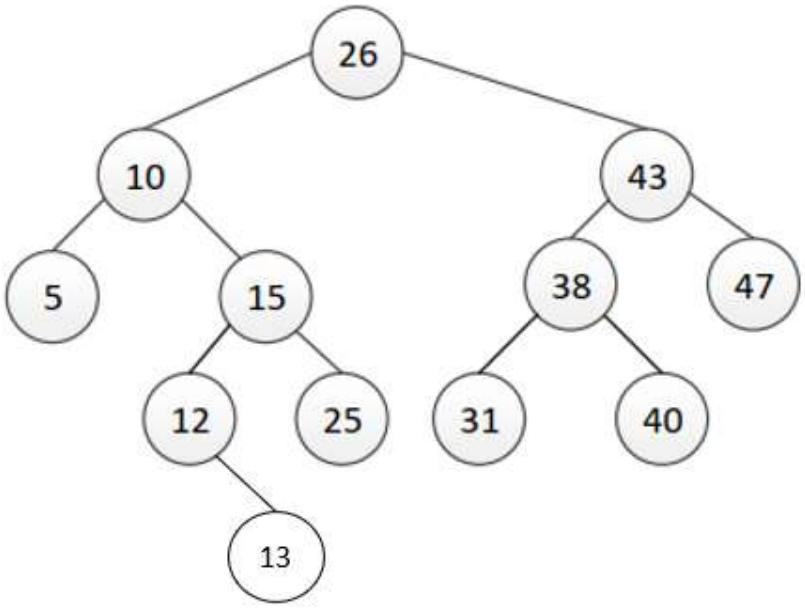
HW5

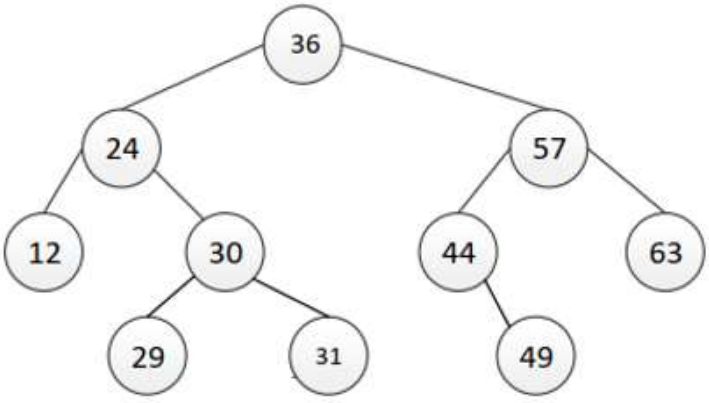
Problem 1

1. Start at root, key of root is 26, 13 < 26, go to left child of root.
2. Key is 10, 13 > 10, go to right child of parent with key 10.
3. Key is 15, 13 < 15, go to left child of parent with key 15.
4. Key is 12, 13 > 12, leaf with key 12 has no right child, so insert a node with key 13 as right child of node with key 12. The final binary search tree is:

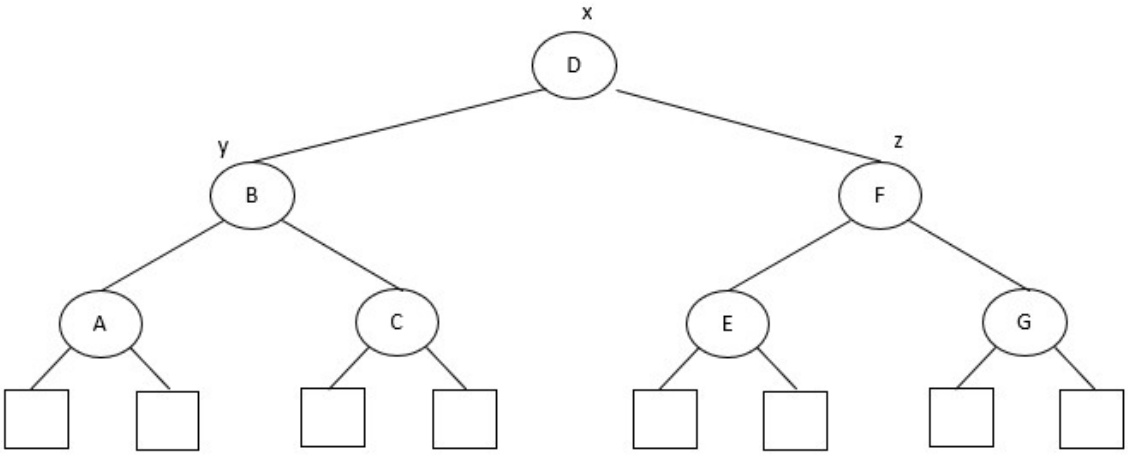


Problem 2

1. Remove node with key 42.
2. There are two possibilities, since the node with key 42 had a left and a right child, the node with the largest key in the left subtree or the node with the smallest key in the right subtree can be moved to the position where node with key 42 was. In this case the node with the largest key from the left subtree, 36, is set as root.
3. Since the node with key 36 only had one child (node with key 31) before it was moved, node with key 31 becomes the new right child of node with key 30.

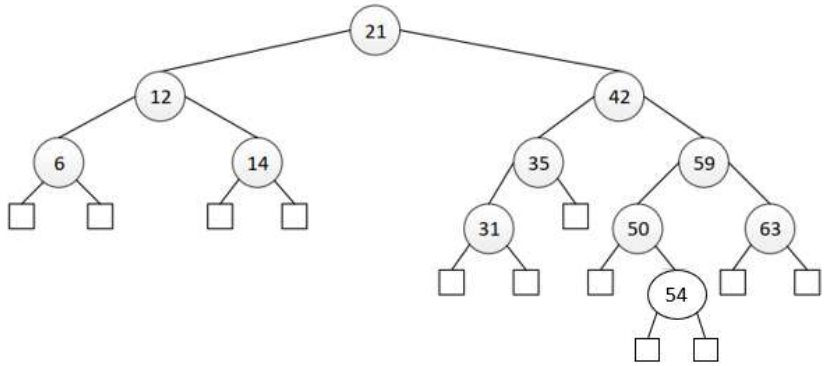


Problem 3

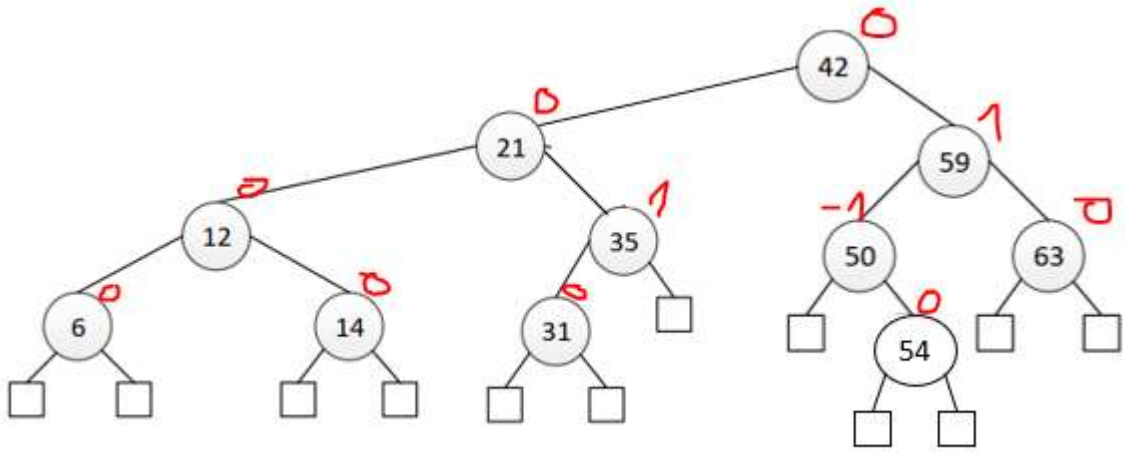


Problem 4

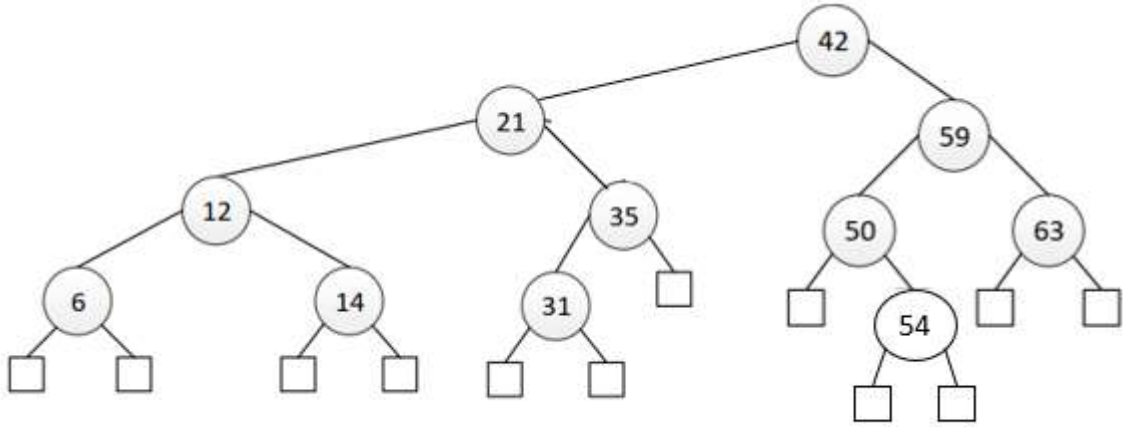
1. Start at root, key of root is 21, 54 > 21, go to right child of root.
2. Key is 42, 54 > 42, go to right child of parent with key 42.
3. Key is 59, 54 < 59, go to left child of parent with key 59.
4. Key is 50, 54 > 50, leaf with key 50 has no right child with a key, so insert a node with key 54 as right child of node with key 50. The binary search tree is:



5. But AVL tree is a binary search tree that satisfies the height-balance property (for every internal node p, heights of children of p differ by at most one), which is not fulfilled, because the height of the left subtree from node with key 21, is 3 and from the right subtree 5, which results in -2. Must rebalance binary search tree.
6. Rotate, so node with key 42 becomes the new root and left subtree (with keys 31 and 35) of node with key 42 becomes the right subtree of node with key 21.
7. Check height-balance property again:



The binary search tree is balanced and therefore it is an AVL tree. The final AVL tree is:



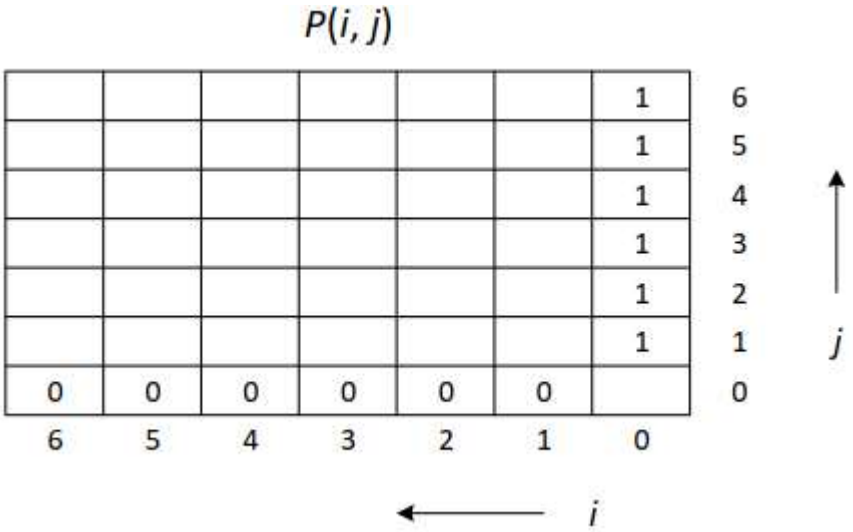
Problem 5

(1) Encode:
B: 1111
D: 011
E: 010
G: 10
C: 00
Bit pattern for BDEGC is: **11110110101000**

(2) Decode:
010: E
1110: A
10: G
011: D
Original string for 010111010011 is: **EAGD**

Problem 6

First, like in class, solve P(0, j) for all j (i.e., j = 1, 2, 3, 4, 5, 6), which are all 1; and solve P(i, 0) for all i (i.e., i = 1, 2, 3, 4, 5, 6), which are all 0:



The other probabilities are, P(i, j):

• 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(4, 1)** = (P(3, 1) + P(4, 0)) / 2 = (1/8 + 0) / 2 = **1/16**

• P(2, 3) = (P(1, 3) + P(2, 2)) / 2 = (7/8 + 1/2) / 2 = 11/16

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

• **P(2, 4)** = (P(1, 4) + P(2, 3)) / 2 = (15/16 + 11/16) / 2 = 26/32 = **13/16**

Problem 7

Sources sorting algorithms:

- Insertion sort: page 111 of our textbook.
- Merge sort: pages 537 and 538 of our textbook.
- Quick sort: page 553 of our textbook.
- Heap sort: https://www.geeksforgeeks.org/heap-sort/

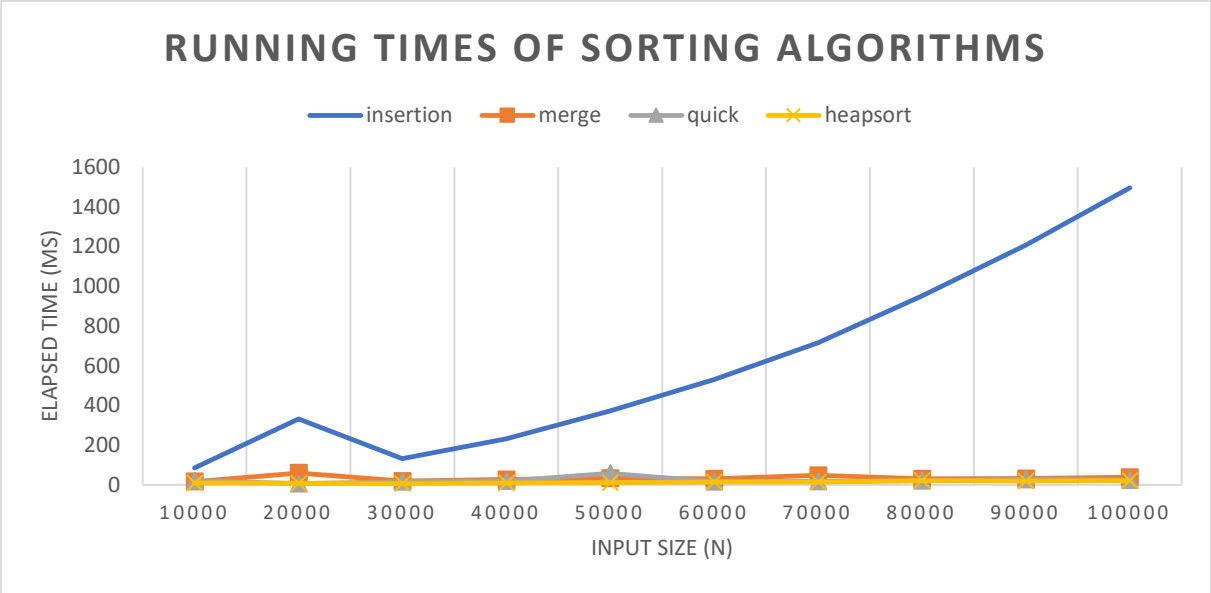
Our textbook: Data Structures and Algorithms in Java.

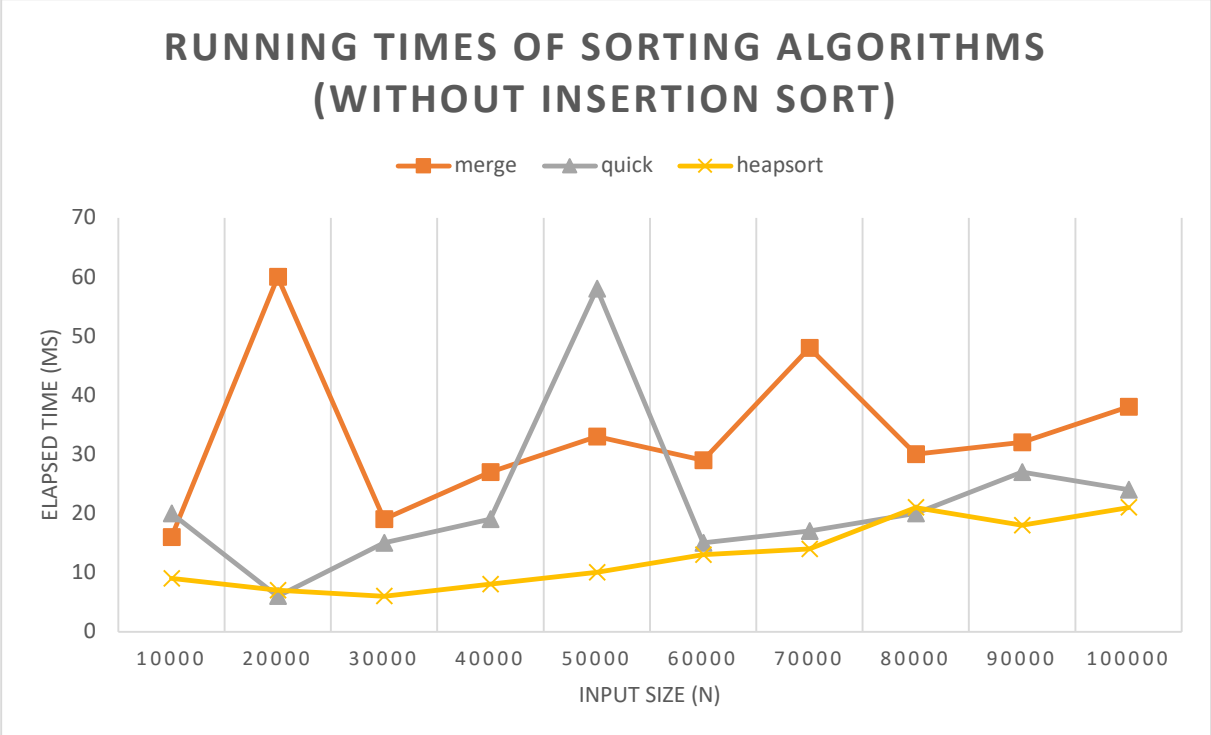
Observation and learned

Table running times of sorting algorithms

n / Algorithm	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
insertion	85	331	132	232	372	530	716	951	1207	1497
merge	16	60	19	27	33	29	48	30	32	38
quick	20	6	15	19	58	15	17	20	27	24
heapsort	9	7	6	8	10	13	14	21	18	21

Entries in the table are elapsed times in milliseconds.





Kasperzack Stefan

I observed that the insertion sort takes the longest time and the insertion times increase the most, because as implemented, in the worst case all integers to the left of the current integer have to be swapped until the current integer can be inserted in the correct place. Then this process is repeated for the next current integer. This results in a running time of $O(n^2)$, which can be seen in the first diagram, and the table also shows that the increase in execution time for insertion sort is greater than linear or O(n log n). The second diagram shows that the heap sort is faster than the quick sort and merge sort, but for input sizes 200,000, 600,000 to 800,000 and 1,000,000 heap and quick sort have about the same execution times (it seems that quick sort and heap sort have similar running times for some distributions of random integer arrays). This can be expressed more precisely with the execution times in the table: heap sort is on average 195 percent faster than quick sort and 287 percent faster than merge sort up to input sizes of 600,000. However, from an input size of 600,000 the running times of heap sort and quick sort almost overlap and the running times of merge sort also approach the running times of quick and heap sort. That agrees with what we learned in the lectures; heap sort seems to be better than quick sort and merge sort for small to medium input sizes. Merge sort had the longest execution times of merge sort, quick sort, and heap sort; and the execution times of quick sort were between the execution times of merge sort and heap sort, but were closer to heap sort than to merge sort. I observed that there were fluctuations in the execution times of insertion, merge, quick, and heap sort. The fluctuations can be influenced by the distribution of the random integers in the arrays. The insertion sort for input size 200,000 had the greatest spike in execution time, which could mean that there were relatively more larger values at the front of the array (small indices) than at the back (large indices). The quick sort for an input size of 500,000 also had a large spike in execution time. This could be explained by badly distributed integers in the array, so that fewer halving splits of the array were possible, so that the quick sort execution time can change from O(n log n) to $O(n^2)$. Another reason for the fluctuating execution times can be that other processes are running in the background in the system that influence the running times. It should be noted that execution times are measured in milliseconds and a change from 6 to 9 milliseconds is a large percentage change, but a small change in time units. I have learned that insertion sort, merge sort, quick sort, and heap sort can be implemented in-place, that is, there is no need to create a new array in memory, but the sorting takes place in the input array, which has the advantage that the sorting does not have to use twice the amount of memory, which with large amounts of data could lead to an OutOfMemoryError being thrown. However, one has to be careful when implementing the merge sort – I have looked at several implementations and pseudocodes and some implementations of the merge sort were not in-place. I conclude that in practice it can be important to implement sorting algorithms and compare the running times for different data sets (and data set sizes) if sorting algorithms are used and a time-efficient execution is important, and not to assume that every O(n log n) algorithm has the same running time for every data set. For the given input sizes and integer arrays: heap sort is on average 4666 percent faster than insertion sort, quick sort is on average 2638 percent faster than insertion sort, and merge sort is on average 1723 percent faster than insertion sort; and heap sort was on average 161 percent and quick sort on average 50 percent faster than merge sort, although all three sorting algorithms are O(n log n) – except quick sort when an array of size N is not halved but partitioned into two arrays with size N - 1 and 1 (with every partitioning), then $O(n^2)$.