Eksamensnoter - Binary Search Tree

André Oskar Andersen (wpr684)

March 24, 2020

12 Binary Search Trees

- Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worsst-case time. If the tree is linear chain if $n$nodes, however, the same operation take $\Theta(n)$ worst-case time. If a binary search tree is built randomly, then the expected height is $O(\lg n)$, so that basic dynamic-set operations on such a tree take $\Theta(\lg n)$ time on average

12.1 What is a binary search tree?

- A binary search tree is organized in a binary tree. We can represent such a tree by a linked data structure in which each node is an object. In addition to a *key* and satellite data, each node contains attributes *left*, *right*, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively.
- If a child or the parent is missing, the appropriate attribute contains the value `NIL`. The root node is the only node in the tree whose parent is `NIL`
- The keys in a binary search tree are always stored insuch a way as to satisfy the *binary-search-tree property*:

  > Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$

12.2 Querying a binary search tree

- Besides the `SEARCH` oepration, binary search trees can support such quieries as `MINIMUM`, `MAXIMUm`, `SUCCESSOR`, and `PREDECESSOR`. Each one has a runtime of $O(h)$ on a binary search tree of height h.

12.3 Insertion and deletion

Insertion

- Like the other primitive operations on search trees, the procedure `TREE-INSERT` runs in $O(h)$ time on a tree of height h

Deletion

- The procedure for deleting a given node z from a binary search tree T takes as arguments pointers to T and z:
  - If z has no left child, then we replace z by its right child, which may or may not be `NIL`.
  - If z has just one child, which is its left child, then we replace z by its left child

  - Otherwise, z has both a left and a right child. We find z's sucessor y (smallest number which is greater than z), which lies in z's right subtree and has no left child. We want to splice y out of its current location and have it replace z in the tree:
    * If y is z's right child, then we replace z by y, leaving y's right child alone
    * Otherwise, y lies within z's right subtree but is not z's right child. In this case, we first replace y bt its own right child, and then we replace z by y

- `TREE-DELETE` runs in $O(h)$ time on a tree of height h

13 Red-Black Trees

- Red-black trees are one of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case

13.1 Properties of red-black trees

- A *red-black tree* is a binary search tree with one extra bit of storage per node: its *color*, which can be either RED of BLACK.

- A red-black tree is a binary tree that satisfies the following *red-black properties*:
  - Every node is either red or black
  - The root is black
  - Every leaf (`NIL`) is blackIf a node is red, then bots its children are blackFor each node, all simple paths from the node to descendant leaves contain the same number of black nodes
  - If a node is red, then both its children are black
  - For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

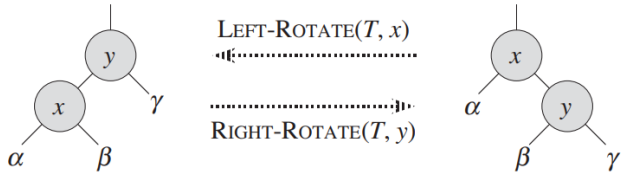
- We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the *black-height* of the node, denoted $bh(x)$. We define the black-height of a red-black tree to be the height of its root

Lemma 13.1

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

- As an immediate consequence of this lemma, we can implement the dynamic-set operations `SEARCH`, `MINIMUM`, `MAXIMUM`, `SUCCESSOR`, and `PREDECESSOR` in $O(\lg n)$, since eahc can run in $O(h)$ time on a binary search tree of height h and any red-black tree on n nodes is a binary search tree with height $O(\lg n)$

13.2 Rotations



- The search-tree operations `TREE-INSERT` and `TREE-DELETE`, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties. To restore these properties, we must change the colors of some of the nodes in the tree and also cahnge the pointer structure

- We change the pointer structure through *rotation*, which is a local operation in a search tree that preserves the binary-search-tree property. There are two kinds of rotations: left ortations and right rotations.

13.3 Insertion

- We can insert a node into an n-node red-black tree in $O(\lg n)$ time.

13.4 Deletion

- Deletion of a node takes time $O(\lg n)$

33 Computational Geometry

33.1 Line-segment properties

- A *convex combination* of two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some α in the range $0 \leq \alpha \leq 1$, we have $x_3 = \alpha x_1 + (1 - \alpha)x_2$ and $y_3 = \alpha y_1 + (1 - \alpha)y_2$. We also write that $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Intuitively, p_3 is any point that is on the line passing through p_1 and p_2 and is on or between p_1 and p_2 on the line.

- Given two distinct points p_1 and p_2, the *line swegment* $p_1\bar{p}_2$ is the set of convex combinations of p_1 and p_2

- We call p_1 and p_2 the *endpoints* of segment $p_1\bar{p}_2$.
- Sometimes the ordering of p_1 and p_2 matters, and we speak of the *directed segment* $\overrightarrow{p_1p_2}$
- If $p_1$is the origin, then we can treat the directed segment $\overrightarrow{p_1p_2}$ as the vector p_2

Cross products

- If $p_1 \times p_2$ is positive, then p_1 is clockwise from p_2 with respect to the origin. If the cross product is negative, then p_1 is counterclockwise from p_2. If the cross product is 0, then the vectors are *colinear*, pointing in either the same or opposite directions