Solution for Assignment 2:

COMP-352

by

Shadi Jiha (40131284)

Concordia University

Department of Computer Science and Software engineering

04 June 2021

Question 1:

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

b) 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]`

c) 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].

d) Yes, it possible, we can use the heap-sort algorithm which has a $O(n \cdot \log(n))$ and $\Omega(n \cdot \log(n))$:

```
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)
```

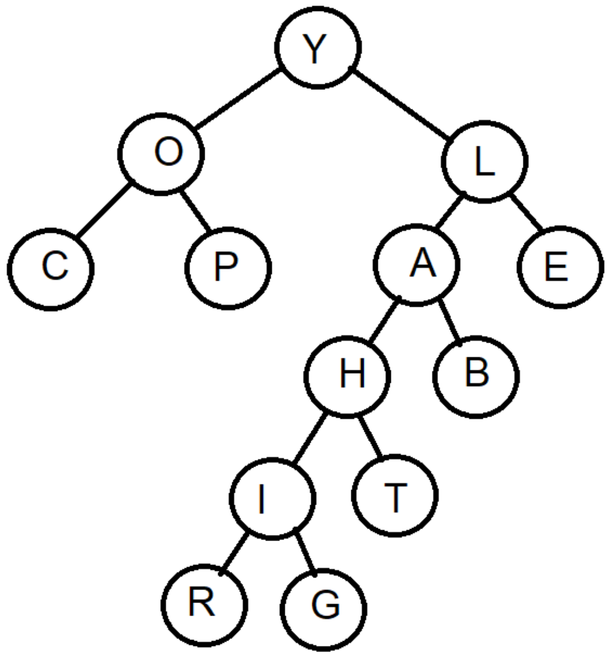
e) The space complexity of this algorithm is $f(n) = 3n$ or $O(n)$. Because we need 3 arrays each of them of size n. It can be improved by swapping elements directly in the array instead of creating new ones.

Question 2:

a) Category 1: List, because the operations for *lookup*, *set* are O(1).
b) 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.
c) Category 3: Sequence because the *add* and *remove* must be done in a sorted array.

Question 3:

a) To get both orders we need to construct to following tree:



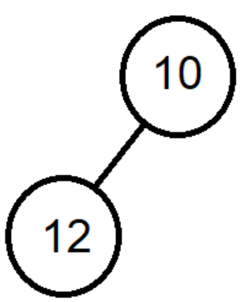
b) The array that will store this binary tree is (All empty spaces are null elements):

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

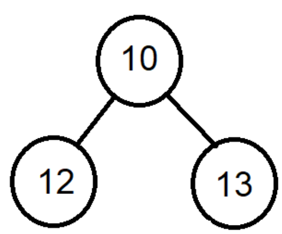
Question 4:

a) This is the insertion step by step:

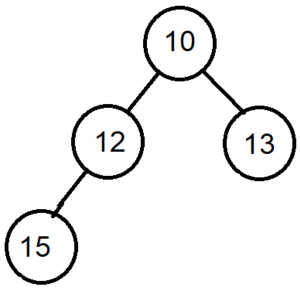
1. Insert 10 then 12:



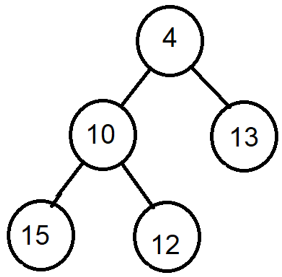
2. Insert 13



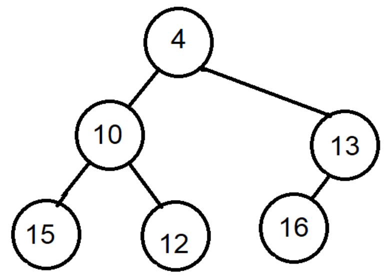
3. Insert 15



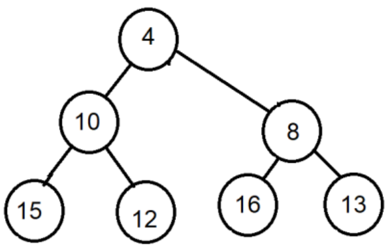
4. Insert 4



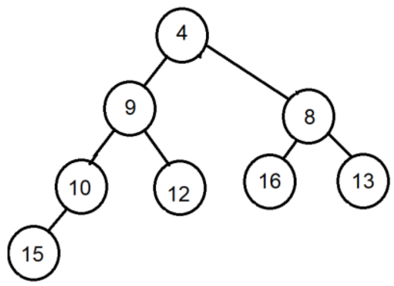
5. Insert 16



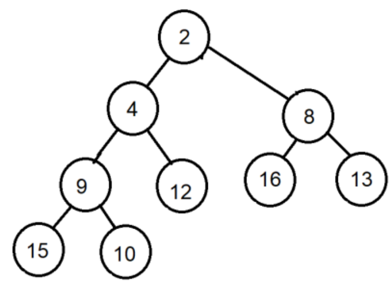
6. Insert 8



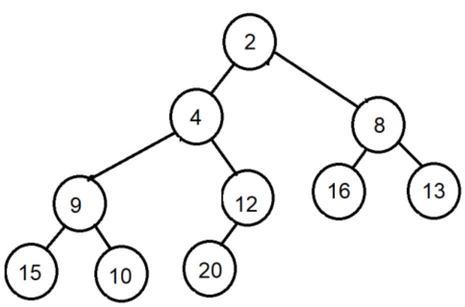
7. Insert 9



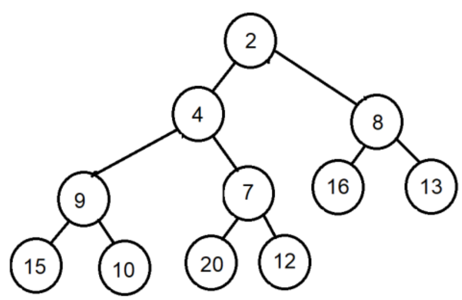
8. Insert 2



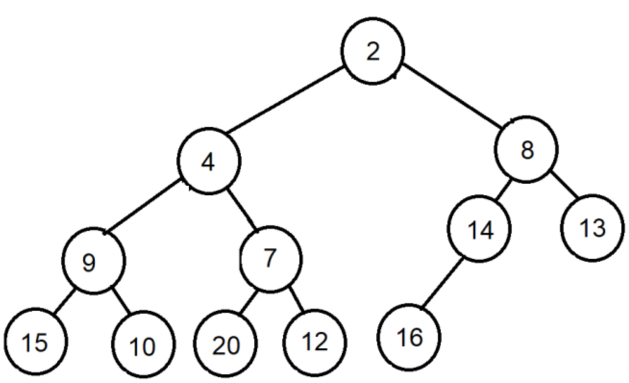
9. Insert 20



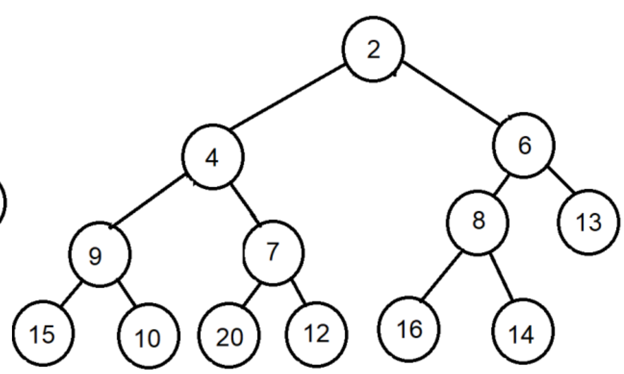
10. Insert 7



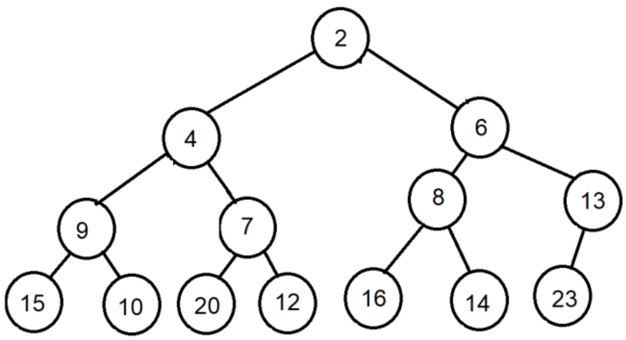
11. Insert 14



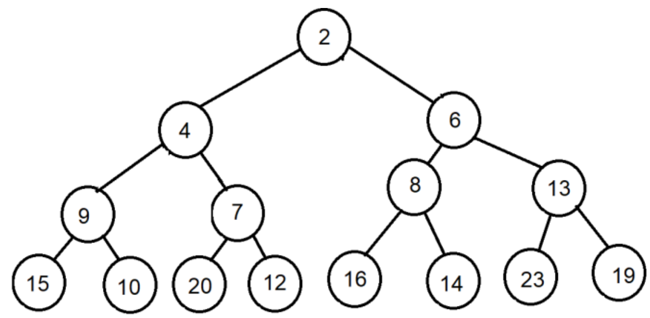
12. Insert 6



13. Insert 23

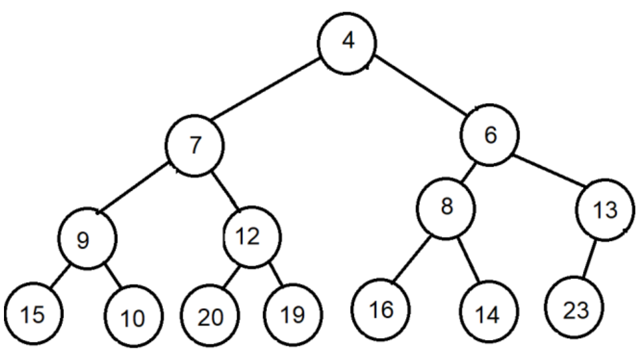


14. Insert 19

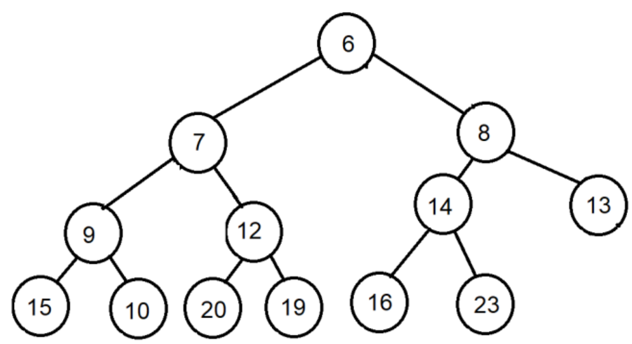


b) Perform removeMin 2 times yields:

First time



Second time and final representation:



<u>Question 5:</u>

a) 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
```

b) 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)
```