Algorithms and Data Structures

Chapter 13

Catherine Durso

cdurso@cs.du.edu

Chapter 13

Our earlier study of binary search trees showed that this data structure supports SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $\Theta(h)$ time, where h is the height of the tree. Balanced binary search trees impose an additional condition on the tree that constrains h to have $h = \Theta(\lg n)$ growth.

Two common balanced binary search trees are red-black trees and AVL trees. Red-black trees use an additional bit, denoting red or black, at each node to maintain balance. AVL trees store the height of the node at each node, thus requiring more additional memory than red-black trees.

The subject here is red-black trees. The balance requirement will complicate insertion and deletion, but all the tree operations will have $\Theta(\lg n)$ worst-case performance.

Red-Black Tree Property

A red-black tree is a binary search tree with an additional data member, $color$, at each node. The $color$ value must be either red or $black$. We will consider the NIL pointers to be the leaves. The tree must satisfy the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Sentinel

In fact, the NIL's will be represented as pointers to the sentinel $T.nil$, which is black. This is for coding convenience.

Black Height

As a consequence of property 5 of red-black trees, the ***black height*** of a node x is well defined as the number of black nodes any path from x to a leaf, counting the leaf but not counting x.

The black height of a red-black tree is the black height of the root.

Balance

Lemma: A red-black tree with n internal (non-leaf) nodes has height at most $2 \lg (n+1)$.

Proof: By induction, the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
Base case:This is true if $bh(x) = 0$. Then x is a leaf and the subtree has $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

Inductive Step

Inductive step: If $bh(x) > 0$ then each child of x has black height $bh(x)$ or $bh(x) - 1$, depending on whether the child is red or black, respectively. We can apply the inductive hypothesis to conclude that each subtree has at least $2^{bh(x)-1} - 1$ internal nodes. Thus the tree rooted at x has $n \geq 2\left(2^{bh(x)-1} - 1\right) + 1 = 2^{bh(x)} - 1$ internal nodes (the $+1$ counts x).

Lemma, cont.

To conclude, let h be the height of the tree. By property 4, at least half the nodes on any path from the root to a leaf, not counting the root, must be black. Thus $bh(T) \geq \frac{h}{2}$. By the inductive argument above, the number n of internal nodes satisfies
$n \geq 2^{\frac{h}{2}} - 1$, so
$n + 1 \geq 2^{\frac{h}{2}}$, and
$\lg(n+1) \geq \frac{h}{2}$, showing
$h \leq 2\lg(n+1)$

Impact

As we will see, the red-black property can be maintained through $\Theta(h)$ insertions and deletions. This leads to the conclusion that SEARCH, MINIMUM, MAXIMUM, PREDECESSOR and SUCCESSOR can all be implemented in $\Theta(2 \lg(n+1)) = \Theta(\lg n)$ worst-case time.

Insertion Issues

Insertion into a red-black tree begins with the usual binary search tree insertion. The new tree node z is colored red with two black $T.nil$ children.

If z was inserted into an empty tree, this leads to a violation of property 2 asserting that the root is black.

The only other possible violation is that $z.p$ is red. This can be repaired using ***rotations*** and recoloring.

Black heights is still well defined, and care will be taken to preserve this.

Rotation

To maintain balance, a red-black tree must sometimes restructure the underlying tree. Rotations are the basic tool for this.

In a left rotation of the node x, the right child y of x is moved up to be the child of x's parent, while x becomes y's left child. The left subtree of y becomes the left subtree of x. All other links are preserved.

A right rotation reverses a left rotation.

Figure 13.2 in your text is a diagram of these rotations.

Insertion Fixup

The insertion of z as a red node may preserve all red-black properties. In this case, no further action is needed. If property 2 is violated, simply recoloring the root black restores the red-black properties.

Otherwise, z's parent is red. There are several cases to consider.

Case 1: If z's aunt is also red, recolor z's parent and aunt black, and z's grandparent red, making this node the new z.This may solve the problem, or may move it up the tree, where it is either a red root, easily fixed, or one of case 1,2, 2', 3, or 3' applies.

Cases 2, 3

Case 2: If z is the red right child of a red parent with a black aunt, and z's parent is the left child of z's grandparent, just left-rotate z's parent. Now the red child will be the left child of the red parent, which is in turn the left child of its parent. This puts us in case 3.

Case 3: If z is the red left child of a red parent that is the red left child of z's grandparent, right-rotate the grandparent. Color the root of the resulting subtree black, and both children red. This restores the red-black properties.

Cases 2', 3'

Case 2': If z is the red left child of a red parent with a black aunt, and z's parent is the right child of z's grandparent, just right-rotate z's parent. Now the red child will be the right child of the red parent, which is in turn the right child of its parent. This puts us in case 3'.

Case 3': If z is the red right child of a red parent that is the red right child of z's grandparent, left-rotate the grandparent. Color the root of the resulting subtree black, and both children red. This restores the red-black properties.

Θ–bounds

Each rebalancing operation is $\Theta(1)$, and at most $\Theta(h)$ operations will need to be performed: the violation may be moved up to the root then recolored.

Deletion

Deletion essentially begins with a binary search tree deletion. Deletion is achieved by splicing out a node, having its parent point directly to one of its children. If the node spliced out is black, this will cause property 2 or property 5 to be violated, and possibly property 4. The basic concept for fixing this is to put the missing black on the child.

If the child was red, then the red-black properties are restored.

Otherwise, the extra black may moved up the tree to the a red node, turning it black, or to the root, where it can be removed. Alternatively, a recoloring

Moving Up

Denote the node with the double black by x. Assume it is a left child.

Case 1: x's sibling is red: convert to case 2, 3, or 4 by rotating and recoloring to arrange for x's sibling to be black.

Case 2: x's sibling w is black and w's children are black: recolor the sibling red, and move the extra black to w's parent, now x.

Moving up, cont.

Case 3: x's sibling w is black and $w.left$ is red and $w.right$ is black: right-rotate at w and recolor to make x's new sibling w black, $w.right$ red, and $w.left$ black. This produces Case 4.

Case 4: x's sibling w is black and $w.right$ is red: left-rotate at $x.p$ and recolor to remove the extra black.

2', 3', and 4'

There are cases 2', 3', and 4' obtained by reversing left and right in the cases above.

Performance

The action for each case is $\Theta(1)$, and at most $\Theta(h) = \Theta(\lg n)$ operations will need to be performed: at worst, the extra black may be moved up to the root then removed.

Conclusion

A balanced binary tree, while intricate to implement, supports $\Theta(\lg n)$ insertion, search, and deletion, successor, and predecessor functions. Elements can be inserted indefinitely.