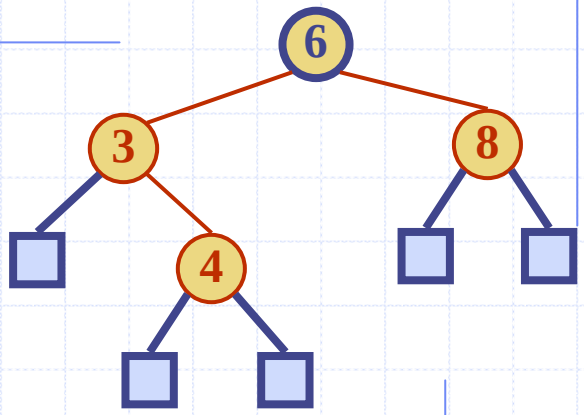
数据结构与算法
内存索引——红黑树



北京大学信息科学技术学院

张铭

http://db.pku.ecu.cn/mzhang/DS/

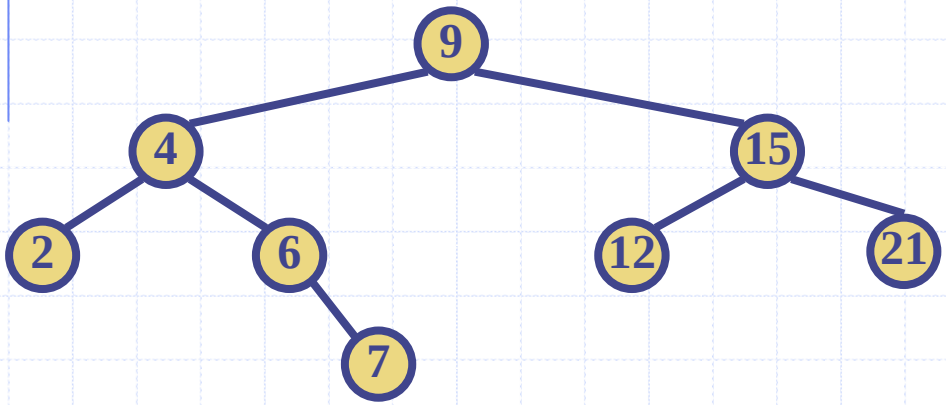
引子：索引的效率问题

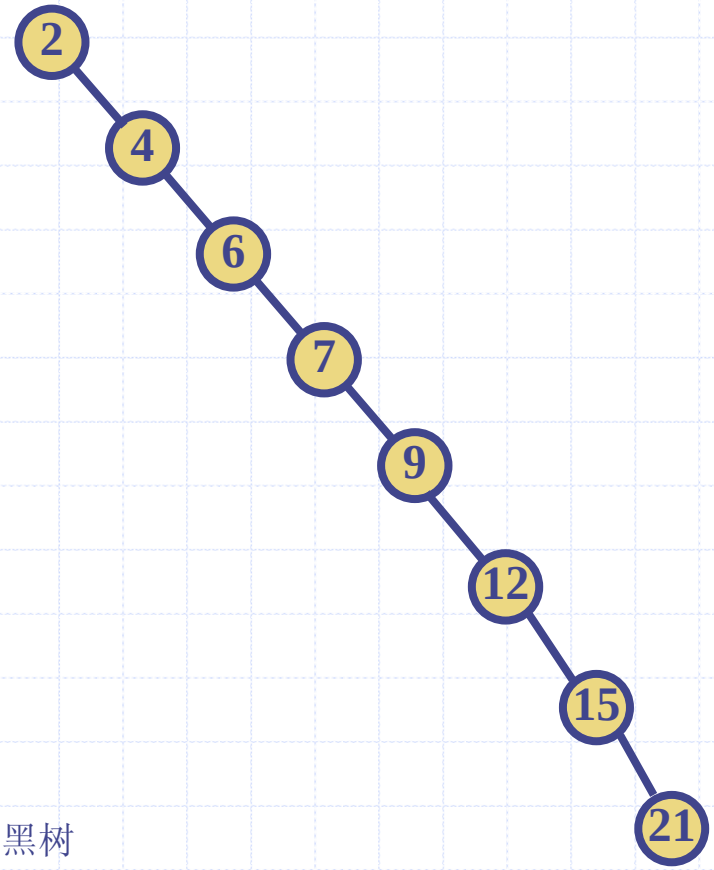
- **索引(indexing)**：把一个关键码与它对应的数据记录的位置相关联
  - (关键码，指针)对，即(key，pointer)
- 三类索引
  - 线性索引：有序数组、索引顺序文件
  - 树型索引：二叉搜索树**(BST)**、**B/B+**树 字符树**……**
  - 散列索引

红黑树之歌

BST的平衡问题

- 输入9,4,2,6,7,15,12,21
- 输入2,4, 6,7, 9, 12,15, 21





- 希望保持理想状况
- 插入、删除、查找时间代价为**O(log*n*)**

红黑树之歌

The Red-Black Tree Song

- I see a brand new node
- I want to paint it black.
-
- We need a balanced tree,
- we've got to paint it black.
-
- I want to find my key in log n time -- thats all,
- Rotating sub-trees 'round sure can be a ball.

The Red-Black Tree Song

- I see a brand new node
- I want to paint it black.
-
- Can't have a lot of red nodes,
- We must paint them black.
-
- Unfortunately, coding them can be a bitch.
- If we had half a brain to splay trees we would switch.

The Red-Black Tree Song

- I see a brand new node
- I want to paint it black.
-
- No time for AVL trees
- we must paint it BLACK.
-
- And if they're still confusing, you should have no fear.
- Because outside this class, of them you'll never hear.
-
- I wanna paint 'em BLACK. Paint nodes black. Again and again.

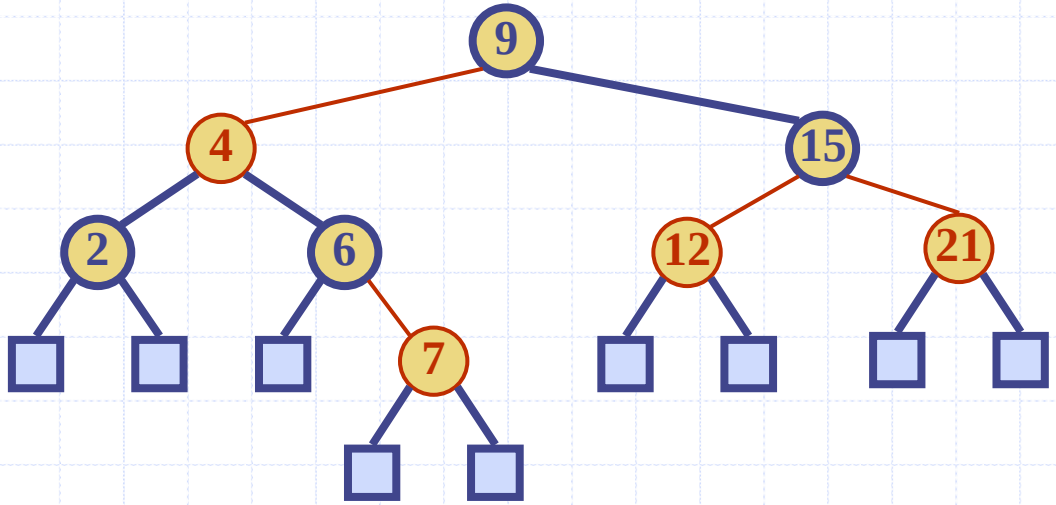
内容提要

- 红黑树定义
  - **red-black tree, 简称RB-tree**
- 红黑树高度
- 结点插入算法
- 结点删除算法

红黑树之歌

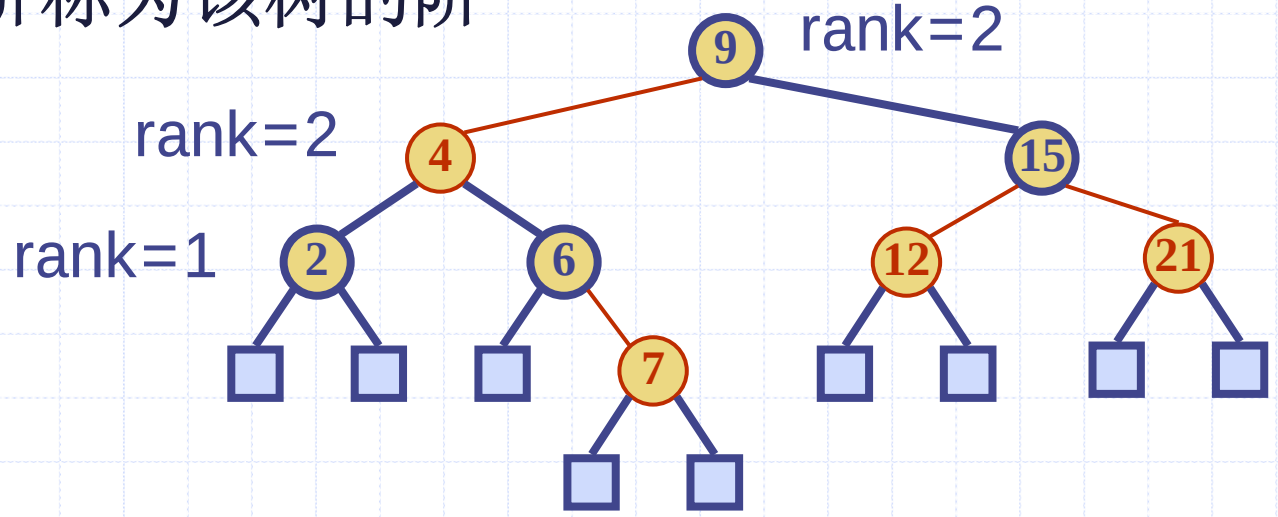
红黑树：平衡的扩充二叉搜索树

- 颜色特征：结点是“红色”或“黑色”；
- 根特征：根结点永远是“黑色”的；
- 外部特征：扩充外部叶结点都是空的“黑色”结点；
- 内部特征：“红色”结点的两个子结点都是“黑色”的
  - 不允许两个连续的红色结点
- 深度特征：任何结点到其子孙外部结点的每条简单路径都包含相同数目的“黑色”结点

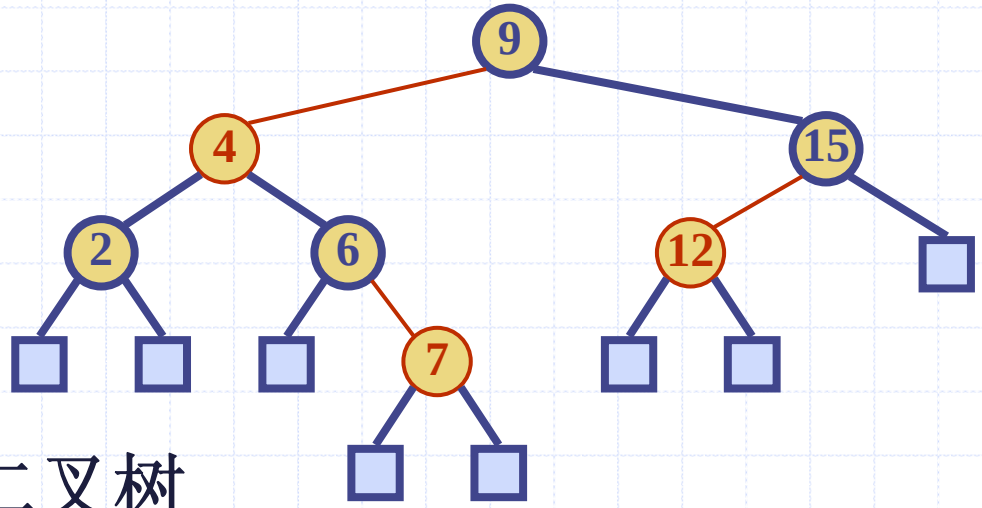


红黑树的阶

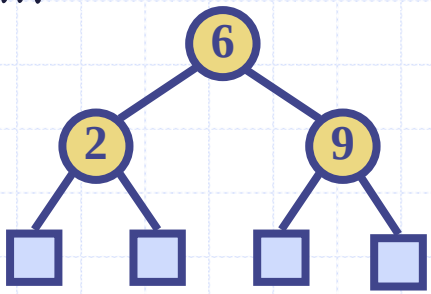
- 结点X的阶（**rank**，也称“黑色高度”）
  - 从该结点到外部结点的黑色结点数量
  - 不包括X结点本身，包括叶结点
- 外部结点的阶是零
- 根的阶称为该树的阶



红黑树的性质



- (1) 红黑树是满二叉树
  - 空叶结点也看作结点
- (2) 阶为k的红黑树路径长度
  - 从根到叶的简单路径长度 最短是k， 最长是2k
  - 即树高 最小是k+1 ， 最高是2k+1
- (3) 阶为k的红黑树的内部结点
  - 最少是一棵完全满二叉树
  - 内部结点数最少是2^k-1



红黑树的性质

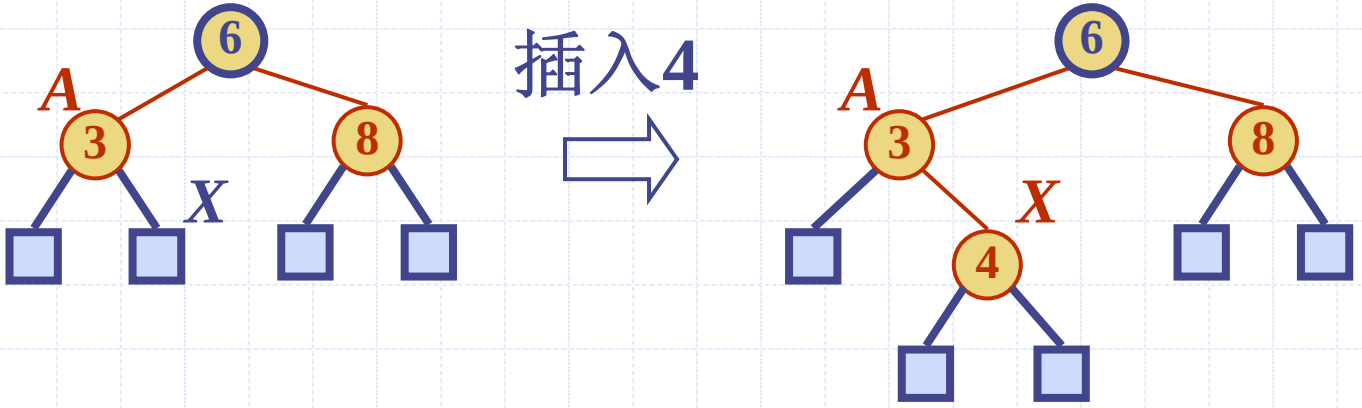
- **(4) n个内部结点的红黑树树高**
  最大是$2\log_2(n+1)+1$

证明：

- 设红黑树的阶为k，设红黑树的树高是h。
- 由性质（2）得$h <= 2k+1$
  - 则 $k >= (h-1)/2$
- 由性质（3）得$n >= 2^k - 1$
  - 即 $n >= 2^{(h-1)/2} - 1$
- 可得出 $h <= 2\log_2(n+1)+1$

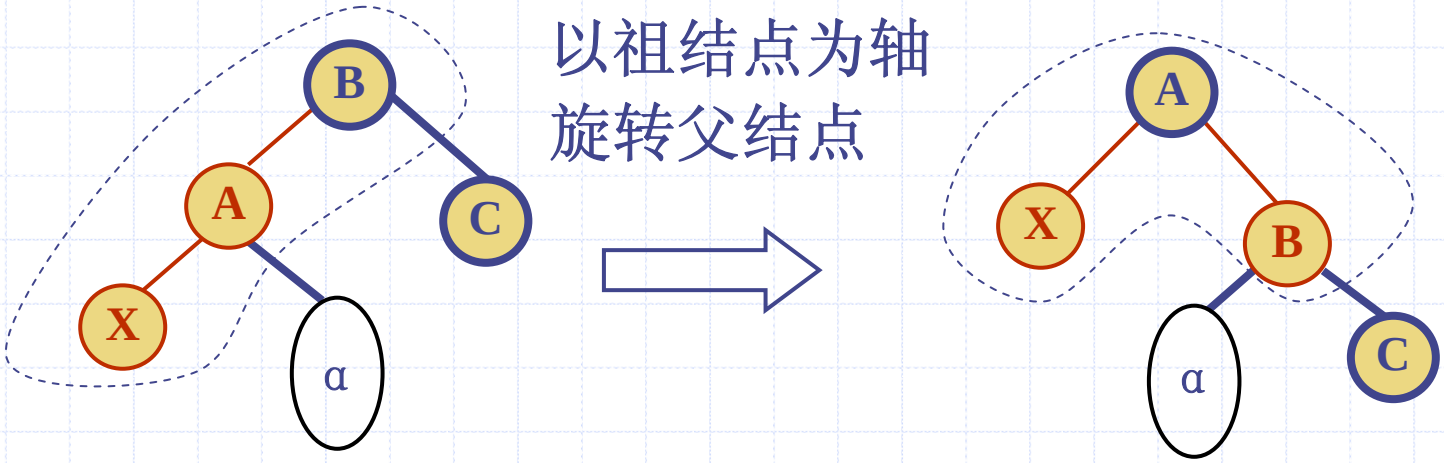
插入算法

- 先调用BST的插入算法
  - 把新记录着色为红色
  - 若父结点是黑色，则算法结束
- 否则，双红调整



插入算法调整1：重构

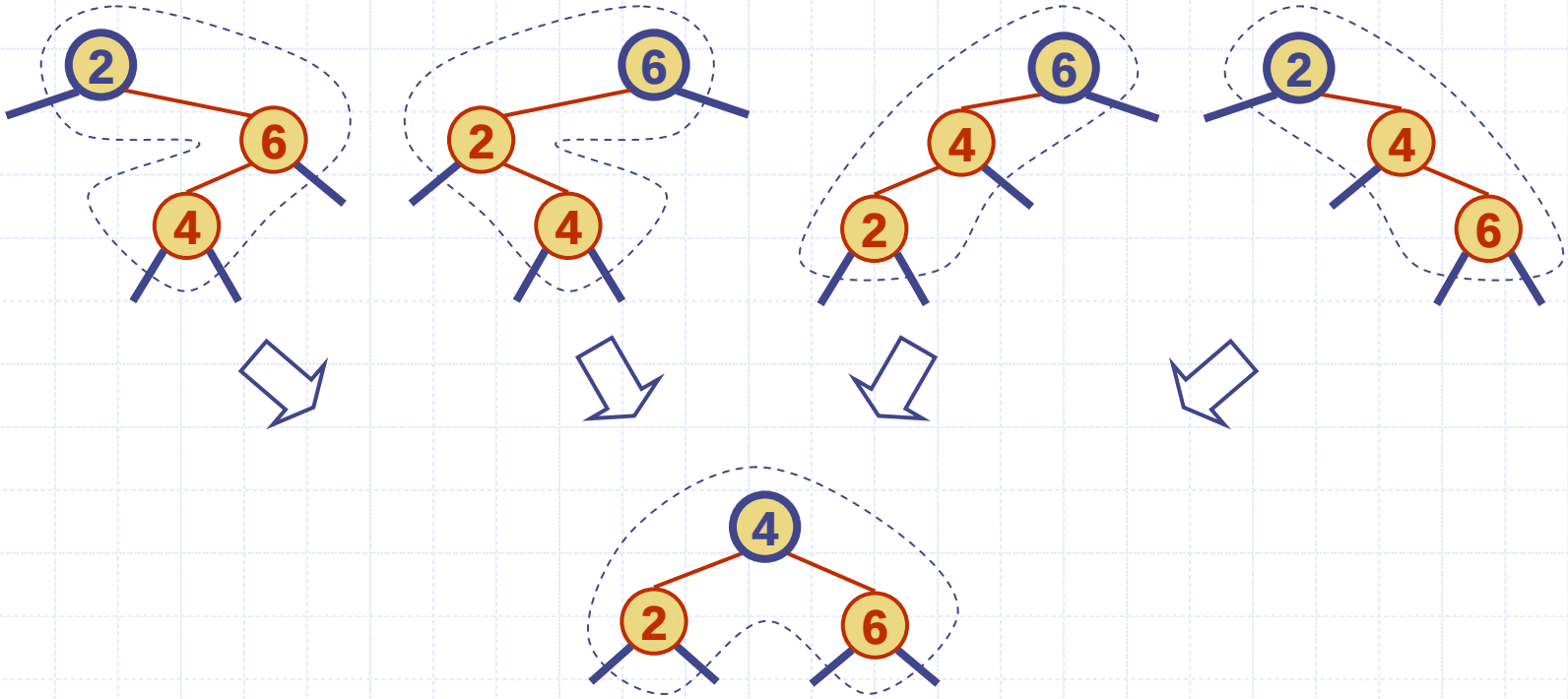
- 情况1：新增结点X的叔父结点是黑色



- 每个结点的阶都保持原值，调整完成

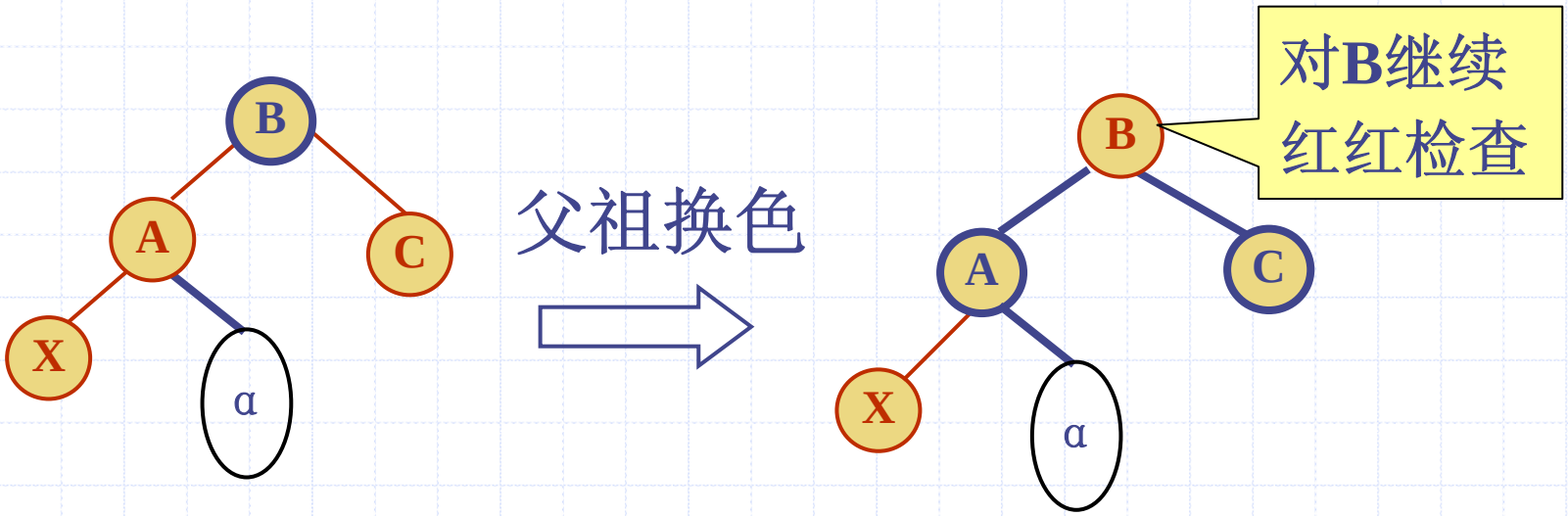
4种形式的结构调整

- 原则：保持BST的中序性质



插入算法调整2：换色

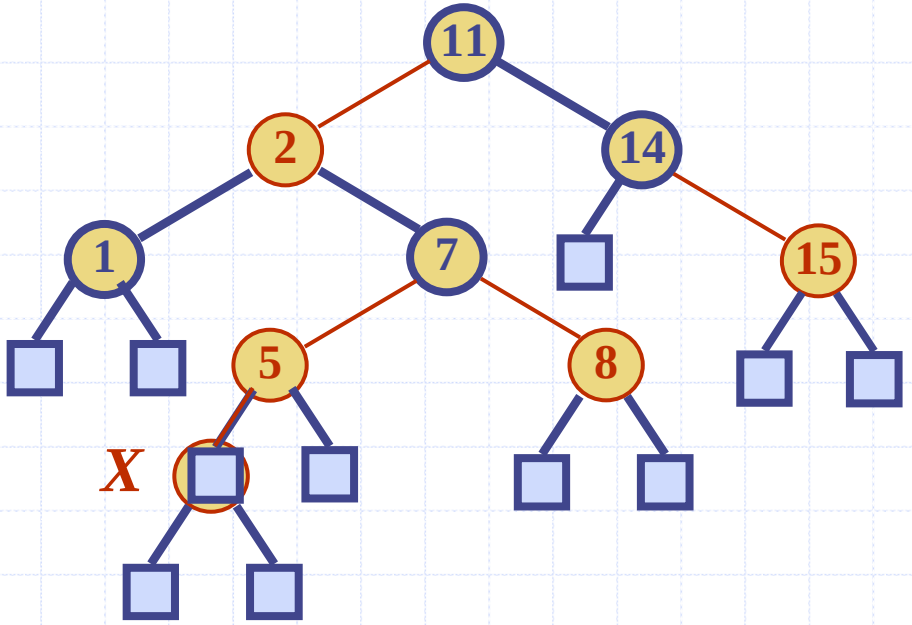
- 情况2：新增结点X的叔父结点也是红色



- 需要继续检查平衡

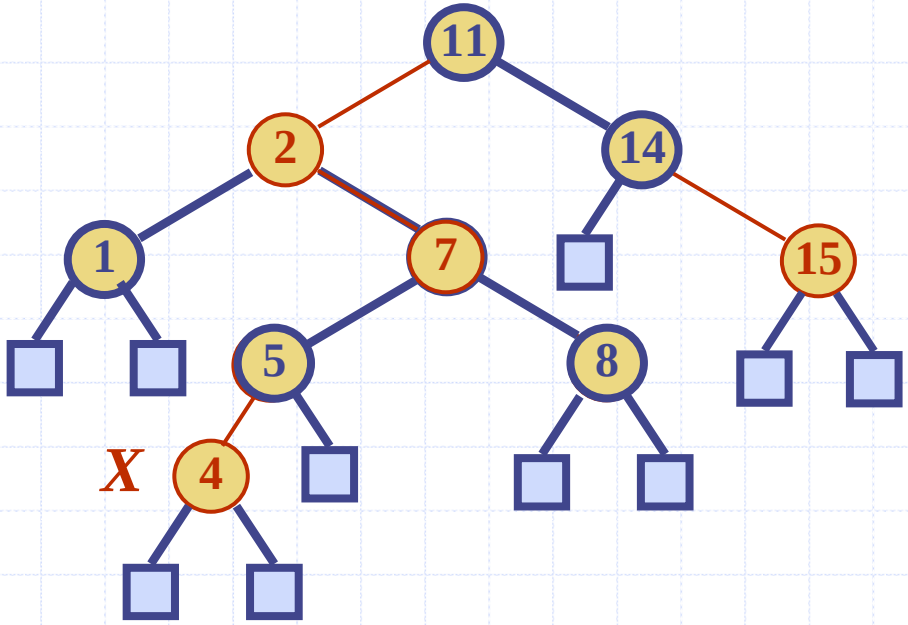
插入4

- 情况2红红冲突
  - 父和叔父也是红
- 父祖换色



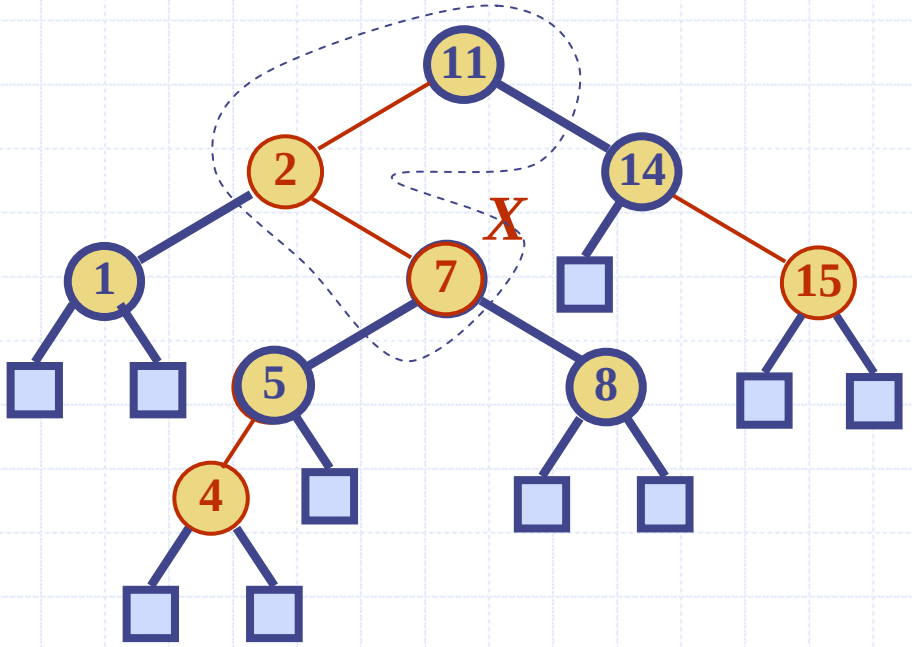
插入4

- 情况2红红冲突
  - 父和叔父也是红
- 父祖换色



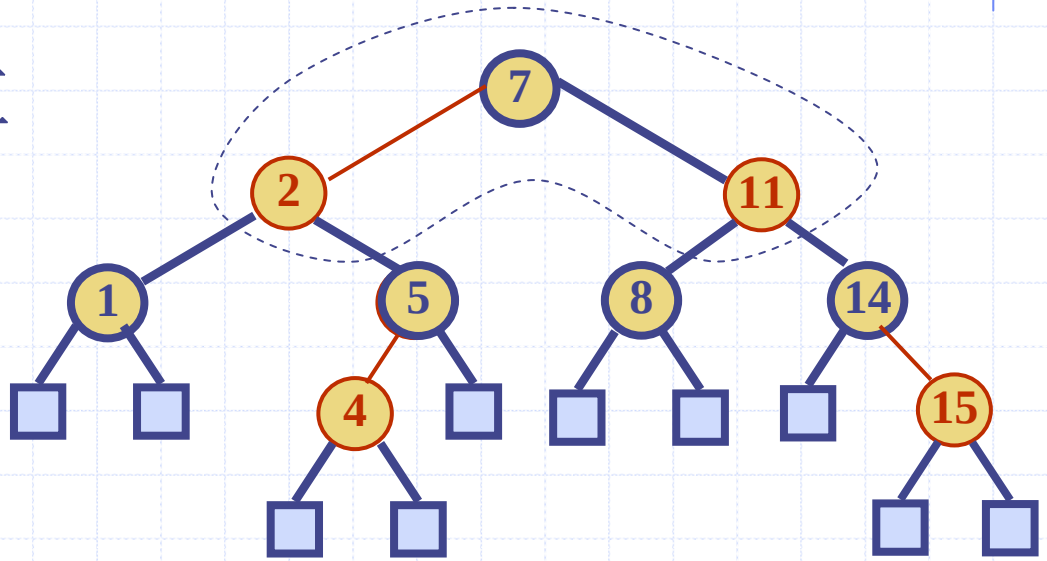
插入4

- 情况2红红冲突
  - 父和叔父也是红
- 父祖换色
- 情况1红红冲突
  - 叔父是黑
- 重构



插入4

- 情况2红红冲突
  - 父和叔父也是红
- 父祖换色
- 情况1红红冲突
  - 叔父是黑
- 重构



插入算法和复杂度分析

***insertItem*(*k*, *o*)**

1. **BST插入，插入新结点 *z*(*k*, *o*)**

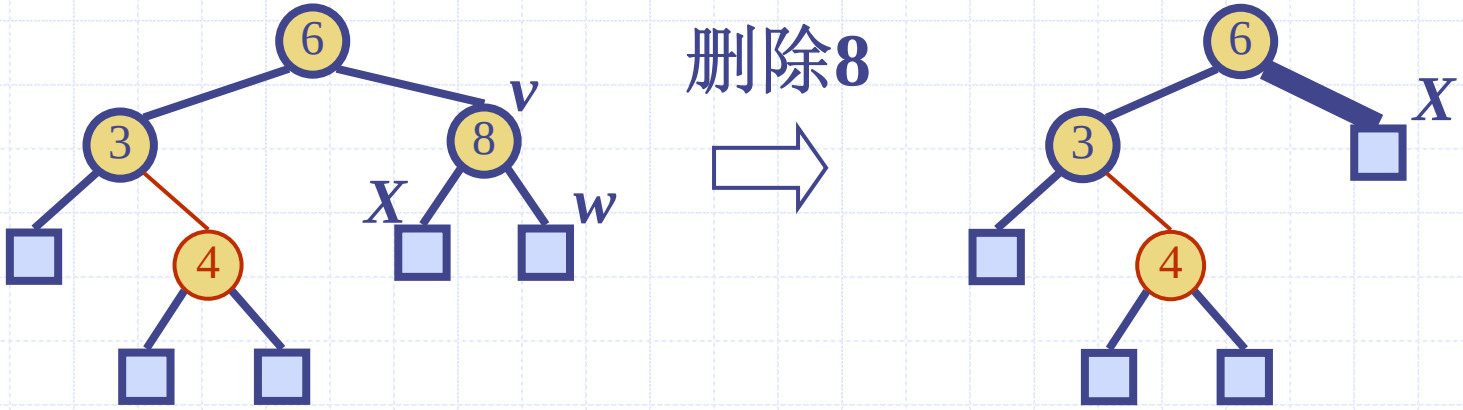
2. **把 *z* 标记为红色**

3. **while *doubleRed*(*z*)**
   if *isBlack*(*sibling*(*parent*(*z*)))
   ***z* ← *restructure*(*z*)**
   return
   else { *sibling*(*parent*(*z*) is red }
   ***z* ← *recolor*(*z*)**

- 红黑树高度为 $O(\log n)$
- 第1步时间代价为 $O(\log n)$
  - 因为访问 $O(\log n)$ 个结点
- 第2步时间代价为$O(1)$
- 第3步时间代价为 $O(\log n)$
  - 最多 $O(\log n)$ 重着色
    - 每次 $O(1)$
  - 最多 $O(1)$ 次 重构着色
- 红黑树结点插入时间代价为 $O(\log n)$

删除算法

- 先调用**BST**的删除算法
  - 待删除的结点有一个以上的外部空指针，则直接删除
  - 否则在右子树中找到其后继结点进行值交换（着色不变）删除
- v 是被删除的内结点，w 是被删外结点，X 是 w的兄弟
  - 如果 v 或者 X 是红色，则把 X 标记为黑色即可
  - 否则，X 需要标记为双黑，根据其兄弟结点 **C** 进行重构调整



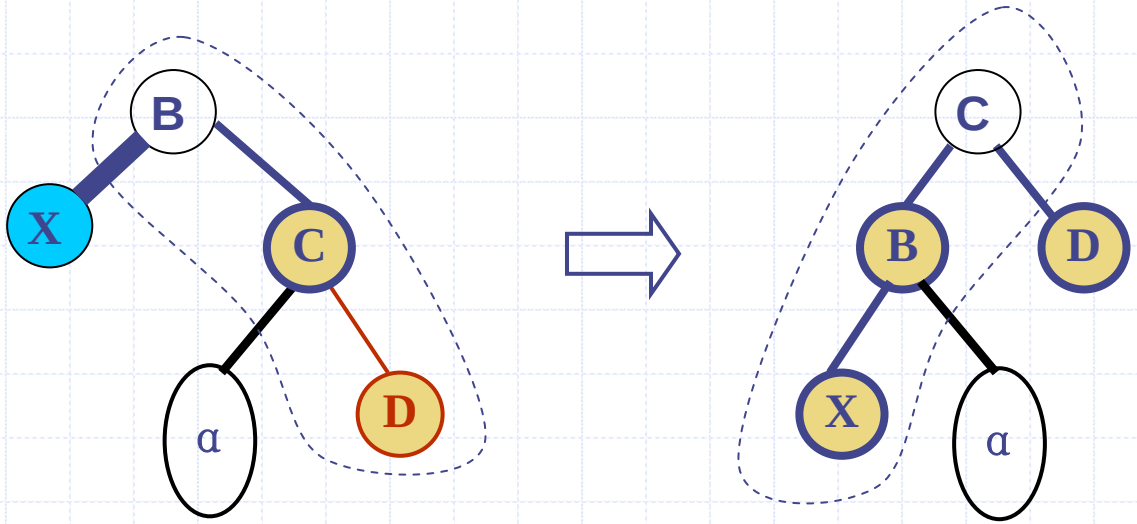
根据双黑 X 的兄弟 C 进行调整

假设X是左子结点（若X为右孩子，则对称）

- 情况1： C 是黑色，且子结点有红色
  - 重构，完成操作
- 情况2：C 是黑色，且有两个黑子结点
  - 换色
  - 父结点B原为红色，可能需要从B继续向上调整
- 情况3： C 是红色
  - 转换状态
  - C转为父结点，调整为情况1或2继续处理

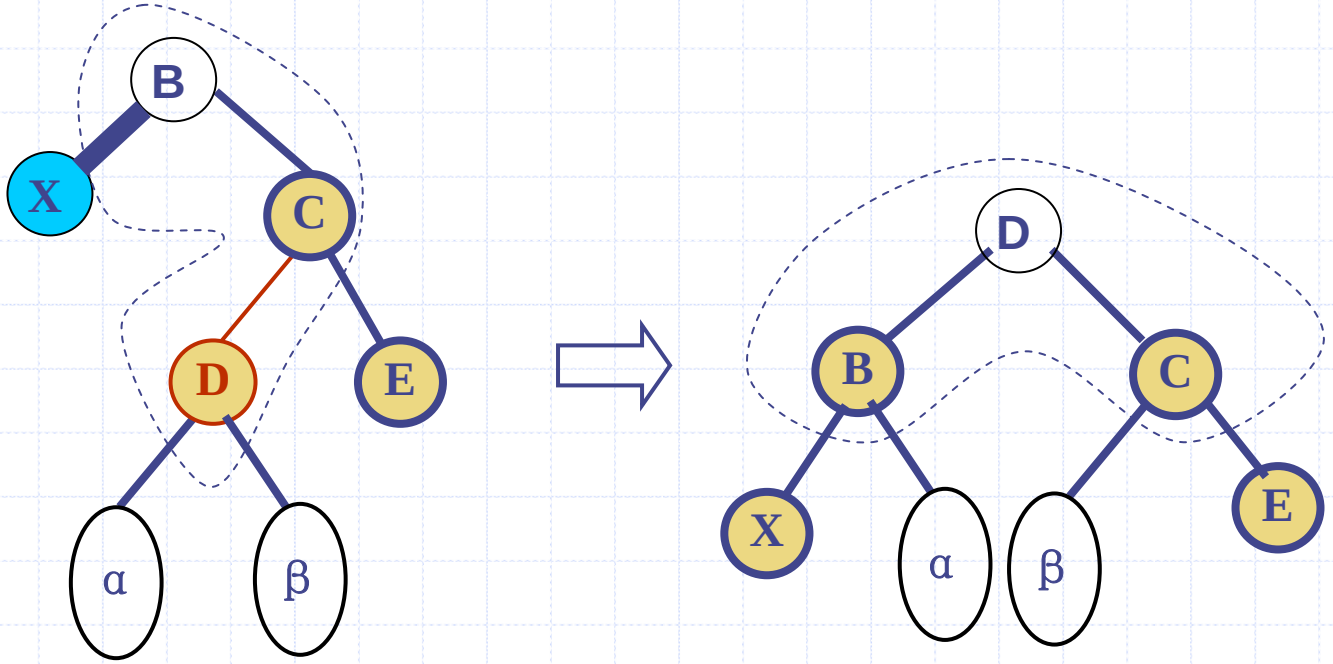
情况1(a)重构：侄子红结点八字

- 将兄弟结点C提上去
- C继承原父结点的颜色
- 然后把B着为黑色，D着为黑色，其他颜色不变即可



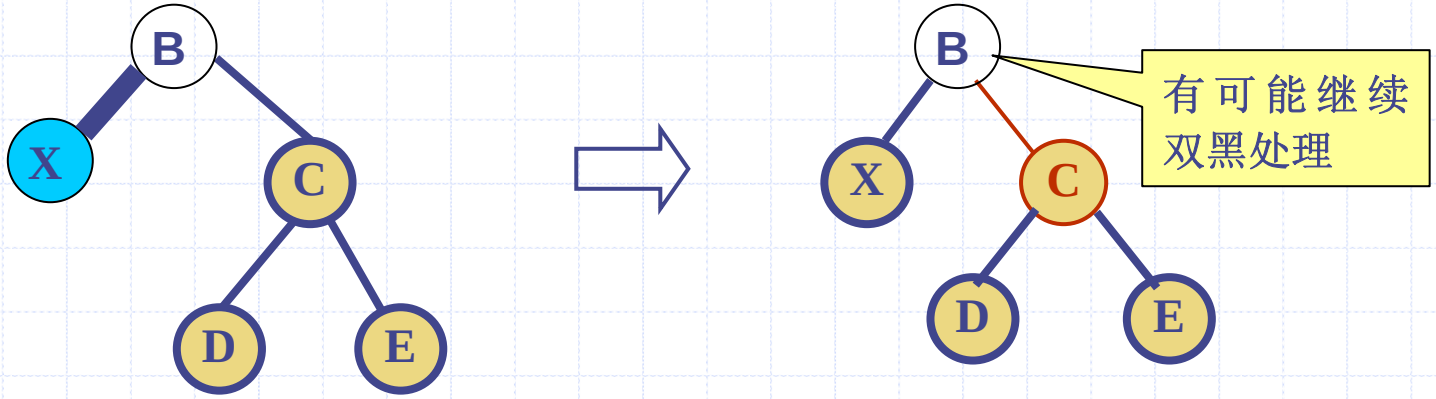
情况1(b)重构：侄子红结点同边顺

- 将D结点旋转为C结点的父结点，D继承原子根B的颜色，B着为黑色



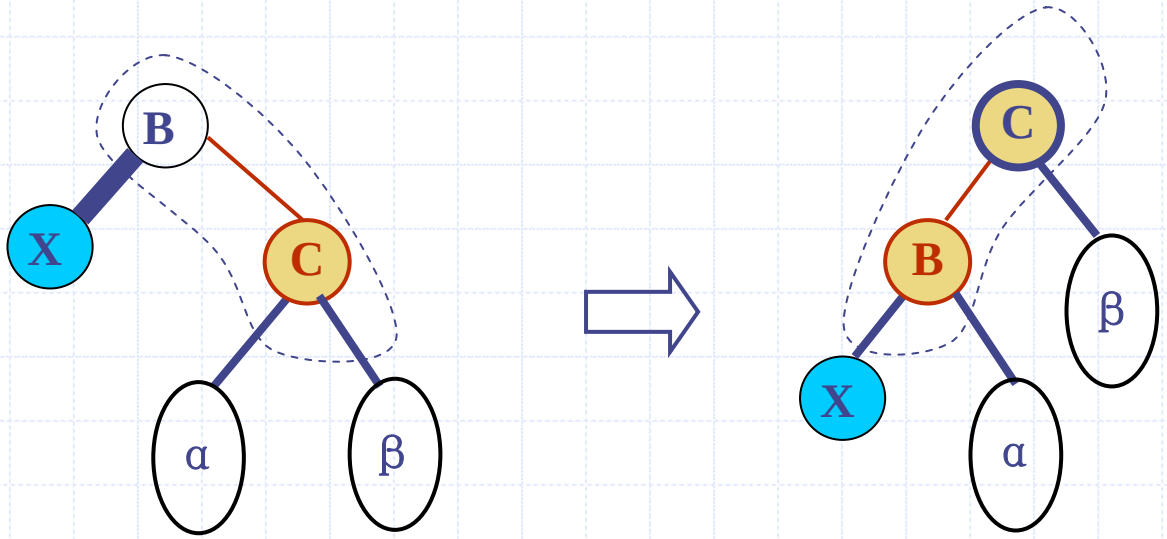
情况2：兄弟是黑色，且有两个黑子结点

- 把C着红色，B着黑色
- 如果B原为红色，则算法结束
- 否则，对B继续作“双黑”调整



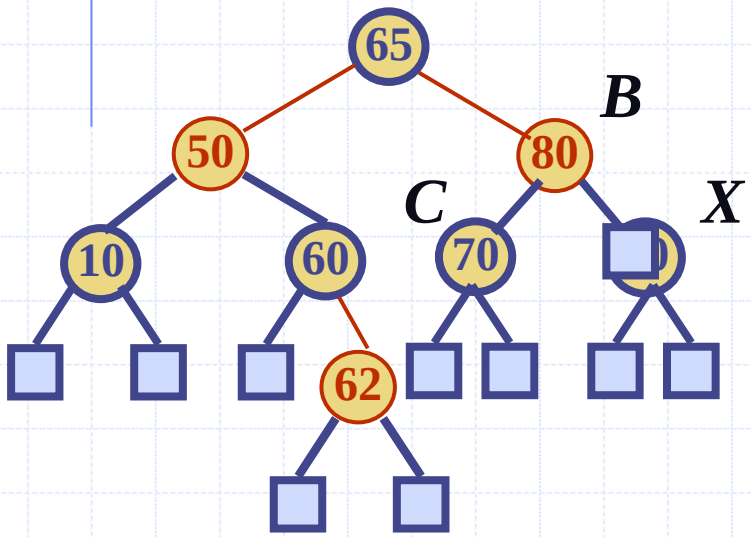
情况3：兄弟C 是红色

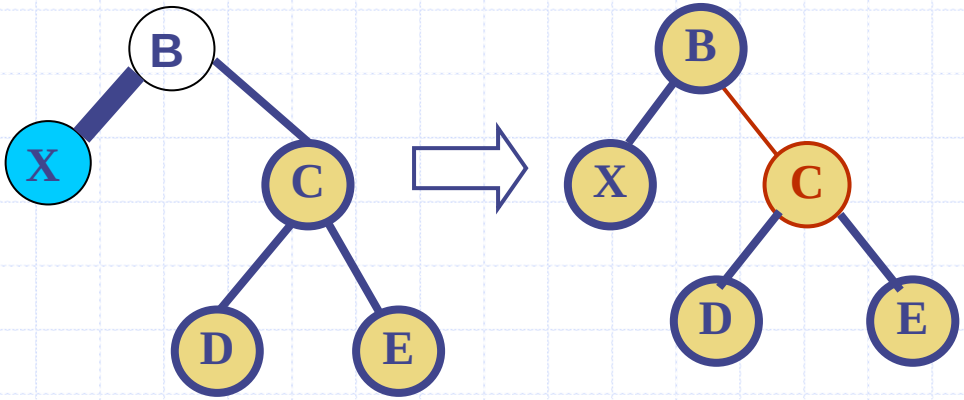
- 旋转
- X结点仍是“双黑”结点，转化为前面2种情况



删除90

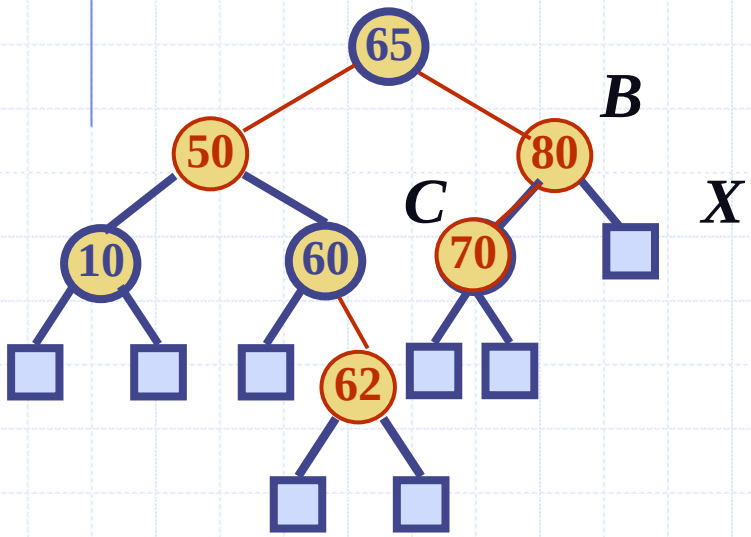
- 当前结点变为80的右黑叶结点
- C 是黑色, 且有两个黑色子结点：情况2

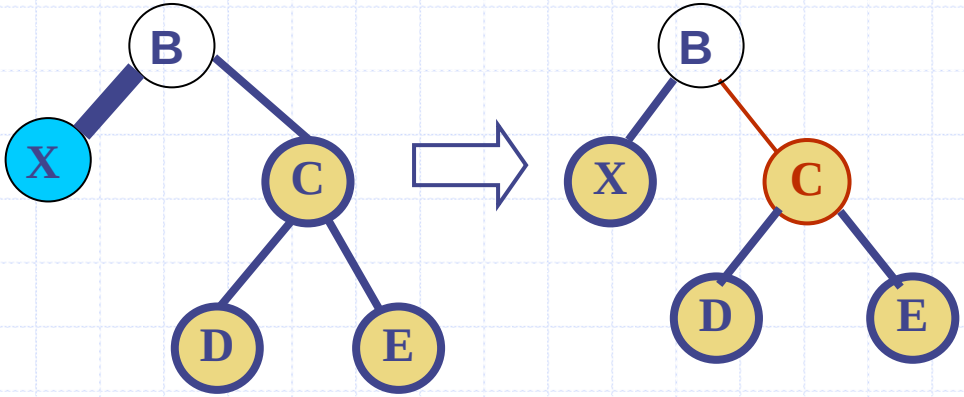




删除90

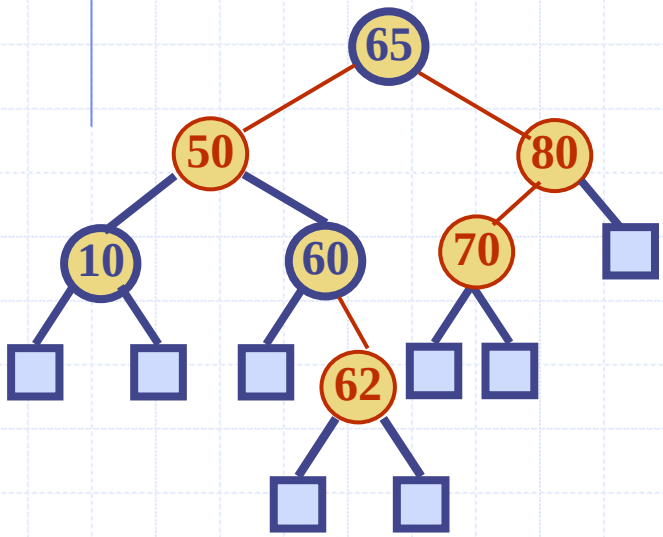
- 当前结点变为80的右黑叶结点
- C 是黑色, 且有两个黑色子结点：情况2换色





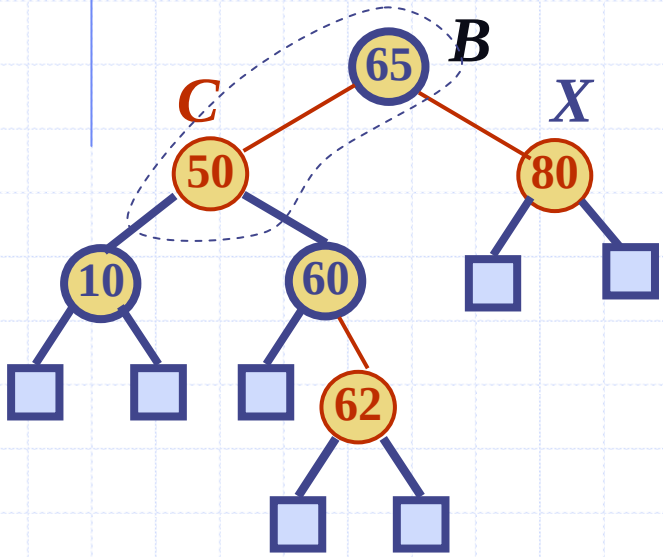
删除70

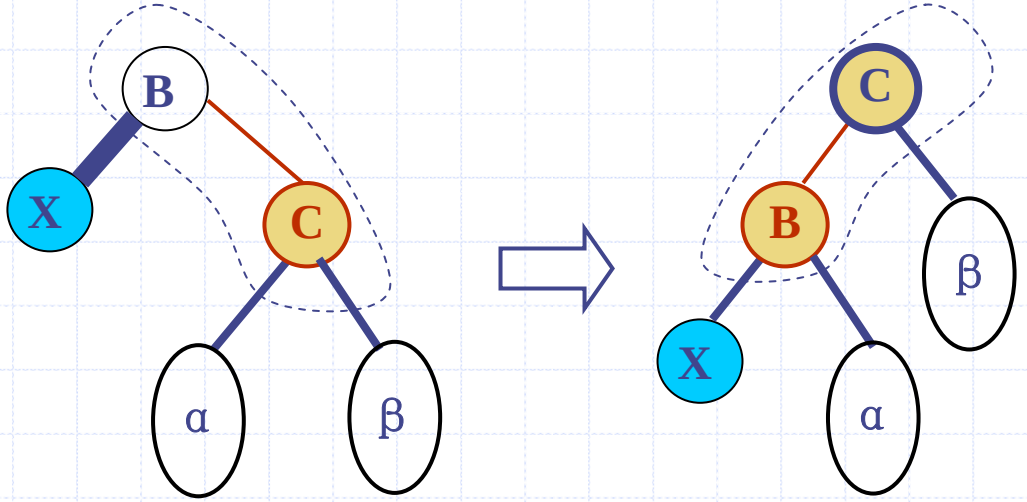
- 红结点，不要调整



删除80

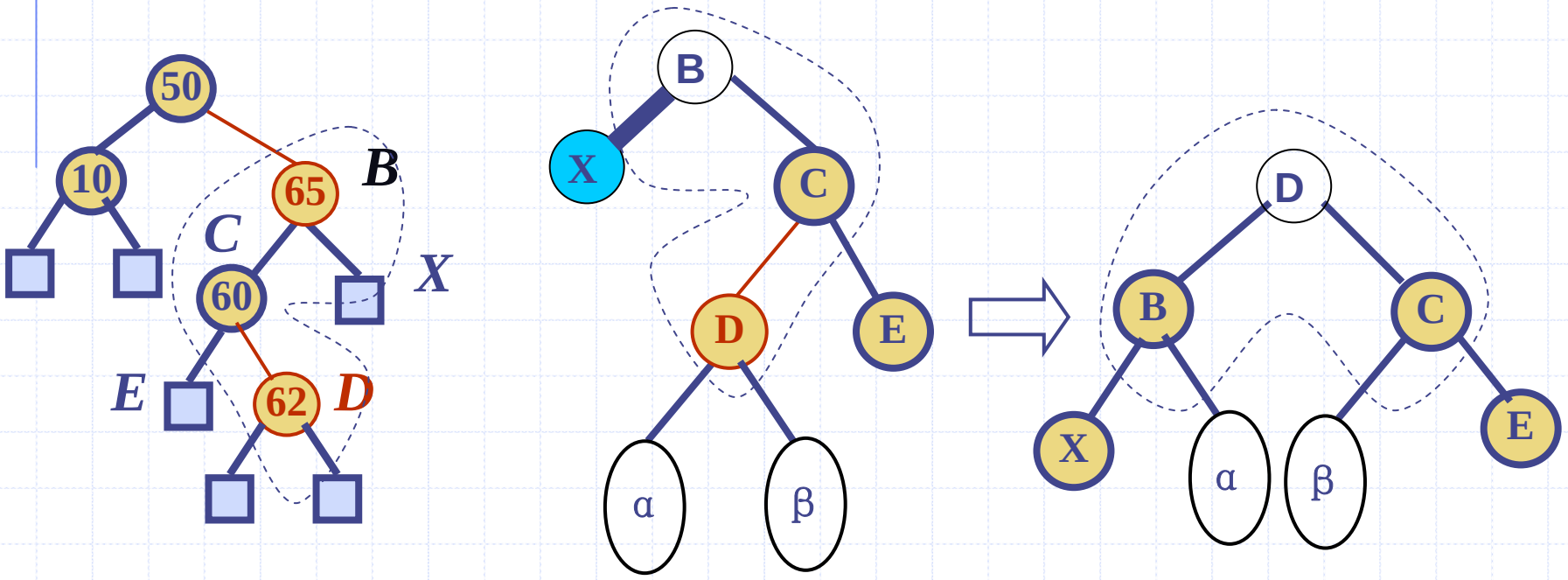
- 当前结点X变为65的右黑叶结点
- C 是红色：情况3状态转换





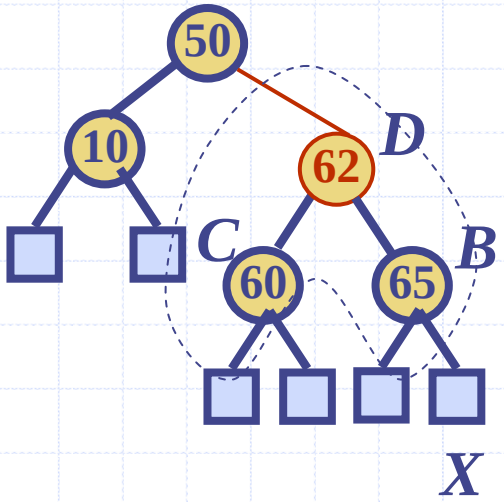
删除80（调整）

- C 是黑色，且左黑、右红：情况1(b)重构



删除80

- 完成调整

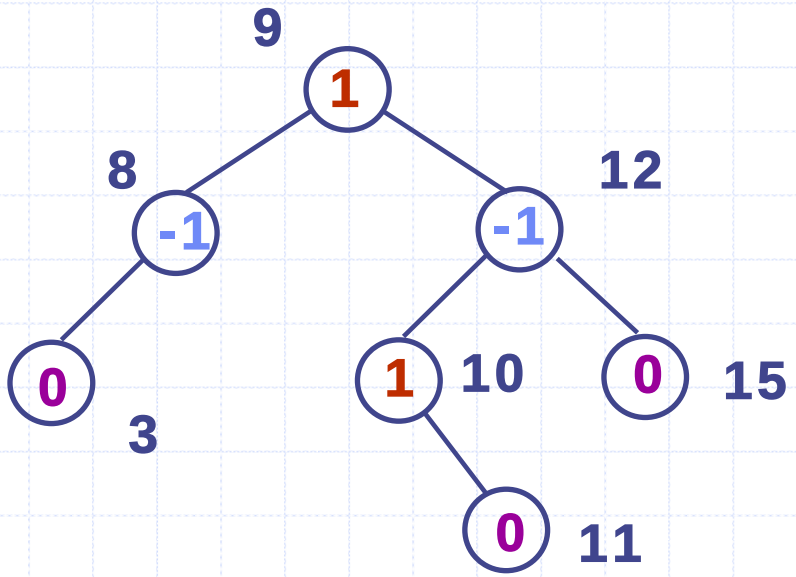


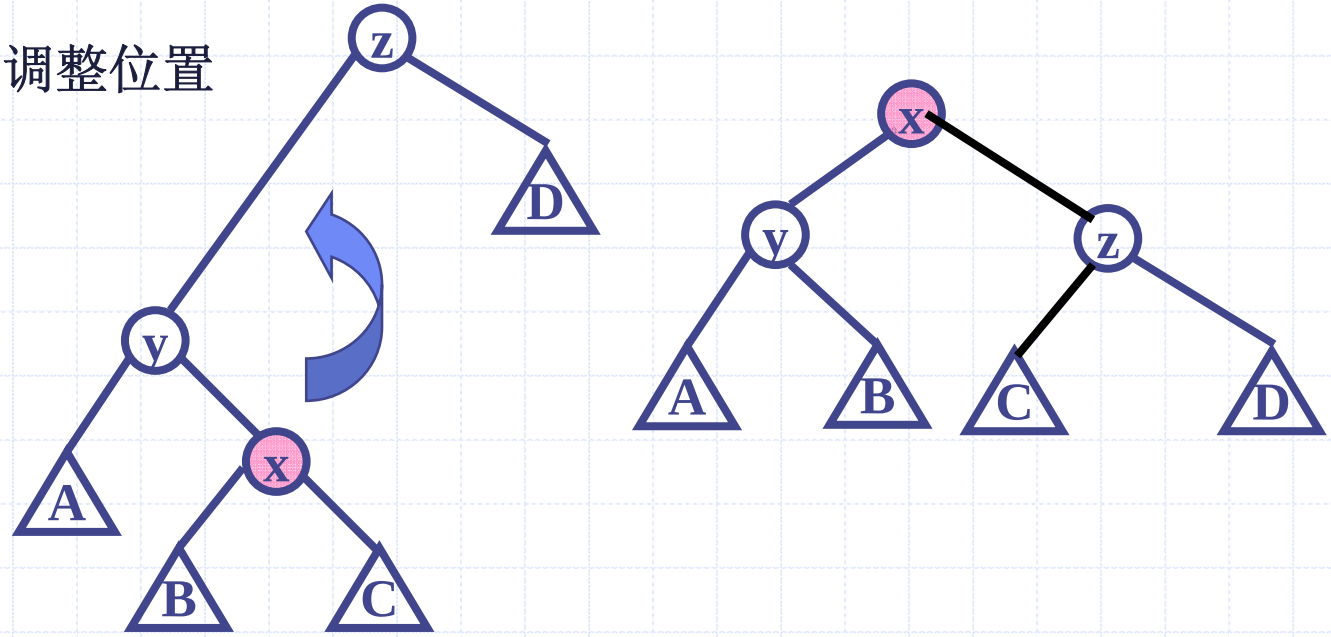
删除操作时间代价

- 其平均和最差检索$O(\log_2 n)$
  - 自底向根的方向调整
- 红黑树构造
- （数据，左指针，右指针，颜色，父指针）
- 自顶向下的递归插入/删除调整方法
  - （数据，左指针，右指针，颜色）
  - 非递归，记录回溯路径

其他BST索引

- AVL树
  - 平衡的BST
  - 左右子树高度差≤1
- 伸展树
  - 随检索而调整位置





红黑树是一种很好的索引结构

- **AVL**树要求完全平衡
- 伸展树与操作频率相关
- **RB-Tree**局部平衡
  - 统计性能好于**AVL**树
  - 且增删记录算法性能好、易实现
- **C++ STL**的**set**、**multiset**、**map**、**multimap**都应用了红黑树的变体

总结

- 红黑树高度平衡
- 结点插入算法
  - 树重构
  - 换色、调整
- 结点删除算法
  - 树重构
  - 重着色
  - 转换状态

谢谢！

北京大学信息学院
张铭

mzhang@db.pku.edu.cn