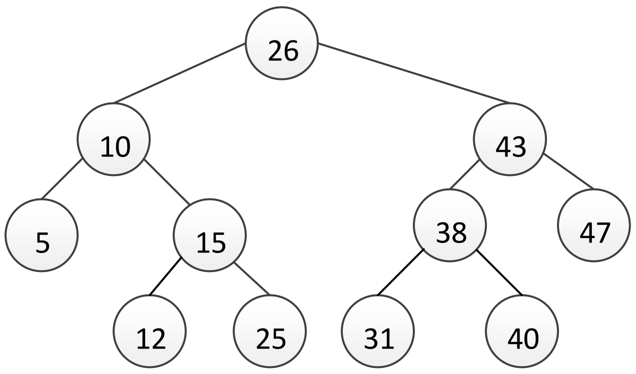
CS526 O2

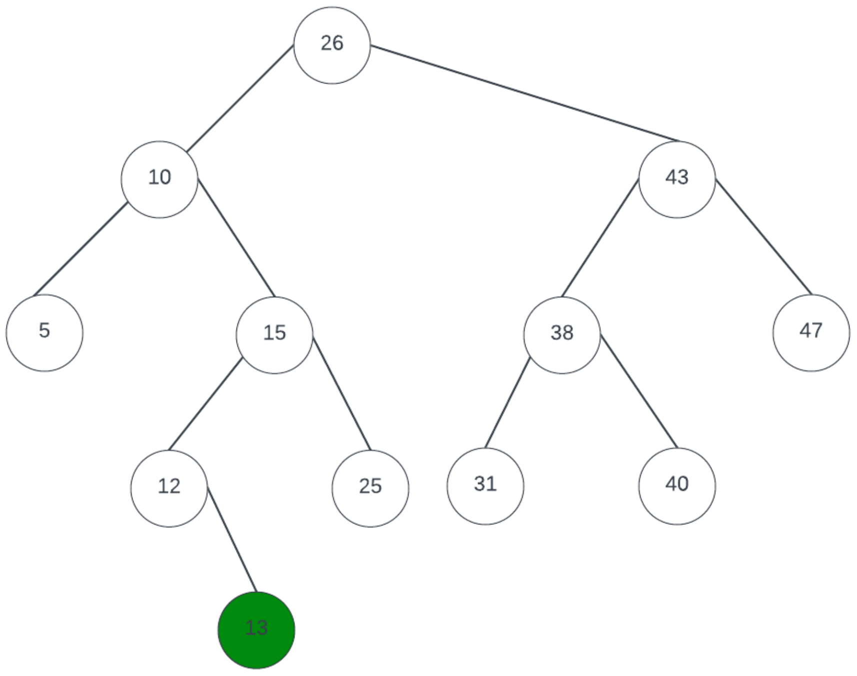
Homework Assignment 5

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

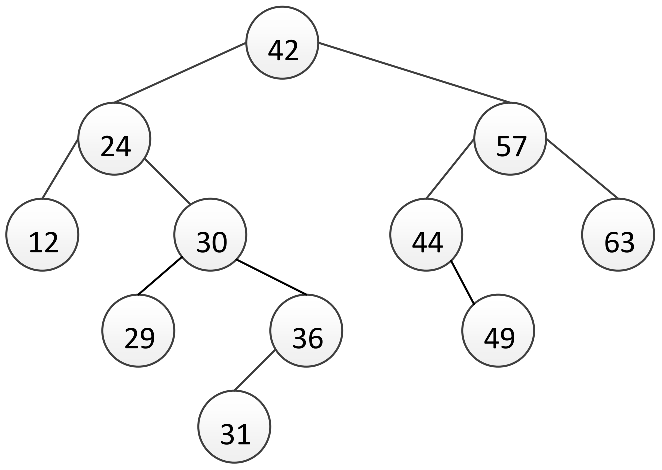


Show the resulting tree if you add the entry with key = 13 to the above tree. You need to describe, step by step, how the resulting tree is generated.

- When we add the entry with key = 13. There is no entry with key =13 in the above binary tree → an entry is added to the leaf node where unsuccessful search ends up.
- We start from the root, and each node we compare key =13 to each entry node's key.
- We have key = 13 < root with key = 26 → move left
- Key =13 > entry with key = 10 → move right
- Key = 13 < entry with key = 15 → move left
- Key =13 > entry with key = 12 (this is leaf node) → add entry with key = 13 as a right child of entry with key = 12.

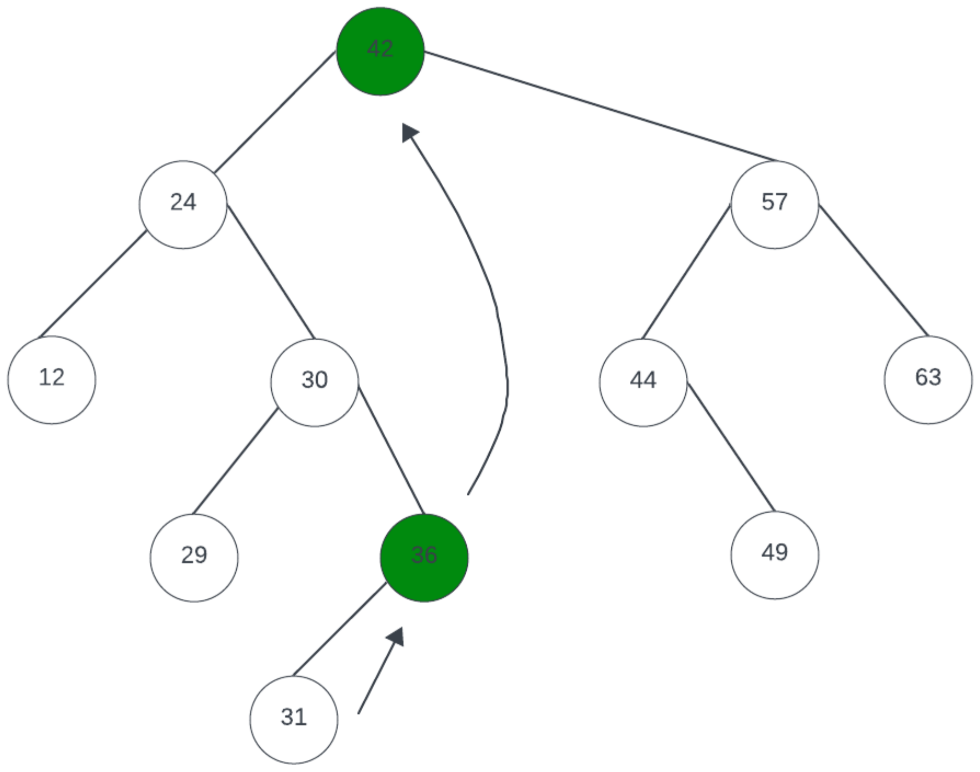


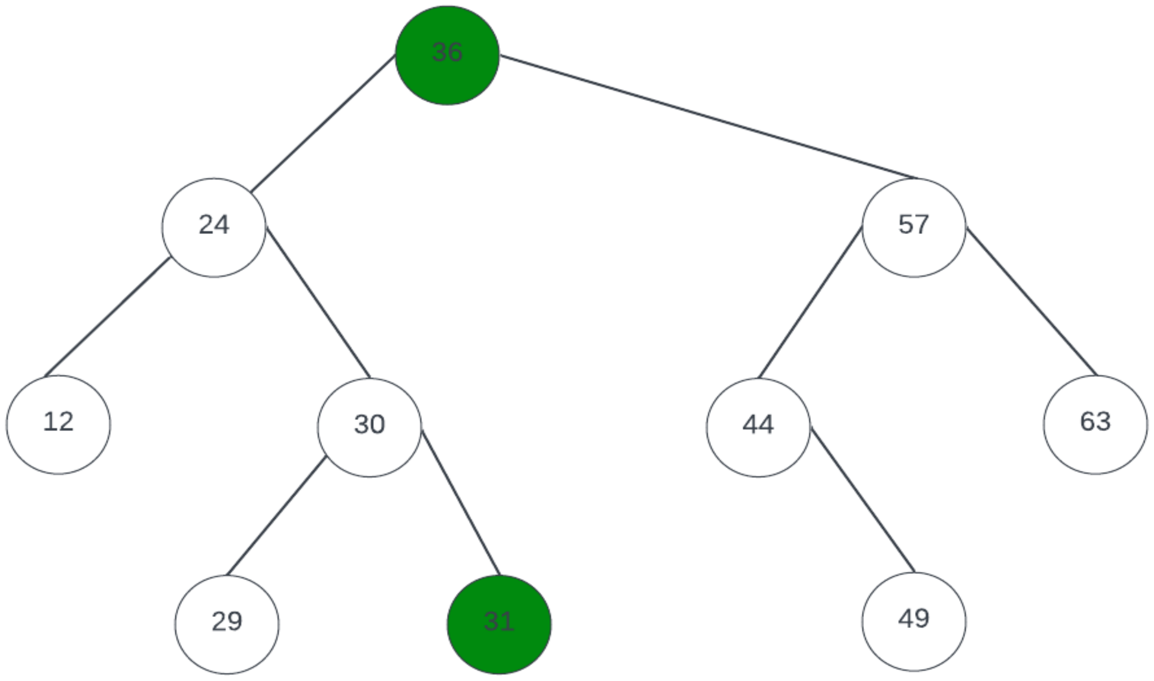
Problem 2 (10 points). Consider the following binary search tree:



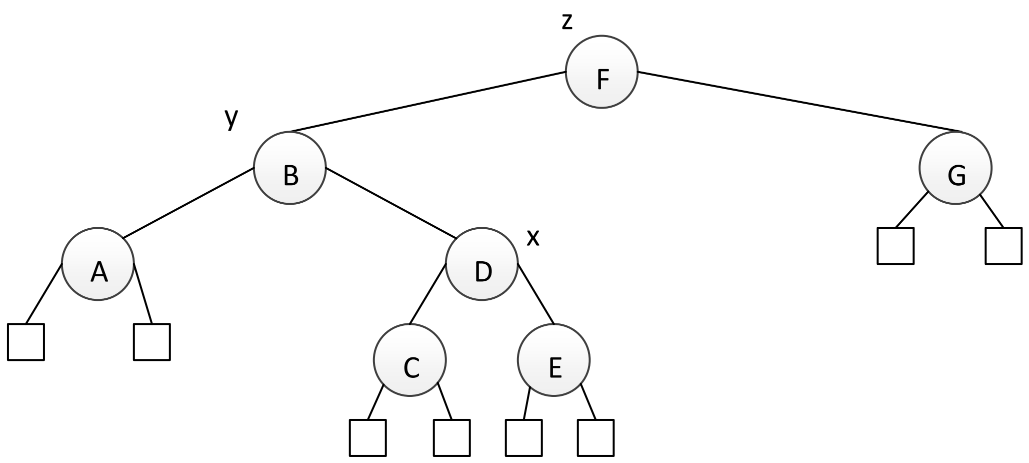
Show the resulting tree if you delete the entry with key = 42 from the above tree. You need to describe, step by step, how the resulting tree is generated.

- First, perform the search for the entry with key = 42 which is root node. This node has two children, both of which are internal nodes.
- Find node r that has the greatest key and less than 42 which is entry with key = 36
- Entry with key = 36 has only left child which is entry with key = 31
- Entry with key = 42 was deleted and entry with key = 36 is in its place
- The subtree rooted at r's left child is promoted to r's position which is entry with key = 31


42
24
57
12
30
44
63
29
36
49
31


36
24
57
12
30
44
63
29
31
49

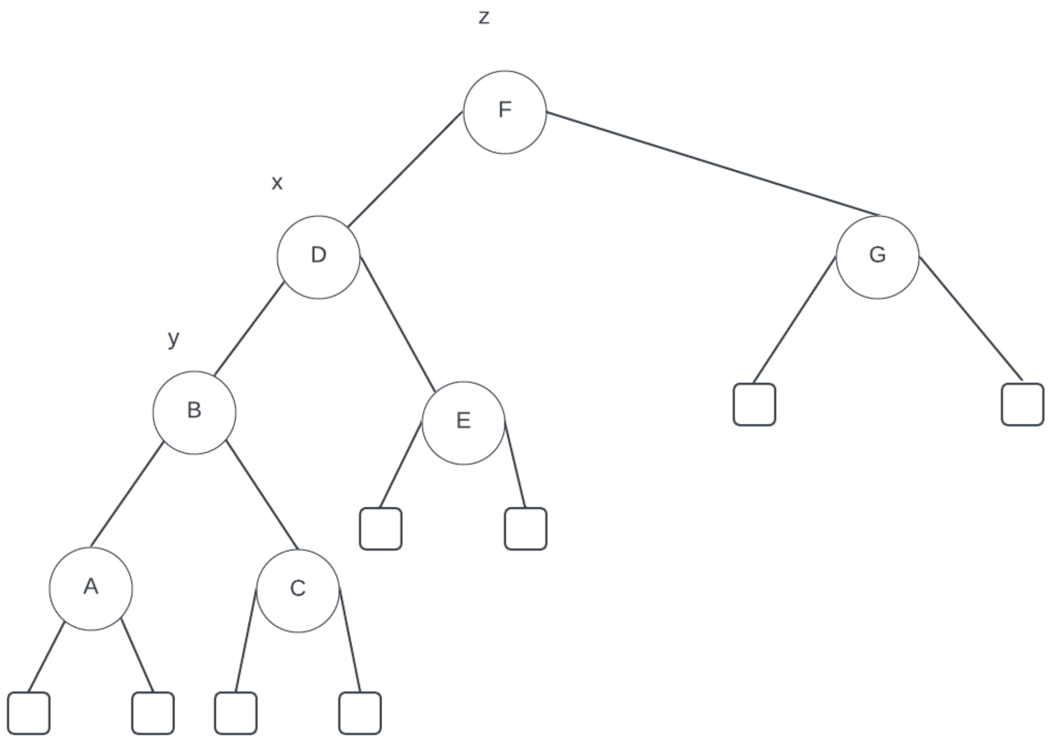
Problem 3 (10 points). Consider the following AVL tree, which is unbalanced:

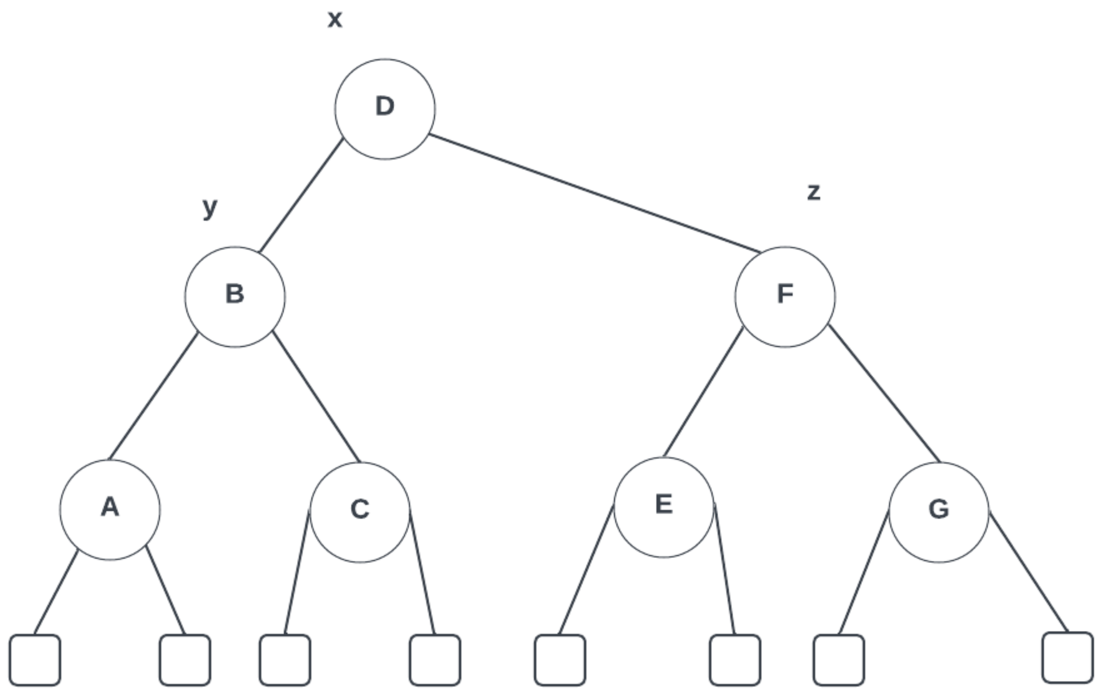


Note that the nodes *F*, *B* and *D* are labeled *z*, *y*, and *x*, respectively, following the notational convention used in the textbook. Apply a trinode restructuring on the tree and show the resulting, balanced tree.

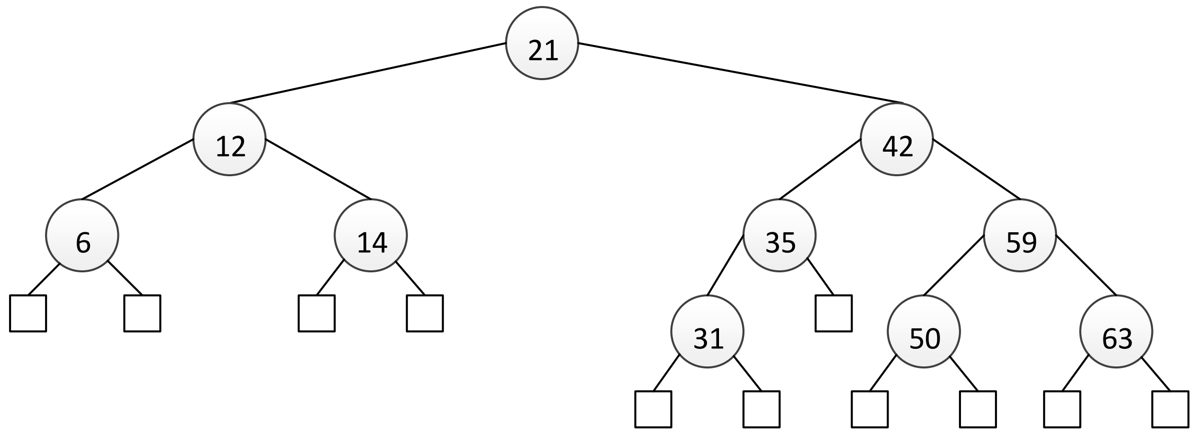
- The tree is unbalance at z (F)
- We have z > y and y < x → y < x < z
- We rotate y and x using left technique.
- Then we continue to right rotate z, x, y

→ the tree is balanced now.

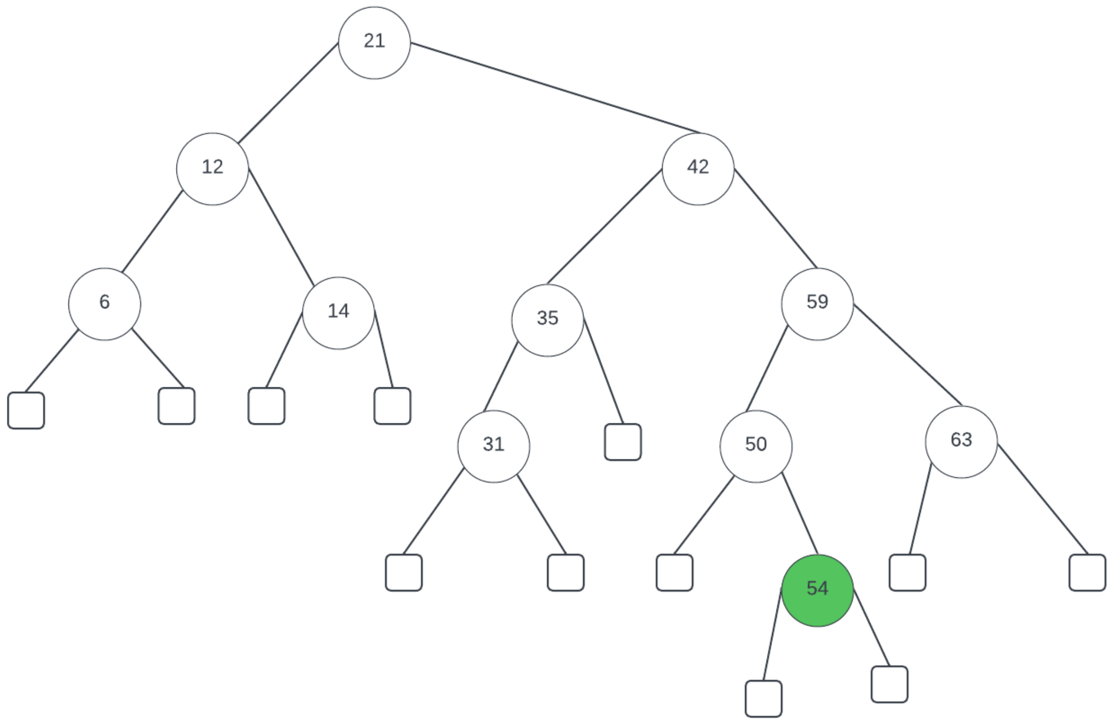

z
F
x
D
G
y
B
E
A
C


x
D
y
B
z
F
A
C
E
G

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

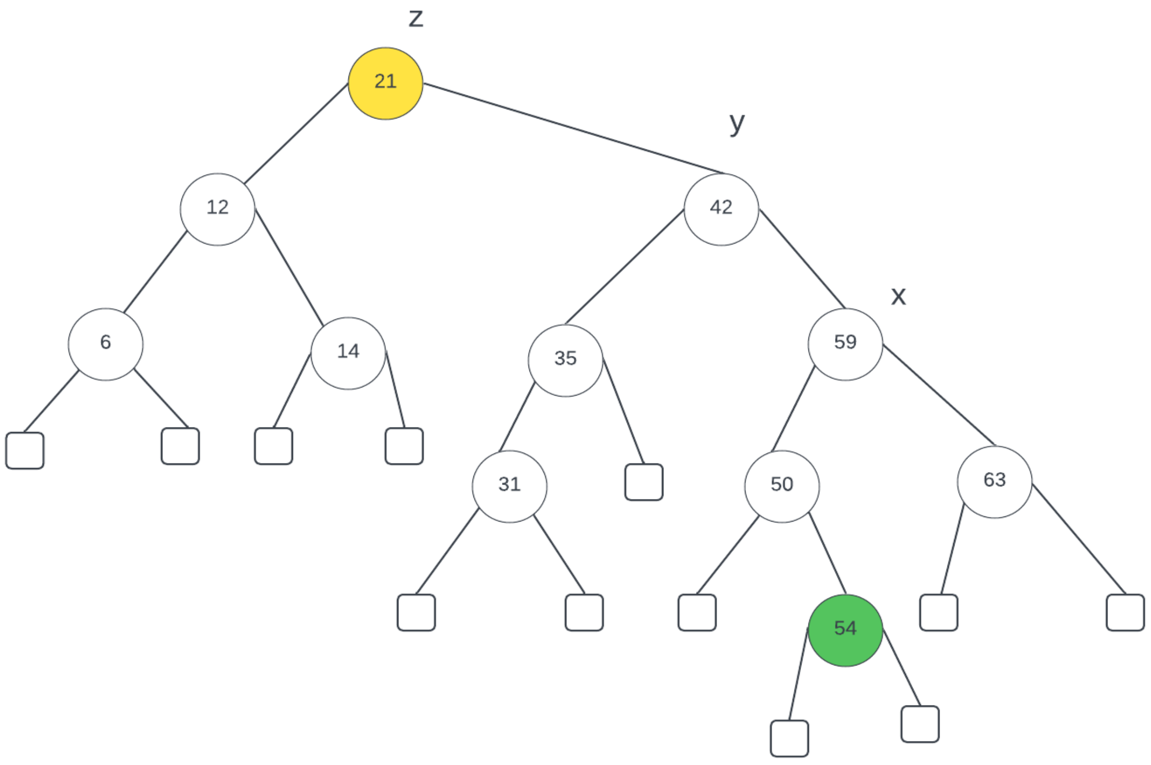


Show the resulting, balanced tree after inserting an entry with key = 54 to the above tree. You must describe, step by step, how the resulting tree is obtained.

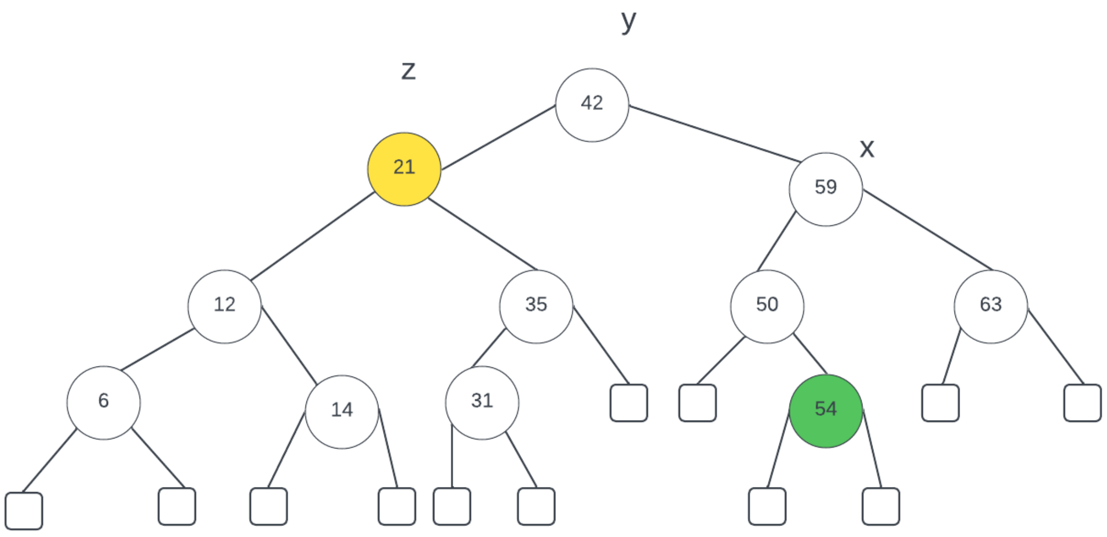


- After inserting an entry with key = 54 to the above tree → the AVL tree above become unbalance at root node with key = 21 Which is z

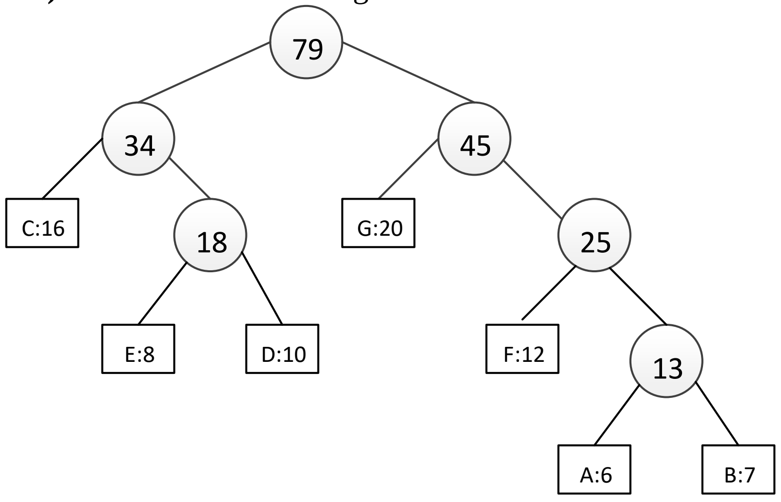
- We identify the y z by following the path which new node added → y is the entry with key 42, z is the entry with key = 59
- We use left rotate technique to rebalance the tree.



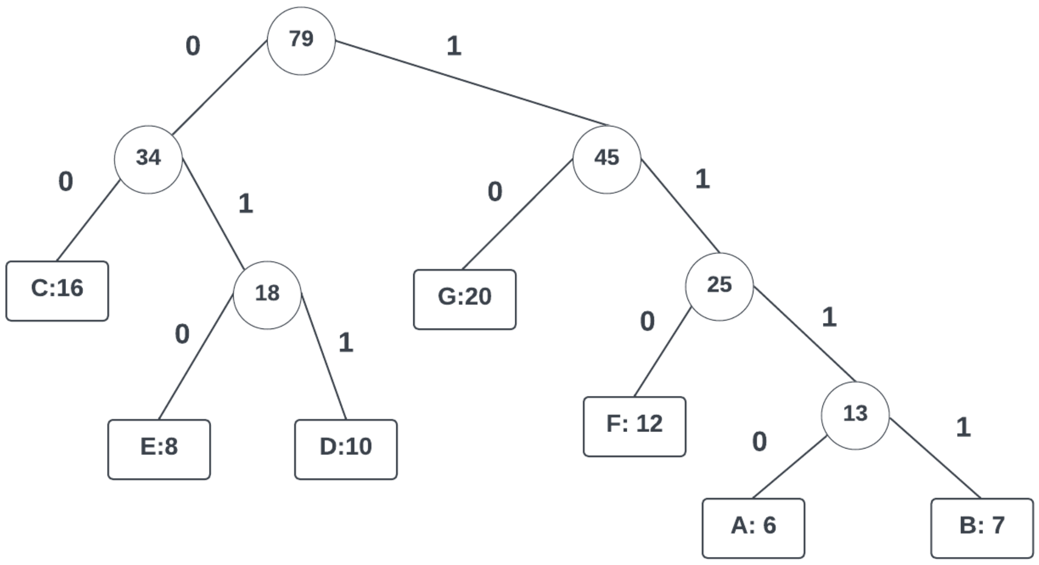
-



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



The Huffman: each edge is labeled 0 or 1. Left-child edge is labeled 0 and right-child edge is labeled 1



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

- B is encoded 1111
- D is encoded 011
- E is encoded 010
- G is encoded 10
- C is encoded 00

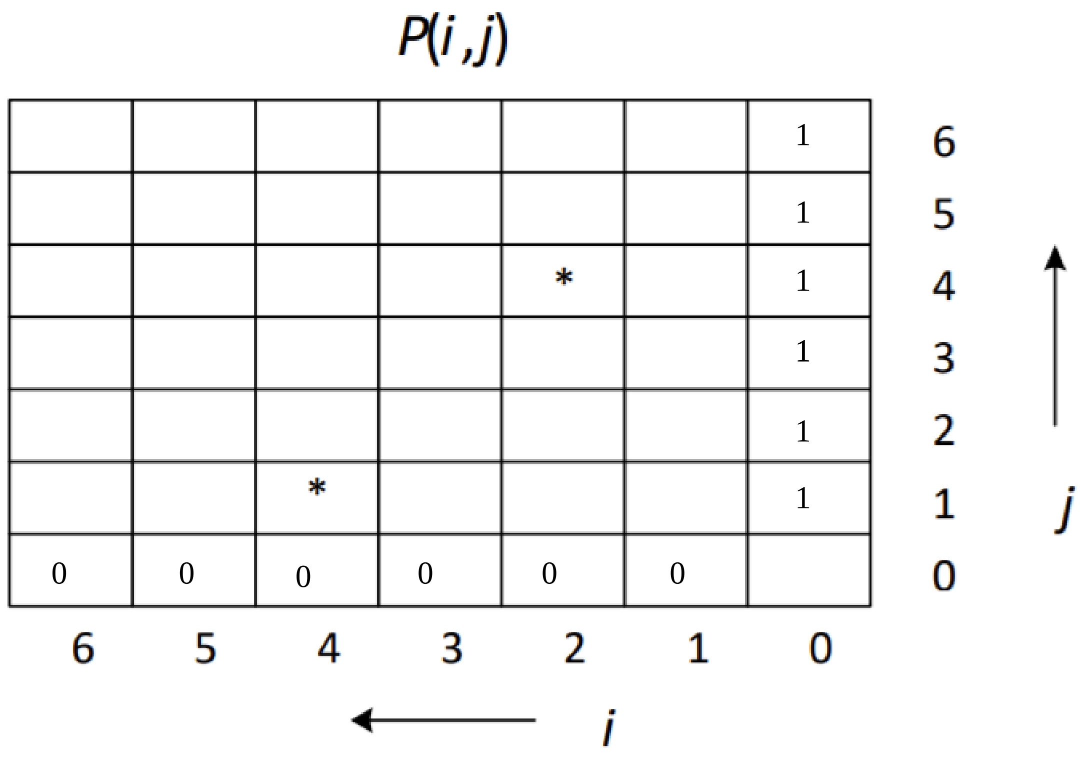
→ The string “BDEGC” to a bit pattern: **11110110101000**

(2) Decode the bit pattern “010 1110 10 011” to the original string using the Huffman tree.
Scan the message one bit at the time, stating from the root of coding tree. Once reached a leaf node or decoded one code-world → return root and repeat the process.

- The three bits 010 are decoded E
- The four bits 1110 are decoded A
- The two bits 10 are decoded G
- The three bits 011 are decoded D

→ The string is **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.



6	5	4	3	2	1	0	
						1	6
						1	5
				*		1	4
						1	3
						1	2
		*				1	1
0	0	0	0	0	0		0

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

Added base case value in the above table.

- Calculate P(4,1) and $P(2, 4)$
  P(1,1) = (P(0,1) + P(1,0))/2 = (1 + 0)/ 2 = 1/2
  P(1,2) = (P(0,2) + P(1,1))/ 2 = (1 + 1/2)/2 = 3/4
  P(2,1) = (P(1,1) + P(2,0))/ 2 = (1/2 + 0)/2 = 1/4

  P(1,3) = (P(0,3) + P(1,2))/ 2 = (1 + 3/4)/2 = 7/8

P(2,2) = (P(1,2) + P(2,1))/ 2 = (3/4 + 1/4)/2 = 1/2
P(3,1) = (P(2,1) + P(3,0))/ 2 = (1/4 + 0)/2 = 1/8

P(1,4) = (P(0,4) + P(1,3))/ 2 = (1 + 7/8)/2 = 15/16
P(2,3) = (P(1,3) + P(2,2))/ 2 = (7/8 + 1/2)/2 = 11/16
P(3,2) = (P(2,2) + P(3,1))/ 2 = (1/2 +1/8)/2 = 5/16

P(4,1) = (P(3,1) + P(4,0))/ 2 = (1/8 + 0)/2 = 1/16
P(2,4) = (P(1,4) + P(2,3))/ 2 = (15/16+11/16)/2 = 13/16

→ The probabilities of P(4, 1) = 1/16
→ The probabilities of P(2,4) = 13/16

Problem 7 (40 points). The goal of this problem is to give students an opportunity to compare and observe how running times of sorting algorithms grow as the input size (which is the number of elements to be sorted) grows. Since it is not possible to measure an accurate running time of an algorithm, you will use an *elapsed time* as an approximation. How to calculate the elapsed time of an algorithm is described below.

You will use four sorting algorithms for this experiment: insertion-sort, merge-sort, quick-sort, and heap-sort. A code of insertion-sort is in page 111 of our textbook. An array-based implementation of merge-sort is shown in pages 537 and 538 of our textbook. An array-based implementation of quick-sort is in page 553 of our textbook. You can use these codes, with some modification if needed, for this assignment. For heap-sort, our textbook does not have a code. You can implement it yourself or you may use any implementation you can find on the internet or any code written by someone else. If you use any material (pseudocode or implementation) that is not written by yourself, you must clearly show the source of the material in your report.

A high-level pseudocode is given below:

```
for n = 10,000, 20,000, . . ., 100,000 (for ten different sizes)
      Create an array of n random integers between 1 and 1,000,000
      Run insertionsort and calculate the elapsed time
      // make sure you use the initial, unsorted array
      Run mergesort and calculate the elapsed time
      // make sure you use the initial, unsorted array
      Run quicksort and calculate the elapsed time
      // make sure you use the initial, unsorted array
      Run heapsort and calculate the elapsed time
```

You can generate *n* random integers between 1 and 1,000,000 in the following way:

Random r = new Random( );
for *i* = 0 to *n* – 1
a[i] = r.nextInt(1000000) + 1

You can also use the *Math.random*( ) method. Refer to a Java tutorial or reference manual on how to use this method.

Note that it is important that you use the initial, unsorted array for each sorting algorithm. So, you may want to keep the original array and use a copy when you run each sorting algorithm.

You can calculate the elapsed time of the execution of a sorting algorithm in the following way:

```
long startTime = System.currentTimeMillis();
call a sorting algorithm
long endTime = System.currentTimeMillis();
long elapsedTime = endTime - startTime;
```

Write a program that implements the above requirements. Again for greater accuracy, you can use System.nanoTime() to get nanoseconds (and then convert to milliseconds).

Collect all elapsed times and show the result (1) as a table and (2) as a line graph.

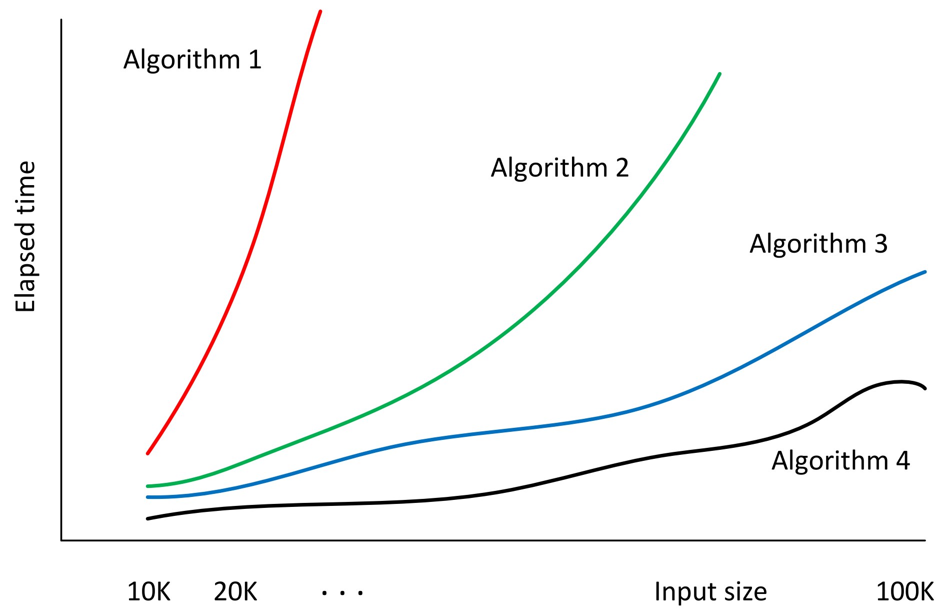
A table should look like:

n *Algorithm*	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
insertion	12	24	51	99	149	212	289	380	500	593
merge	3	2	3	3	5	6	6	8	8	8
quick	3	2	1	2	2	4	4	4	5	6
heapsort	2	2	4	3	4	5	6	7	8	9

n *Algorithm*	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
insertion	12	25	53	94	158	231	321	422	533	658
merge	2	2	2	4	4	6	7	8	8	9
quick	2	1	2	2	3	3	3	5	5	5
heapsort	2	2	3	3	4	5	6	6	8	9

Entries in the table are elapsed times in milliseconds.

A line graph should show the same information but as a graph with four lines, one for each sorting algorithm. The *x*-axis of the graph is the input size *n* and the *y*-axis of the graph is the elapsed time in milliseconds. Your graph should look like the following example (Note: this is just an example and your graph will look different):



You don't need to write a Java program to generate the graph. Once you have all elapsed times, you can plot the graph using any other tools, such as typical spreadsheet software.

Note that, in your result, the running time of an algorithm may not increase as (theoretically) expected. It is possible that the running time of an algorithm may decrease a bit as the input size increases in a part of your graph. As long as the general trend of your graphs is acceptable, there will be no point deduction. So, you should not be too much concerned about that.

You must provide a 1-2 paragraph conclusion including the observations you collected, analysis of the results, and discussion of what you learned.

Name the program *Hw5_P7.java*.

Deliverable

You need to submit the following files:

- *Hw5_P1_P6.pdf*: This file must include:
- Answers to problems 1 through 5.
- Discussion/observation of Problem 7: This part must include what you observed and learned from this experiment and it must be "substantive."
- *Hw5_P7.java*

- Other files, if any.

Combine all files into a single archive file and name it *LastName_FirstName_hw5.EXT*, where *EXT* is an appropriate archive file extension, such as *zip* or *rar*.

Grading

Problem 1 through Problem 6:

- For each problem, up to 6 points will be deducted if your answer is wrong.

Problem 7:

- There is no one correct output. As far as your output is consistent with generally expected output, no point will be deducted. Otherwise, up to 30 points will be deducted.
- If there are no sufficient inline comments, up to 10 points will be deducted.
- If your conclusion/observation/discussion is not **substantive**, points will be deducted up to 10 points