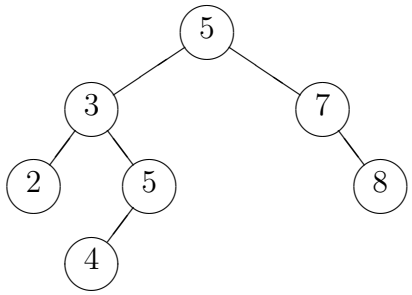
COMP 460 Lecture Notes

R. I. Greenberg

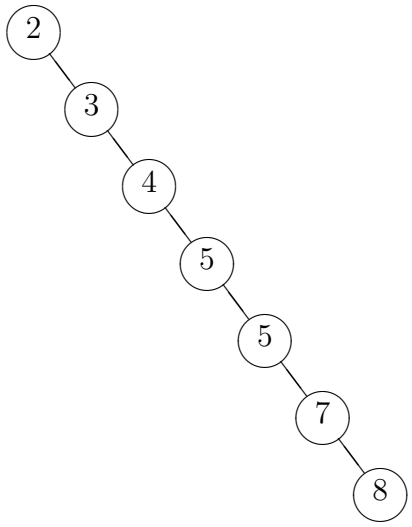
1 Red-Black Trees

A red-black tree is a special kind of *binary search tree*. Recall *BST property*: for any node x containing key k, all keys in left subtree of x are $\leq k$, and all keys in right subtree of x are $\geq k$.

Exs.:



or



A BST supports:

- Insert
- Search
- Delete
- Min
- Max
- Successor
- Predecessor

each in $O(h)$ time, where h is the height of the tree. They have good avg.-case behavior but bad worst-case.

Red-black trees keep the height of a tree on n nodes at $O(\lg n)$ to guarantee good worst-case behavior. (There are also other "balanced" search tree schemes, e.g., AVL trees, 2-3 trees, B-trees, splay trees.)

1.1 Basic Properties

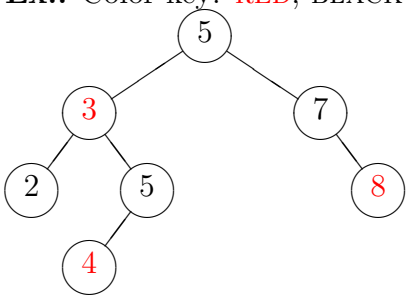
A red-black tree is a *binary search tree*, but think of null (NIL) pointers (to nonexistent children) as pointers to external nodes (leaves), while key-bearing nodes are referred to as internal nodes. (In fact, convenient for detailed coding to change all NILs to pointers to a special dummy object NIL(T), a *sentinel*.)

In addition, nodes have a new field for color (red or black).

red-black properties:

1. Every node is either RED or BLACK.
2. Root is black.
3. Every leaf (NIL) is black.
4. Red nodes have 2 black children.
5. For each node, every simple path down to a leaf contains the same no. of black nodes.

Ex.: Color key: RED, BLACK



Def: *black-height* of node x, denoted $bh(x)$, is no. of black nodes on a path down to a leaf. (Count leaf but not x.) Well-defined by property 5.

The key lemma: A red-black tree w. n internal nodes has height at most $2\lg(n+1)$.

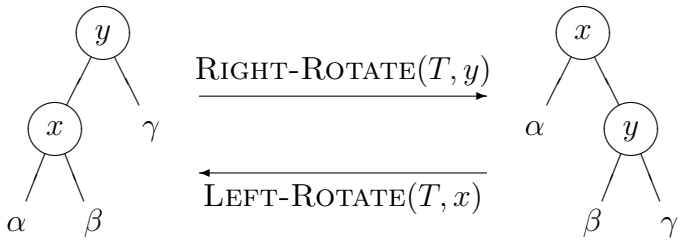
Proof: The longest root-to-leaf path is at most twice as long as the shortest by properties 1, and 4, and 5. So if the height is $> 2\lg(n+1)$, every root-to-leaf path has length $> \lg(n+1)$, and the number of leaves is $> n+1$, which corresponds to a number of internal nodes $> n$.

We now know that all the search tree queries run in $O(\lg n)$ time on a red-black tree.

But we need more work for INSERT and DELETE in order to maintain the red-black properties when we perform them.

1.2 Rotations

An important type of manipulation used during INSERT and DELETE:



Note: Preserves inorder key ordering.

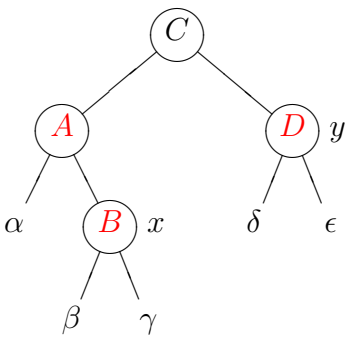
1.3 Insertion

Use regular BST TREE-INSERT to insert new node x, and color it RED. May violate only "red has black children" propertyand do it only btwn. x and its parent.

No problem if parent of x is BLACK. Otherwise, 6 cases; we'll look at the 3 arising when $left[p[p[x]]] = p[x]$:

Color key:
RED
BLACK
All Greek letters are subtrees w. a black root.

- Case 1: x's uncle y is RED:



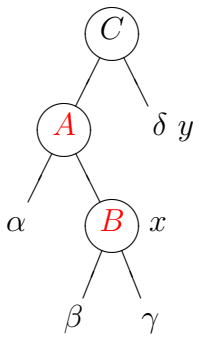
Flip colors of A, C, and D. (x initially inserted as a leaf, but we push violation up the tree, so may need to consider a picture like this with the Greek letters being more than just a leaf.)

(Same thing if x is a left child.)

  - Does not affect "consistent black-height" prop. (5).

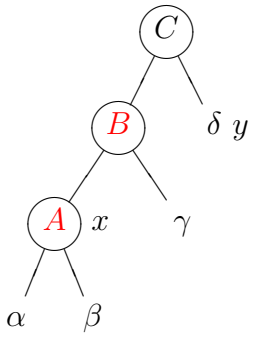
– Eliminates red-red violation btwn. A and B. At worst one violation remains btwn. C and its parent. Step up the tree.

- Case 2: x's uncle y is BLACK and x is a right child:

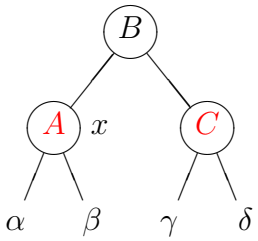


Do a left rotation at A to get the following, which is also case 3.

- Case 3: x's uncle y is BLACK and x is a left child:



Do a right rotation at C and flip the colors of B and C:



Here no black-heights change, and there is no violation of the "red nodes have black children" prop. (4).

Considering the 3 cases together: Either fix all problems immediately, or push red-red violation up the tree. If get to root, just color root BLACK. (In fact, always blacken root for convenience.)

Time:
$O(\lg n)$ for TREE-INSERT
$O(\lg n)$ for fix-up by at most walking up one path to root, doing constant-time operations along the way.

1.4 Deletion

Start essentially as in regular BST TREE-DELETE. This causes the splicing out of a node y. The node spliced out is either the actual node to be deleted or, if it has two children, its

successor. (In the case where the children of y are both NIL, we can think of the sentinel object NIL(T) as y's single child.)

(In the 3rd edition of the text, the above procedure is slightly modified so that one will actually delete the node requested for deletion and not cause any pointers from other program components into the tree to become stale.)

If y red, no effect on r-b props.

If y black, call RB-DELETE-FIXUP to restore "consistent black-height" prop. (5). Let x be the sole child of the spliced out black y. We want to give x an extra black. Do it if x red (and hence no problem w. "red has black children" prop. (4). Otherwise, push the extra black up the tree to a point where it can be fixed by rotations and recolorings, or, if it reaches the root, just remove it. There are 8 cases; the book shows the 4 in which x is a left child.

The time is again $O(\lg n)$ for ordinary delete and walking up one path for fixup.