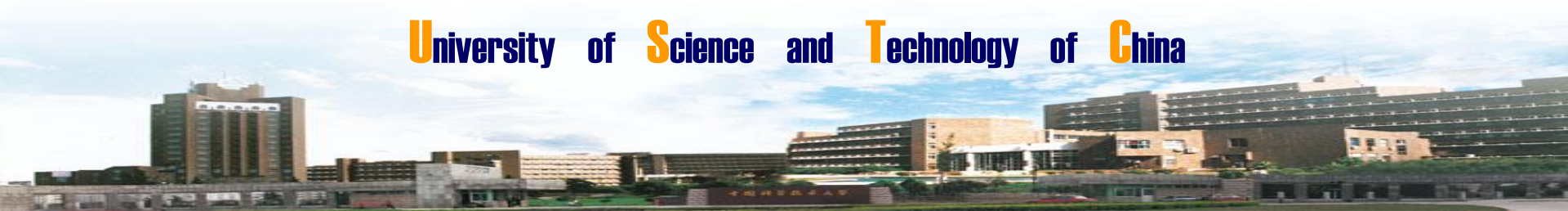
算法基础

主讲人： 庄连生

Email: { *lszhuang@ustc.edu.cn* }

***Spring 2018*，USTC**



第八讲 红黑树及其扩张

内容提要：

- 红黑树性质
- 红黑树的操作
- 红黑树的扩张

第八讲 红黑树及其扩张

内容提要：

- 红黑树性质
- 红黑树的操作
- 红黑树的扩张

红黑树性质

- 二叉查找树

① 定义：一颗二叉查找树是按二叉树结构来组织的；

② 性质：设x是二叉查找树中的一个结点。如果y是x的左子树中的一个结点，则key[y]<=key[x]；如果y是x的右子树中的一个结点，则key[y]>=key[x]。

- 二叉查找树可以按照中序遍历算法按排序顺序输出树中的所有关键字；

* 定理：如果x是一棵包含n个结点的子树的根，调用中序遍历算法来输出二叉查找树中所有关键元素的时间复杂度为 θ (n).

- 注意：根据二叉查找树的中序遍历结果并不能推断出二叉查找树的结构。

红黑树性质

- 对于一棵高度为h的二叉查找树，其动态集合操作Search、Minimum、Maximum、Successor、Predecessor、Insert、Delete的运行时间为 θ (h)。

  => 树的高度决定了在树上操作的成本；

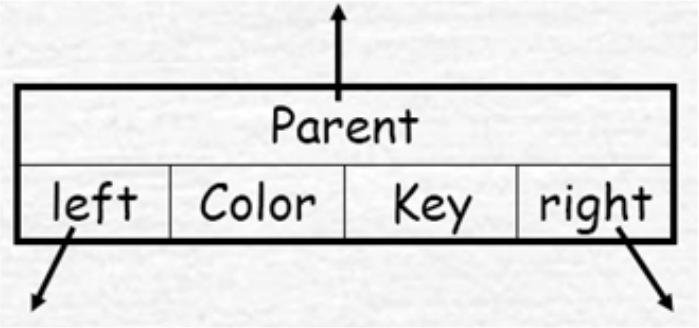
- 一些搜索树的高度：
  - 平衡二叉搜索树：O(logn)
  - 1962年提出的AVL树：≤1.44logn
  - 1972年提出的红黑树：≤2log(n+1)

- 红黑树是通过对二叉查找树结点上增加一个存储位表示结点的颜色（红色或黑色），通过对任何一条从根到叶子的路径上各个结点着色方式的限制，确保没有一条路径会比其他路径长出两倍，从而接近平衡。

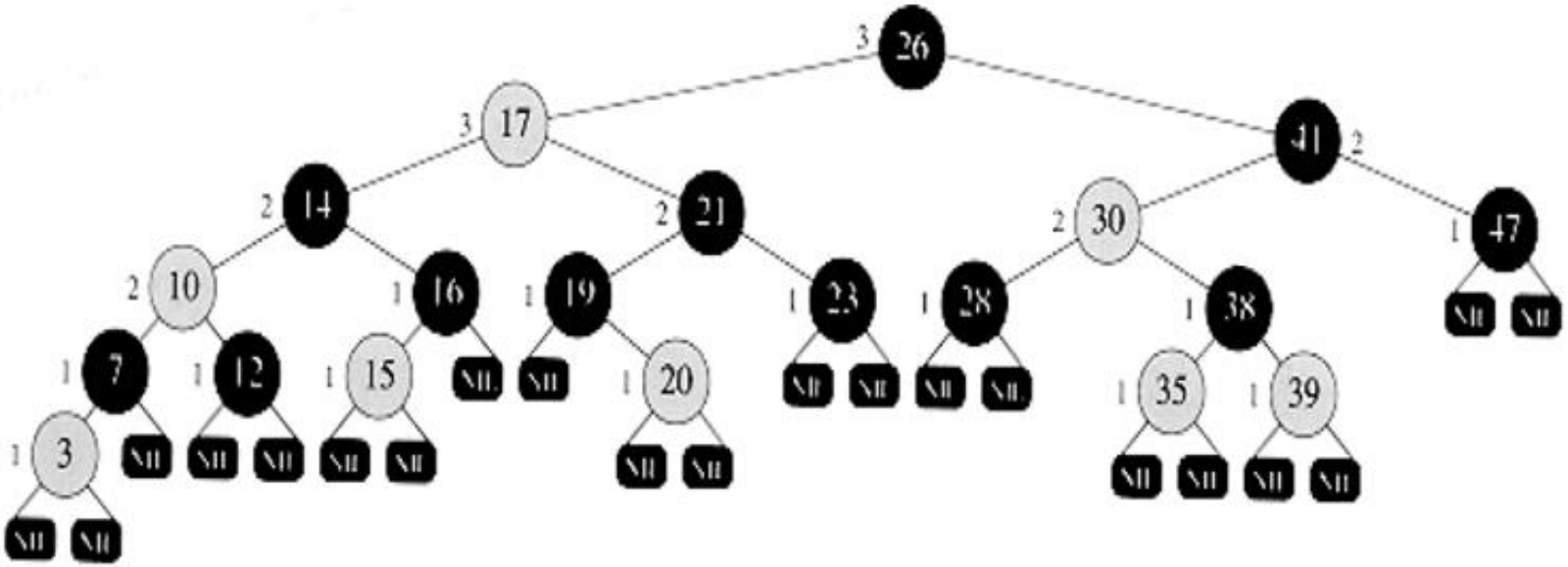
红黑树性质

- 定义：红黑树是满足如下性质的二叉查找树：
  - ① 每个结点必须为红色或黑色； //性质1
  - ② 根为黑色； //性质2
  - ③ 树中的nil叶子为黑色； //性质3
  - ④ 若结点为红，则其两个子孩子必为黑； //性质4
  - ⑤ 对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。 //性质5

- 结点结构：



红黑树性质



表达方式：

图(a)每个空指针域均连接到一个叶节点nil，比较浪费存储空间；

图(b)所有空指针域共享一个哨兵nil[T]，nil[T]为黑色；

图(c)省略nil[T]；

红黑树性质

- 定义2：从某个结点x出发（不包括该结点）到达一个叶结点的任意一条路径上，黑色结点的个数称为该结点x的黑高度，用bh(x)表示。
- 定义3：红黑树的黑高度定义为其根结点的黑高度，记为bh(root[T])。
- 引理13.1：一棵n个内结点的红黑树的高度至多为2lg(n+1)

红黑树性质

- **Proof:**

① 先证对任何以x为根的子树其内节点数 $\geq 2^{bh(x)}-1$

归纳基础：当bh(x)=0时，x就是nil[T]

∴ $2^{bh(x)}-1=2^0-1=0$ 即为0个内节点，正确

归纳假设：对x的左右孩子命题正确

归纳证明：∵ x的左右孩子的黑高或为bh(x)或为bh(x)-1

∴ x的内点数=左孩子内点数+右孩子内点数+1

$\geq (2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1$

$= 2^{bh(x)}-1$

即第①点得证。

红黑树性质

- **Proof(Cont.):**

② 证明bh(root[T])≥h/2, h为红黑树的树高

∵ 红点的孩子必为黑 //红黑树的性质4

∴ 红点的层数<h/2

因此 ⇒ bh(root[T])≥h/2

③ 证明最后结论

∵ 红黑树有n个内点

由① ⇒ $n \geq 2^{bh(root[T])}-1 \geq 2^{h/2}-1$

∴ ⇒ $h \leq 2\log(n+1)$

第八讲 红黑树及其扩张

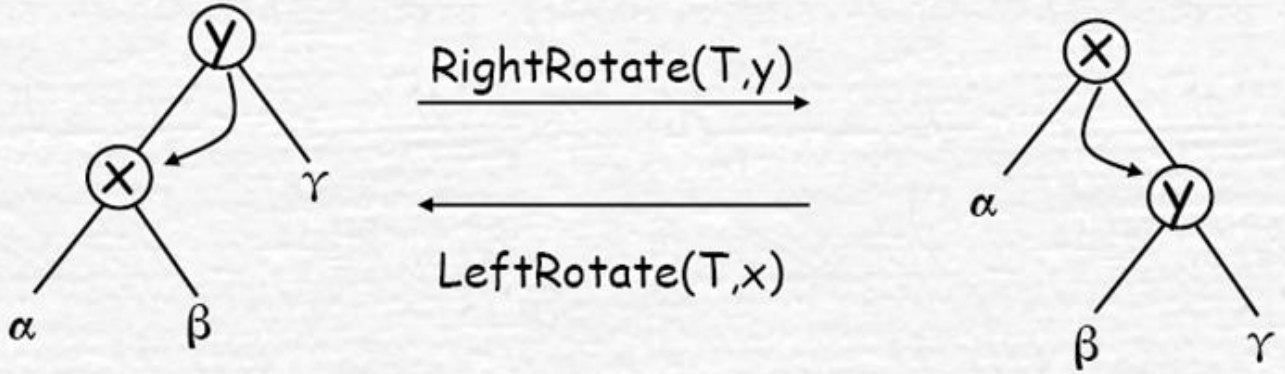
内容提要：

- 红黑树性质
- 红黑树的操作
- 红黑树的扩张

红黑树的操作—旋转

旋转操作

左、右旋转的图示

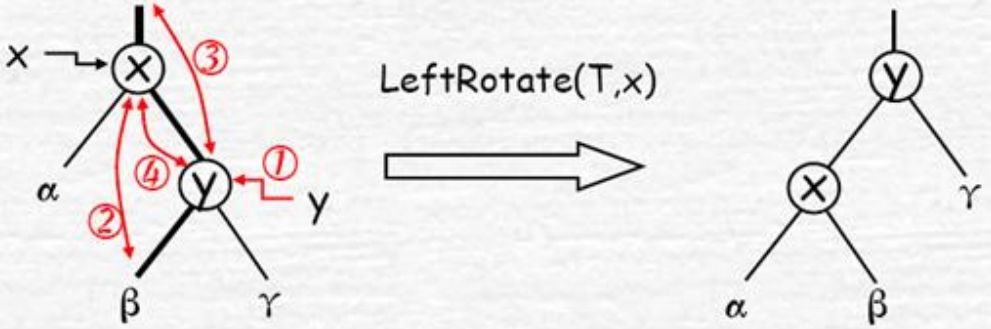


注：旋转过程中二叉搜索树(BST)性质不变：$\alpha \leqslant x \leqslant \beta \leqslant y \leqslant \gamma$

红黑树的操作—旋转

- 左旋以x和y之间的链为“支轴”进行，使y成为该子树新的根，x成为y的左孩子，而y的左孩子则成为x的右孩子。

- 左旋图示：



- 步骤解释：需要变动的是3根粗链

临界情形

① y←right[x] //记录指向y节点的指针

② right[x]←left[y], p[left[y]]←x // β 连到x右 β=nil[T]

③ p[y]←p[x], p[x]的左或右指针指向y //y连到p[x] P[x]=nil[T]，即x为根

④ Left[y]←x, p[x]←y //x连到y左

注：- 要注意先后顺序；- 每条边的修改涉及双向；

- 要考虑临界情形(特殊情形)；

红黑树的操作—旋转

- 左旋转算法：

```
LeftRotate(T, x)
{ //假定right[x] ≠ nil[T]
  //step ①
   y ← right[x];
  //step ②
   right[x] ← left[y]; p[left[y]] ← x;
  //step ③
   p[y] ← p[x];
   if p[x]=nil[T] then        //x是根
       root[T] ← y;           //修改树指针
   else if x=left[p[x]] then left[p[x]] ← y;
        else right[p[x]] ← y;
  //step ④
    left[y] ← x; p[x] ← y;
}
T(n)=O(1)
```

红黑树的操作—插入

- 算法步骤：

Step 1：将结点z按照BST树规则插入红黑树中，z是叶子节点

Step 2：将结点z涂红；

Step 3：调整使其满足红黑树的性质；

- 插入算法

```
RBInsert(T, z)
{  y ← nil[T];              //y用于记录：当前扫描节点的双亲节点
   x ← root[T];             //从根开始扫描
   while x ≠ nil[T] do      //查找插入位置
   {  y ← x;
      if key[z] < key[x] then    //z插入x的左边
         x ← left[x];
      else
         x ← right[x];           //z插入x的右边
   }
```

红黑树的操作—插入

```
p[z] ← y;                    //y是z的双亲
if y = nil[T] then           //z插入空树
  root[T] ← z;               //z是根
else
  if key[z] < key[y] then
     left[y] ← z;            //z是y的左子插入
  else
     right[y] ← z;           //z是y的右子插入
 left[z] ← right[z] ← nil[T];
 color[z] ← red;
 RBInsertFixup(T, z);
}
```

时间： T(n)=O(logn)

红黑树的操作—插入

- 函数RBInsertFixup用来实现对红黑树进行调整，以保证红黑树性质；

- 调整分析
  - idea：通过旋转和改变颜色，自下而上调整（z进行上溯），使树满足红黑树；
  - z插入后违反红黑树的情况：

    ∵z作为红点，其两个孩子为黑（nil[T]）

    ∴满足性质1，3，5

    可能违反性质2：z是根

    可能违反性质4：p[z]是红
  - 调整步骤：

    (1)若z为根，将其涂黑；

    (2)若z为非根，则p[z]存在

    ①若p[z]为黑，无需调整

红黑树的操作—插入

②若p[z]为红，违反性质4，则需调整

∵ p[z]为红，它不为根

∴ p[p[z]]存在且为黑

- 分6种情况进行调整：

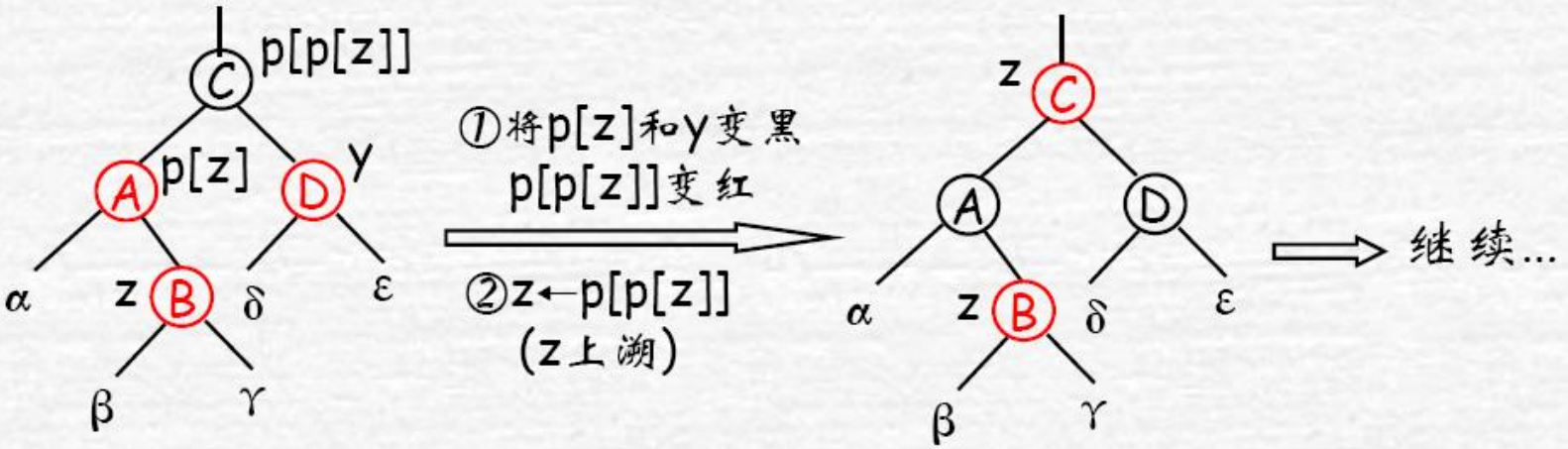
其中

case1~3为z的双亲p[z]是其祖父p[p[z]]的左孩子，

case4~6为z的双亲p[z]是其祖父p[p[z]]的右孩子。

红黑树的操作—插入

Case 1：z的叔叔y是红色



注：(1)变换后，新的z(上溯后)可能违反性质4，故调整最多至根；

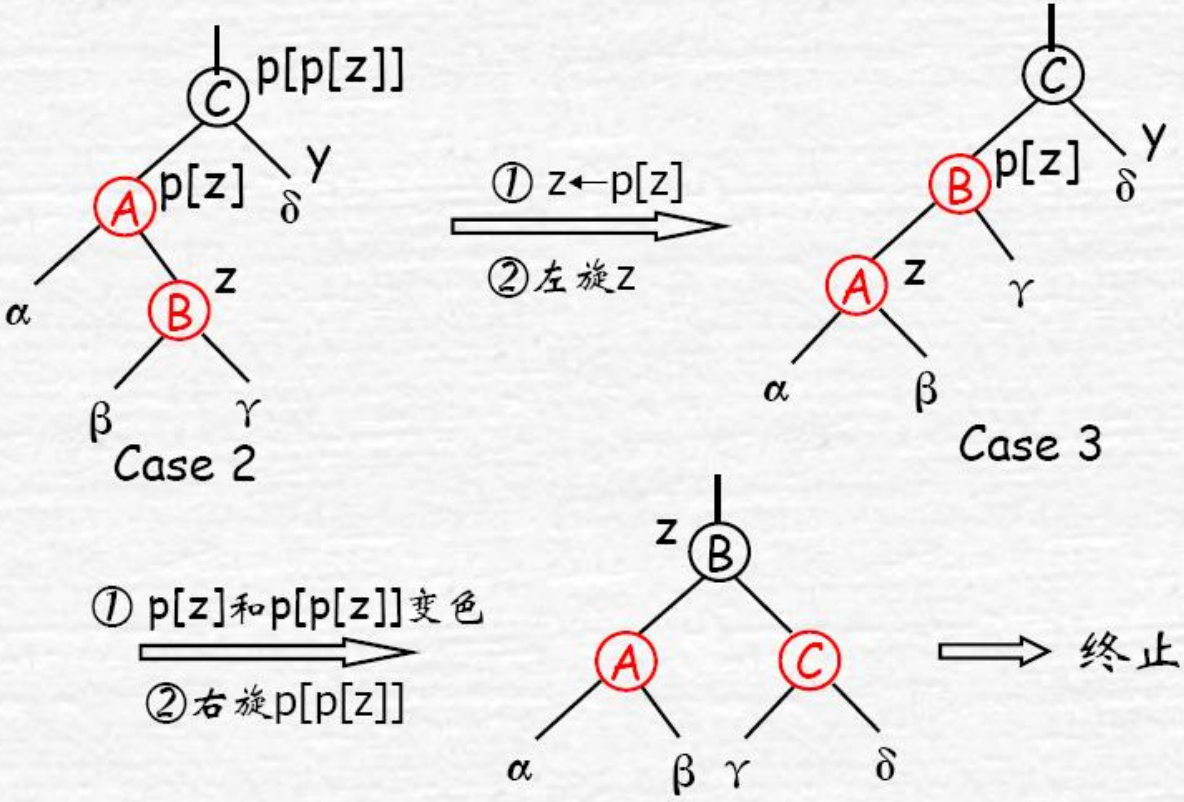
(2)若红色传播到根，将根涂黑，则树的黑高增1；//临界处理

(3)z是p[z]的左、右孩子均一样处理；

红黑树的操作—插入

Case 2：当z的叔叔y是黑色，且z是双亲p[z]的右孩子

Case 3：当z的叔叔y是黑色，且z是双亲p[z]的左孩子



红黑树的操作—插入

- RBInsertFixup算法

```
RBInsertFixup(T, z)
{  while ( color[p[z]]=red ) do
   { //若z为根，则p[z]=nil[T]，其颜色为黑，不进入此循环
     //若p[z]为黑，无需调整，不进入此循环
      if p[z]=left[p[p[z]]] then   //case 1,2,3
      {  y ← right[p[p[z]]];     //y是z的叔叔
         if color[y]=red then   //case 1
         {  color[y]=black; color[p[z]]=black;
            color[p[p[z]]]=red; z ← p[p[z]];
          }
         else    //case 2 or case 3   y为黑
```

红黑树的操作—插入

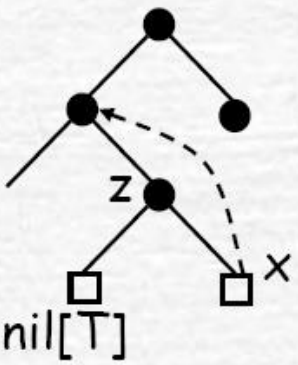
```
        else    //case 2 or case 3  y为黑
        {  if z=right[p[z]] then //case 2
           {  z ← p[z];        //上溯至双亲
              leftRotate(T, z);
            }//以下为case 3
            color[p[z]]=black; color[p[p[z]]]=red;
            RightRotate(T, p[p[z]]); //p[z]为黑，退出循环
           } //case 1's endif
        } //case 2 or 3' s
       else  //case 4,5,6's 与上面对称
       { … … }
    } //endwhile
  color[root[t]] ← black;
}
```

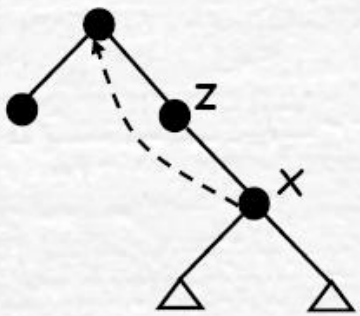
红黑树的操作

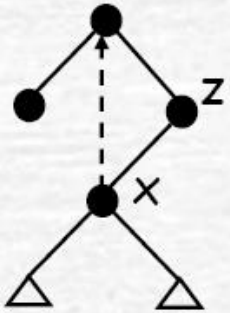
- 算法的时间复杂性
  - 调整算法的时间：O(logn)
  - 整个插入算法的时间：O(logn)
  - 调整算法中至多使用2个旋转

红黑树的操作—删除

- z删除后BST的调整
  - case 1：z为叶子；　case 2：z只有一个孩子(非空)







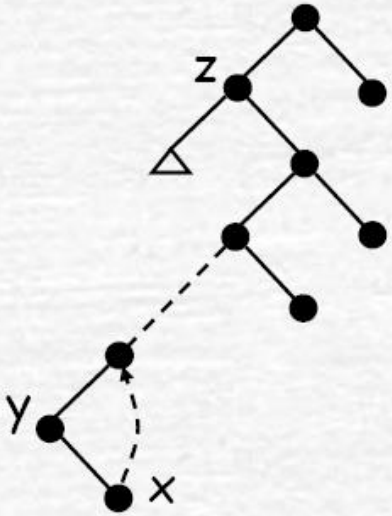
注：(1)删除z，连接x。这里x是z的中序后继；

(2)case 1是case 2的特例，因为处理模式是一样的。

(3)z是p[z]的左孩子，类似讨论；

红黑树的操作—删除

- case 3：z的两个孩子均非空；



注：(1)找z的中序后继，即找z的右子树中最左下节点y；

(2)删除y，将y的内容copy到z，再将y的右子连到p[y]左下。

红黑树的操作—删除

- 删除算法

```
RBDelete(T, z)
{  if (left[z]=nil[T]) or (right[z]=nil[T]) then //case 1,2
      y ← z;          //后面进行物理删除y
   else //z的两子树均非空，case 3
      y ← TreeSuccessor(z);  //y是z的中序后继
   //此时，y统一地是x的双亲节点且是要删除节点
   //x是待连接到p[y]的节点，以下要确定x
   if left[y] ≠ nil[T] then //本if语句综合了case1,2,3的x
      x ← left[y];
   else
      x ← right[y];
   //以下处理：用x取代y与y的双亲连接
   p[x] ← p[y];
   if p[y]=nil[T] then  //y是根
      root[T] ← x;          //根指针指向x
   else //y非根
      if  y=left[p[y]] then  //y是双亲的左子
         left[p[y]] ← x;
      else
         right[p[y]] ← x;
```

红黑树的操作—删除

```
    if y≠z then                    //case 3
       y的内容copy到z;
    if color[y]=black then
       RBDeleteFixup(T, x);   //调整算法
    return y;     //实际是删除y节点
}
```

- 删除操作总是在只有一边有孩子的节点或者叶子节点上进行的，绝不会在一个有二个孩子的节点上进行删除操作。而successor函数只有在节点有2个孩子的时候被调用，这个时候，该函数一定是沿节点的右子树向下进行的，最终会找到一个只有一个孩子的节点。

- 如果删除一个红色的节点，红黑性质得以保持：
  ① 树中各节点的黑高度都没有变化；
  ② 不存在两个相邻的红色节点；
  ③ 根仍然是黑色的，因为如果y是红的，就不可能为根；

红黑树的操作—删除

- 如果删除的是黑色节点会出现如下问题：

① 删除了根且一个红色节点做了新的根，则性质1将会被破坏；

② 该节点和其父节点都是红的，则违反了性质4；

③ 删除y将导致先前包含y的任何路径上黑节点的个数减少1个，破坏性质5；

- 调整算法：RBDeleteFixup(T, x)
  - 讨论

x：或是y的唯一孩子；或是哨兵nil[T]

可以想象将y的黑色涂到x上，于是

- 若x为红，只要将其涂黑，调整即可终止；
- 若x为黑，将y的黑色涂上之后，x是一个双黑节点，违反性质1。

处理步骤如下：

step 1：若x是根，直接移去多余一层黑色(树黑高减1)，终止；

step 2：若x原为红，将y的黑色涂到x上，终止；

step 3：若x非根节点，且为黑色，则x为双黑。通过变色、旋转使多余黑色向上传播，直到某个红色节点或传到根；

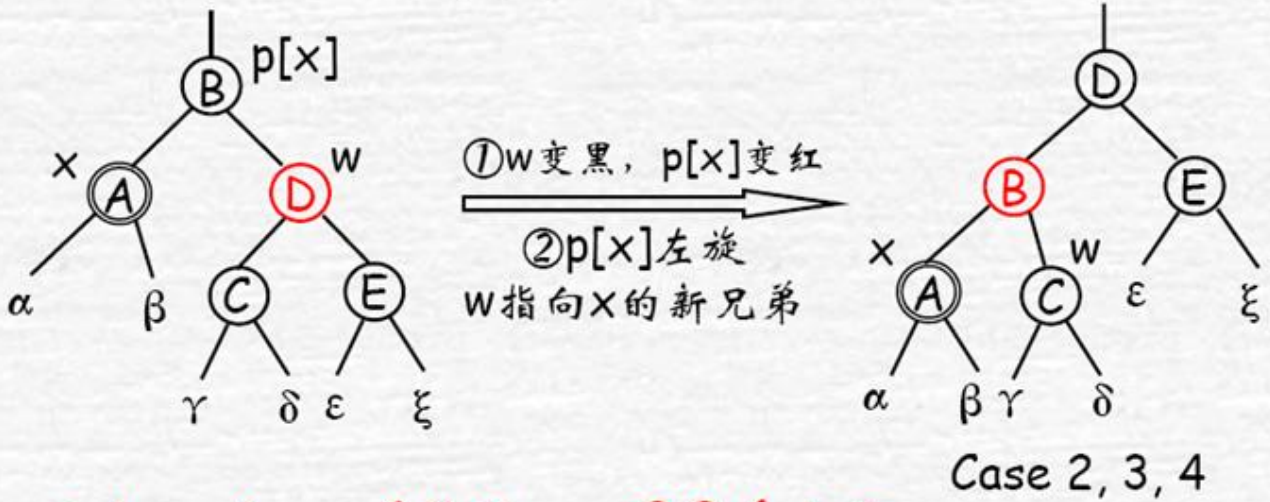
红黑树的操作—删除

- 调整分8种情况

case 1~4为x是p[x]的左子；case 5~8为x是p[x]的右子

case 1：x的兄弟w是红色

∵ w是红，　∴ p[x]必黑

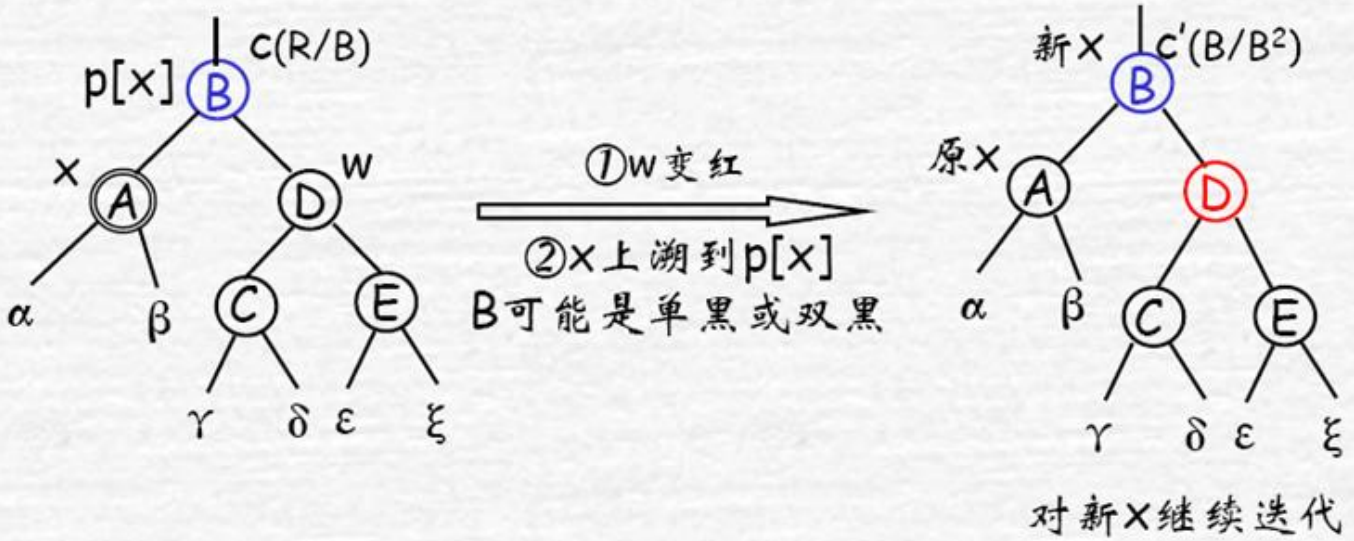


Case 2, 3, 4

目标：将case1转为case2,3,4处理

红黑树的操作—删除

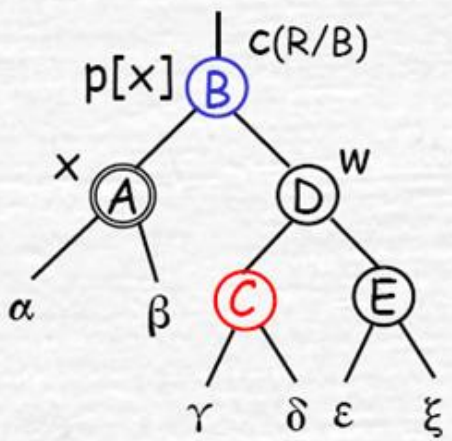
case 2：X的黑兄弟W的两个孩子均为黑



目标： x上移到B，通过A和D的黑色上移

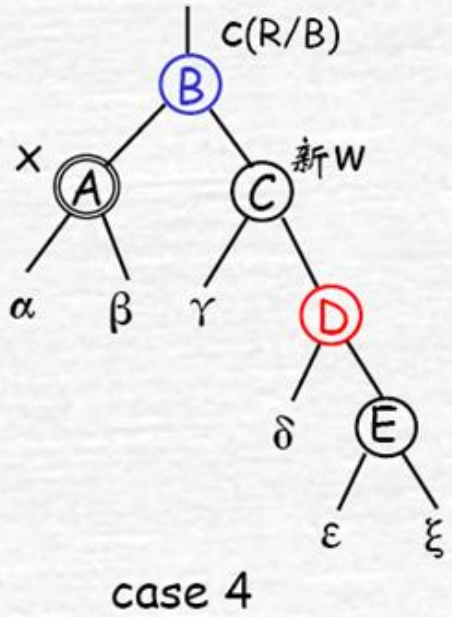
红黑树的操作—删除

case 3：X的黑兄弟W的右子为黑且左子为红



①W左子变黑，W变红

②W右旋，W指向X的新兄弟

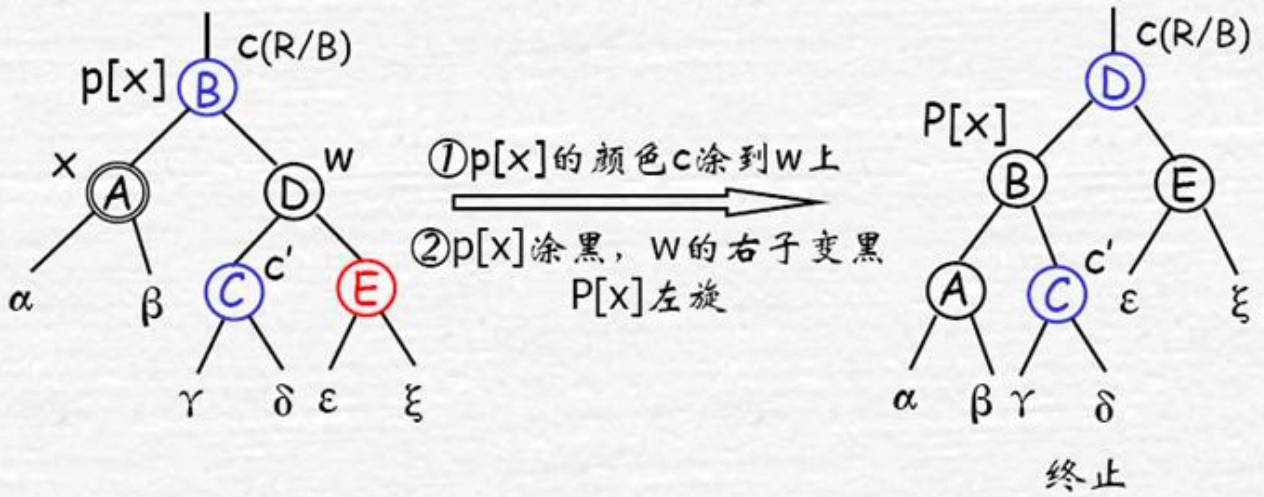


case 4

目标：将case3转为case4

红黑树的操作—删除

case 4：x的黑兄弟w的右子为红(左子为黑或红)



目标：终结处理。x的黑色上移给B，B的原色下移给D，D将黑色下移给C和E，通过旋转解决矛盾点C

- DeleteFixup(T, x)算法：P289

红黑树的操作—删除

```
RBDeleteFixup(T,x)
{//
  while  x≠root[T]  and color[x] = BLACK
         do  if  x=left[p[x]] {
                 then  w ← right[p[x]]
                       if  color[w]=RED
                           then  color[w] ← BLACK;
                                 color[p[x]] ← RED;
                                 LEFT-ROTATE( T, p[x] )
                                 w ← right[ p[x] ];
                       if color[left[w]] = BLACK  and  color[right[w]]=BLACK
                           then  color[w] ← RED
                                 x ← p[x]
                           else if  color[right[w]]=BLACK
                                    then  color[left[w]] ← BLACK
                                          color[w] ← RED
                                          RIGHT-ROTATE( T, w)
                                          w ← right[p[x]]
                                  color[w] ← color[p[x]]
                                  color[p[x]] ← BLACK
                                  color[right[w]] ← BLACK
                                  LEFT-ROTATE( T, p[x])
                                  x ← root[T]
          } else {
              //处理情况和上面对称，只要把right和left对换就可以了；
          }
```

第八讲 红黑树及其扩张

内容提要：

- 红黑树性质
- 红黑树的操作
- 红黑树的扩张

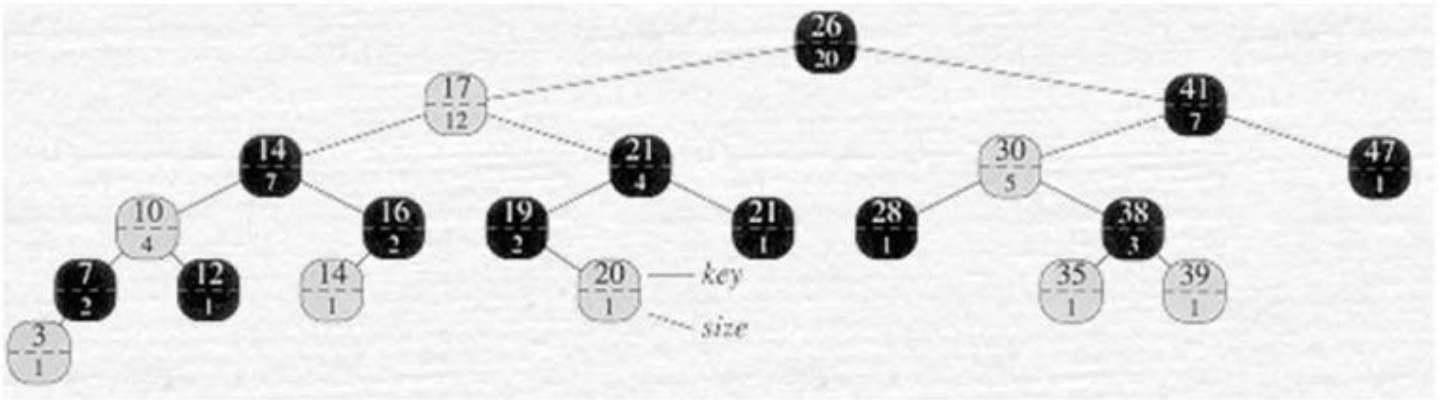
红黑树的扩张—顺序统计树

- OS树的定义：

  OS(Order-Statistic)树是一棵扩充的红黑树，在每个节点上扩充一个域size[x]而得到的。Size[x]域表示以x为根的子树中内部节点的总数（包括x），即子树大小。

$$size[x]=\begin{cases}0 & if\ x=nil[T]\\ size[left[x]]+size[right[x]]+1 & other\end{cases}$$

- OS树的一个图例：



红黑树的扩张—顺序统计树

- 选择问题：在以x为根的子树中，查找第i个最小元素；
- 主要步骤：

① 计算以x为根的子树中节点x的排序r（=左子树节点个数+当前节点）；

② 如果i = r，则x就是待查找的最小元素；

③ 如果i < r，则第i小元素就在x的左子树中，到左子树中递归查找第i小元素；

④ 如果i > r，则第i小元素就在x的右子树中，到右子树中递归查找第i-r小元素；

- OS-SELECT(x, i)

```
{
    r ← sizeof[left[x]] + 1;
    if i = r
        then return r;
     elseif i < r
        then return OS-SELECT(left[x], i );
      else return OS-SELECT( right[x], i-r);
}
```

红黑树的扩张—顺序统计树

- 求秩问题：在OS树中，给定元素x求其在中序遍历得到的线性序中x的位置；
- 主要步骤：

  Step 1: 计算在以x为根的子树中，x的秩r ；

  Step 2: ①如x是根，则返回r；

  ②若x是双亲的左孩子，则x在以p[x]为根的子树中的秩是r；

  ③若x是双亲的右孩子，则x在以p[x]为根的子树中的秩是
  r + sizeof[left[p[x]]] + 1;

  ④x上移至p[x]；

  Step 3：重复② ③ ④，直至情况①成立时终止。

- 时间复杂度：O(log n)

红黑树的扩张—顺序统计树

- OS-RANK(T, x)

```
{
    r ← sizeof[left[x]] + 1;
    y ← x ;
    while  y≠root[T]
           do if  y=right[p[y]]
                  then r ← r+size[left[p[y]]]+1;
              y ← p[y];
    return r;
}
```

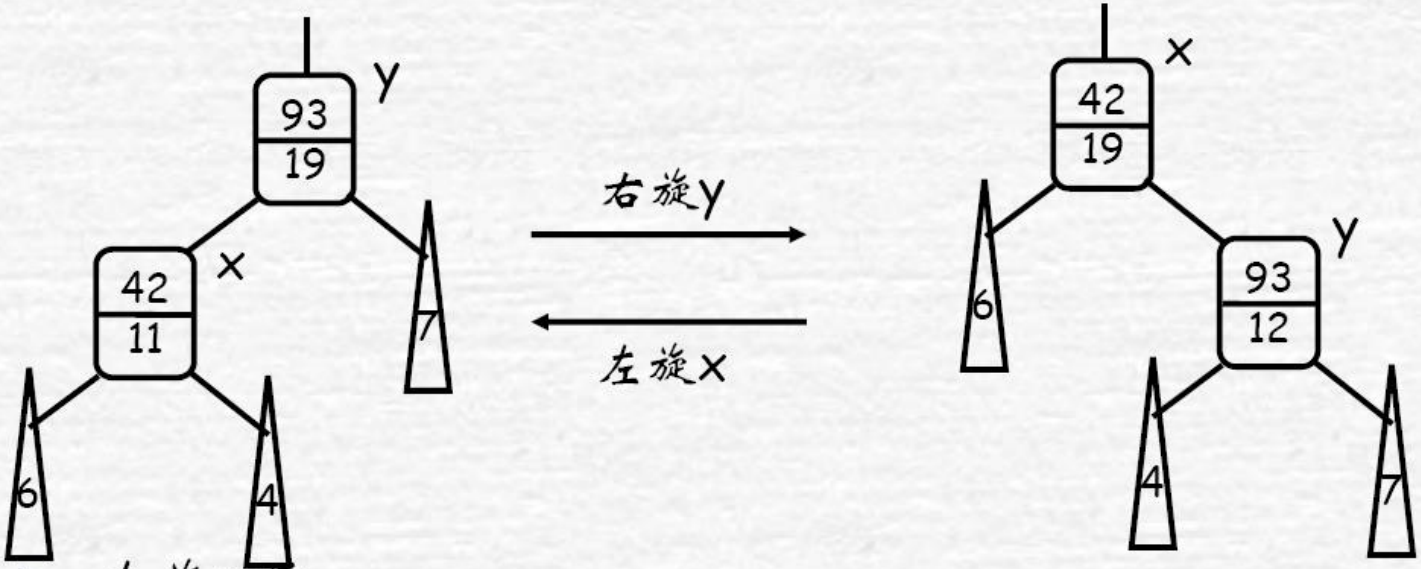
- 时间复杂度：O(lg n)

红黑树的扩张—顺序统计树

- 插入算法
  - Phase 1：从根向下插入新节点，将搜索路径上所经历的每个节点的size+1，新节点的size置为1；
    - 附加成本：O(logn)
  - Phase 2：采用变色和旋转方法，从叶子向上调整；
    - 变色不改变size；
    - 旋转可能改变size：
      ∵ 旋转是局部操作，
      又，只有轴上的两个节点的size可能违反定义
      ∴ 只需要在旋转操作后，对违反节点size进行修改
    - 附加成本：旋转为O(1)，总成本为O(logn)，

红黑树的扩张—顺序统计树

- 例：LeftRotate(T, x)



➢ 左旋x后：

size[y] ← size[x]

size[x] ← size[left[x]]+size[right[x]]+1

∵插入过程至多有2个旋转

∴附加成本为O(1)

红黑树的扩张—顺序统计树

- 删除算法
  - Phase 1：物理上删除y，在删除y时从y上溯至根，将所经历的节点的size均减1；
    - 附加成本：O(logn)
  - Phase 2：采用变色和旋转方法，从叶子向上调整；
    - 变色不改变size；
    - 旋转可能改变size，至多有3个旋转；
    - 附加成本：O(logn)

- Remark：上面介绍的插入和删除均是有效维护，有效维护保证扩充前后的基本操作的渐近时间不变。

红黑树的扩张—如何扩张

- 扩充的目的：设计新的操作、加速已有的操作；
- 扩充的步骤：
  ① 选择基础数据结构；
  ② 确定要在基础数据结构中添加附加信息
  ③ 可以有效地维护附加信息
  ④ 设计新的操作
- 例如：OS树
  ① 选择红黑树作为基本数据结构
  ② 在红黑树上增加Size域；
  ③ 有效性证明；
  ④ 设计OS-SELECT、OS-RANK操作

红黑树的扩张—如何扩张

- 定理14.1

  假设f是红黑树T的n个节点上扩充的域，

  对∀x∈T，假设x的f域的内容能够仅通过节点x、left[x]、right[x]的信息（包括f域）的计算就可以得到，则

  扩充树上的插入和删除维护操作（包括对f域的维护）不改变原有的渐近时间O(logn)。

红黑树的扩张—区间树

- **问题描述：**

一个事件占用一段连续事件，因此，每个事件可以用一个区间来表示。我们经常要查询一个由时间区间数据构成的数据库，以找出特定的区间内发生了什么事件。如何设计有效的数据结构来维护这样一个区间数据库？

比如：特定时间课程查询或活动查询。

- **区间树是解决该问题的一个非常有效的数据结构！**

红黑树的扩张—区间树

- 区间树是通过扩张红黑树以支持由区间构成的动态集合上的操作；
- 基本概念：
  - ① 区间：表示占用一段连续时间的事件；
  - ② 闭区间：实数的有序对[t1,t2]，t1 ≤t2；
  - ③ 区间的对象表示：区间[t1,t2]可以用对象i表示，有两个属性：low[i] = t1，high[i]=t2；
  - ④ 区间重叠：

$$i \cap i' \neq \varnothing \Leftrightarrow (low[i] \leqslant high[i']) \text{ and } (low[i'] \leqslant high[i])$$

区间树支持下列操作：

INTERVAL-INSERT(T, x)：将包含区间域int 的元素x插入到区间树T中；
INTERVAL-DELETE(T,x)：从区间树T中删除元素x；
INTERVAL-SEARCH(T, i)：返回一个指向区间树T中元素x的指针，使int[x]与i重叠；若集合中无此元素存在，则返回nil[T]；

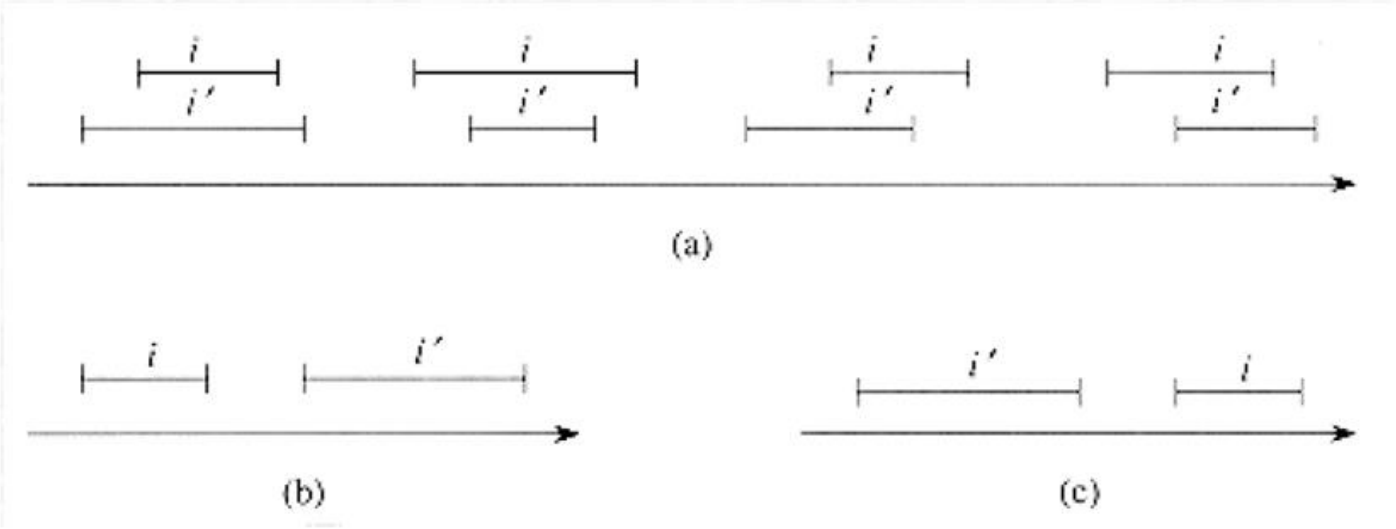
红黑树的扩张—区间树

- 区间三分法：

  (a)i和i'重叠

  (b)i在i'前：high[i]<low[i']

  (c)i'在i前：high[i']<low[i]



红黑树的扩张—区间树

- 扩充步骤：

Step 1：基本结构

以红黑树为基础，对$\forall x \in T$，x包含区间int[x]的信息(低点和高点)，key=low[int[x]]。

Step 2：附加信息

max[x]=max(high[int[x]], max[left[x]], max[right[x]])

Step 3：维护附加信息(有效性)

由定理14.1及max的定义⇒有效

Step 4：开发新操作

查找与给定区间重叠的区间

节点x

红黑树的扩张—区间树

- 查找算法IntervalSearch(T, i)

  基本思想：

  step 1：x ← root[T]；//从根开始查找

  step 2：若x≠nil[T]且i与int[x]不重叠

  　　if x的左子树非空且左子树中最大高点≥low[i] then

  　　x ← left[x]；//到x的左子树中继续查找

  　　else //左子树必查不到，到右子树查

  　　x ← right[x]；

  step 3：返回x　//x=nil or i和x重叠

关键点：如何理解所寻找路径的安全性？即如果存在着重叠区间一定可以找到。

红黑树的扩张—区间树

- 算法的正确性
  - 关键说明：如果存在重叠区间，则一定会找到。
  - 定理14.2
    - case 1：若算法从x搜索到左子树，则左子树中包含一个与i重叠的区间或在x的右子树中没有与i重叠的区间；
    - case 2：若从x搜索到右子树时，则在左子树中不会有与i重叠的区间
  - 证明：先证case2，然后再证case1
    - case 2：走x右分支条件是

      left[x]=nil or max[left[x]]<low[i]

      若左子树为空，则左子树不含有与i重叠的区间；

      若左子树非空，由于有max[left[x]]<low[i]

      ∵ max[left[x]]是左子树中最大高点

      ∴ 左子树中的区间不可能与i重叠

红黑树的扩张—区间树

- case 1：

若左子树包含与i重叠的区间，则走左分支正确；

若左子树无区间与i重叠，下证x的右子树中也无区间与i重叠。

∵ max[left[x]]≥low[i]

∴ 存在区间i′∈{x 左子树的区间}，使得

high[i′] = max[left[x]]≥low[i]

⇒ 只有i′和i重叠或i在i′前

又左子树无区间与i重叠 ∴只有i在i′前，即 high[i]<low[i′]

对∀区间i″∈{x右子树的区间}，由BST性质有 low[i′]≤low[i″]

由此 ⇒ high[i]<low[i″]， ∴ i和i″不重叠

谢谢！

Q & A