# Data Structures and Algorithms in Java™

**Sixth Edition**

**Michael T. Goodrich**
Department of Computer Science
University of California, Irvine

**Roberto Tamassia**
Department of Computer Science
Brown University

**Michael H. Goldwasser**
Department of Mathematics and Computer Science
Saint Louis University

# Instructor's Solutions Manual

WILEY

# Chapter 8 Trees

## Hints and Solutions

### Reinforcement

**R-8.1) Hint** Be sure not to get confused between depth (which is for a single node) and height (which is for the entire tree).

**R-8.2) Hint** Recall that the worst-case running time for the depth method is $O(n)$.

**R-8.3) Hint** Use the definitions of height and depth, and note that the height of the tree has to be realized by some path from the root to an external node. In addition, using induction is the easiest justification technique to use with this proposition.

**R-8.4) Hint** Use a new parameter, $n_p$, which refers to the number of positions in the subtree rooted at $p$.

**R-8.4) Solution** The running time is $O(n_p)$, where $n_p$ is the number of positions in the subtree rooted at position $p$.

**R-8.5) Hint** Review the binary tree ADT.

**R-8.6) Hint** Consider using dummy nodes.

**R-8.7) Hint** You need to give four values—the minimum and maximum number of internal nodes and the same for external nodes. In addition, one of these four values is 1.

**R-8.7) Solution** A tree that is one long path would have $n-1$ internal nodes and 1 external node. A tree that is proper (and thus odd $n$) would have $\frac{n-1}{2}$ internal nodes and $\frac{n+1}{2}$ external nodes.

**R-8.8) Hint** Use the formulas presented in the book that relate external nodes, internal nodes, and height.

**R-8.9) Hint** Consider how any proper binary tree with $n$ nodes can be related to a proper binary tree with $n-2$ nodes.

**R-8.10) Hint** Each subtree is rooted at a node of the tree; give the value associated with each such node.

**R-8.11) Hint** This one is a real puzzler, and it doesn't even use the operators $+$ and $*$.

**R-8.11) Solution** The tree expresses the formula $(6/(1-(5/7)))$.

**R-8.12) Hint** Review how arithmetic expressions can be represented with binary trees.

**R-8.13) Hint** Assume that the container of children is implemented as a positional list.

**R-8.13) Solution** The size and isEmpty methods run in $O(1)$ assuming we maintain a top-level count of the number of nodes in the tree. The methods root, parent, isRoot, isInternal, isExternalare also constant since we need only to check local fields of the node or the tree's root to determine these. The numChildren method is constant-time, because it is simply the size method of the secondary container of children. Reporting children($p$) requires $O(c_p+1)$ time because it requires iterating through the entire collection of children (even if the container were empty). The analysis of depth and height were given in Section 8.1.3.

**R-8.14) Hint** Think about a tree that is really a path.

**R-8.15) Hint** Recall the definition of the level numbering and use this definition when passing information from a node to its children.

**R-8.16) Hint** All of these methods can be easily performed using formulas involving level numbers.

**R-8.17) Hint** Use a substitution of $g(p) = f(p)+1$.

**R-8.18) Hint** Draw the tree on a separate sheet of paper and then mark nodes as they are visited, writing down the output produced for each visit.

**R-8.19) Hint** Draw the tree on a separate sheet of paper and then mark nodes as they are visited, writing down the output produced for each visit.

**R-8.19) Solution** 3 1 + 3 x 9 5 - 2 + - 3 7 4 - x 6 + -

**R-8.20) Hint** Draw a tree with one node, one with two nodes, and then one with three nodes.

**R-8.20) Solution** It is not possible for the postorder and preorder traversal of a tree with more than one node to visit the nodes in the same order. A preorder traversal will always visit the root node first, while a postorder traversal node will always visit an external node first.

It is possible for a preorder and a postorder traversal to visit the nodes in the reverse order. Consider the case of a tree with only two nodes.

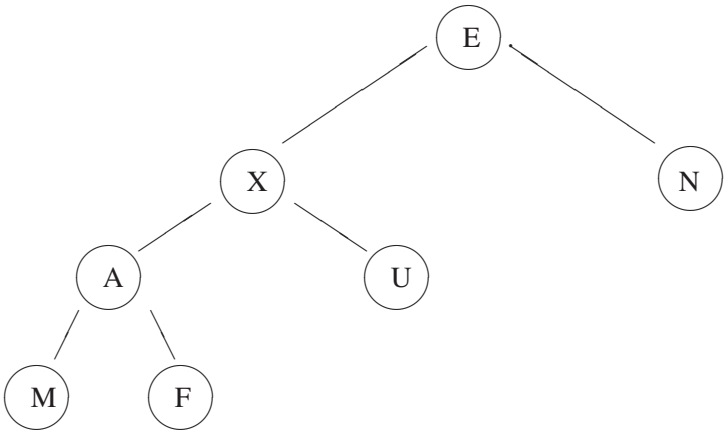
**R-8.21) Hint** Draw a tree with one node and then one with three nodes (and note it is impossible to draw a proper binary tree with exactly two nodes).

**R-8.21) Solution** It is not possible for the post- and preorder traversals to visit the nodes of a proper binary tree in the same order for the same reason in the previous question.

It is not possible for the post- and preorder traversals to visit the nodes of a proper binary tree in the reverse order. Let $a$ be the root of a proper binary tree and let $T_1$ and $T_2$ be the left and right subtrees. A postorder traversal would visit the postorder traversal of $T_1$, the postorder traversal of $T_2$ and then node $a$ while the preorder traversal would visit node $a$, the preorder traversal of $T_1$ and then the preorder traversal of $T_2$. Clearly the postorder and preorder traversals cannot be the reverse of each other since in both cases, all the nodes of $T_1$ are visited before all the nodes of $T_2$.

**R-8.22) Hint** Draw the tree starting with the beginning and end characters of the two character strings.

**R-8.22) Solution** The tree is drawn below.




**R-8.23) Hint** We already got you started...

**R-8.23) Solution**

$\{1\}$
$\{2,3,4\}$
$\{3,4,5,6\}$
$\{4,5,6,7,8,9,10\}$
and so on.

**R-8.24) Hint** Draw the tree again and use a pencil and paper to mark how the recursive calls move around the tree.

**R-8.25) Hint** The trick is to keep track of how much of the current line you have already used up and whether the next piece of output will exceed the 80 character limit.

**R-8.26) Hint** Argue that this method basically performs the same computations as a familiar tree traversal method.

**R-8.26) Solution** The running is $O(n)$ since parenthesize is just a preorder traversal with output.

## Creativity

**C-8.27) Hint** Define the method recursively.

**C-8.28) Hint** Modify an algorithm for computing the depth of each position so that it computes path lengths at the same time.

**C-8.29) Hint** Use the fact that we can build $T$ from a single root tree via a series of pairs of insertLeft and insertRight operations.

**C-8.30) Hint** The minimum value will occur when $T$ has one external node.

**C-8.31) Hint** Consider how you could possibly combine a tree with maximum depth with a tree with minimum depth.

**C-8.31) Solution** Let $T_1$ be a tree of $n/2$ nodes in a single path from the root to a single external node $v$. And let $T_2$ be a tree of $n/2$ nodes having $\lfloor n/2 \rfloor + 1$ external nodes. Now, create $T$ by attaching $T_2$ to $T_1$ at $v$.

**C-8.32) Hint** First show that there is a node with three external node children, and then consider what changes when the children of that node are all removed.

**C-8.32) Solution** We will show this using induction. For $n_I = 0$, then $n_E = 2n_I + 1 = 1$. This is obviously true. For $n_I = 1$, then $n_E = 2n_I + 1 = 2 + 1 = 3$. Again, this is obviously true from our problem definition. Now let us assume that the $n_E$ equation holds true for $k' < k$, i.e., for any $n_I = k' < k$, $n_E = 2n_I + 1$.

Now consider $n_I = k$. Then, $n_E = 2(k-1) + 1 + (3-1)$. That is, the number of external nodes is equal to the number of external nodes for a tree with $k-1$ internal nodes plus 3 (we added an internal node which must have 3 children) minus 1 (in creating the new internal node, we made an external node into an internal node). Thus, $n_E = 2k - 2 + 3 = 2k + 1$. This is what we needed to show.

**C-8.33) Hint** Use the definition to derive a recursive algorithm.

**C-8.34) Hint** Explore the different ways you might attach external nodes at the bottom level of a tree.

**C-8.35) Hint** Explore with pen and paper.

**C-8.35) Solution** There are 5 such trees.

**C-8.36) Hint** Can you tell how many elements are removed?

**C-8.36) Solution** Based on our current representation, this would require $O(n_p)$ time where $n_p$ is the number of nodes in the subtree rooted at position $p$.

**C-8.37) Hint** There are many special cases to consider when $p$ and $q$ neighbor each other.

**C-8.38) Hint** Try to avoid conditionals when possible.

**C-8.39) Hint** Recursion can be very helpful.

**C-8.40) Hint** Build the new tree in preorder fashion.

**C-8.41) Hint** You may use the technique from either of the previous two problems.

**C-8.42) Hint** Use a tree traversal.

**C-8.42) Solution** We can accomplish the task of printing the element stored at $p$ along with the height of the subtree rooted at $p$ by using a postorder traversal. During this traversal, we will find the height of each subtree. The height for a subtree at $p$ will be 0 if $p$ is a leaf and otherwise one more than the height of the max child. We can print out the element at $p$ and its computed height during the postorder visit.

**C-8.43) Hint** Derive a formula that relates the depth of a position $p$ to the depths of positions adjacent to $p$.

**C-8.43) Solution** This can be done using a preorder traversal. When doing a "visit" in the traversal, simply store the depth of the position's parent incremented by 1. Now, every node will contain its depth.

**C-8.44) Hint** Try to compute node heights and balance factors at the same time.

**C-8.44) Solution** One way to do this is the following: for external nodes, set height and balance to be zero. Then, do the following for the post-visit method:

**Algorithm** post-visit():
  **if** $v$.isInternal($v$) **then**
    **if** $v.leftchild.height > v.rightchild.height$ **then**
      $v.height = v.leftchild.height + 1$;
    **else**
      $v.height = v.rightchild.height + 1$;
    $v.balance = abs(v.rightchild.height - v.leftchild.height)$;
  print $v.balance$

**C-8.45) Hint** Think about what could be the worst-case number of nodes that would have to be traversed to answer each of these queries.

**C-8.45) Solution** The algorithms are as follows:

**Algorithm** preorderNext(Node v):
  **if** $v$.isInternal() **then**
    **return** $v$'s left child
  **else**

```
    Node p = parent of v
    if v is left child of p then
      return right child of p
    else
      while v is not left child of p do
        v = p
        p = p.parent
      return right child of p
```

**Algorithm** inorderNext(Node v):

```
  if v.isInternal() then
    Node p = left child of v
    while p has left child do
      p = left child of p
      return p
  else
    Node p = parent of v
    if v is left child of p then
      return p
    else
      while v is not left child of p do
        v = p
        p = p.parent
      return p
```

**Algorithm** postorderNext(Node v):

```
  if v.isInternal() then
    p = parent of v
    if v is right child of p then
      return p
    else
      v = right child of p
      while v is not external do
        v = left child of v
      return v
  else
    p = parent of v
    if v is left child of p then
      return right child of p
    else
      return p
```

The worst-case running times for these algorithms are all $O(h)$ where $h$ is

the height of the tree $T$.

**C-8.46) Hint** Use a stack.

**C-8.46) Solution**

**Algorithm** inorder(Tree $T$):

```
Stack S ← new Stack()
Node v ← T.root()
push v
while S is not empty do
  while v is internal do
    v ← v.left
    push v
  while S is not empty do
    pop v
    visit v
    if v is internal then
      v ← v.right
      push v
    while v is internal do
      v ← v.left
      push v
```

**C-8.47) Hint** The algorithm from Exercise C-8.45 will be useful.

**C-8.48) Hint** The algorithm from Exercise C-8.45 will be useful.

**C-8.49) Hint** The algorithm from Exercise C-8.45 will be useful.

**C-8.50) Hint** Use induction to show that no crossing edges are produced.

**C-8.51) Hint** Use induction to show that no crossing edges are produced.

**C-8.52) Hint** The answer to the first part of (c) is "yes".

**C-8.52) Solution**

a) yes

b) no

c) yes, postorder.

**C-8.53) Hint** The $y$ coordinates can be the same as in the binary tree case. The trick, then, is to find a good substitute for the inorder number used in the binary tree drawing algorithm.

**C-8.54) Hint** First derive a formula for $post(p) - pre(p)$.

**C-8.54) Solution** $post(p) = desc(p) - 1 - depth(p) + pre(p)$. To prove this, consider the difference, $post(p) - pre(p)$. This difference is equal to all the positions counted by postorder but not preorder (the strict descendants of $p$, whose number is $desc(p) - 1$) minus all the positions counted by preorder but not postorder (the ancestors of $p$, whose number is $depth(p)$).

**C-8.55) Hint** It helps to know the relative depths of $p$ and $q$.

**C-8.55) Solution** The algorithm is given below.

**Algorithm** LCA(Node $v$, Node $w$):

```
int v_dpth = v.depth
int w_dpth = w.depth
while v_dpth > w_dpth do
  v = v.parent
while w_dpth > v_dpth do
  w = w.parent
while v ≠ w do
  v = v.parent
  w = w.parent
return v
```

**C-8.56) Hint** Consider what numbers $f(p)+1$, $f(q)+1$, and $f(a)+1$ look like in binary.

**C-8.57) Hint** Be careful—the path establishing the diameter might not include the root of the tree.

**C-8.58) Hint** Consider a recursive algorithm.

**C-8.58) Solution** The algorithm is given below.

**Algorithm** indentedParentheticRepresentation(Tree $T$, Position $v$, int *indent*):

```
print out indent number of tabs
if T.isExternal(v) then
  print v.element().toString()
else
  indent+ = 1
  print v.element().toString() + "("
  Enumeration children_of_v = T.children(v)
  while children_of_v.hasMoreElements() do
    Position w = (Position) children_of_v.nextElement()
    indentedParentheticRepresentation(T, w, indent)
  indent− = 1
  print out indent number of tabs
```

**C-8.59) Hint** First convert the infix expression into its equivalent binary tree representation.

**C-8.60) Hint** Consider a recursive algorithm.

## Projects

**P-8.61) Hint** What are natural update methods for this representation?

**P-8.62) Hint** Use a Java list for the children.

**P-8.63) Hint** Review the definition of this representation to get the details right.

**P-8.64) Hint** You can use an ArrayList of positions to represent the tree path.

**P-8.65) Hint** Consider building the expression tree using a recursive algorithm, as described in the book.

**P-8.66) Hint** The essential part of this structure is just a binary tree, so take each part one step at a time.

**P-8.67) Hint** This is a huge project, so plan accordingly.

**P-8.68) Hint** Use recursion where appropriate.

**P-8.69) Hint** You don't have to use the drawing algorithm presented in the book.

**P-8.70) Hint** You can use a generalization of the binary-tree drawing algorithm presented in the book.

**P-8.71) Hint** We are considering the tree of all descendants here.

**P-8.72) Hint** Use the drawing style presented in the book.