600.226 Data Structures – Spring 2004 – Prof. Jason Eisner

Homework 9 – Due 6 pm Friday April 16, 2004

*NAME:*_____ *EMAIL:*_____

Summary:

Goal of this homework: Become comfortable with balancing search trees and sorting sequences, without having to write any actual code. Note that questions like these are fair game on the final exam, so you may want to play with more examples on your own.

Collaboration policy for this homework: You must follow the CS Dept. Academic Integrity Code, and this time may not collaborate with another student, although you may work together to understand the textbook material in a general way.

How to hand in your work: Slip it under Prof. Eisner's office door (NEB 324A). Don't forget to put your name at the top!

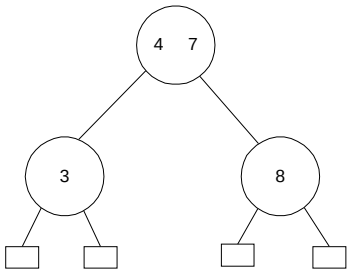
What to hand in: A well-stapled-together printout of this handout (double-sided if at all possible), with your name and answers written clearly on it.

Acknowledgments: Thanks to Usman Zaheer for turning this assignment into a nice-looking handout.

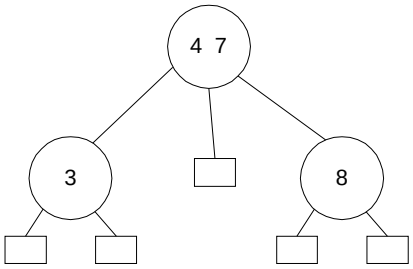
HINT: We spent several lectures developing and experimenting with these algorithms, so you should already have a fairly good understanding of them. Online animations of the algorithms are available from the course homepage (see the links in the "Resources" column of the schedule grid), and may help you think about what is going on. To review and solidify your understanding, chapters 9-10 of the textbook are your friends.

1. Some of these trees are valid (2,4) trees (i.e., multi-way search trees that satisfy the size and depth invariants). A square box represents a missing child subtree (i.e., null instead of a node). Circle the valid (2,4) trees and cross out the others.

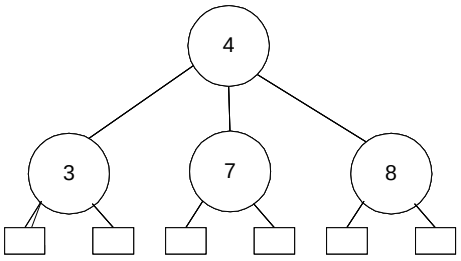
a)



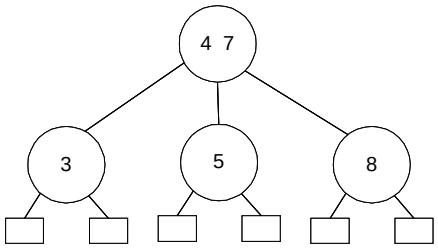
b)



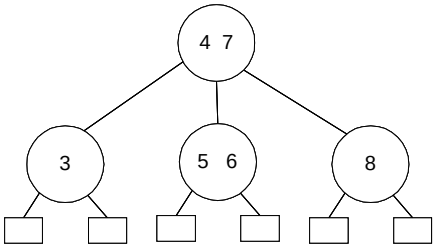
c)



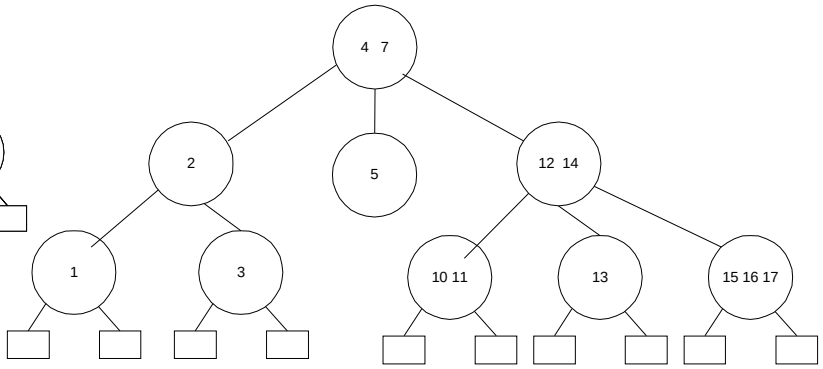
d)



e)

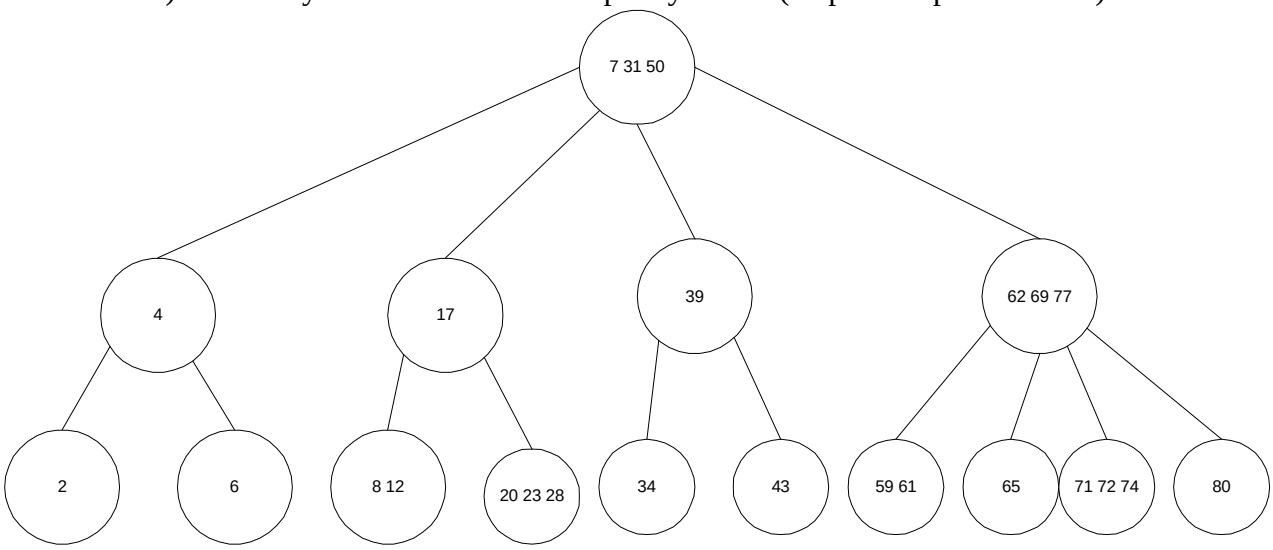


f)

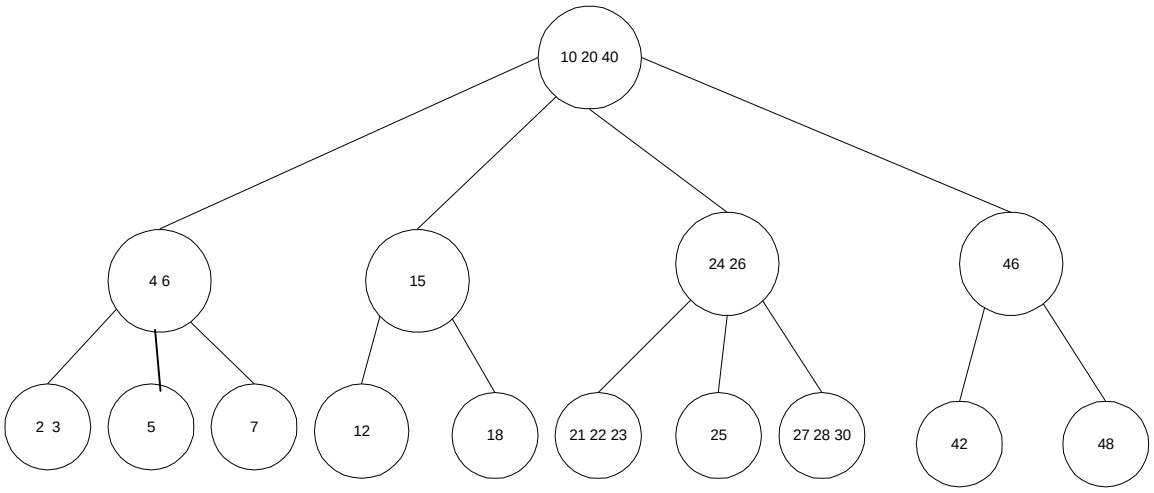


Note: In this and following examples, null subtrees of a (2,4) tree are not shown (i.e., the square boxes are left out).

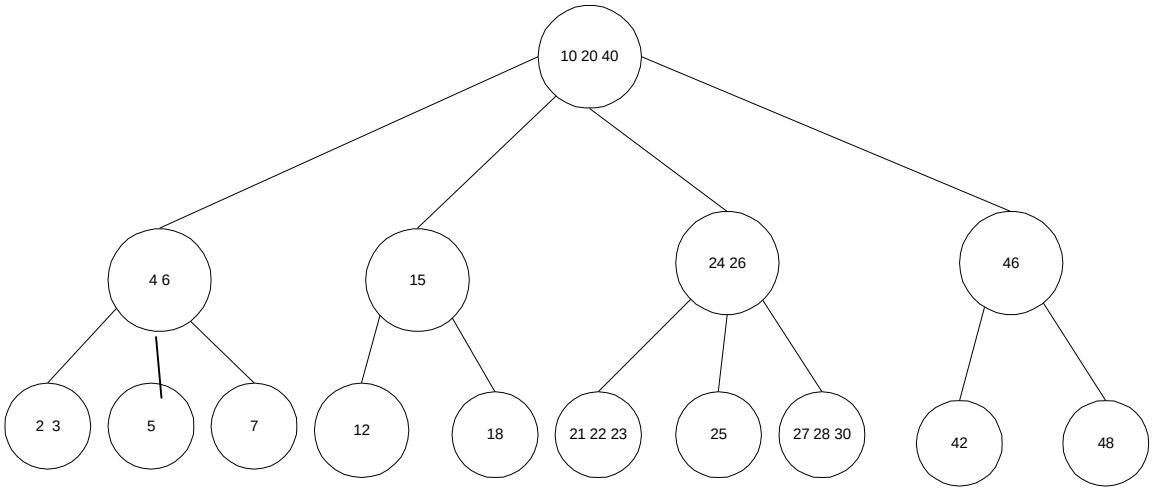
2. Show what this (2,4) tree looks like after inserting 13, 25, 15 and 73 (in that order). You may show intermediate steps if you like (helpful for partial credit).



3. Show what this (2,4) tree looks like after removing keys 2, 5, 40 and 25 (in that order). Remember the trick for removing an internal key by first swapping it with another key.



4. In the same (2,4) tree, shown again below, clearly explain exactly what happens, and why, when key 15 is removed. Clearly show all intermediate steps. No shortcuts—do what the computer would do.



5.

a) What is the maximum number of keys than can be held in a (2,4) tree of height h? Check your answer for small values of h. It should be an exact formula (do not use big-Oh).

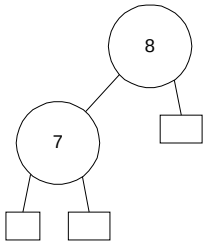
b) Same question, as (a), but with a minimum number of keys.

c) Use your answers, in part (a) and (b), to prove that h is O (log n), where n is the total number of keys in the (2,4) tree (Hint: Section 9.2 of the textbook proves this for AVL trees, which is harder).

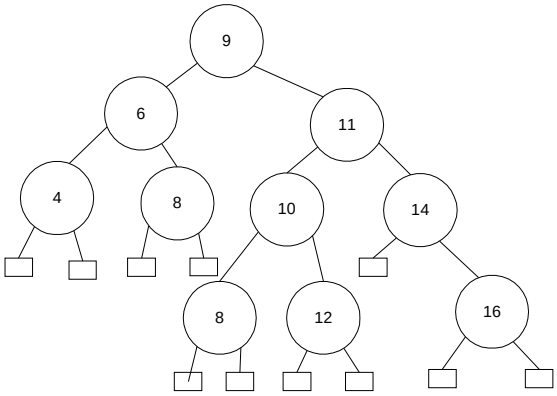
6. Give an example of a (2,4) tree and a key whose removal will change the height of the tree. Explain and show exactly what happens when this key is removed.

7. Some of these trees are valid AVL trees (i.e., binary search trees that satisfy the height-balance invariant). A square box represents a missing child subtree (i.e., null instead of a node). Circle the valid AVL trees and cross out the others. You may want to mark the height of each node.

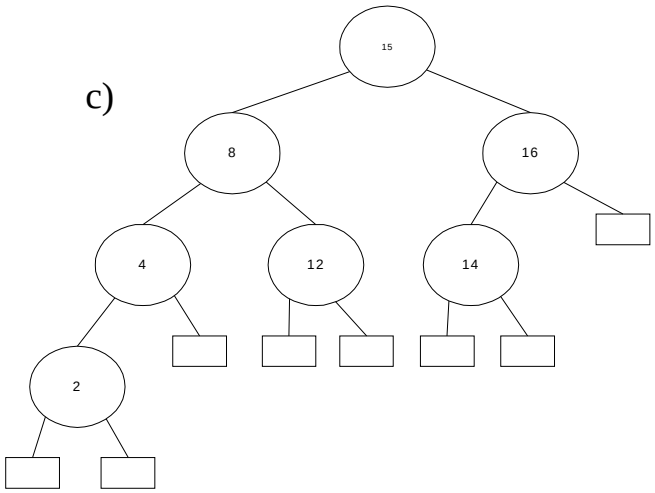
a)



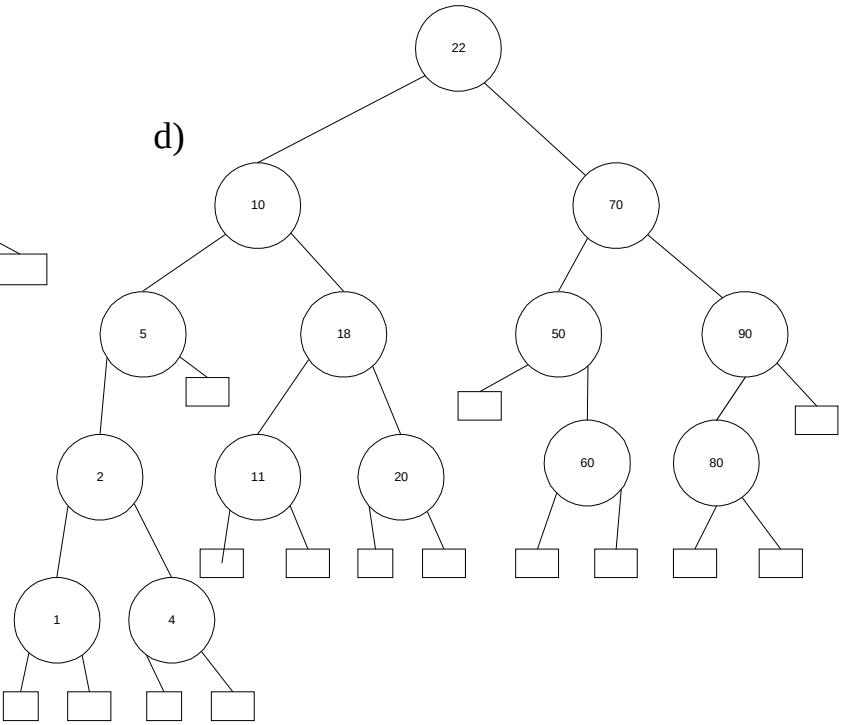
b)



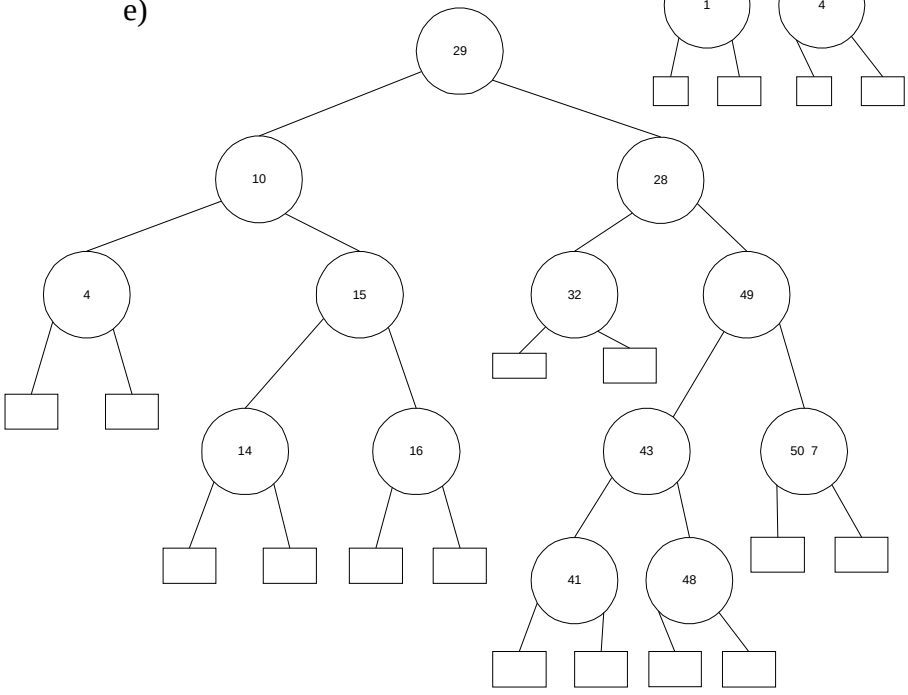
c)



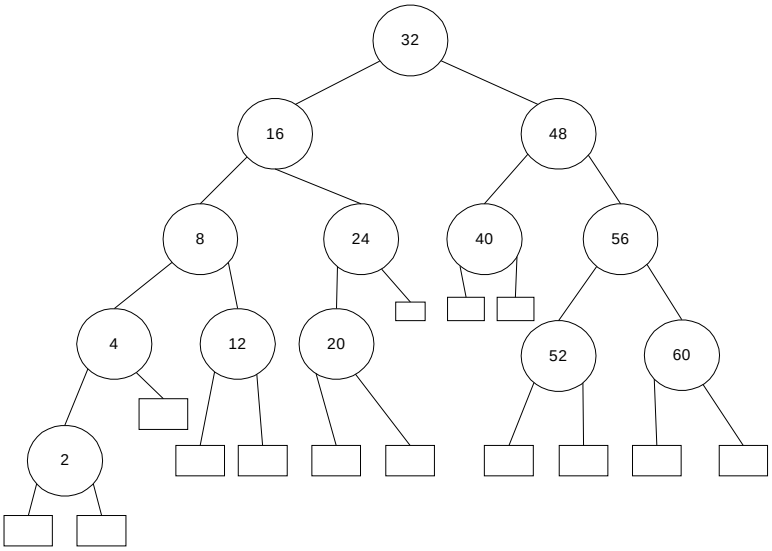
d)



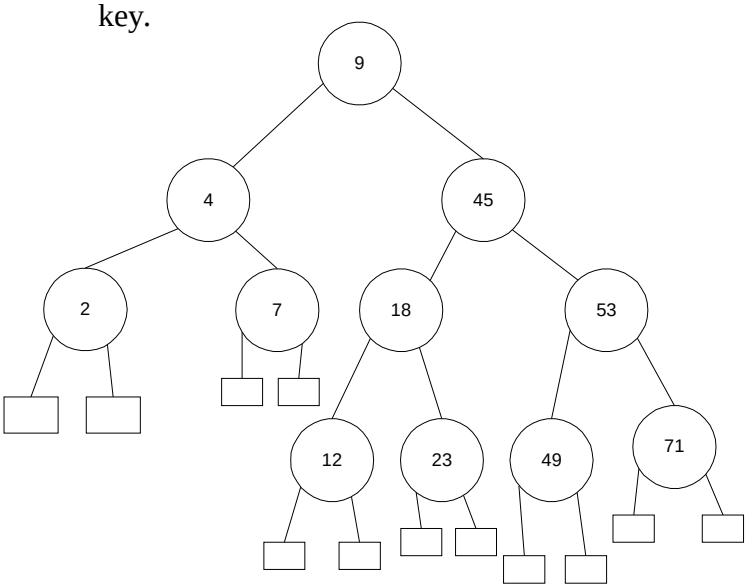
e)



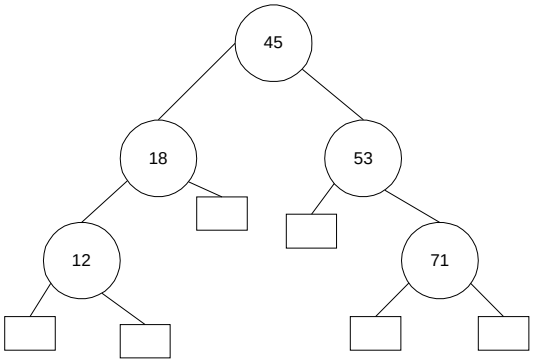
8. Show what this AVL tree looks like after 6, 5, and 54 have been inserted (in that order). You may show intermediate steps if you like (helpful for partial credit). Hint: during restructuring, remember that a single rotation gathers up 2 keys into a multiway node before splitting them again; a double rotation gathers up 3 keys.



9. Show what this AVL tree looks like after removing nodes 7 and 45 (in that order). Remember the trick for removing an internal key by first swapping it with another key.

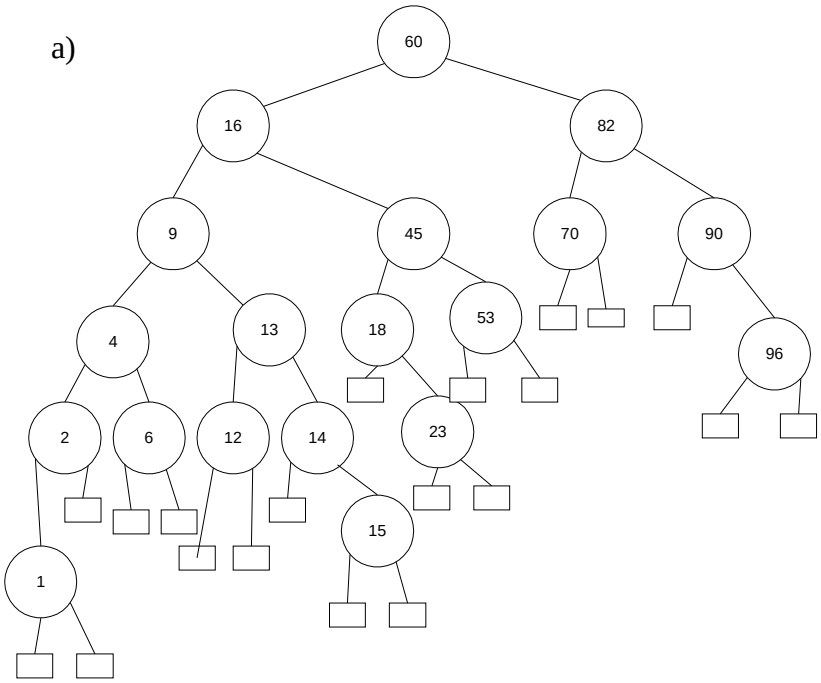


10. Show what this AVL tree looks like after removing node 45 (See figures 9.5-9.6 of the textbook).

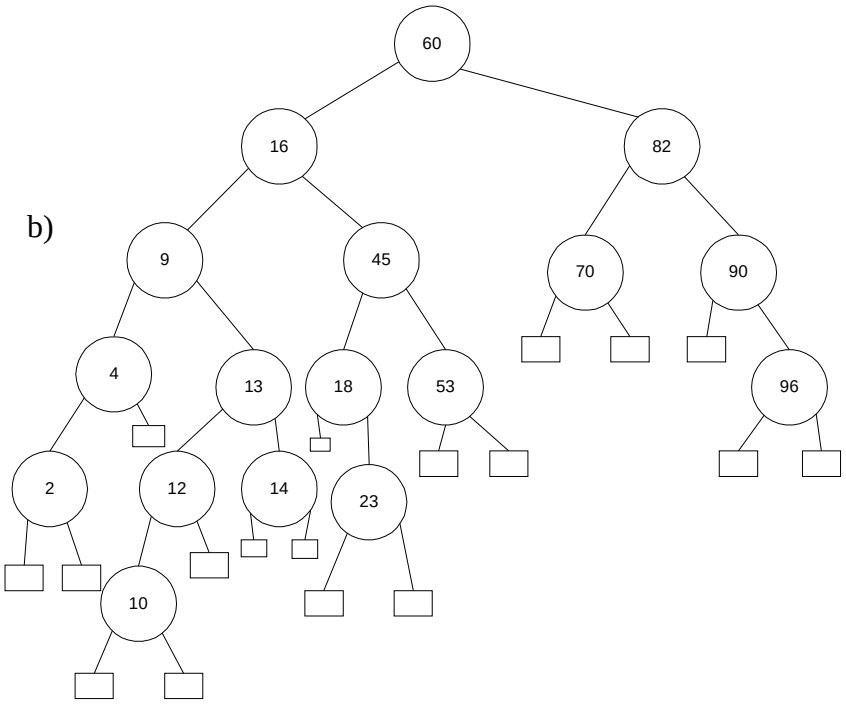


11. These two trees (a) and (b) are very similar. They are not AVL trees because the root node is unbalanced; fix them by adding some more descendants (as few as possible!) to the node with key 82. Then in each case, show what the tree looks like after removing the node with key 53.

a)



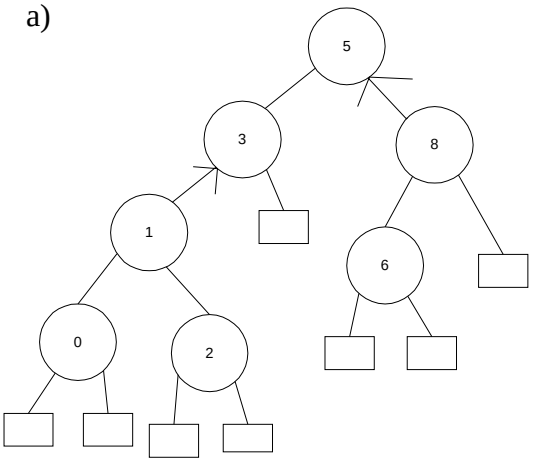
b)



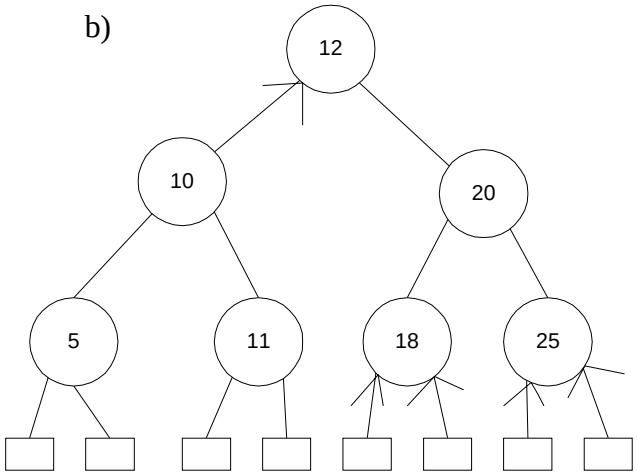
12. Answer question R-9.4 from the book. Try to give the smallest possible example.

13. Some of the following trees are valid red-black trees. Circle them and cross out the others. (Note: Red nodes are drawn here with an arrow pointing up to the parent node. This notation reminds you that the red node would be absorbed into the parent in the corresponding (2,4) tree.)

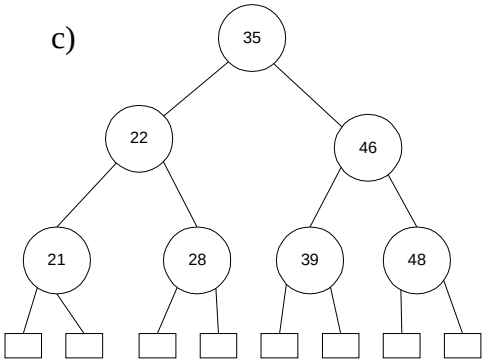
a)



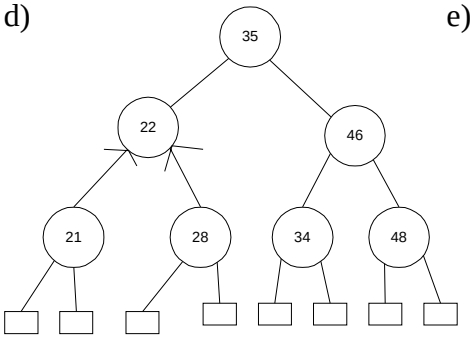
b)



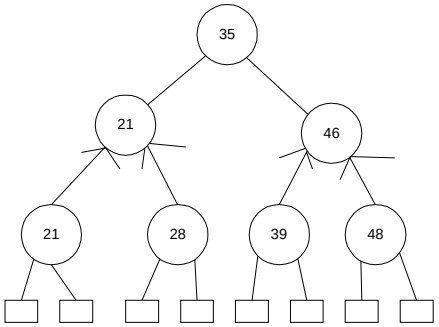
c)



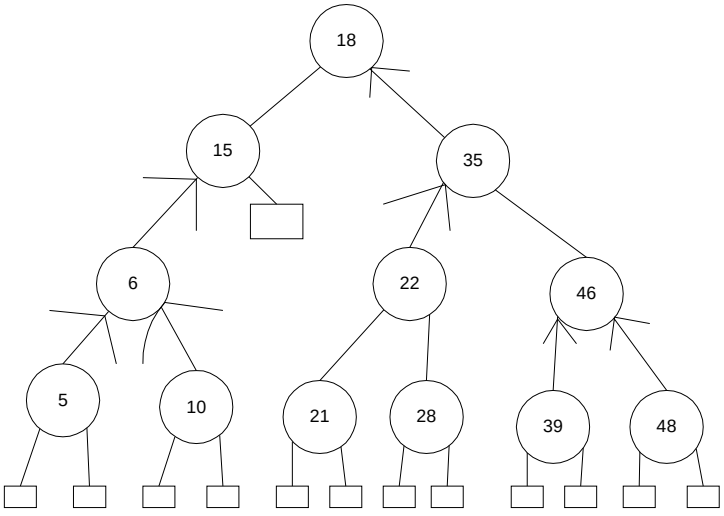
d)



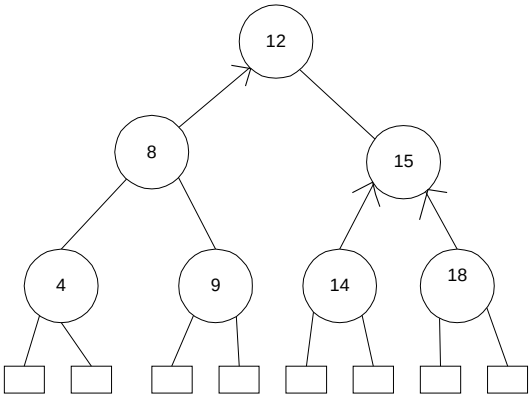
e)



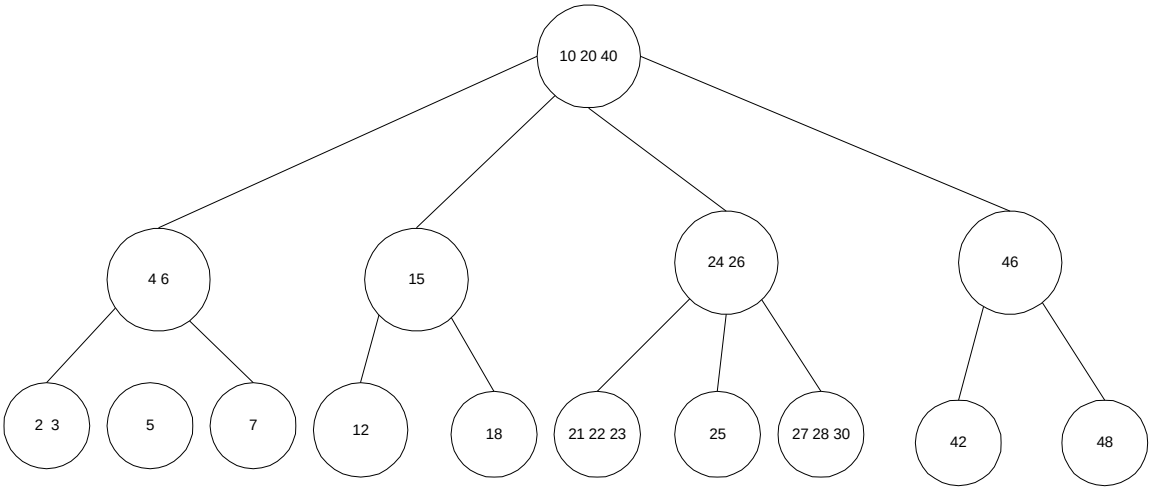
f)



14. Draw the (2,4) tree corresponding to this red-black tree:



15. Draw the red-black tree corresponding to the (2,4) tree shown in question 3, repeated below.



16. Draw what the red-black tree of question 14 looks like after inserting 5, 6 and 17 (in that order). Show the corresponding (2,4) tree as well.

17. Draw what the red-black tree of question 14 looks like after inserting 17, 19 and 20 (in that order). Do not include your changes from question 16. Show the corresponding (2,4) tree as well.

18. Draw what the red-black tree of question 14 looks like after inserting 5, 6 and 7 (in that order). Do not include your changes from question 16 or 17. Show the corresponding (2,4) tree as well.

Read the textbook discussion of removal from red-black trees, and explicitly study Figs. 9.37-9.38 [or in the 2nd edition, 9.30-9.31].

19. Can the double red problem appear temporarily while *removing* a key from a red-black tree? Justify your answer.

20. Figures 9.33 through 9.36 in the textbook [or in the 2nd edition, 9.26 through 9.29] each show a local change to a red-black tree. Do any of these local changes affect the black depth of any external node? Justify your answer carefully. (Hint: A red, black, or double-black node has black depth that is respectively 0, 1, or 2 more than its parent's black depth.)

21. In figures 9.37–9.38 [or in the 2nd edition, 9.30-9.31] the black depth of the external nodes stays at 2 throughout. In fact the book never explains how the black depth can ever get reduced. We'll explore that case in this question.

a) Continue from the end of Fig. 9.38 [or in the 2nd edition, 9.31], showing what happens in the red-black tree and the corresponding (2,4) tree when you now remove key 4. Is this case 1, 2 or 3 of removal?

b) Now remove key 14 and again show what happens. Is this case 1, 2 or 3 of removal?

c) According to your answer in question 20, what is the black depth of the external nodes now? Is this really the depth? And, does it correspond as usual to the height of the (2,4) tree? Explain the apparent paradox.

22. Show what this array looks like after each major step of in-place selection sort:

17 3 4 9 18 26 8 6 15

23. Show what the same array looks like after each major steps of in-place insertion sort:

17 3 4 9 18 26 8 6 15

24. Same problem for in-place quick-sort (section 10.3.1) on the same array. Use 18 as your first pivot, in subsequent steps; state the random pivot that you are using.

17 3 4 9 18 26 8 6 15

25. Same as question 24, but this time be the adversary: Choose the worst possible sequence of pivots. What does the result tell you about the worst-case runtime of quick-sort?

17 3 4 9 18 26 8 6 15

26. Show the major steps of merge-sort on the same array. You do not have to sort in-place. When you are dealing with a sub-array of odd size, split it as evenly as possible, with the left half of the array being one element longer than the right half.

17 3 4 9 18 26 8 6 15

27. The textbook's skip list algorithms (Chapter 8) are needlessly complicated, because they are written iteratively. Write pseudocode that solves the problem recursively instead.

The trick is to write a recursive private helper method SkipSearch(k, smaller, bigger). In general, SkipSearch(k, smaller, bigger) should look for k between the tower under smaller and the tower under bigger. It can be called whenever smaller and bigger are positions in the same level of the skip list, and it is known that key(smaller) $\leq$ k < key(bigger).

The public method SkipSearch(k) can simply call SkipSearch(k, topleft, topright), where topleft and topright are the $-\infty$ and ∞ positions in the top level of the skip list. (Note the overloading here: there are two *different* methods called SkipSearch. They have different numbers of arguments, and the non-recursive one calls the recursive one.)

Note that it is unnecessary for SkipSearch(k, smaller, bigger) to compare k with key(smaller) or key(bigger), since the caller is supposed to guarantee already that key(smaller) $\leq$ k < key(bigger). For full efficiency and full credit, your pseudocode (unlike the book's) should avoid doing both of those unnecessary comparisons.

28. Now write recursive pseudocode for SkipInsert(k,e). Your pseudocode should be correct, elegant and efficient by the time you hand it in. You should not call SkipSearch or the textbook's InsertAfterAbove. The trick is to call a recursive private method, much as before. This recursive method should return the position of the new node that it inserted somewhere between positions smaller and bigger, or null if it did not insert such a node on that level.

    Note: You should not need the before() or above() methods (as in exercise C-8.9 [C-8.8 in the 2nd edition]). This saves space, since it allows us to represent the skip list using only after and below pointers: each row is only a singly linked list, as is each column.

29. **Extra credit** (continue answer on reverse side if necessary)**:** Your SkipSearch and SkipInsert methods should look somewhat alike, since a new key is inserted in exactly the place where search will find it. How could you avoid the duplicate code while still using a nice recursive design? Hint: Both methods could call the same private method …