CS 61B Data Structures and Programming Methodology

Aug 11, 2008
David Sun
Announcements

- Final is to be held this Thursday from 11:00 – 2:00 pm in 306 and 310 Soda.
  - Open book format.
  - If you have conflicts and have not contacted me please send me an email asap.
- Project 3 due Aug 12 at 11a.m!
  - Solve at least 15 hard puzzles under the time limit to earn full points for the programming part.
- Regrades for Midterm 1 and 2 are should be visible through glookup.
- Midterm 2 question 1 (d) is nontrivial.
  - The question is treated as a bonus question, so the midterm is now out of 49.
  - If you answered along the lines of $O(\log_3 n)$ you will have received some points. If you were not awarded any points for a similar answer come and see me.
Splay Tree

• A splay tree is a type of balanced binary search tree. Structurally, it is identical to an ordinary binary search tree; the only difference is in the algorithms for finding, inserting, and deleting entries.

• All splay tree operations run in $O(\log n)$ time on average, where $n$ is the number of entries in the tree. Any single operation can take $\Theta(n)$ time in the worst case. But any sequence of $k$ splay tree operations, with the tree initially empty and never exceeding $n$ items, takes $O(k \log n)$ worst-case time.
Rotation

• Rotation is a procedure that keeps splay tree (and other balanced search trees) balanced.
• There are two types -- a left rotation and a right rotation -- and each is the other’s reverse.
• The rotation procedure preserves the binary search tree invariant, i.e. after the rotation, the left subtree is less than or equal to the root and the right subtree is more than or equal to the root.
find

1. Walk down the tree until we find the entry with key $k$, or reach a dead end (just like binary search tree).
2. In both cases, splay the last visited node $X$ (where the search ended) up the tree to become the new root via a sequence of rotations.
   1. Recently accessed entries are near the root of the tree, so if you access the same few entries repeatedly, accesses will be very fast.
   2. If $X$ lies deeply down an unbalanced branch of the tree, the splay operation will improve the balance along that branch.
find

• Case 1 $X$ is the right child of a left child
  – let $P$ be the parent of $X$, and let $G$ be the grandparent of $X$, first rotate $X$ and $P$ left and then rotate $X$ and $G$ right.
  – called the zig-zag.
• Case 1.1 $X$ is a left child and $P$ is a right child:
  – rotate $X$ and $P$ right, then $X$ and $G$ left.
• Case 2 $X$ is the left child of a left child
  – start with the grandparent, and rotate $G$ and $P$ right. Then, rotate $P$ and $X$ right.
  – called the zig-zig.
• Case 2.1 $X$ is the right child of a right child
  – rotate $G$ and $P$ left. rotate $P$ and $X$ left.
find

• Repeatedly apply zig-zag and zig-zig rotations to $X$.
  – each pair of rotations raises $X$ two levels higher in the tree.
  – eventually, either $X$ will reach the root (and we're done), or $X$ will become the child of the root

• Case 3 $X$'s parent $P$ is the root:
  – rotate $X$ and $P$ so that $X$ becomes the root. This is called "zig" case.
Observations

• First, in none of these three cases does the depth of a subtree decrease by more than two.
• Second, every time $X$ takes two steps toward the root (zig-zig or zig-zag) every node in the subtree rooted at $X$ moves at least one step closer to the root.
• It can be shown that a node that initially lies at depth $d$ on the access path from the root to $X$ moves to a final depth no greater than $3 + d/2$. In other words, all the nodes deep down the search path have depths roughly halved.
Other Operations

- **min/max:**
  - Find the entry with the minimum or maximum key (using binary search). The node containing the minimum or maximum key is splayed to the root.

- **Insert**
  - Insert the new entry \((k, v)\), just like in an ordinary binary search tree. Then, it splays the new node to the root.

- **Remove**
  - If the node containing \(k\) is removed if it has zero or one children.
  - If it has two children, the node with the next higher key is removed instead. In either case, let \(X\) be the node removed from the tree. After \(X\) is removed, splay \(X\)'s parent to the root.
  - If the key \(k\) is not in the tree, splay the node where the search ended to the root, just like in a find() operation.
• In 2000, Danny Sleator and Robert Tarjan won the ACM Kanellakis Theory and Practice Award for their papers on splay trees and amortized analysis.

• Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in many other software.
Red-Black Trees

• Red-black tree is a binary search tree with the following additional properties:
  1. Each 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. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

• Used for Java’s TreeSet and TreeMap types.
Red-black Tree

• Every red-black tree corresponds to a 2-3-4 tree, and the operations on one correspond to those on the other.
• Each node of 2-3-4 tree corresponds to a cluster of 1–3 red-black nodes in which the top node is black and any others are red.
find

• Exactly the same as a normal binary search tree.
insert

- Insertions begins by adding the node $n$ to the tree as if it were an ordinary binary search tree.
- The newly inserted node is then colored red.
- Two properties of red-black trees can be violated:
  - Property 2: root needs to be black, which occurs if $n$ is the root.
  - Property 4: a red node cannot have a red child, which occurs when $n$’s parent is red.
- First case is easy to deal with, just color the root black.
Preserving Property 4

• Case 1: $n$’s parent and uncle are both red.
  – $n$’s grandparent must be black (since the tree was valid before the insertion)
  – Color $n$’s parent and uncle red, and the grandparent black.
  – Preserve property 1 and 4 for the grandparent.

• Case 2: $n$ is a right child, $n$’s parent is red but $n$’s uncle is black.
  – Do a left rotation through the parent so the parent is the new child
    and child is the new parent.
  – Apply the solution for case 3.

• Case 3: $n$ is a left child, $n$’s parent is red but $n$’s uncle is black.
  – Do a right rotation through the grandparent, so the new node and the
    former grandparent become the child of the parent.
  – Exchange the colors of the parent and the former grandparent.