DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Adam Drozdek

Data Structures and Algorithms in C++

Fourth Edition

Adam Drozdek

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s).
Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

CENGAGE Learning®

Data Structures and Algorithms in C++, Fourth Edition
by Adam Drozdek

Executive Editor: Marie Lee

Senior Product Manager: Alyssa Pratt

Associate Product Manager: Stephanie Lorenz

Content Project Manager: Matthew Hutchinson

Art Director: Cheryl Pearl

Print Buyer: Julio Esperas

Compositor: PreMediaGlobal

Proofreader: Andrea Schein

Indexer: Sharon Hilgenberg

© 2013 Cengage Learning

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at
Cengage Learning Customer & Sales Support, www.cengage.com/support

For permission to use material from this text or product, submit all requests online at **www.cengage.com/permissions**
Further permissions questions can be emailed to
permissionrequest@cengage.com

Library of Congress Control Number: 2012942244

ISBN-13: 978-1-133-60842-4

ISBN-10: 1-133-60842-6

Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil and Japan. Locate your local office at:
www.cengage.com/global

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

To learn more about Cengage Learning, visit **www.cengage.com**

Purchase any of our products at your local college store or at our preferred online store **www.cengagebrain.com**

Some of the product names and company names used in this book have been used for identification purposes only and may be trademarks or registered trademarks of their respective manufacturers and sellers.

Any fictional data related to persons or companies or URLs used throughout this book is intended for instructional purposes only. At the time this book was printed, any such data was fictional and not belonging to any real persons or companies.

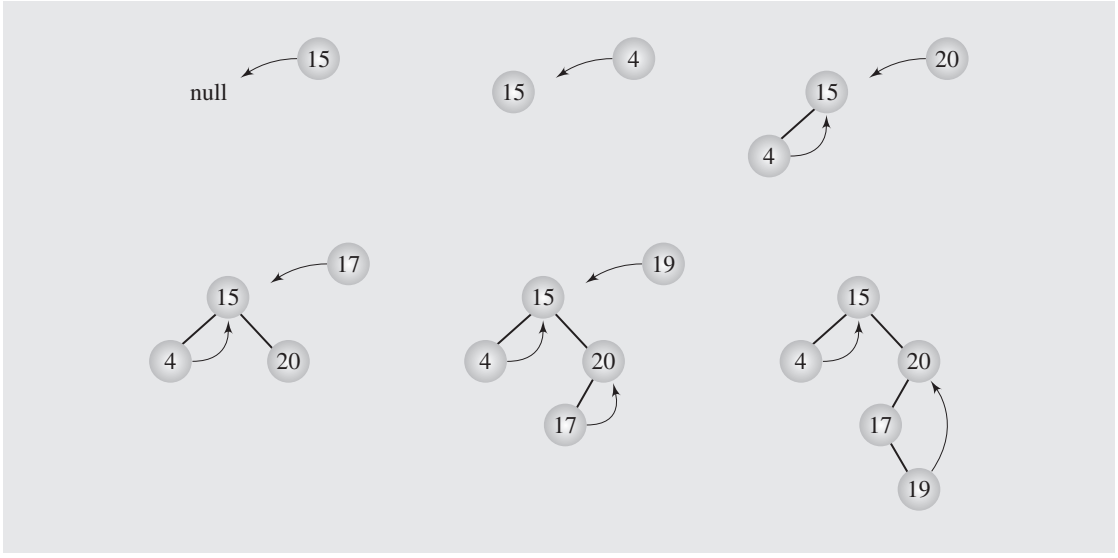
Cengage Learning reserves the right to revise this publication and make changes from time to time in its content without notice.

The programs in this book are for instructional purposes only. They have been tested with care, but are not guaranteed for any particular intent beyond educational purposes. The author and the publisher do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs.

Printed in the United States of America
1 2 3 4 5 6 7 18 17 16 15 14 13 12

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

FIGURE 6.25 Inserting nodes into a threaded tree.

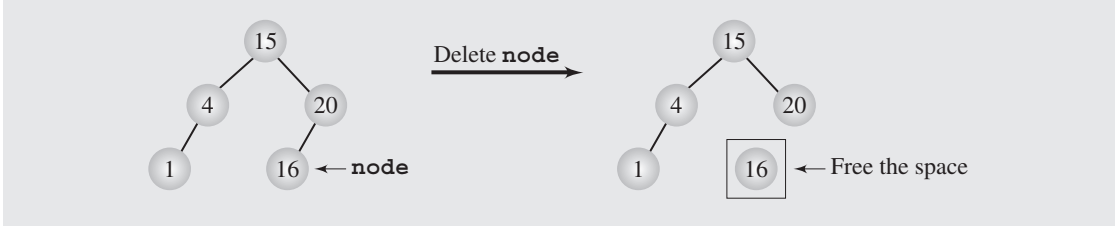


6.6 Deletion

Deleting a node is another operation necessary to maintain a binary search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has. There are three cases of deleting a node from the binary search tree:

1. The node is a leaf; it has no children. This is the easiest case to deal with. The appropriate pointer of its parent is set to null and the node is disposed of by `delete` as in Figure 6.26.

FIGURE 6.26 Deleting a leaf.

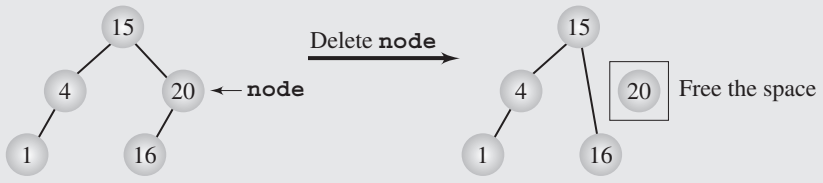


2. The node has one child. This case is not complicated. The parent's pointer to the node is reset to point to the node's child. In this way, the node's children are lifted up by one

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

level and all great-great-. . . grandchildren lose one "great" from their kinship designations. For example, the node containing 20 (see Figure 6.27) is deleted by setting the right pointer of its parent containing 15 to point to 20's only child, which is 16.

FIGURE 6.27 Deleting a node with one child.



3. The node has two children. In this case, no one-step operation can be performed because the parent's right or left pointer cannot point to both the node's children at the same time. This section discusses two different solutions to this problem.

6.6.1 Deletion by Merging

This solution makes one tree out of the two subtrees of the node and then attaches it to the node's parent. This technique is called *deleting by merging.* But how can we merge these subtrees? By the nature of binary search trees, every key of the right subtree is greater than every key of the left subtree, so the best thing to do is to find in the left subtree the node with the greatest key and make it a parent of the right subtree. Symmetrically, the node with the lowest key can be found in the right subtree and made a parent of the left subtree.

The desired node is the rightmost node of the left subtree. It can be located by moving along this subtree and taking right pointers until null is encountered. This means that this node will not have a right child, and there is no danger of violating the property of binary search trees in the original tree by setting that rightmost node's right pointer to the right subtree. (The same could be done by setting the left pointer of the leftmost node of the right subtree to the left subtree.) Figure 6.28 depicts this operation. Figure 6.29 contains the implementation of the algorithm.

It may appear that `findAndDeleteByMerging()` contains redundant code. Instead of calling `search()` before invoking `deleteByMerging()`, `findAndDeleteByMerging()` seems to forget about `search()` and searches for the node to be deleted using its private code. But using `search()` in function `findAndDeleteByMerging()` is a treacherous simplification. `search()` returns a pointer to the node containing `el`. In `findAndDeleteByMerging()`, it is important to have this pointer stored specifically in one of the pointers of the node's parent. In other words, a caller to `search()` is satisfied if it can access the node from any direction, whereas `findAndDeleteByMerging()` wants to access it from either its parent's left or right pointer data member. Otherwise, access to the entire subtree having this node as its root would be lost. One reason for this is the fact that `search()` focuses on the

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

FIGURE 6.28 Summary of deleting by merging.

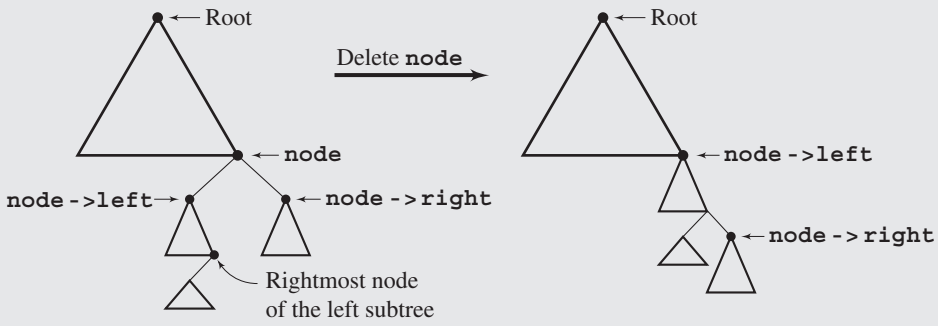


FIGURE 6.29 Implementation of an algorithm for deleting by merging.

```
template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node) {
    BSTNode<T> *tmp = node;
    if (node != 0) {
        if (!node->right)           // node has no right child: its left
             node = node->left;     // child (if any) is attached to its
                                    // parent;
        else if (node->left == 0)   // node has no left child: its right
             node = node->right;    // child is attached to its parent;
        else {                      // be ready for merging subtrees;
             tmp = node->left;      // 1. move left
             while (tmp->right != 0)// 2. and then right as far as
                                    // possible;
                tmp = tmp->right;
             tmp->right =           // 3. establish the link between
                node->right;        //    the rightmost node of the left
                                    //    subtree and the right subtree;
             tmp = node;            // 4.
             node = node->left;     // 5.
        }
        delete tmp;                 // 6.
     }
}
```

Continues

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

FIGURE 6.29 *(continued)*

```
template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0) {
        if (node->el == el)
             break;
        prev = node;
        if (el < node->el)
             node = node->left;
        else node = node->right;
    }
    if (node != 0 && node->el == el)
         if (node == root)
              deleteByMerging(root);
         else if (prev->left == node)
              deleteByMerging(prev->left);
         else deleteByMerging(prev->right);
    else if (root != 0)
         cout << "element" << el << "is not in the tree\n";
    else cout << "the tree is empty\n";
}
```

node's key, and `findAndDeleteByMerging()` focuses on the node itself as an element of a larger structure, namely, a tree.

Figure 6.30 shows each step of this operation. It shows what changes are made when `findAndDeleteByMerging()` is executed. The numbers in this figure correspond to numbers put in comments in the code in Figure 6.29.

The algorithm for deletion by merging may result in increasing the height of the tree. In some cases, the new tree may be highly unbalanced, as Figure 6.31a illustrates. Sometimes the height may be reduced (see Figure 6.31b). This algorithm is not necessarily inefficient, but it is certainly far from perfect. There is a need for an algorithm that does not give the tree the chance to increase its height when deleting one of its nodes.

6.6.2 Deletion by Copying

Another solution, called *deletion by copying,* was proposed by Thomas Hibbard and Donald Knuth. If the node has two children, the problem can be reduced to one of two simple cases: the node is a leaf or the node has only one nonempty child. This can be done by replacing the key being deleted with its immediate predecessor (or successor). As already indicated in the algorithm deletion by merging, a key's predecessor is the key in the rightmost node in the left subtree (and analogically, its immediate successor is the key in the leftmost node in the right subtree). First, the predecessor has to be

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

FIGURE 6.30 Details of deleting by merging.

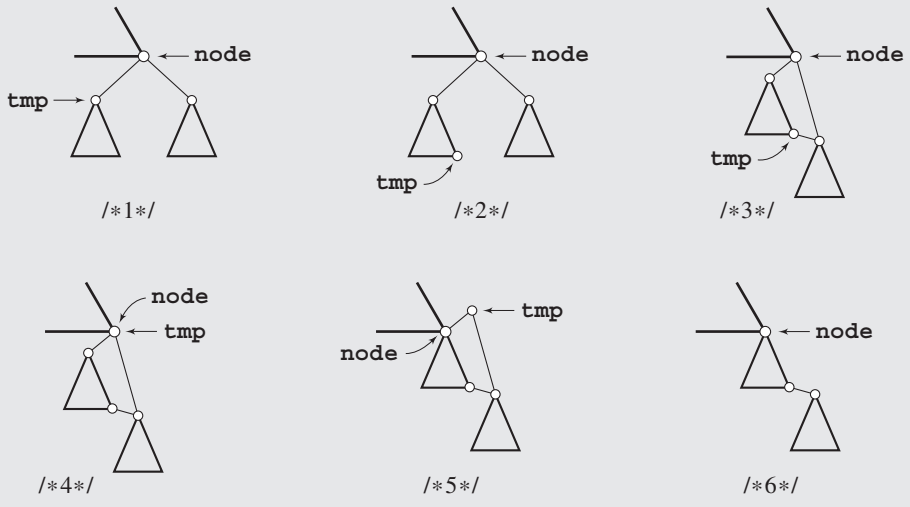
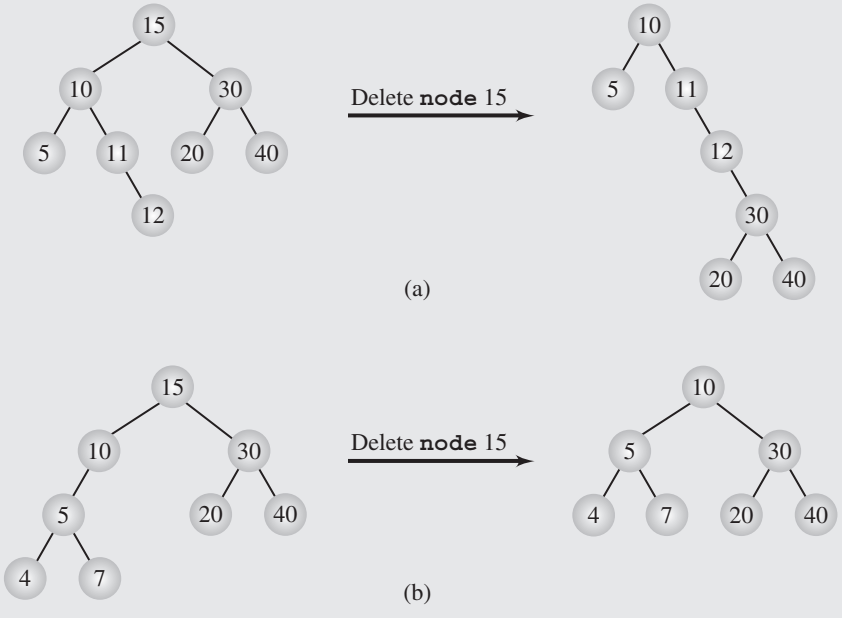


FIGURE 6.31 The height of a tree can be (a) extended or (b) reduced after deleting by merging.



Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

located. This is done, again, by moving one step to the left by first reaching the root of the node's left subtree and then moving as far to the right as possible. Next, the key of the located node replaces the key to be deleted. And that is where one of two simple cases comes into play. If the rightmost node is a leaf, the first case applies; however, if it has one child, the second case is relevant. In this way, deletion by copying removes a key k_1 by overwriting it by another key k_2 and then removing the node that holds k_2, whereas deletion by merging consisted of removing a key k_1 along with the node that holds it.

To implement the algorithm, two functions can be used. One function, `deleteByCopying()`, is illustrated in Figure 6.32. The second function, `findAndDeleteByCopying()`, is just like `findAndDeleteByMerging()`, but it calls `deleteByCopying()` instead of `deleteByMerging()`. A step-by-step trace is shown in Figure 6.33, and the numbers under the diagrams refer to the numbers indicated in comments included in the implementation of `deleteByCopying()`.

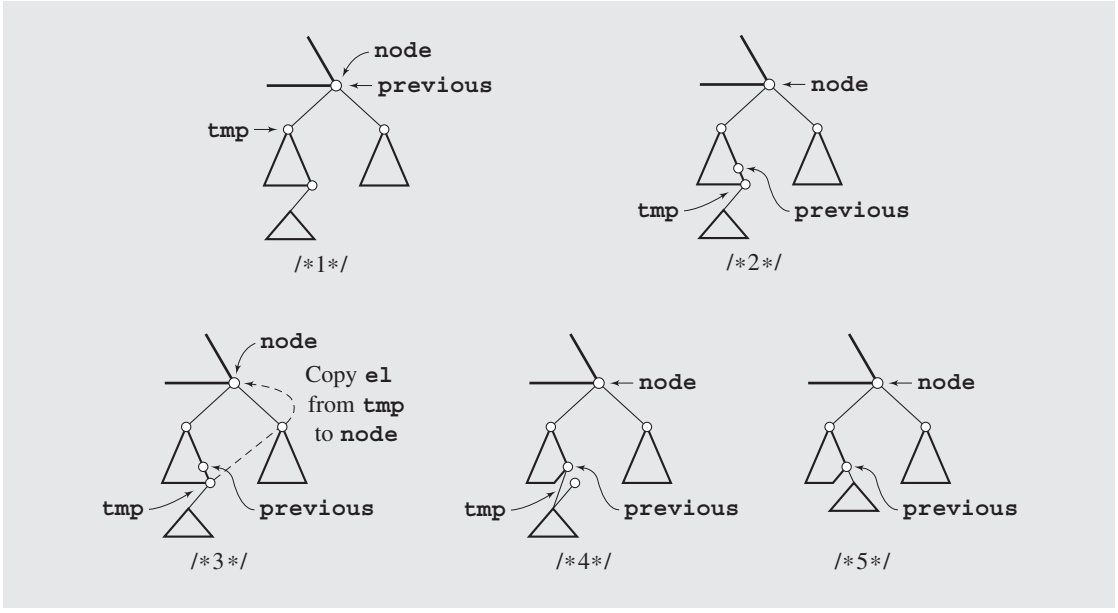
FIGURE 6.32 Implementation of an algorithm for deleting by copying.

```
template<class T>
void BST<T>::deleteByCopying(BSTNode<T>*& node) {
    BSTNode<T> *previous, *tmp = node;
     if (node->right == 0)                    // node has no right child;
          node = node->left;
     else if (node->left == 0)                // node has no left child;
          node = node->right;
     else {
          tmp = node->left;                   // node has both children;
          previous = node;                    // 1.
          while (tmp->right != 0) {           // 2.
              previous = tmp;
              tmp = tmp->right;
          }
          node->el = tmp->el;                 // 3.
          if (previous == node)
               previous ->left  = tmp->left;
          else previous ->right = tmp->left;  // 4.
     }
     delete tmp;                              // 5.
}
```

This algorithm does not increase the height of the tree, but it still causes a problem if it is applied many times along with insertion. The algorithm is asymmetric; it always deletes the node of the immediate predecessor of the key in `node`, possibly reducing the height of the left subtree and leaving the right subtree unaffected. Therefore, the right subtree of `node` can grow after later insertions, and if the key

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

FIGURE 6.33 Deleting by copying.



in `node` is again deleted, the height of the right tree remains the same. After many insertions and deletions, the entire tree becomes right unbalanced, with the right subtree bushier and larger than the left subtree.

To circumvent this problem, a simple improvement can make the algorithm symmetrical. The algorithm can alternately delete the predecessor of the key in `node` from the left subtree and delete its successor from the right subtree. The improvement is significant. Simulations performed by Jeffrey Eppinger show that an expected internal path length for many insertions and asymmetric deletions is $\Theta(n \lg^3 n)$ for n nodes, and when symmetric deletions are used, the expected IPL becomes $\Theta(n \lg n)$. Theoretical results obtained by J. Culberson confirm these conclusions. According to Culberson, insertions and asymmetric deletions give $\Theta(n\sqrt{n})$ for the expected IPL and $\Theta(\sqrt{n})$ for the average search time (average path length), whereas symmetric deletions lead to $\Theta(\lg n)$ for the average search time, and as before, $\Theta(n \lg n)$ for the average IPL.

These results may be of moderate importance for practical applications. Experiments show that for a 2,048-node binary tree, only after 1.5 million insertions and asymmetric deletions does the IPL become worse than in a randomly generated tree.

Theoretical results are only fragmentary because of the extraordinary complexity of the problem. Arne Jonassen and Donald Knuth analyzed the problem of random insertions and deletions for a tree of only three nodes, which required using Bessel functions and bivariate integral equations, and the analysis turned out to rank among "the more difficult of all exact analyses of algorithms that have been carried out to date." Therefore, the reliance on experimental results is not surprising.

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.