INTRODUCTION TO ALGORITHMS

A Creative Approach

UDI MANBER

INTRODUCTION TO ALGORITHMS

A Creative Approach

UDI MANBER
University of Arizona

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

Library of Congress Cataloging-in-Publication Data

Manber, Udi.
Introduction to algorithms.

Includes bibliographies and index.
1. Data structures (Computer science)
2. Algorithms. I. Title.
QA76.9.D35M36 1989 005.7'3 88-2186
ISBN 0-201-12037-2

Reproduced by Addison-Wesley from camera-ready copy supplied by the author.

Reprinted with corrections October, 1989

EFGHIJ-DO-943210

Deletion

Deletions are generally more complicated. It is easy to delete a leaf; we need only to change the pointer to it to be nil. It is also not hard to delete a node that has only one child; the pointer to the node is changed to point to its child. However, if the node we want to delete has two children, then we need to find a place for the two pointers. Let B be a node with two children whose key we want to delete (see Fig. 4.11). In the first step, we exchange the key of B with a key of another node X, such that (1) X has at most one child, and (2) deleting X (after the exchange) will leave the tree consistent. In the second step, we delete X, which now has the key of B which we wanted to delete. We can easily delete X, because it has at most one child. To preserve the consistency of the tree, the key of X must be at least as large as all the keys in the left subtree of B, and must be smaller than all the keys in the right subtree of B. Notice that the key of X in Fig. 4.11 satisfies these constraints: it is the largest among the keys in the left subtree of B. X is called the **predecessor** of B in the tree. X cannot have a right child, since otherwise it would not have the largest key in that subtree. The deletion algorithm is presented in Fig. 4.12.

Complexity The running times of search, insert, and delete depend on the shape of the tree and the location of the relevant node. In the worst case, the search path would take us all the way to the bottom. All the other steps in the algorithms require only constant time (e.g., the actual insertion, the exchange of keys in the deletion). So, the worst-case running time is the maximal length of a path from the root to a leaf, which is the height of the tree. If the tree is reasonably balanced (we will define balance shortly), then its height is approximately $\log_2 n$, where n is the number of nodes in the tree. All the operations are efficient in this case. If the tree is unbalanced, then these operations are much less efficient.

If the keys are inserted into a binary search tree in a random order, then the expected height of the tree is $O(\log n)$ — more precisely, $2 \ln n$. In this case, the search and insert operations are efficient. In the worst case, however, the height of the tree can be n (when the tree is a simple linked list). Trees with long paths can result, for example, from insertions in a sorted, or close to sorted, order. Also, deletions may cause problems even if they occur in a random order. The main reason for that is the asymmetry of always using the predecessor to replace a deleted node. If there are frequent deletions,

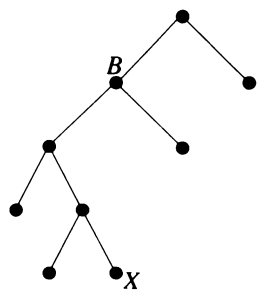

Figure 4.11 Deleting a node with two children.