*Solution for Assignment 2:*

COMP-352

by

Shadi Jiha (40131284)

Concordia University

Department of Computer Science and Software engineering

04 June 2021

**Question 1:**

* 1. The big-O of this algorithm is because the major part of the algorithm (from line 4 to 12) contains 2 nested loops which, each of them has iteration in worst case.
  2. The big-Omega is because the major part of the algorithm (from 4 line to 12) contains 2 nested where at best the statement is skipped. However, even if the statement if skipped, the double for loops still have to run.

1. Here is the result of the algorithm:

Input: {60, 35, 81, 98, 14, 47}

Output: {14, 35, 47, 60, 81, 98}

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Array A | Array Var | Array S |
| After line 3 | [60, 35, 81, 98, 14, 47] | [0, 0, 0, 0, 0, 0] | [0, 0, 0, 0, 0, 0] |
| After line 12 | [60, 35, 81, 98, 14, 47] | [3, 1, 4, 5, 0, 2] | [0, 0, 0, 0, 0, 0] |
| After line 15 | [60, 35, 81, 98, 14, 47] | [3, 1, 4, 5, 0, 2] | [14, 35, 47, 60, 81, 98] |

1. The algorithm is sorting the array in ascending order. The values in the Var array are the order of elements A in the array S. For example, if the input is array [10, 9, 1, 2, 4, 8], then the array S will be [1, 2, 4, 8, 9, 10] and the Var array is [5, 4, 0, 1, 2, 3].
2. Yes, it possible, we can use the heap-sort algorithm which has a and :
3. The space complexity of this algorithm is or . Because we need 3 arrays each of them of size n. It can be improved by swapping elements directly in the array instead of creating new ones.

**Algorithm** DoSomething(A, n)

**Input:** Array A of size n

**Output**: Sorted Array A

**for** i ← n / 2 - 1 **to** i >← 0 **do**

heapify(A, n, i)

**for** i ← n - 1 **to** i > 0 **do**

temp ← A[0]

A[0] ← A[i]

A[i] ← A[0]

heapify(A, i, 0)

**Algorithm** heapify(array, n, i)

**Input:** array of size n, i is the node index to sort

**Output**: a sorted branch of tree array of node i

max ← i

l ← 2 \* i + 1

r ← 2 \* i + 2

**if** l < n **and** array[l] > array[max] **then**

max ← l

**if** r < n **and** array[r] > array[max] **then**

max ← r

**if** max != i **then**

swap ← array[i]

array[i] ← array[max]

array[max] ← swap

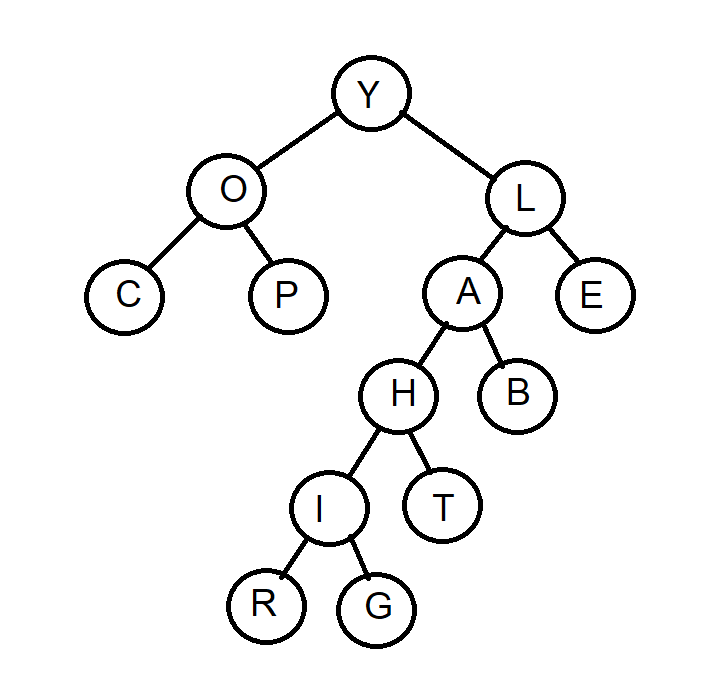
heapify(array, n, max)

**Question 2:**

1. Category 1: List, because the operations for *lookup*, *set* are O(1).
2. Category 2: Positional, because containers need to be eliminated or added in particular position before or after a certain container relative to the start position of the array.
3. Category 3: Sequence because the *add* and *remove* must be done in a sorted array.

**Question 3:**

1. To get both orders we need to construct to following tree:



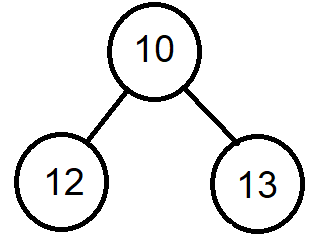
1. The array that will store this binary tree is (All empty spaces are null elements):

|  |  |
| --- | --- |
| Y|O|L|C|P|A|E| | | | |H|B| | | | | | | | | | |I|T| | | | | | | | | | | | | | |
| | | | |R|G| | | | | | | | | | | | | | | | | |  |

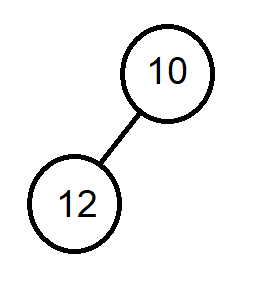
**Question 4:**

1. This is the insertion step by step:

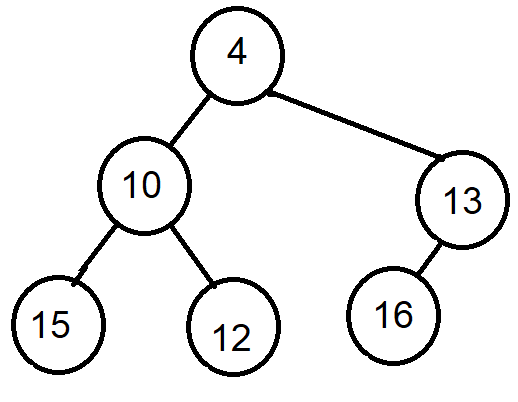
2. Insert 13



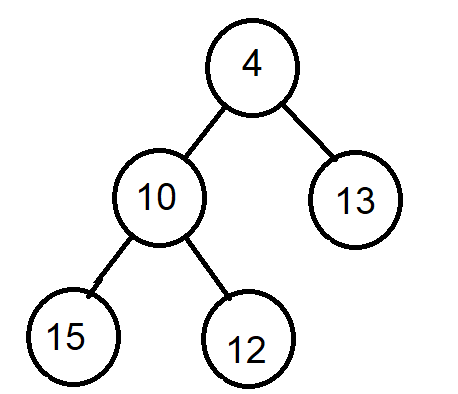
1. Insert 10 then 12:



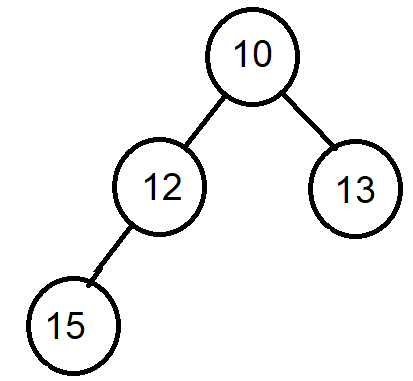
5. Insert 16



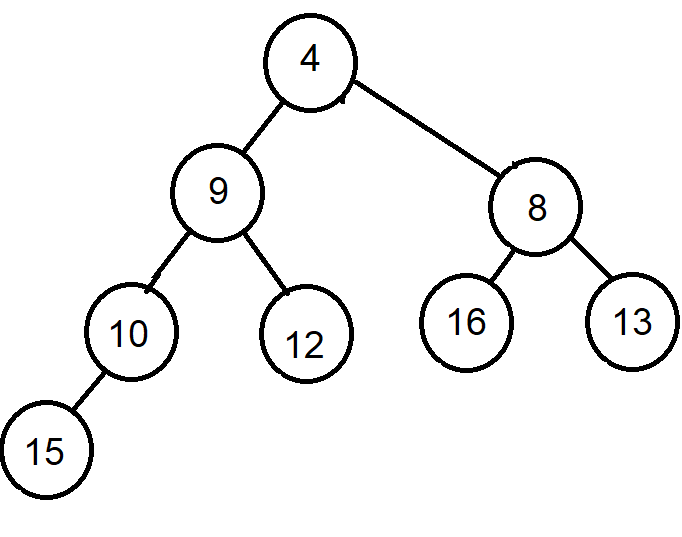
4. Insert 4



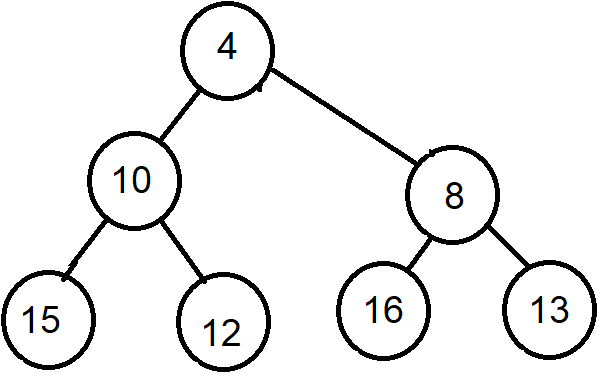
3. Insert 15



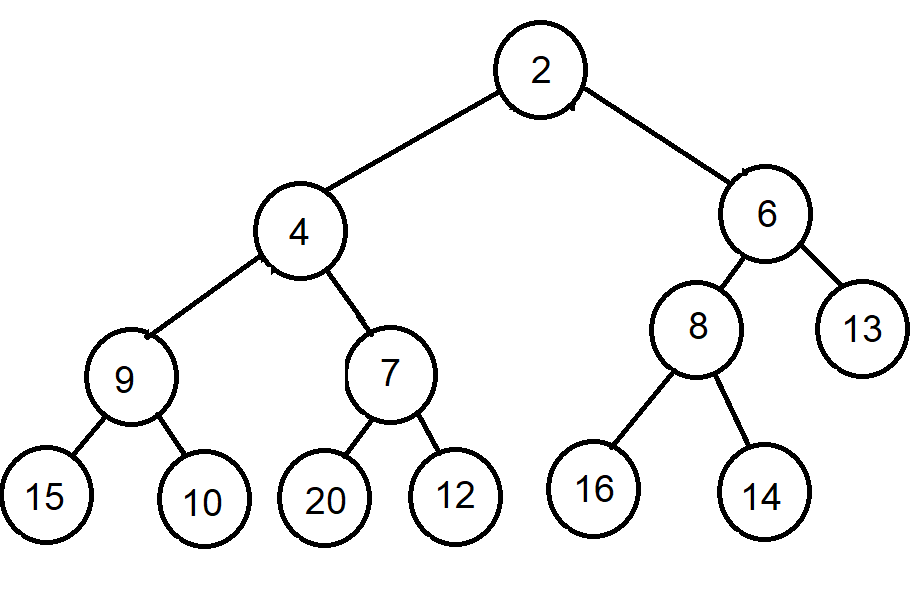
7. Insert 9



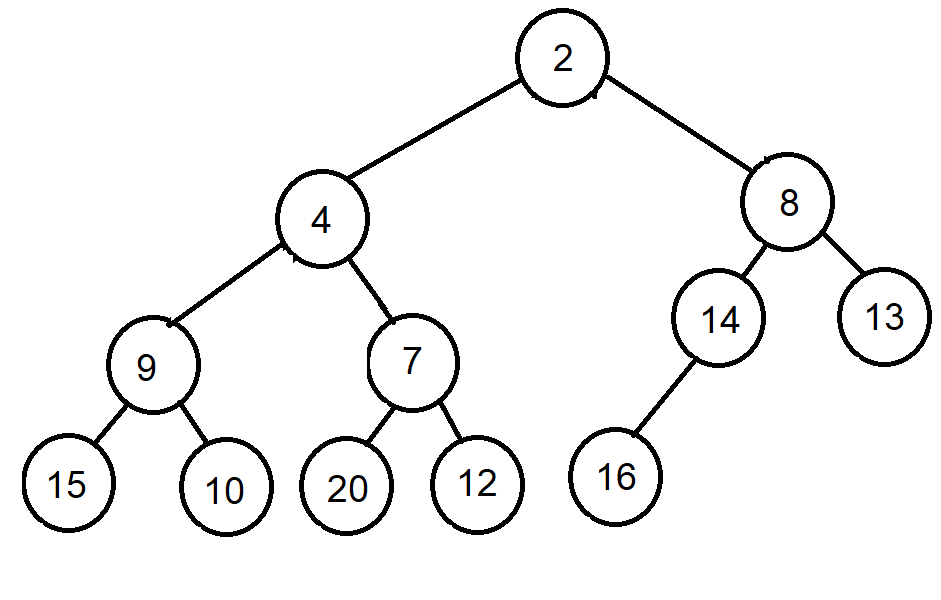
6. Insert 8



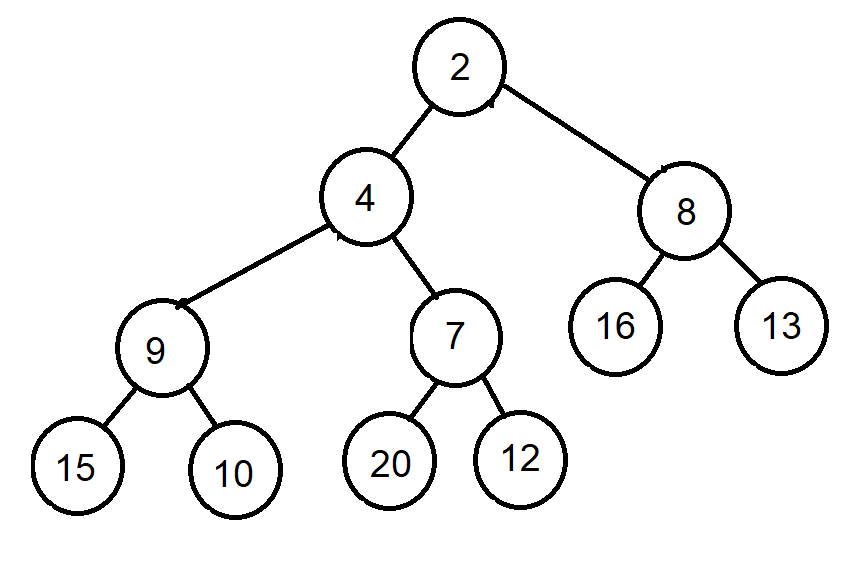
12. Insert 6



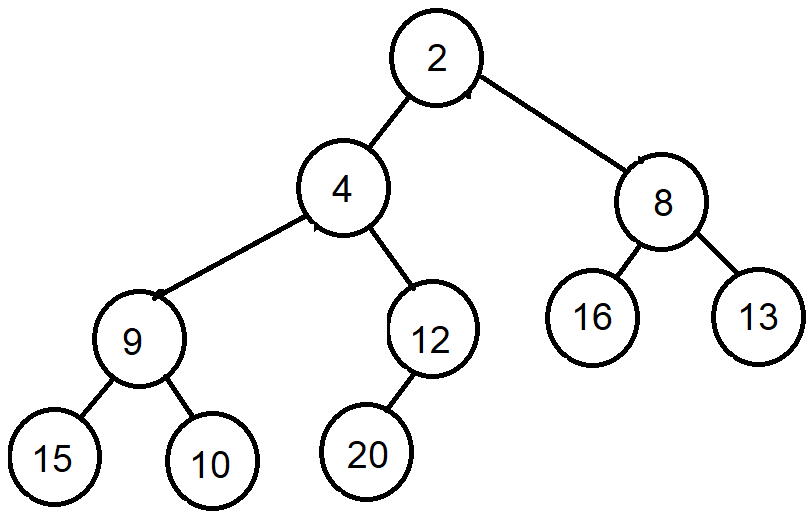
11. Insert 14



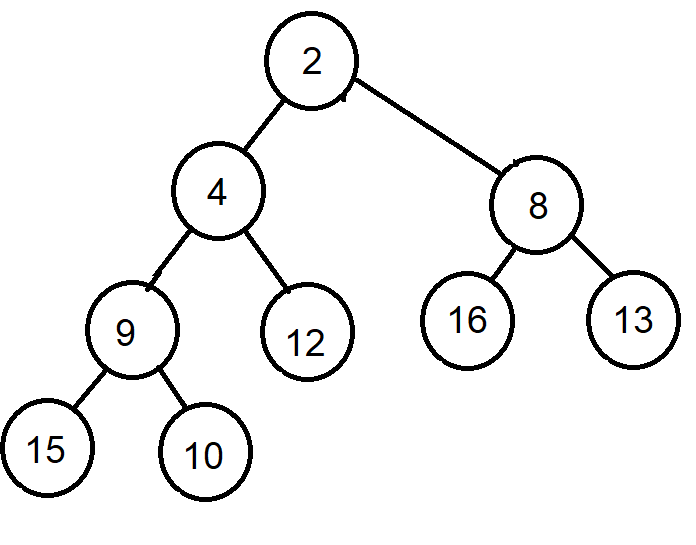
10. Insert 7



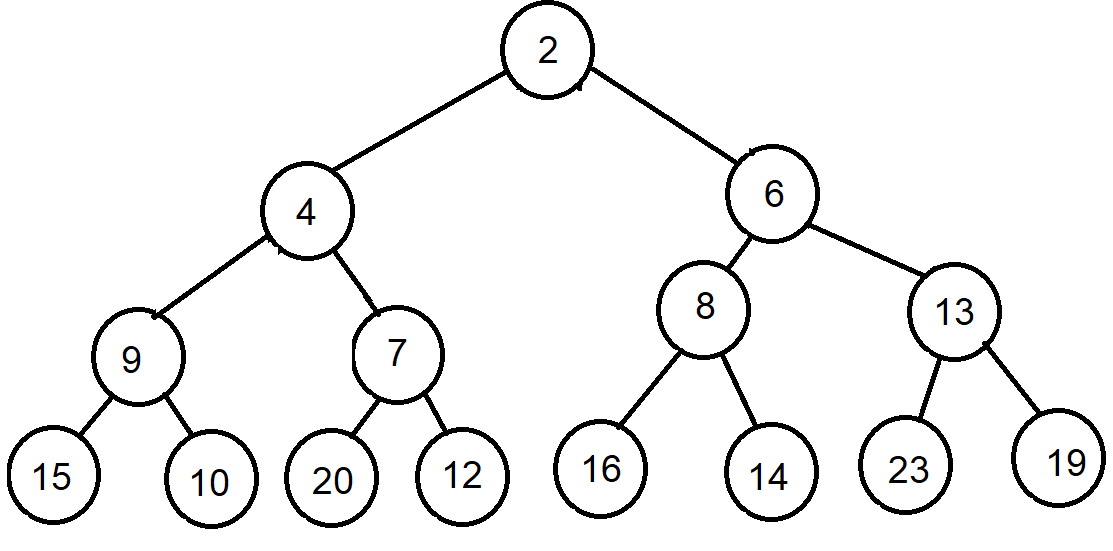
9. Insert 20



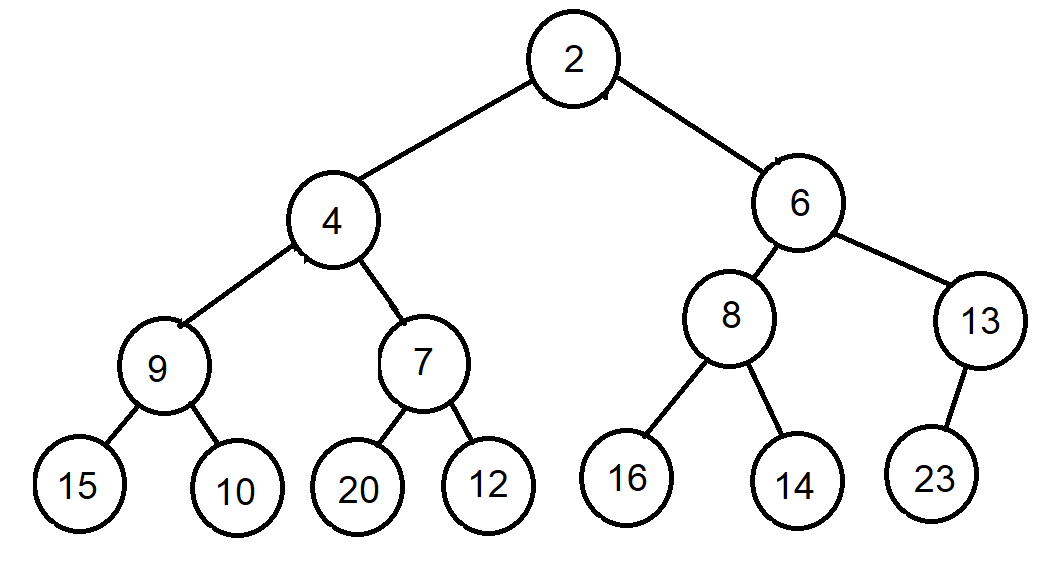
8. Insert 2



14. Insert 19

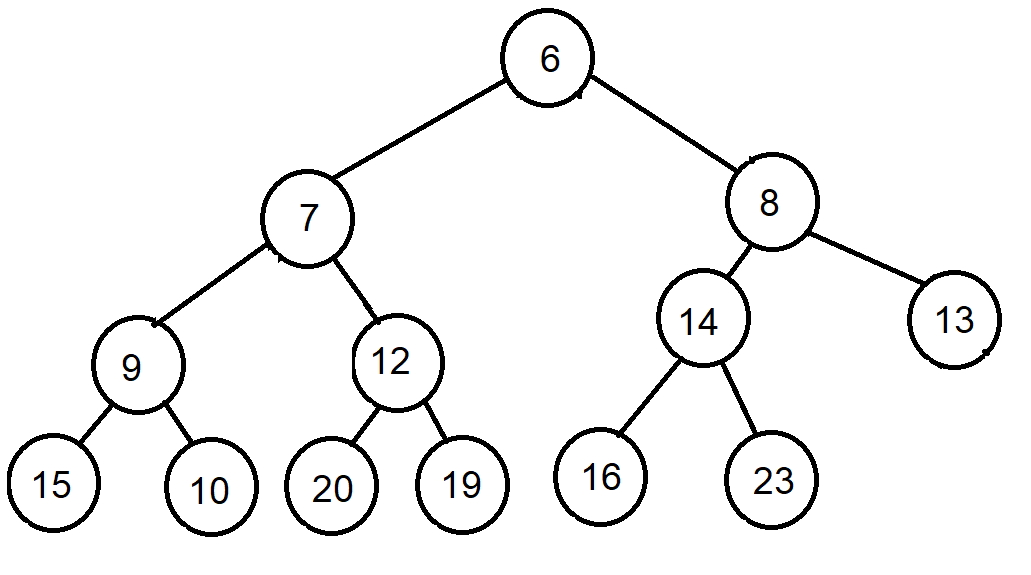


13. Insert 23

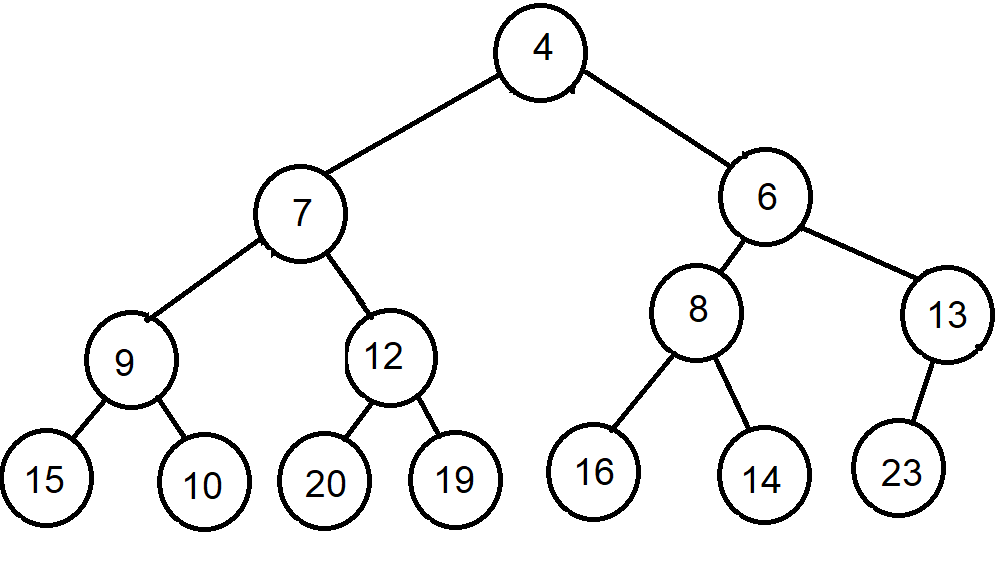


1. Perform removeMin 2 times yields:

Second time and final representation:



First time



**Question 5:**

1. The following algorithm has a complexity of O(n) where n is the number of nodes the tree contains

**Algorithm** depthOfNode(node)

**Input:** The node you want to compute its depth

**Output**: The depth of that node

// Call the helper funtion

**return** depthOfNode(root, node.data, 0)

/\*\*

\* This is a helper function used by

\*/

**Algorithm** depthOfNode(node, data, level)

**Input:** ***node*** the Node to compute its depth*,* ***data***the target data*,* ***level***the current level

**Output**: The depth of the node

**if** node = null **then**

**return** 0

**if** node.data = data **then**

**return** level

down ← depthOfNode(node.left, data, level + 1)

**if** down !← 0 **then**

**return** down

down ← depthOfNode(node.right, data, level + 1)

**return** down

1. The following algorithm has a complexity of O(n) where n is the number of node of the tree. Because it is recursive without any loops inside it.

**Algorithm** count-Full-Nodes(t)

**Input:** The node to calculate if its children are full (default is *root of tree*

**Output**: number of full nodes of tree

**if** t = null **or** t.left = null **or** t.right = null **then**

**return** 0

**else**

**return** 1 + count-Full-Nodes(t.left) + count-Full-Nodes(t.right)