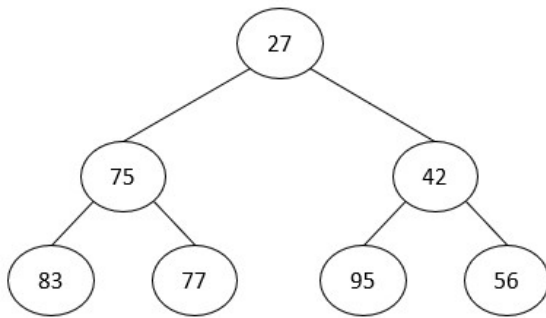


HW4

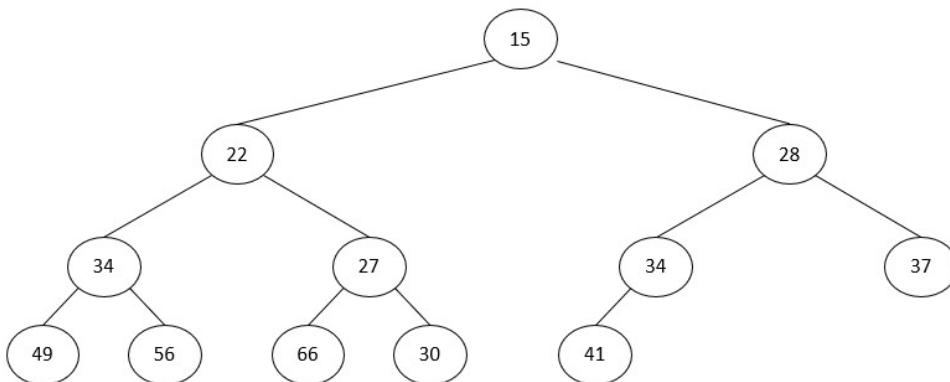
Problem 1

1. Since a heap is a complete binary tree, entry with key = 27 is inserted as the right child of node with key 56.
2. Since $27 < 56$, node with key 27 is swapped with its parent.
3. Since $27 < 42$, node with key 27 is swapped with its parent. The root is reached. So, stop. The final heap is:



Problem 2

1. Remove the root node with key 5 and move the last node with key 34 up to be the new root.
 2. Right child has smaller key, so node with key 34 is swapped with node with key 15.
 3. Left child has smaller key, so node with key 34 is swapped with node with key 28. And stop.
- The final heap is:

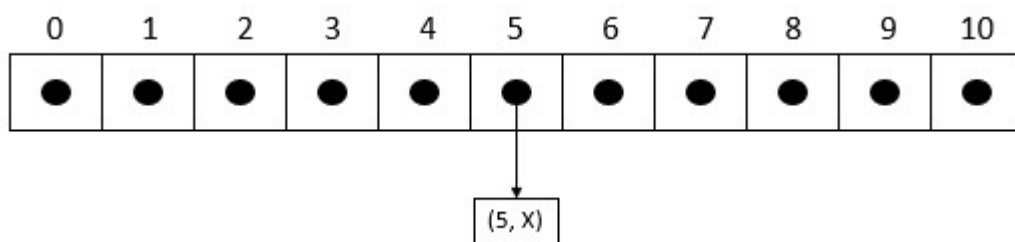


Problem 3

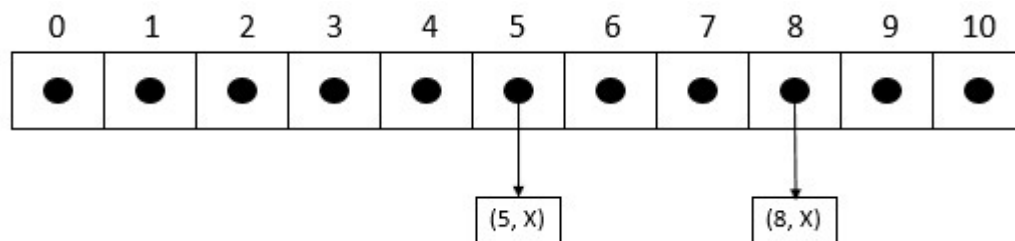
The division method is used to determine the indices (integer mod N). And since no values are given, X is set as value.

integer	N	integer mod N
5	11	5
8	11	8
44	11	0
23	11	1
12	11	1
20	11	9
35	11	2
32	11	10
14	11	3
16	11	5

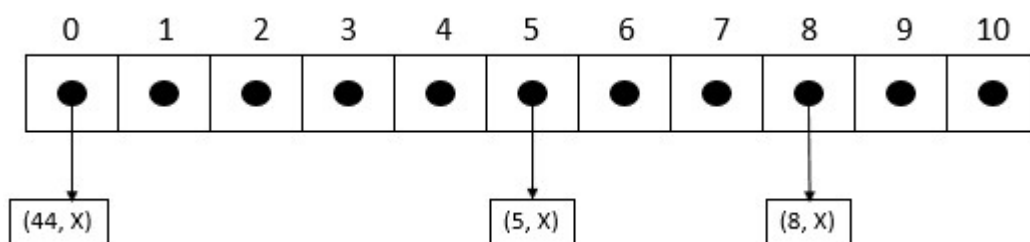
1. Insert 5



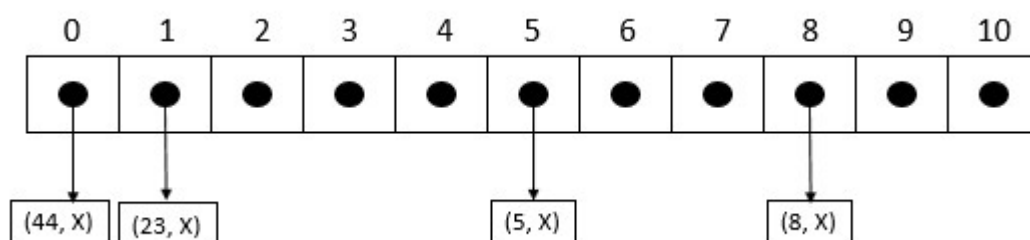
2. Insert 8



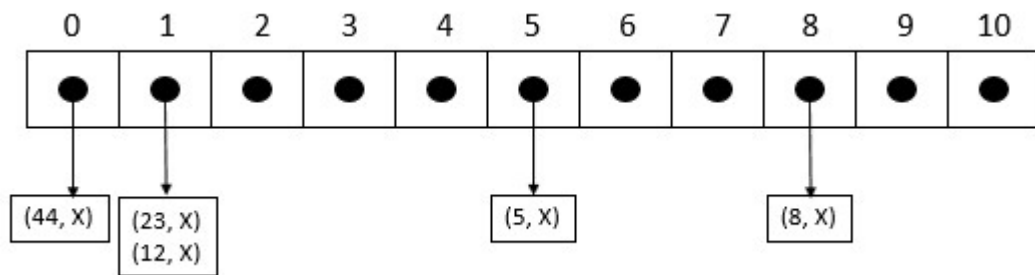
3. Insert 44



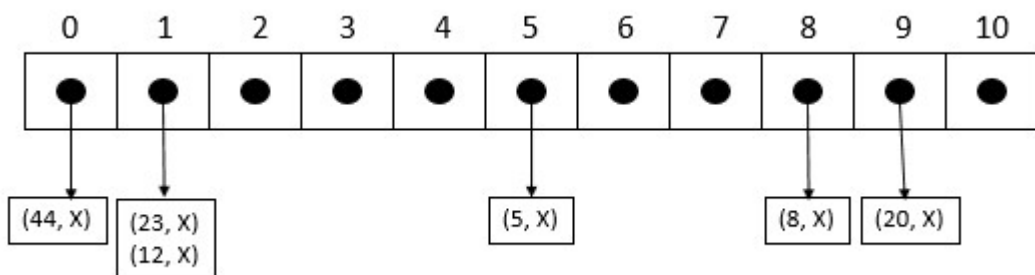
4. Insert 23



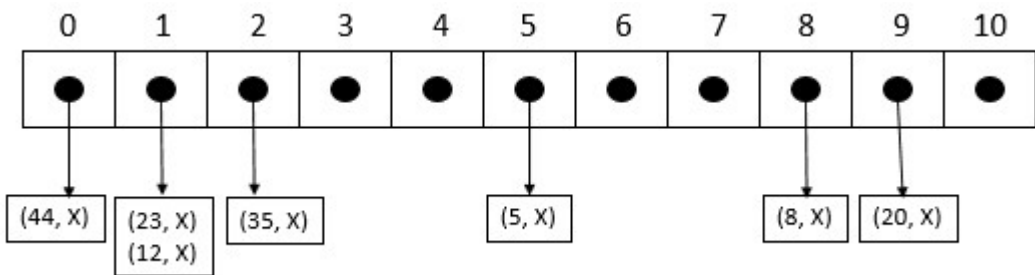
5. Insert 12



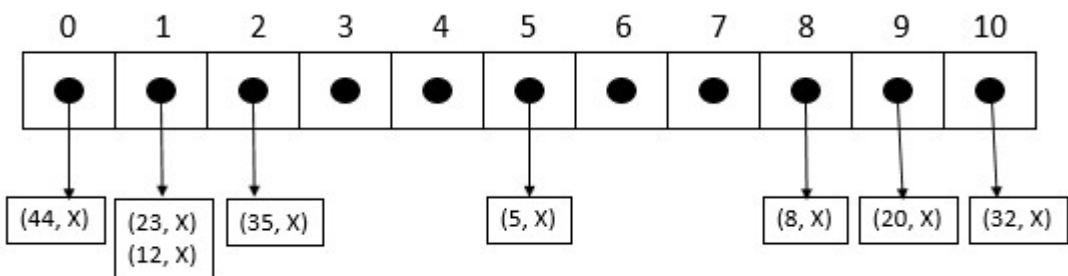
6. Insert 20



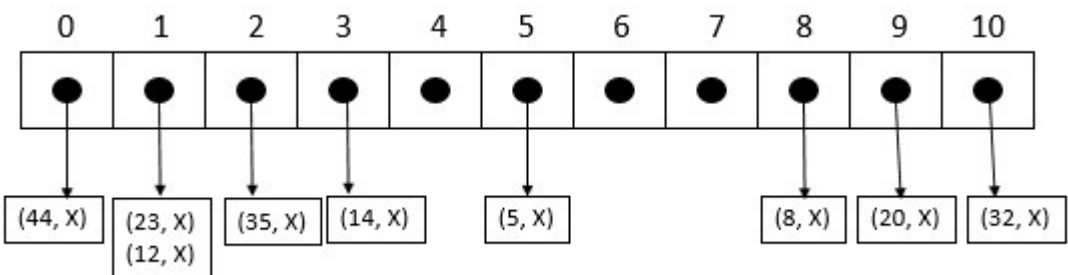
7. Insert 35



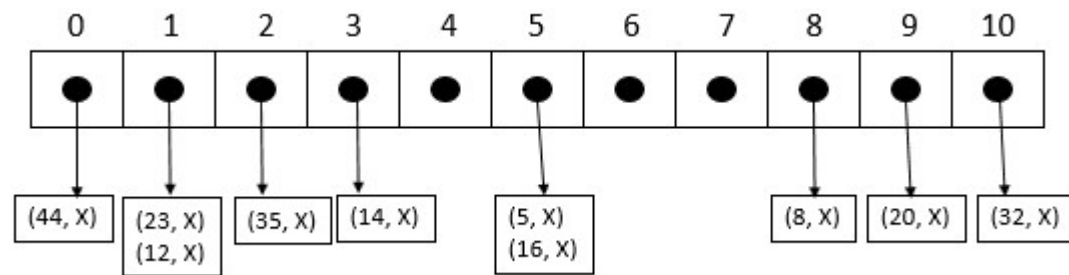
8. Insert 32



9. Insert 14



10. Insert 16

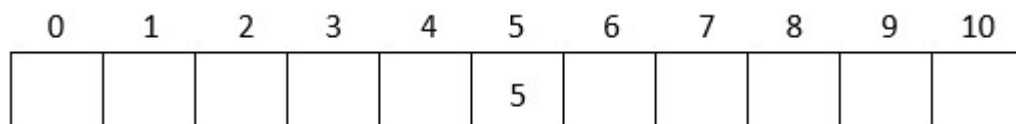


Problem 4

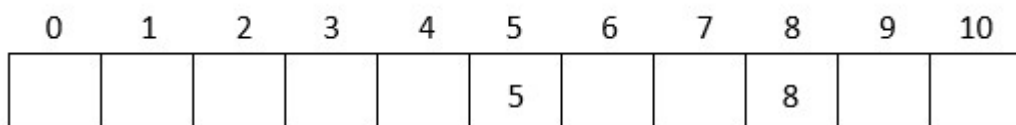
The division method is used to determine the indices (integer mod N) – before collisions.

integer	N	integer mod N
5	11	5
8	11	8
44	11	0
23	11	1
12	11	1
20	11	9
35	11	2
32	11	10
14	11	3
16	11	5

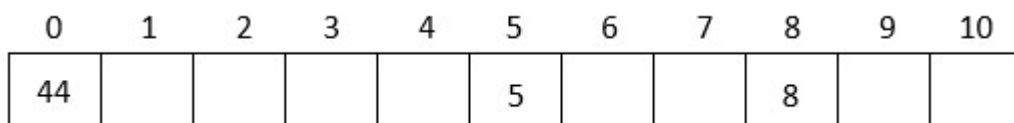
1. Insert 5



2. Insert 8




3. Insert 44



4. Insert 23

0	1	2	3	4	5	6	7	8	9	10
44	23				5			8		

5. Insert 12 - collision




0	1	2	3	4	5	6	7	8	9	10
44	23	12			5			8		

6. Insert 20

0	1	2	3	4	5	6	7	8	9	10
44	23	12			5			8	20	

7. Insert 35 - collision




0	1	2	3	4	5	6	7	8	9	10
44	23	12	35		5			8	20	

8. Insert 32


0	1	2	3	4	5	6	7	8	9	10
44	23	12	35		5			8	20	32

9. Insert 14 - collision



0	1	2	3	4	5	6	7	8	9	10
44	23	12	35	14	5			8	20	32

10. Insert 16 - collision



0	1	2	3	4	5	6	7	8	9	10
44	23	12	35	14	5	16		8	20	32

Problem 5

1. The hash function $h(k) = k \bmod 13$ gives for $k = 16$: $h(16) = 3$; but 3 is occupied with 29.
2. The second hash function $h'(k) = 1 + (k \bmod 11)$ gives for $k = 16$: $h'(16) = 6$. And now $h(k, i) = (h(k) + i \cdot h'(k)) \bmod N$ has to be calculated.
3. For $k = 16$ and $i = 1$: $h(16, 1) = (h(16) + h'(16)) \bmod 13$ gives $h(16, 1) = 9$; but 9 is occupied with 48.
4. Repeat 3. for $i = 2$: $h(16, 2) = (h(16) + 2 \cdot h'(16)) \bmod 13$ gives $h(16, 2) = 2$; but 2 is occupied with 2.
5. Repeat for $i = 3$: $h(16, 3) = (h(16) + 3 \cdot h'(16)) \bmod 13$ gives $h(16, 3) = 8$; but 8 is occupied with 21.
6. Repeat for $i = 4$: $h(16, 4) = (h(16) + 4 \cdot h'(16)) \bmod 13$ gives $h(16, 4) = 1$; 1 is empty, store 16.

0	1	2	3	4	5	6	7	8	9	10	11	12
	16	2	29		18			21	48	15		

Problem 6

Observation and learned

I observed that the average total times for inserting and searching for each data structure can fluctuate when the code is run several times. Especially, when I am using other programs besides running the code. The insert and search total average execution times for all data structures fluctuated from 5 to 52 percent around the respective total average execution time for several executions of the code. The greatest fluctuation was found when searching in the HashMap: the slowest execution of the search in the HashMap took 225 percent longer than the fastest execution of the search. And the average search speed in ArrayList and LinkedList only fluctuated by 5 percent around the total average. I think, the fluctuations depend on the allocation of resources by the system, but also on the random numbers generated and the internal implementation of the data structures. When inserting the numbers into the HashMap, the fluctuation was high, which could be due to that more collisions occur in one execution and fewer collisions in another; which depends on the distribution of the random numbers. The HashMap was slower than ArrayList and LinkedList when inserting: it took on average 48 percent longer to insert the integers into the HashMap than into the ArrayList and 228 percent longer than into the LinkedList. When inserting, the HashMap was the slowest because the values had to be hashed and possible collisions had to be resolved. The integers can be inserted fastest in the LinkedList and second fastest in the ArrayList. The fact that the ArrayList is slower than the LinkedList is probably due to that the ArrayList has to increase the array size (before the internal array is completely filled) and has to copy all elements to a larger array. Nonetheless, it was surprising that the LinkedList is faster than the ArrayList when inserting, because the LinkedList has distributed its nodes over the random-access memory and because the LinkedList has to set more values when a node is inserted; the LinkedList must set previous reference, element,

and next reference (and increment the size), while the ArrayList only needs to set the element (and increment the size). When the search times are compared, the HashMap is on average more than 99 percent faster than the ArrayList or LinkedList. This is because the entire HashMap does not have to be searched for the key, but only the hash code for the key has to be determined and the index is then determined via the hash code in order to look up via the index whether the key is in the HashMap (assumed that there were no collisions before); but in the worst case, the entire ArrayList or LinkedList have to be iterated until the searched value is found or it is determined that the value does not exist. It was also surprising that searching the LinkedList took about 170 percent long than searching the ArrayList, which could be because the LinkedList has distributed its nodes over the random-access memory. I also noticed that there are several different ways to create random integers in Java, with Random, Math.random(), and ThreadLocalRandom; however, it does not seem that one of the variants is faster or slower than the other. I learned that the random numbers generated are actually pseudo-random. This means that the generated numbers are generated with the use of a seed – and are dependent on this seed. This means that the same pseudo-random numbers are generated for the same seed. If I were to implement the pseudocode again, I would put all three data structures into an array and iterate over the data structure array inside the main for loop to reduce the amount of code by almost two thirds. I conclude that it is incorrect to assume that in practice $O(1) = O(1)$ and $O(n) = O(n)$, because the constants and lower order terms are not taken into account in the time complexity notation, but in practice these are relevant. And because the runtimes in practice depend on the data set used. This becomes particularly clear when time complexity and actual average times are compared. Below are the worst-case time-complexities for HashMap, ArrayList, and LinkedList – taken from the lecture materials:

	Insert	Search
HashMap	put(): $O(1)^*$	containsKey(): $O(1)^*$
ArrayList	add(): $O(1)$	contains(): $O(n)$
LinkedList	add(): $O(1)$	contains(): $O(n)$
*: based on the assumption that there are few collisions		

And the actual average runtimes:

in milliseconds	Insert	Search
HashMap	29.5	4.9
ArrayList	19.9	9183
LinkedList	9	24766