Alexandria University
Faculty of Engineering

Assignment NO. 5

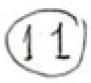
By:

Ali Mohamed Taima Hummus

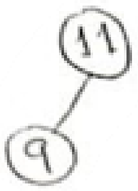
Question1:

a)

Insert(11)

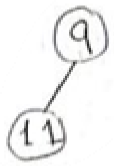


Insert(9)



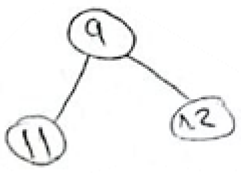
9<11

Heapify[swap(9,11)]



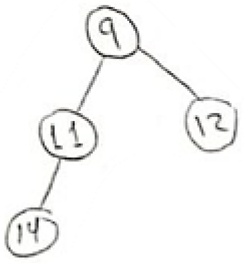
MinHeap

Insert(12)



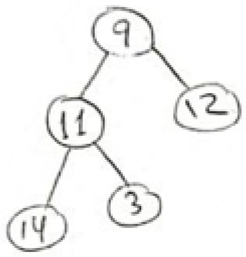
MinHeap

Insert(14)



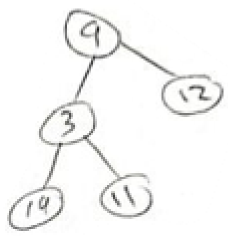
MinHeap

Insert(3)



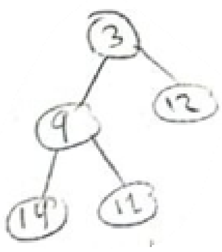
3<11

Heapify[swap(3,11)]



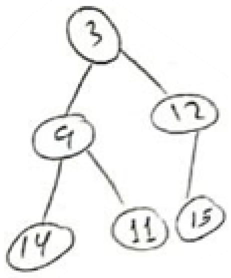
3<9

Heapify[swap(3,9)]



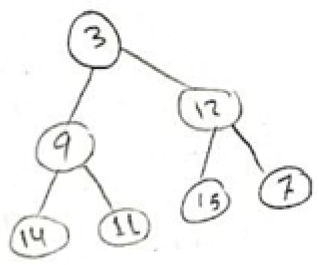
MinHeap

Insert(15)



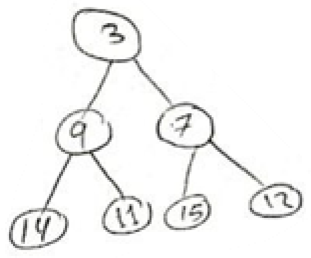
MinHeap

Insert(7)



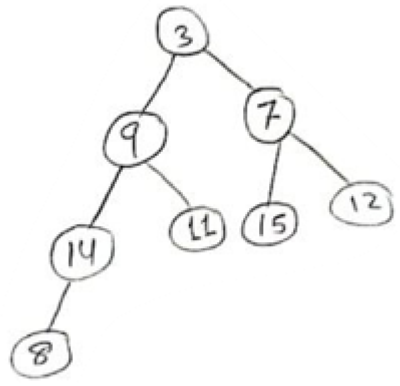
7<12

Heapify[swap(7,12)]



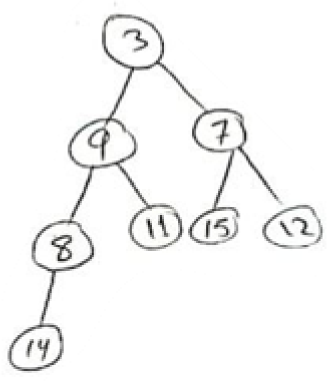
MinHeap

Insert(8)



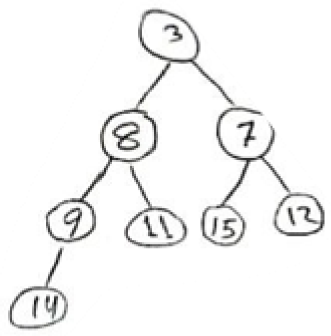
8<14

Heapify[swap(8,14)]



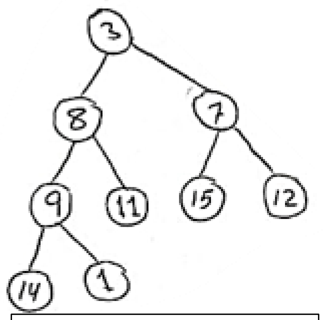
8<9

Heapify[swap(8,9)]



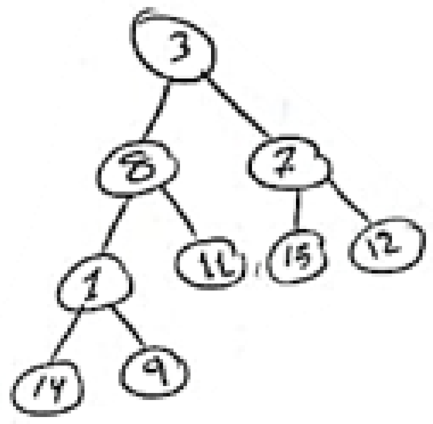
MinHeap

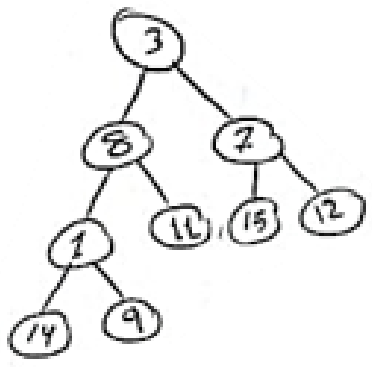
Insert(8)



1<9

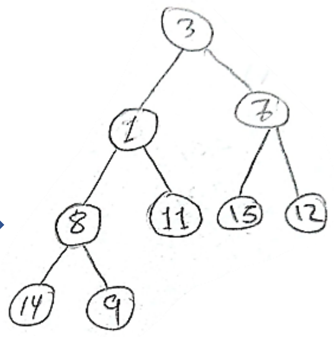
Heapify[swap(1,9)]





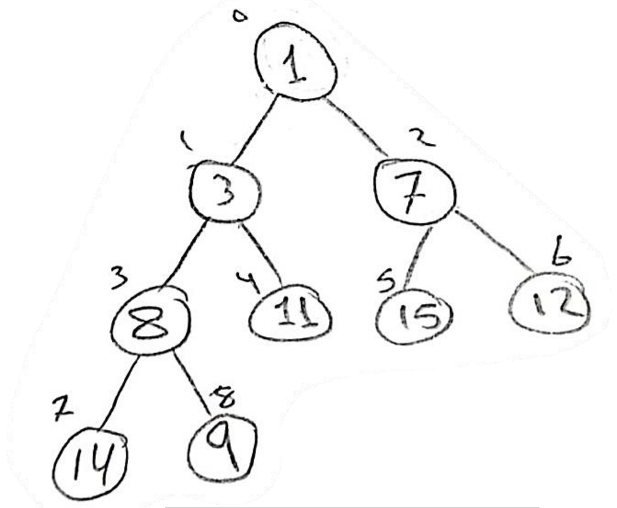
1<8

Heapify[swap(1,8)]



1<3

Heapify[swap(1,3)]



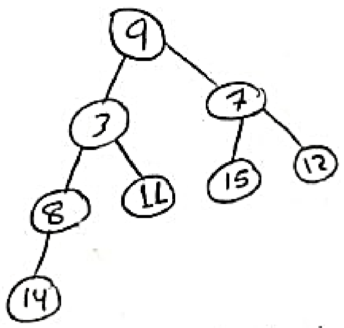
The final tree is MinHeap

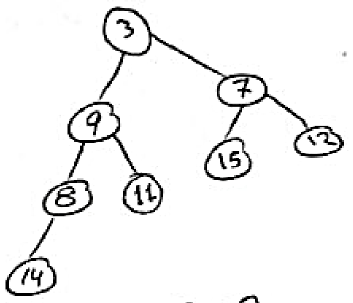
b)

1b)

Delete min (1)

- To delete (1), I should swap(Last element(9), 1) and delete (1)





- The result is not a min heap

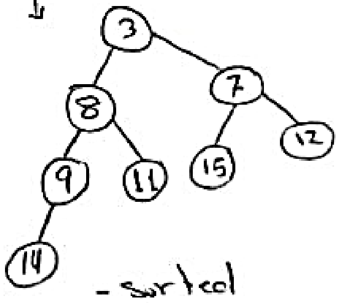
∵ 3<9 & 7<9 but (3<7)

∴ swap(3,9)

∵ 8<9

∴ swap(8,9)

↓

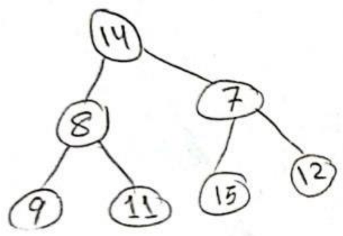


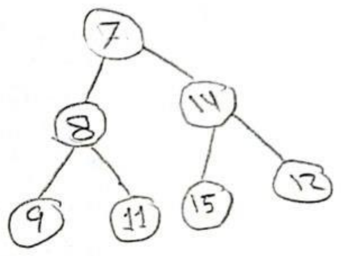
- sorted

(b)

Delete min (3)

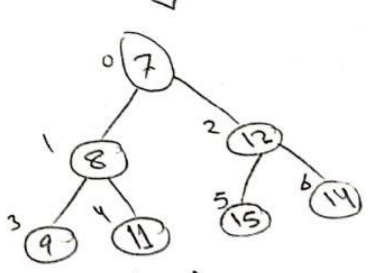
- To delete (3), I should swap (last element (14), 3) and delete (3)





- The result isn't a min heap
  ∵ 8<14 & 7<14 but (7<8)
  ∴ swap(7,14)

∵ 12 < 14
∴ swap(12,14)

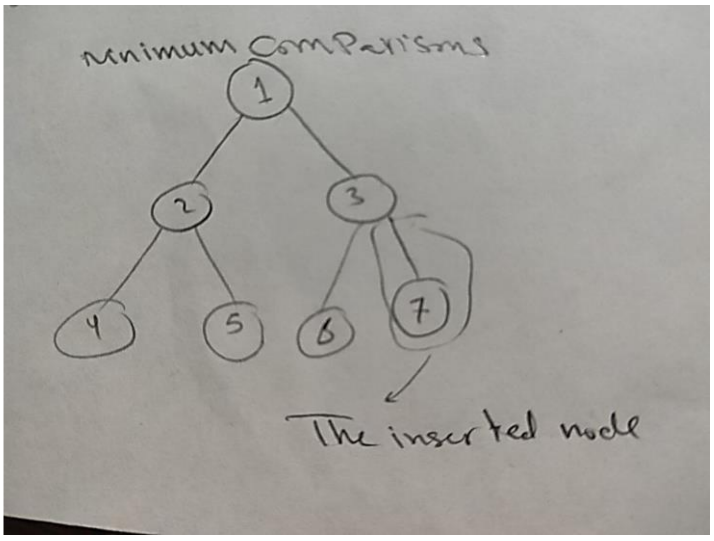


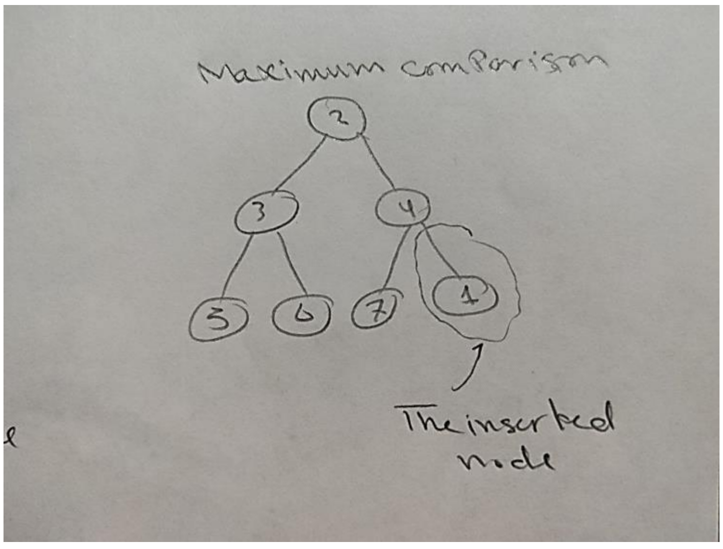
- Sorted

c)

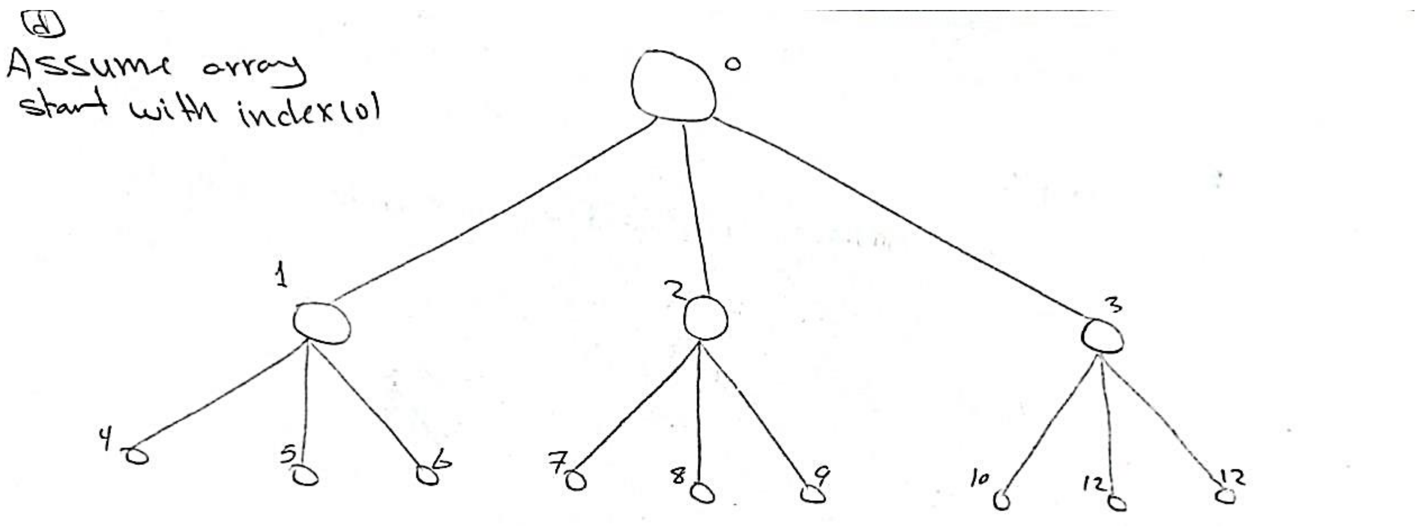
When we insert a new node there are options:

1. The new node doesn't change the properties of the Min-heap and in this case we will make Minimum comparisons and it equal to O(1)
2. The new node distort the Min-heap properties and in this case, we will need to make equal to the height of our balanced tree that equal to logN, the maximum comparisons will equal to O(logN)





d)



- For Parent
    - $\lfloor (i-1)/3 \rfloor$
        - We can generalize this formula for any K and it will be $\lfloor (i-1)/k \rfloor$
- For children
    - For left child
        - $(3 \times i) + 1$
    - For Middle child
        - $(3 \times i) + 2$
    - For right child
        - $(3 \times i) + 3$
    - We can generalize the formula of children for any k
        - $(k \times i) + 1, (k \times i) + 2, (k \times i) + 3, \dots \dots \dots \dots, (k \times i) + k$

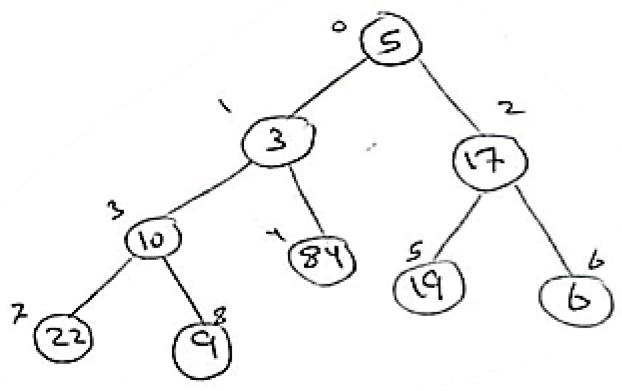
Question2:

a)

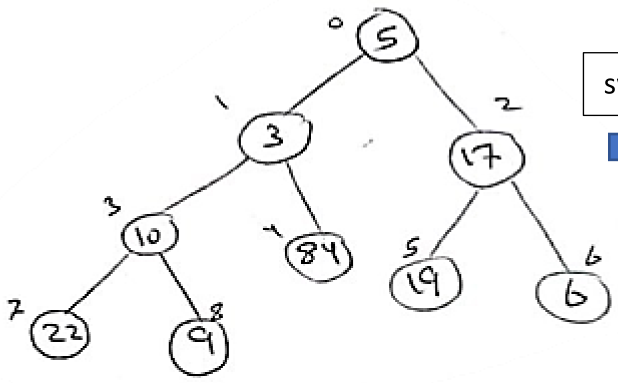
In both cases the complexity will be $O(nlog(n))$, because even though the array is sorted the algorithm will deal with it as nature, it will build the heap and sort it
and it the array is not sorted the complexity will be $O(nlog(n))$

b)

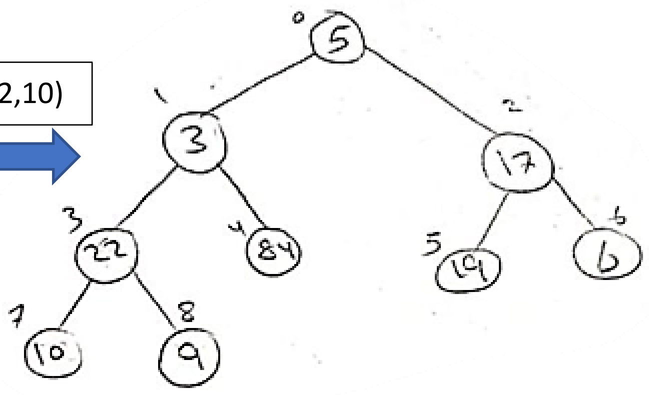
The default type for Heap is a MaxHeap, so we suppose that we will build a MaxHeap



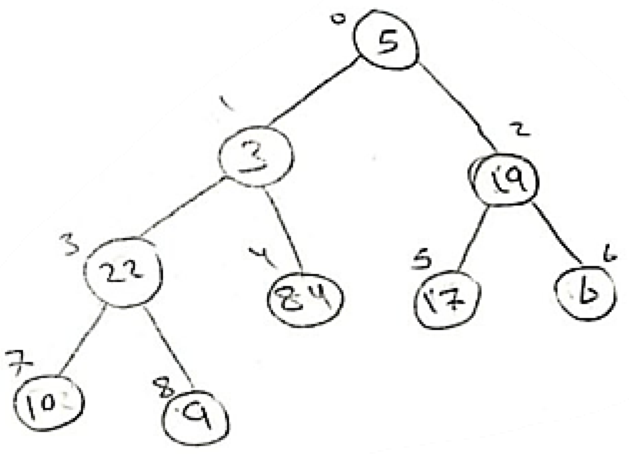
- To build a Heap, we will start from array[(n/2 – 1)] = array[(9/2-1)] = array[3] and make Heapify until array[0]
- So start form array[3]
    - For array[3]
        - 22>10
            - Heapify[swap(22,10)]



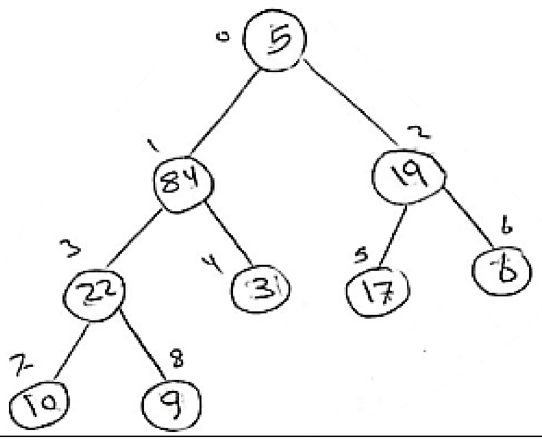
swap(22,10)



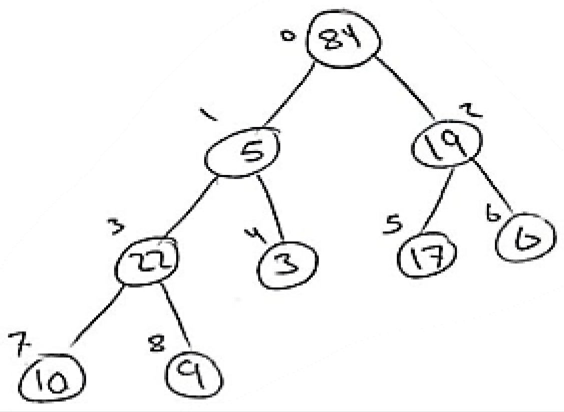
- For array[2]
    - 19>17
        - Heapify[swap(19, 17)]



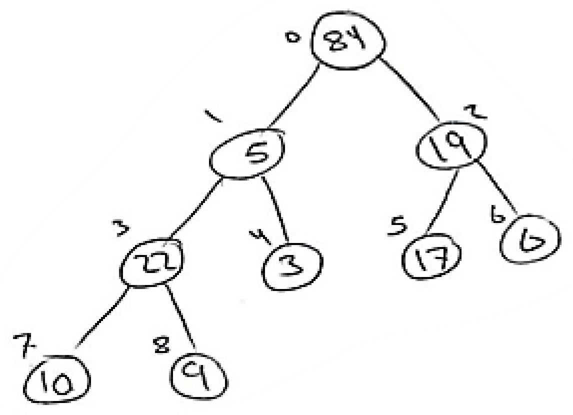
- For array[1]
    - 22>3 and 84>3, so we get Max(84, 22) = 84
        - Heapify[swap(84, 3)]



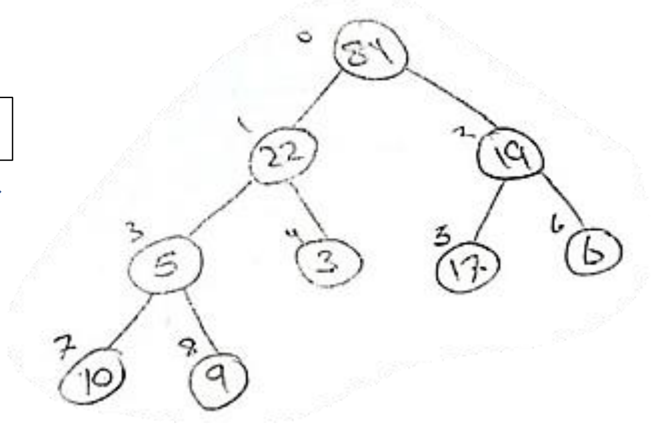
- For array[0]
    - 84 > 5 and 9>5, so get max(84, 9) = 84
        - Heapify[swap(84, 5)]



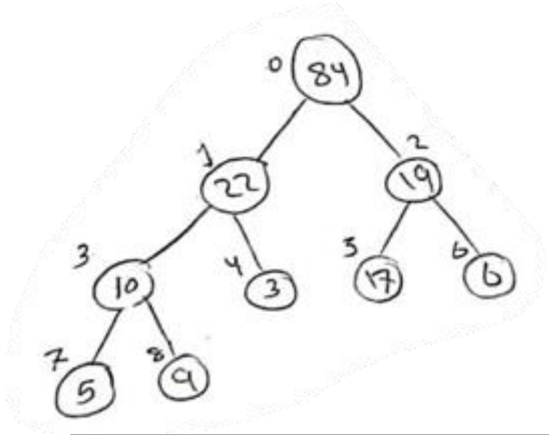
- For array[1]
    - 22>5
        - Heapify[swap(22, 5)]



swap(22,5)

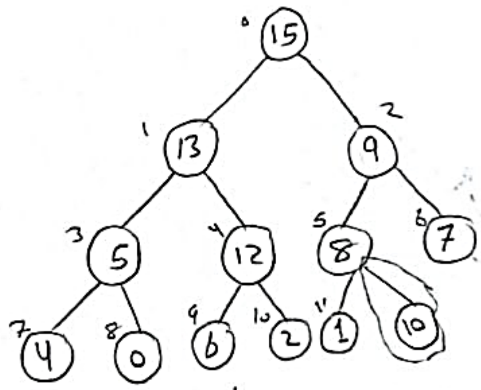


- For array[3]
    - 10 >5 and 9>5, so get max(10,9) = 10
        - Heapify[swap(10, 5)]



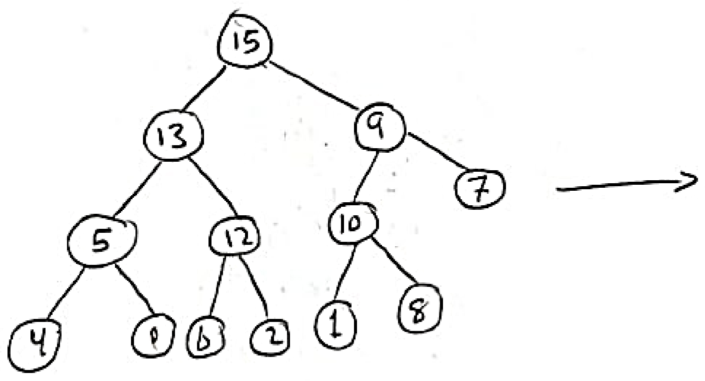
MaxHeap satisfy all conditoins

c)

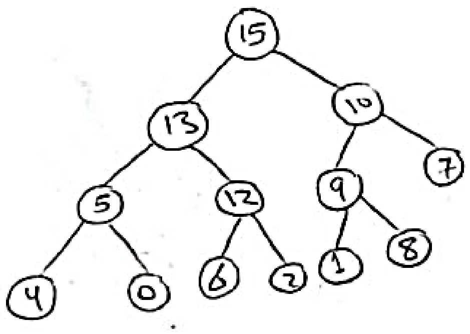


(c)

- After insert (10), I find that 10 > 8, so I should make a heapify

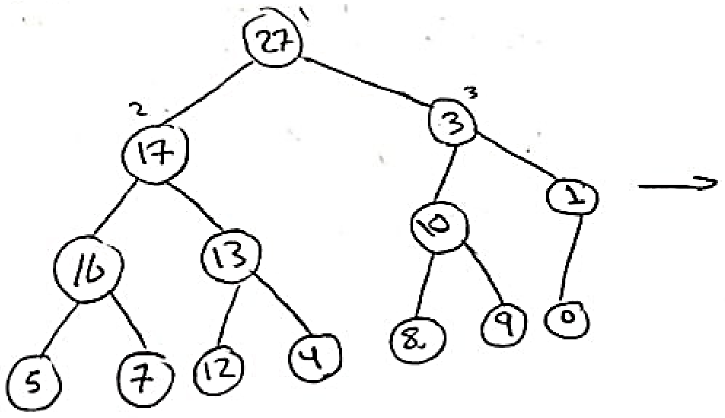


10 > 9
- swap(10, 9) (Heapify)

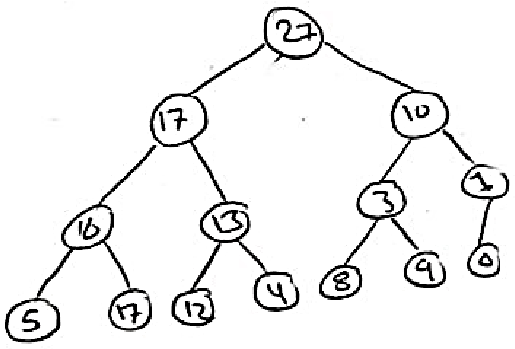


- Sorted

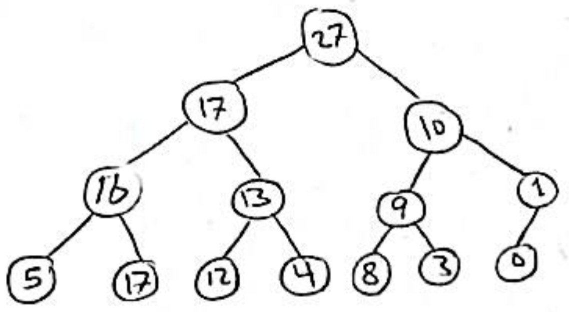
d)



∵ 10 > 3
∴ heapify[swap(10, 3)]



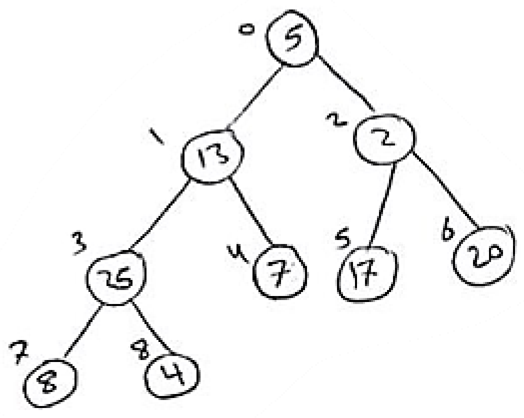
∵ 8 > 3 & 9 > 3, so take
max(8, 9) = 9
∴ heapify[swap(9, 3)]



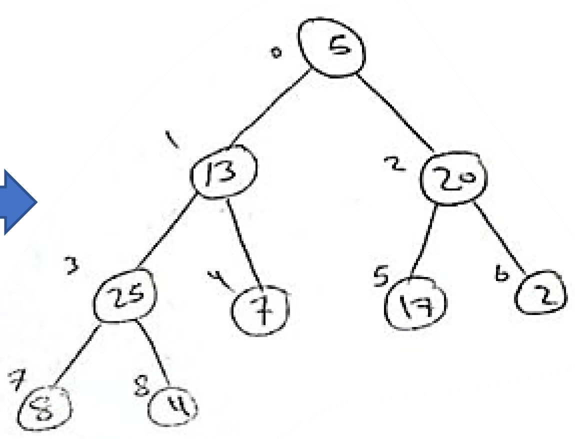
- Sorted

e)

The default type for Heap is a MaxHeap, so we suppose that we will build a MaxHeap

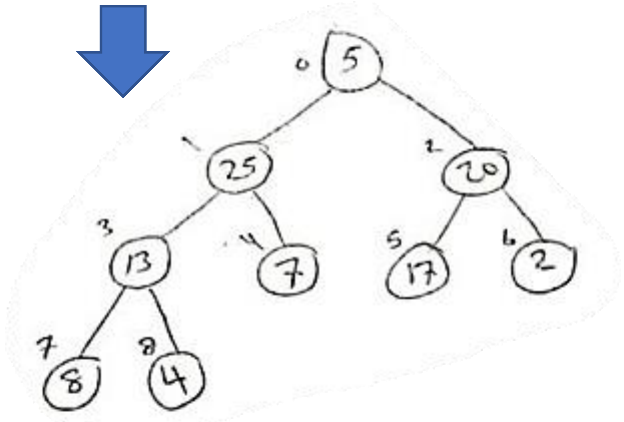


swap(20, 2)

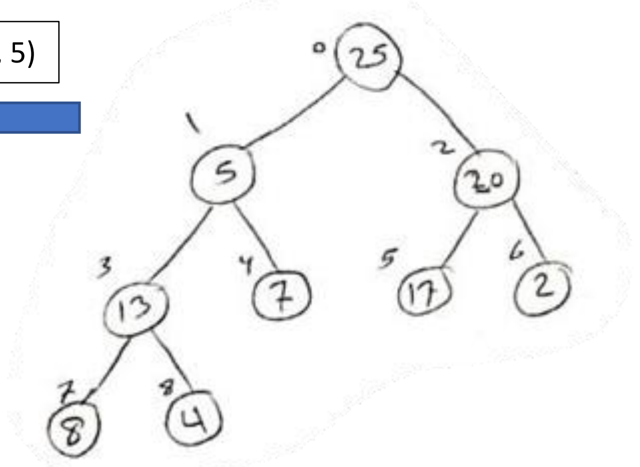


- First we will bild a MaxHeap by applying heapify from array[(n/2 – 1)] = array[(9/2-1)] = array[3] and make Heapify until array[0]
- For array[3]
  - 25>4 and 25>4
    - MaxHeap
- For array[2]
  - 17>2 and 20>2, so get Max(20, 17)= 20
    - Heapify[swap(20, 2)]

- for array[1]
  - 25>3
    - Heapify[swap(25,13)]

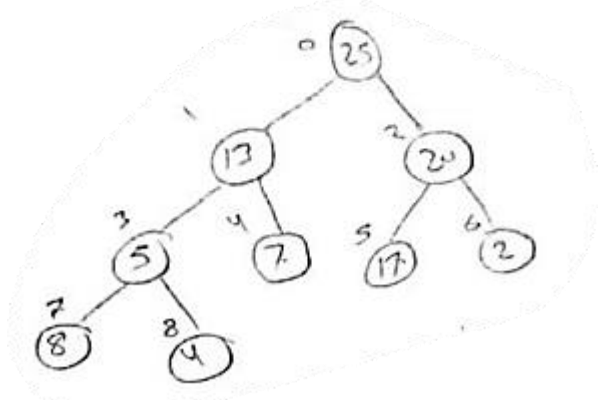


- For array[0]
  - 25>5 and 20>5, so get max(25,20) = 25
    - Heapify[swap(25,5)]

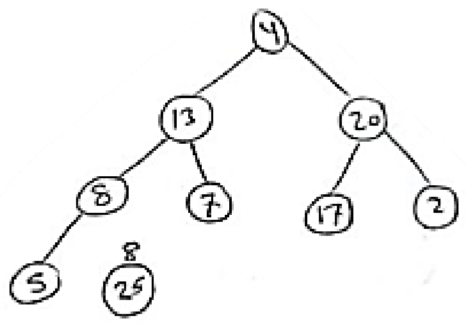


- For array[1]
  - 13>5 and 7>5, so get Max(13, 7) = 13
    - Heapify[swap(13, 5)]

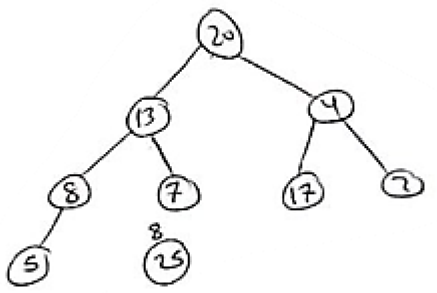
swap(13, 5)



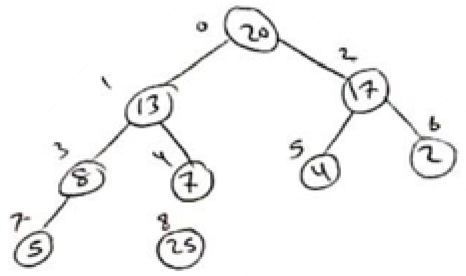
- MaxHeap, so we want to make a heap sort
- To make a heap sort we will swap array[0] and array[size – 1] and each swap we will decrement the size by 1 and stop until the size become 1
- So first swap(arr[0], arr[8])
  - Size = 8



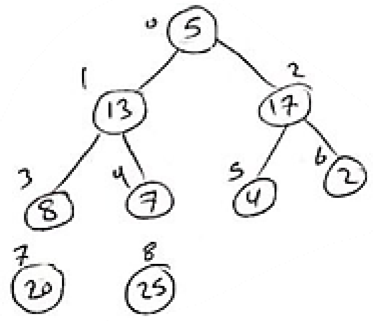
Heapify[swap(20, 4)]



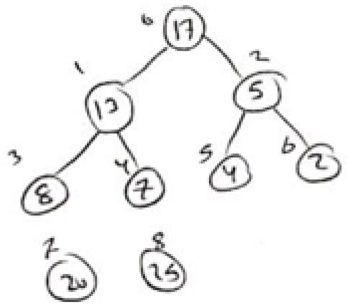
Heapify[swap(17, 4)]



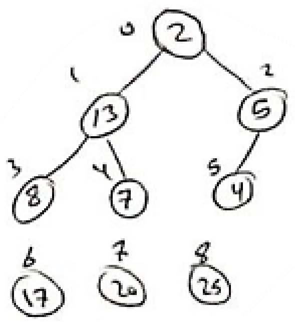
- Get min[swap(arr[0], arr[7])]
    - Size = 7

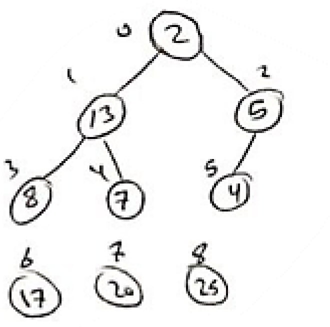


Heapify[swap(17, 5)]

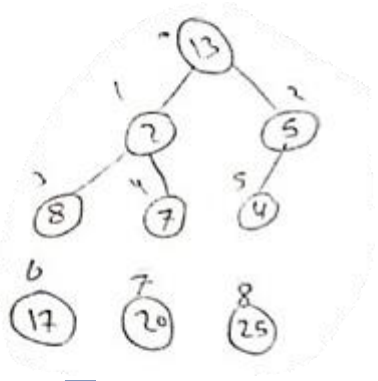


- Get min[swap(arr[0], arr[6])]
    - Size = 6

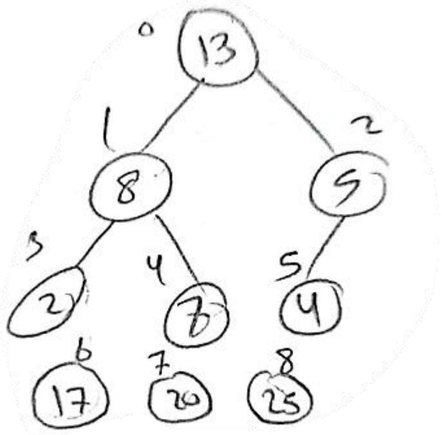




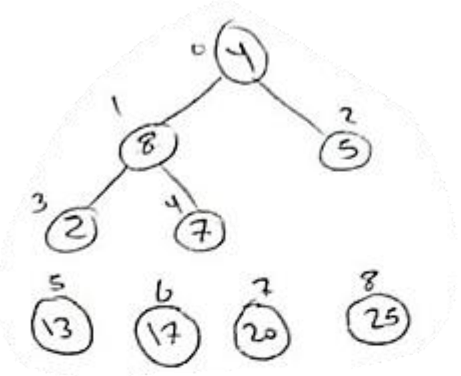
Heapify[swap(13, 2)]



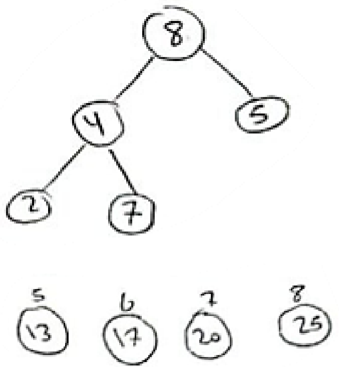
Heapify[swap(8, 2)]



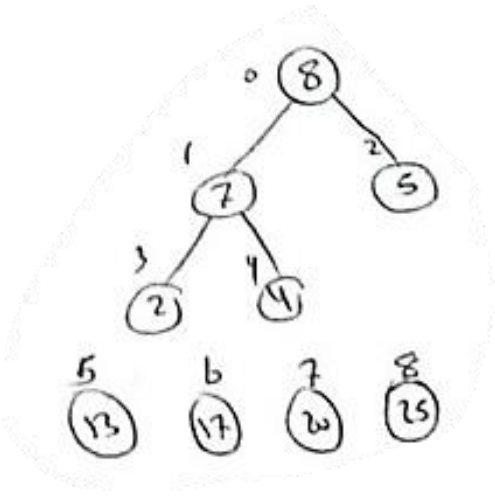
- Get min[swap(arr[0], arr[5])]
  - Size = 5

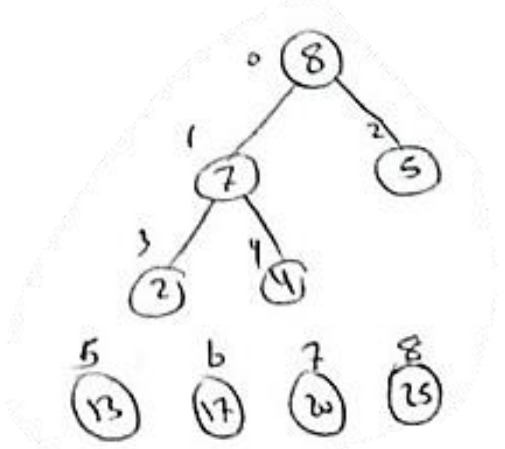


Heapify[swap(8, 4)]

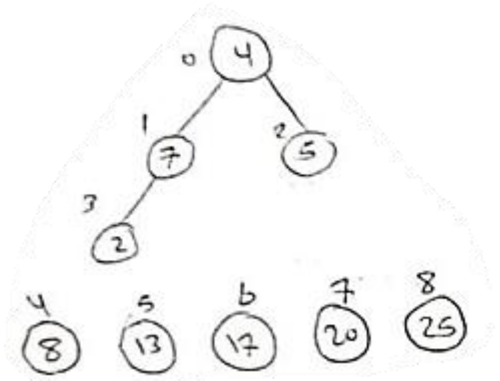


Heapify[swap(7, 4)]

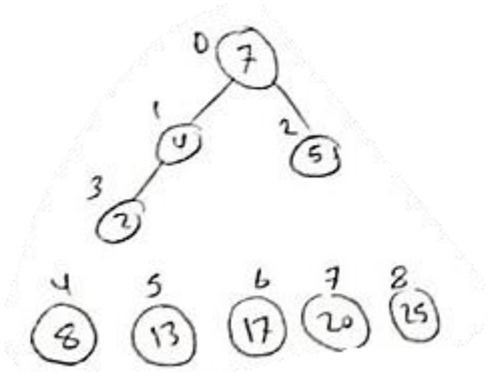




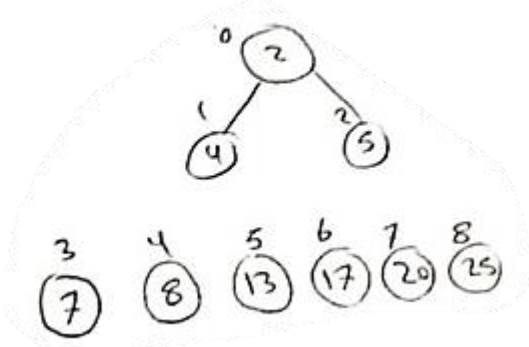
- Get min[swap(arr[0], arr[4])]
    - Size = 4



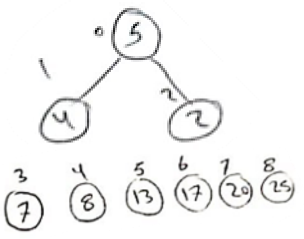
Heapify[swap(7, 4)]



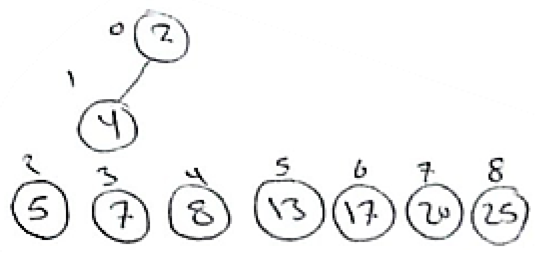
- Get min[swap(arr[0], arr[3])]
    - Size = 3

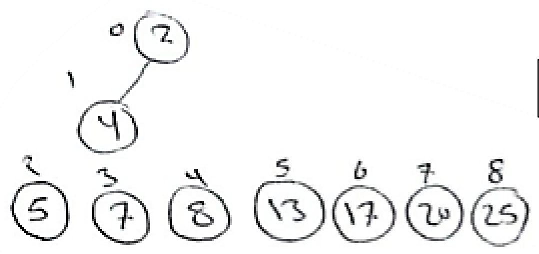


Heapify[swap(5, 2)]

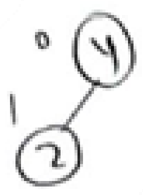


- Get min[swap(arr[0], arr[2])]
    - Size = 2



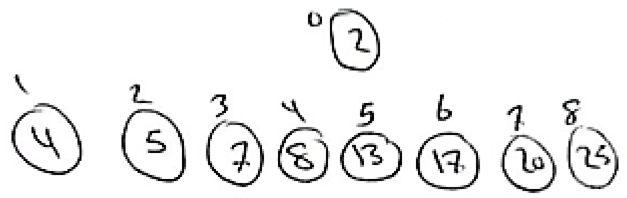


Heapify[swap(13, 2)]



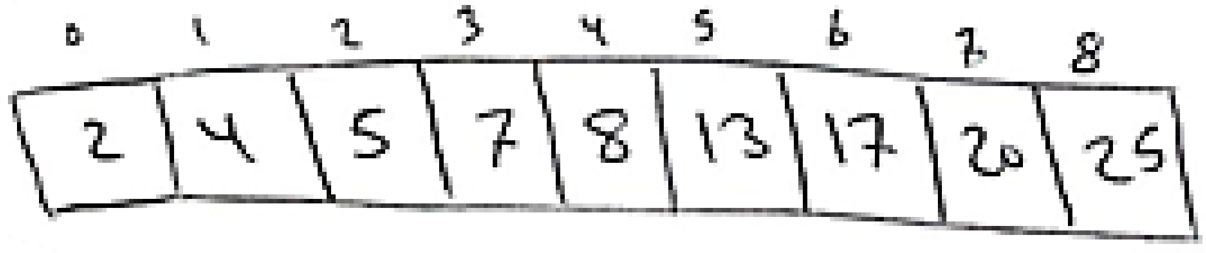
Heapify[swap(4, 2)]

- Get min[swap(arr[0], arr[1])]
    - Size = 1

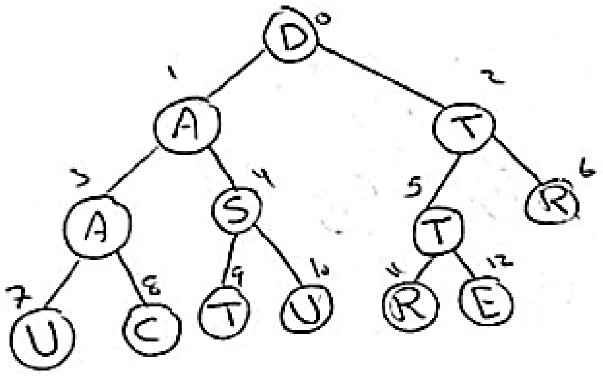


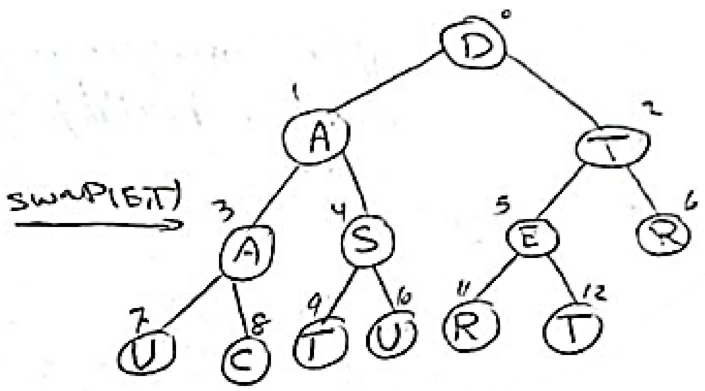
Here we find that the size bcome 1, so we should stop

The final sorted array



f)

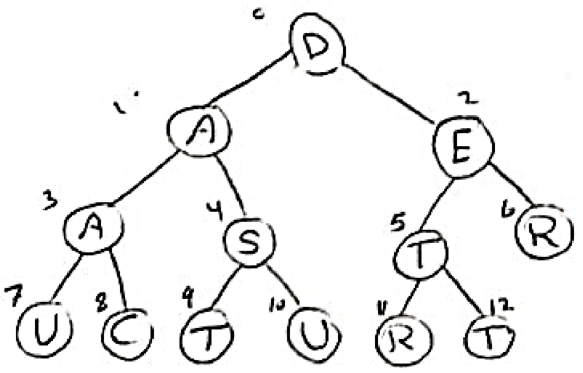




- I suppose array is zero-index
- To bild Min-heap I will apply heapify from $\left(\frac{n}{2}\right) - 1$ to 0
- $\left(\frac{13}{2}\right) - 1 = 6 - 1 = 5$ so we will start from array[5] to array[0]
- For array[5]
    - R<T and E<T, so we take Min(E, R) = E
        - Heapify[swap(E, T)]

- For array[4]
    - S<T and S<U
        - Satisfy Min-heap
- For array[3]
    - A<U and A<C
        - Satisfy Min-heap
- For array[2]
    - E<T and R<T, so we will take Min(E, R) = E
        - Heapify[swap(E, T)]

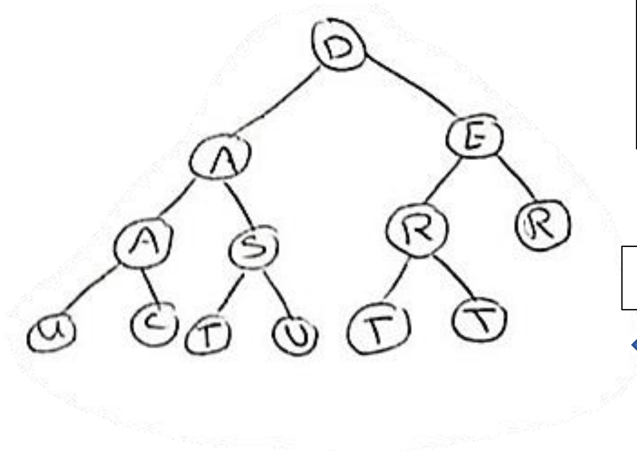
swap(E, T)

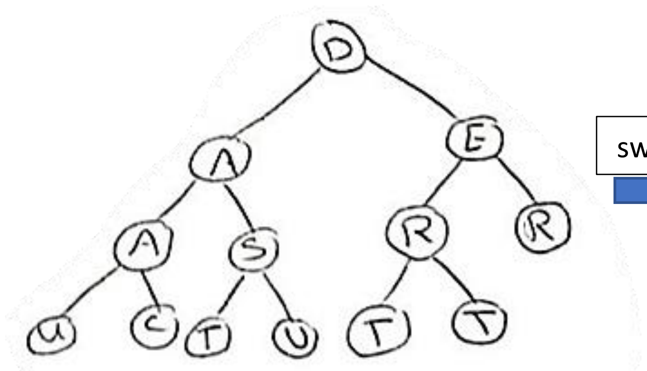


When we make a heapify we should check the child that we changed and show it still satisfy Min-heap condition or not, if not satisfy we make a heapify for it and so on.

- For array[5]
    - R<T
        - Heapify[swap(R, T)]

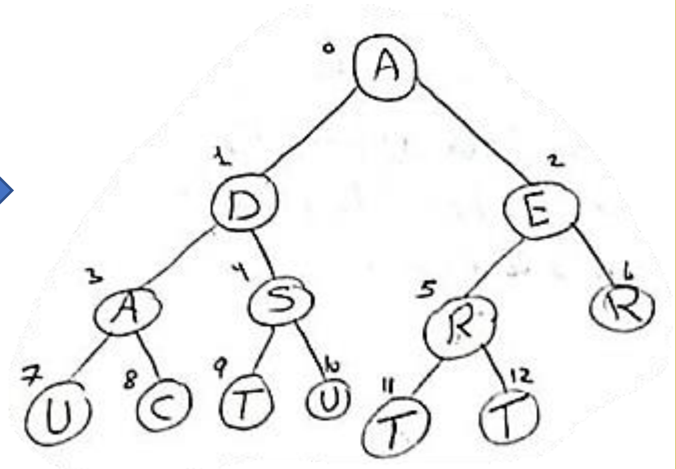
swap(R, T)





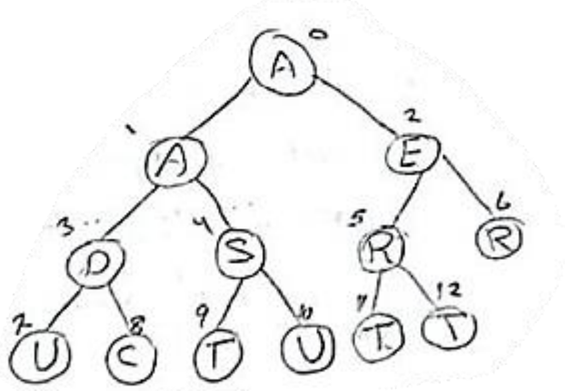
- For array[1]
    - A == A and A<S
        - Satisfy Min-heap
- For array[0]
    - A<D
        - Heapify[swap(A, D)]

swap(A, D)



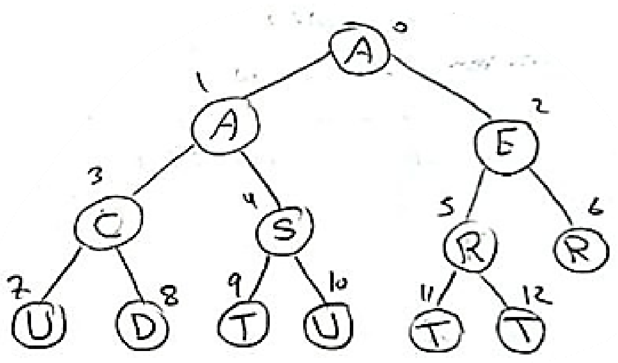
- As we say before we should check the child the changed
- For array[1]
    - A<D
        - Heapify[swap(A, D)]

swap(A, D)



- For array[3]
    - C<D
        - Heapify[swap(C, D)]

swap(C, D)



This is the final tree that satisfy all heap conditions

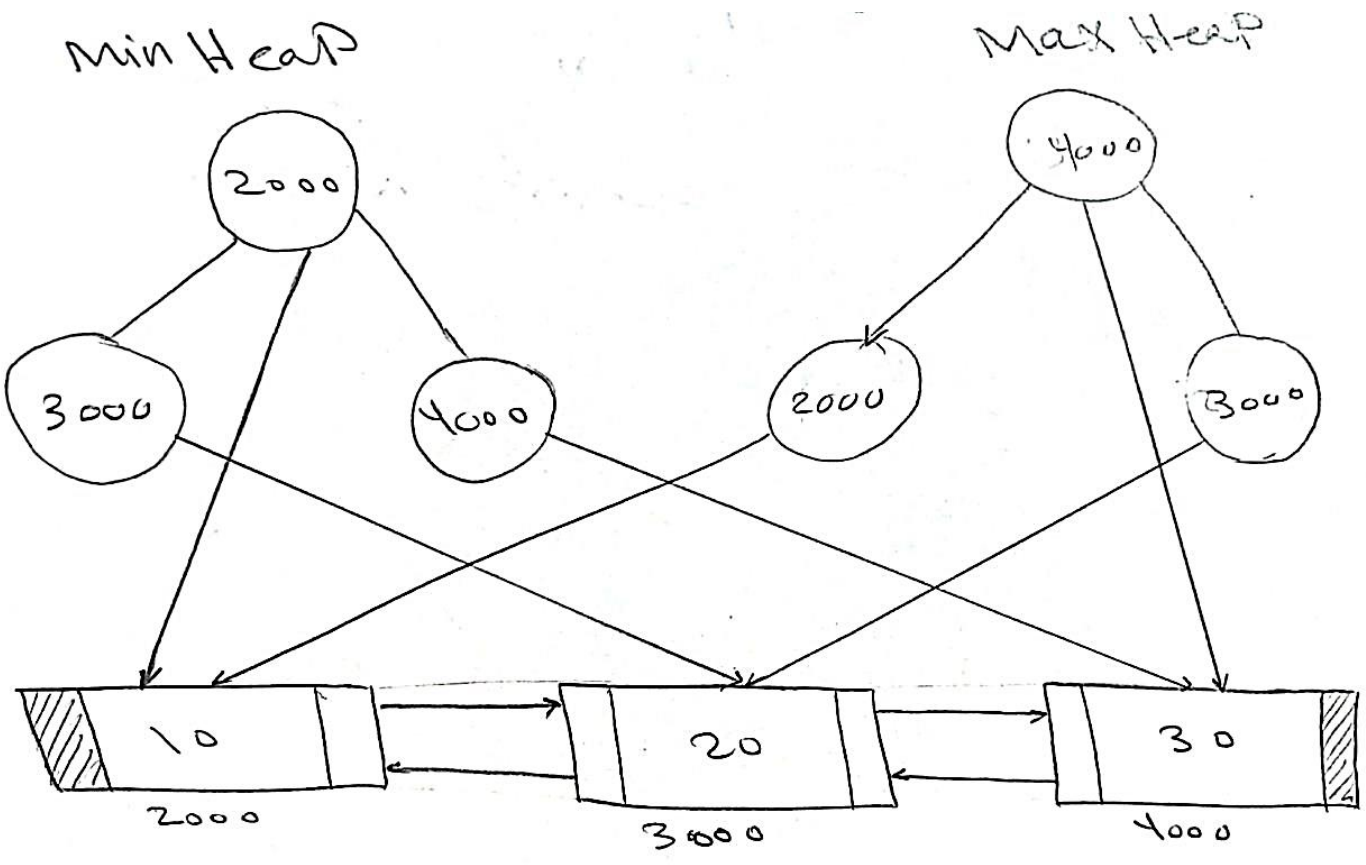
Question3:

a)

To delete the node, we will delete it then make a heapify to save the heapify order property even though the element below it satisfty all the heap order conditions, I should check them to make sure, so it will take O(log(n)) in all cases

b)

- I can combine min-Heap and max-Heap, but the problem I face that I should make a synchronization between the two heaps(which mean when I delete an element from any heap I should delete it also from the other heap.
- To solve this problem I use a double linked list(I select double linked list because it easy to delete an element from the middle compared to single linked list, and save memory compared to array).
  in double linked list I will put the elements and synchronous them in minheap and maxheap by passing the addresses of elements to them and order them according to the heap type
  - For example minimum element will be in the double linked list and there is a pointer in the root of minheap contain the address of this element and similarly for maxheap
- The complexity for each operation will be:
  - Insert()
    - We will insert at the beginning of linked list and the complexity will be O(1), but to insert the address in the minHeap and maxHeap take O(log(n)), so the overall complexity is O(log(n))
  - deletMin()
    - We get the address of minimum value node from root of Min Heap. We use this address to find the node in double linked list. From the doub linked list, we get node of maxHeap. We delete node from all three. We can delete a node from double linked list in O(1) time. delete() operation for max and min heaps take O(log(n)) time, so the overall complexity is O(log(n))
  - deletMax()
    - is similar to deleteMin(), We get the address of maximum value node from root of max Heap. We use this address to find the node in double linked list. From the doub linked list, we get node of minHeap. We delete node from all three. We can delete a node from double linked list in O(1) time. delete() operation for max and min heaps take O(log(n)) time, so the overall complexity is O(log(n))



c)

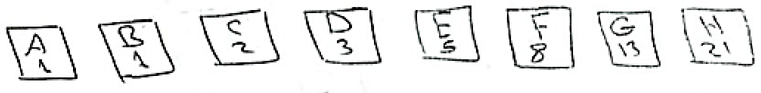
We can achieve that by make a sorted array where the smallest element is at the first position and largest element is at the last position, but the complexities of insert(), deleteMin(), deleteMax() will be O(n) instead of O(log(n)) when we using heaps in the previous example.

Question4:

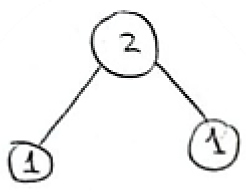
a)

First we should find the Huffman code for each character to decode the binary that given

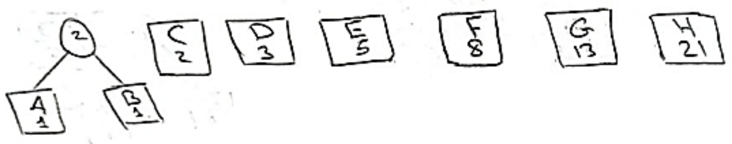
Step(1): we should sort elements according to their frequencies, but the given sequence is already sorted



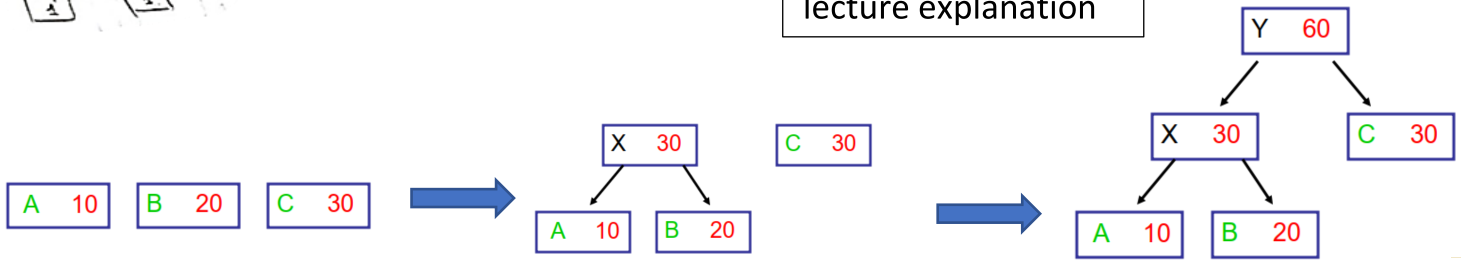
Step(2): take the first two minimum frequencies and create a parent for them contain their summation



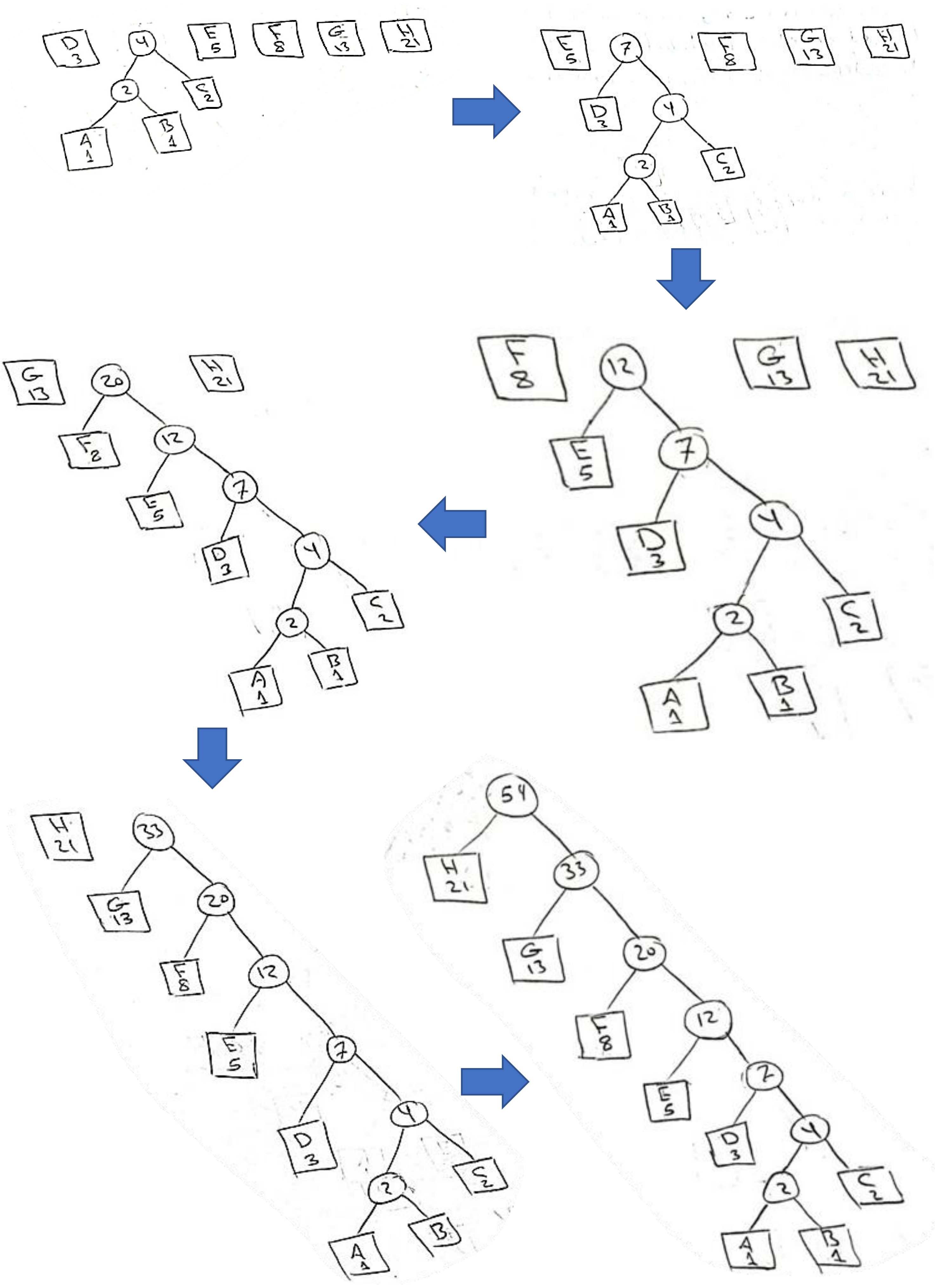
Step(3): sort the new sequence

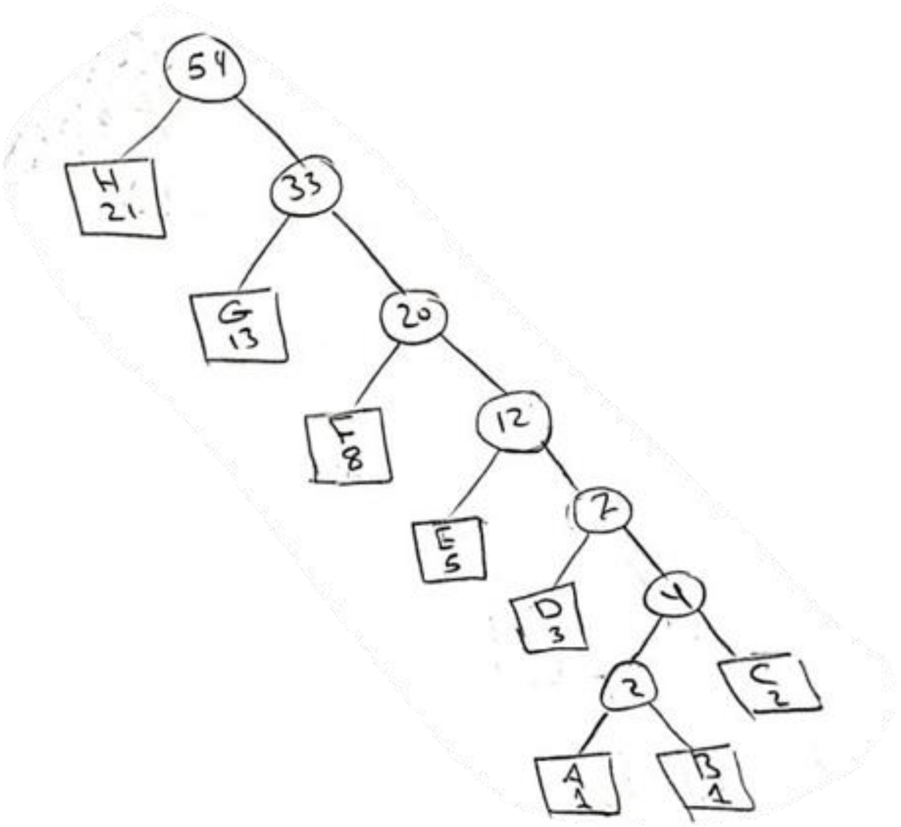


I am working on this problem and problem (e) according to the lecture explanation

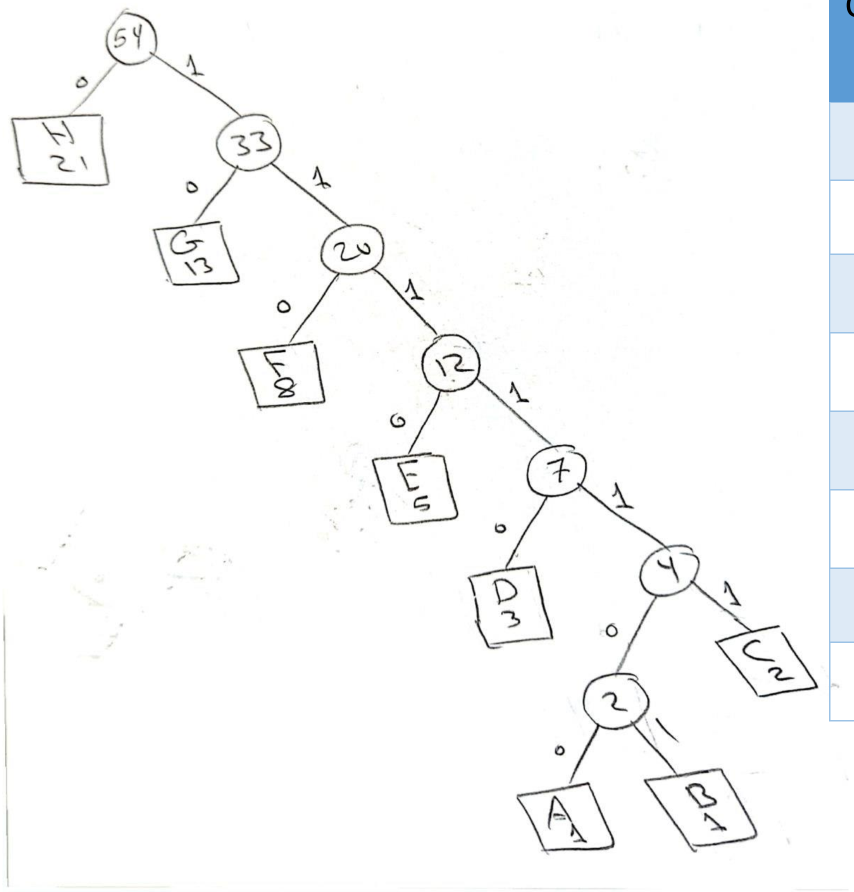


Step(4): repeat step(2) and step(3) until construct a tree from the sequence





This is the final result and we will name the left edge with "0" and right edge with "1"



Char	Frequency	Chode	Number of bits
A	1	1111100	7
B	1	1111101	7
C	2	111111	6
D	3	11110	5
E	5	1110	4
F	8	110	3
G	13	10	2
H	21	0	1

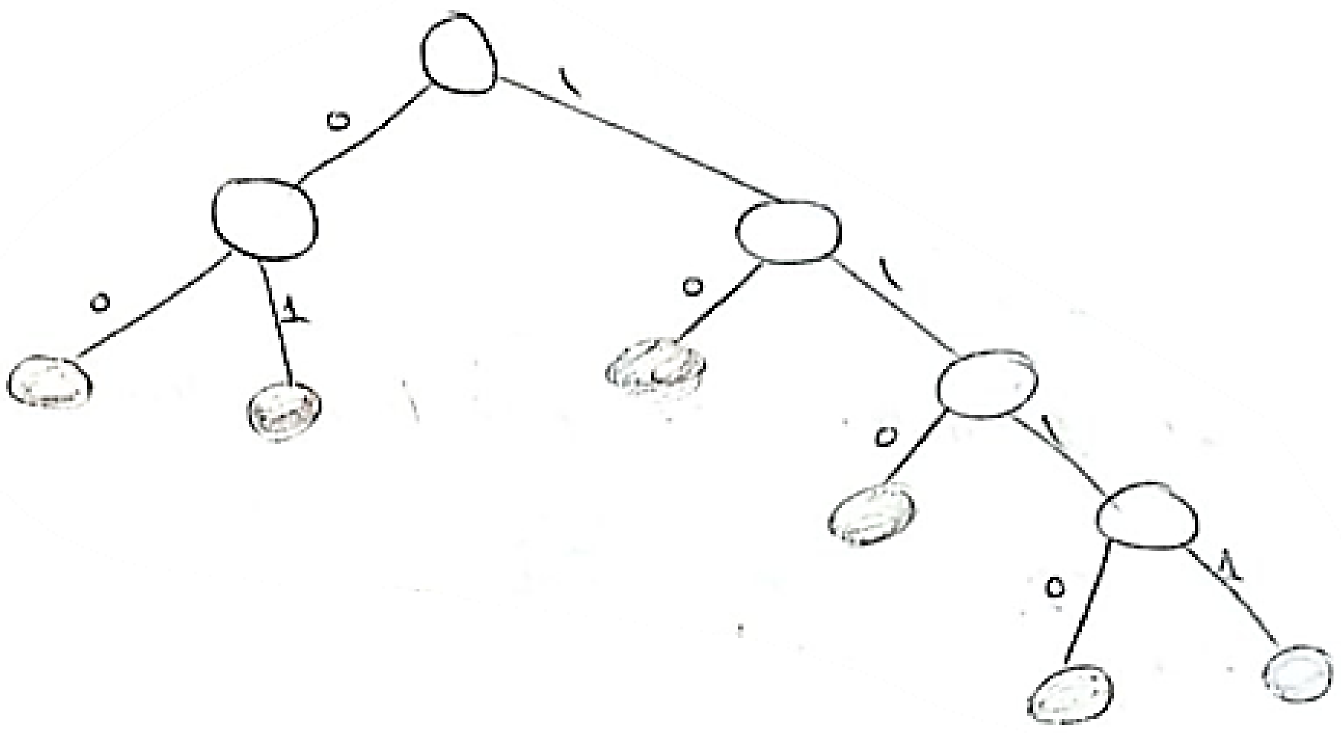
After generating the code for each character we can decode the given code

111110011111101111101110 = **ACHDE**

1111100	111111	0	11110	1110
A	C	H	D	E

b)

- The **sum** of **frequencies** of **first 4 elements** should be **greater** than the **sum** of **frequencies** of **last two elements**
- Let assume the **frequencies** of first four elements are **A, B, C, D** and the **frequencies** of last two elements are **E, F**
    - So the relation will be **A + B + C + D > E + F**
- Ther are many choices for each frequency, but I will assume some number for each frequency: A = 10, B = 25, C = 30, D = 40, E = 45, F = 50.
    - If we add A + B + C + D = 105 and E + F = 95, **A + B + C + D > E + F** as we say



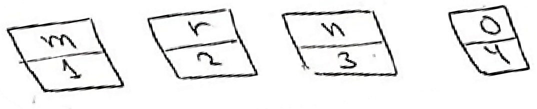
c)

noomorrnon

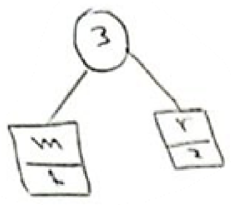
first method

I will use Huffman coding:

Char	Frequency
m	1
r	2
n	3
O	4

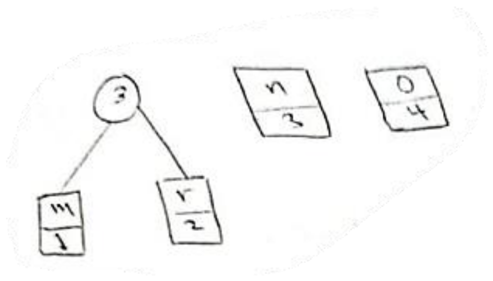


Step(1): I will sort them

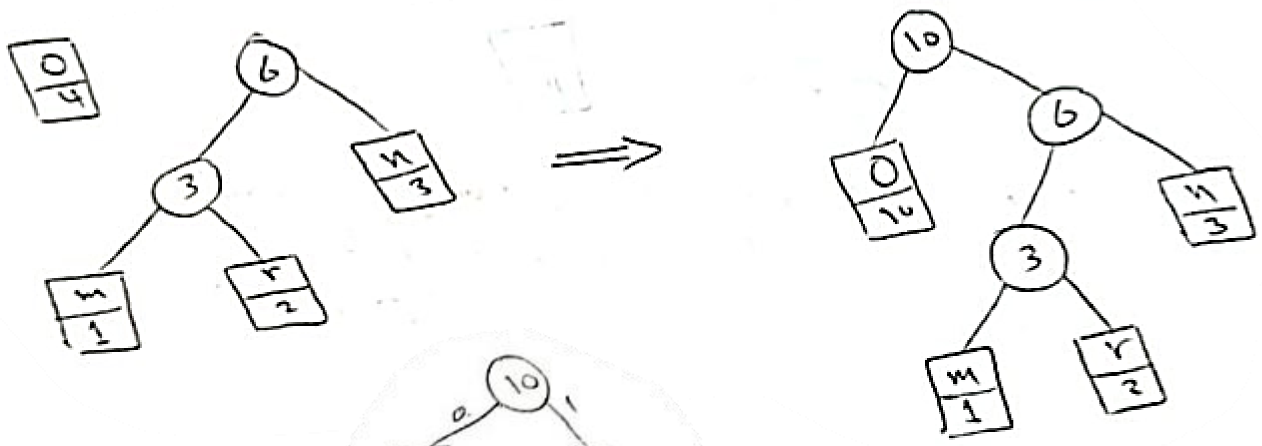


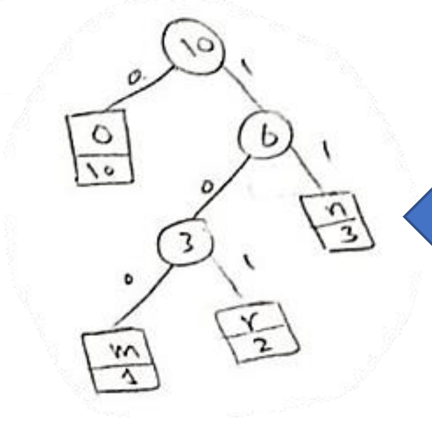
Step(2): take the first two minimum frequencies and create a parent for them contain their summation

Step(3): sort the new sequence



Step(4): repeat step(2) and step(3) until construct a tree from the sequence





This is the final result and I will name the left edge with "0" and right with "1"

Char	Frequency	Code	Number of bits
m	1	100	3
r	2	101	3

n	3	11	2
o	4	0	1

Total number of bits = 1*3 + 2*3 + 3*2 + 4*1 = **19 bits**

Second method

We use fixed size method by assuming the size of each char equal to 8 bits, so the total number of bits = 8 * 10 = **80 bits**

We can see that Huffman methode save 80 – 19 = **61 bits**