CSI 3334, Data Structures

Lecture 11: Red-Black and Splay Trees

Date: 2012-10-2
Author(s): Charles Stokes, Joe Flowers
Lecturer: Fredrik Niemela

1 Red-Black Trees

Red-Black trees are a popular tree structure for real time applications. They are used in the Linux Kernel, as well as the C++ STL (map). Red-black trees are height balanced. Since find operations are bound by height for worst case analysis, and since find operations are inherent in many other operations (such as insert and delete), maintaining height balance for a tree increases efficiency. The find algorithm loses efficiency when compared to AVL trees which maintain both height and weight balance. However, maintaining balance in a red-black tree is more efficient than an AVL tree, making it a better choice for some real-time applications.

1.1 Invariants

Height balance is maintained by abstractly assigning nodes as red or black, and maintaining the following invariants:

1. A node is either red or black.
2. The root is black.
3. All nill leaves are black (Instead of null pointers, many implementations use black "nill" leaves, or compensate for null pointers by considering them black leaf nodes).
4. Both children of every red node is black.
5. Any simple path from a given node to any descendent leaf contains the same number of black nodes.

The result of maintaining these is a height balance: The length of the path from the root to the furthest leaf is never more than twice the length of the path from the root to the closest leaf. In effect, the tree's height grows logarithmically.

1.2 STL version

The C++ STL implements the Map with a Red-Black tree. Since this requires a real time tree (it isn't prebuilt), and numerous find operations are performed, a height balanced Red-Black tree is useful. However, in the STL implementation, it should be noted that nodes are not colored, instead links between nodes are colored red or black. The structure maintains color information of each node's parent node link.

1.3 Maintaining Invariants

Either an insert or delete can require re-balancing.

Insert: use a find operation to find the place to insert the new value. Assume the node is red. Recursively call a re-balancing function that updates colors and performs rotations as necessary.

Delete: Find the node to delete, using the same algorithm as the BST delete algorithm, and delete the node. Note that, this is always reduced correcting the balance and deleting a node with at most one child, since in the BST delete algorithm, the successor to an interior node is a leaf node. The successor was copied into the interior node without changing it's color, and the successor node removed from the tree.

2 Splay Trees

Splay trees are not a general purpose tree structure. Accessing a node moves it to the top, no other balance conditions are guaranteed. Whereas the balance of a BST is dependent upon the only the order in which nodes were inserted into the tree, splay tree balance is dependent upon the order in which they were accessed. The worst case scenario for balance is when nodes are accessed in ascending or descending order. Random access is generally OK on average, although this doesn't necessarily yield the multi access advantage. It is most efficient, nearly constant time, to access a recently accessed node. An example use case is a cache (files may be accessed multiple times).

Accessing a node "splays" it to the top of the tree. Tree rotations are performed to promote the node based on one of three conditions:

1. If the node is the left or right child of it's parent
2. If the node is the root or not
3. If the parent is the left or right child of it's parent

The splay operation will either perform a rotation or double rotation based on these conditions, to move the node closer to the root.