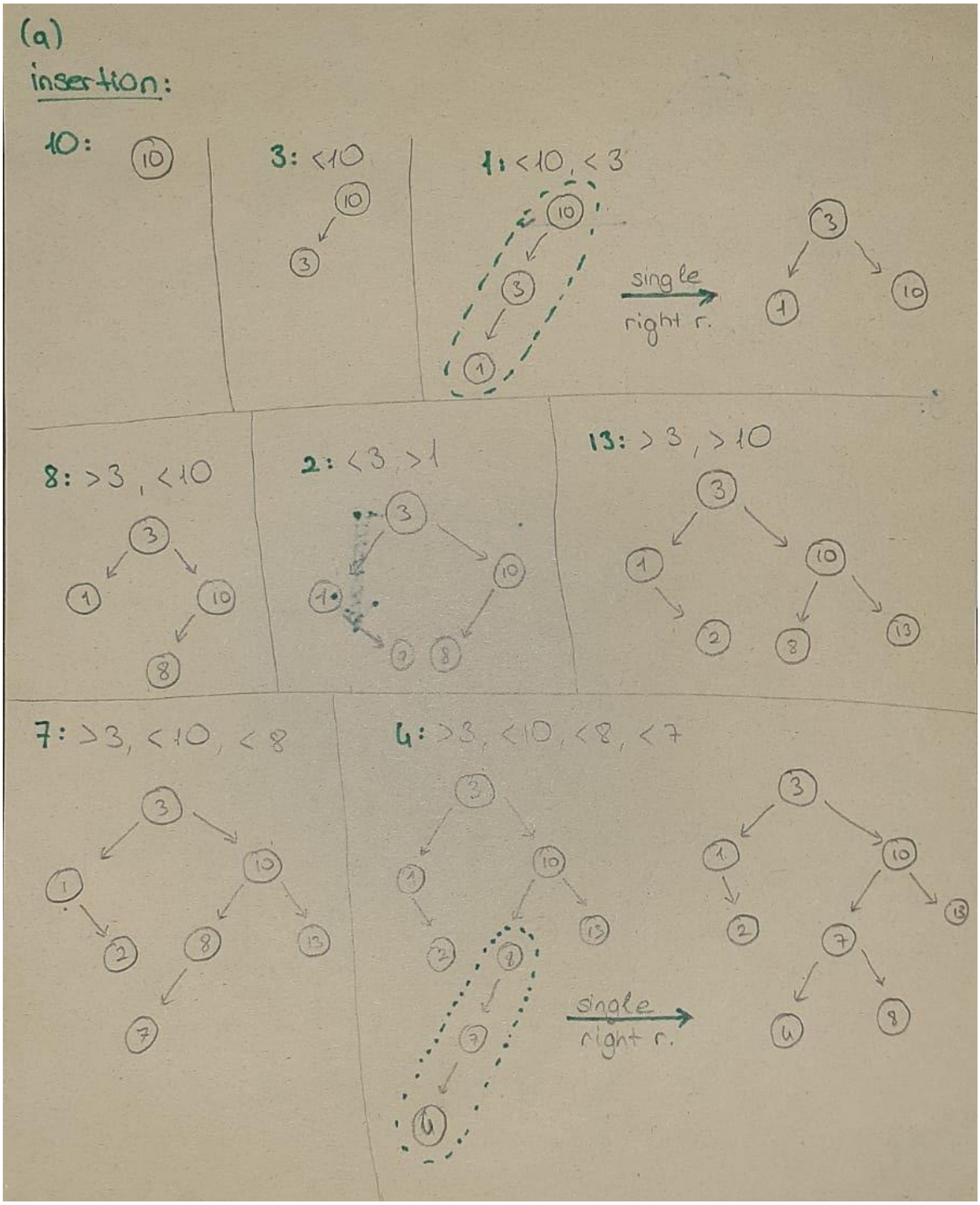
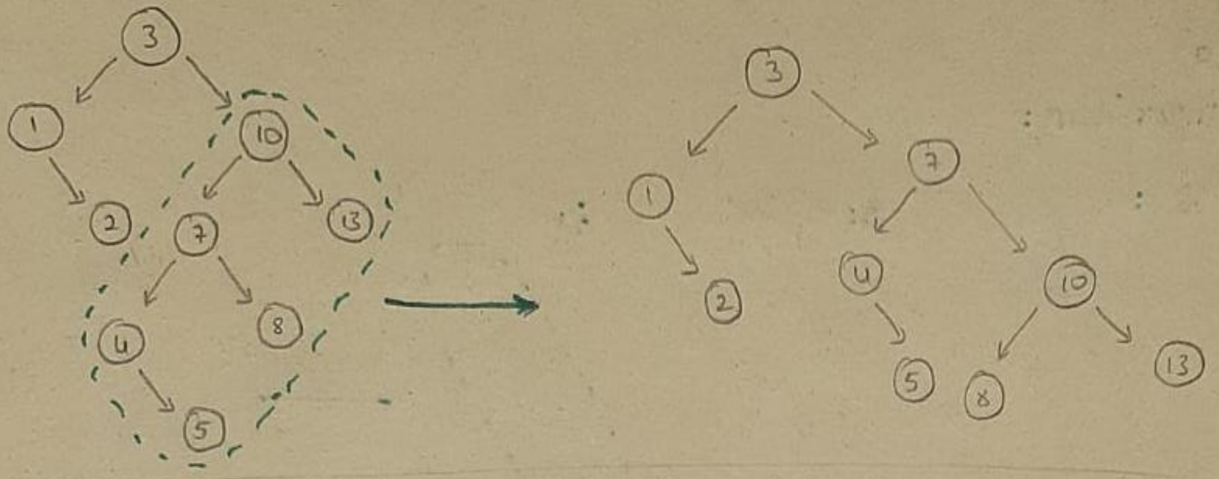
Sümeyye ACAR
22103640
CS202 – 01
HW3

Question 1 (a)

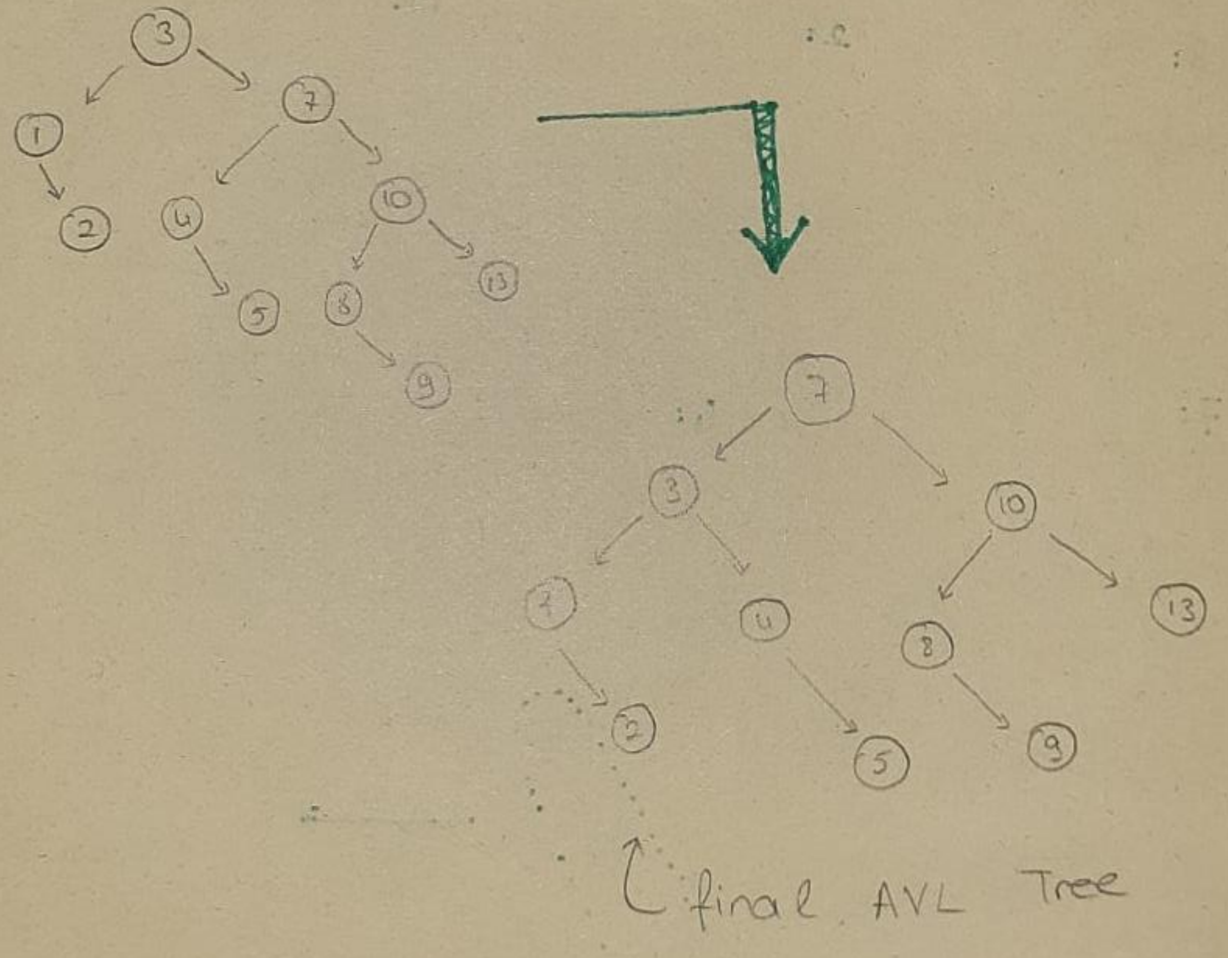

(a)
insertion:
10:
3: <10
1: <10, <3
single right r.
8: >3, <10
2: <3, >1
13: >3, >10
7: >3, <10, <8
4: >3, <10, <8, <7
single right r.

Question 1 (a)

5: >3, <10, <7, >4



9: >3, >7, <10, >8

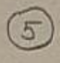


Question 1 (b)

(b)

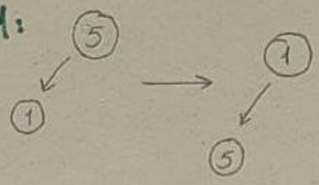
insertion:

5:



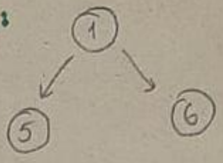
array= [5....]

1:



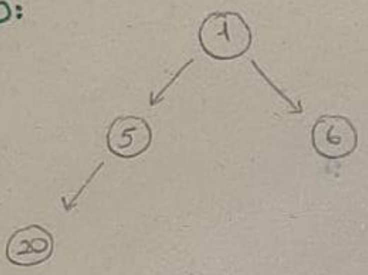
array= [1,5...]

6:



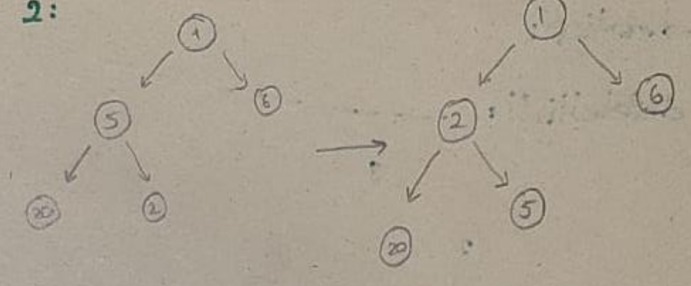
array= [1,5,6...]

20:



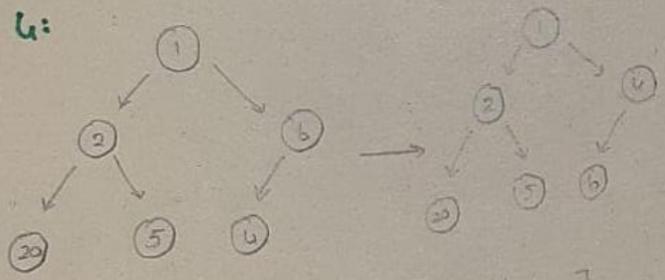
array= [1,5,6,20...]

2:



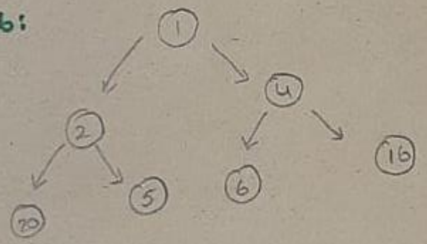
array= [1,2,6,20,5...]

4:



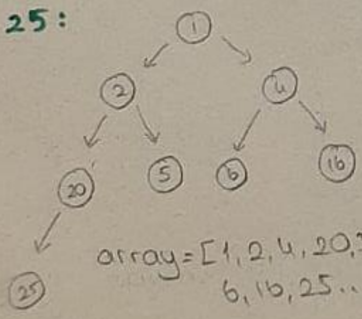
array= [1,2,4,20,5,6...]

16:



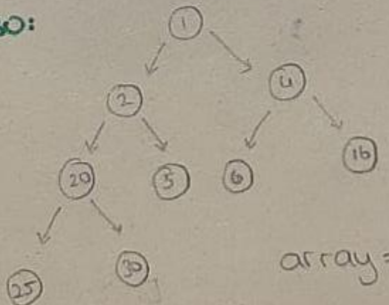
array= [1,2,4,20,5,6,16...]

25:



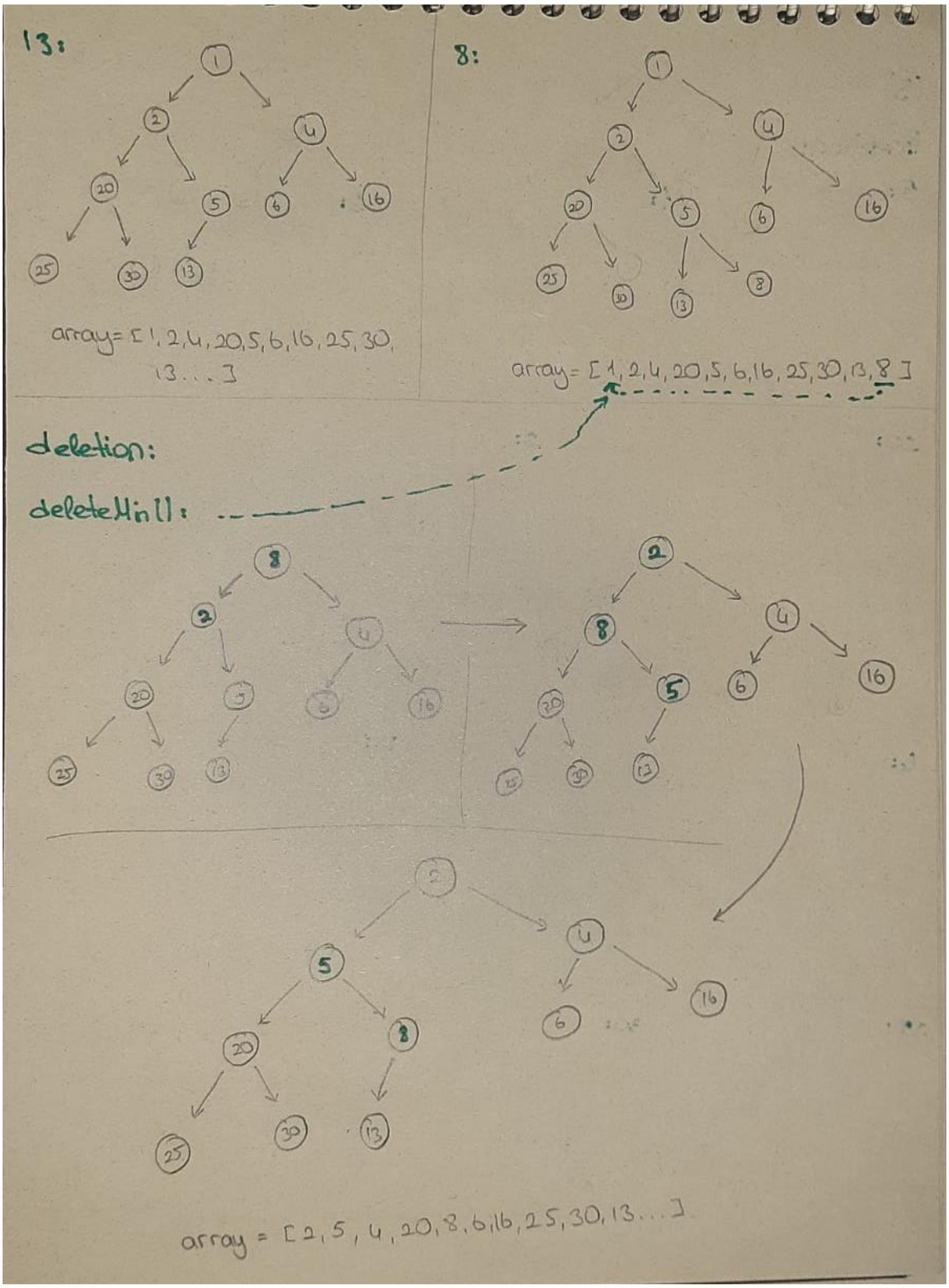
array=[1,2,4,20,5,
6,16,25...]

30:



array= [1,2,4,20,5,6,16,25,30...]

Question 1 (b)



13:

array= [1, 2, 4, 20, 5, 6, 16, 25, 30, 13 . . .]

8:

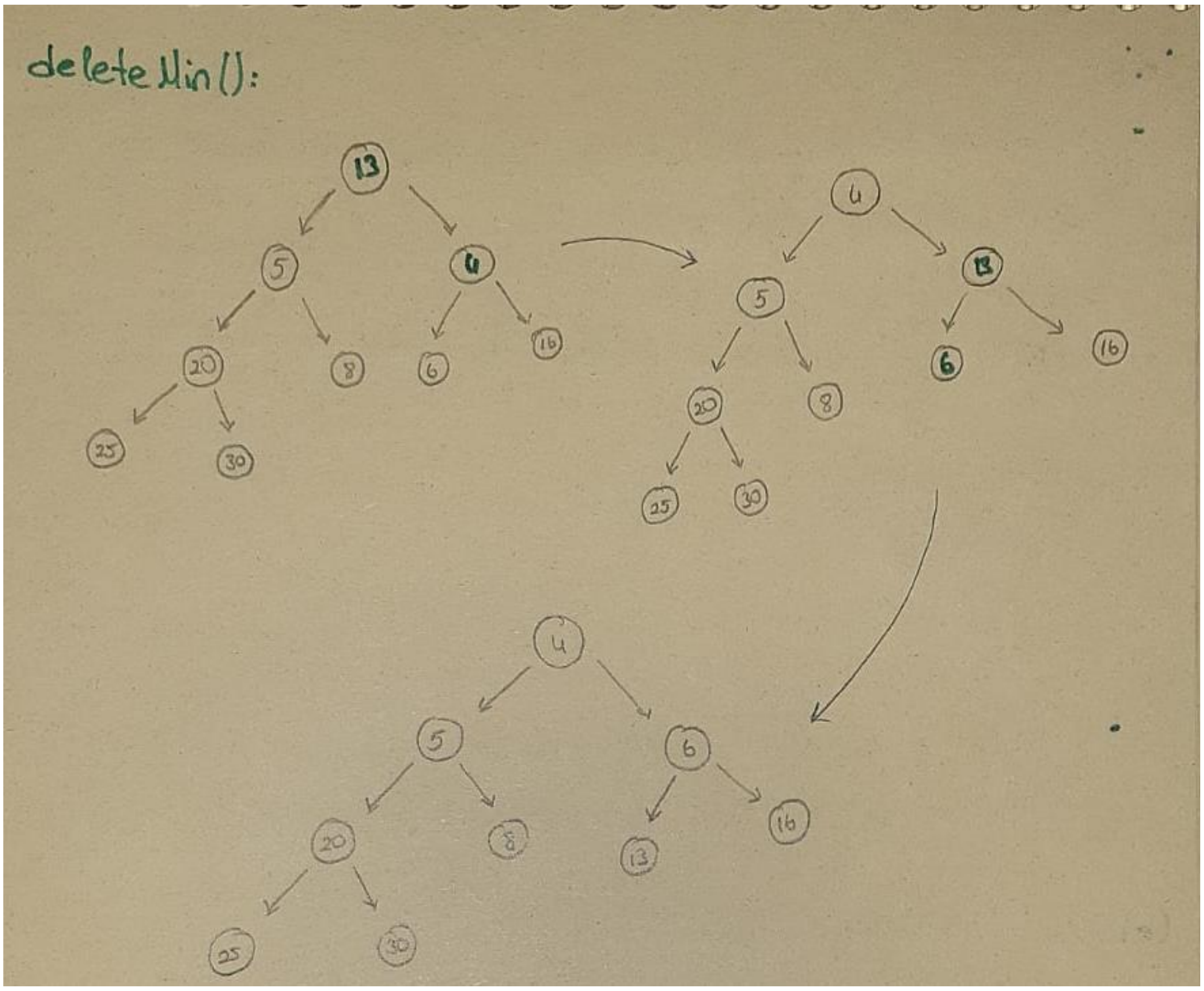
array= [1, 2, 4, 20, 5, 6, 16, 25, 30, 13, 8]

deletion:

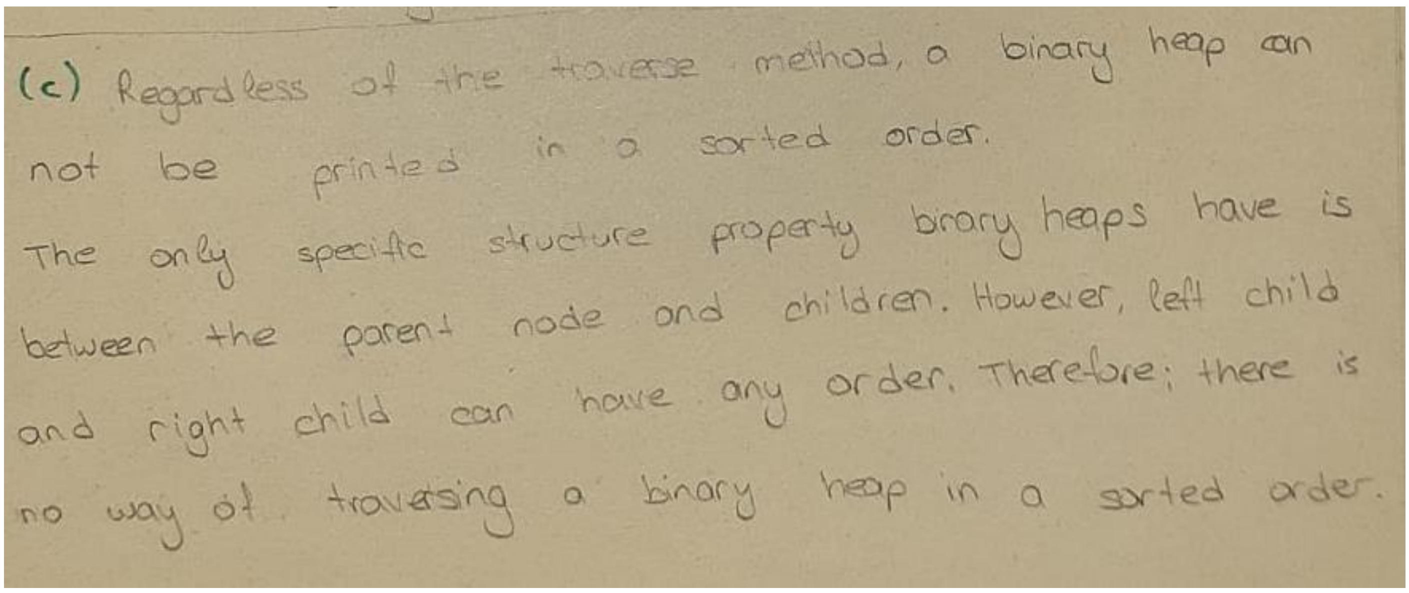
deleteMin():

array = [2, 5, 4, 20, 8, 6, 16, 25, 30, 13 . . .]

Question 1 (b)

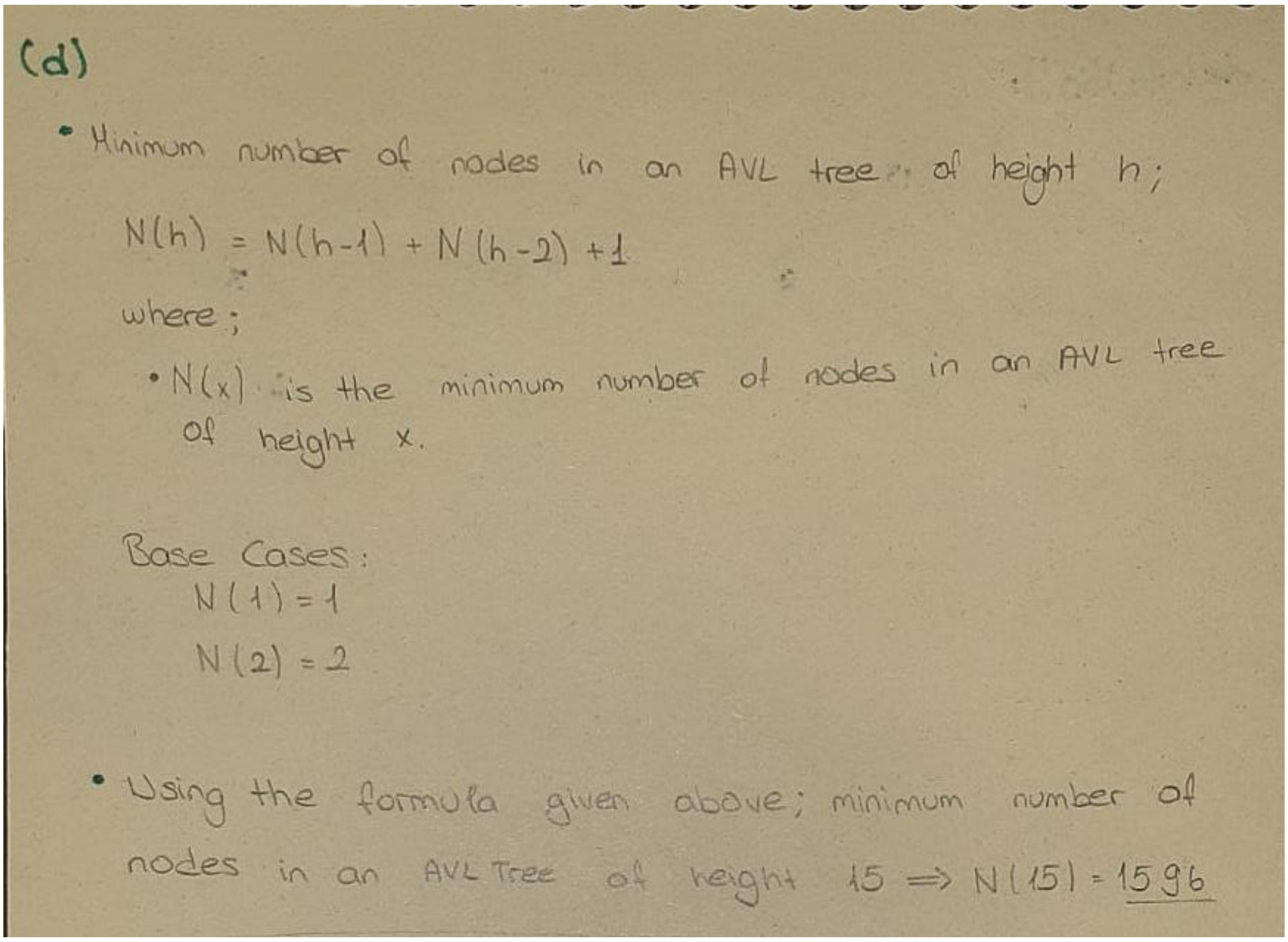


Question 1 (c)



(c) Regardless of the traverse method, a binary heap can not be printed in a sorted order.
The only specific structure property binary heaps have is between the parent node and children. However, left child and right child can have any order. Therefore; there is no way of traversing a binary heap in a sorted order.

Question 1 (d)



(d)

- Minimum number of nodes in an AVL tree of height h;

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

  where;

  - $N(x)$ is the minimum number of nodes in an AVL tree of height x.

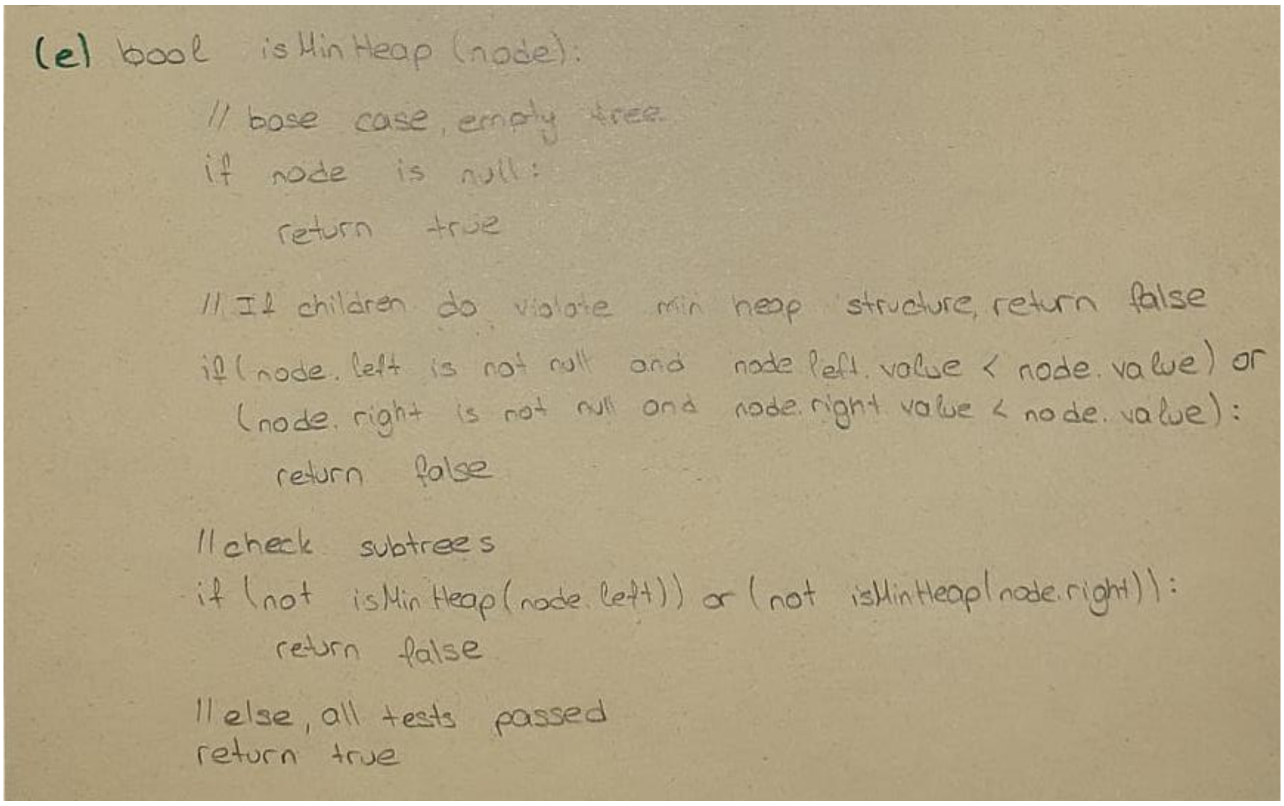
  Base Cases:

  $N(1) = 1$

  $N(2) = 2$

- Using the formula given above; minimum number of nodes in an AVL Tree of height 15 $\Rightarrow N(15) = \underline{1596}$

Question 1 (e)



```
(e) bool isMinHeap(node):
        // base case, empty tree
        if node is null:
            return true

        // If children do violate min heap structure, return false
        if(node.left is not null and node.left.value < node.value) or
          (node.right is not null and node.right.value < node.value):
            return false

        //check subtrees
        if (not isMinHeap(node.left)) or (not isMinHeap(node.right)):
            return false

        //else, all tests passed
        return true
```

Question 2

Heap Data Structure:

A max heap is a specialized binary tree in which each parent holds a value greater or equal to its child nodes. Therefore, the maximum element is always at the root node. However, there is no specific instruction how the child nodes are placed. Furthermore; both right and left subtrees have to be max heaps too.

• The implemented heap class for this assignment contains the following;

• Constructor: heap(){} → initializes the array representation of the heap, sets size to 0

• Destructor: ~heap(){} → deletes the array and sets size to -1.

• Size: int size→the number of nodes in the heap, also last node's index in the array +1

• Array representation: int* my_heap → array representation of the heap (the nodes are placed starting from the lowest level to highest, leftmost node to rightmost node)

• Insertion: void insert(const int){} → if the heap is full, increases the arrays total quota. Then inserts the new value to the end of the array. In a while loop, the new node is moved to its proper place

• Maximum: int maximum(){} → returns -1 if the heap is empty, else; the root nodes value (max element)

• Delete max element: int popMaximum(){} → declares and initializes an integer which will hold the total number of comparisons done while rebuilding process. Returns the helper function popMaximum(int&) with this integer as the parameter.

• Delete max element (helper function): int popMaximum(int&){} → if the heap is empty returns -1. Else; root node's value becomes the last node's (last element of array– highest level rightmost node) value, the size is decreased by 1, and calls the heapRebuild(const int, int&) method. Returns the comparison count.

• Heapify: void heapRebuild(const int, int&){} → The first parameter is the root nodes index in the array (where the rebuilding should start from) and the second parameter is the comparison count (the integer which was used as parameter while calling popMaximum(int&)). Taking the given root's index as the parent node's index, calculating the children's indexes (2*parent+1 and 2*parent+2). Via a comparison, determining which child's value is greater. Via a second comparison, determining if the larger child value is greater than the parent's value (violating the heap structure rule) if so; swapping them and calling the heapsort method again with the larger child's index and same comparison count variable. Which makes maximum of 2 comparisons in one rebuild call.

Heap Sort Function:

void heapsort(heap unsortedHeap, int size, string outp){}; The function provides an efficient way of sorting a max heap in descending order while keeping track of the total number of comparisons made during the process. It takes three parameters: first one represents the heap to be sorted, `size`, indicates the number of elements in the heap, and `outp`, a string representing the output file name.

The function starts by declaring and initializing an integer variable `comp` to zero, which will soon be used to keep track of the total number of comparisons. Then, it opens the output file specified by `outp` using an `ofstream` object and displays an error message if the file opening fails.

The actual sorting process takes place within a for loop that runs `size` times. In each iteration, the function writes the maximum value of the heap, which is found via 'maximum()' method, to the opened output file. It then pops the maximum element from the heap using 'popMaximum()' and adds the returned number of 'popMaximum()' to `comp` variable to keep track of the number of comparisons made.

After the for loop, the function writes a statement into the file about the total number of comparisons and appends 'comp' to it. Lastly, the function closes the output file.

Given data examples and their total number of comparisons:

N=	**10^3**	**$2*10^3$**	**$3*10^3$**	**$4*10^3$**	**$5*10^3$**
Number of Comparison=	11034	25097	40232	56164	72385

Conclusion:

The total number of comparisons given above are less than the worst case in which all reBuild() calls would end up increasing the number of comparisons by 2 as it is expected in an average case. Meaning, heap data structure violations do not occur in every level of every single popMaximum() operation thanks to the randomness of the data being read into the heap.