Red-Black Trees
Motivation

• BST – basic operations take $O(h)$
• Worst case – no better than a list
• Red-Black trees guarantee a balanced tree resulting in a time $O(lg n)$
• Definition – Red-Black tree is a BST with one extra bit – its color (Red/Black)
• Use color to constrain how nodes are placed so that no path is more than twice as long as another – approx. balances the tree
Red-Black trees

• Empty children - NIL
• A BST a red-black tree if
  ‒ Every node is red or black
  ‒ Root is black
  ‒ Every leaf (NIL) is black
  ‒ If a node is red, both its children are black
  ‒ For each node, all paths from root to descendant leaves have the same number of black nodes
Leaves and Sentinel node

- Assume each leaf is an explicit black NIL node
- Alternatively, one black sentinel set to NIL could be used for the entire tree nil[T]
Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL’s have black-height 0. (b) The same red-black tree but with each NIL replaced by the single sentinel nil[T], which is always black, and with black-heights omitted. The root’s parent is also the sentinel. (c) The same red-black tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the remainder of this chapter.
Red-Black trees

- **Black-height** \( bh(x) \) – Number of black nodes in any path from, but not including, a node \( x \) down to the leaf

**Lemma:** A red-black tree with \( n \) internal nodes has height at most \( 2\log(n+1) \)
--Uses the result that any subtree rooted at \( x \) has at least \( 2^{bh(x)}-1 \) nodes
Rotations

• TREE-INSERT and TREE-DELETE may violate the red-black property

• *Rotation* helps restore these properties
  – Left rotation
    Left rotate a node $x$, assumes the right child $y$ is not nil[T]. Makes the right child ($y$) the root and the root ($x$) the left child, and $y$’s left as $x$’s right child
  – Right rotation
**Figure 13.2** The rotation operations on a binary search tree. The operation \texttt{LEFT-ROTATE}(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation \texttt{RIGHT-ROTATE}(T, y). The letters \(\alpha, \beta,\) and \(\gamma\) represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in \(\alpha\) precede \(\text{key}[x]\), which precedes the keys in \(\beta\), which precede \(\text{key}[y]\), which precedes the keys in \(\gamma\).
LEFT-ROTATE($T$, $x$)

1. $y \leftarrow right[x]$  \quad \triangleright \text{Set } y.$
2. $right[x] \leftarrow left[y]$  \quad \triangleright \text{Turn } y\text{’s left subtree into } x\text{’s right subtree}.$
3. $p[left[y]] \leftarrow x$
4. $p[y] \leftarrow p[x]$  \quad \triangleright \text{Link } x\text{’s parent to } y.$
5. if $p[x] = nil[T]$
   6. then $root[T] \leftarrow y$
   7. else if $x = left[p[x]]$
      8. then $left[p[x]] \leftarrow y$
      9. else $right[p[x]] \leftarrow y$
10. $left[y] \leftarrow x$  \quad \triangleright \text{Put } x \text{ on } y\text{’s left}.$
11. $p[x] \leftarrow y$
**Figure 13.3** An example of how the procedure `LEFT-ROTATE(T, x)` modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.
**Insert**

\[\text{RB-INSERT}(T, z)\]

\[y = T.nil\]
\[x = T.root\]
\[\textbf{while } x \neq T.nil\]
\[\quad y = x\]
\[\quad \textbf{if } z.key < x.key\]
\[\quad \quad x = x.left\]
\[\quad \textbf{else } x = x.right\]
\[z.p = y\]
\[\textbf{if } y == T.nil\]
\[\quad T.root = z\]
\[\textbf{elseif } z.key < y.key\]
\[\quad y.left = z\]
\[\textbf{else } y.right = z\]
\[z.left = T.nil\]
\[z.right = T.nil\]
\[z.color = \text{RED}\]
\[\text{RB-INSERT-FIXUP}(T, z)\]
RB-INSERT-FIXUP

• 6 cases, 3 symmetric to other 3 based on based on whether \(p[z]\) is left or right child of \(p[p[z]]\)
• If z’s uncle (y) is red – Case 1
  else Case 2, 3
CASE 1: z’s uncle y is RED

Figure 13.5 Case 1 of the procedure RB-INSERT. Property 4 is violated, since z and its parent p[z] are both red. The same action is taken whether (a) z is a right child or (b) z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward paths from a node to a leaf have the same number of blacks. The while loop continues with node z’s grandparent p[p[z]] as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.
Figure 13.6  Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 4 is violated in either case 2 or case 3 because $z$ and its parent $p[z]$ are both red. Each of the subtrees $\alpha$, $\beta$, $\gamma$, and $\delta$ has a black root ($\alpha$, $\beta$, and $\gamma$ from property 4, and $\delta$ because otherwise we would be in case 1), and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 5: all downward paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The while loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.
RB-INSERT-FIXUP(T, z)

while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK  // case 1
            y.color = BLACK   // case 1
            z.p.p.color = RED  // case 1
            z = z.p.p         // case 1
    else if z == z.p.right
        z = z.p           // case 2
        LEFT-ROTATE(T, z) // case 2
        z.p.color = BLACK // case 3
        z.p.p.color = RED // case 3
        RIGHT-ROTATE(T, z.p.p) // case 3
    else (same as then clause with “right” and “left” exchanged)
T.root.color = BLACK
Figure 13.4  The operation of RB-INSERT-FIXUP. (a) A node z after insertion. Since z and its parent p[z] are both red, a violation of property 4 occurs. Since z’s uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree shown in (b). Once again, z and its parent are both red, but z’s uncle y is black. Since z is the right child of p[z], case 2 can be applied. A left rotation is performed, and the tree that results is shown in (c). Now z is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in (d), which is a legal red-black tree.