目的黑节点。这也就意味着， $bh(x)$ 的值是唯一的！

2) 根据红黑色的"特性(4)"，即如果一个节点是红色的，则它的子节点必须是黑色的"可知，从节点 x 出发到达叶节点"所经历的黑节点数目" \geq "所经历的红节点的数目"。假设 x 是根节点，则可以得出结论" $bh(x) \geq h/2$ "。

□ 定理13.1

- ✓ 具有 n 个内部结点的红黑树高度最多为 $2\lg(n+1)$ 。

□ 证明:

✓ 设红黑树的高度为 h ，即要证明 $h \leq 2\lg(n+1)$

$$\checkmark \Rightarrow \frac{h}{2} \leq \lg(n+1)$$

$$\checkmark \Rightarrow 2^{\frac{h}{2}} \leq n+1$$

$$\checkmark \Rightarrow 2^{\frac{h}{2}} - 1 \leq n$$

1) 根据红黑树性质④和⑤，从根到叶结点（不包括根节点）的任何一条简单路径上黑色结点个数至少 $h/2$ ；
所以，根的黑高 $bh \geq h/2$

$$\text{所以, } 2^{bh} - 1 \geq 2^{\frac{h}{2}} - 1$$



如果 $2^{\frac{h}{2}} - 1 \leq 2^{bh} - 1 \leq n$ 成立，则定理13.1得证



- 下面通过"数学归纳法"开始论证高度为 h 的红黑树，它的包含的内节点个数至少为 $2^{bh(x)}-1$ 个"。

1) 归纳基础：当树的高度 $h=0$ 时，

内节点个数是0， $bh(x)$ 为0， $2^{bh(x)}-1$ 也为 0。显然，原命题成立。

2) 当 $h>0$ ，

当树的高度为 h 时，

对于节点 x (x 为根节点)，其黑高度为 $bh(x)$ 。

对于节点 x 的左右子树，它们黑高度为 $bh(x)$ 或者 $bh(x)-1$ 。

根据(02)的已知条件，我们已知 " x 的左右子树，即高度为 $h-1$ 的节点，它包含的节点至少为 $2^{bh(x)-1}-1$ 个"；

所以，节点 x 所包含的节点至少为 = x 的左子树内部结点+ x 的右子树内部结点+1

即， $= (2^{bh(x)-1}-1) + (2^{bh(x)-1}-1) + 1 = 2^{bh(x)}-1$ 。

因此，原命题成立。

- 特别的，当 x 为根结点时，子红黑树至少包含 $2^{bh}-1$ 个内部结点。

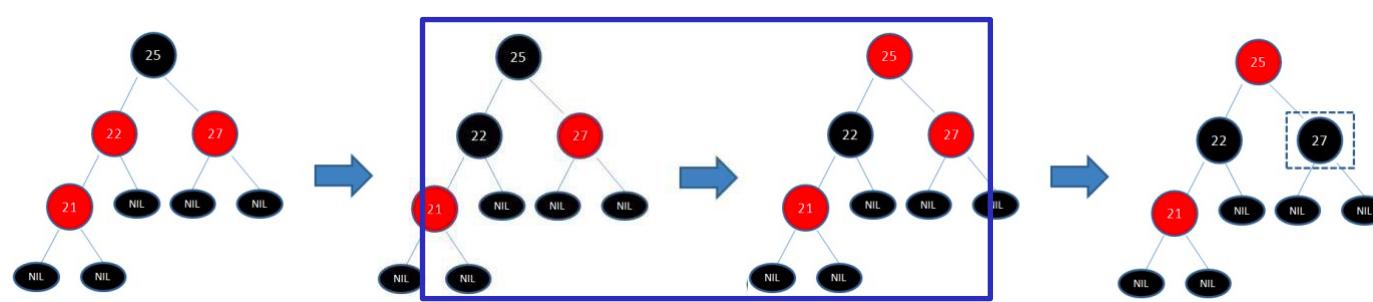
因此，“一棵含有 n 个节点的红黑树 $2^{\frac{h}{2}} - 1 \leq 2^{bh} - 1 \leq n$ 成立，则定理13.1得证。



旋转

- 红黑树的基本操作是添加、删除。在对红黑树进行添加或删除之后，都会用到变色和旋转方法。
 - ✓ 添加或删除红黑树中的节点之后，红黑树就发生了变化，可能不满足红黑树的5条性质，也就不再是一颗红黑树了，而是一颗普通的树。
 - ✓ 而通过变色和旋转，可以使这颗树重新成为红黑树。简单点说，变色和旋转的目的是让树保持红黑树的特性。

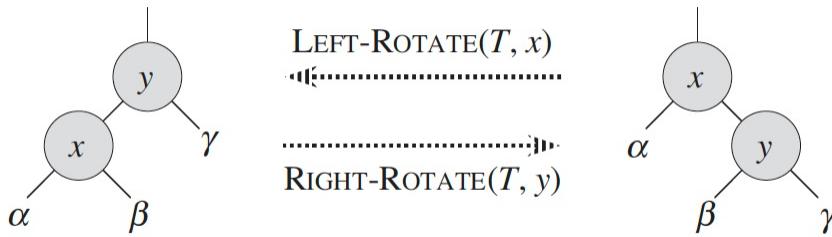
- 变色：



违反了规则4

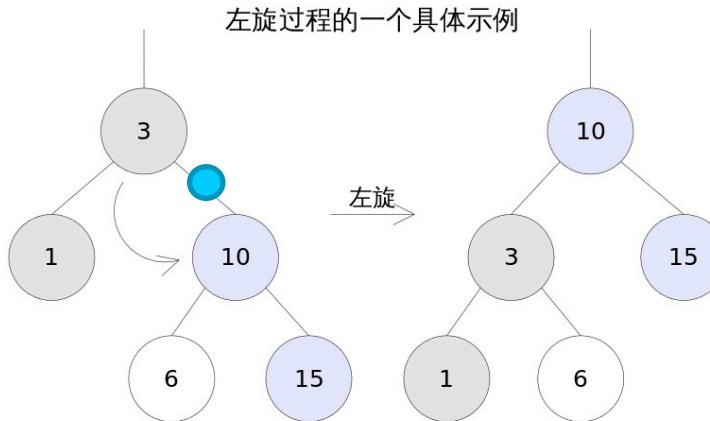
违反了规则5

- 旋转包括两种：左旋 和 右旋。



1) 对 x 进行左旋，意味着，将“ x 的右孩子”设为“ x 的父亲节点”；即，将 x 变成了一个左节点(x 成了为 y 的左孩子)!。因此，左旋中的“左”，意味着“被旋转的节点将变成一个左节点”。

2) 对 x 进行右旋，意味着，将“ x 的左孩子”设为“ x 的父亲节点”；即，将 x 变成了一个右节点(x 成了为 y 的右孩子)!。因此，右旋中的“右”，意味着“被旋转的节点将变成一个右节点”。



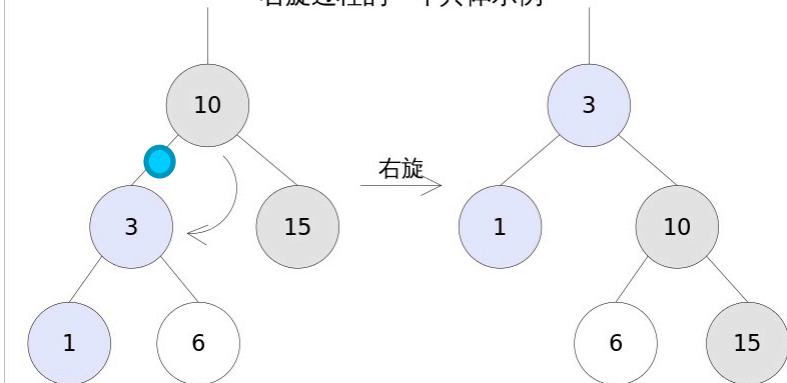
LEFT-ROTATE(T, x)

```

1  y = x.right           // set y
2  x.right = y.left      // turn y's left subtree into x's right subtree
3  if y.left ≠ T.nil
4      y.left.p = x
5  y.p = x.p
6  if x.p == T.nil
7      T.root = y
8  elseif x == x.p.left
9      x.p.left = y
10 else x.p.right = y
11 y.left = x            // put x on y's left
12 x.p = y

```

右旋过程的一个具体示例





红黑树的插入操作

插入一个新结点分为二步完成：

1. 按照二叉查找树的方式插入新结点z
 2. 将插入的节点着色为“红色”。
(不会违背"特性(5)"!)
 3. 恢复红黑树的特性
- 算法如下：

RB-INSERT(T, z)

```
1   $y = T.nil$ 
2   $x = T.root$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11  elseif  $z.key < y.key$ 
12       $y.left = z$ 
13  else  $y.right = z$ 
14   $z.left = T.nil$ 
15   $z.right = T.nil$ 
16   $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )
```

第三步：通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。

第二步中，将插入节点着色为"红色"之后，不会违背"特性(5)"。那它到底会违背哪些特性呢？

a) 对于"特性(1)"，显然不会违背了。

因为我们已经将它涂成红色了。

b) 对于 "特性(2)"，显然也不会违背（根节点不为空）。

插入操作不会改变根节点。所以，根节点仍然是黑色。

c) 对于"特性(3)"，显然不会违背了。

这里的叶子结点是指的空叶子节点，插入非空节点并不会对它们造成影响。

d) 对于"特性(4)"，是有可能违背的！

那接下来，想办法使之"满足特性(4)"，就可以将树重新构造成红黑树了。

(1) 每个节点或者是黑色，或者是红色。

(2) 根节点是黑色。

(3) 每个叶子节点是黑色。 [注意：这里叶子节点，是指为空的叶子节点！]

(4) 如果一个节点是红色的，则它的子节点必须是黑色的。

(5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。



接下来，将分析插入红色节点后红黑树的情况。这里假设要插入的节点为 Z，Z 的父节点为 z.p，祖父节点为 z.p.p，叔叔节点为 y。插入红色节点后，会出现情况，分别如下：

情况一：

插入的新节点 Z 是红黑树的根节点；



情况二：

Z 的父节点是黑色，这种情况下，性质4（每个红色节点必须有两个黑色的子节点）和性质5没有受到影响，不需要调整。



情况三：

Z 节点的父节点是红色，这种情况下，性质4相冲突。这种情况下，被插入节点一定存在非空祖父节点的；

进一步的讲，被插入节点也一定存在叔叔节点（即使叔叔节点为空，我们也视之为存在，空节点本身就是黑色节点）。我们依据“叔叔节点的情况”，将这种情况进一步划分为3种情况(Case)。

RB-INSERT-FIXUP(T, z)

```
1  while  $z.p.color == \text{RED}$ 
2    if  $z.p == z.p.p.left$ 
3       $y = z.p.p.right$ 
4      if  $y.color == \text{RED}$ 
5         $z.p.color = \text{BLACK}$                                 // case 1
6         $y.color = \text{BLACK}$                             // case 1
7         $z.p.p.color = \text{RED}$                          // case 1
8         $z = z.p.p$                                 // case 1
9      else if  $z == z.p.right$ 
10         $z = z.p$                                   // case 2
11        LEFT-ROTATE( $T, z$ )
12         $z.p.color = \text{BLACK}$                       // case 2
13         $z.p.p.color = \text{RED}$                      // case 3
14        RIGHT-ROTATE( $T, z.p.p$ )
15      else (same as then clause
16        with "right" and "left" exchanged)
17       $T.root.color = \text{BLACK}$ 
```



分别处理上述3种插入修复情况：

插入修复case1：

祖父结点的另一个子结点（叔叔结点）是红色。

当前结点的父结点是红色，如下伪代码所示：

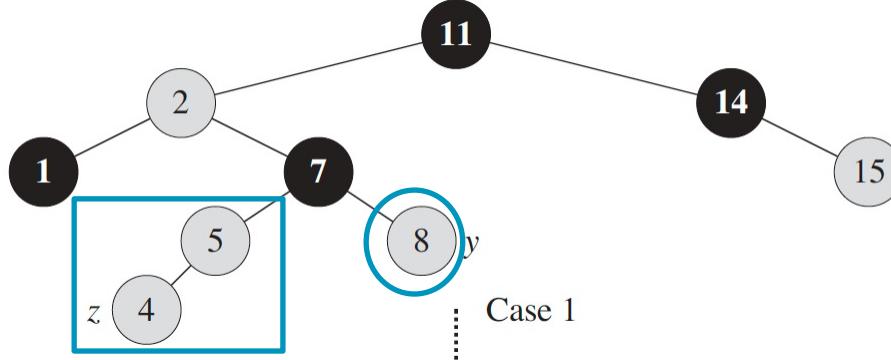
```

1 while z.p.color == RED
2   if z.p == z.p.p.left
3     y = z.p.p.right
4   :
5   else (same as then clause
6       with "right" and "left" exchanged)
7
8 T.root.color = BLACK

```

与此同时，父结点又分为是祖父结点的左孩子还是右孩子，根据对称性，我们只要解开一个方向就可以了。

这里只考虑父结点为祖父左孩子的情况，如下图所示。

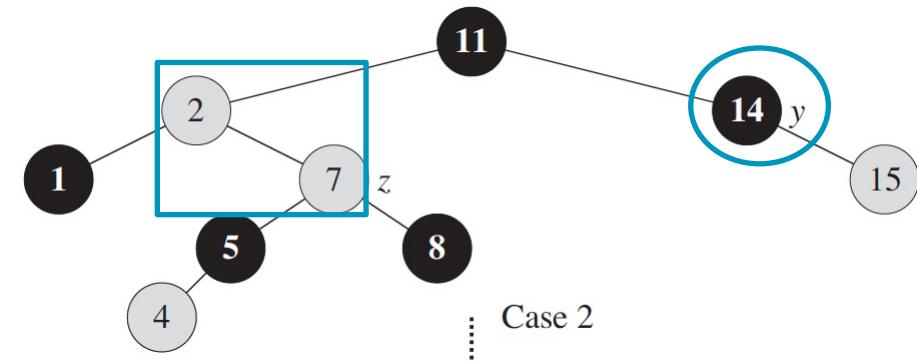


对此，我们的解决策略是：将当前节点的父节点和叔叔节点涂黑，祖父结点涂红，把当前结点指向祖父节点，从新的当前节点重新开始算法。即如下代码所示：

```

4   if y.color == RED
5     z.p.color = BLACK
6     y.color = BLACK
7     z.p.p.color = RED
8     z = z.p.p

```



于是，插入修复情况1可能直接结束，也可能转换成了插入修复情况2。



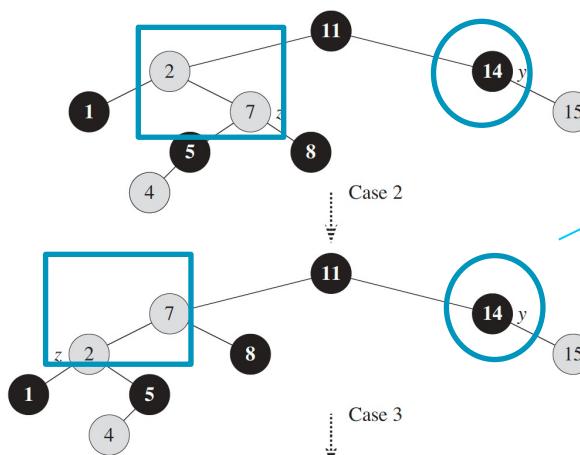
插入修复case2:

- 1) 当前节点的父节点是红色, 叔叔节点是黑色,
- 2) 当前节点是其父节点的右孩子

解决对策是：把当前结点指向父节点，然后以新的当前节点为支点左旋，操作代码为

```
: 9      else if z == z.p.right
10         z = z.p
11         LEFT-ROTATE(T, z)
```

解决对策是：当前节点的父节点做为新的当前节点，以新当前节点为支点左旋。即如下代码所示：

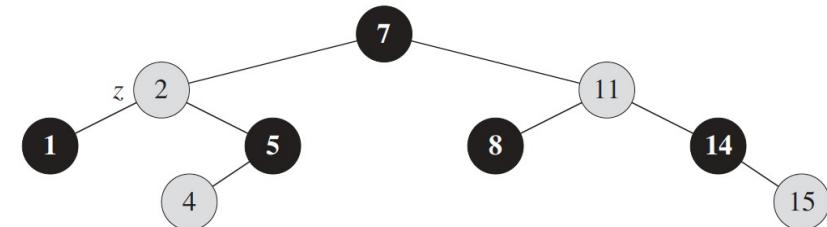


插入修复case3:

- 1) 当前节点的父节点是红色, 叔叔节点是黑色,
- 2) 当前节点是其父节点的左孩子

解决对策是：父节点变为黑色，祖父节点变为红色，在祖父节点为支点右旋，操作代码为：

```
12     z.p.color = BLACK
13     z.p.p.color = RED
14     RIGHT-ROTATE(T, z.p.p)
```



Case 1、2、3就是一个完整的插入修复情况的操作流程



RB-Insert算法分析:

∴ 红黑树的高度 $h=O(\lg n)$

Tree_Insert算法时间为: $O(\lg n)$

RB-Insert-fixup算法时间为: $O(\lg n)$

∴ RB-Insert算法时间为: $O(\lg n)$



红黑树的删除操作

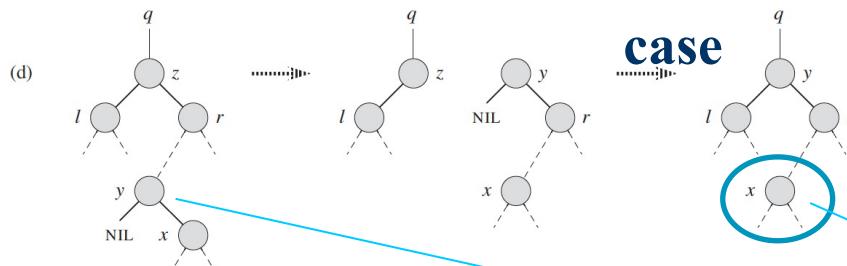
为了保证满足搜索二叉树的结构：我们可以选择其左儿子中的最大元素或者右儿子中的最小元素放到待删除节点的位置，就可以保证搜索二叉树结构的不变。

1) 这和"删除常规二叉查找树中删除节点的方法是一样的"。分3种情况：

① 被删除节点没有儿子。那么，直接将该节点删除就OK了。

② 被删除节点只有一个儿子。那么，直接删除该节点，并用该节点的唯一子节点顶替它的位置。

③ 被删除节点有两个儿子。



RB-TRANSPLANT(T, u, v)

```

1  if  $u.p == T.nil$ 
2     $T.root = v$ 
3  elseif  $u == u.p.left$ 
4     $u.p.left = v$ 
5  else  $u.p.right = v$ 
6   $v.p = u.p$ 

```

子树v替换子树u

通过改变颜色和执行旋转来恢复红黑的性质

记录结点y的踪迹：
Y是要替换z的结点

RB-DELETE(T, z)

```

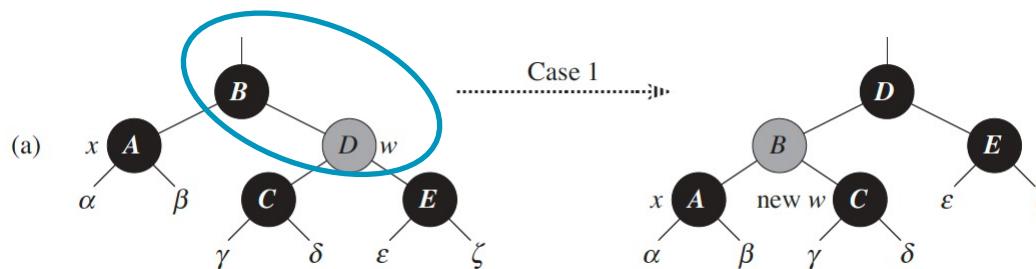
1  y = z
2  y-original-color = y.color
3  if  $z.left == T.nil$ 
4    x = z.right
5    RB-TRANSPLANT( $T, z, z.right$ )
6  elseif  $z.right == T.nil$ 
7    x = z.left
8    RB-TRANSPLANT( $T, z, z.left$ )
9  else  $y = \text{TREE-MINIMUM}(z.right)$ 
10   y-original-color = y.color
11   x = y.right
12   if  $y.p == z$ 
13     x.p = y
14   else RB-TRANSPLANT( $T, y, y.right$ )
15     y.right = z.right
16     y.right.p = y
17   RB-TRANSPLANT( $T, z, y$ )
18   y.left = z.left
19   y.left.p = y
20   y.color = z.color
21   if  $y-original-color == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )

```



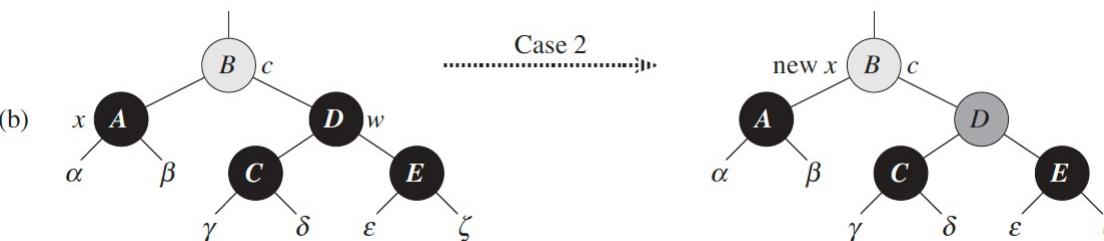
case1: x 的兄弟 w 为红色，则 w 的儿子必然全黑， w 父亲 p 也为黑。

Note: x 子树相对于其兄弟 w 子树少一个黑色节点（原 y 被删除）



改变 p 与 w 的颜色，同时对 p 做一次左旋，这样就将情况1转变为情况2,3,4的一种。

case2: x 的兄弟 w 为黑色， w 的两个儿子也都是黑的（否则，则相应进入了情况3、情况4）， x 与 w 的父亲颜色可红可黑。



0: 如果 x 为红色，则 x 置为黑，则整棵树平衡

1) 因为 x 子树相对于其兄弟 w 子树少一个黑色节点（原 y 被删除 or case1 变化后的情况）：

所以，可以将 w 置为红色，这样， x 子树与 w 子树黑色节点一致，保持了平衡。

2) new x 为 x 与 w 的父亲。

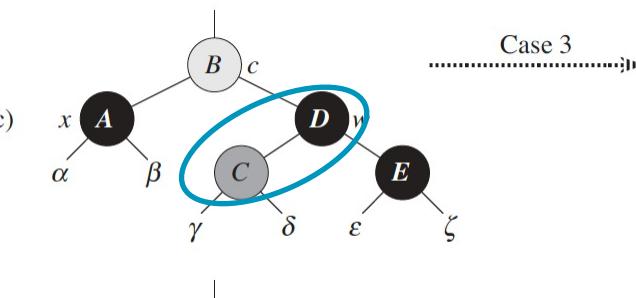
new x 相对于它的兄弟节点new w 少一个黑色节点。

3) 如果new x 为红色，则将new x 置为黑，则整棵树平衡。否则，情况2转变为情况1,2,3,4。



case3: w为黑色，w左孩子红色，右孩子黑色

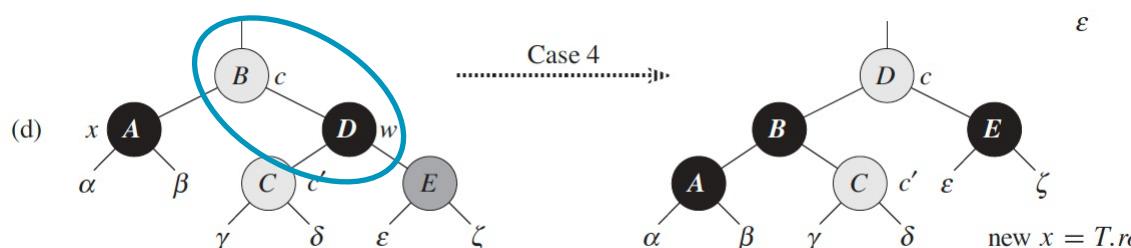
Note: x子树相对于其兄弟w子树少一个黑色节点



交换w与左孩子的颜色，对w进行右旋。转换为情况4

case4: w为黑色，右孩子为红色。

Note: x子树相对于其兄弟w子树少一个黑色节点



交换w与父亲p颜色，同时对p做左旋。这样左边缺失的黑色就补回来了，同时，将w的右儿子置黑，这样左右都达到平衡。

上述四种情况，每一次旋转（左旋、右旋）都伴随着颜色的交换。

总结这四种状况：

1) **case2:** 最好理解，减少右子树的一个黑色节点，使x与w平衡，将不平衡点上移至x与w的父亲。进行下一轮迭代。

2) **case1:** 如果w为红色，通过旋转，转成成case2,3,4进行处理。而case3转换为case4进行处理。也就是说，**case4**是最接近最终解的情况。

3) **情况4:** 右儿子是红色节点，那么将缺失的黑色交给右儿子，通过旋转，达到平衡。



恢复搜索树的红黑性质伪代码

RB-DELETE-FIXUP(T, x)

```
1  while  $x \neq T.root$  and  $x.color == \text{BLACK}$ 
2      if  $x == x.p.left$ 
3           $w = x.p.right$ 
4          if  $w.color == \text{RED}$ 
5               $w.color = \text{BLACK}$ 
6               $x.p.color = \text{RED}$ 
7              LEFT-ROTATE( $T, x.p$ )
8               $w = x.p.right$ 
9          if  $w.left.color == \text{BLACK}$  and  $w.right.color == \text{BLACK}$ 
10              $w.color = \text{RED}$ 
11              $x = x.p$ 
12         else if  $w.right.color == \text{BLACK}$ 
13              $w.left.color = \text{BLACK}$ 
14              $w.color = \text{RED}$ 
15             RIGHT-ROTATE( $T, w$ )
16              $w = x.p.right$ 
17              $w.color = x.p.color$ 
18              $x.p.color = \text{BLACK}$ 
19              $w.right.color = \text{BLACK}$ 
20             LEFT-ROTATE( $T, x.p$ )
21              $x = T.root$ 
22     else (same as then clause with "right" and "left" exchanged)
23      $x.color = \text{BLACK}$ 
```

RB-delete算法时间：

∴ 红黑树的高度 $h = O(\lg n)$

Tree-delete算法时间为
 $O(\lg n)$
RB-delete-fixup算法时
间为 $O(\lg n)$

∴ RB-delete算法时间为
 $O(\lg n)$