A&D

Exercise Session 6

Nil Ozer

A&D Overview

A&D

Math Basics
- Asymp. Notation
- Induction
- Loop-Counting

Max-Subarray-Sum
- naive, divide-conquer, inductive
- complexity

Searchs
- Linear Search
- Binary Search
- Lower-Bound

Sorts
- Bubble Sort
- Selection Sort
- Insertion Sort
- merge Sort
- Quicksort
- Heapsort
- Lower-Bound

Data Structures
- Array
- List (Linked, Doubly)
- Trees (BST, Heap Tree, AVL Tree)

DP
- Fibonacci
- Max-Subarray-Sum
- Jump-Game
- Longest-Common-Subseq.
- Edit-Distance
- Subset Sum
- Knapsack
- Longest Ascending Subseq.

Graph Definitions
- Definitions
- Eulerian Tours
- Topological Sorting

Graph Searchs
- DFS
- BFS

Shortest Paths

one-to-all
- BFS usage
- Dijkstra
- Bellman-Ford
- negative-closed w. detection

all-to-all
- One-to-all usage
- Floyd-Warshall
- Johnson
- #walks using A_G

- overview

MST
- Prim
- Boruvka
- Kruskal

Outline

- Quiz
- Exercise Sheets
- Data Structures II
- DP I

Quiz

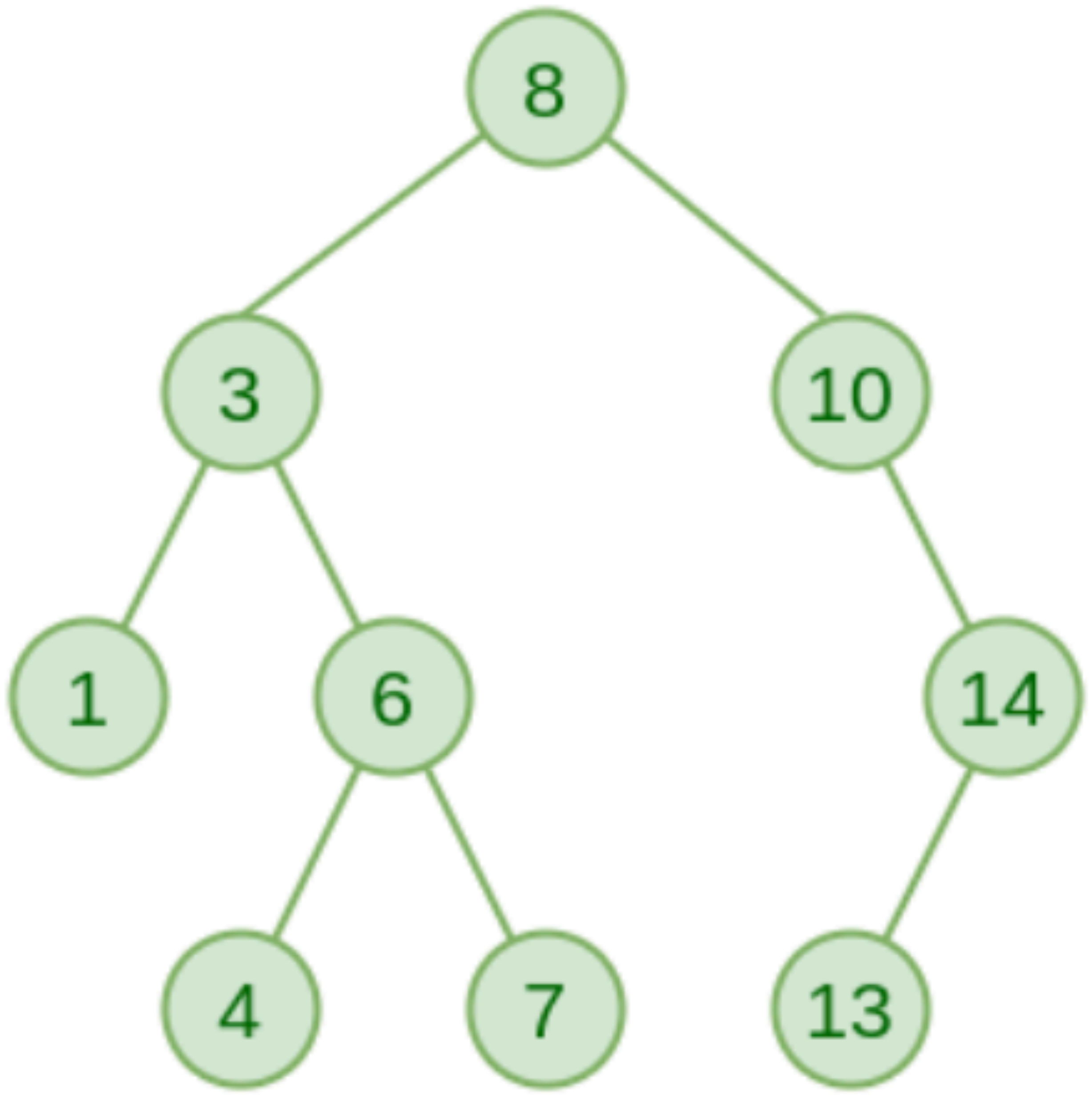
Exercise Sheets

- Exercise Sheet 4 bonus feedback left for next time
- Exercise Sheet 5 non-bonus questions left for next time

- Exercise Sheet 5 peergrading
  - 5.3 this week
  - Emails will be sent

Data Structures II

BST
Terminology



Root Node: The topmost node of the heap. Holds the maximum element !

Parent Node: A node that has one or more child nodes.

Child Node: A node directly connected to another node when moving away from the root.

Leaf Node: A node with no children (located at the bottom level).

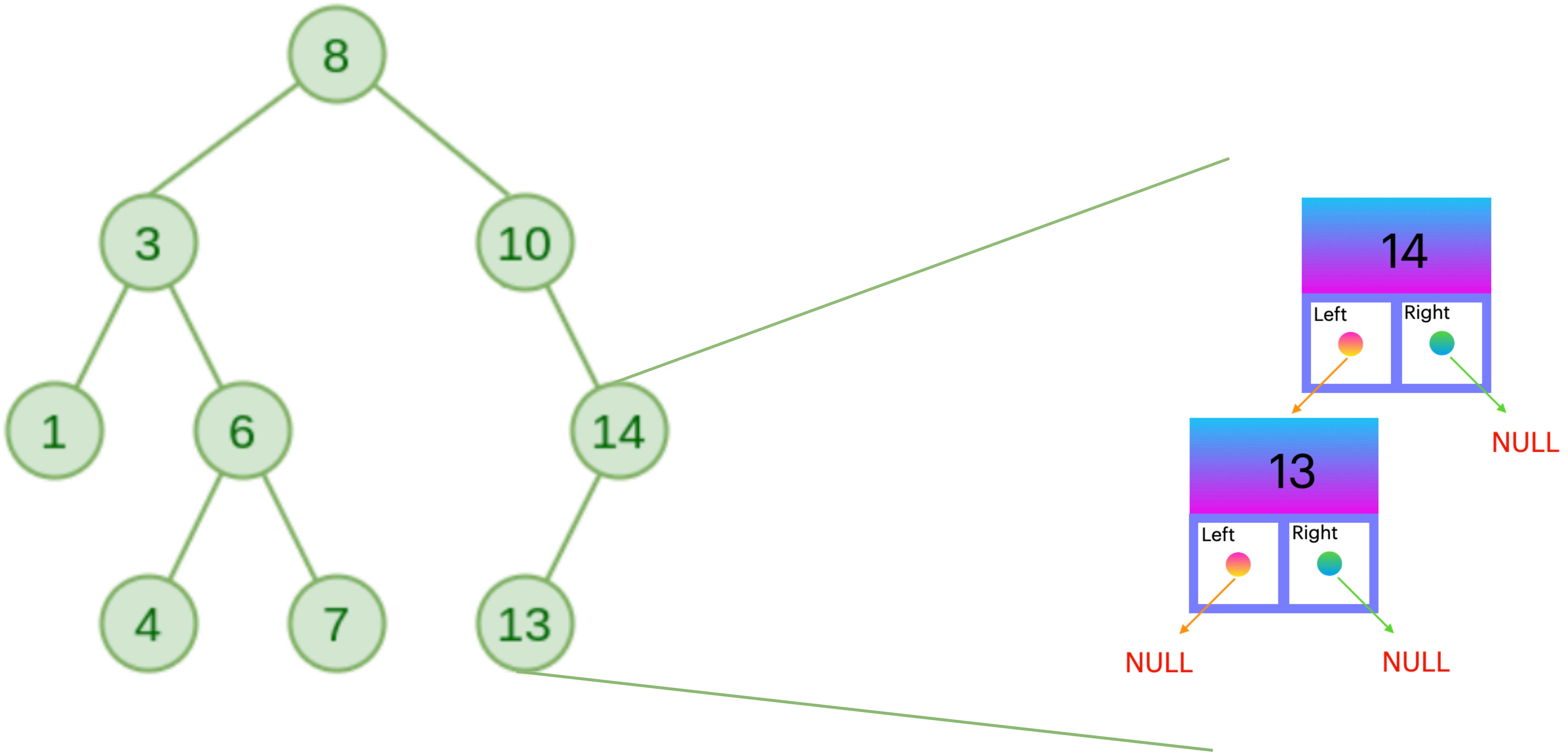
Sibling Nodes: Nodes that share the same parent.

Level: The depth or layer of the node, where the root is at level 0.

Height: The longest path from the root node to a leaf.

Height of the root = 1

BST
Terminology



Root Node: The topmost node of the heap. Holds the maximum element !

Parent Node: A node that has one or more child nodes.

Child Node: A node directly connected to another node when moving away from the root.

Leaf Node: A node with no children (located at the bottom level).

Sibling Nodes: Nodes that share the same parent.

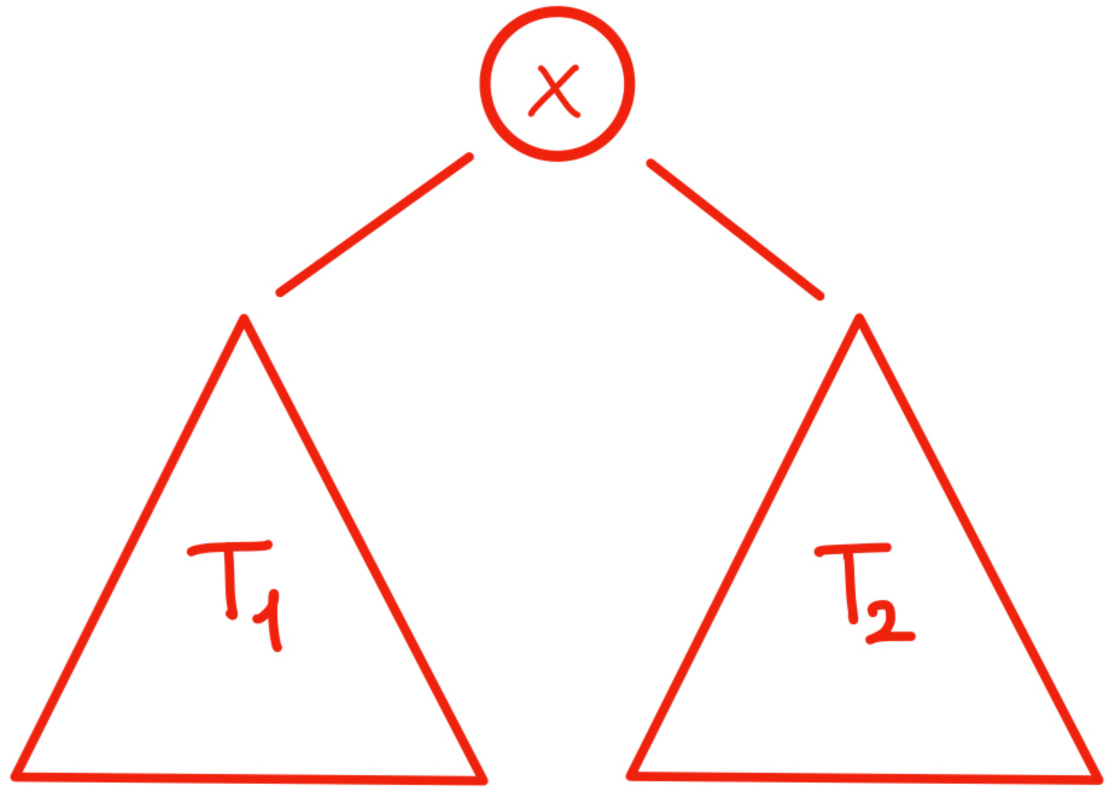
Level: The depth or layer of the node, where the root is at level 0.

Height: The longest path from the root node to a leaf.

Height of the root = 1

BST Condition

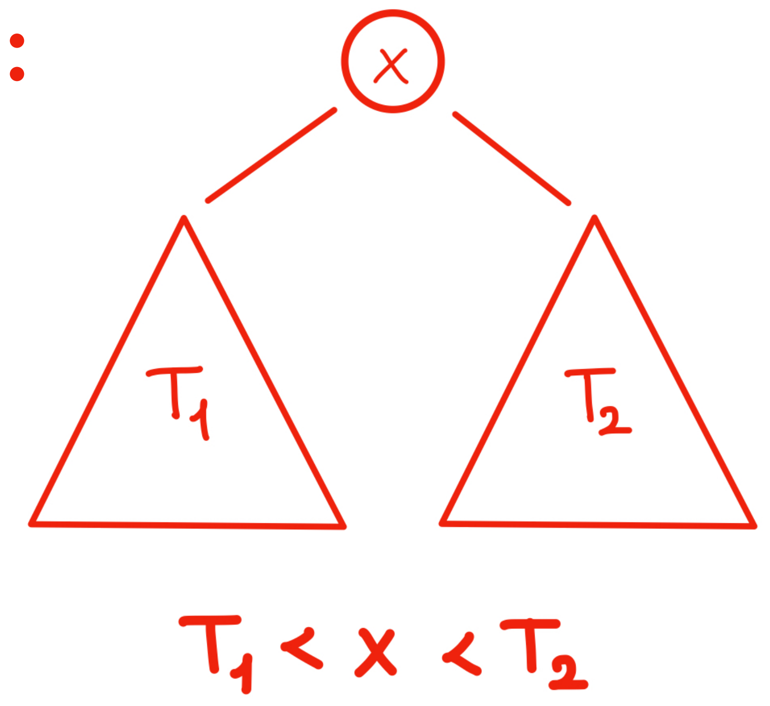
Each node in a BST has at most two children, left child and a right child, with the left child containing values the parent node and the right child containing values greater than the parent node. Every node in the left subtree is less than the root, and every node in the right subtree is greater than the root.

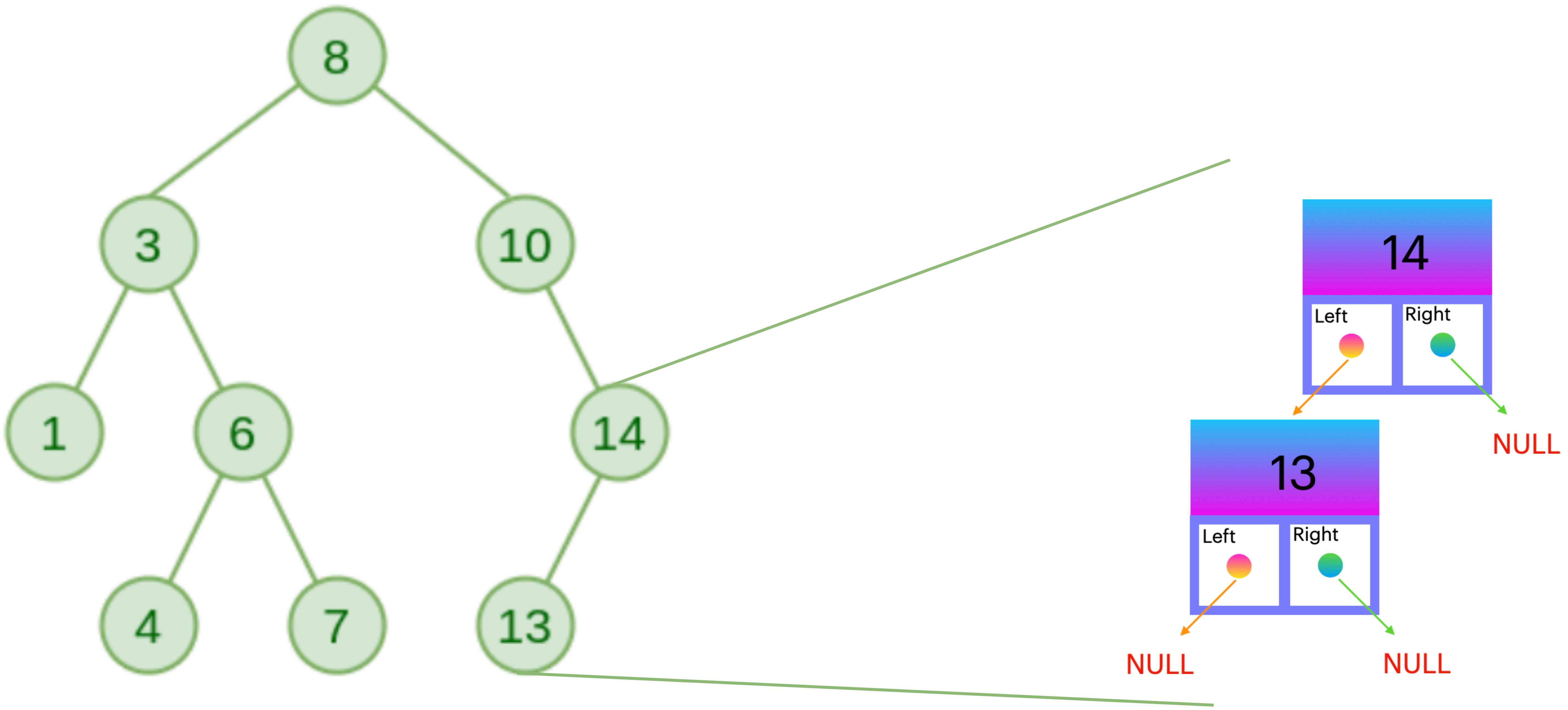


$$T_1 < x < T_2$$

BST

BST Condition :





Root Node: The topmost node of the heap. Holds the maximum element !

Parent Node: A node that has one or more child nodes.

Child Node: A node directly connected to another node when moving away from the root.

Leaf Node: A node with no children (located at the bottom level).

Sibling Nodes: Nodes that share the same parent.

Level: The depth or layer of the node, where the root is at level 0.

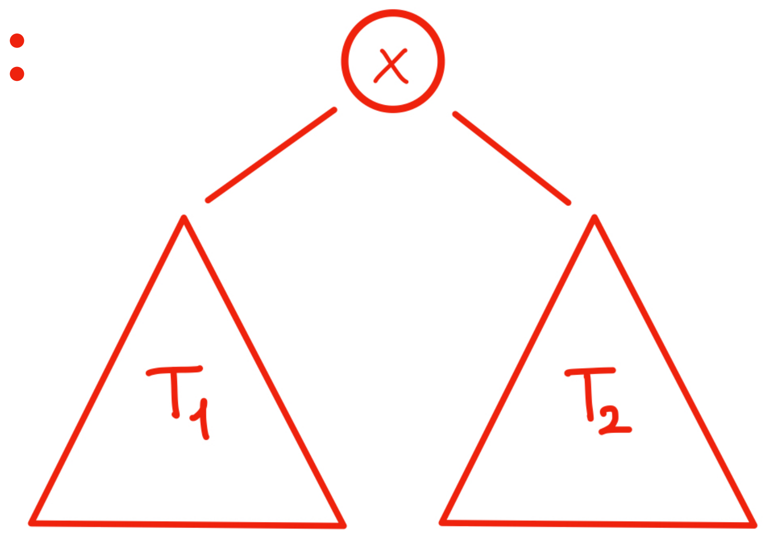
Height: The longest path from the root node to a leaf.

Height of the root = 1

BST

Search(x)

BST Condition :



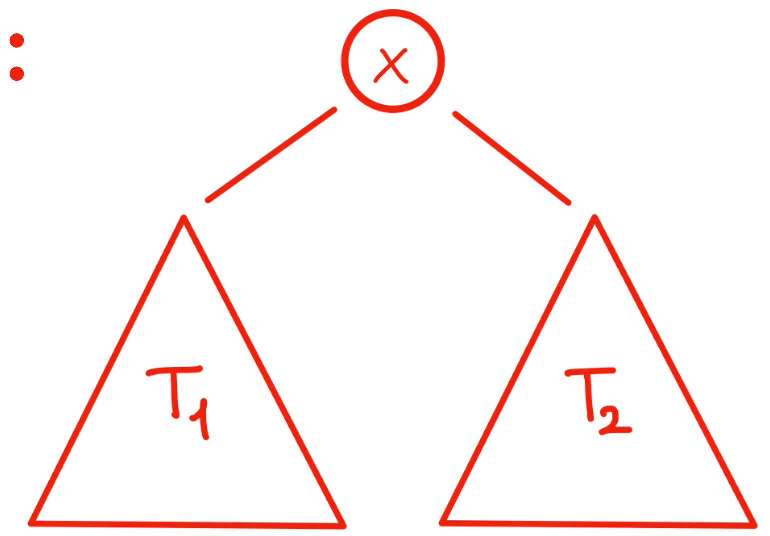
$T_1 < x < T_2$

We use the BST condition

1. If curr == null , you're at the leaf and you haven't found x. Your search is done
2. If curr == x , you've found x !!
3. If curr < key , search the right subtree
4. If curr > key , search the left subtree
5. Repeat 3 and 4 starting from the root until you have one of the cases 1 and 2

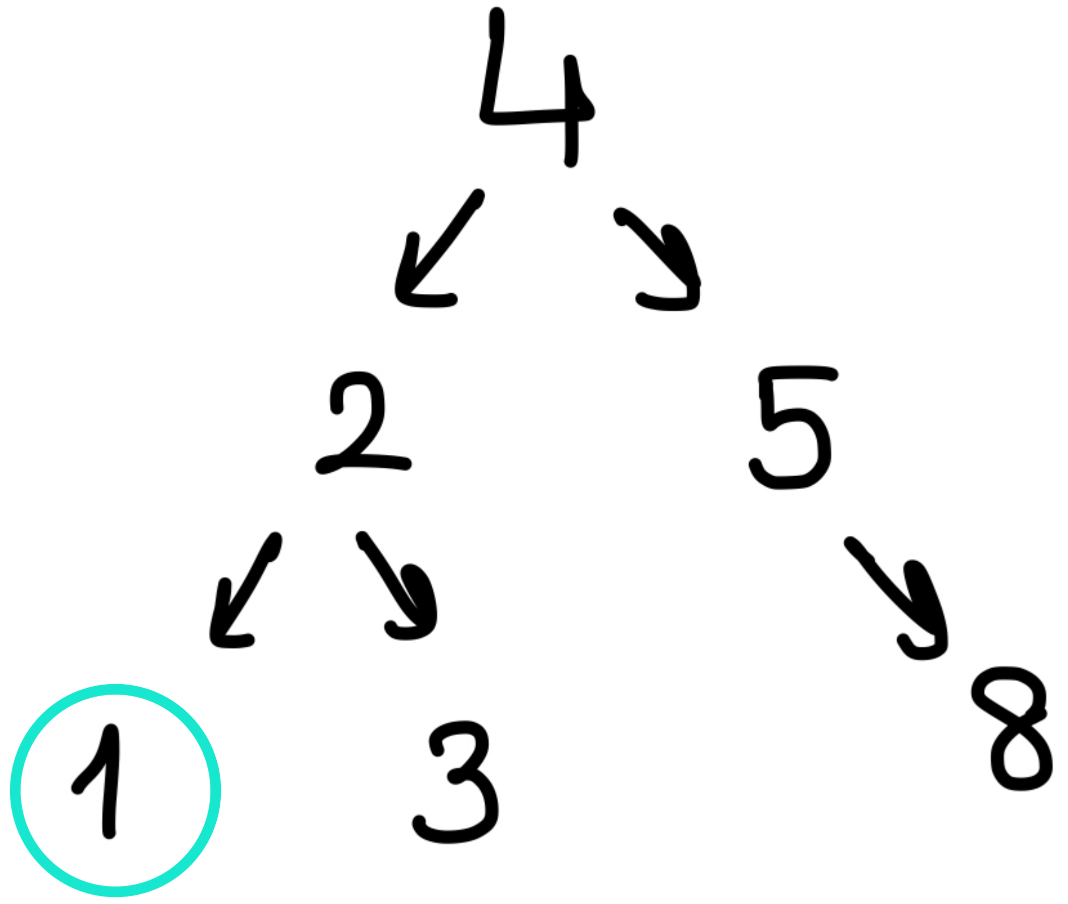
BST
remove(x)

BST Condition :

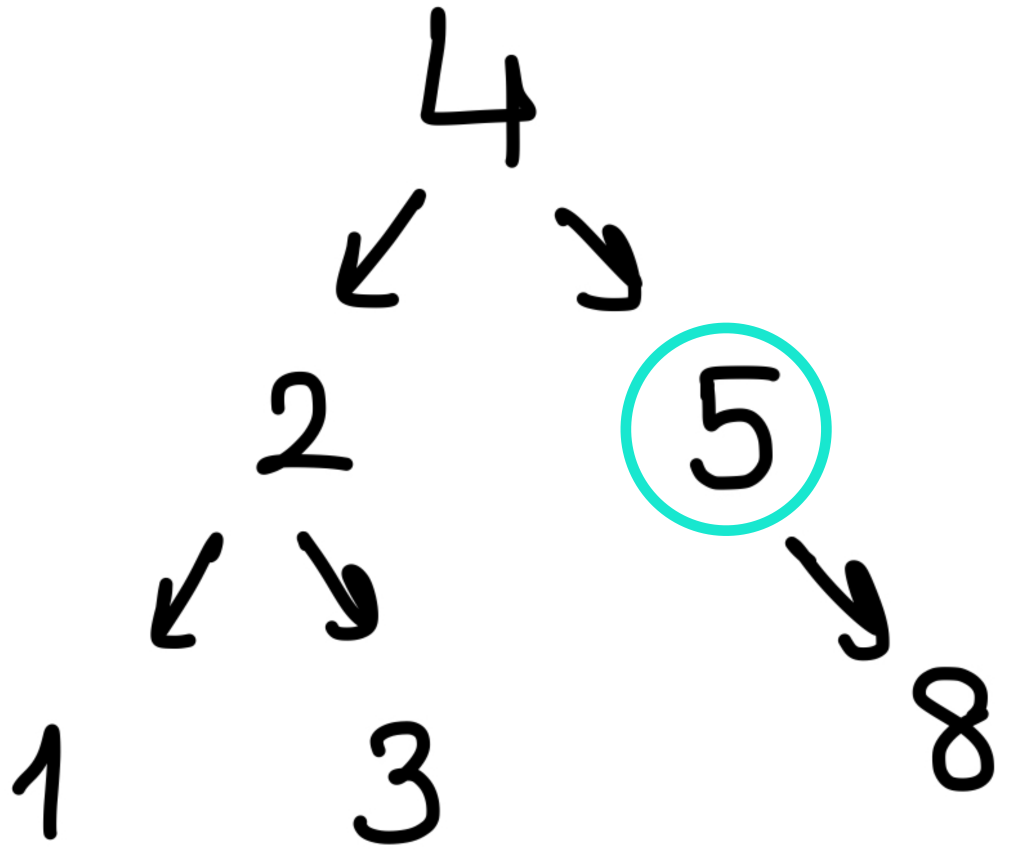


$T_1 < x < T_2$

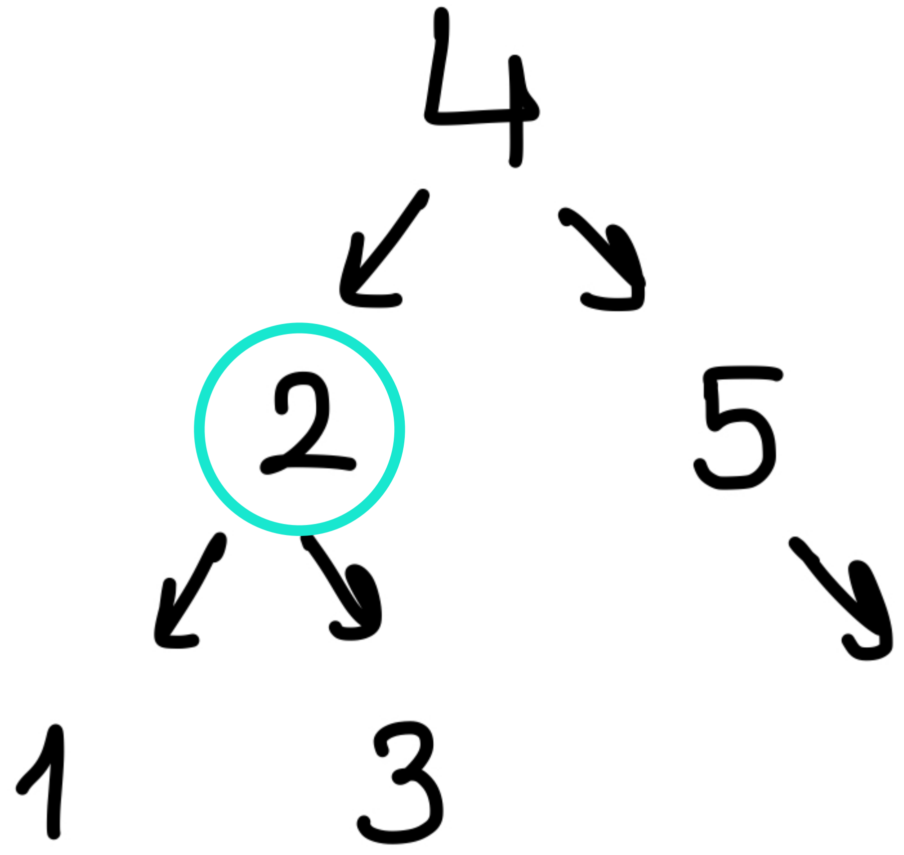
Case 1 : x has no children



Case 2: x has 1 child

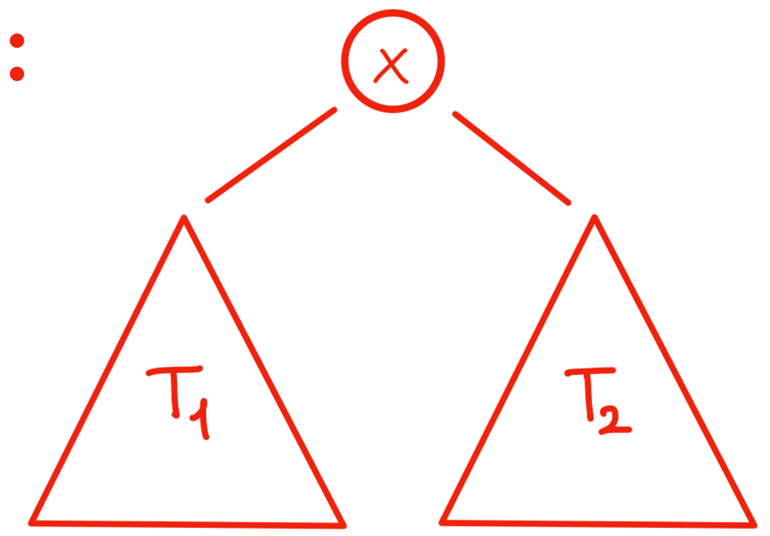


Case 3 : x has 2 children



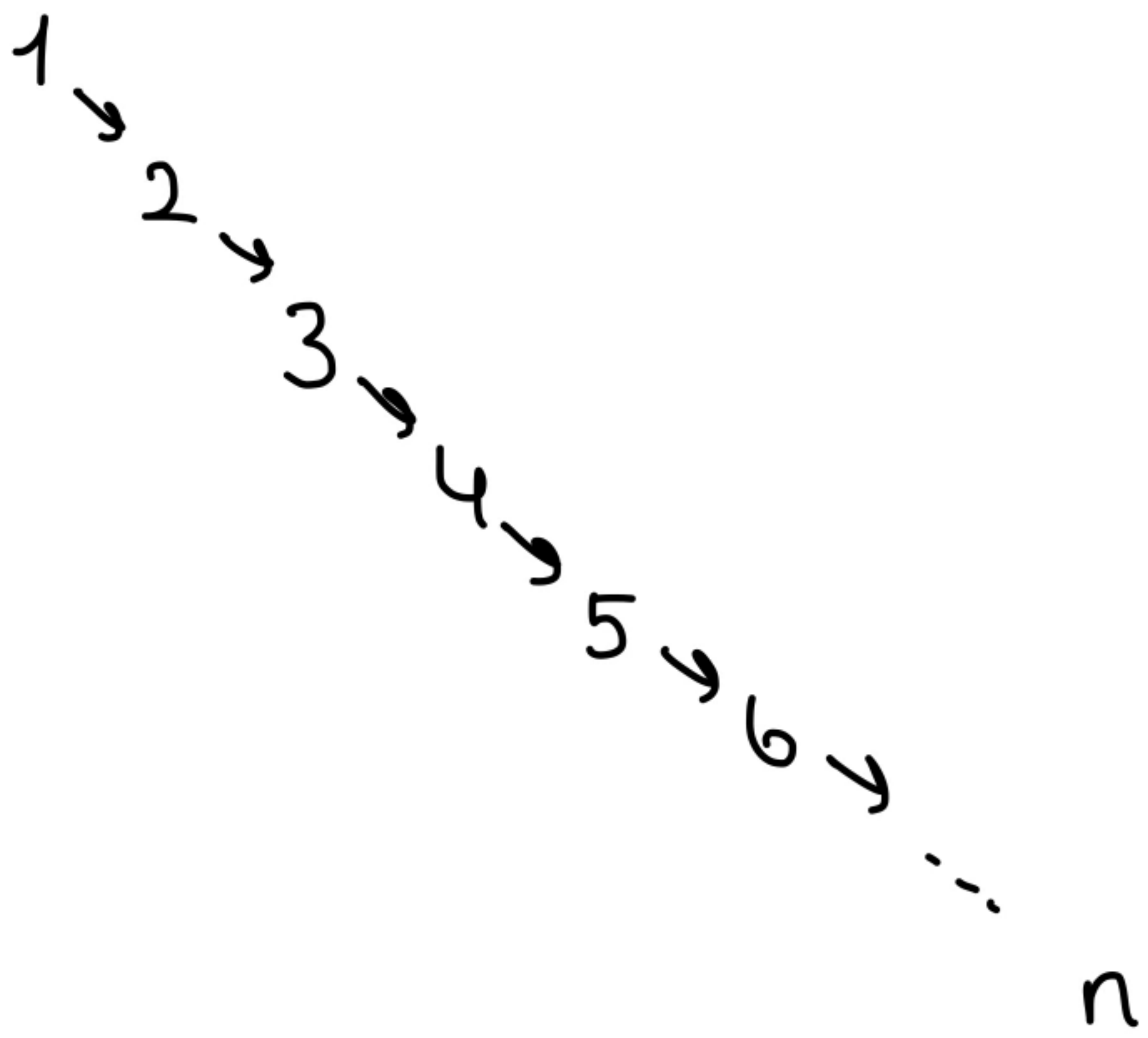
BST
Problem

BST Condition :



$T_1 < x < T_2$

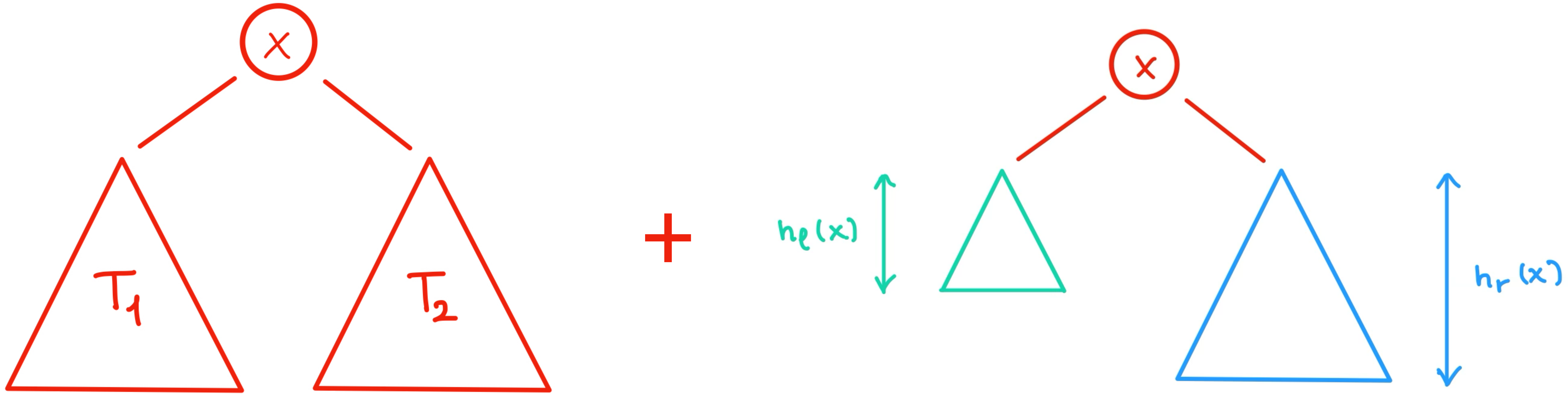
- Searching is in O(h)
- Our h doesn't have to be log(n). There's a case where h = n :



Still a BST !!

AVL Tree Condition

AVL tree is a self-balancing BST where the difference between heights of left and right subtrees cannot be more than one for all nodes.

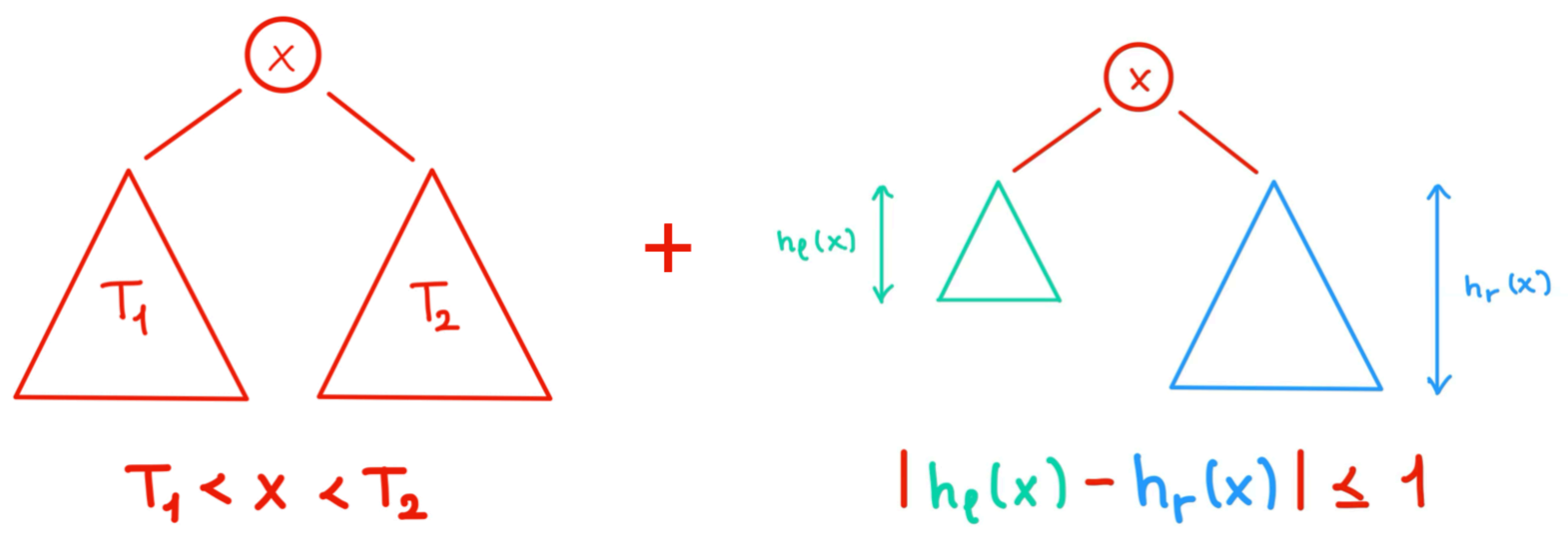


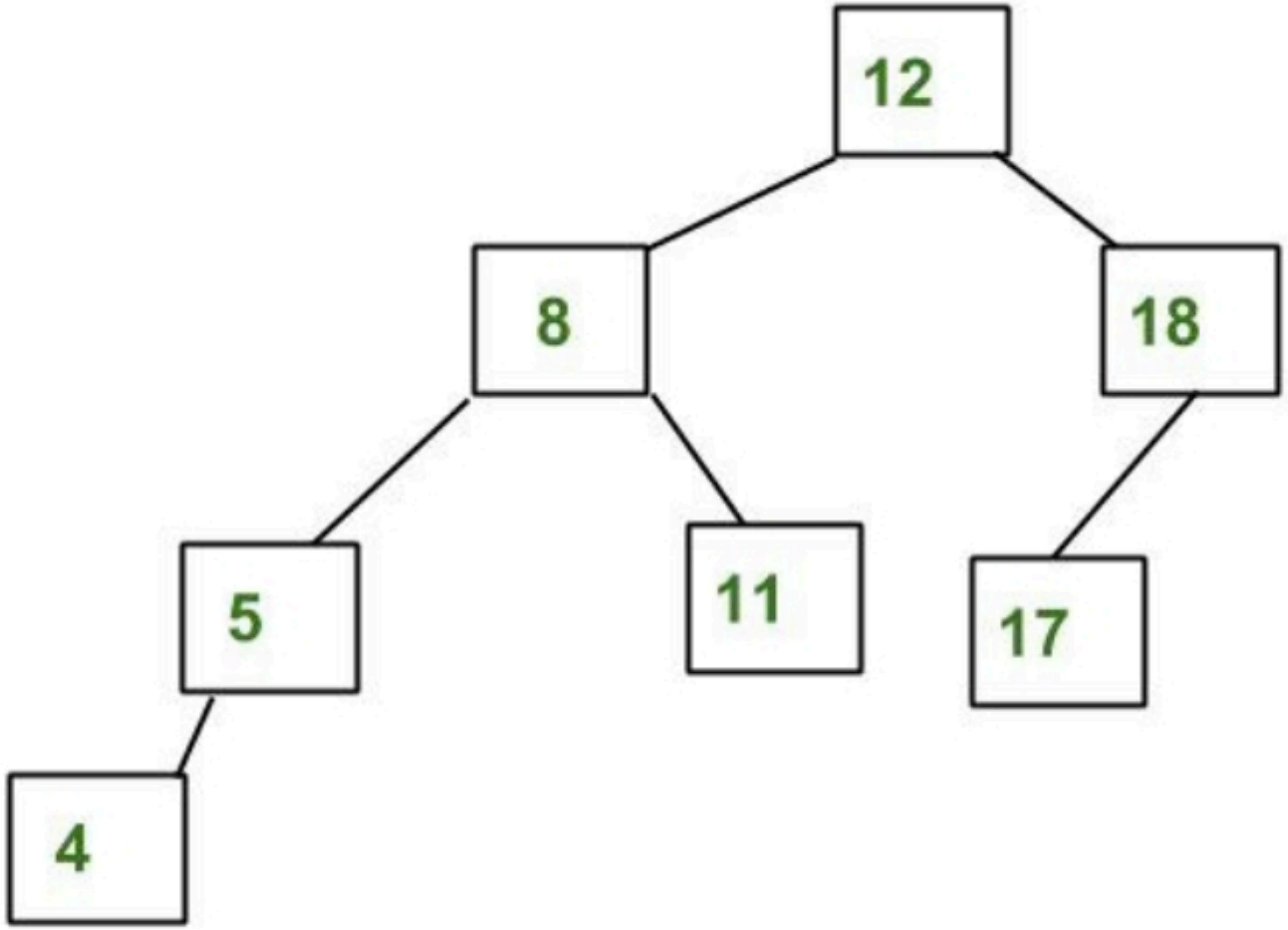
$$T_1 < x < T_2$$

$$|h_l(x) - h_r(x)| \leq 1$$

AVL-Tree or not ?

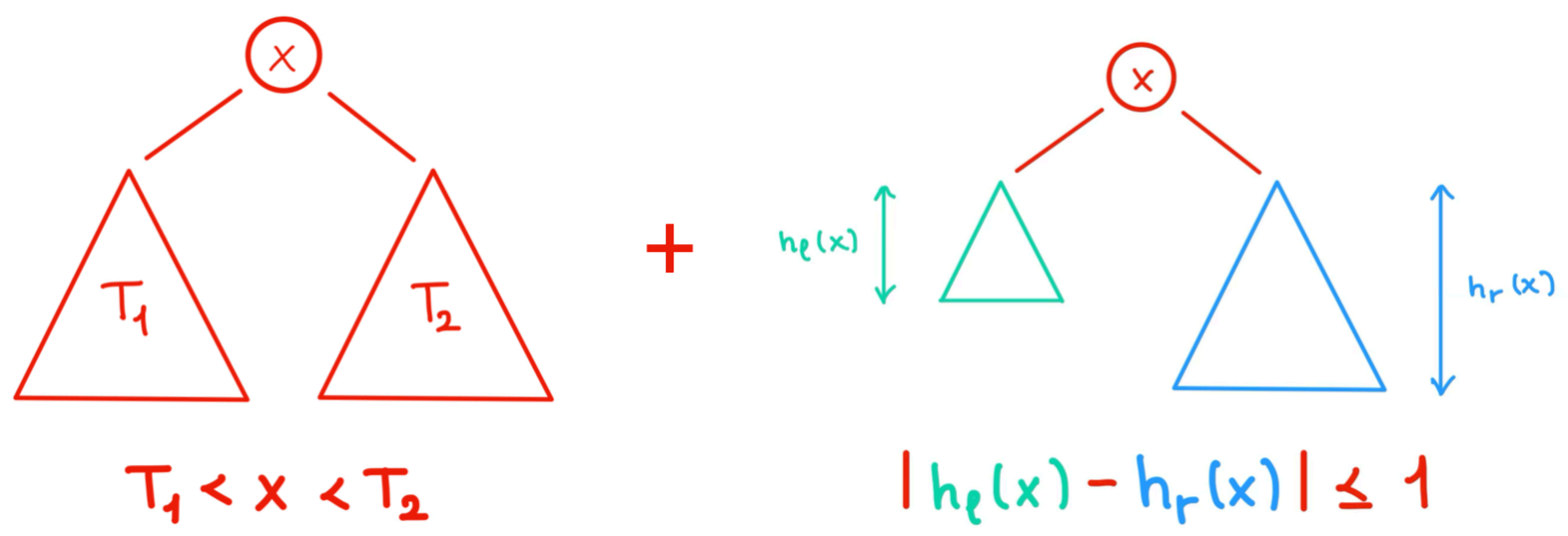
AVL-Tree Condition :

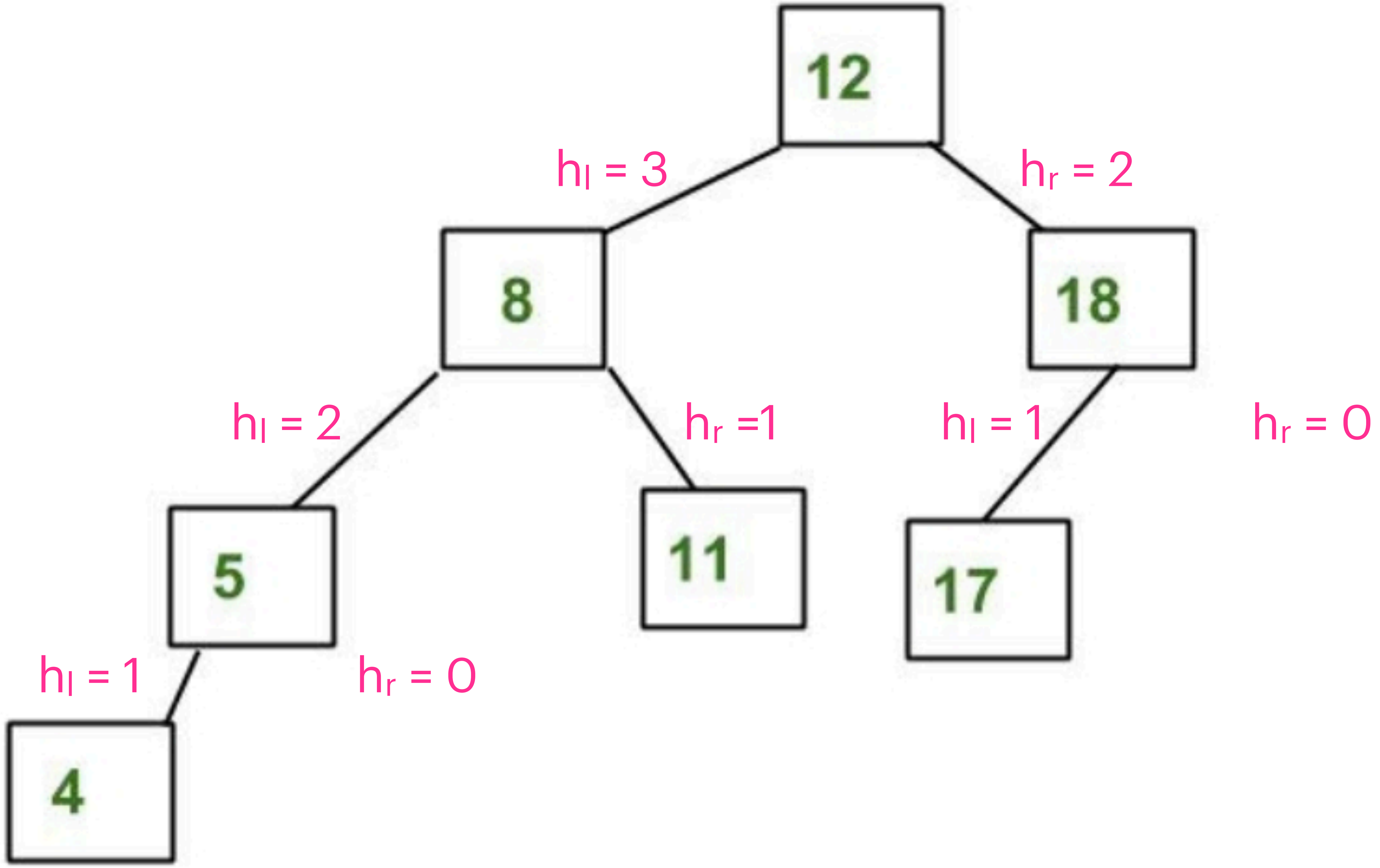




AVL-Tree or not ?

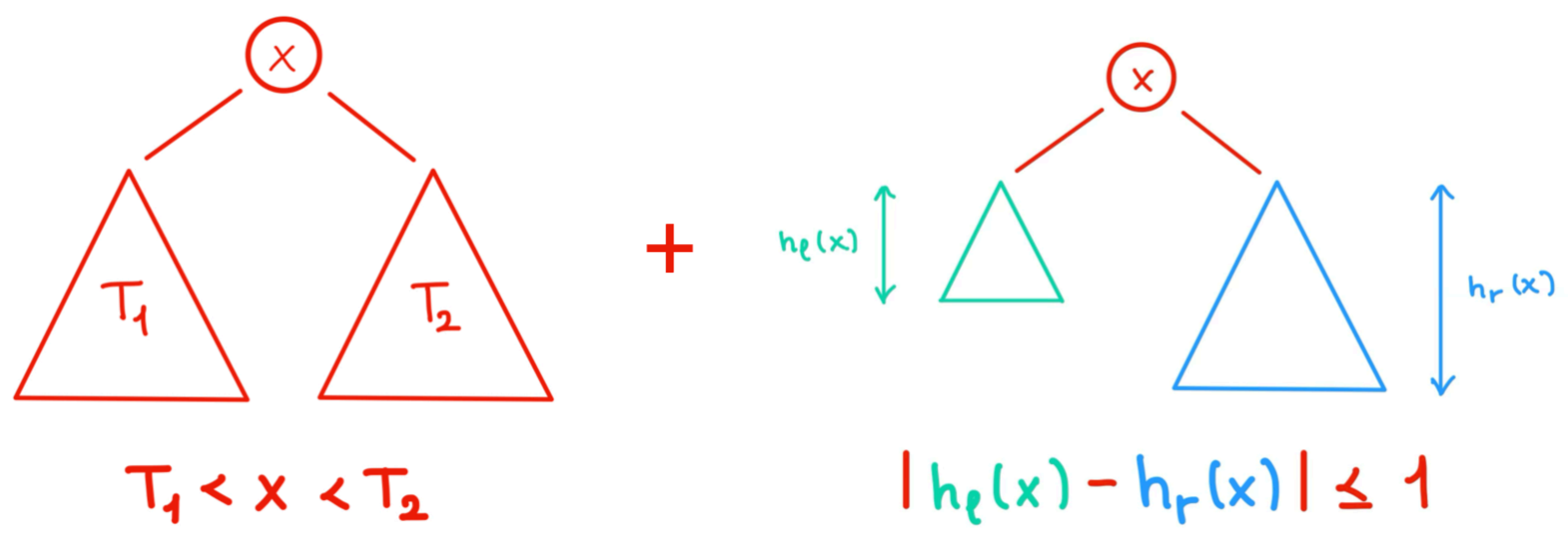
AVL-Tree Condition :

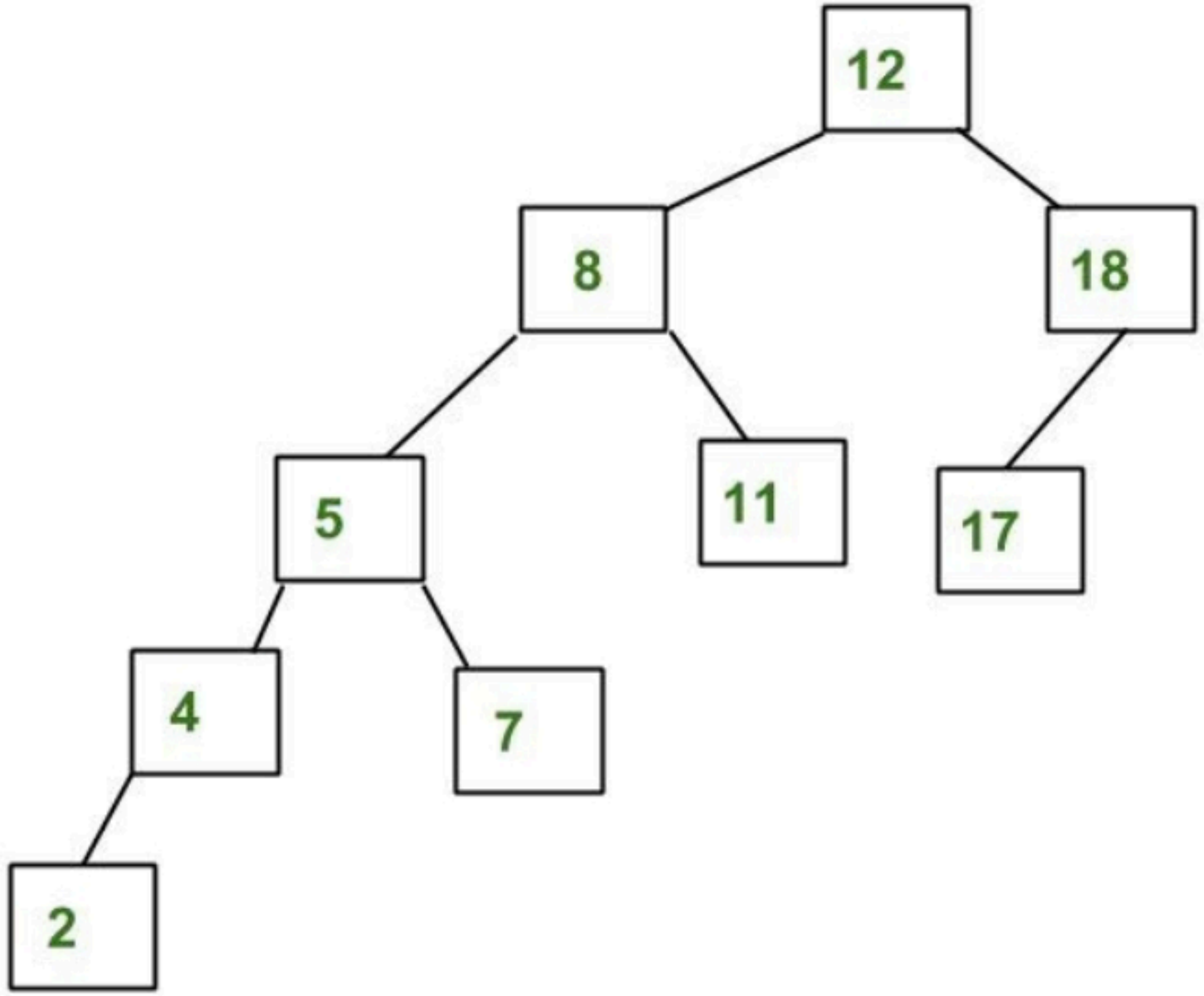




AVL-Tree or not ?

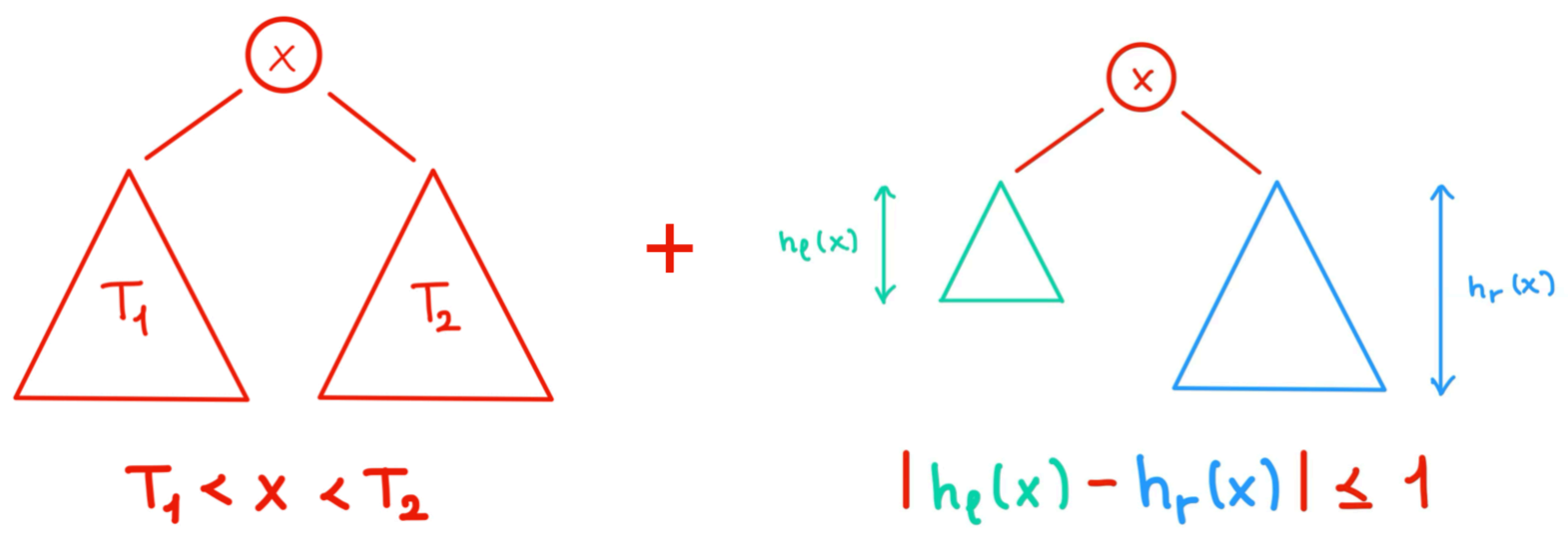
AVL-Tree Condition :

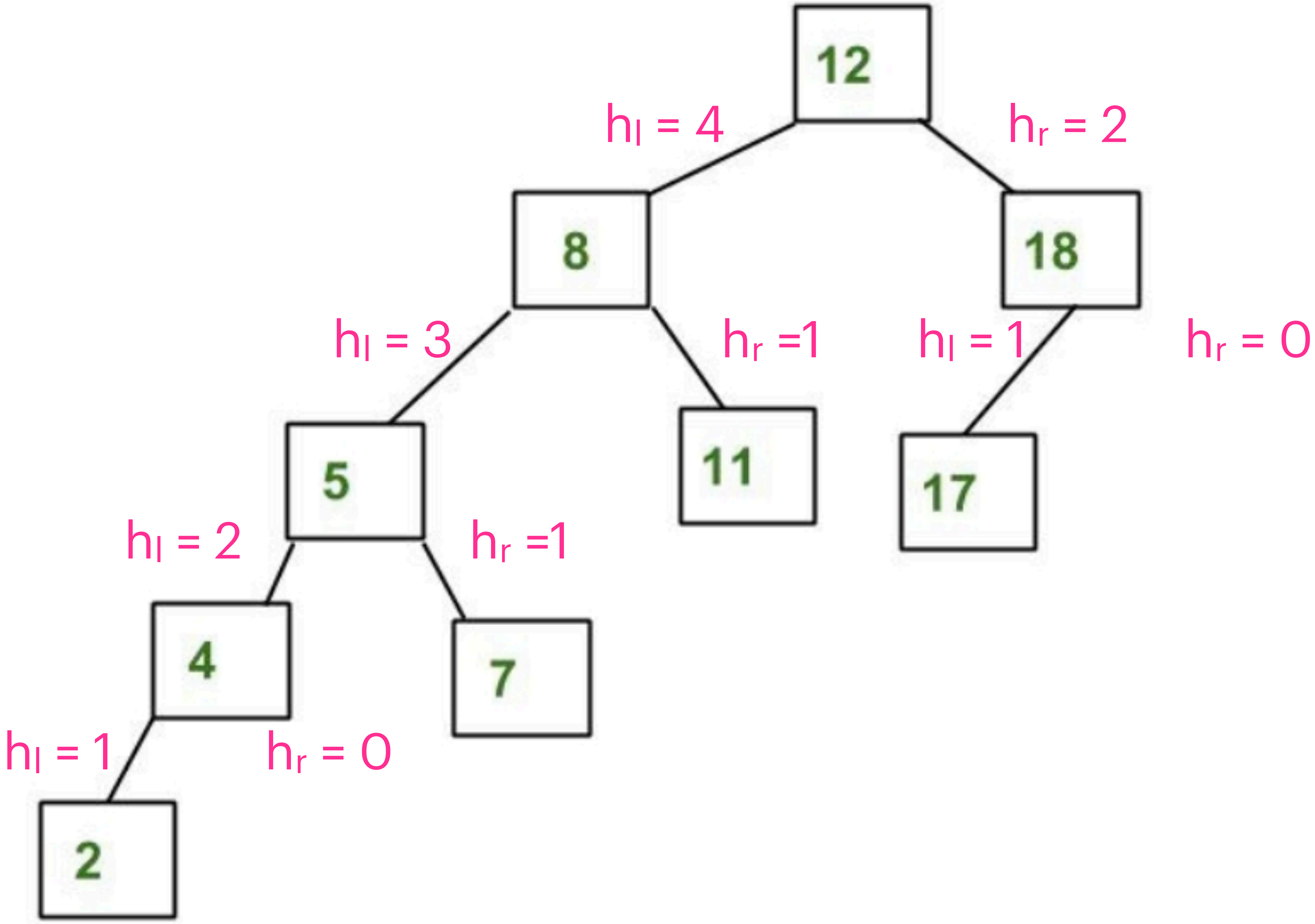




AVL-Tree or not ?

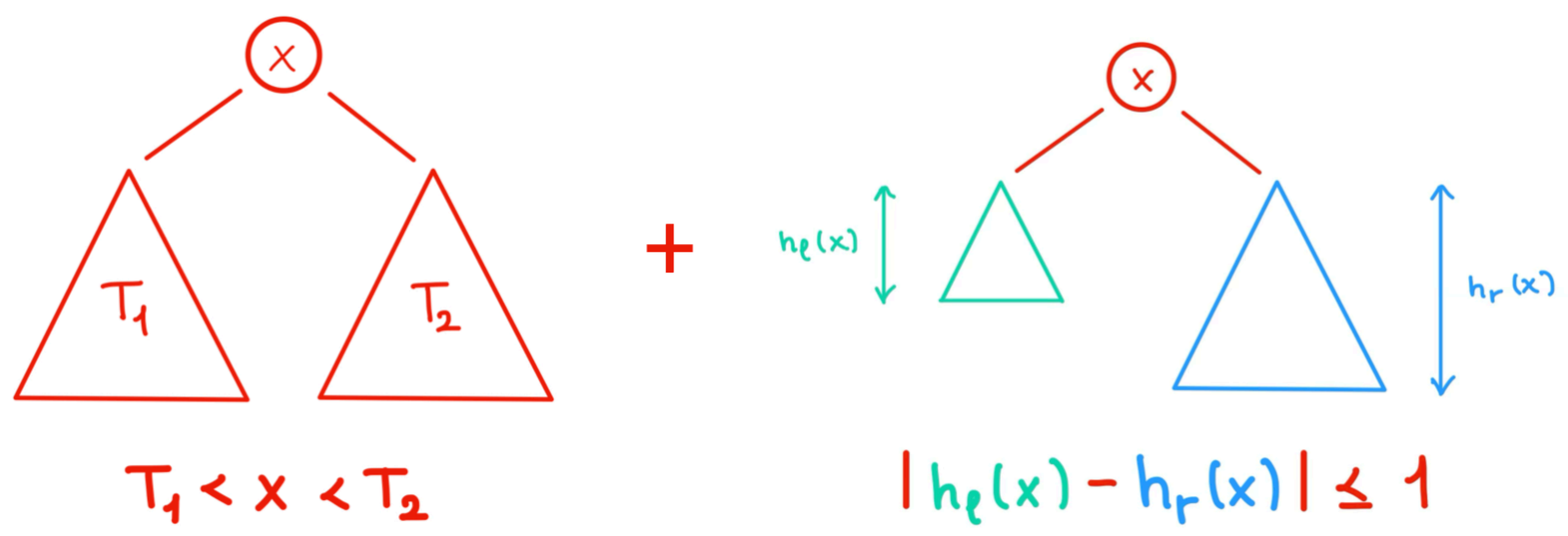
AVL-Tree Condition :

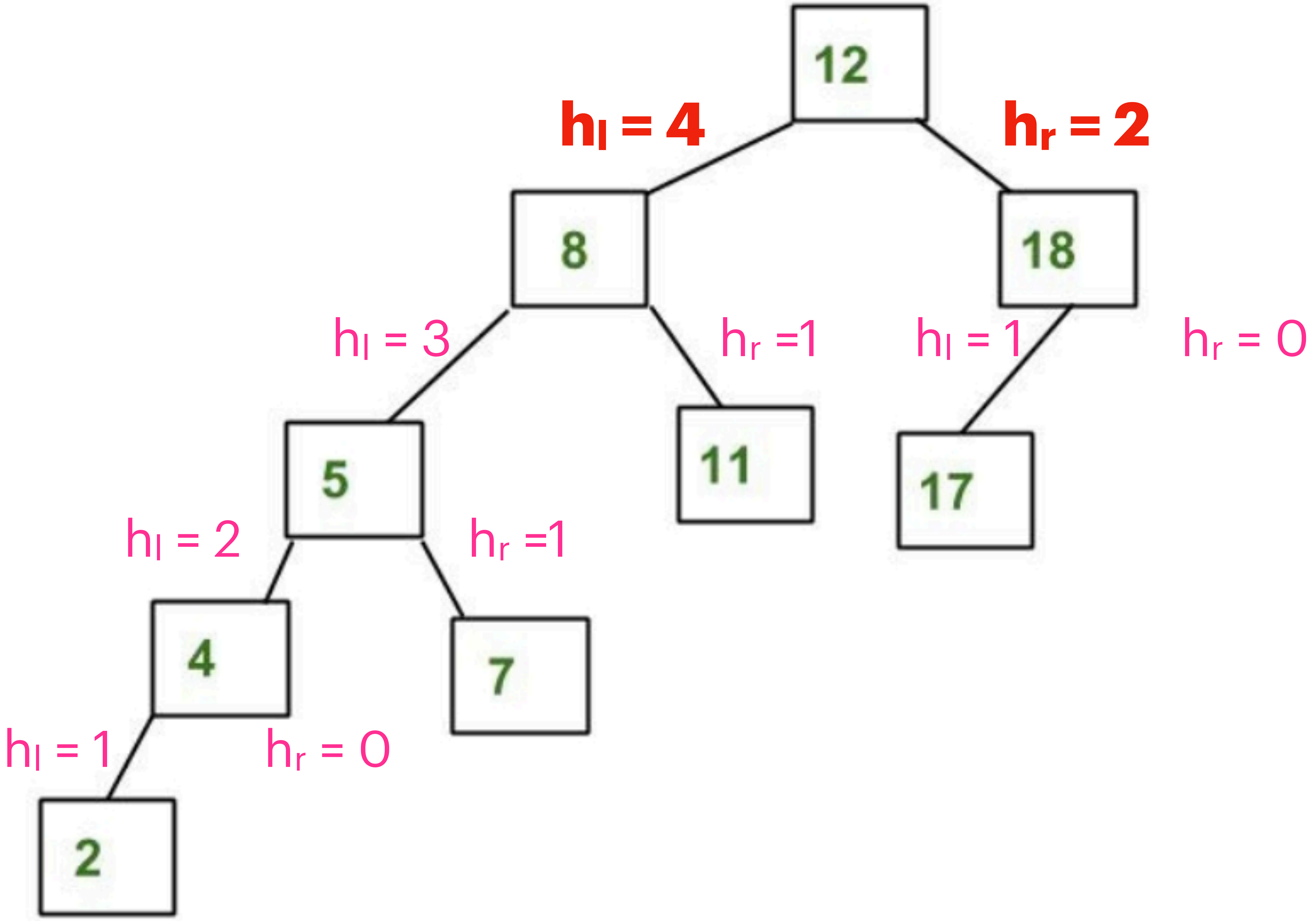




AVL-Tree or not ?

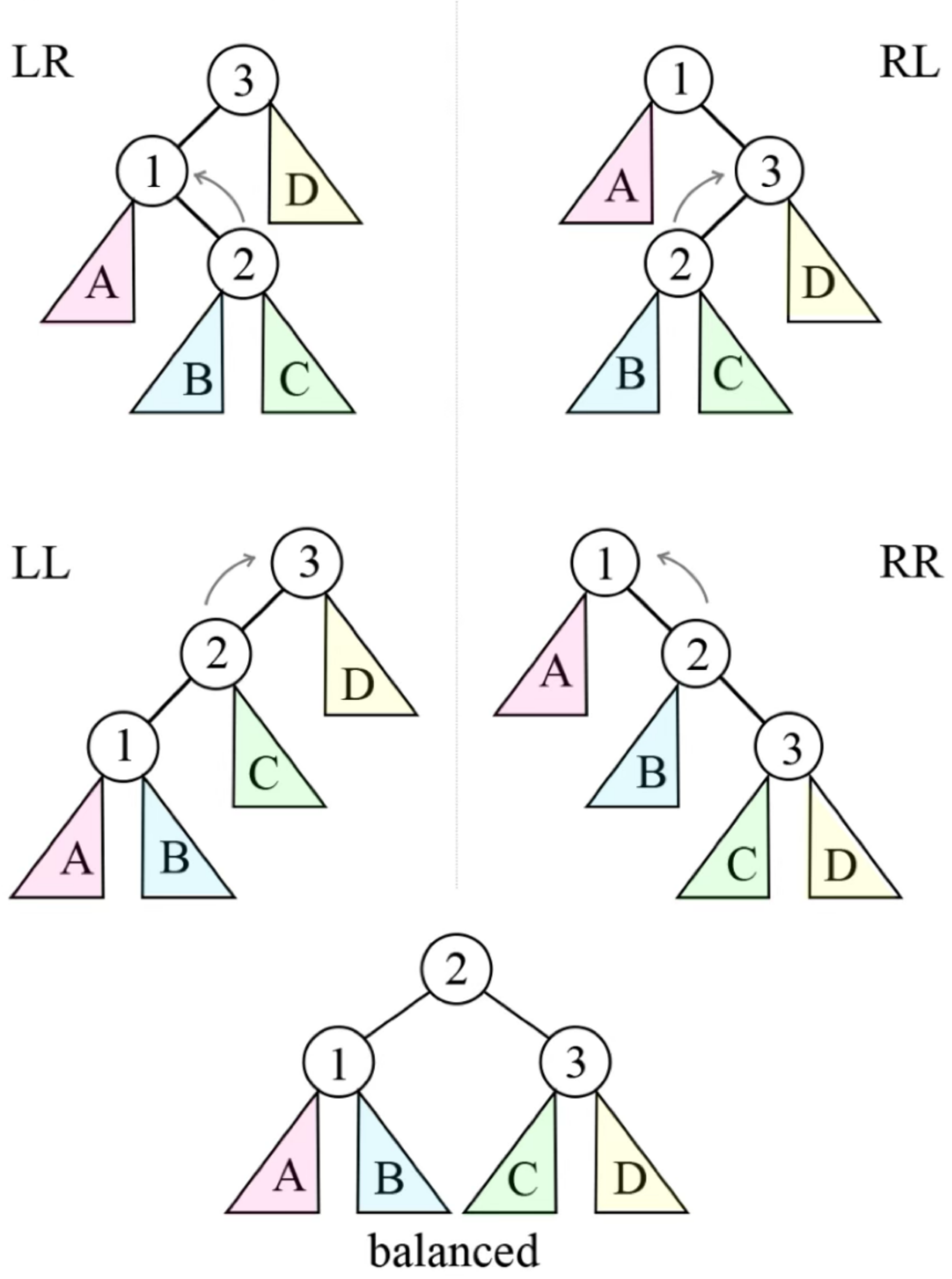
AVL-Tree Condition :



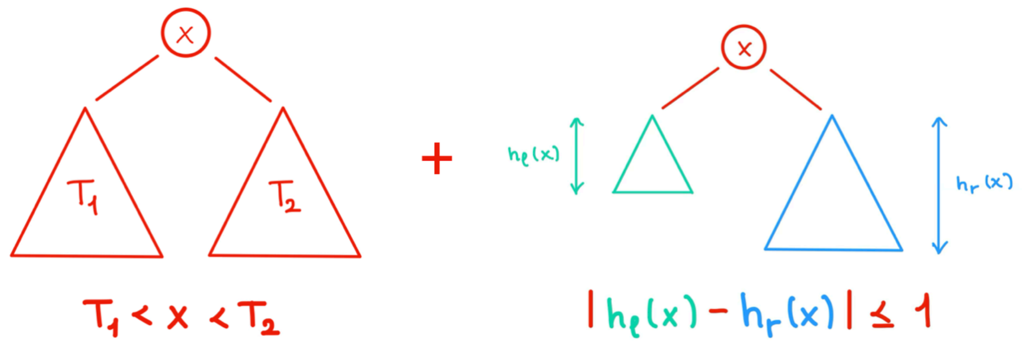


AVL Tree
Rebalancing

Figure 3.2: All possible rotations and their next state



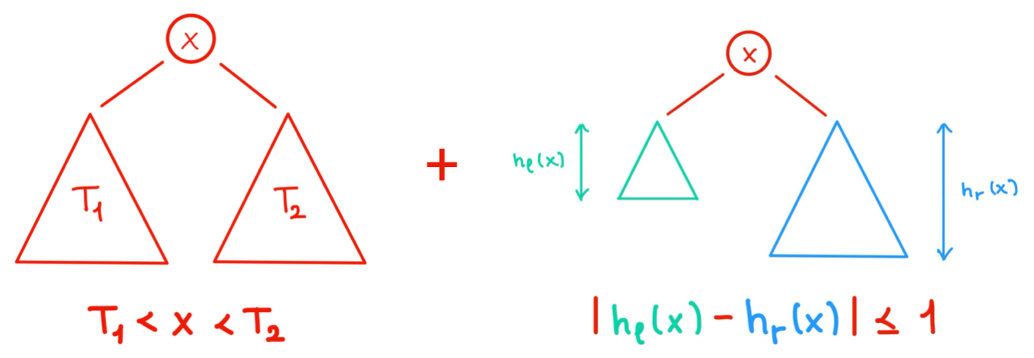
AVL-Tree Condition :

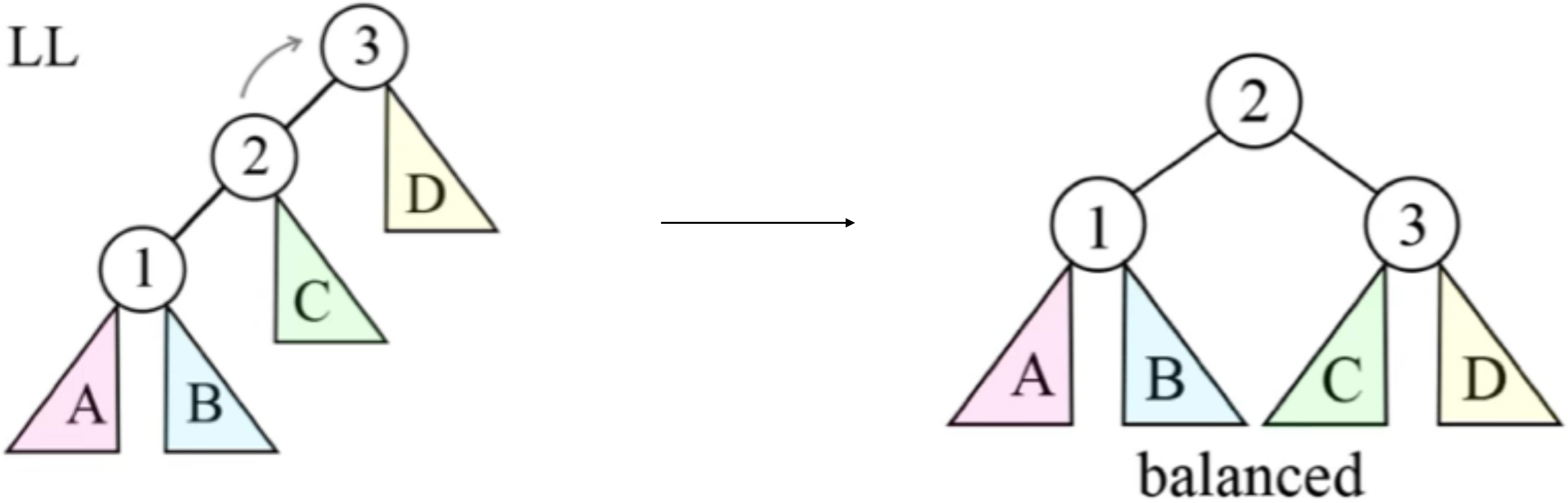


AVL Tree

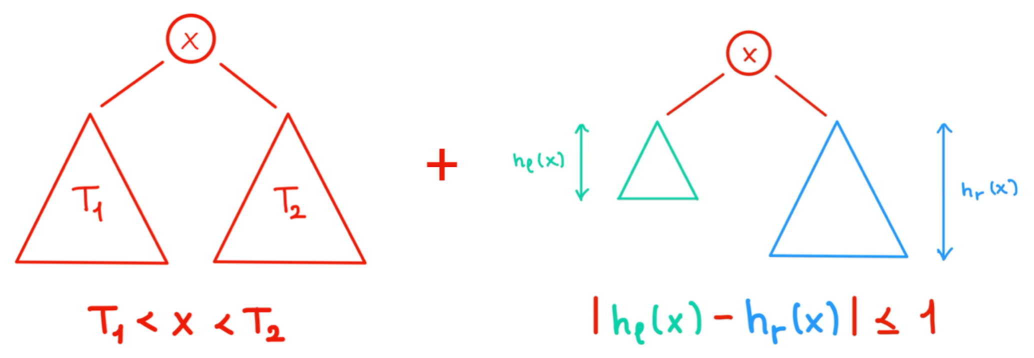
LL - Rotation

AVL-Tree Condition :



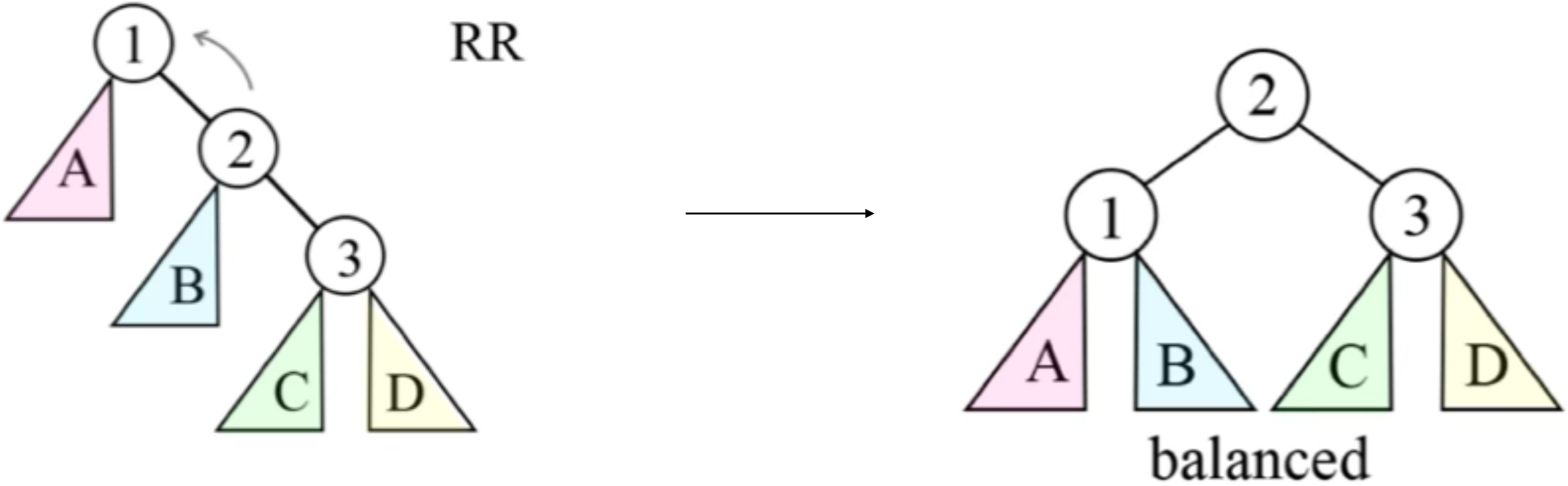


AVL-Tree Condition :



AVL Tree

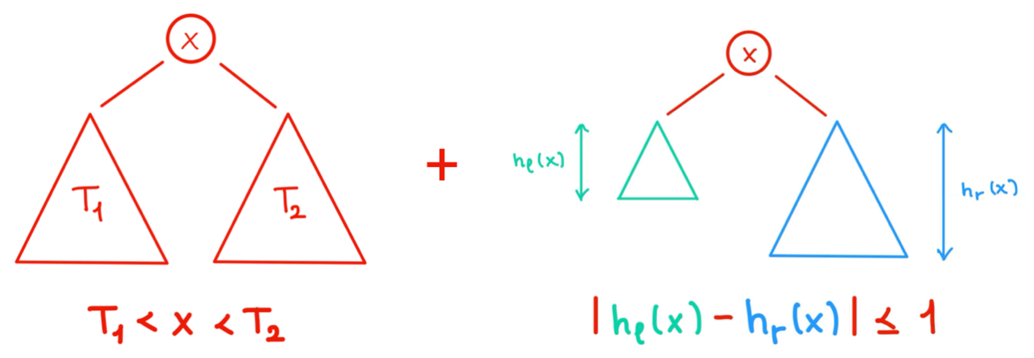
RR - Rotation

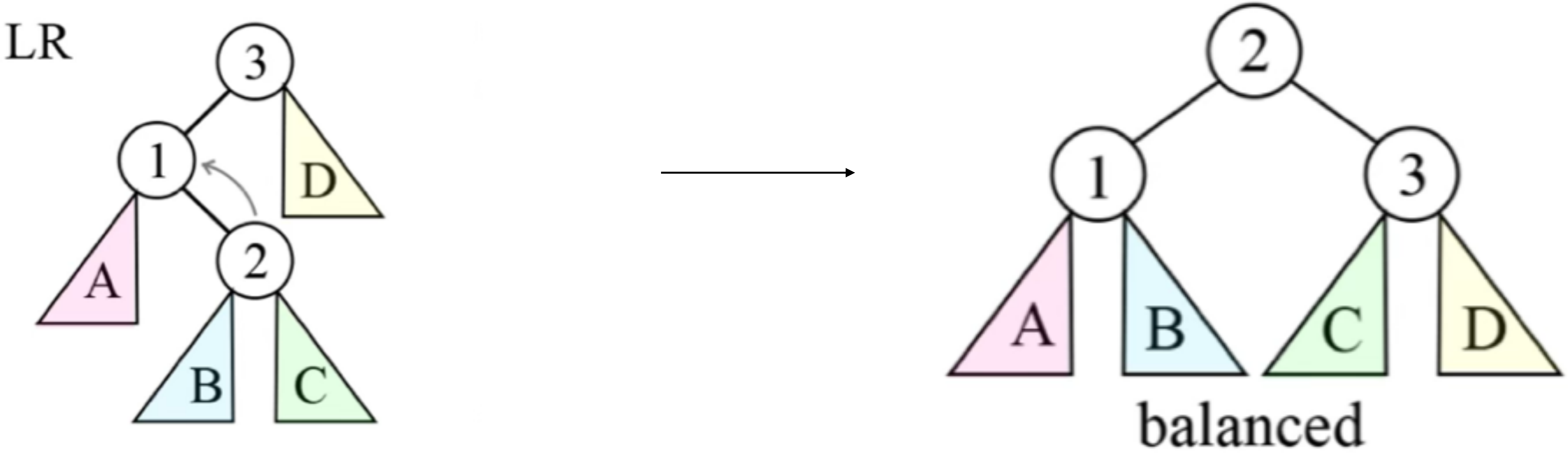


AVL Tree

LR - Rotation

AVL-Tree Condition :

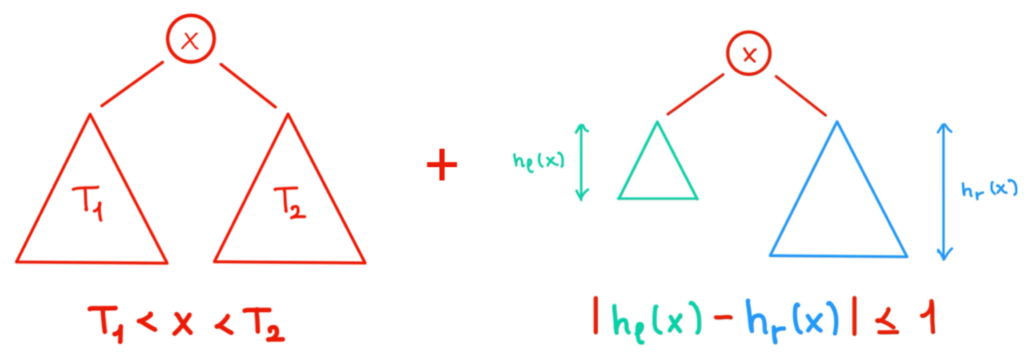


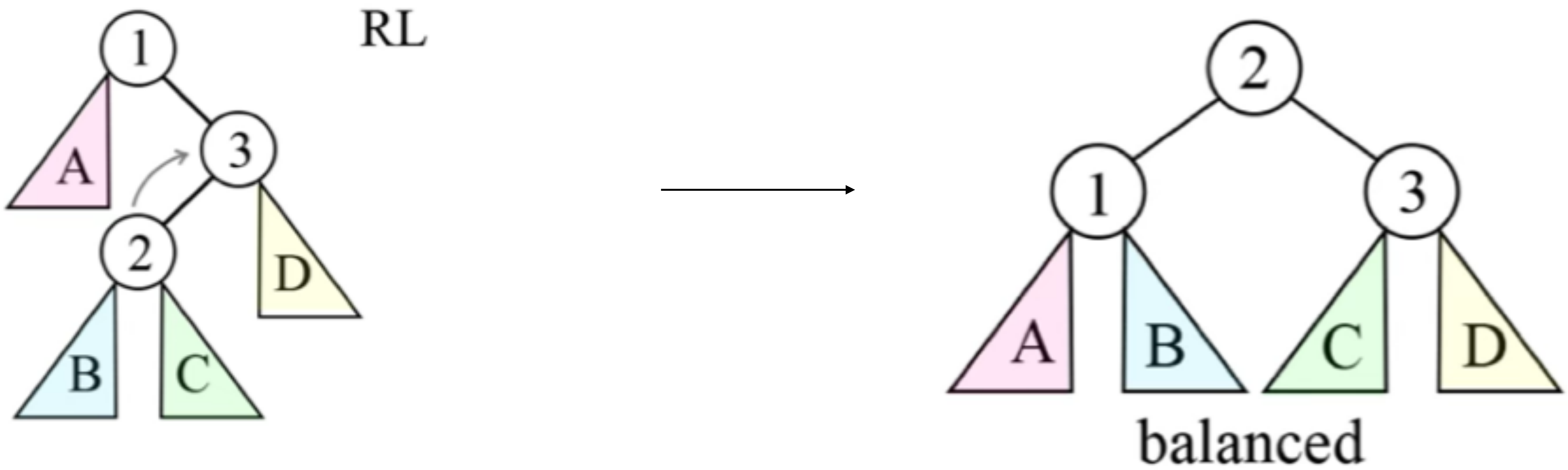


AVL Tree

RL - Rotation

AVL-Tree Condition :



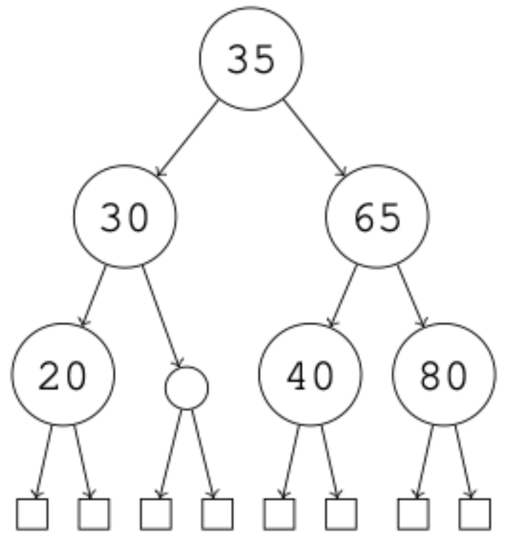


BST and AVL-Tree

Exam Question (FS23)

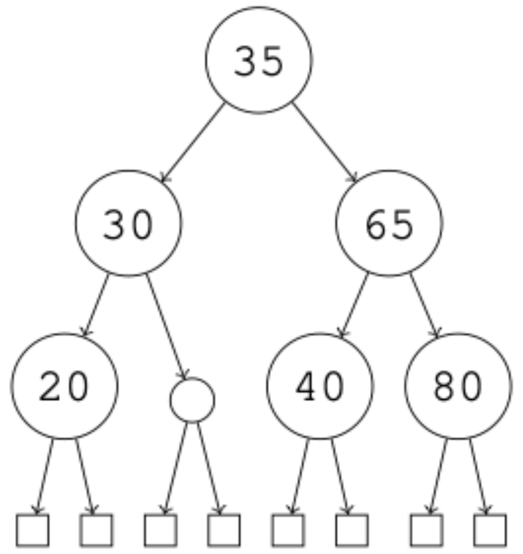
/ **3 P** b) *Search trees:*

i) Draw the **binary search tree** obtained from the following tree by performing the two operations `INSERT`(45) and `DELETE`(30), in that order.



ii) Draw the **AVL tree** that is obtained when inserting the keys $3, 2, 7, 6, 8, 9$ in this order into an empty tree (it suffices to draw only the final tree).

iii) Draw the **AVL tree** that is obtained by deleting key 65 from the tree below.



Trees

Exam Tipps

- Know the tree condition , always keep in mind !

- Know how to insert, know how to delete
- Be able to illustrate an example by hand !

- Don’t mix up the trees !!!

Let’s take a break

DP

DP

How to learn

- Theory, written tasks :
  - Exam questions T3 !!
  - Exercise sheets
  - geeksforgeeks

Always a combination of the ideas dicussed in lecture !

- Coding :
  - CodeEx exercises , my videos
  - Old Exam exercises
  - Leetcode https://leetcode.com/studyplan/dynamic-programming/

Table ? 🙅🏻‍♀️

DP

Written Question Format

a) Provide a *dynamic programming* algorithm that, given an array A of n pairwise distinct positive integers as above, returns `True` if there are two different (non-empty) subsets I_1 and I_2 of $\{1, \ldots, n\}$ (i.e. $\emptyset \neq I_1, I_2 \subseteq \{1, \ldots, n\}$ and $I_1 \neq I_2$) such that $\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i$ and `False` otherwise. In particular, the algorithm does *not* have to return the sets I_1 and I_2.

For example,

- For the array $[2, 3, 4, 5, 7]$ the output should be `True` since for $I_1 = \{1, 2, 4\}$ and $I_2 = \{2, 5\}$ we have $\sum_{i \in I_1} a_i = 2 + 3 + 5 = 10 = 3 + 7 = \sum_{i \in I_2} a_i$.
- For the array $[2, 3, 4, 10, 20]$ the output should be `False` since there are no two different non-empty subsets I_1 and I_2 of $\{1, 2, 3, 4, 5\}$ that satisfy $\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i$.

In order to obtain full points, your algorithm should run in time $O(n \cdot S)$, where $S = \sum_{i=1}^{n} a_i$. In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

DP
Introduction Exercise

Consider the recurrence

$$
\begin{aligned}
A_1 &= 1 \\
A_2 &= 2 \\
A_3 &= 3 \\
A_4 &= 4 \\
A_n &= A_{n-1} + A_{n-3} + 2A_{n-4} \text{ for } n \geq 5.
\end{aligned}
$$

Compute A_n using bottom-up dynamic programming and state the run time of your algorithm. Address the following aspects in your solution:

(1) *Definition of the DP table:* What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry?

(2) *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.

(3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?

(4) *Extracting the solution*: How can the final solution be extracted once the table has been filled?

(5) *Run time*: What is the run time of your solution?

DP

How to learn

- Theory, written tasks :
  - Exam questions T3 !!
  - Exercise sheets
  - geeksforgeeks

Always a combination of the ideas dicussed in lecture !

- Coding :
  - CodeEx exercises , my videos
  - Old Exam exercises
  - Leetcode https://leetcode.com/studyplan/dynamic-programming/

Table ? 🙅🏻‍♀️

Maximum Subarray Sum

Problem : find the subarray that has the maximum sum

Examples :	Inputs :	Outputs :	Subarray :
	[2, 3, -8, 7, -1, 2, 3]	11	[7, -1, 2, 3]
	[-2, -4]	-2	[-2]
	[5, 4, 1, 7, 8]	25	[5, 4, 1, 7, 8]

Maximum Subarray Sum

Problem : find the subarray that has the maximum sum

Idea : Randmax $R_j = \max_{i \leq j} S_{ij}$ where $S_{ij} = a_i + a_{i+1} + \ldots + a_j$

Definition of the DP table : $DP[i] = R_i$

Computation of an entry :

Initialization : $DP[0] = A[0]$

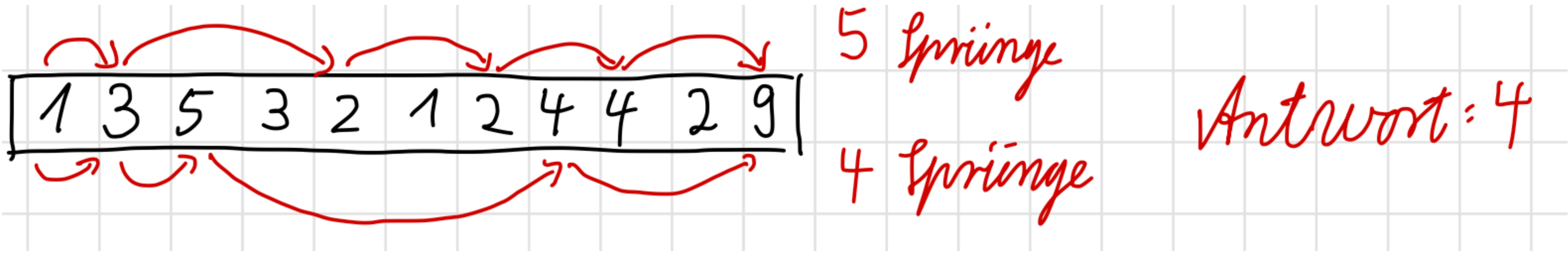
Recursion : $DP[i] = \max\{ \underbrace{a_i}_{\text{New subarray starting with } a_i} , \underbrace{R_{i-1} + a_i}_{\text{Adding } a_i \text{ to the current subarray}} \}$

Extracting the solution : The solution is max{ DP[i], 0 }

Jump Game

Problem : Given an array where each element represents the max number of steps that can be made forward from that index,find the minimum number of jumps to reach the end of the array starting from index 0.

Example:



Jump Game

Problem : Given an array where each element represents the max number of steps that can be made forward from that index,find the minimum number of jumps to reach the end of the array starting from index 0.

Definition of the DP table : DP[i] = “Minimum number of jumps to reach i”

Computation of an entry :

Initialization : $DP[0] = 0$

Recursion : $DP[i] = \min \{ 1 + DP[j] \mid 1 \leq j < i \wedge j + A[j] \geq i \}$

Extracting the solution : The solution is at DP[n-1]

Longest Common Subsequence

Problem : Given two strings, A and B, the task is to find the length of the Longest Common Subsequence

A subsequence is a string generated from the original string by deleting 0 or more characters and without changing the relative order of the remaining characters.

Examples :	Inputs :	Outputs :	Subsequence :
	“ABC" and “ACD”	2	“AC”
	‘AGGTAB” and “GXTXAYB”	4	“GTAB”
	“ABC” and “CBA”	1	“A” , “B” or “C”

Longest Common Subsequence

A subsequence is a string generated from the original string by deleting 0 or more characters and without changing the relative order of the remaining characters.

💡 Idea : For every pair A[i] and B[j] there are exactly 3 options

- Use them in the subsequence
- Don't use A[i]
- Don't use B[i]

Teilfolge : - Reihenfolge muss stimmen
- Lücken ok

Alignment:

```
T I _ G E R
Z I E G E _
```

DP[0...n][0...m]

Definition of the DP table : DP[i][j] = LCS of A[0..i] and B[0..j]

Computation of an entry :

Initialization : DP[0][j] = 0 DP[i][0] = 0

Recursion :

$$DP[i][j] = \begin{cases} \max\{\underbrace{DP[i-1][j-1]+1}_{\text{use them in the subsequence}},\ DP[i-1][j],\ DP[i][j-1]\} & \text{if } A[i] == B[j] \\ \max\{\underbrace{DP[i-1][j]}_{\text{don't use A[i]}},\ \underbrace{DP[i][j-1]}_{\text{don't use B[j]}}\} & \text{else} \end{cases}$$

Extracting the solution : The solution is at DP[n][m]

Edit Distance

Problem : Given two strings A and B, find the minimum number of edits (operations) to convert A into B

Operations :

Replace : Replace a character at any index of A with some other character

Insert : Insert any character after or before any index of A

Remove : Remove a character of A

Examples :	Inputs :	Outputs :	Operations :
	“cat" and “cut”	1	replace a with u
	“sunday” and “saturday”	3	convert un to atur : replace n by r insert a, insert t

Operations: Replace : Replace a character at any index of A with some other character

Insert : Insert any character after or before any index of A

Remove : Remove a character of A

Edit Distance

Idea : For every element of A , 3 things can happen

- will be deleted
- something gets inserted afterwards
- will be replaced to match B[j]

Definition of the DP table : DP[i][j] = ED of A[0..i] and B[0..j] DP[0...n][0...m]

Computation of an entry :

Initialization : DP[i][0] = i DP[0][j] = j

Recursion :

add B[j] to the end

$$DP[i][j] = \begin{cases} \min \{ DP[\,i-1\,][\,j\,] + 1 \,,\; DP[\,i\,][\,j-1\,] + 1 \,,\; DP[\,i-1\,][\,j-1\,] \} & \text{if } A[i] == B[j] \\ \min \{ DP[\,i-1\,][\,j\,] + 1 \,,\; DP[\,i\,][\,j-1\,] + 1 \,,\; DP[\,i-1\,][\,j-1\,] + 1 \} & \text{else} \end{cases}$$

delete A[i] replace A[i] with B[j]

Extracting the solution : The solution is at DP[n][m]

CodeExpert

Edit Distance to Subsequence

Next Week

Questions

Feedbacks , Recommendations

Nil Ozer