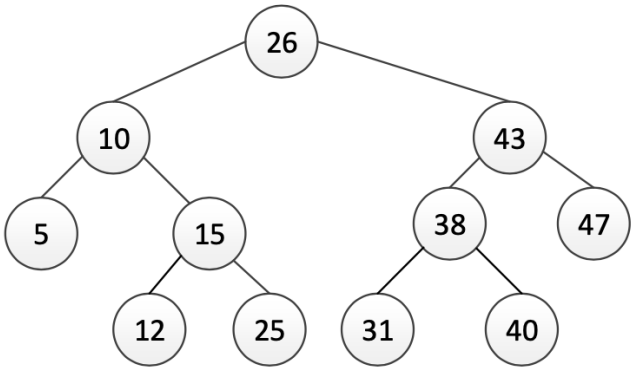
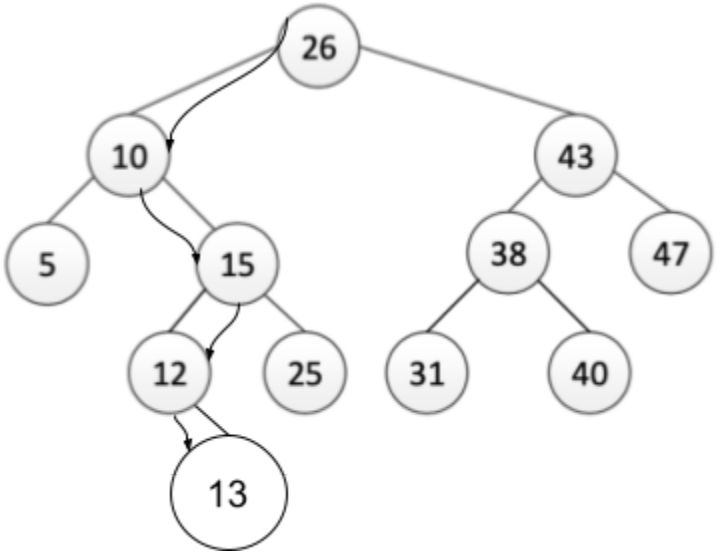
Problem 1 (10 points). Consider the following binary search tree:

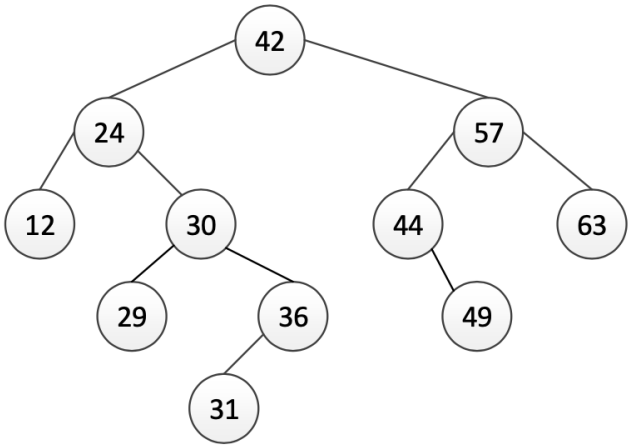


Show the resulting tree if you add the entry with key = 13 to the above tree. You need to describe, step by step, how the resulting tree is generated.

When we are inserting an entry, we first perform a search operation. If there is no entry with the key, then we add an entry at the leaf node where the unsuccessful search ended up.



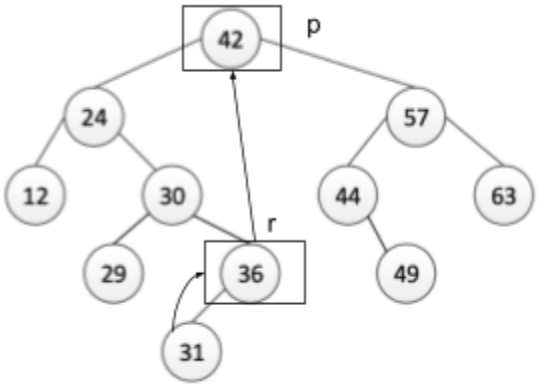
Problem 2 (10 points). Consider the following binary search tree:

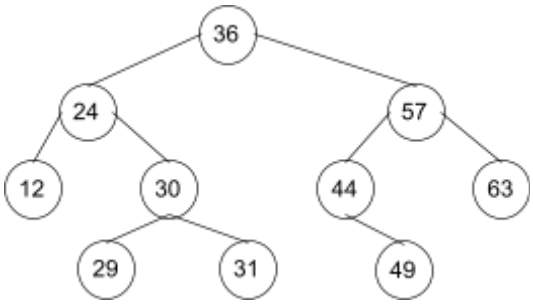


Show the resulting tree if you delete the entry with key = 42 from the above tree. You need to describe, step by step, how the resulting tree is generated.

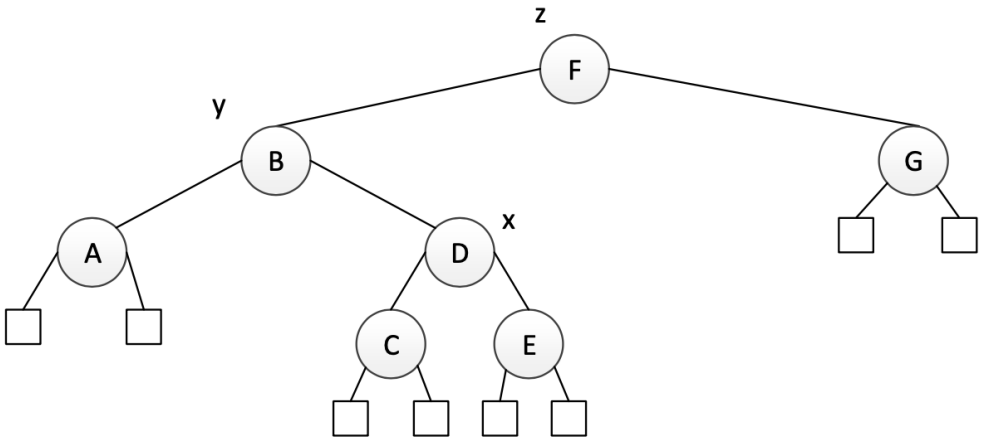
When we are deleting an entry, we first perform a search operation. If there is no entry with the key, If we reach a leaf node, then there is no node with that key, so nothing needs to be done. If not, we need to consider the following two cases. Let be the node to be deleted.

1. In the first case, at most one child of p is an internal node.
2. In the second case, p has two children, both which are internal

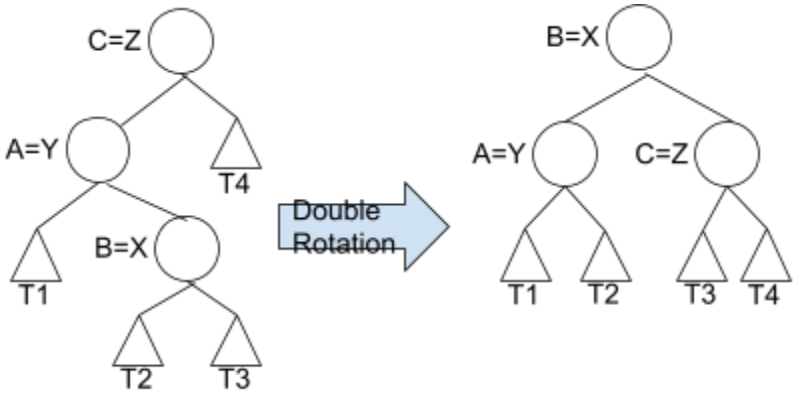


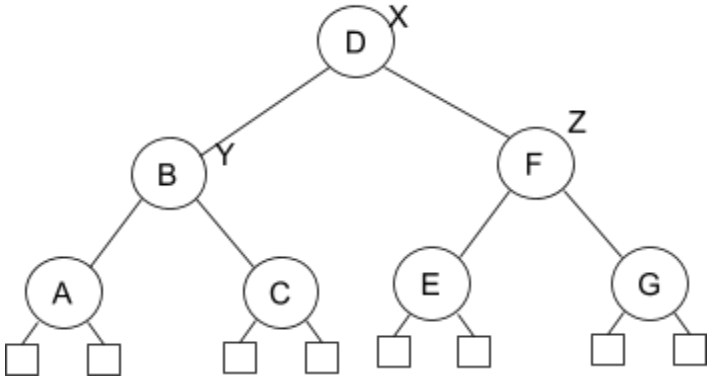


Problem 3 (10 points). Consider the following AVL tree, which is unbalanced:

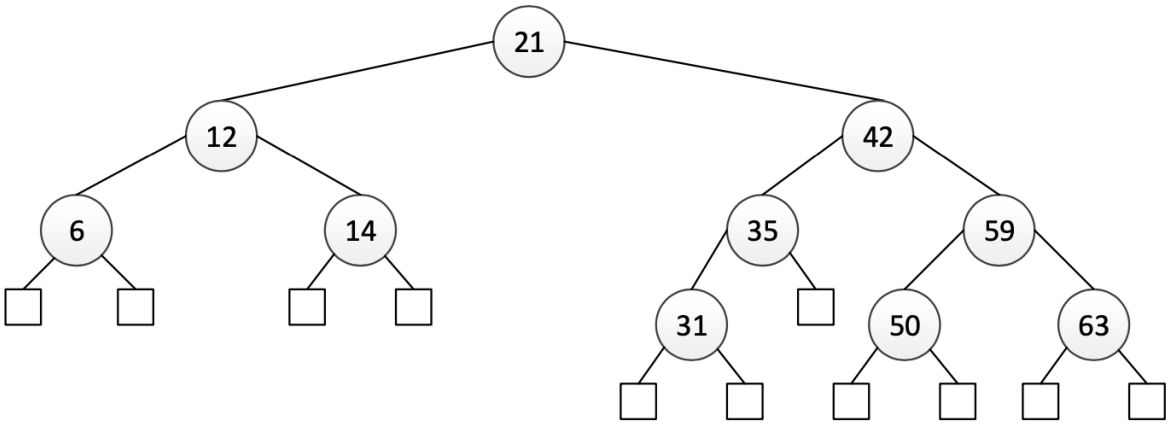


Note that the nodes *F*, *B* and *D* are labeled *z*, *y*, and *x*, respectively, following the notational convention used in the textbook. Apply a trinode restructuring on the tree and show the resulting, balanced tree.

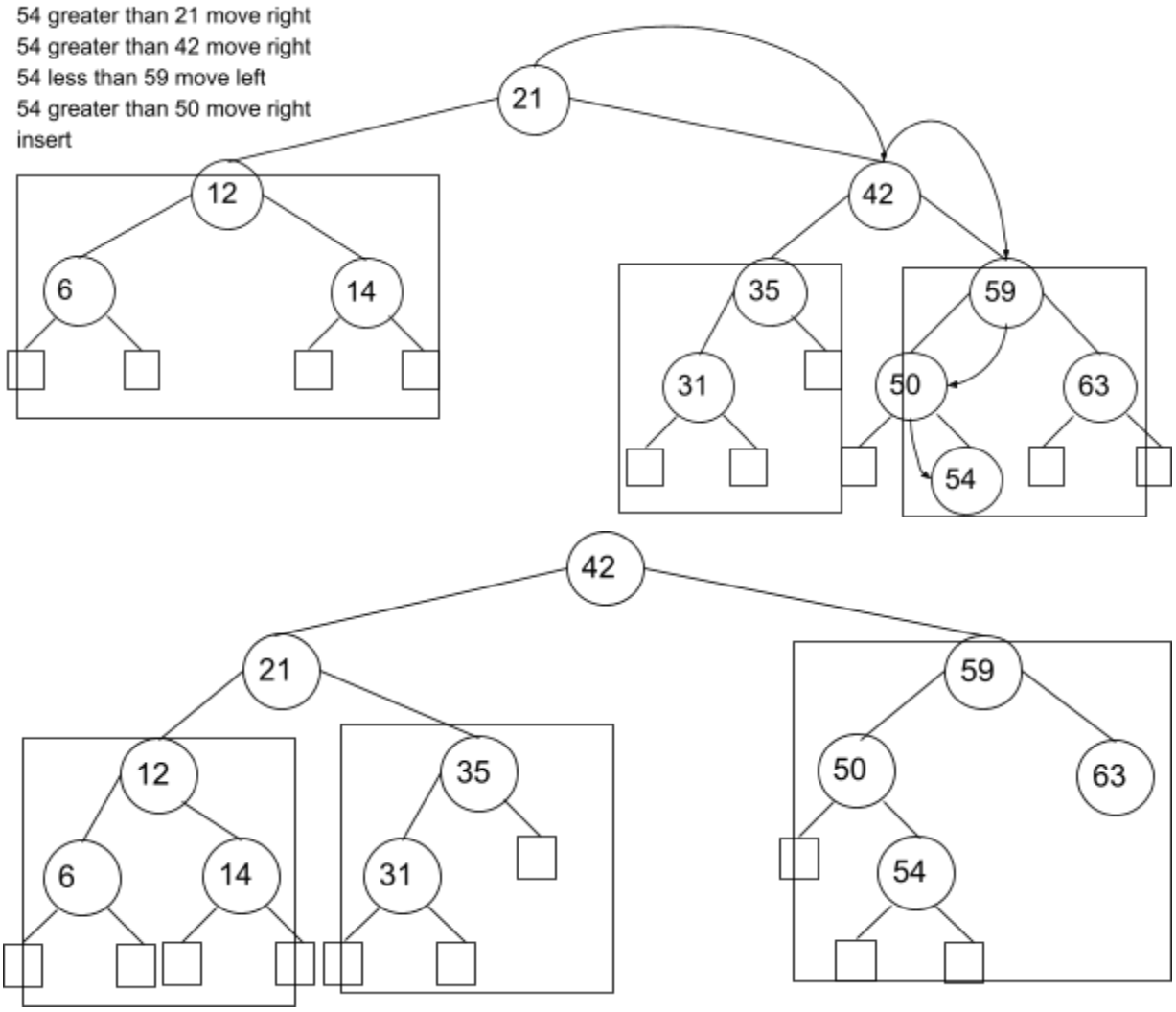




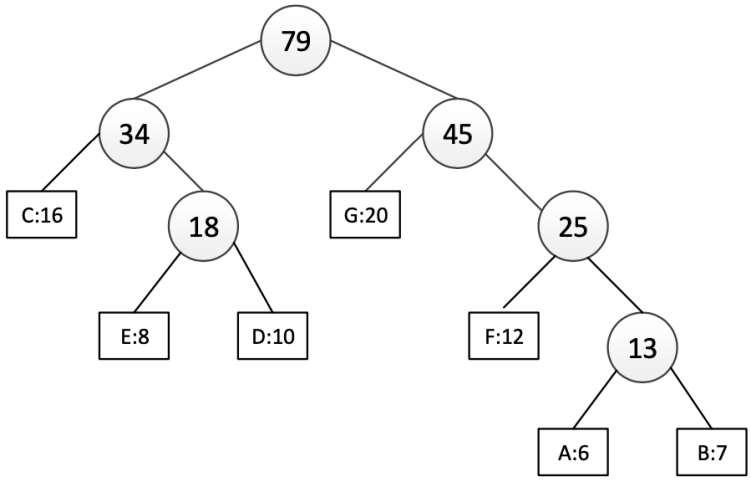
Problem 4 (10 points). Consider the following AVL tree.



Show the resulting, balanced tree after inserting an entry with key = 54 to the above tree. You must describe, step by step, how the resulting tree is obtained.



Problem 5 (10 points). Consider the following Huffman tree:



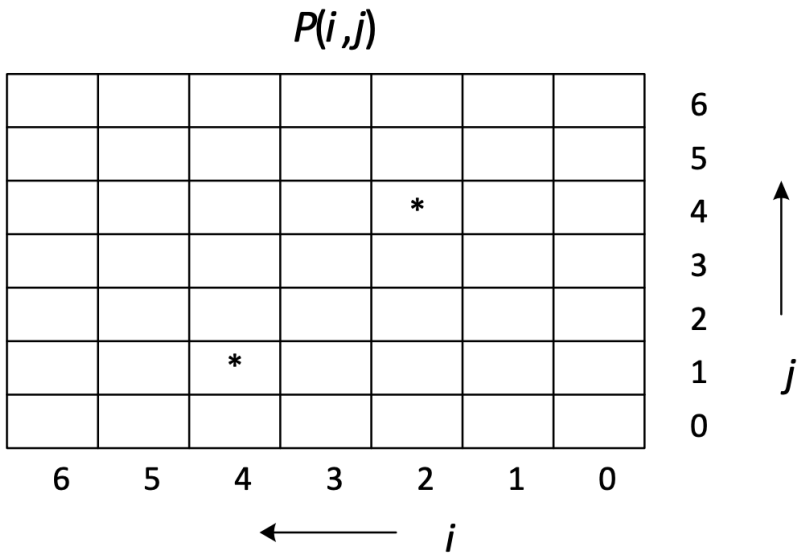
(1) Encode the string "BDEGC" to a bit pattern using the Huffman tree.

(2) Decode the bit pattern "010111010011" to the original string using the Huffman tree.

1.
B: 1111
D: 011
E: 010
G: 10
C: 00
11110110101000

2.
010: E
1110: A
10: G
011: D
EAGD

Problem 6 (10 points). This question is about the *World Series* problem that we discussed in the class. The following is the probability matrix for the problem.



Calculate the probabilities of $P(4, 1)$ and $P(2, 4)$, which are marked with *. You must show all intermediate steps and calculations.

$$p(i,j) = 1, \; if \; i = 0 \; and \; j > 0$$
$$= 0, \; if \; i > 0 \; and \; j = 0$$
$$= (p(i-1, j) + p(i, j-1))/2, \; if \; i > 0 \; and \; j > 0$$

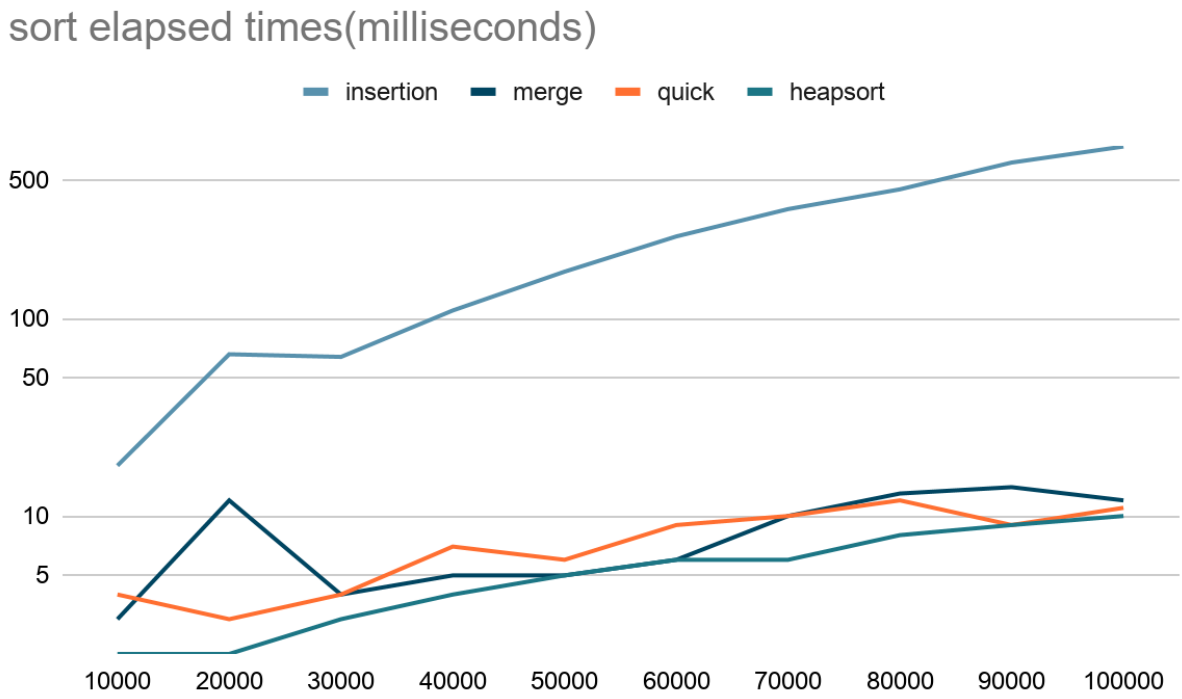
P(4,1):
P(4,1) = (P(3,1) + P(4,0))/2 = (? + 0)/2 = 1/16
P(3,1) = (P(2,1) + P(3,0))/2 = (? + 0)/2 = 1/8
P(2,1) = (P(1,1) + P(2,0))/2 = (? + 0)/2 = 1/4
P(1,1) = (P(0,1) + P(1,0))/2 = (1 + 0)/2 = ½

P(2,4):
P(2,4) = (P(1,4) + P(2,3))/2 = (?(15/16) + ?(11/16))/2 = 13/16
P(1,4) = (P(0,4) + P(1,3))/2 = (1 + ?(7/8))/2 = 15/16
P(1,3) = (P(0,3) + P(1,2))/2 = (1 + ?(3/4))/2 = 7/8
P(1,2) = (P(0,2) + P(1,1))/2 = (1 + 1/2)/2 = ¾
P(2,3) = (P(1,3) + P(2,2))/2 = (7/8 + ?(1/2))/2 = 11/16
P(2,2) = (P(1,2) + P(2,1))/2 = (3/4 + 1/4)/2 = ½

Problem 7 (40 points).:

Algorithm/n	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
insertion	18	66	64	110	173	261	359	452	618	745
merge	3	12	4	5	5	6	10	13	14	12
quick	4	3	4	7	6	9	10	12	9	11
heapsort	2	2	3	4	5	6	6	8	9	10

[Finished in 3.7s]



Thought:
By just looking at the table and the chart i seems like that in general
insertion > merge > quick > heap time

For insertion test we have not tested any on small dataset, the smallest one is 10000 seemes like it's already too big for insertion which is uses O(N) or 0(N^2) but it seemes like that insertion time did not increase too dramatically N^2 it kind increated in a linear way, maybe the dataset number we have used are too close to each other

For merge sort the best case and worst case are all 0(N*log N) even it increases it will not increase too much. The above graph shows a great visual representation. But it does not seem to have a consistent speed on any size of data, maybe to speed of it is in between 3 & 14, but in the graph what i see is that the bigger the data is it take an average of more time, may be when we test more data it will give a better representation。

For quick sort the worst in O(N^2), and it works well in small arrays, but it seems like it's doing very fine in the range of 10000 - 100000 which have a similar run time as merge-sort O(n log n).

Heap-sort which runs the best here, it has the same run time as merge and quick O(n log n). But data range of 10000 - 100000 seems like it's not big enough for heap to show it disadvantage.