Ege Ozan Özyedek
CS202-1
21703374

Assignment 3 – Heaps and AVL Trees

Question 1

For question parts 1.a and 1.b, a digital notepad app was used to visualize the binary trees and function calls. I used some symbols to increase the understandability of the drawings and decided it would be best to put a short directive. It can be found below.

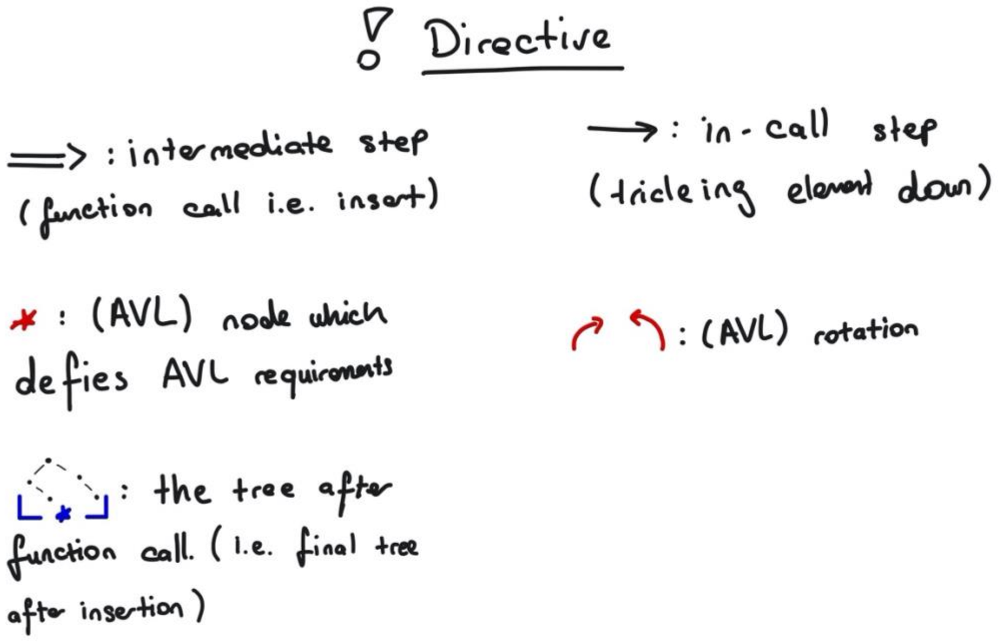

Directive
: intermediate step (function call i.e. insert)
: in-call step (trickleing element down)
: (AVL) node which defies AVL requirements
: (AVL) rotation
: the tree after function call. (i.e. final tree after insertion)

Figure 1: Directive for the drawings of question 1.a and 1.b

a) This question requires the insertion of 13, 6, 3, 7, 2, 4, 11, 0, -1 and 1 in the given order into an AVL tree. AVL trees keep itself balanced, meaning the height difference between the left and right subtrees have to be at most one. The drawing of the trees and insertions to an AVL tree can be found on the next page.

Ege Ozan Özyedek
CS202-1
21703374

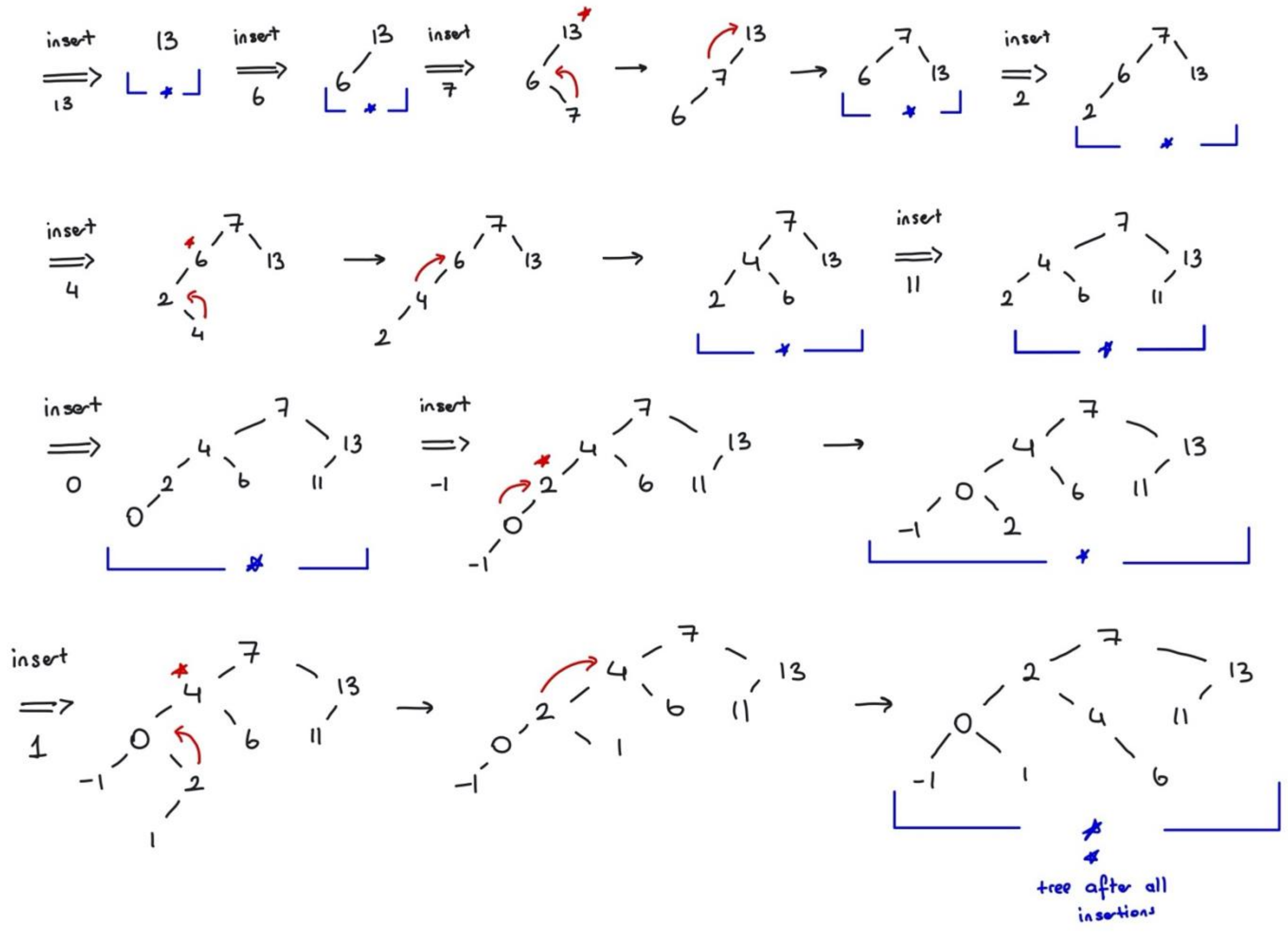


Figure 2: Answer to question 1.a

b) This question requires the insertion of 12, 8, 3, 7, 4, 5, 13, 2, 6, 10 and 1 in the given order into a min-heap. After the insertion, the function deleteMin is called twice. This function deletes the root node. This is because min-heaps have the minimum element at the parent node, while having elements bigger than the parent node on the left and right children. The drawing of the trees and insertions to the heap can be found on the next page.

Ege Ozan Özyedek
CS202-1
21703374

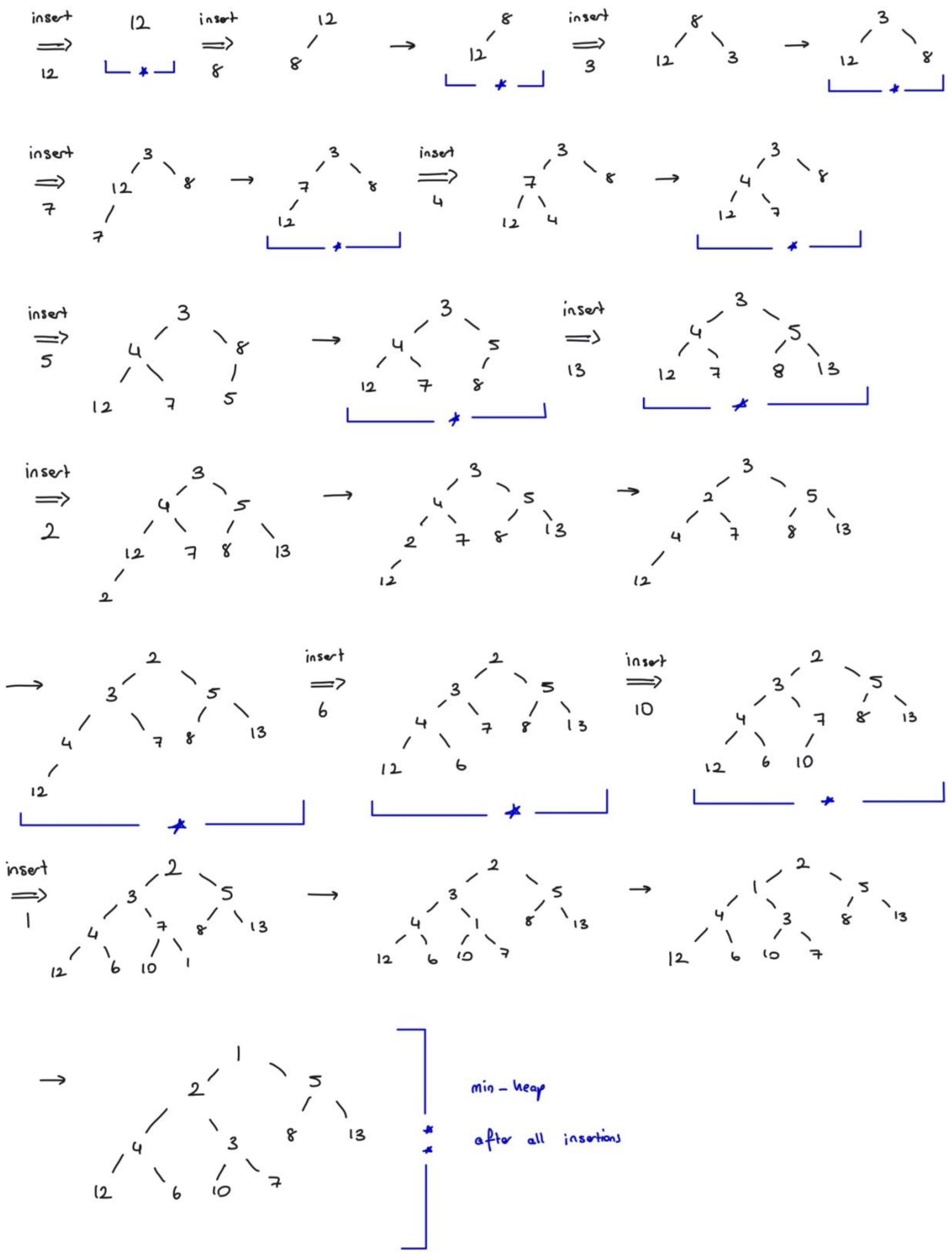


Figure 3: Answer to question 1.b, the insertion of given elements into the min-heap

Ege Ozan Özyedek
CS202-1
21703374

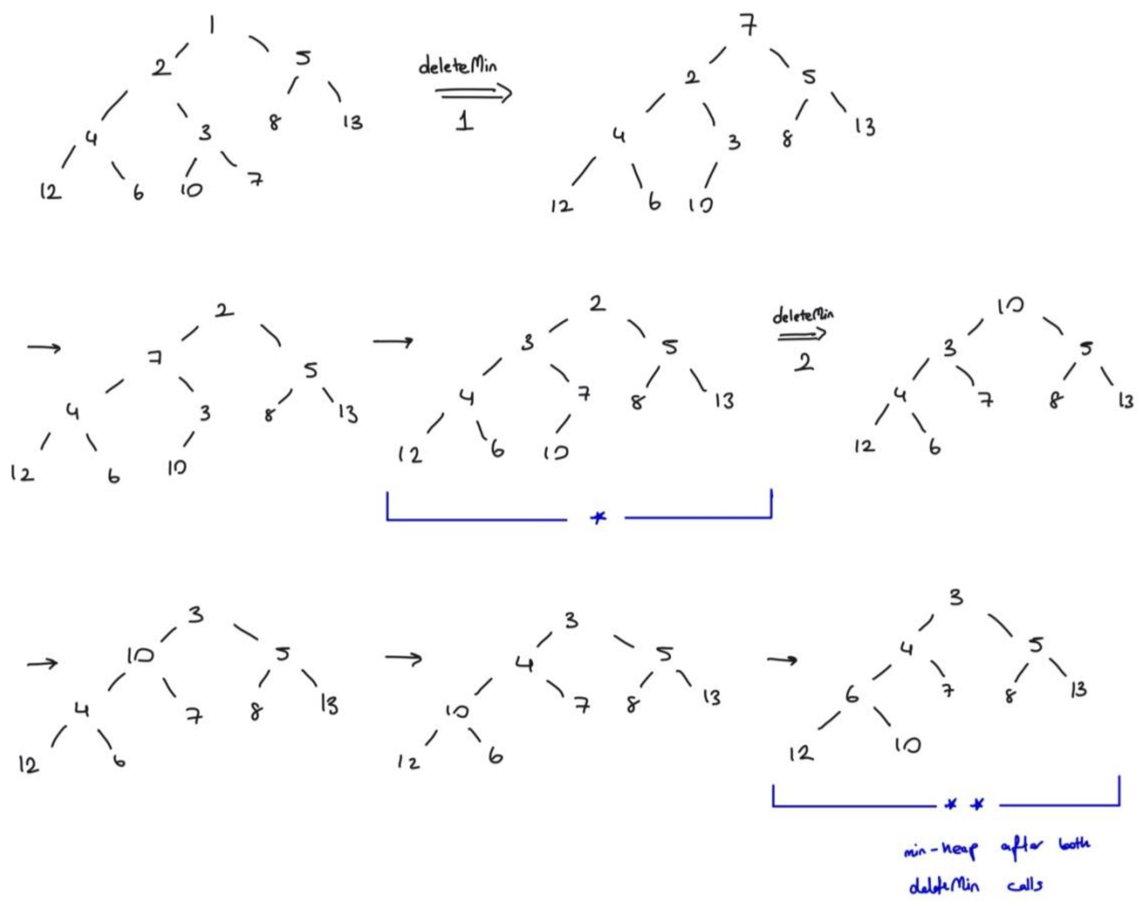


Figure 4: Answer to question 1.b, calling deleteMin twice for the min-heap

c) For this question, consider the binary heap we have found above as the final result of question 1.b.

Table 1: preOrder traversal for binary min-heap

3	4	6	12	10	7	5	8	13

Table 2: inOrder traversal for binary min-heap

12	6	10	4	7	3	8	5	13

Table 3: preOrder traversal for binary min-heap

12	10	6	7	4	8	13	5	3

It can be observed that indifferent of the traversal option, simply traversing a binary tree does not output a sorted array of the heap elements. Unlike a BST, a heap does not differentiate between its child nodes, as long as both nodes are bigger (min-heap) or smaller (max-heap) than the parent node, it suffices as a heap. Hence there is no definite comparison between the child nodes. The traversal options, in the sense of sorting a tree, operate according to the difference between the children and the parent. Since this difference is much more blurred in a heap, it is not possible to sort a heap through traversal. Regardless, there already exists an algorithm which sorts a heap, it's called the heapsort algorithm and it is discussed and implemented in question 2.

d) The expression of the minimum number of nodes for an AVL Binary Tree is as follows:

$$n(h) = 1 + n(h-1) + n(h-2)$$

This is a recurrence relation which outputs the minimum number of nodes needed for a tree with height h. n(h) represents the minimum node count for the tree with height h, the number 1 represents the root of the tree n(h), while n(h-1) and n(h-2) represent the children of the root (left and right subtrees with given heights). This is because of the AVL tree property, which is the fact that an AVL tree has to have at most a height difference of one between two sub-trees. We also have prior knowledge that n(1) = 1 (height = 1, only root exists) and n(0) = 0 (height = 0, tree is empty). Hence using these we can calculate the minimum number of nodes any AVL tree with height h.

The question further asks us to find the minimum number of nodes for an AVL tree with height h = 15. There are two ways to calculate this value, either we can calculate all n(15) to n(2) values by hand and finally sum all values with their correct multipliers to find the result, or we can write a C++ code which recursively calculates this value. I have chosen the second way to calculate it, the code of this function as well as the result can be found below.

```
int findMinNodesAVL(int h) {
   if (h == 0)
      return 0;
   else if (h == 1)
      return 1;
   else
      return 1 + findMinNodesAVL(h-1) + findMinNodesAVL(h-2);
}
```

Console Output: **Minimum number of nodes for AVL tree of height 15 : 1596**

Ege Ozan Özyedek
CS202-1
21703374

e) There are two prerequisites to keep in mind while trying to find whether a regular binary tree is a min-heap. First, for each parent node, its children should have a bigger value than itself. Second, since all binary heaps are complete trees, the tree should also be a complete tree. To check whether a binary tree is complete or not, I will use the method which I also used in HW2's programming assignment. If the tree is complete, each element index has to be smaller than the size of the tree. The indexing is done by assuming the parent has index i, and its children has indexes $i_{left} = 2i + 1$ and $i_{right} = 2i + 2$. Size represents the number of nodes in the binary tree, and the node represents a single node object in the tree.

bool isMinMeap(**TreeNode*** node, **int** index, **const** int size)
initial call: *isMinHeap(root, 0, size)*

```
if node == NULL          // stopping statement for the recursive function
   return true

else if index < size                                   // is complete check
         and node->data < node->left->data             // left child bigger?
         and node->data < node->right->data            // right child bigger?

   return isMinHeap(node->left, 2 * index + 1, size)
            and isMinHeap(node->right, 2 * index + 2, size)   // recursive call

else
   return false       // is not complete or children elements are smaller than parent
```

Ege Ozan Özyedek
CS202-1
21703374

Question 2

- **Heap:** The Binary Heap is a data structure which stores elements based on key values, similar to that of a binary search tree. However, the main difference between the heap and BSTs lies in the placement of values. Heaps come in two main structures, max-heap and min-heap. Since the implementation of the code contains max-heap, I will be focusing on this structure. The heap stores the maximum value as the parent, while both children of said parent has to have a lower value than the parent. This is different from BSTs which have smaller values on the left tree and bigger on the right tree. The heap is always a complete tree, and I will be using this fact while finding the number of comparisons.

- **Heapsort:** The Heapsort algorithm uses the heap data structure to sort an array of elements. The algorithm first inserts all elements into the heap using the insert function. Then, using the popMaximum function, obtains the maximum element in the heap while removing said element. This way the heapsort algorithm obtains all elements in descending order from the heap. The obtained elements are stored in another array in ascending order (the obtained max element is added to the end of the array). This array is used to write an output file which contains the array elements separated on each line, as well as the number of comparisons counted in the sorting of the array.

- **Number of Comparisons:** The number of comparisons for the heapsort algorithm consist of the number of comparisons that occur in the heapRebuild function. Although I have considered the possibility in which the comparisons made in the construction of the array were also added to the total count, I found that I couldn’t find a certain expression for it. Unlike heapRebuild, the construction of the array does not have a definite number of comparisons and is solely based on ordering of the input array given. Considering this, I will start the calculations for the number of comparisons in the Heapsort algorithm.

  Our first piece of knowledge is that a max-heap always has a larger value as the parent and smaller values as children. This also reveals that from the root to the last leaf of the heap, the elements are ordered in descending order.

  Another piece of information at hand is the fact that all binary heaps are complete trees. It is also known as a theorem that a complete binary tree containing n nodes has

$$\mathrm{h} = \lceil \log_2(n+1) \rceil$$

  This information is crucial, as it reveals the fact that the algorithm considers h nodes when traversing from the root to the last leaf.

The final knowledge needed to understand the computation amount is how the popMaximum algorithm works. It replaces the root of the heap with the last element of the heap and decreases the size count, which "deletes" the root element. Since the last element of the heap is smaller than all node elements from the root to height h-1, the heap has to position this element back to its rightful place while also in the process making the heap comply with its requirements. Hence after the deletion of the first element heapRebuild is called. heapRebuild recursively compares all h-1 nodes with the root of the tree which is smaller than all h-1 nodes. It does not go in one path however, the traversal may continue right or left considering which element has the highest value. Hence, the root element has to be compared to h-1 regardless, but the added uncertainty of direction gives us the result that at each recursive call 2 comparisons are made. Hence it can be concluded from the above explanation that the heapRebuild function, hence the heapsort function makes 2h-2 comparisons to sort one element of the array.

Combining all remarks made above, we can conclude that for the sorting of a single element the amount of comparisons needed are

$$2(h-1) = 2(\lceil \log_2(n+1) \rceil - 1)$$

However, as it was emphasized, this is for the sort of one element, while the heap has n nodes. Additionally, the number of nodes change as the function gets sorted. The total number of nodes decrease from the initial node count n=N (capital N representing the size of the heap) to n=1, which has only the minimum element. The comparison amount of each element should be summed up to find the final comparison amount. Hence, by using the fact that the sum of the first k numbers equal

$$\sum_{k=1}^{k=t} k = \frac{k(k+1)}{2}$$

we can obtain the final expression for the comparison amount of HeapSort.

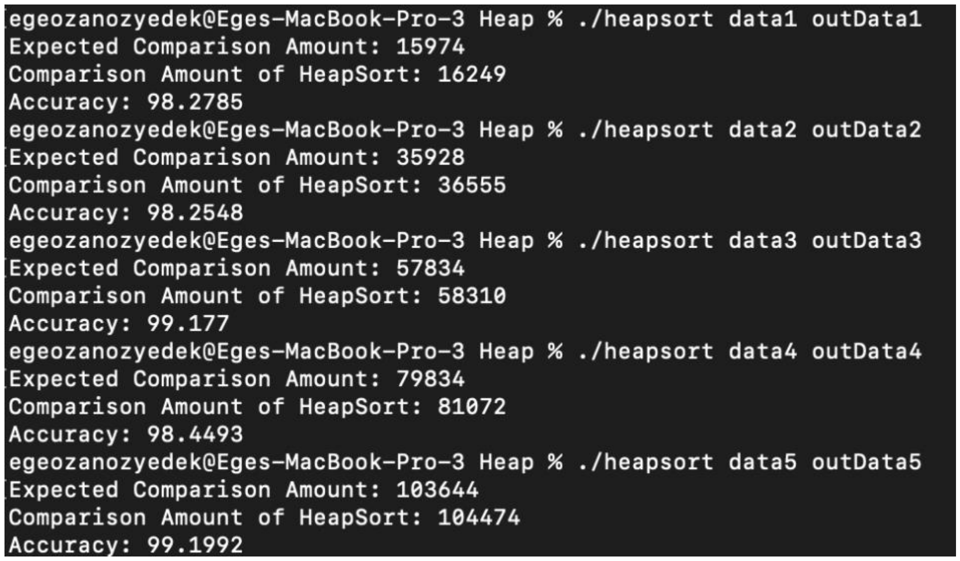
$$number\ of\ comparisons = 2\sum_{n=1}^{n=N} h - 1$$

$$number\ of\ comparisons = 2\sum_{n=1}^{n=N} \lceil \log_2(n+1) \rceil - 1$$

For the expected computations, I used C++'s math.h library, which has log2() and ceil() functions which are needed for the computation. The calculated results and the results obtained from the given data can be found below. It should be noted that the heapRebuild function that I've written only adds a second comparison if a right node exists. In the case that it doesn't, it does not add the comparison and that call of the heapRebuild function only adds one comparison to the total. Hence the final results may not be exact but will have a high accuracy. The accuracy percentage will also be displayed.

Table 4: Number of comparisons for given data files

Data	**Calculated**	**Result**	**Accuracy (%)**
data1	15,974	16,249	98.28
data2	35,928	36,555	98.25
data3	57,834	58,310	99.18
data4	79,834	81,072	98.45
data5	103,644	104,474	99.20



```
egeozanozyedek@Eges-MacBook-Pro-3 Heap % ./heapsort data1 outData1
Expected Comparison Amount: 15974
Comparison Amount of HeapSort: 16249
Accuracy: 98.2785
egeozanozyedek@Eges-MacBook-Pro-3 Heap % ./heapsort data2 outData2
Expected Comparison Amount: 35928
Comparison Amount of HeapSort: 36555
Accuracy: 98.2548
egeozanozyedek@Eges-MacBook-Pro-3 Heap % ./heapsort data3 outData3
Expected Comparison Amount: 57834
Comparison Amount of HeapSort: 58310
Accuracy: 99.177
egeozanozyedek@Eges-MacBook-Pro-3 Heap % ./heapsort data4 outData4
Expected Comparison Amount: 79834
Comparison Amount of HeapSort: 81072
Accuracy: 98.4493
egeozanozyedek@Eges-MacBook-Pro-3 Heap % ./heapsort data5 outData5
Expected Comparison Amount: 103644
Comparison Amount of HeapSort: 104474
Accuracy: 99.1992
```

Figure 5: Terminal output which show the above results