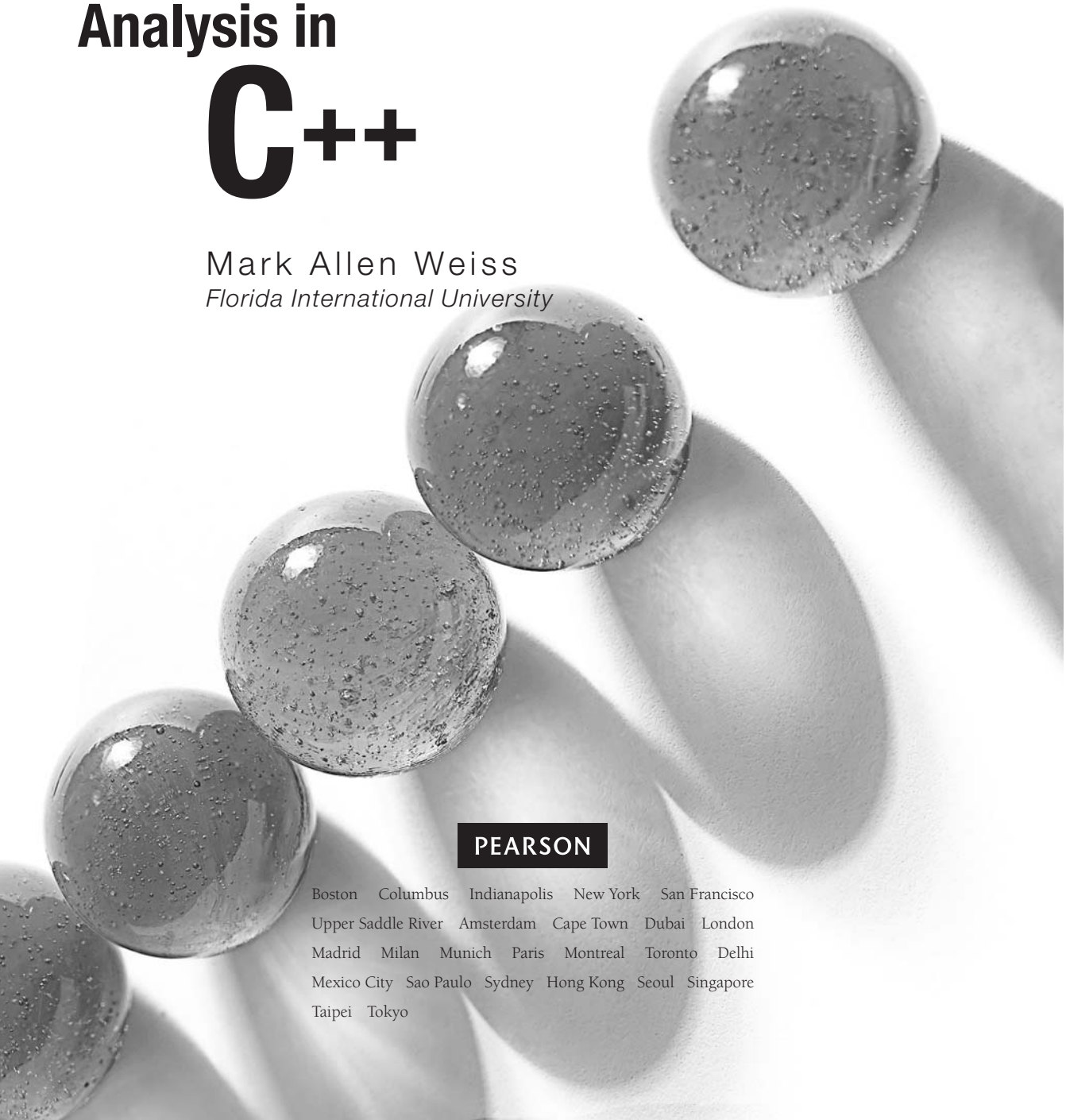
Fourth Edition

Data Structures and Algorithm Analysis in C++

Mark Allen Weiss
Florida International University



PEARSON

Boston Columbus Indianapolis New York San Francisco
Upper Saddle River Amsterdam Cape Town Dubai London
Madrid Milan Munich Paris Montreal Toronto Delhi
Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore
Taipei Tokyo

Editorial Director, ECS: Marcia Horton
Executive Editor: Tracy Johnson
Editorial Assistant: Jenah Blitz-Stoehr
Director of Marketing: Christy Lesko
Marketing Manager: Yez Alayan
Senior Marketing Coordinator: Kathryn Ferranti
Marketing Assistant: Jon Bryant
Director of Production: Erin Gregg
Senior Managing Editor: Scott Disanno
Senior Production Project Manager: Marilyn Lloyd
Manufacturing Buyer: Linda Sager
Art Director: Jayne Conte
Cover Designer: Bruce Kenselaar
Permissions Supervisor: Michael Joyce
Permissions Administrator: Jenell Forschler
Cover Image: © De-kay | Dreamstime.com
Media Project Manager: Renata Butera
Full-Service Project Management: Integra Software Services Pvt. Ltd.
Composition: Integra Software Services Pvt. Ltd.
Text and Cover Printer/Binder: Courier Westford

Library of Congress Cataloging-in-Publication Data

Weiss, Mark Allen.
Data structures and algorithm analysis in C++ / Mark Allen Weiss, Florida International University. — Fourth edition.
pages cm
ISBN-13: 978-0-13-284737-7 (alk. paper)
ISBN-10: 0-13-284737-X (alk. paper)
1. C++ (Computer program language) 2. Data structures (Computer science) 3. Computer algorithms. I. Title.
QA76.73.C153W46 2014
005.7′3—dc23
2013011064

10 9 8 7 6 5 4 3 2 1

PEARSON

www.pearsonhighered.com

ISBN-10: 0-13-284737-X
ISBN-13: 978-0-13-284737-7

will be `insert(x,p->left)` or `insert(x,p->right)`. Either way, `t` is now a reference to either `p->left` or `p->right`, meaning that `p->left` or `p->right` will be changed to point at the new node. All in all, a slick maneuver.

4.3.4 remove

As is common with many data structures, the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a link to bypass the node (we will draw the link directions explicitly for clarity). See Figure 4.24.

The complicated case deals with a node with two children. The general strategy is to replace the data of this node with the smallest data of the right subtree (which is easily found) and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second `remove` is an easy one. Figure 4.25 shows an initial tree and the result of a deletion. The node to be deleted is the left child of the root; the key value is 2. It is replaced with the smallest data in its right subtree (3), and then that node is deleted as before.

The code in Figure 4.26 performs deletion. It is inefficient because it makes two passes down the tree to find and delete the smallest node in the right subtree when this is appropriate. It is easy to remove this inefficiency by writing a special `removeMin` method, and we have left it in only for simplicity.

If the number of deletions is expected to be small, then a popular strategy to use is **lazy deletion:** When an element is to be deleted, it is left in the tree and merely *marked* as being deleted. This is especially popular if duplicate items are present, because then the data member that keeps count of the frequency of appearance can be decremented. If the number of real nodes in the tree is the same as the number of "deleted" nodes, then the depth of the tree is only expected to go up by a small constant (why?), so there is a very small time penalty associated with lazy deletion. Also, if a deleted item is reinserted, the overhead of allocating a new cell is avoided.

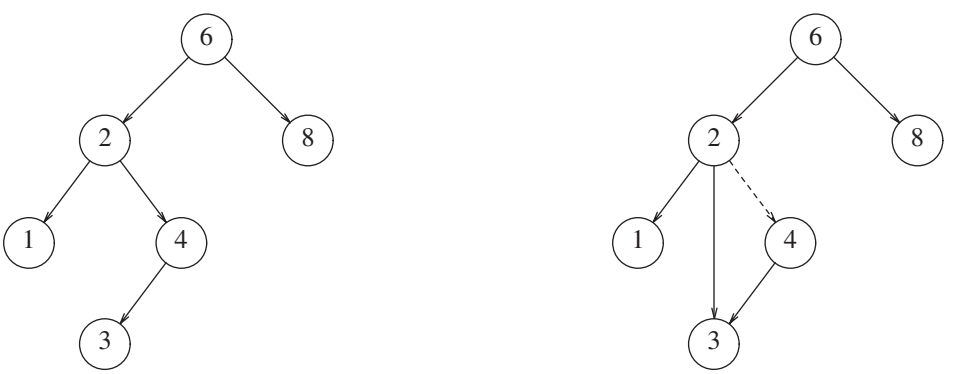


Figure 4.24 Deletion of a node (4) with one child, before and after

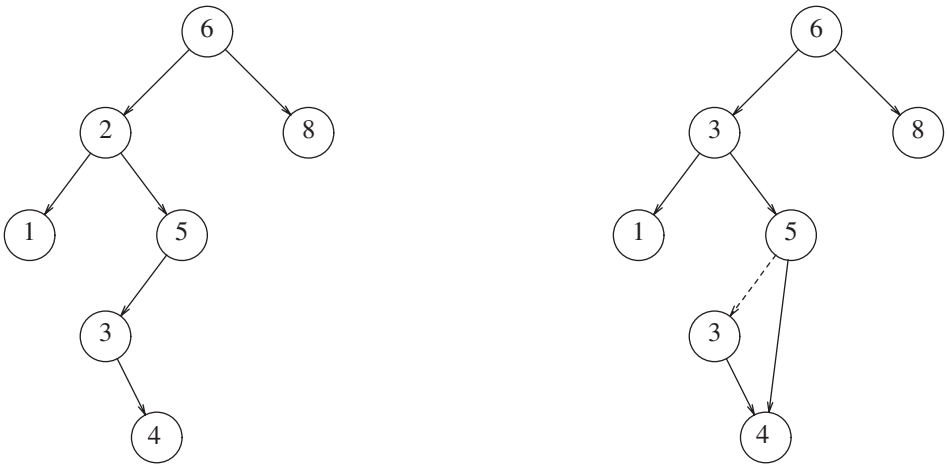


Figure 4.25 Deletion of a node (2) with two children, before and after

```
1   /**
2    * Internal method to remove from a subtree.
3    * x is the item to remove.
4    * t is the node that roots the subtree.
5    * Set the new root of the subtree.
6    */
7   void remove( const Comparable & x, BinaryNode * & t )
8   {
9       if( t == nullptr )
10          return;   // Item not found; do nothing
11      if( x < t->element )
12          remove( x, t->left );
13      else if( t->element < x )
14          remove( x, t->right );
15      else if( t->left != nullptr && t->right != nullptr ) // Two children
16      {
17          t->element = findMin( t->right )->element;
18          remove( t->element, t->right );
19      }
20      else
21      {
22          BinaryNode *oldNode = t;
23          t = ( t->left != nullptr ) ? t->left : t->right;
24          delete oldNode;
25      }
26  }
```

Figure 4.26 Deletion routine for binary search trees