Wellesley College CS231 Algorithms October 30, 1996
Handout #23

RED-BLACK TREES

Reading: CLR Chapters 14 and 15

Definition

A **red-black tree** (RBT) is a binary search tree that satisfies the following red-black properties:

RBT1. Every node (i.e. non-leaf) has a color that is either red or black.

RBT2. Every leaf (i.e. `nil`) is black.

RBT3. If a node is red, both children are black.

RBT4. Every path from a given node down to any descendant leaf contains the same number of black nodes. The number of black nodes on such a path (not including the initial node) is the **black-height** (`bh`) of the node.

RBT5. The root of the tree is black (not a CLR property, but should be).

Balance Property of Red-Black Trees

Consider a subtree with n nodes (non-leaves) rooted at any node x within a red-black tree. Then the following relationships hold:

$\text{height}(x) \quad \text{bh}(x) \quad \text{height}(x)/2$

$2^{\text{height}(x)} > n \quad 2^{\text{bh}(x)} - 1$

$2 \lg(n+1) \quad \text{height}(x) > \lg(n)$

The last relationship is the sense in which a red-black tree is balanced.

--**Dynamic Set Operations on Red-Black Trees**

Since a red-black tree is a binary search tree, the following operations work as on BSTs with no modifications:

Search, Minimum, Maximum, Predecessor, and Successor

Insert and Delete are similar to BST versions, but it may be necessary to update colors and tree structure to preserve RBT properties. It also helps to make some assumptions:

- every red-black tree T has a distinguished dummy header node header[T] whose left child is the actual tree and whose color is black.

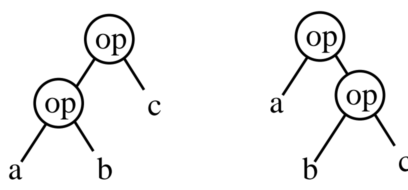
- every leaf is a data structure with parent and color attributes. (As noted in CLR, can get by with a single distinguished leaf nil[T] per red-black tree.)

Rotation

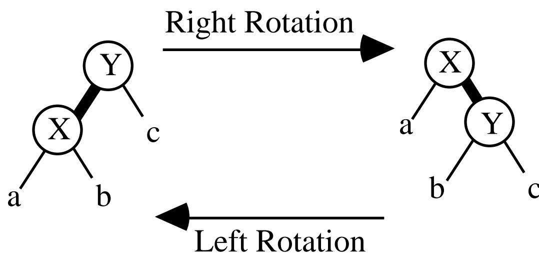
A binary operator *op* is said to be **associative** if for all a, b, and c

(a *op* b) *op* c = a *op* (b *op* c)

This relationship can be expresssed in tree form as:



In a BST, can view a tree node T with left subtree l and right subtree r as an operator that designates the sequence of labels of all nodes in T in an inorder traversal. From this perspective, nodes are associative binary operators:



Changing a BST from one of these forms to the other is **safe** in the sense that it preserves the binary search tree property . Moving from the left form to the right form is called a **right rotation** because it changes the thick edge to move down to the right. A **left rotation** changes the thick edge to move down towards the left. A rotation can be specified either by the root node and a direction (Y right or X left) or by the labels of the two nodes of the thick edge (XY).

Insertion into a Red-Black Tree

RB-Insert(T, x) has the following steps:

> *Step 1:* Use BST insertion to insert x into T, and color x red. This does not change black-height, so RBT1, RBT2, and RBT4 are preserved.
> *Step 2:* RBT4 may now be false in the case where both x and parent[x] are red (in which case parent[parent[x]] must be black). Reassert RBT3 by a sequence of the three types of moves described below.
> *Step 3:* Sometimes, the moves of Step 2 may leave the root red. In this case, reassert RBT5 by coloring it black.

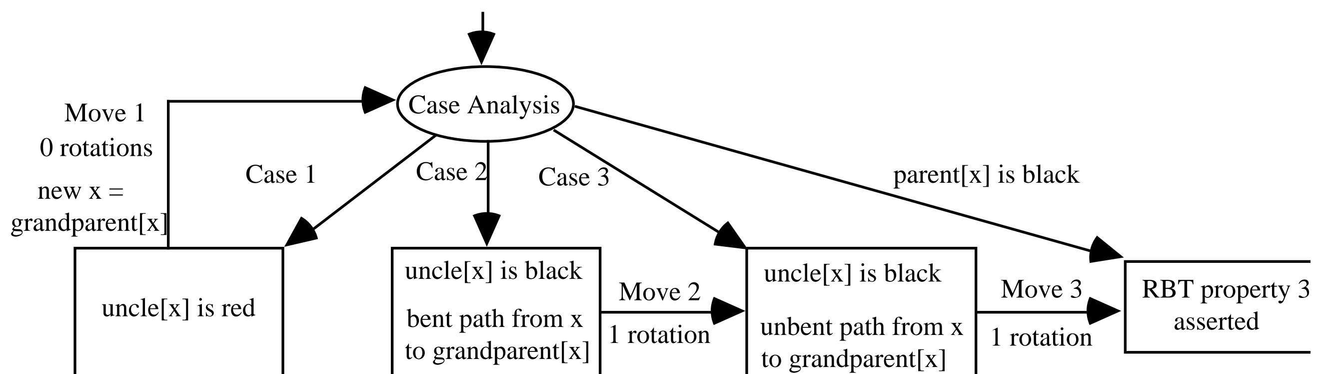
CLR characterize the moves for reasserting RBT3 in three cases. In each case, assume that x is red, parent[x] is red, and parent[parent[x]] is black. Each move preserves the black-height of the tree (convince yourself that this is true for each case)! We consider the cases in the opposite order of CLR (study these descriptions in conjunction with the pictures in CLR!):

> *Case 3:* The uncle of x is black and the path from x's grandparent to x consists of two edges with the same direction (left-left or right-right). Rotate about grandparent[x]/parent[x] and swap their colors. x's parent is the same node as before, but its color is now black; so RBT3 is reasserted in one move with one rotation.

> *Case 2:* The uncle of x is black and the path from x's grandparent to x consists of two edges with different directions (left-right or right-left). Rotate about parent[x]/x. This does not remove the violation of RBT3, but does enable Case 3; so RBT3 will be reasserted in 2 moves with two rotations.

> *Case 1:* The uncle of x is red. Distribute the blackness of x's grandparent to x's parent and x's uncle, and make x's grandparent red. If x's great-grandparent is black, RBT3 is reasserted in one move with zero rotations. But if x's great-grandparent is red, we have only moved the violation two nodes up the tree, and must try more moves with the new x referring to the grandparent of the original x. In the worst case, Case 1 is repeated until x becomes the root.
> Since the tree has height (lg(n)), it can be repeated only (lg(n)) times.

The following diagram summarizes the possible flows between the cases.



Deletion from a Red-Black Tree

RB-Delete(T, x) has the following steps:

Step 1: Use BST deletion to delete x from T. When both of x's children are non-leaves, x is replaced by Successor(x); in this case, ensure that Successor(x) is colored with x's color. RBT1, RBT2, RBT3, are maintained.

Step 2: Let y refer to the fringe node that is spliced out as part of Step 1 (this is either x itself or Successor(x)). If y was red, RBT4 is preserved and we are done. But if y was black, RBT4 is now false. In this case, reassert RBT4 by transferring the blackness of y to the child z of y that replaced y. If z is red, recolor it to black, and we are done. But if z was previously black, it is now "doubly-black". (Note: RBT5 can only be violated if y is a (necessarily black) root with one red child z and one (necessarily black) leaf child. In this case, the root is replaced by z, and transferring the blackness of y to z reasserts RBT5.)

Step 3: Propagate double-black node up the tree by a sequence of the four types of moves described below until the extra black is absorbed or the root of the tree is reached (in which case the extra black can be removed without affecting the tree's black-height).

CLR characterize the moves for reasserting RBT4 in four cases. In each case, assume that z is initially doubly-black. Each move preserves the black-height of the tree (convince yourself that this is true for each case)! We consider the cases in the opposite order from CLR. (Study these descriptions in conjunction with the pictures from CLR. Note: what I call z, CLR calls x.)

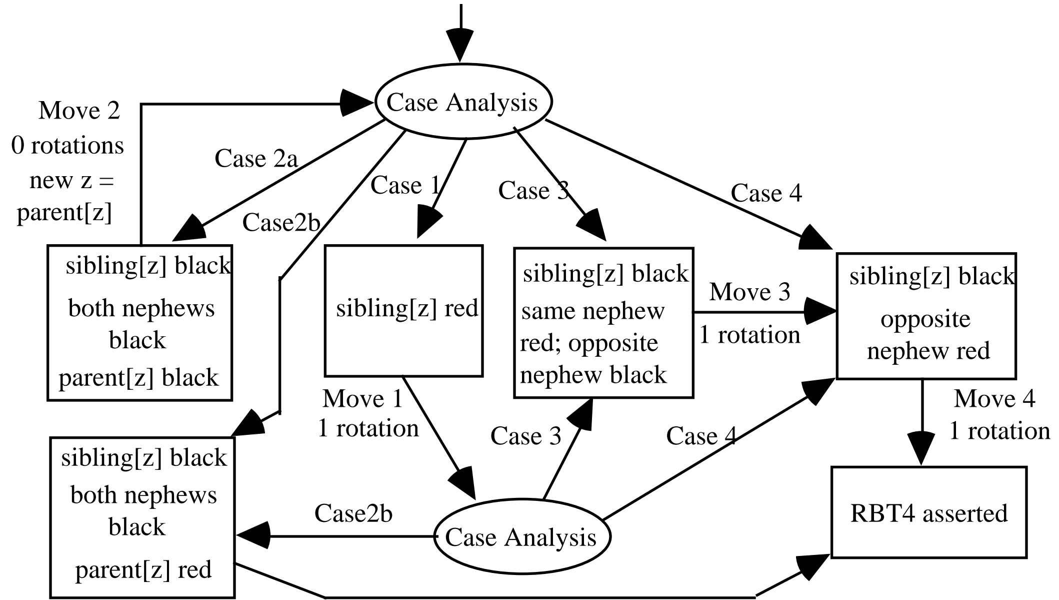
Case 4: z's sibling is black and its "opposite" nephew is red. RBT4 is reasserted by a rotation about parent[z]/sibling[z] and a recoloring.

Case 3: z's sibling is black, its "opposite" nephew is black, and its "same" nephew is red. A rotation about sibling[z]/same-nephew[z] and a recoloring leads to Case 4. So RBT4 is reasserted in two moves with two rotations.

Case 2: z's sibling is black, and both nephews are black. Merge the extra black of z and the black of sibling[z] and move this blackness to parent[z] (leaving sibling[z] red). If parent[z] was initially red, it is now black, and the extra black has been reabsorbed in one move with zero rotations. But if parent[z] was originally black, it is now doubly-black and we must repeat the doubly-black removal process with parent[z] as the new z. In the worst case, Case2 is encountered at each node on the way to the root for a total of (lg(n)) moves.

Case 1: z's sibling is red, implying that z' parent and two nephews are necessarily black (by RBT3). Rotate about parent[z]/sibling[z] and exchange the colors of these two nodes. After this rotation, z's sibling is black, and one of Case2, Case3, or Case4 applies.

The following state diagram summarizes doubly-black removal. Note that RBT4 can be reasserted in (lg(n)) moves with at most 3 rotations.



Augmenting Red-Black Trees

Can often improve running time of additional operations on a data structure by caching extra information in the header node or data nodes of a data structure. Must insure that this information can be updated efficiently for other operations.

Example 1: Store the length of a linked or doubly linked list in a header node.

Example 2: Store a pointer to the maximum node in a sorted linked list.
(Why wouldn't it help for an unsorted linked list?)

Example 3: Store the size of every red-black subtree in the root of that subtree.

size[leaf] = 0
size[node] = 1 + size[left[node]] + size[right[node]]

Can use size field to:

- Determine size of tree in (1) worst-case time.
- Perform Select(T, k) (i.e. find the kth order statistic) in (lg(n)) worst-case time.
- Determine the rank of a given node x in (lg(n)) worst-case time.

Insert and Delete can update the size field efficiently (i.e., without changing the asymptotic running time of Insert and Delete):

Insert: In downward phase searching for insertion point, increment sizes by one. In upward "fix-up" phase, update sizes at each rotation.

Delete: After deleting node y, decrement sizes on path to root[T] by one. In upward "fix-up" phase, update sizes at each rotation.

In general, can efficienly augment every node x of a red-black tree with the a field that stores the result of any function f that depends only on key[x], f(left[x]), and f(right[x]). Such a field is:

- for Insert, only needs to be updated on path from insertion point to root.
- for Delete, only needs to be updated on path from deletion point to root.
- easy to update at every rotation.