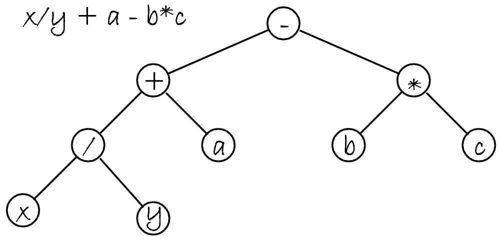
6.1.1

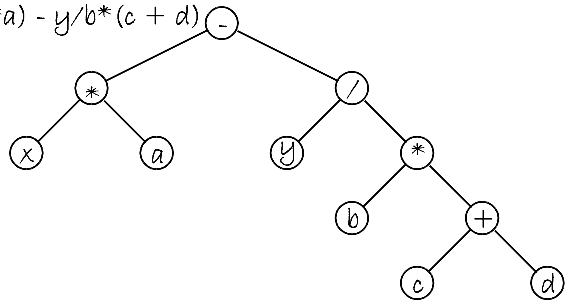
Draw binary expression trees for the following infix expressions. Your trees should enforce the Java rules for operator evaluation (higher-precedence operators before lower-precedence operators and left associativity).

a. `x / y + a – b * c`
b. `(x * a) – y / b * (c + d)`
c. `(x + (a * (b – c))/d`

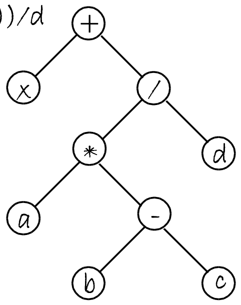
x/y + a - b*c



(x*a) - y/b*(c + d)

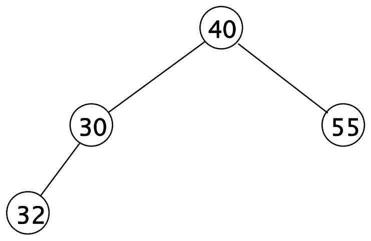


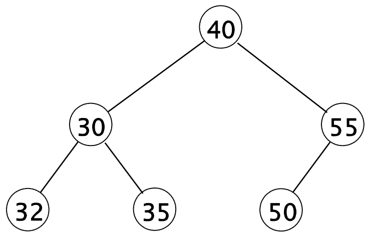
(x + (a*(b - c))/d



6.1.3

For each tree shown below, answer these questions. What is its height? Is it a full tree? Is it a complete tree? Is it a binary search tree? If not, make it a binary search tree.





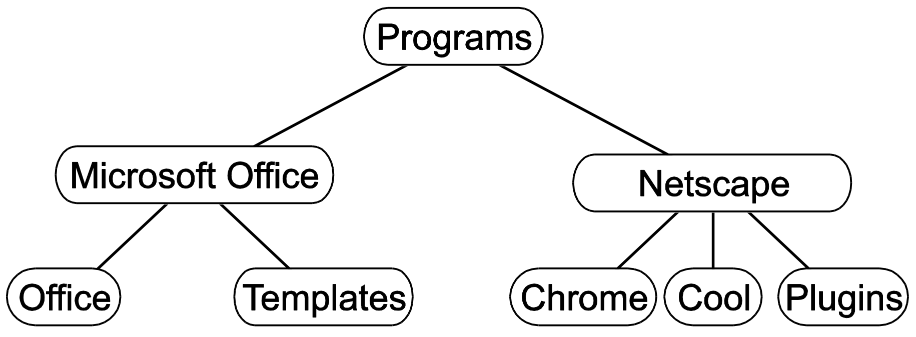
Height is 3
Not a full tree

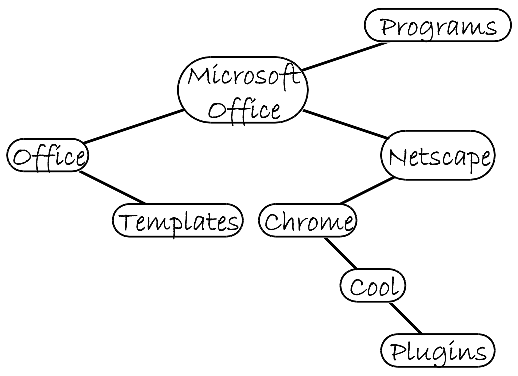
Height is 3
Not a full tree

Is a complete tree	Is a complete tree
Not a binary search tree	Not a binary search tree

6.1.5

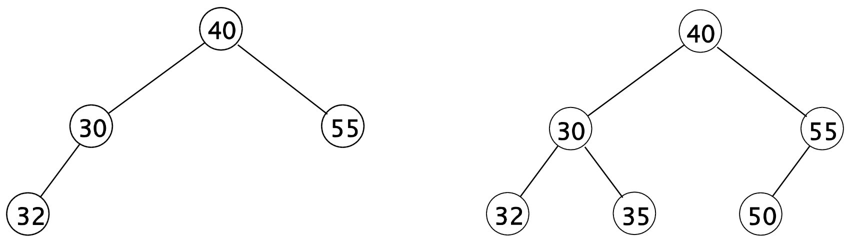
Represent the general tree in Figure 6.1 as a binary tree.





6.2.1

For the following trees:



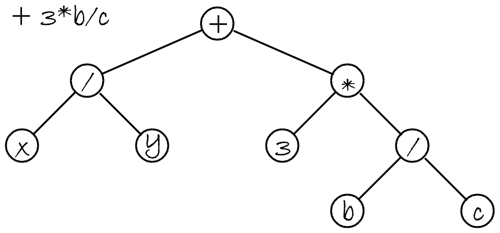
If visiting a node displays the integer value stored, show the inorder, preorder, and postorder traversal of each tree.

In: 32 30 40 55	32 30 35 40 50 55
Pre: 40 30 32 55	40 30 32 35 55 50
Post: 32 30 55 40	32 35 30 50 55 40

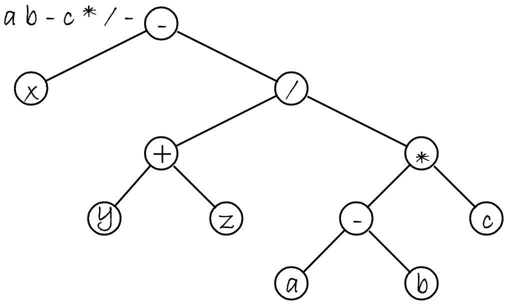
6.2.3

Draw an expression tree corresponding to each of the following:
a. Inorder traversal is `x / y + 3 * b / c` (Your tree should represent the Java meaning of the expression.)
b. Postorder traversal is `x y z + a b - c * / -`
c. Preorder traversal is `* + a - x y / c d`

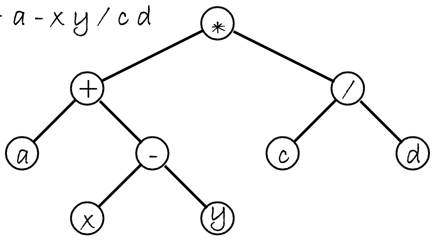
x/y + 3*b/c



x y z + a b - c * / -



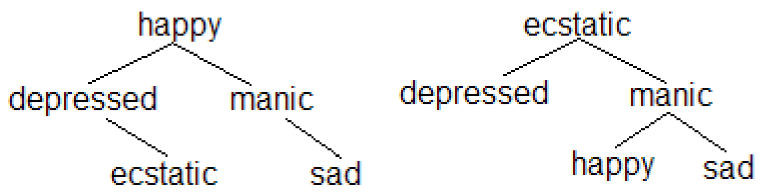
* + a - x y / c d



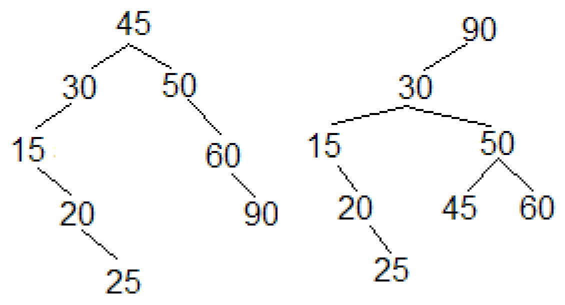
6.4.1

Show the tree that would be formed for the following data items. Exchange the first and last items in each list, and rebuild the tree that would be formed if the items were inserted in the new order.
a. happy, depressed, manic, sad, ecstatic
b. 45, 30, 15, 50, 60, 20, 25, 90

a.



b.



6.4.3

Show or explain the effect of removing the nodes *kept, cow* from the tree in Figure 6.13.

Removing node "kept" will have no other effect, as the node has no children. Node "killed" now has a left child of null. Removal of node "is" will result in the child node "in" moving up into its spot, and that is all. Node "jack" now has the left child of "in". Removal of node "cow" will result in node "cow" being replaced by the largest item in its left sub-tree, namely node "corn". Node "corn" now has a left child of "build" and a right child of "dog". Node "cock" now has a right child of null.

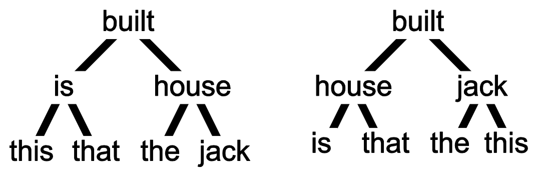
6.4.5

The algorithm for deleting a node does not explicitly test for the situation where the node being deleted has no children. Explain why this is not necessary.

Because the situation of a node having no children is covered by the first condition. If the node has no children, we need not find a replacement, i.e. replace this node with null. If the left child is null, we are to replace the node with it's right child. If the node has no children, the right child is null. We replace the node with null, and that is what we needed to do.

6.5.1

Show the heap that would be used to store the words *this, is, the, house, that, jack, built,* assuming they are inserted in that sequence. Exchange the order of arrival of the first and last words and build the new heap.



6.5.3

Show the result of removing the number 18 from the heap in Figure 6.26. Show the new heap and its array representation.

0	1	2	3	4	5	6	7	8	9	10	11
8	20	29	26	28	39	66	37	89	76	32	74

6.5.5

Show the printer queue after receipt of the following documents.

time stamp	size
1100	256
1101	512
1102	64
1103	96

time stamp	size
1102	64
1103	96
1100	256
1101	512

6.6.1

What is the Huffman code for the letters *a, j, k, l, s, t, v* using Figure 6.35?

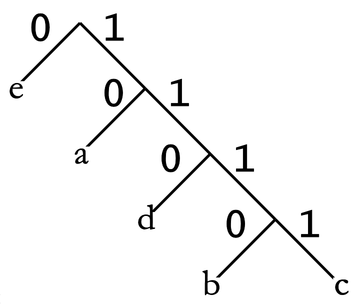
a 1010, j 1100001011, k 11000011, l 10111, s 0011, t 1101, v 1100000

6.6.3

Trace the execution of method `decode` for the Huffman tree in Figure 6.37 and the encoded message string "11101011011001111".

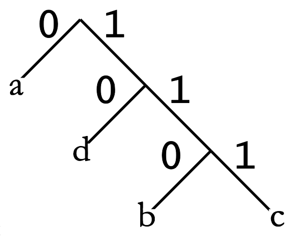
decode("11101011011001111")

result: ""



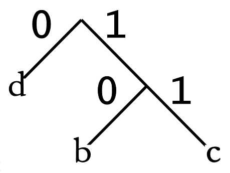
currentTree:

codedMessage.charAt(0) == 1



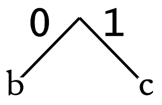
currentTree:

codedMessage.charAt(1) == 1



currentTree:

codedMessage.charAt(2) == 1

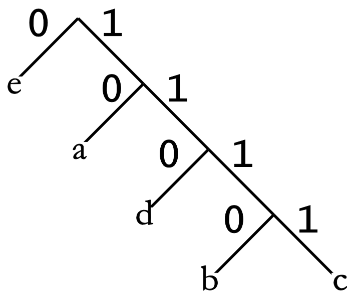


currentTree:

codedMessage.charAt(3) == 0

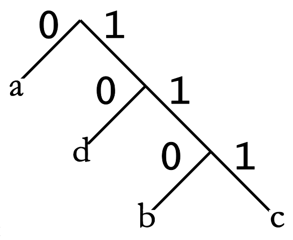
currentTree: b

result "b"



currentTree:

codedMessage.charAt(4) == 1

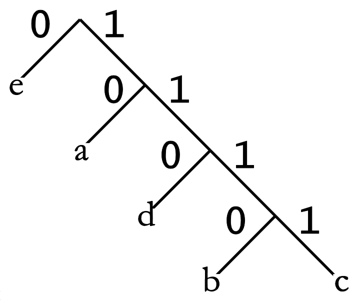


currentTree:

codedMessage.charAt(5) == 0

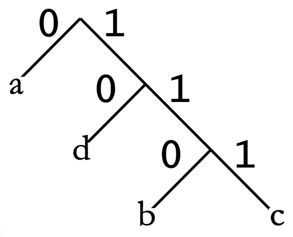
currentTree: a

result: "ba"



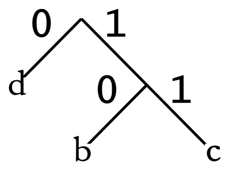
currentTree:

codedMessage.charAt(6) == 1



currentTree:

codedMessage.charAt(7) == 1

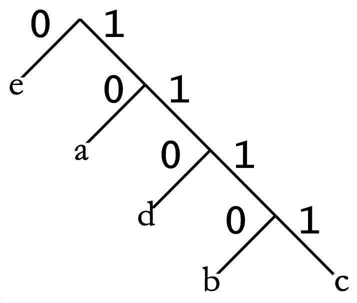


currentTree:

codedMessage.charAt(8) == 0

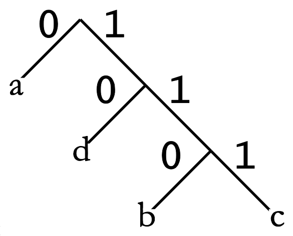
currentTree d

result: "bad"



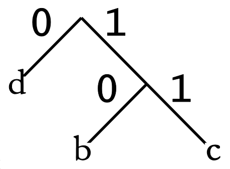
currentTree:

codedMessage.charAt(9) == 1



currentTree:

codedMessage.charAt(10) == 1

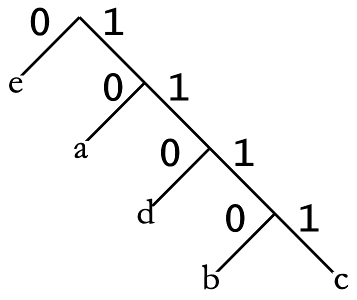


currentTree:

codedMessage.charAt(11) == 0

currentTree d

result: "badd"

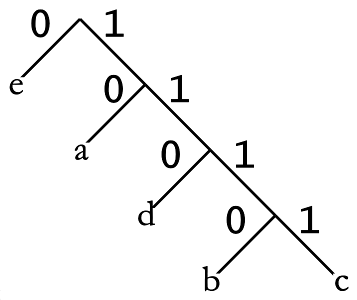


currentTree:

codedMessage.charAt(11) == 0

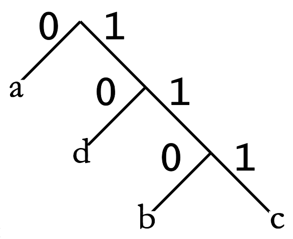
currentTree: e

result: "badde"



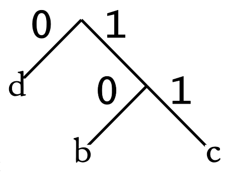
currentTree:

codedMessage.charAt(13) == 1



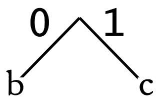
currentTree:

codedMessage.charAt(14) == 1



currentTree:

codedMessage.charAt(15) == 1



currentTree:

codedMessage.charAt(16) == 1

currentTree: c

result "baddec"

6.6.5

What would the Huffman code look like if all symbols in the alphabet had equal frequency?

The code length for each symbol would be the same length or one less than the length. It would be simple binary code with the length of $\log_2 n$ where n was the number of symbols.