5SENG003C Algorithms Theory Design and Implementation

ICT Mock Exam Discussion

Kavya Atapattu
kavya.a@iit.ac.lk

- **Read the instructions completely and carefully. Sometimes the format of the answer is given in the question already and you have to obey that.**

- **You can make your complete answer using the correct piece of theory knowledge.**

- **Your steps should be in logical order and fully describe the way of arriving at the final answer.**

- **Step-by-step marking is going to be done.**

Question 1

Suppose you have the following runtime data for an algorithm. What complexity class do they indicate?

Input size	Seconds
1000	7
2000	15
4000	31
8000	63
16000	125

- **Concept Covered:**

  Under the Empirical Approach to the complexity of an algorithm

Theory Recap: Basic rules in Empirical Method

1. **Repeatedly doubling the input size will always increase the runtime approximately by the same amount .**
    - The complexity class of this algorithm: Logarithmic
    - Big-O notation: O(Log n)
2. **Repeatedly doubling the input size will always multiply the runtime approximately by 2.**
    - The complexity class of this algorithm: Linear
    - 2^1 = 2 or else we can say log_2(2)= 1
    - Big-O notation: O(n^1)
3. **Repeatedly doubling the input size will always multiply the runtime approximately by 4.**
    - The complexity class of this algorithm: Quadratic
    - 2^2 = 4 or else we can say log_2(4)= 2
    - Big-O notation: O(n^2)
4. **Repeatedly doubling the input size will always multiply the runtime approximately by 8.**
    - The complexity class of this algorithm: Cubic
    - 2^3 = 8 or else we can say log_2(8)= 3
    - Big-O notation: O(n^3)

5. **Repeatedly increasing the input size by a fixed amount will always multiply the runtime approximately by some fixed amount**
    - The complexity class of this algorithm: Exponential
    - Big-O notation: O(2^n)

Question 2

Consider the following code fragment. Based on its structure, what is its complexity class?

```
int s = 0;
for(int i = 0; i < n; i++)
    for(int j = 0; j < 6; j++)
        m += j;
```

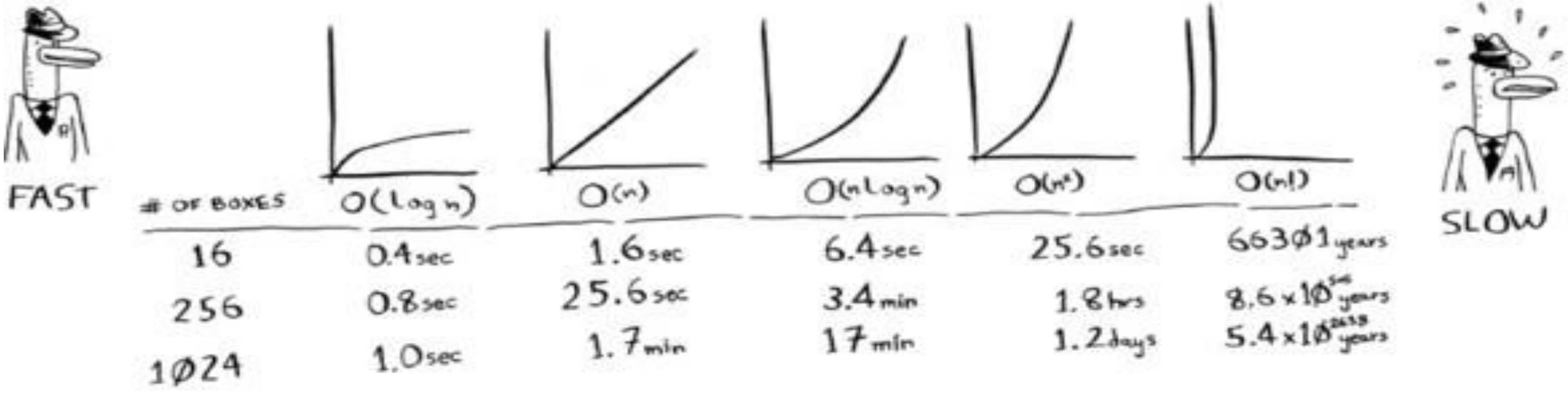
- **Concept Covered:**

  Under Big-O notation and some important complexity classes

Should recap implementation examples of the following:

// To do

1. Constant Time Complexity – O(1)
2. Linear Time Complexity – O(n^1)
3. Quadratic Time Complexity - O(n^2)
4. Cubic Time Complexity - O(n^3)
5. Quasilinear Time Complexity – O(n log n)



Question 2 – Sample Answer

- The outer loop runs n times.
- The inner loop runs a constant number of times (6 times) regardless of the value of n.
- Therefore, the time complexity of the algorithm is O(n * 6), which simplifies to O(6n).
- In big O notation, we drop the constant coefficient, so the final time complexity is O(n).
- So, the complexity class of this algorithm is linear, and its big O notation is O(n).

It's your responsibility to create a comprehensive answer deserving a perfect score of 10/10.

Question 3

Suppose we use Binary Search to find the value 3 on the following array. Which array elements get checked, in which order, and why?

index	0	1	2	3	4	5	6
value	1	2	3	5	8	13	21

- **Concept Covered:**

  Under Binary Search

Theory Recap: Binary Search

1st Check

```
mid = (first + last) / 2
    = (0+6)/2 = 3
```

1. We are searching for 3 in this array

Index	0	1	2	3	4	5	6
Value	1	2	3	5	8	13	21
	first			mid			last

2nd Check

```
mid = (first + last) / 2
    = (0+2)/2 = 1
```

2. We are searching for 3 but 5>3, So focus on Left half

Index	0	1	2	3	4	5	6
Value	1	2	3	5	8	13	21
	first	mid	last				

3rd Check

```
mid = (first + last) / 2
    = (2+2)/2 = 2
```

3. We are searching for 3 but 2<3, So focus on Right half

Index	0	1	2	3	4	5	6
Value	1	2	3	5	8	13	21
			mid first last				

Question 3 – Sample Answer

- The elements that get checked are a[3], a[1], a[2].

First Check 1

1.start = 0, end = 6, mid = (0 + 6) / 2 = 3
2.Compare the value at index 3 (5) with the target value (3).
    1. Since 5 > 3, move the end index to mid - 1.

Second Check 2

1.start = 0, end = 2, mid = (0 + 2) / 2 = 1
2.Compare the value at index 1 (2) with the target value (3).
    1. Since 2 < 3, move the start index to mid + 1.

Third Check

1.start = 2, end = 2, mid = (2 + 2) / 2 = 2
2.Compare the value at index 2 (3) with the target value (3).
    1. We found the target value.

- So, the array elements checked and the order is as follows:

    ✓ Check index 3 (value = 5)
    ✓ Check index 1 (value = 2)
    ✓ Check index 2 (value = 3)

It's your responsibility to create a comprehensive answer deserving a perfect score of 10/10.

Question 4

Suppose we run Bubble Sort to sort the following array in increasing order. What does the array look like after the first 3 iterations, and why?

index	0	1	2	3	4	5	6
value	13	2	21	3	1	8	5

- **Concept Covered:**

  Under Sorting Algorithms → Bubble Sort

Bubble Sort Demonstration

- Compare the adjacent elements --> a[i] and a[i+1]
- If those elements are not sorted --> swap

n = 7

13	2	21	3	1	8	5
0	1	2	3	4	5	6

We need to sort this array in ascending order (Increasing order)

Bubble Sort Demo (cont.)

Round = 1
i = 0

We need to sort this array in ascending order

Correct order (Do not swap)	
Smaller value	**Larger value**

Incorrect order (Swap !!!)	
Larger value	**Smaller value**

Number of comparisons need
i=0 --> (n-1-i) --> 7-1-0 = 6

Array at the end of 1st iteration

Unsorted | Sorted

2	13	3	1	8	5	21
0	1	2	3	4	5	6

Compare, Swap !!!

13	2	21	3	1	8	5
0	1	2	3	4	5	6

Compare, Do not Swap

2	13	21	3	1	8	5
0	1	2	3	4	5	6

Compare, Swap !!!

2	13	21	3	1	8	5
0	1	2	3	4	5	6

Compare, Swap !!!

2	13	3	21	1	8	5
0	1	2	3	4	5	6

Compare, Swap !!!

2	13	3	1	21	8	5
0	1	2	3	4	5	6

Compare, Swap !!!

2	13	3	1	8	21	5
0	1	2	3	4	5	6

Bubble Sort Demo (cont.)

Round = 2
i = 1

We need to sort this array in ascending order

Correct order (Do not swap)	
Smaller value	**Larger value**
Incorrect order (Swap !!!)	
Larger value	**Smaller value**

Number of comparisons need
i=0 --> (n-1-i) --> 7-1-1 = 5

Unsorted | Sorted

2	3	1	8	5	13	21
0	1	2	3	4	5	6

Array at the end of 2nd iteration

Compare, Do not Swap

2	13	3	1	8	5	21
0	1	2	3	4	5	6

Compare, Swap !!!

2	3	13	1	8	5	21
0	1	2	3	4	5	6

Compare, Swap !!!

2	3	13	1	8	5	21
0	1	2	3	4	5	6

Compare, Swap !!!

2	3	1	13	8	5	21
0	1	2	3	4	5	6

Compare, Swap !!!

2	3	1	8	13	5	21
0	1	2	3	4	5	6

Bubble Sort Demo (cont.)

Round = 3
i = 2

We need to sort this array in ascending order

Correct order (Do not swap)	
Smaller value	**Larger value**

Incorrect order (Swap !!!)	
Larger value	**Smaller value**

Number of comparisons need
i=0 --> (n-1-i) --> 7-1-2 = 4

Unsorted | Sorted

2	1	3	5	8	13	21
0	1	2	3	4	5	6

Array at the end of 3rd iteration

Compare, Do not Swap

2	3	1	8	5	13	21
0	1	2	3	4	5	6

Compare, Swap !!!

2	3	1	8	5	13	21
0	1	2	3	4	5	6

Compare, Do not Swap

2	1	3	8	5	13	21
0	1	2	3	4	5	6

Compare, Swap !!!

2	1	3	8	5	13	21
0	1	2	3	4	5	6

Question 4 – Sample Answer

Now that you understand how bubble sort works, try to compose a complete answer.

It's your responsibility to create a comprehensive answer deserving a perfect score of 10/10.

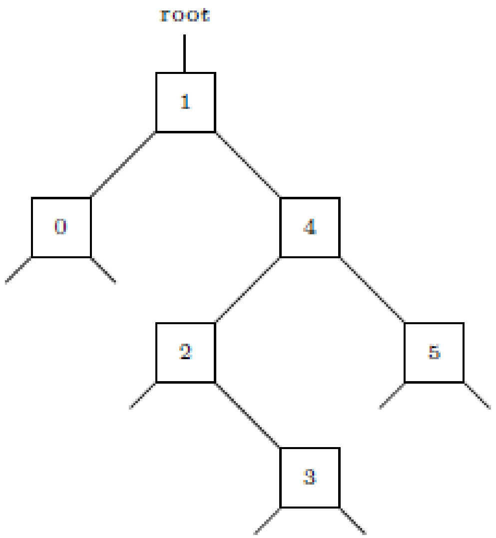
Should recap these sorting algorithms as well:

// To do

1. Selection Sort → Brute Force
2. Merger Sort → Divide and Conquer

Question 5

Suppose we remove the value 4 from the following binary search tree. What does the resulting tree look like?



To enter your answer, write the contents of the tree one level at a time, using "–" to indicate missing nodes. For example, the above tree would be written

1			
0	4		
–	–	2	5
–	3	–	–

- **Concept Covered:**

  Under Binary Search Tree→ Deletion

Theory Recap: Binary Search Tree – Node Deletion

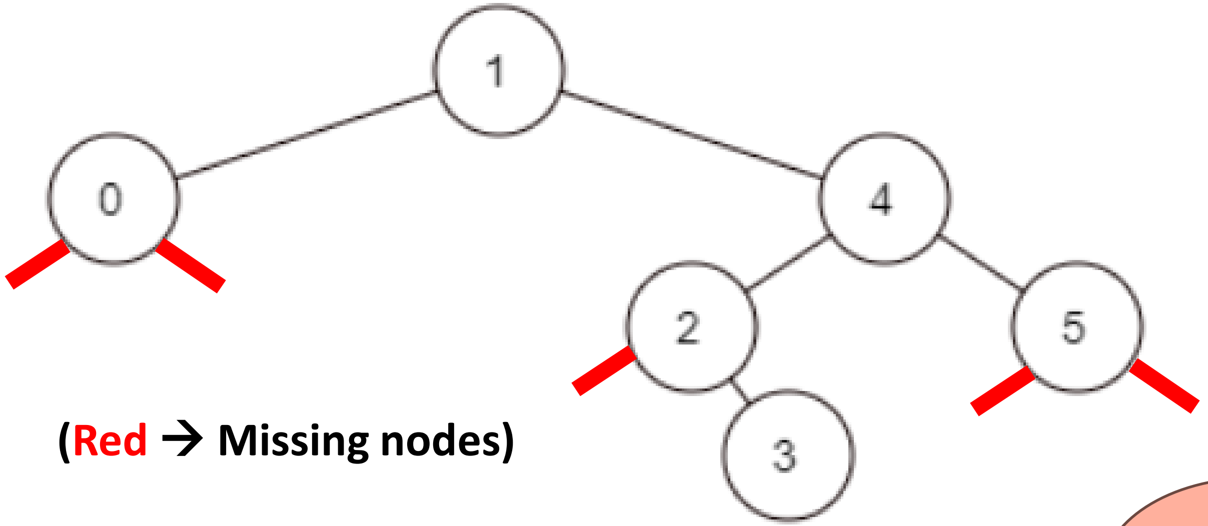
There are 3 cases that can happen when you are trying to delete a node. If it has,

- **No subtree (no children):** This one is the easiest one. You can simply just delete the node, without any additional actions required.

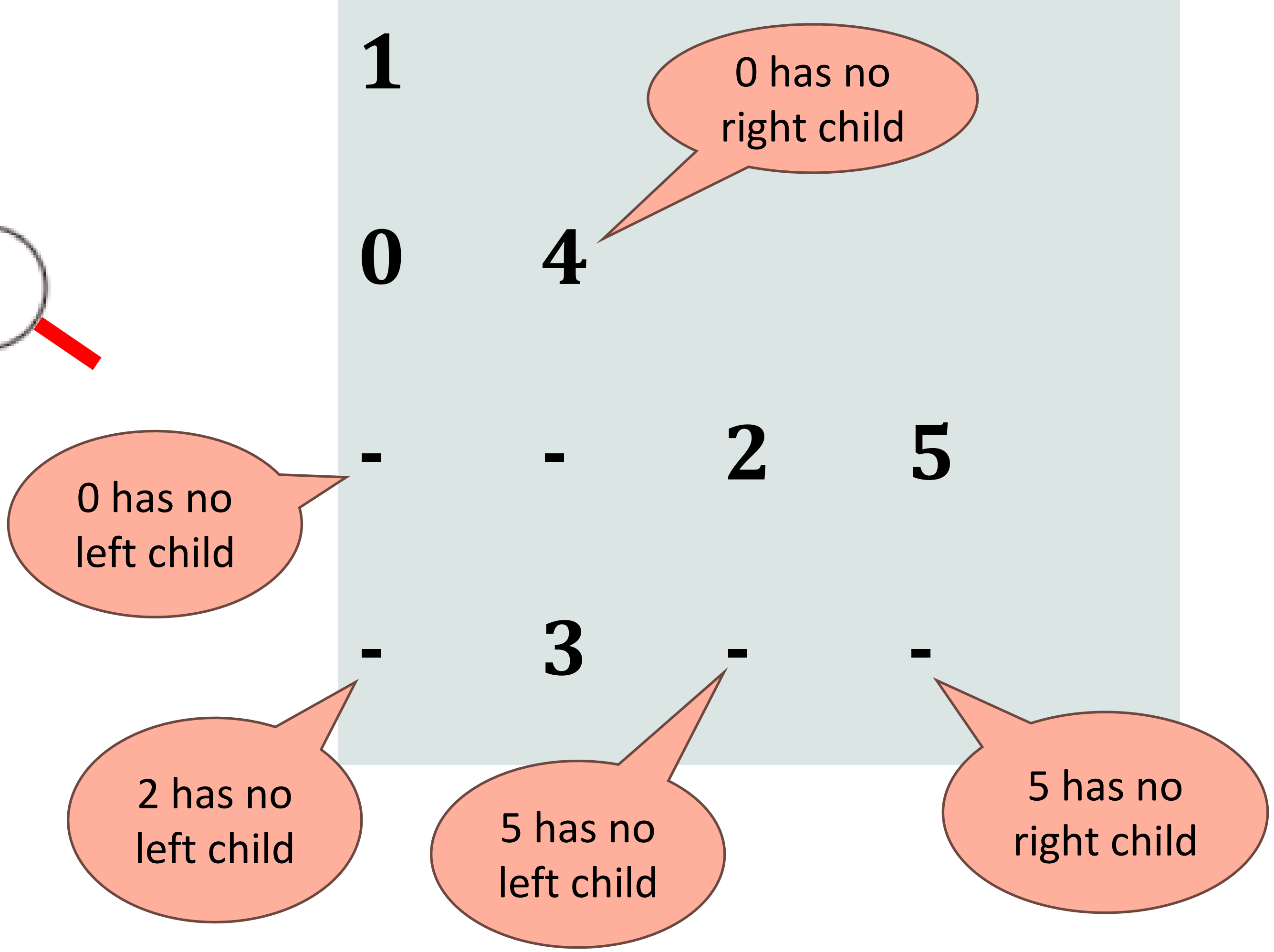
- **One subtree (one child):** You have to make sure that after the node is deleted, its child is then connected to the deleted node's parent.

- **Two subtrees (two children):** You have to find and replace the node you want to delete with its successor
  - The letfmost node (minimum) in the right subtree **OR**
  - The rightmost node (maximum) in the left subtree **(Lecture)**

Understanding the Answer Format



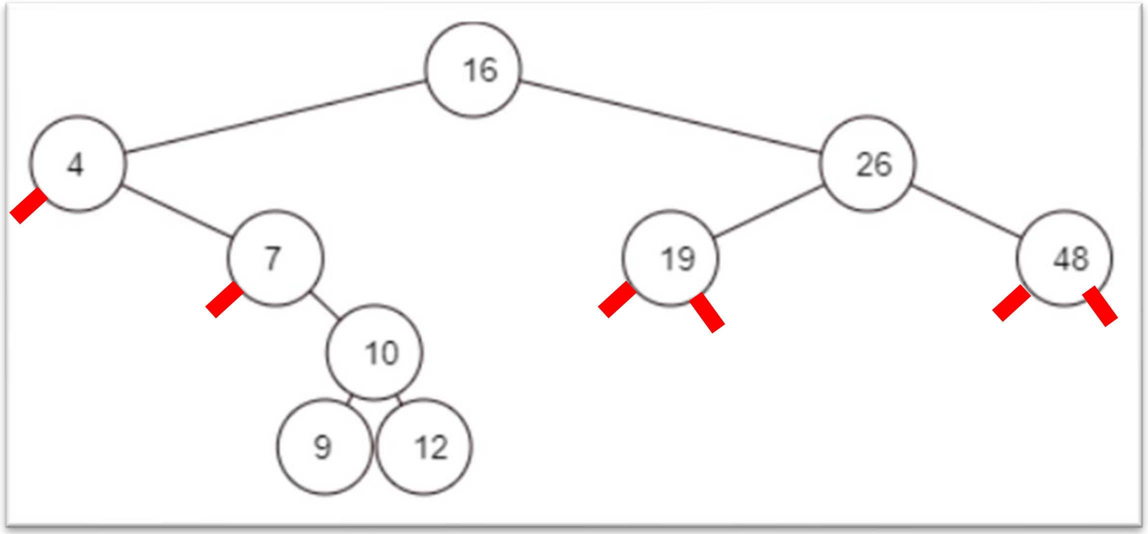
(Red → Missing nodes)



- The contents of the tree should be written one level at a time

- If there is a parent in the upper level, **and** that parent has a missing child/children in one level below, then only we denote that missing child/children with -

Write the content of the tree using the format given: A Practise Exercise

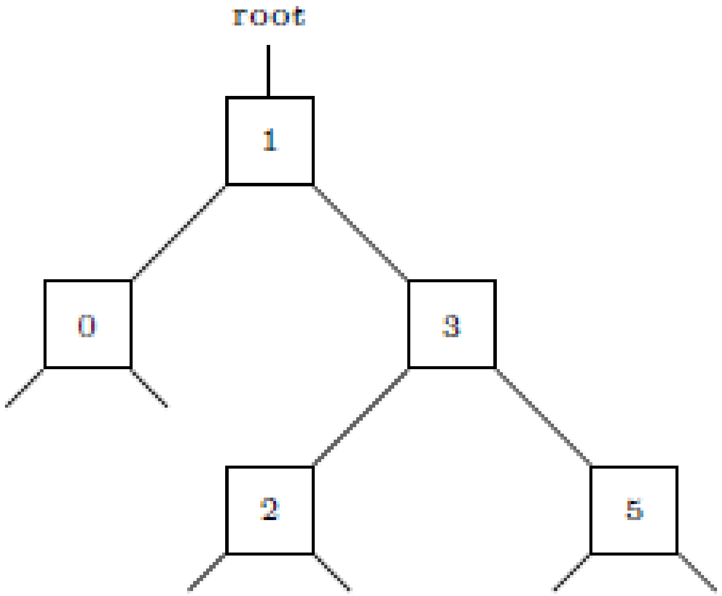


See the missing nodes (indicate by red)
Those are the nodes we should represent by -

16					
4	**26**				
-	**7**	**19**	**48**		
-	**10**	**-**	**-**	**-**	**-**
9	**12**				

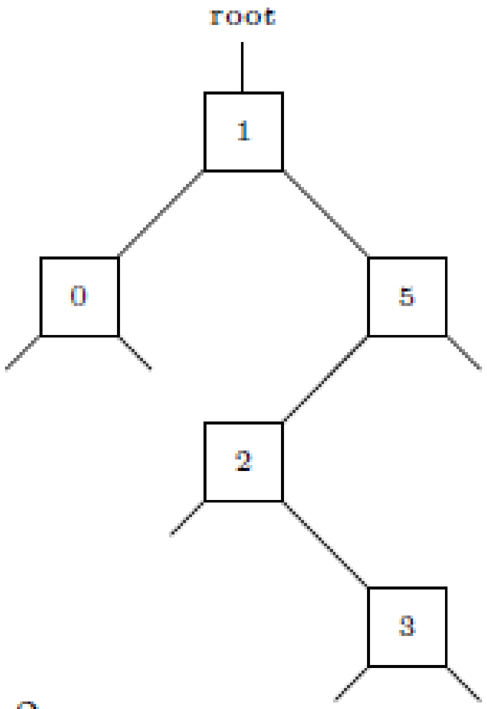
Question 5 – Sample Answer

Option 1:
4 has been replaced by the rightmost node (maximum) in the left subtree.



1
0 3
– – 2 5

Option 2:
4 has been replaced by the left most node (minimum) in the right subtree.



1
0 5
– – 2 –
– 3

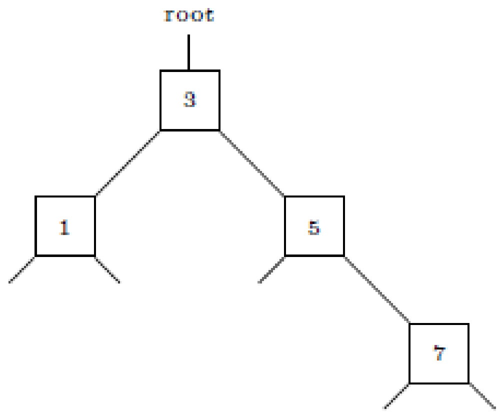
Should recap :

// To do

1. Binary Search Node Deletion
2. Binary Search Node Insertion

Question 6

Suppose we insert the value 6 in the following AVL tree, and re-balance it. What does the resulting tree look like?

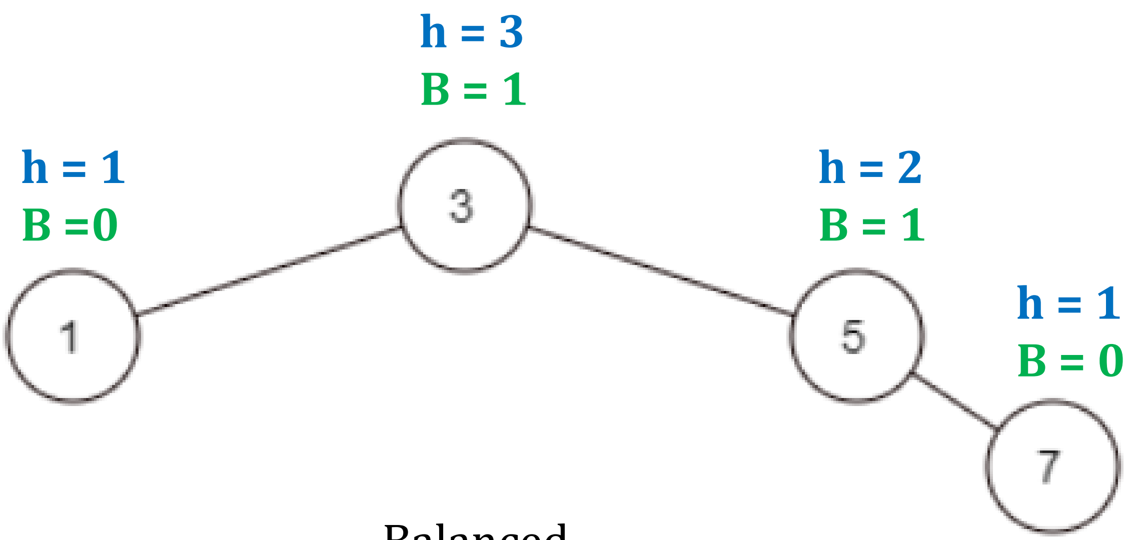


To enter your answer, write the contents of the tree one level at a time, using "–" to indicate missing nodes. For example, the above tree would be written

3
1 5
– – – 7

- **Concept Covered:**

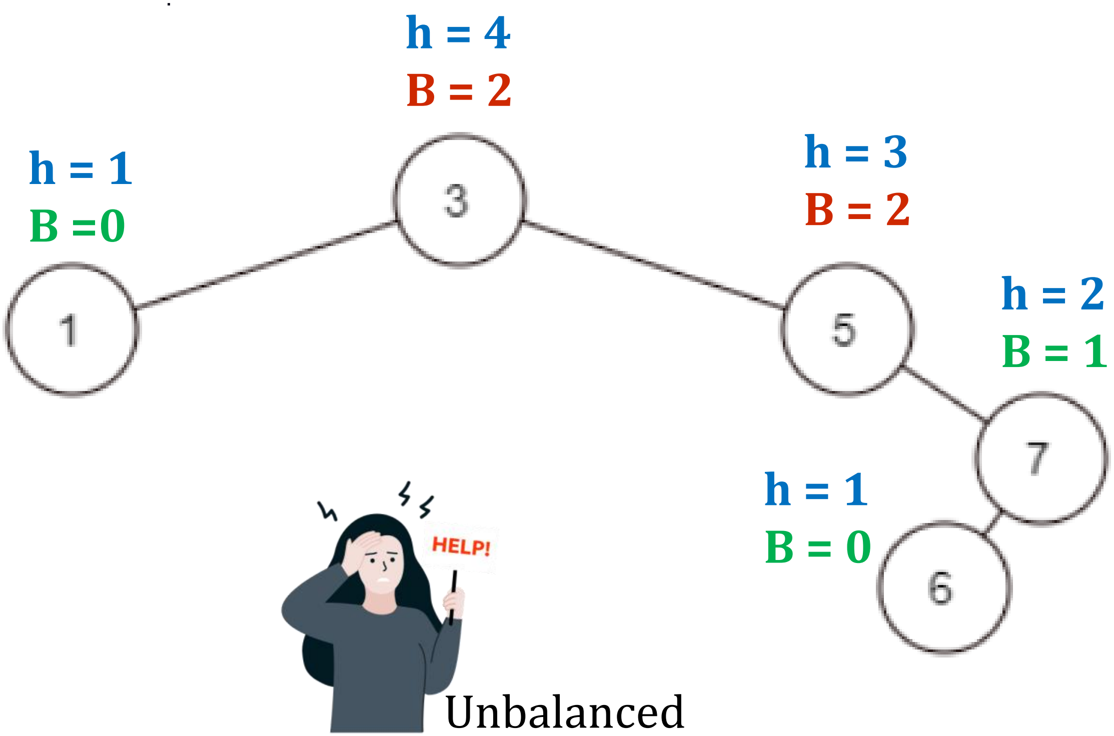
  Under Balanced Binary Search Tree→ AVL Trees → Inserting



Balanced

B = L - R

Inserting 6



Unbalanced

AVL Trees -Recap

- An AVL tree is an example of a balanced binary search tree.

Every AVL tree $\longrightarrow$ **BST**

AVL tree $\xleftarrow{\text{NOT}}$ **Every BST**

Definition

An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

So, the balance property for an AVL tree is that:

the left and right sub-trees differ by at most 1 in height.

AVL Trees –Recap (cont.)

AVL Insertion

Insert the new node into the AVL tree as a **leaf node**

- Same way like in BST

Are the AVL tree is balanced ?

- If answer is **NO,** go to next stage
- If the answer is **YES** we are done with the insertion.

Re-balance the AVL tree

- Use rotations
  Left
  Right
  Left-Right
  Right-Left

We need extra information to be stored with every node → **Balance Factor (BF)**

AVL Trees –Recap (cont.)

Balance Factor (BF)

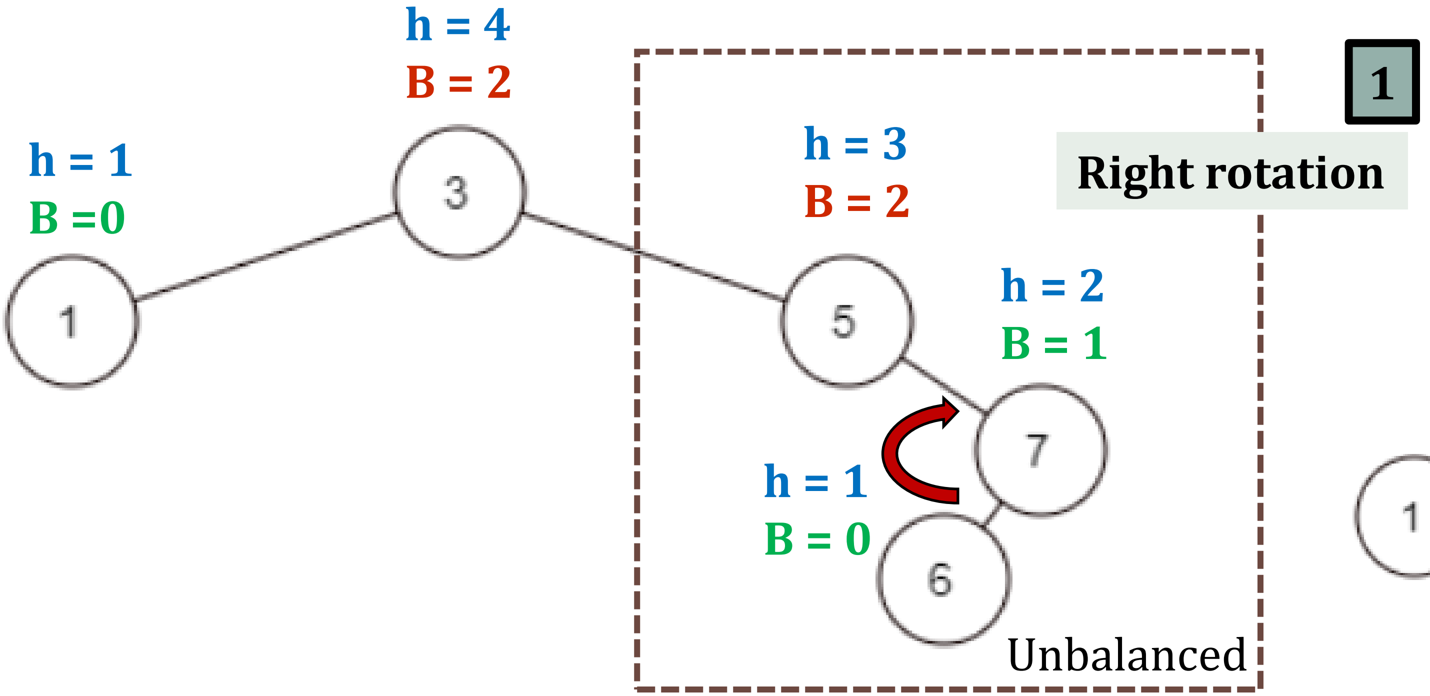
- Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

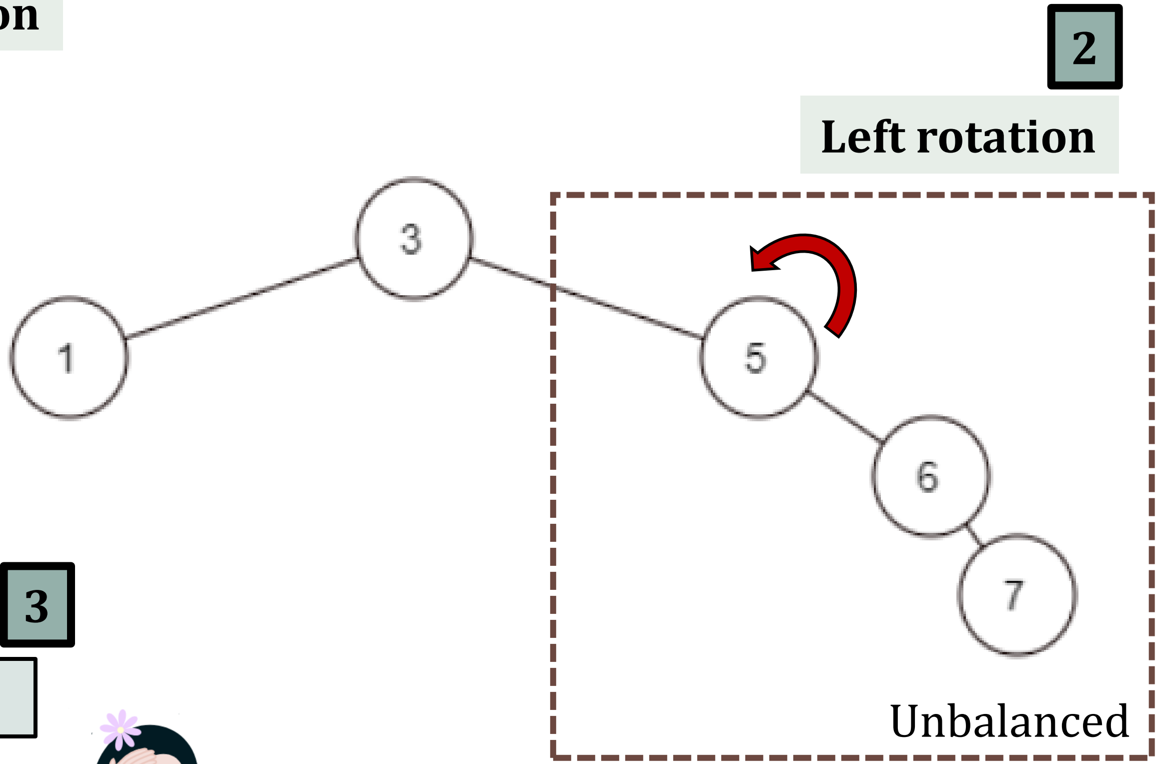
Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

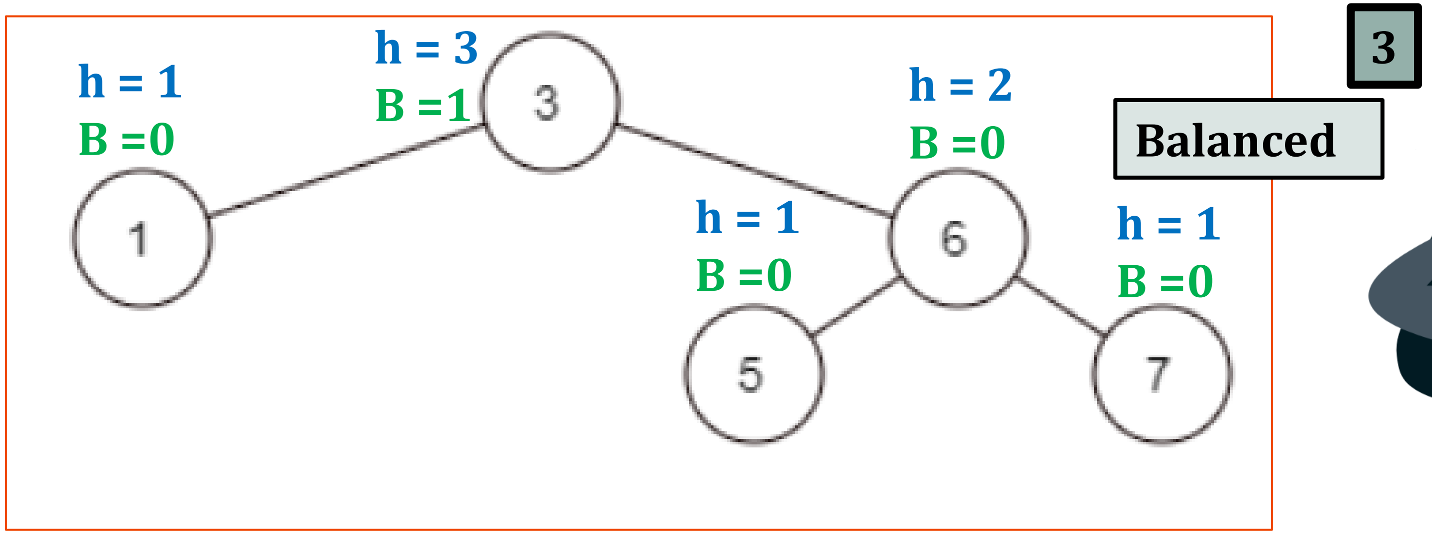
- The self-balancing property of an AVL tree is maintained by the balance factor. The value of the balance factor should always be -1, 0 or +1.

Back to the question

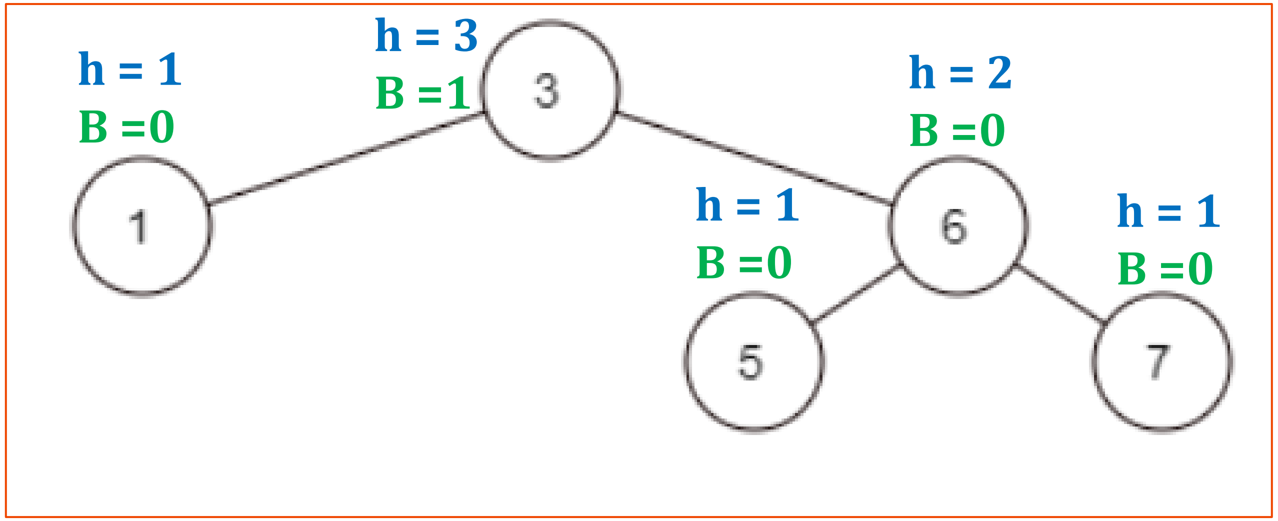
- RL rotation is performed if a node is inserted into the left subtree of the right subtree.







Question 6 – Sample Answer



Balanced

3			
1	**6**		
-	**-**	**5**	**7**

Recap all rotation methods used to rebalance a tree when balance property is violated during node insertion or deletion:

// To do

1. Left Rotation
2. Right Rotation
3. Left-Right Rotation
4. Right-Left Rotation

Question 7

Consider the following array representation of a Min-Heap. How does this change after inserting the value 0?

index	0	1	2	3	4
value	1	2	4	5	3

- **Concept Covered:**

  Heaps→ Min-Heaps→ Inserting

Heaps - Recap

- A heap is a data structure which uses a binary tree for its implementation. (Not Binary **Search** Tree !!!)
- It is the base of the algorithm heapsort and also used to implement a priority queue.
- It is basically a complete binary tree and generally implemented using an array.
- The root of the tree is the first element of the array.

Heaps - Recap (cont.)

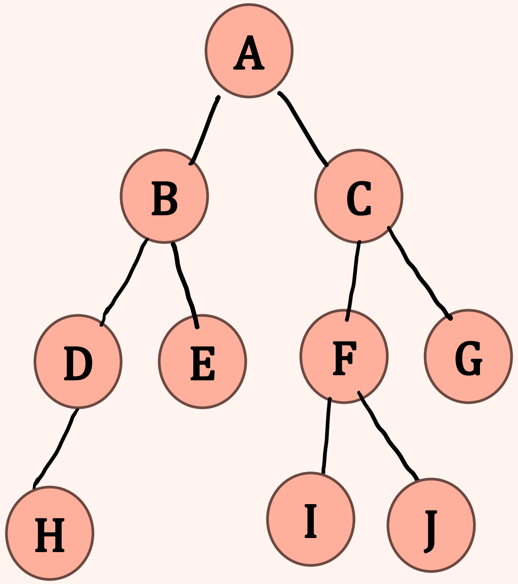
Full Binary Tree and Complete Binary Tree

Full Binary Tree	Complete Binary Tree
A full binary tree is a binary tree in which all of the nodes have either 0 or 2 offspring. In other terms, a full binary tree is a binary tree in which all nodes, except the leaf nodes, have two offspring	Tree in which every level, except possibly the last, is completely filled, and all nodes of bottom level should be filled from left to right.

Heaps – Recap (cont.)

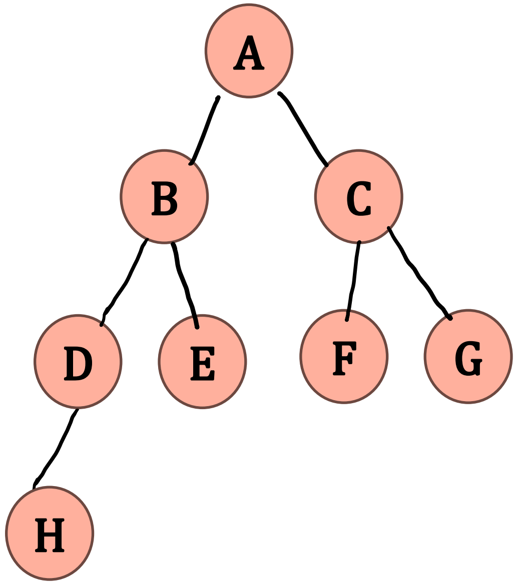
Complete Full Binary Tree and Complete Binary Tree

Tree A



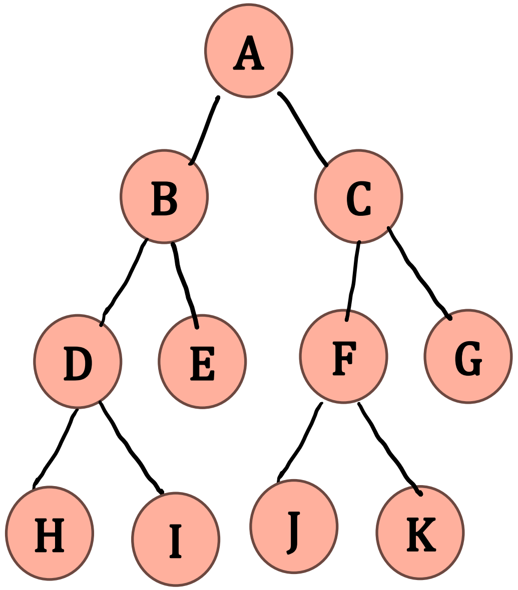
Full: NO
Complete: NO

Tree B



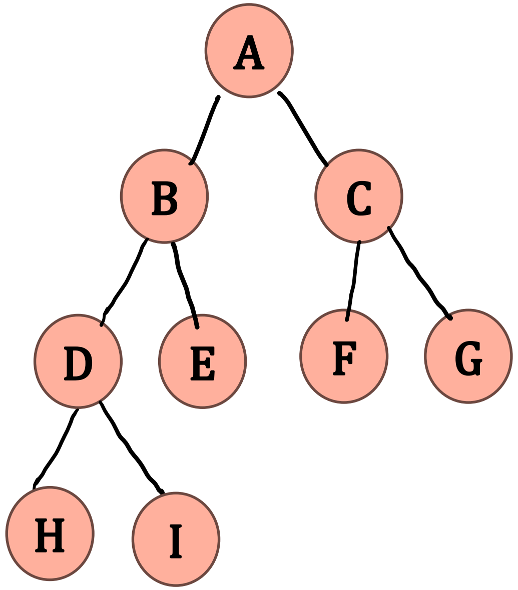
Full: NO
Complete: YES

Tree C



Full: YES
Complete: NO

Tree D



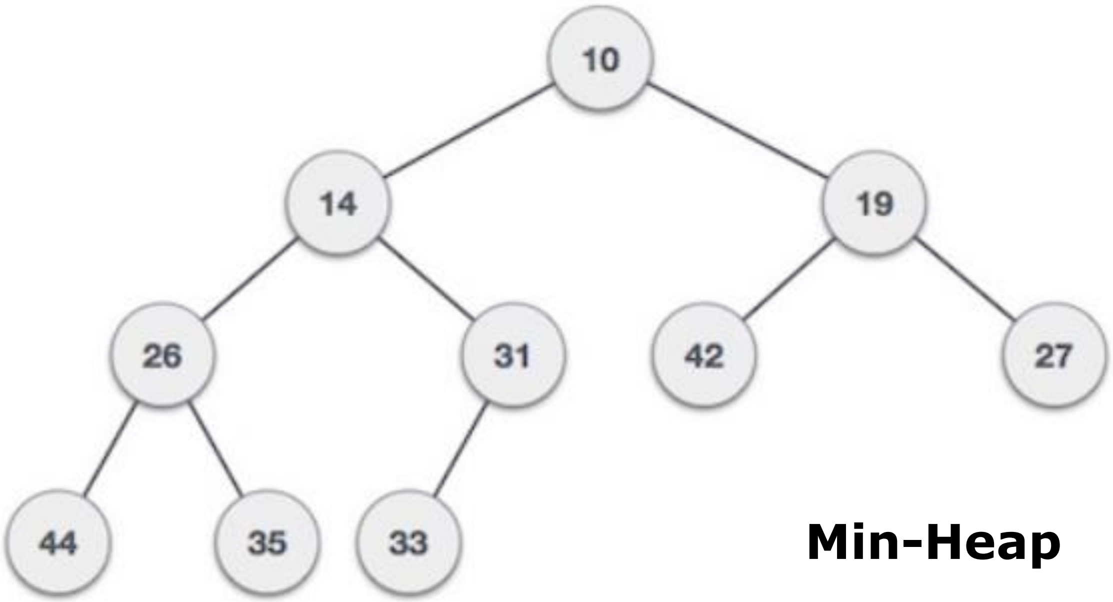
Full: YES
Complete: YES

Heaps – Recap (cont.)

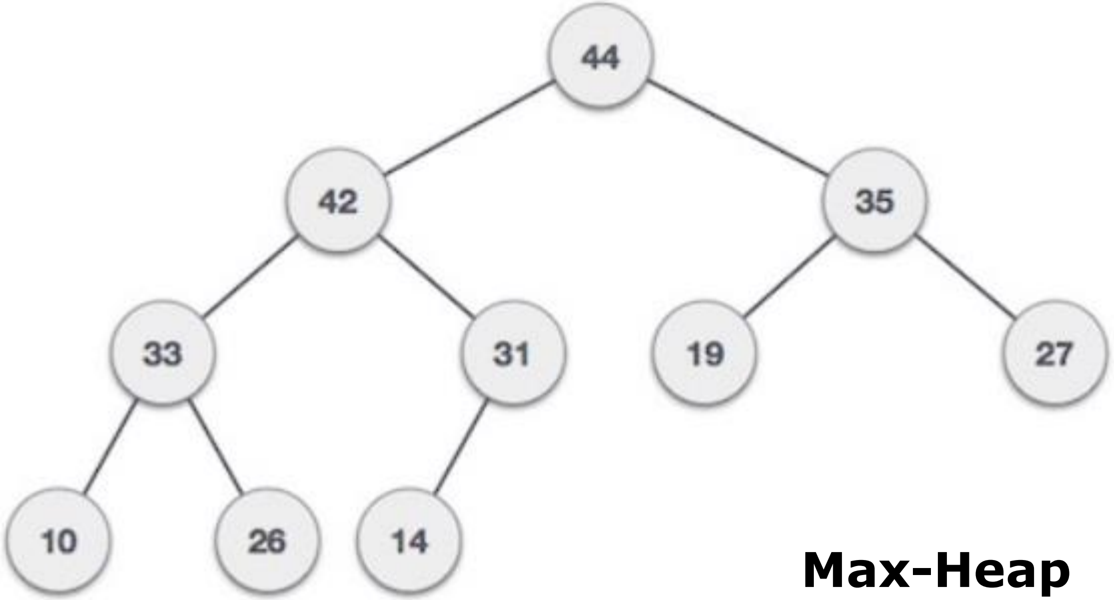
Min Heap and Max Heap

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called **max heap property**.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called **min heap property.**



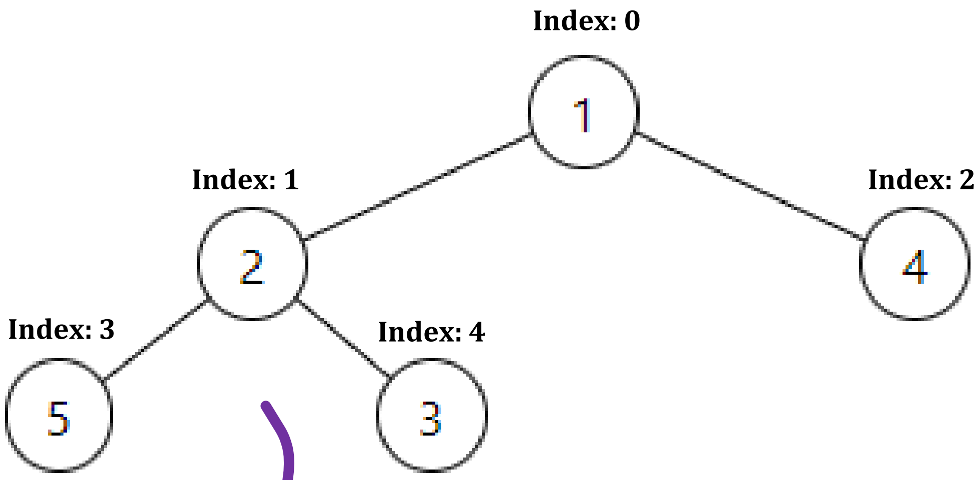
Min-Heap



Max-Heap

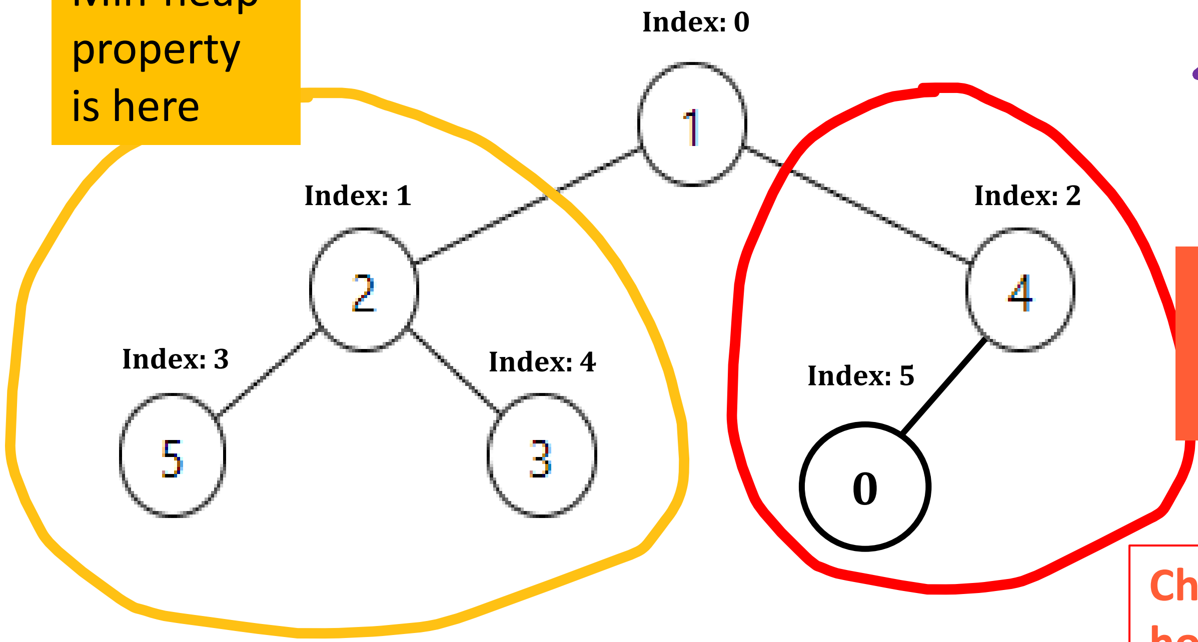
Back to the question

Index	0	1	2	3	4
Value	1	2	4	5	3



Inserting 0

Min-heap property is here

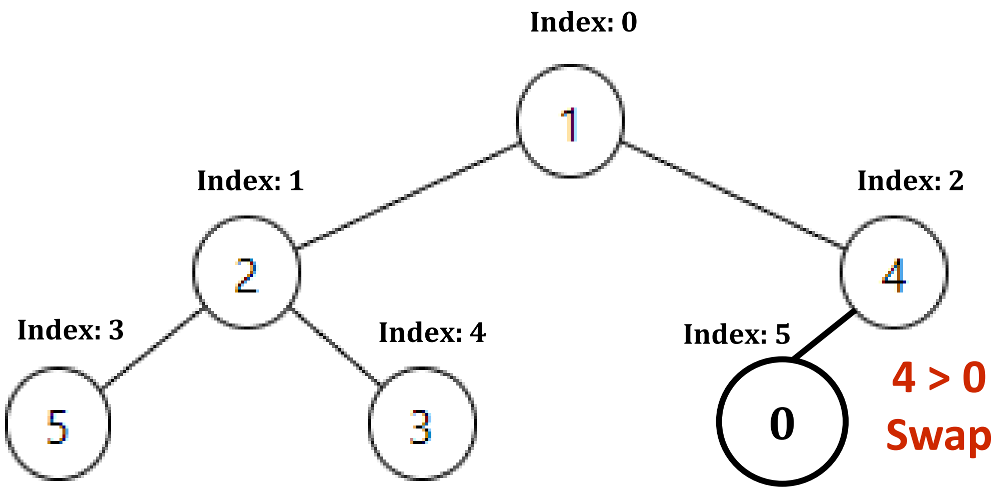


Min-heap property is not here

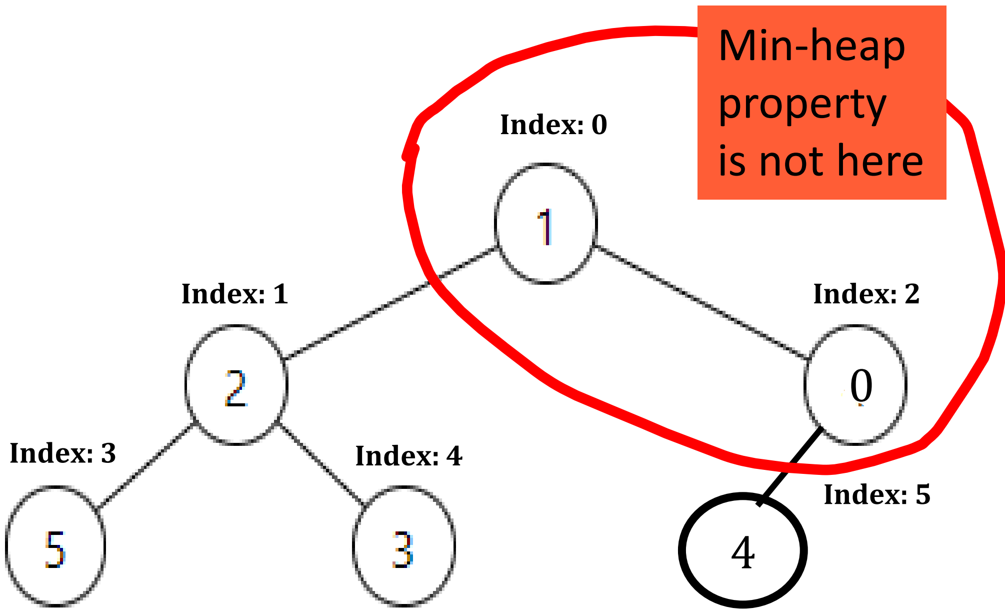
Index	0	1	2	3	4	5
Value	1	2	4	5	3	0

Check whether every sub-tree is holding the min-heap property

Back to the question



Comparison and arrange

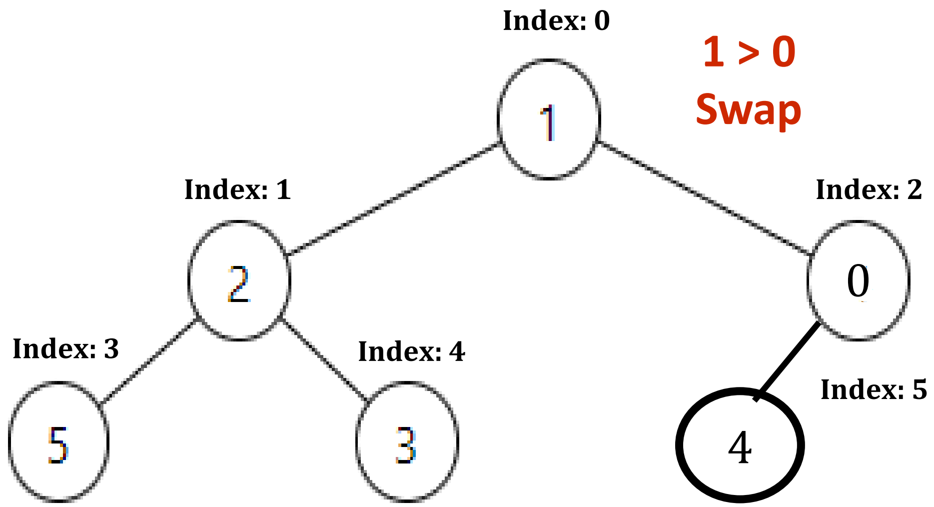


Index	0	1	2	3	4	5
Value	1	2	4	5	3	0

Index	0	1	2	3	4	5
Value	1	2	0	5	3	4

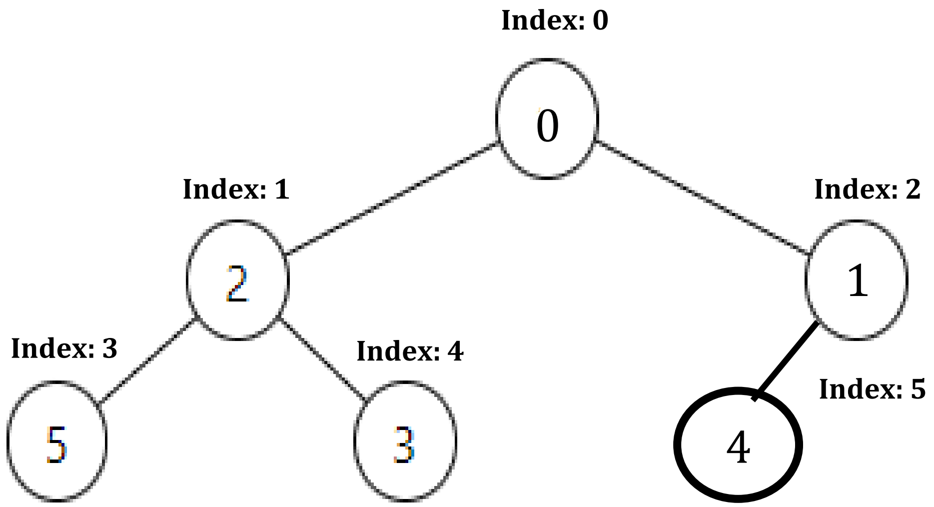
Check whether every sub-tree is holding the min-heap property

Back to the question



1 > 0
Swap

Comparison and arrange



Index	0	1	2	3	4	5
Value	1	2	4	5	3	0

Index	0	1	2	3	4	5
Value	0	2	1	5	3	4

Check whether every sub-tree is holding the min-heap property

- YES, the end

Recap:

// To do

1. Min- Heap
   - Insertion
   - Deletion
2. Max-Heap
   - Insertion
   - Deletion

How to restore the min heap and max heap properties while inserting and deleting nodes

Question 8

Consider the undirected graph given by

$$V = \{a, b, c, d, e\}$$
$$E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}, \{b, c\}, \{c, d\}, \{d, e\}\}.$$

Write the adjacency matrix of this graph.

- **Concept Covered:**

  Graphs→ Representation → Adjacency Matrix

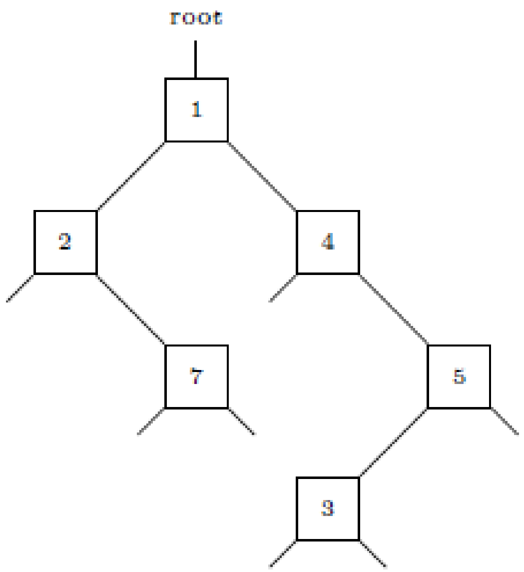
Question 8 – Sample Answer

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

- It’s a 5×5 matrix.
- Since this is an undirected graph, whenever we have an edge from x to y, we have an edge from y to x as well. Therefore this is a **symmetric matrix**.
- In the definition of edges (in set E) we have seven edges, but in the matrix, we have fourteen 1’s.

Question 9

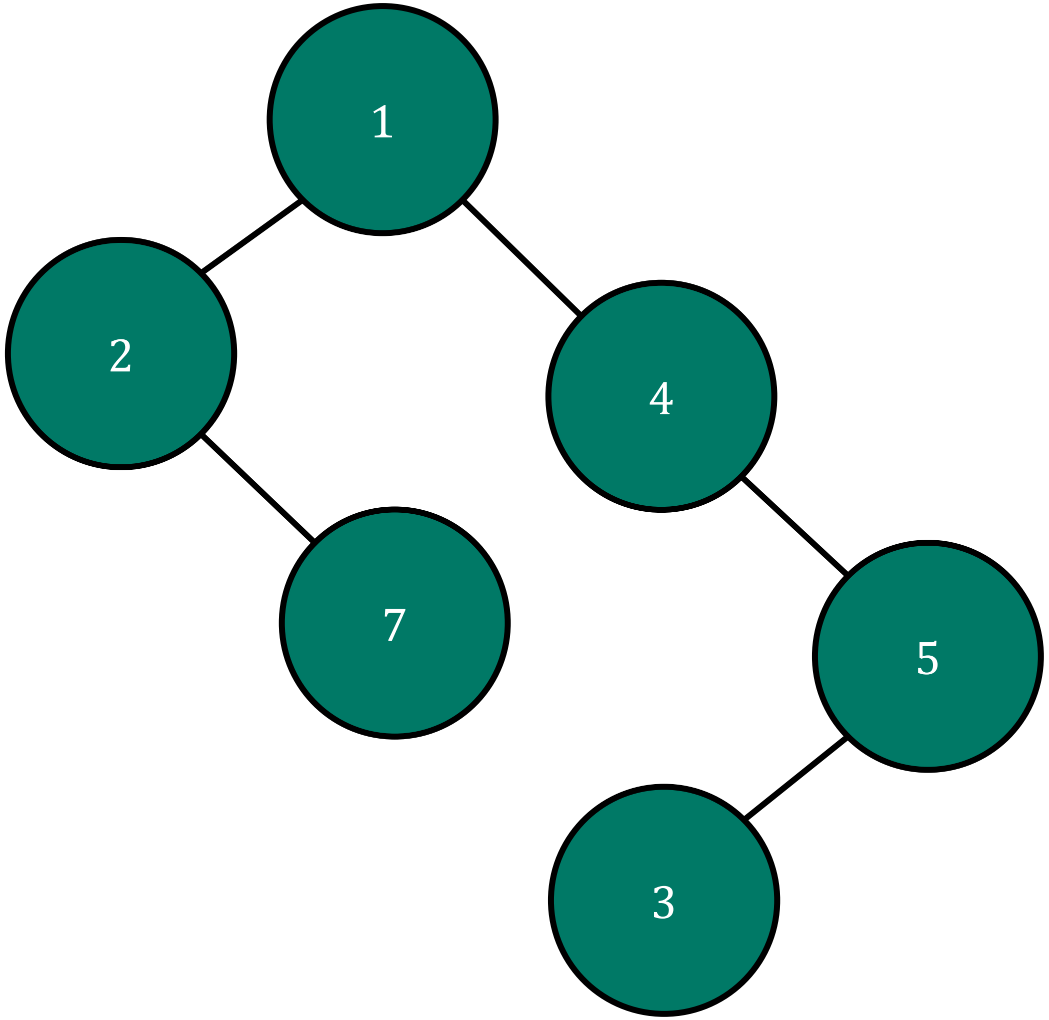
Suppose we use an in-order traversal to output the following tree. What does this traversal do? What is the resulting output?



- **Concept Covered:**

  Binary Trees→ Tree Traversal → In-order

In-order traversal demonstration

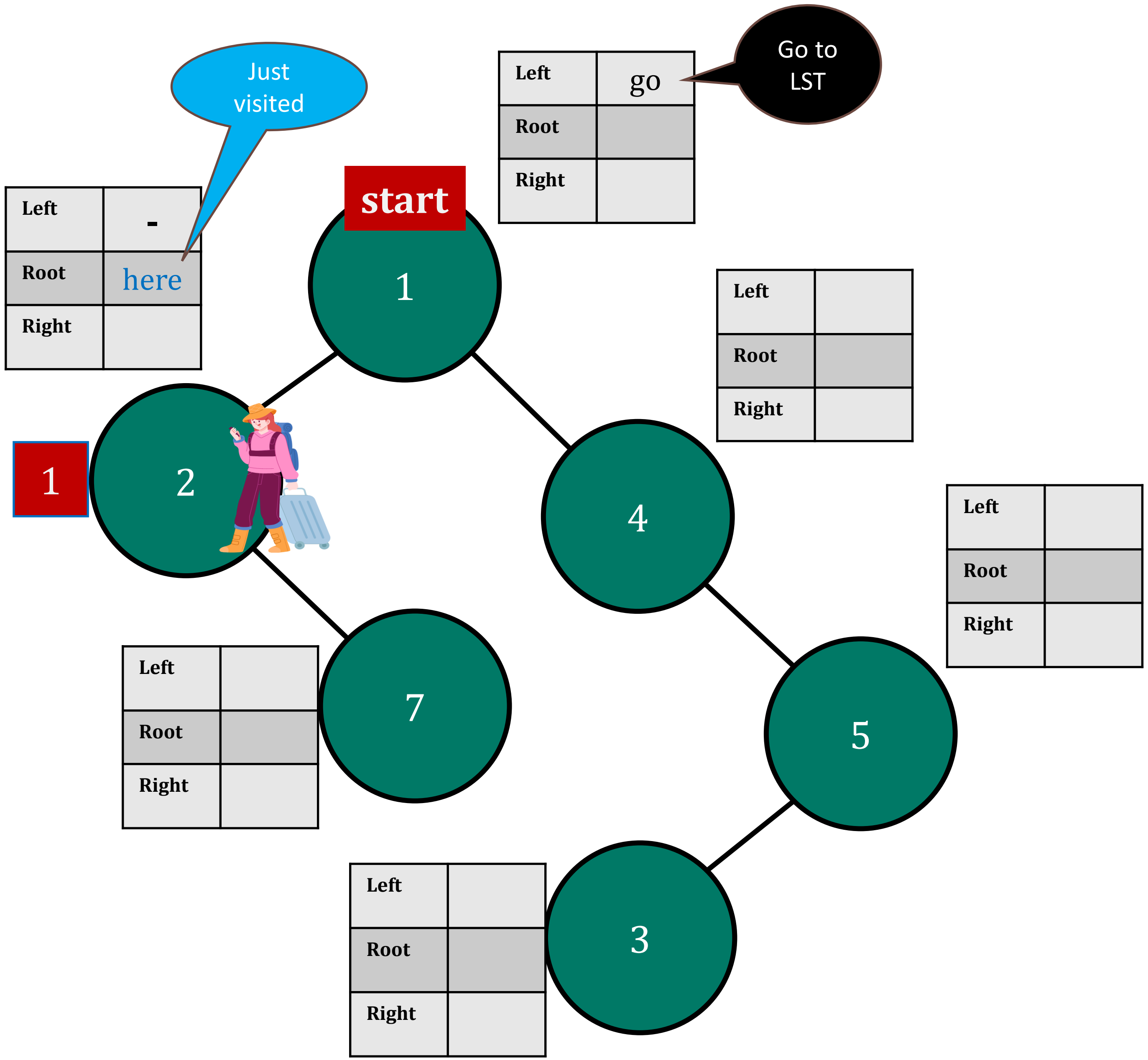


In order tree traversal means

First visit : Left Sub Tree
Next visit: The Root
Last visit: Right Sub Tree

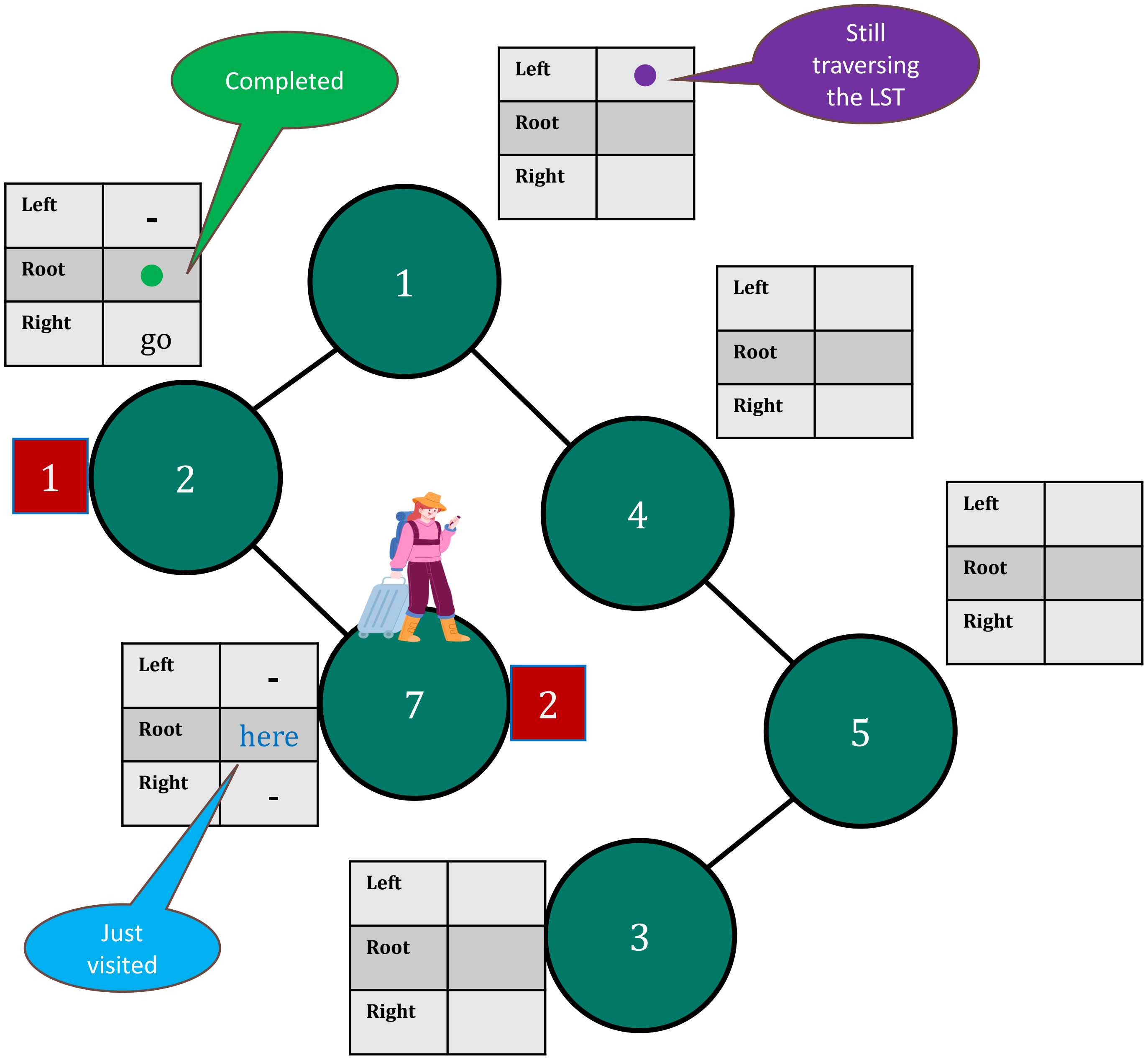
Whenever you are visiting a node, follow the above order

In-order traversal demonstration (cont.)



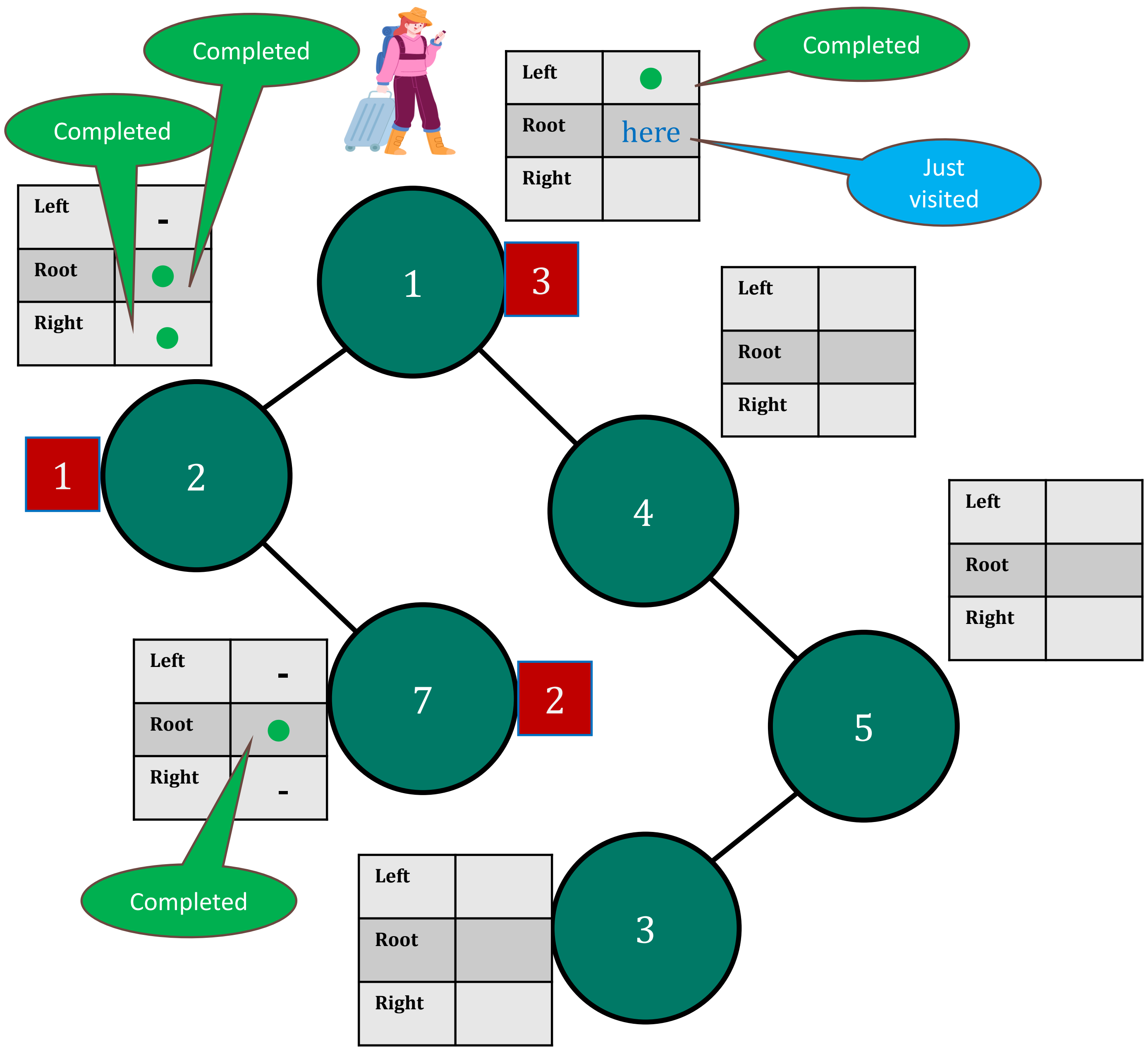
2

In-order traversal demonstration (cont.)



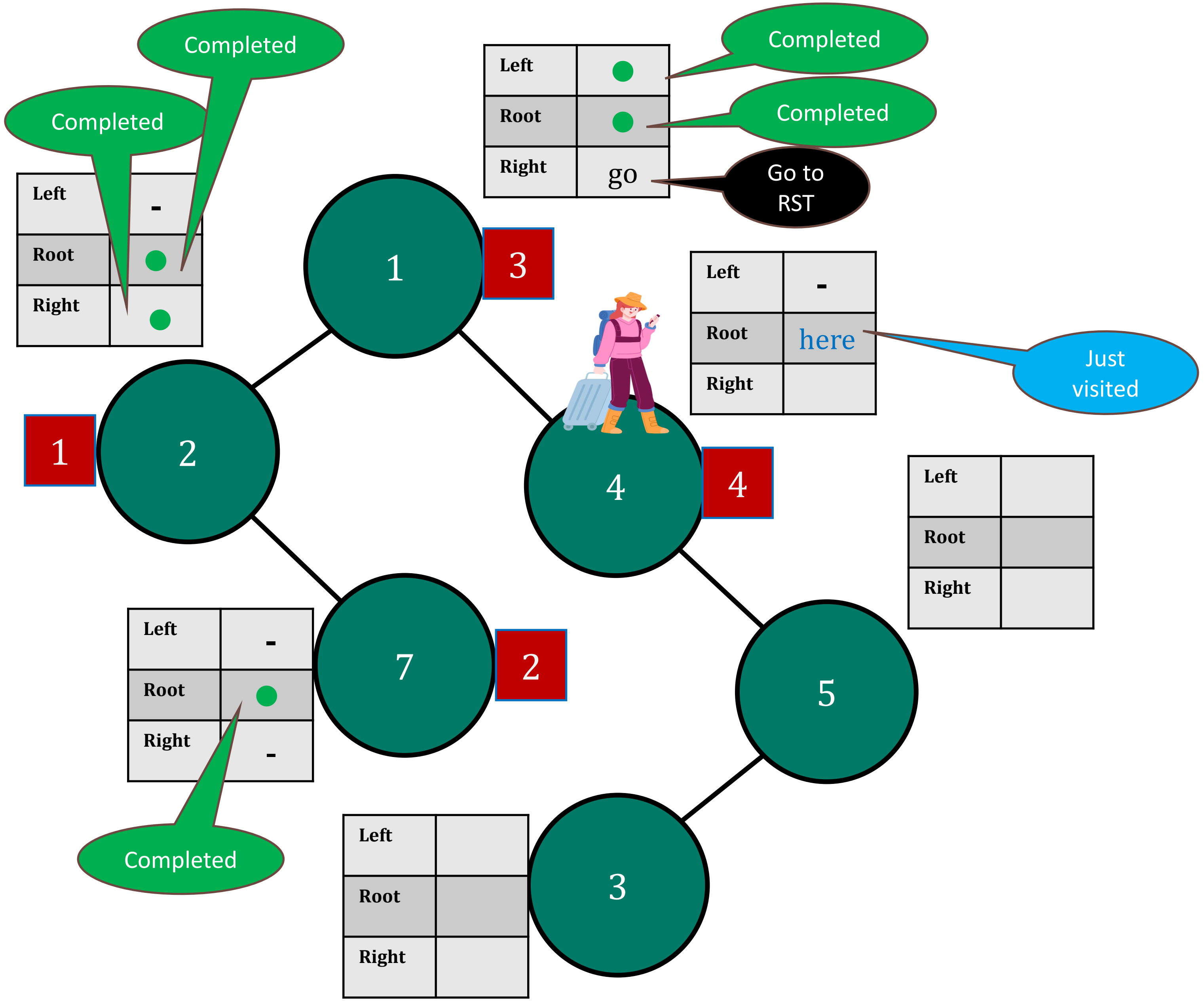
$2 \rightarrow 7$

In-order traversal demonstration (cont.)



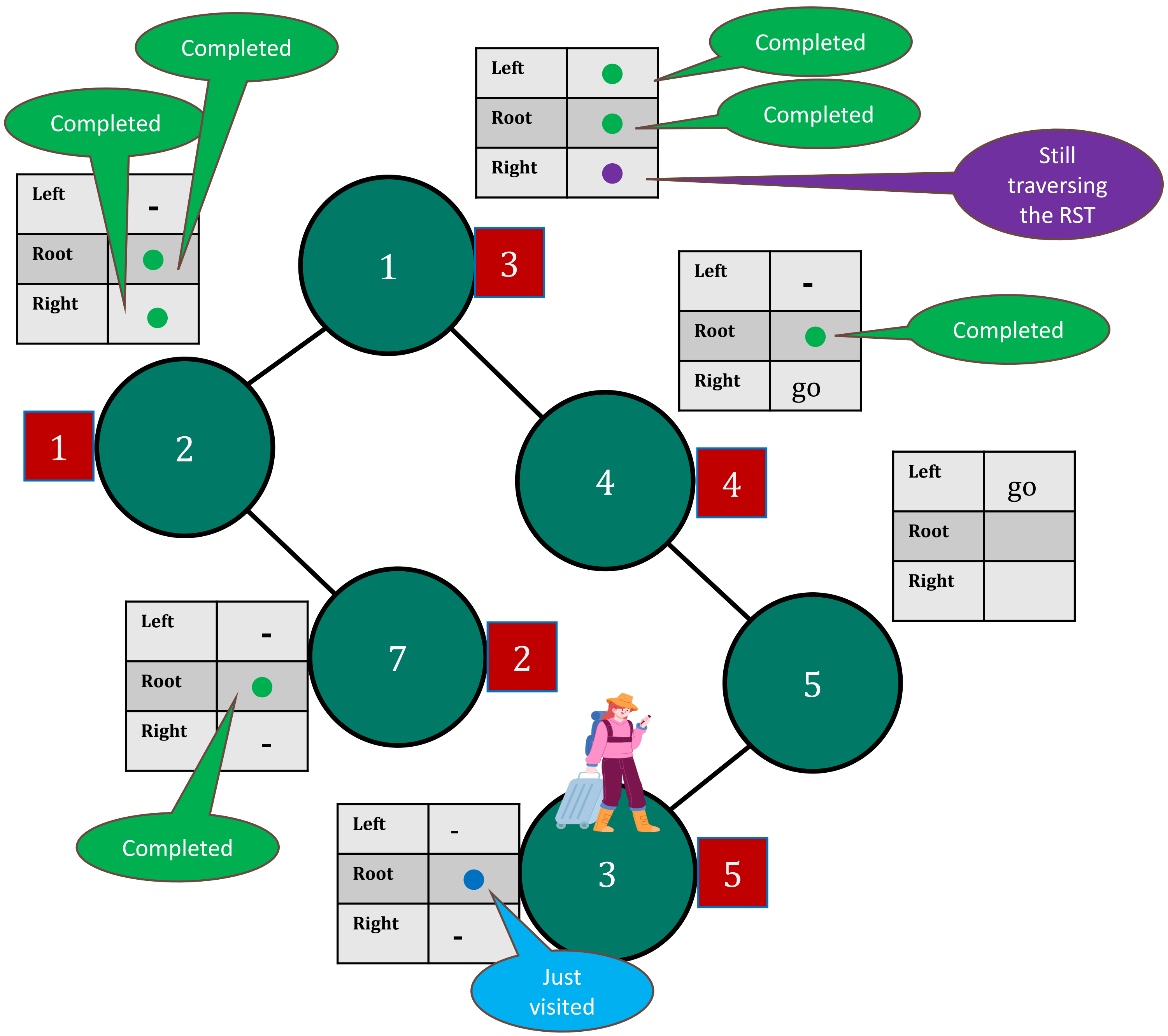
2 → 7 → 1

In-order traversal demonstration (cont.)



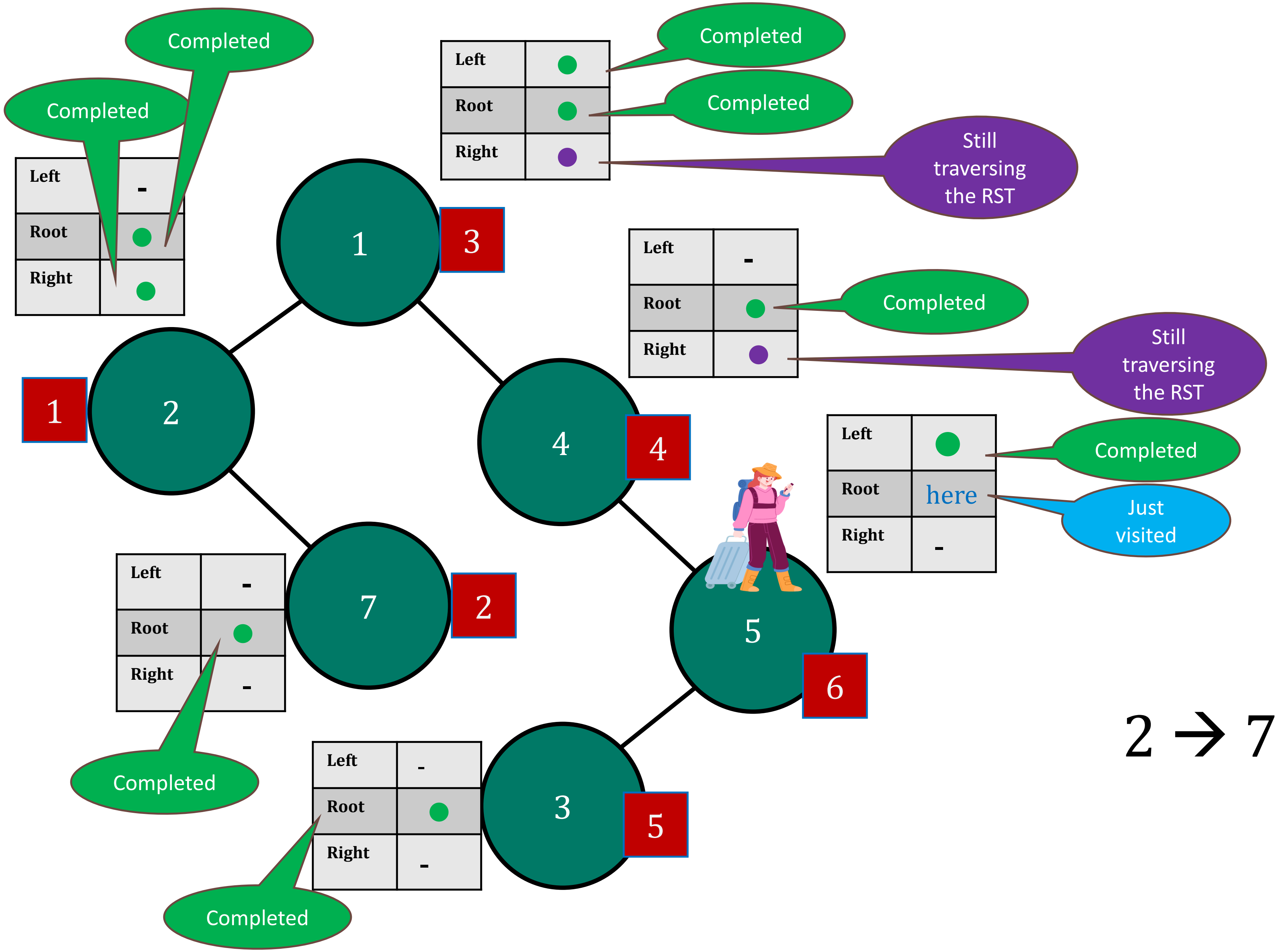
2 → 7 → 1 → 4

In-order traversal demonstration (cont.)



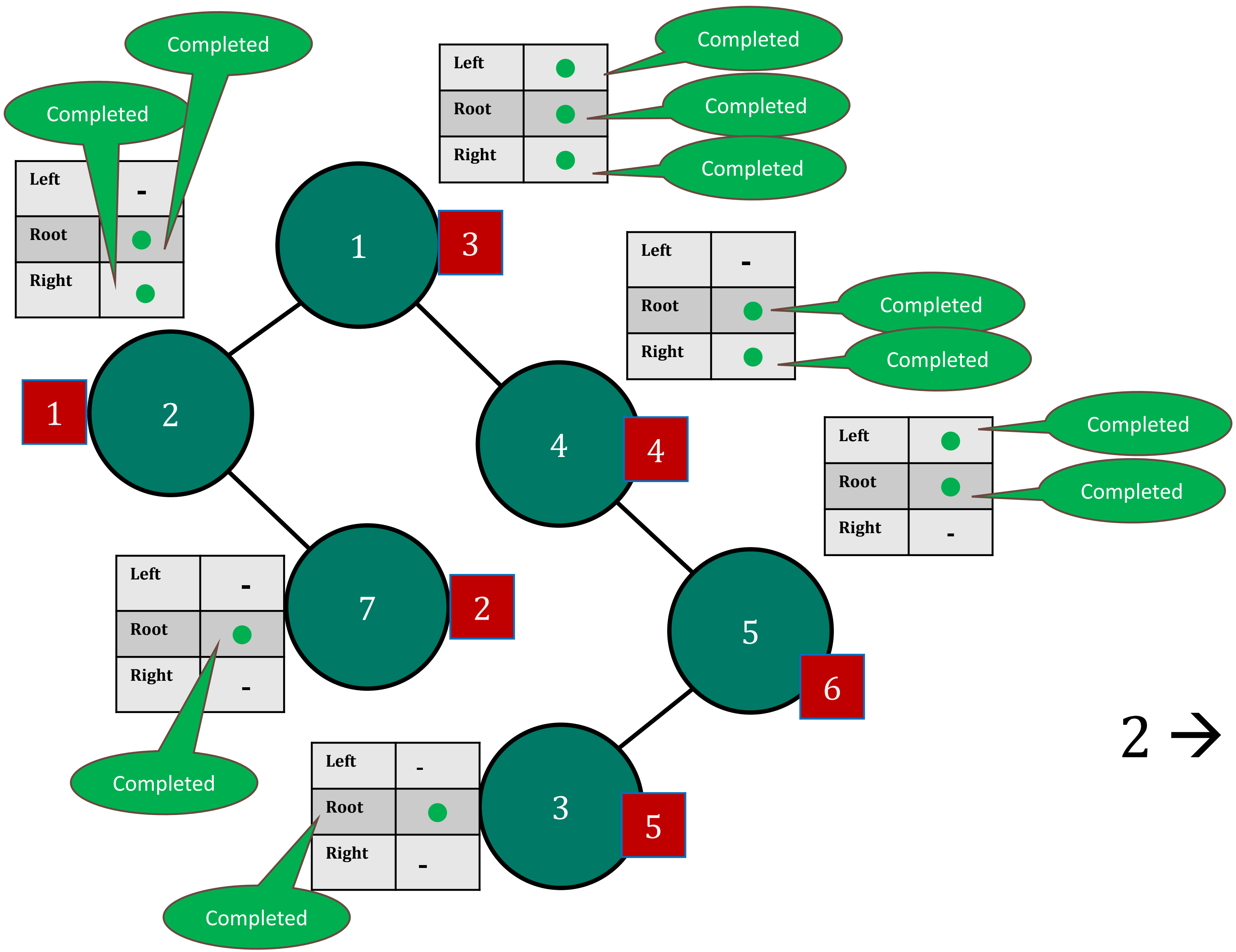
2 → 7 → 1 → 4 → 3

In-order traversal demonstration (cont.)



2 → 7 → 1 → 4 → 3 → 5

In-order traversal demonstration (cont.)



$2 \rightarrow 7 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 5$

Question 10

For this question, consider the following problem:

Input: An array of positive integers Output: An integer that is **not** in the array

What would a brute force algorithm for this problem do? What is its complexity in terms of the size n of the array?

Question 10 – Sample Answer

A brute force algorithm first checks if there is a 1 in the array. If there is a 1, it then checks if there is a 2, and so on.

It checks each possible value by comparing it to all values in the array. For each value it checks, it makes up to n comparisons.

**It will check up to n + 1 values before it find one that is not in the array
for a total of n(n + 1) comparisons, which is in O(n^2).**

Question 10 – Using an example

The input array contains: Positive Integers
Output: An integer that is not in the array

Think of the input array which has a size of 6. (n= 6)
Think about the **worst-case** to determine the time complexity

Input Array

value	6	5	4	3	2	1
index	0	1	2	3	4	6

Input Array =

value	6	5	4	3	2	1
index	0	1	2	3	4	6

The input array contains: Positive Integers
Output: An integer that is not in the array

- We are starting from the smallest positive integer i = 1
- i = 1: 1 checks with 6,5,4,3,2,1 (n comparisons) (why ? → That's what brute force does) No output → increment i
- i = 2: 2 checks with all values in the array (n comparisons) (why ? → That's what brute force does) No output → increment i
- i = 3: 3 checks with all values in the array (n comparisons) (why ? →That's what brute force does) No output → increment i
- i = 4: 4 checks with all the values in the array (n comparisons) (why ? → That's what brute force does) No output → increment i
- i = 5: 5 checks with all the values in the array (n comparisons) (why ? → That's what brute force does) No output → increment i
- i = 6: 6 checks with all the values in the array (n comparisons) (why ? → That's what brute force does) No output → increment i
- i = 7: 7 checks with all the values in the array (n comparisons) (why ? → That's what brute force does) Output 7 (After n+1 checkings)

All together n(n+1) comparisons

O(n (n+1))
O(n^2+n)
O(n^2)
That means the time complexity class is quadratic


GOOD
LUCK