Advanced Data Structures I

5.1 Binary Search Trees

5.2 Red-Black Trees

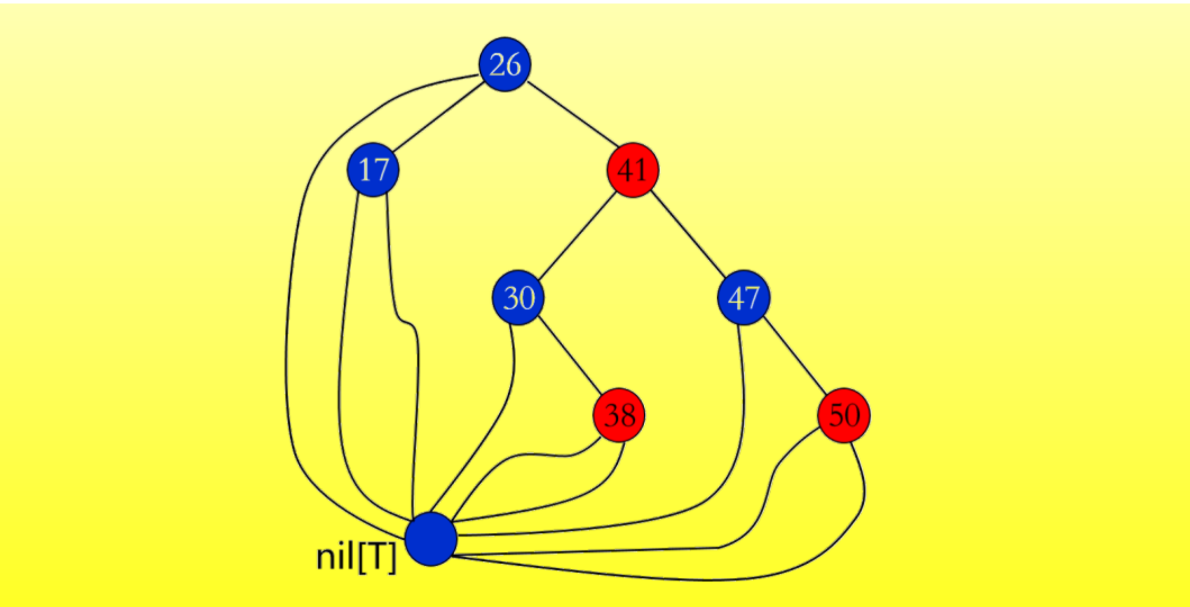
5.2.1 Overview

- 红黑树是二叉搜索树的变种，但是他是平衡树
- 红黑树的树高为稳定的$O(lgn)$,n是结点的数量
- 每个操作最坏时间复杂度为$O(lgn)$

Binary search tree + 1 bit per node:

- color域:red or black
- key，left，right，p，这些域都是从BST继承来的
- 所有的叶子结点的color是黑色的
- 用一个nil，表示所有的叶子结点 color[nil] = black
- root的parent也是nil

一个红黑树的例子



5.2.2 Red-Black 属性

1. 每个结点要么是红色的，要么是黑色的
2. 根节点是黑色的
3. 所有的leaf都是黑色的
    - 所有的real node 都有两个孩子结点
4. 如果一个结点是红色的，那它的孩子结点都是黑色的
    - 不能有连续两个结点是红色的
5. 对每个结点而言，从当前节点到它最底下的叶子节点包含相同数量的黑色结点

5.2.3 红黑树的高度

height of a node：

- number of edges in a longest path tp a leaf

black-height of a node x，bh（x）：

- 一条从x到leaf中黑色结点的数量

红黑树的black-height是根节点的黑高

- What is the minimum black-height of a node with height h? A height-h node has black-height $\geq h/2$
- Theorem: A red-black tree with n internal nodes has height $h \leq 2\,lg(n+1)$

证明：高度边界

- Prove: n-node RB tree has height $h \leq 2\,lg(n+1)$
- Claim: A subtree rooted at a node x contains at least $2^{bh(x)} - 1$ internal nodes
- Proof by induction on height h

- Base step: x has height 0 (i.e., NULL leaf node)
  1. So $bh(x) = 0$
  2. So subtree contains $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes (TRUE)
- Inductive step: x has positive height and 2 children
  1. Each child has black-height of $bh(x)$ or $bh(x) - 1$
  2. So the subtrees rooted at each child contain at least $2^{bh(x)-1} - 1$ internal nodes
  3. Thus subtree at x contains $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodes (TRUE)

- Thus at the root of the red-black tree: $n \geq 2^{bh(root)} - 1 \Rightarrow n \geq 2^{h/2} - 1 \Rightarrow h \leq 2\,lg(n+1)$
- Thus $h = O(lg\ n)$

5.2.4 Worst-Case Time

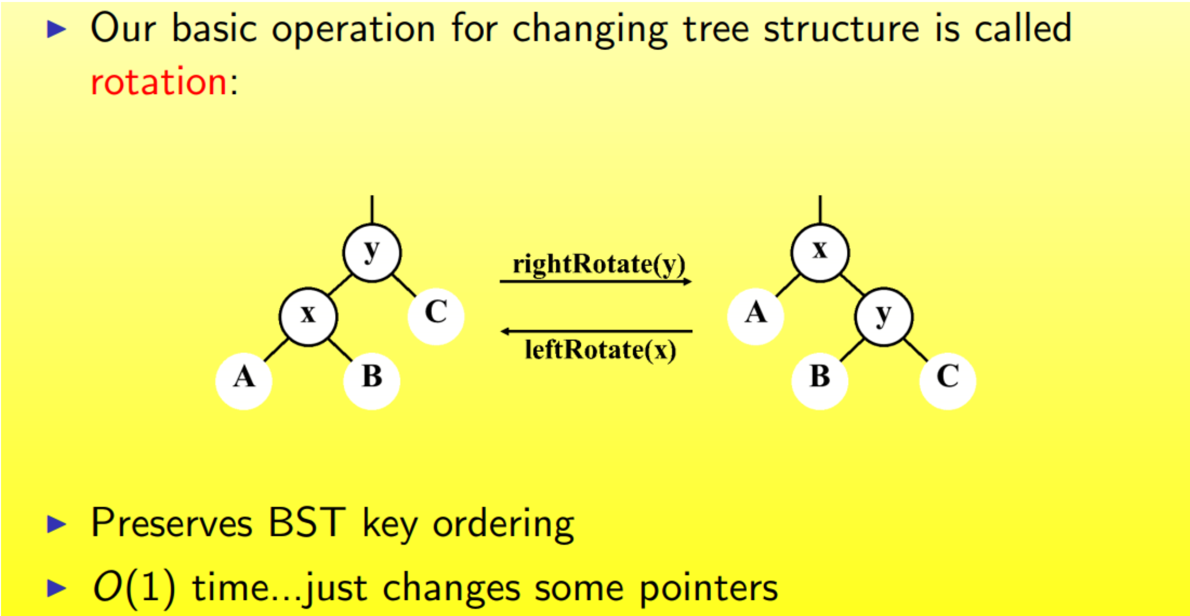
红黑树的树高为$O(lgn)$

以下这些方法都花费$O(lgn)$时间

- minimum()

- maximum()
- successor()
- predecessor()
- search()
- insert() and delete()
    - 需要特殊关注，因为它们修改了红黑树

旋转（rotation）



5.2.5 红黑树的insertion（）

the basic idea:

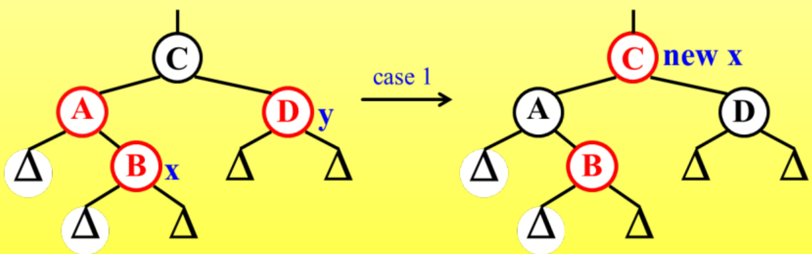
- 将结点x插入红黑树，颜色标为red
- 性质2可能被打破（如果x是根节点且为红色的话），如果如此，别的性质没有打破的情况下，把x涂成black
- 性质4可能被打破（父结点可能也是红色的），如果如此，调整颜色后上浮，直到可以调整好所有的位置
- 总时间可以是$O(lgn)$的

伪代码pseudocode I

```
RBInsert(T, x)
TreeInsert(T, x)
color[x] ← RED
while x ≠ root[T] and color[p[x]] = RED do
    if p[x] = left[p[p[x]]] then
        y ← right[p[p[x]]]
        if color[y] = RED then
            color[p[x]] ← BLACK
            color[y] ← BLACK
            color[p[p[x]]] ← RED
            x ← p[p[x]]
        else
            if x = right[p[x]] then
                x ← p[x]
                LeftRotate(x)
            color[p[x]] ← BLACK
            color[p[p[x]]] ← RED
            RightRotate p[p[x]]
    else
        same as above, but exchanging 'right' and 'left'
color[root[T]] ← BLACK
```

- Case 1:

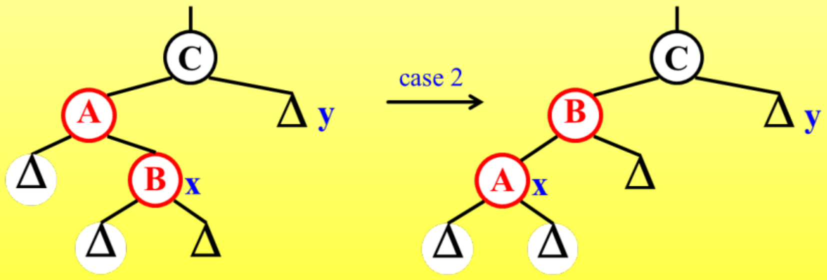
  - Case 1: uncle is red:
    In figures below, all △'s are equal-black-height subtrees



  - Change colors of some nodes, preserving r-b property 5: all downward paths have equal b.h. The while loop now continues with x's grandparent as the new x

- Case 2:

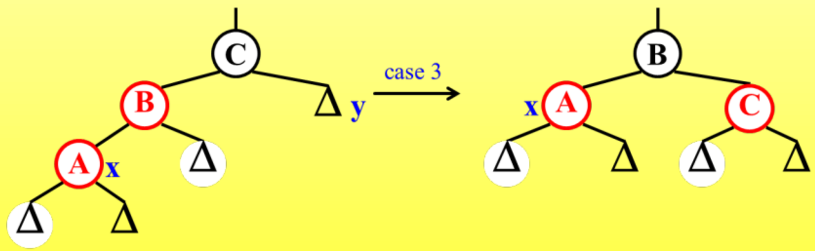
- Case 2: uncle is black
  Node x is a right child



- Set x=p[x]. Transform to case 3 via a left-rotation
- This preserves property 5: all downward paths contain same number of black nodes

- case 3:

- Case 3: uncle is black
  Node x is a left child



- Perform some color changes and do a right rotation
- Again, preserves property 5: all downward paths contain same number of black nodes

5.2.6 Delete

BST Delete

- Case 1: If vertex to be deleted is a leaf, just delete it
- Case 2: If vertex to be deleted has just one child, replace it with that child
- Case 3: If vertex to be deleted has two children, then swap it with its successor

Bottom-Up Deletion

- Do ordinary BST deletion. Eventually a "case 1" or "case 2" will be conducted. If deleted node, U, is a leaf, think of deletion as replacing with the NULL pointer, V. If U had one child, V, think of deletion as replacing U with V
- What can go wrong? If U is red? If U is black?

Fixing the problem

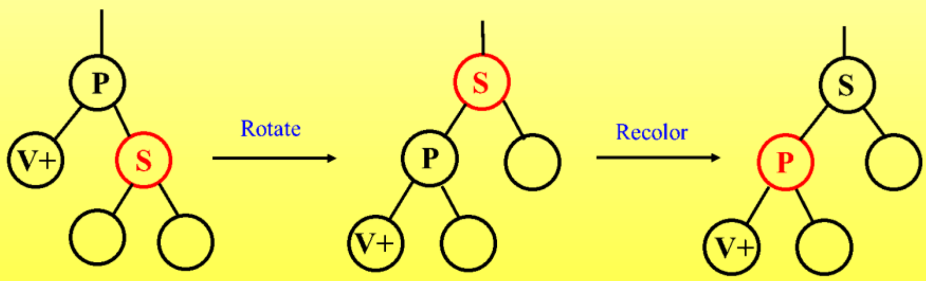
- Think of V as having an extra unit of blackness. This extra blackness must be absorbed into the tree (by a red node), or propagated up to the root and out of the tree
- There are four cases our examples and rules assume that V is a left child. There are symmetric cases for V as a right child

Terminology

- The node just deleted was U
- The node that replaces it is V, which has an extra unit of blackness
- The parent of V is P
- The sibling of V is S

RB Delete: Case 1

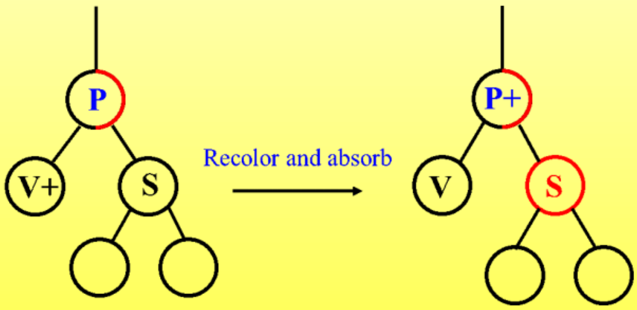
- Case 1: V's sibling, S, is red



- NOT a terminal case One of the other cases will now apply
- All other cases apply when S is black

RB Delete: Case 2

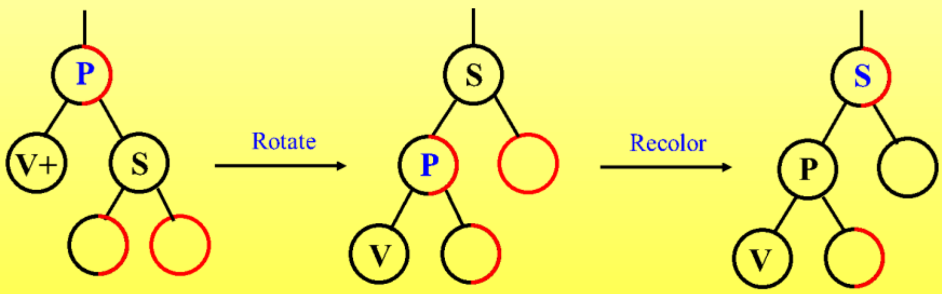
- Case 2: V's sibling, S, is black and has two black children



- Recolor S to be red
- P absorbs V's extra blackness:
  1. If P is red, we're done
  2. If P is black, it now has extra blackness and problem has been propagated up the tree

RB Delete: Case 3

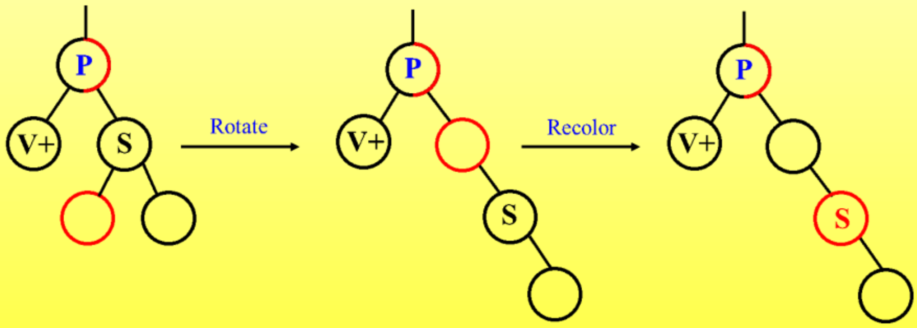
- Case 3: S is black, S's right child is red



- 1. Rotate S around P
  2. Swap colors of S and P, and color S's right child black
- This is the terminal case we're done

RB Delete: Case 4

- Case 4: S is black, S's right child is black and S's left child is red



- 1. Rotate S's left child around S
  2. Swap color of S and S's left child before rotation
  3. Now in case 3. e.g., V's sibling is black, which has a red right child.