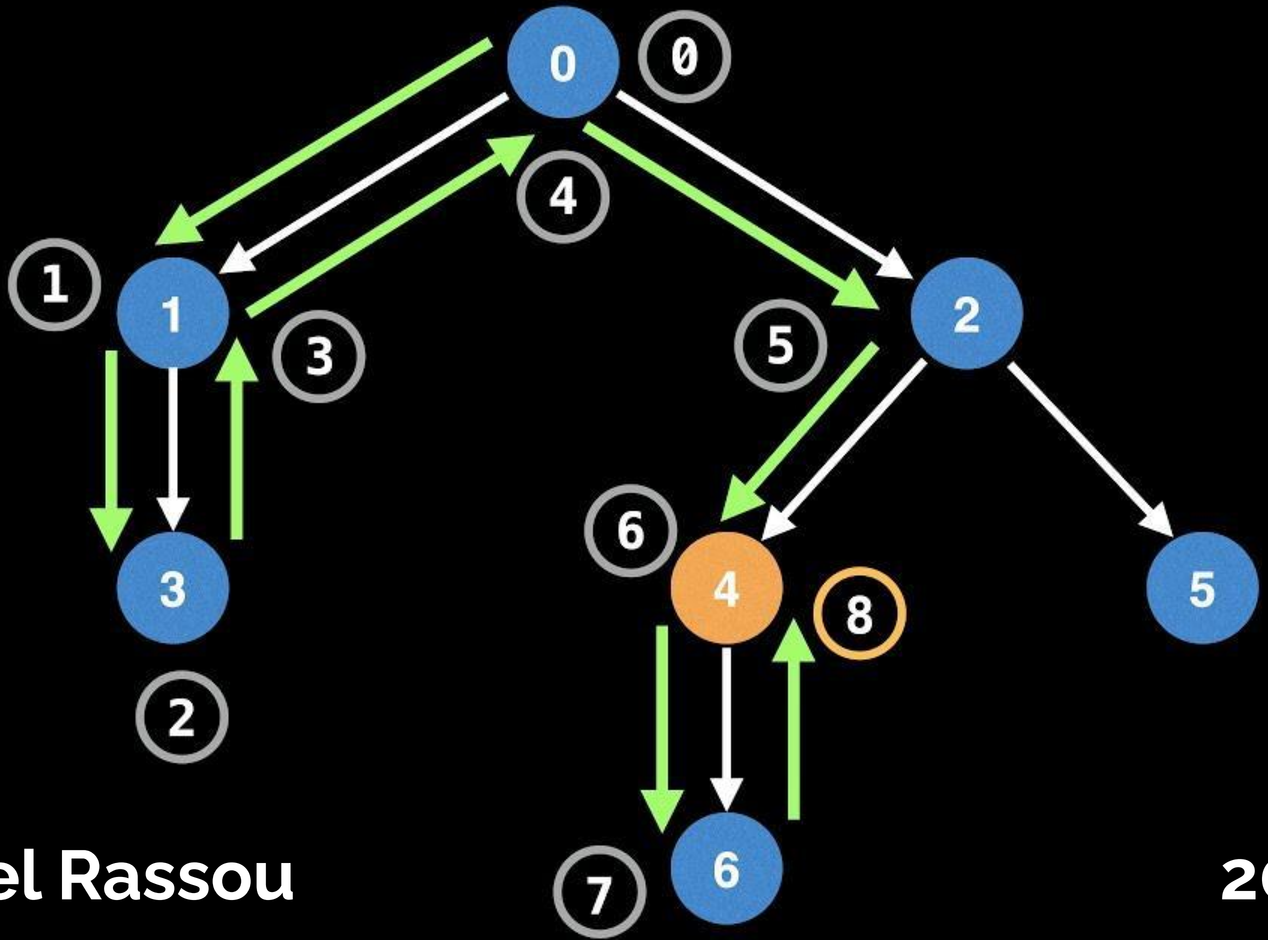
Lowest Common Ancestor

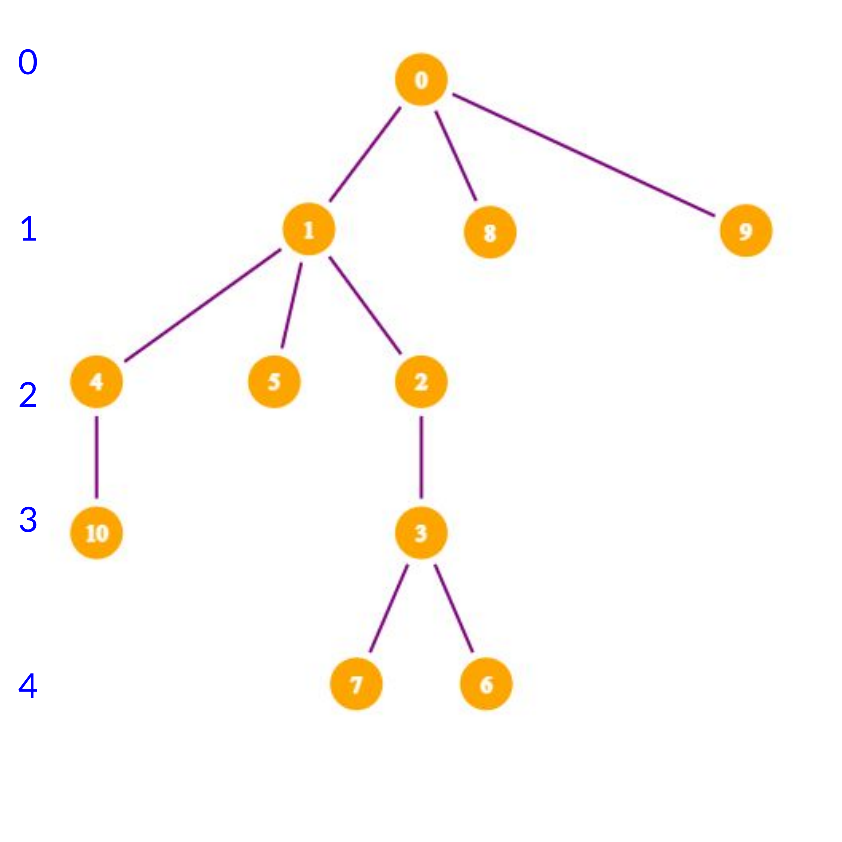


Emmanuel Rassou

2023

Before we embark on a new adventure...

What is the lowest common ancestor?

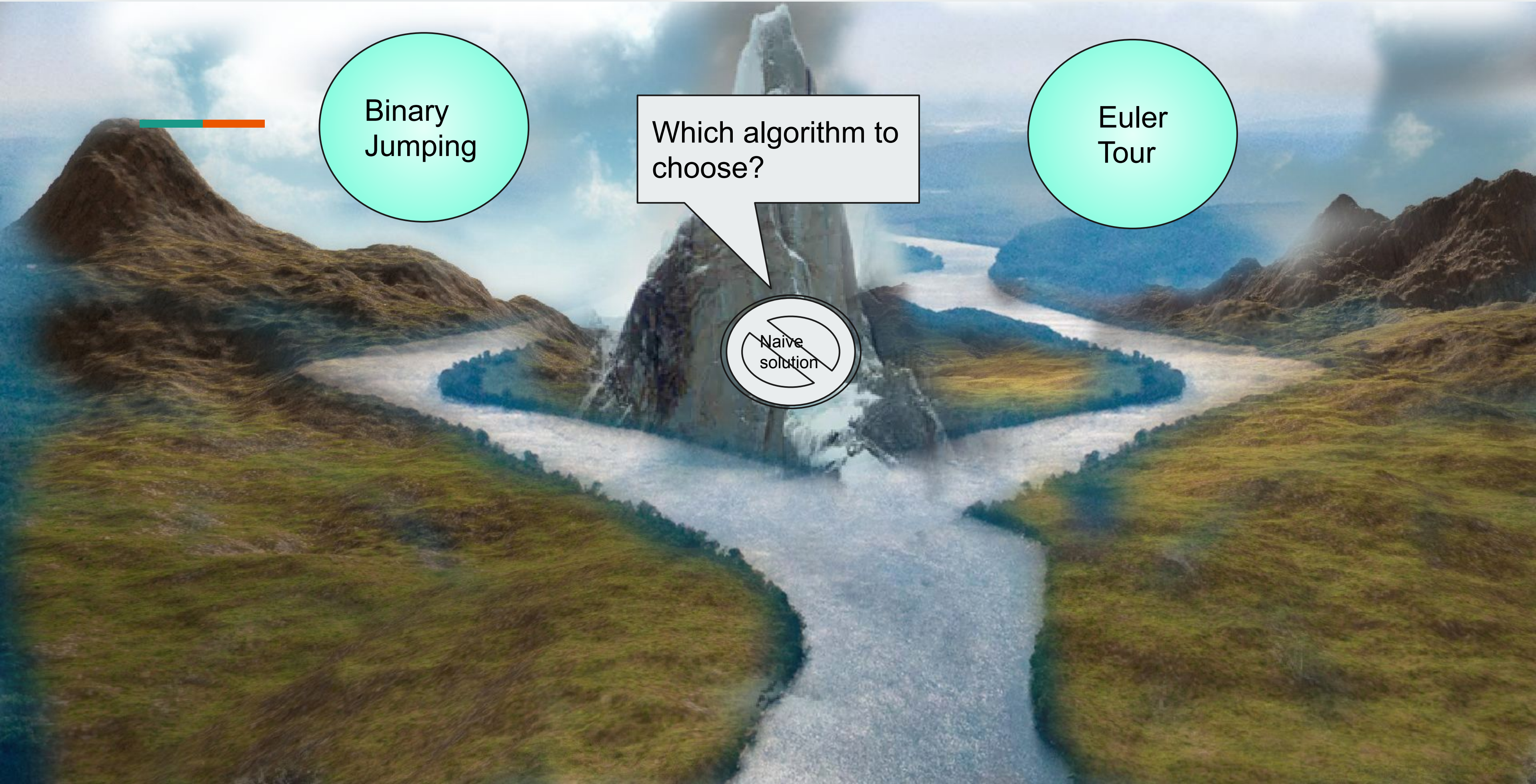


Given two nodes on a **TREE** (rooted at node 0) ,their LCA is the node that is a *parent* (ancestor) of each node AND has the maximal depth.

It is the 'best' meeting point between the two nodes, if you can only travel up the tree.

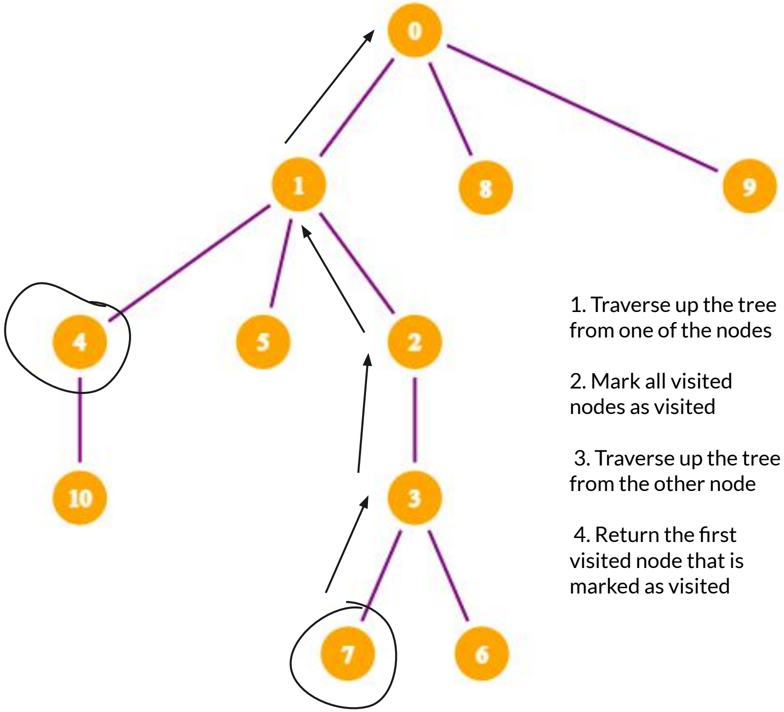
What is the LCA of node 4 and 6?

Solve queries: find LCA of nodes a & b quickly


Binary Jumping
Which algorithm to choose?
Naive solution
Euler Tour

NAIVE SOLUTION

O(N) time per query

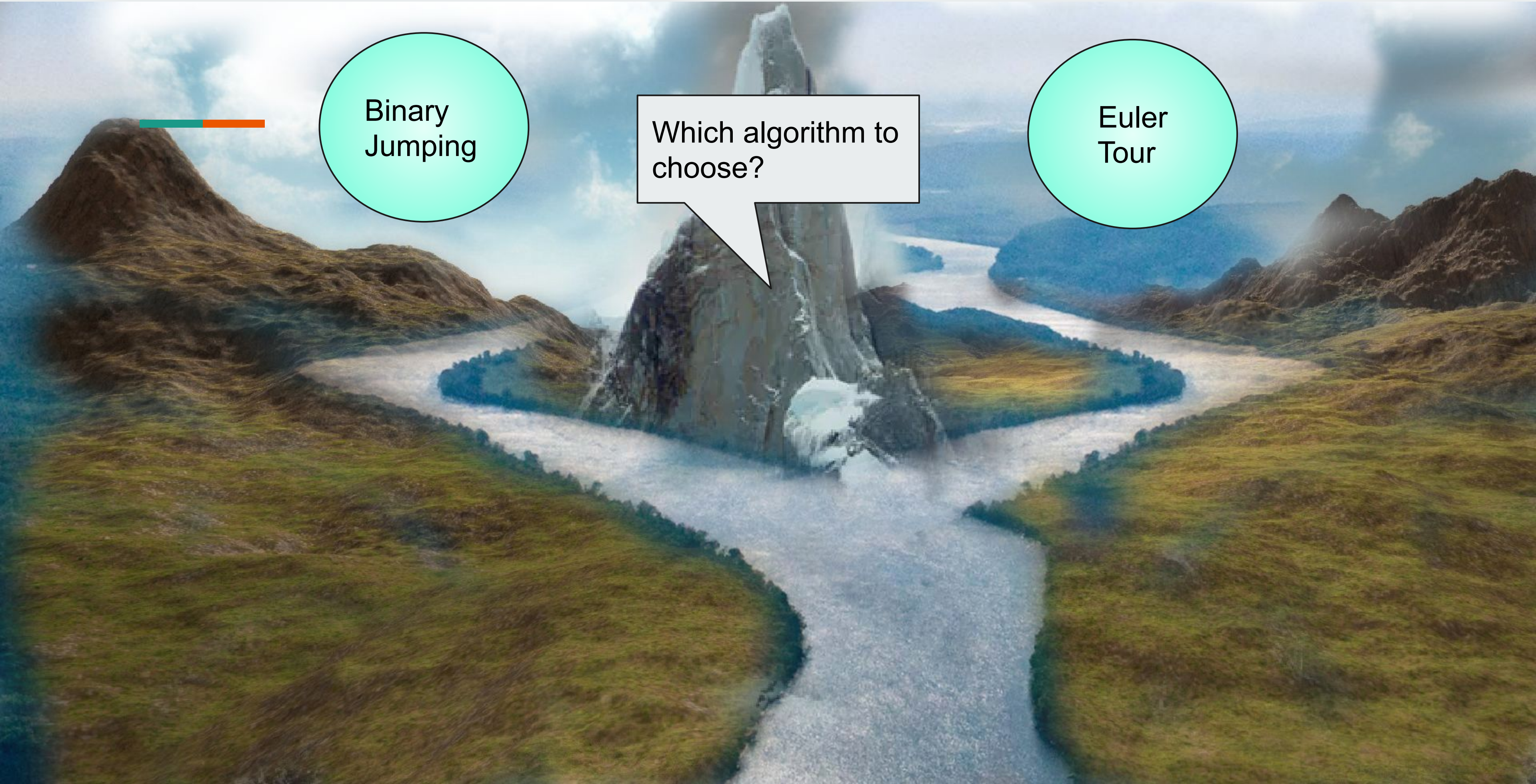


1. Traverse up the tree from one of the nodes

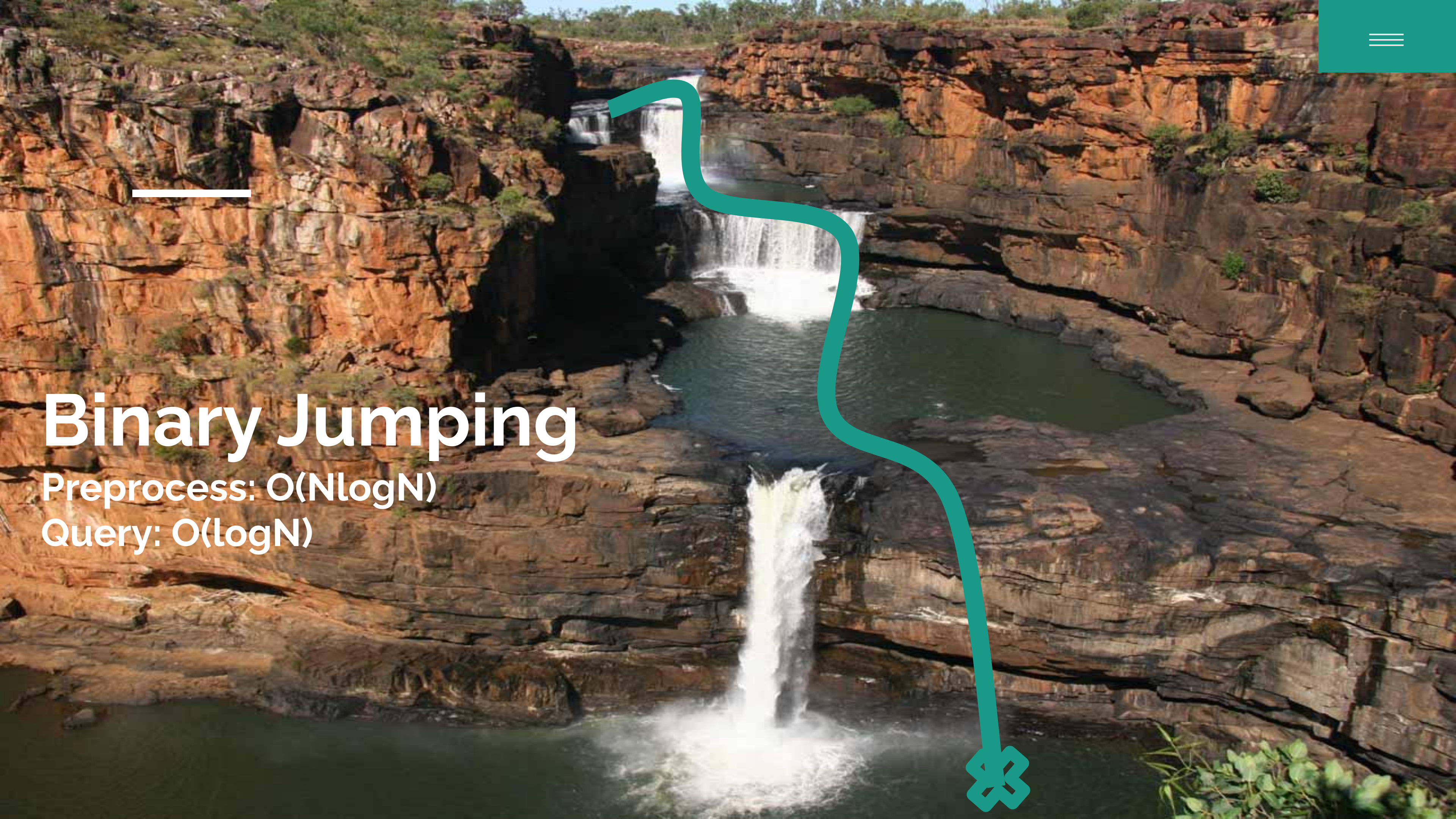
2. Mark all visited nodes as visited

3. Traverse up the tree from the other node

4. Return the first visited node that is marked as visited



Binary Jumping
Which algorithm to choose?
Euler Tour

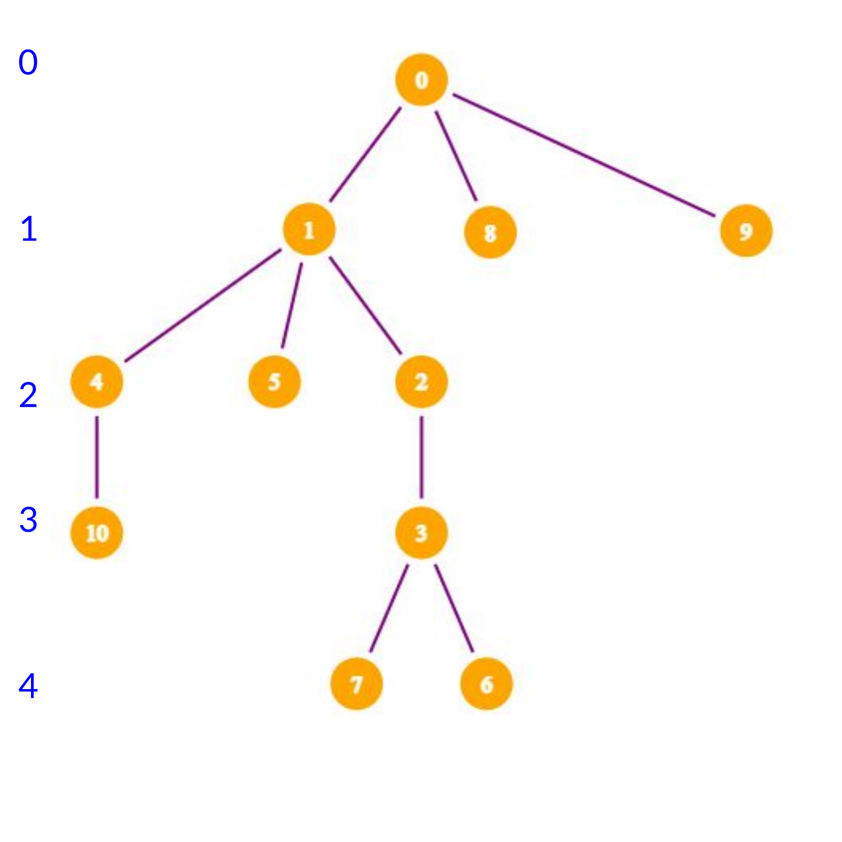


Binary Jumping

Preprocess: O(NlogN)

Query: O(logN)

Algorithm using Sparse table and Binary jumping



1) Preprocessing using sparse table:
   Calculate for each node the 2^k th parent

```
for(I = 1; I <= N; I++)
   for(J = 1; (1 << J) < N; J++)
    if(sparse_table[I][J-1] !=-1)
        sparse_table[I][J] =
sparse_table[sparse_table[I][J-1]][J-1];
```

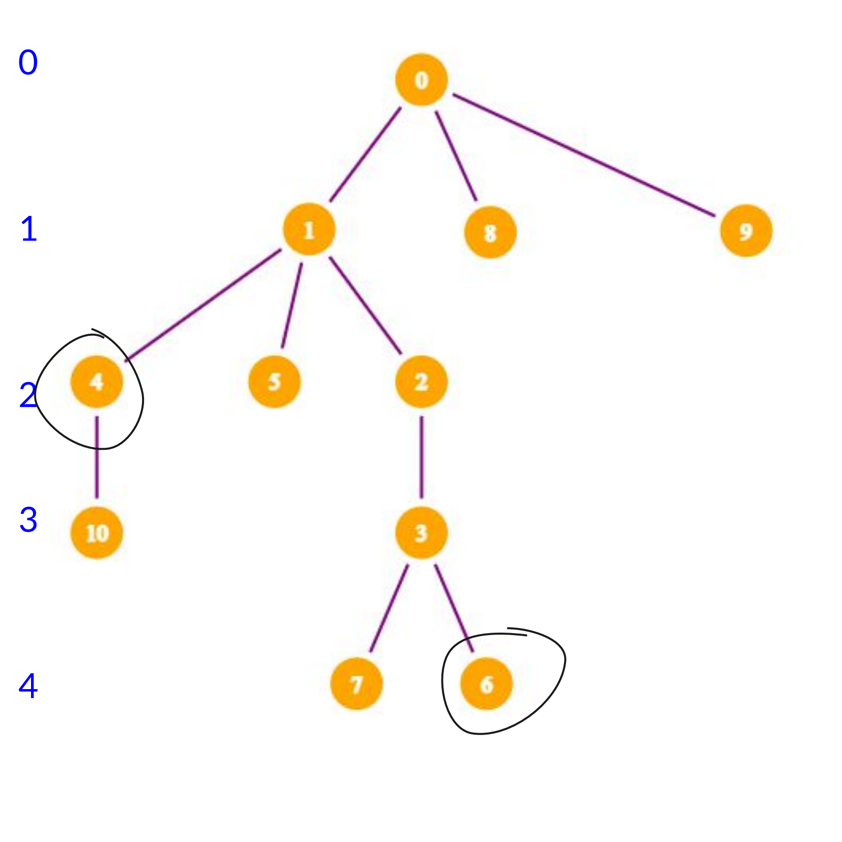
NOTE* Set sparse_table[i][j] = -1 for out of bounds jumps.

Example-if node *u* has to climb 37 edges to reach the LCA:
$37 = 32 + 4 + 1 = 100101_2$

Then we make a jump of 32. Set this to node u
Make a jump of 4. Set this to node u.
Make a jump of 1. Set this to node u.

1^+) Run a simple DFS to get depth of each node

Algorithm using Sparse table and Binary jumping



2) if depth a ≠ depth b:
 Move the deeper one up until depth a = depth b
 Use binary jump for logN time

3) Implementation detail:
 After 2) if a == b then we have found our LCA
 find a number **log** such that `2^(log+1) > depth[a]`

4)
```
for(I = log; I>=0; I--)
   if(sparse_table[a][I] !=-1 && sparse_table[a][I]
!= sparse_table[b][I])
       a = sparse_table[a][I];
       b = sparse_table[b][I];
```

Intuitively :

- Start with the biggest jumps 2^I and make the jumps smaller and smaller
- If we overshoot our LCA, just decrease our jump to 2^{I-1}
- Otherwise the 2^I th ancestor of a and b are NOT equal
 So we set a and b to be their respective ancestors
 Also decrease our jump to size 2^{I-1}

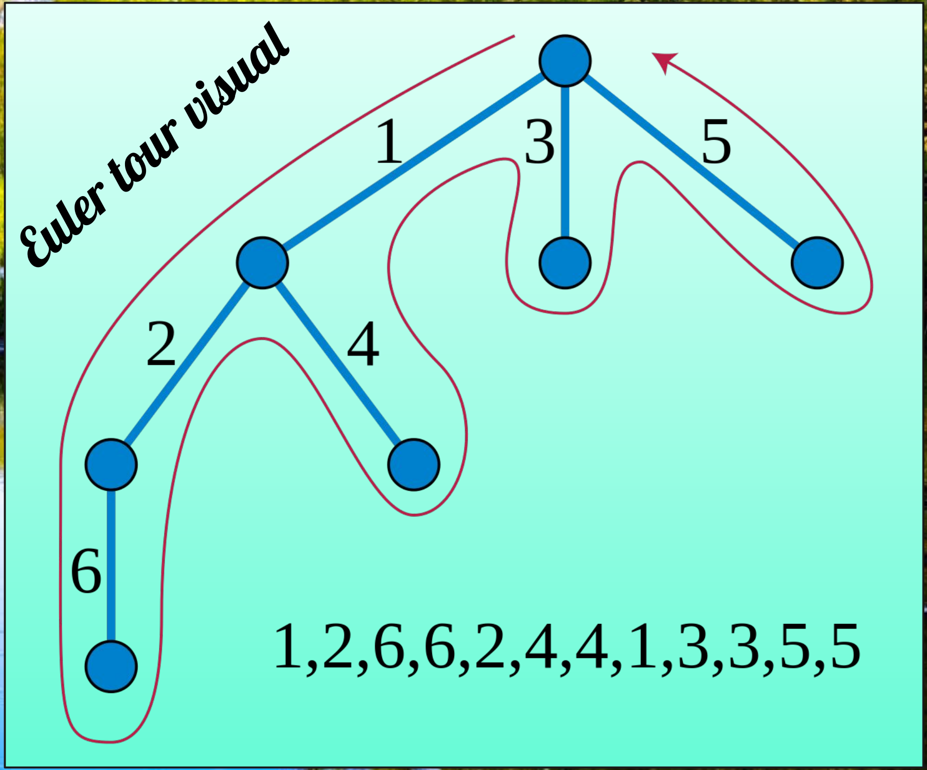
Our jump will eventually reach size 2^0 = 1 in which case we find our LCA

Euler Tour

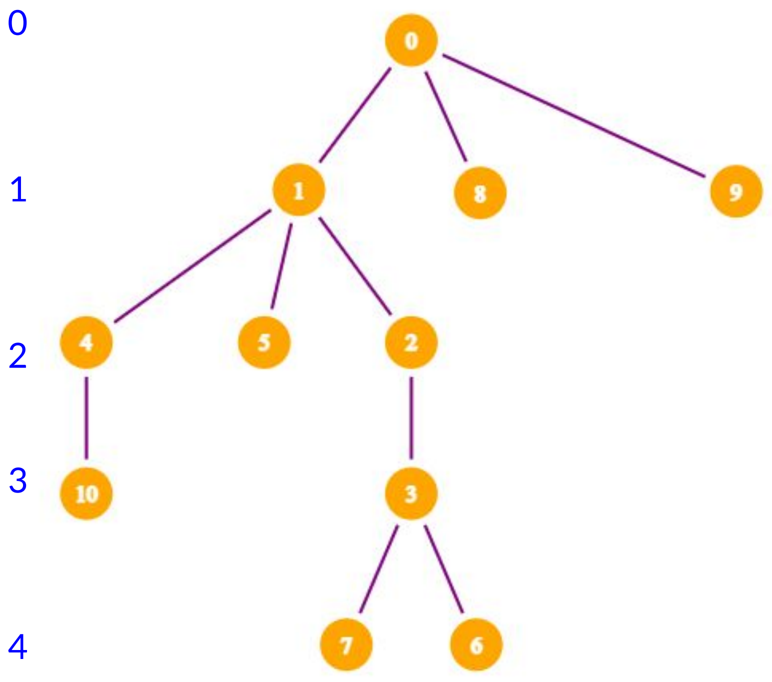
Finally … We have found peace

Euler tour technique

Preprocess: O(NlogN)
Query: O(logN)



Algorithm using Euler tour array and RMQ



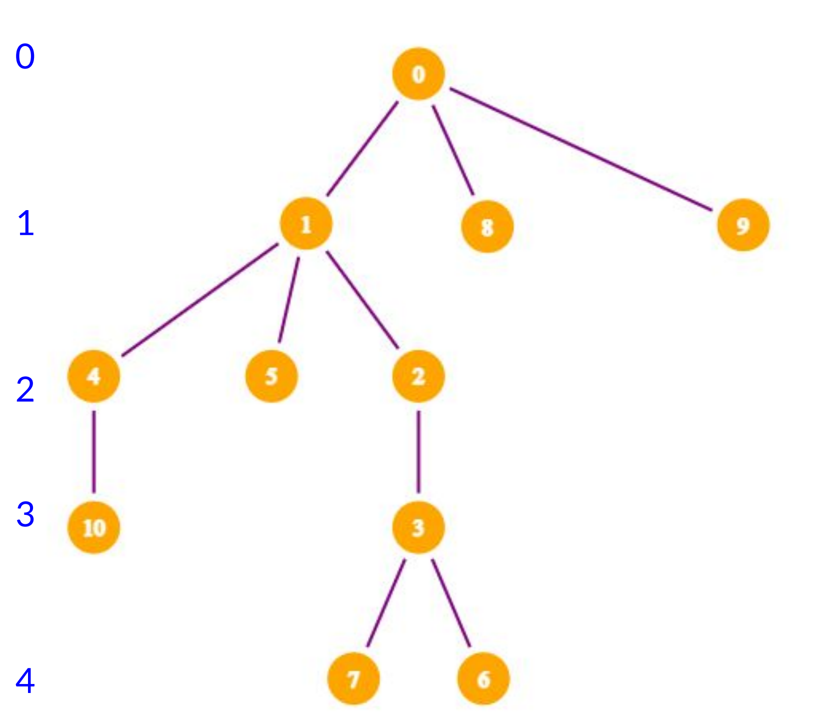
0	1	4	10	4	1	5	1	2	3	7	3	6	3	2	1	0	8	0	9	0

1) Run DFS on the tree :
Add a node to the array *euler* whenever the node is called (time in) and whenever the dfs backtracks from one of its children.
Pseudocode:

Dfs (current_node, parent_node):
  euler.push(current_node)
  For each child of current_node:
    Dfs(child , current_node)
    euler.push(current_node)

To note : each node is added to the euler **c + 1** times, where c is the number of children of the node. Therefore, **euler.size() = 2n - 1**

Algorithm using Euler tour array and RMQ



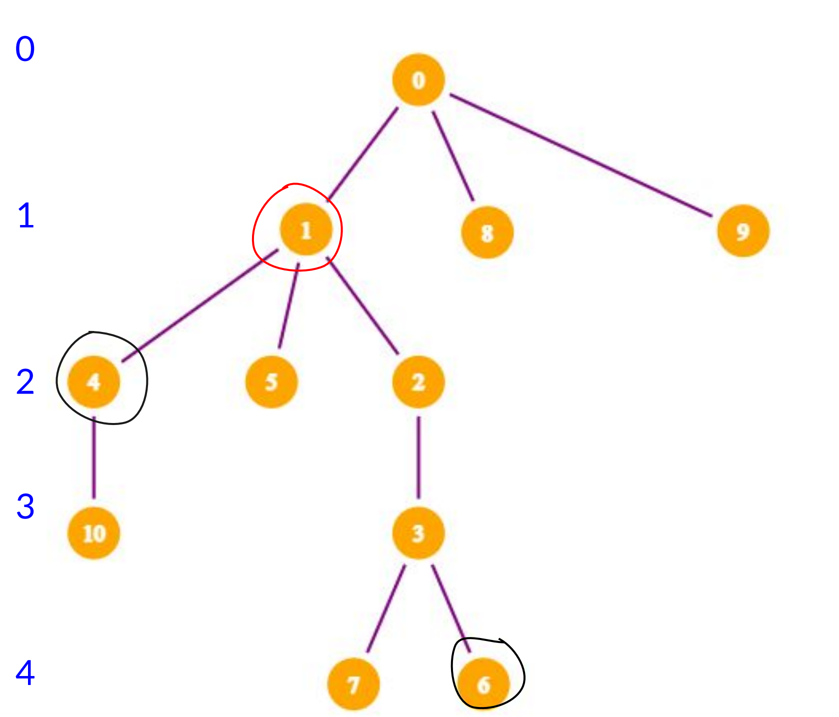
We also want to keep track of at least one occurrence for each node in the euler array. For simplicity, we record the first time for each node in the *tin* array

Pseudocode:

```
timer =0
Dfs (current_node, parent_node):
        euler.push( current_node)
        tin[current_node] = timer
        timer ++
        For each child of current_node:
                Dfs(child , current_node)
                euler.push(current_node)
                timer ++
```

0	1	4	10	4	1	5	1	2	3	7	3	6	3	2	1	0	8	0	9	0

Algorithm using Euler tour array and RMQ



0	1	4	10	4	1	5	1	2	3	7	3	6	3	2	1	0	8	0	9	0

2) Find LCA of node a and b:

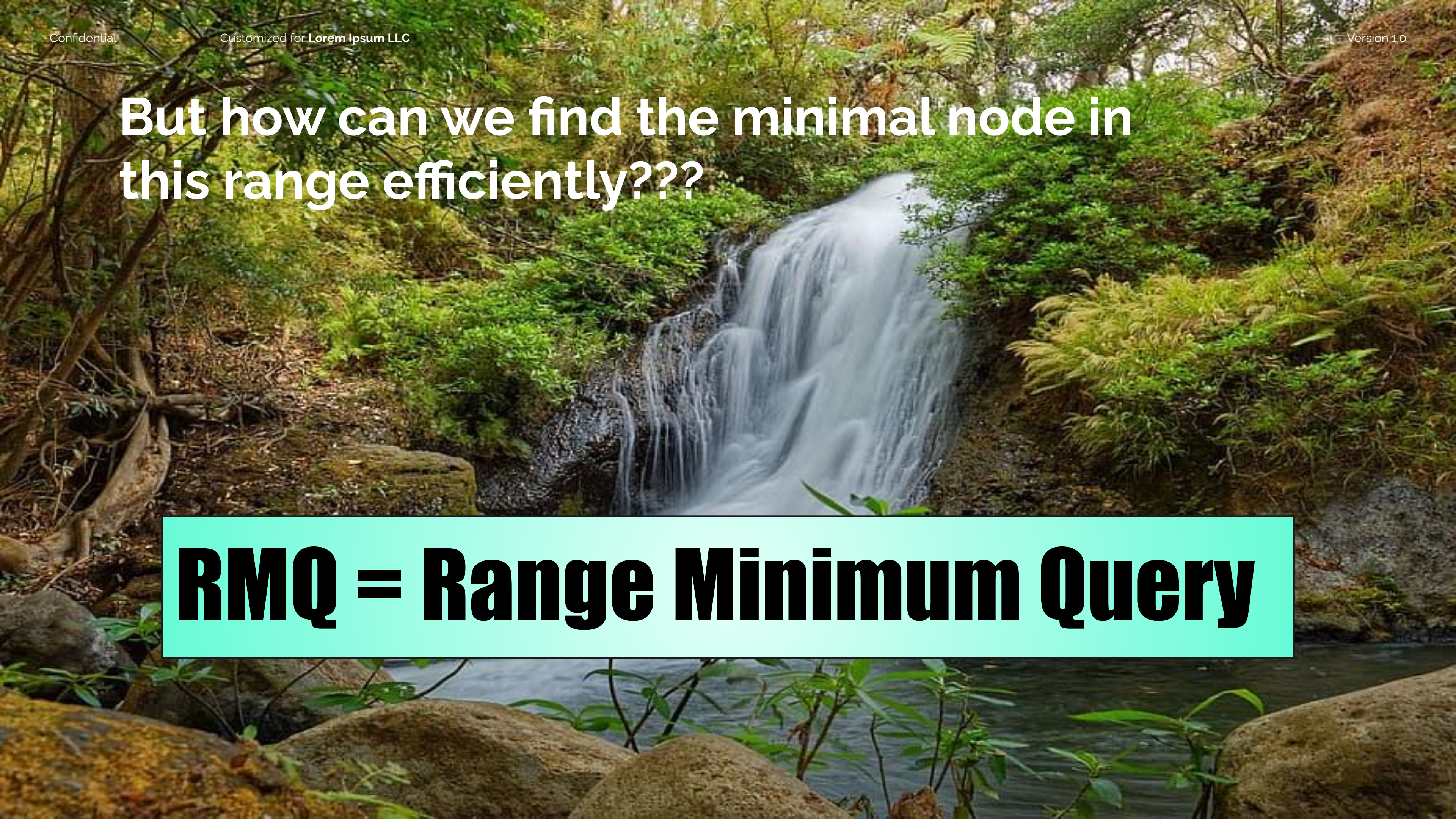
- ★ Pick any node a in the euler array (call its index i_a)
- ★ Pick any node b in the euler array (call its index i_b)
- ★ In the range [i_a, i_b] the LCA is the node with minimal depth (closest to the root)

Concrete example:

Find LCA node 4 and node 6
We can use our *tin* array:
tin[4] = 2, tin[6] = 12
From all the nodes between indices 2 and 12 the one closest to the root is node 1.

(Implementation detail : instead of looking at **depths** of nodes we can alternatively check **time in** of each node
If depth[a] < depth[b] then tin[a] < tin[b]
So makes no difference!)

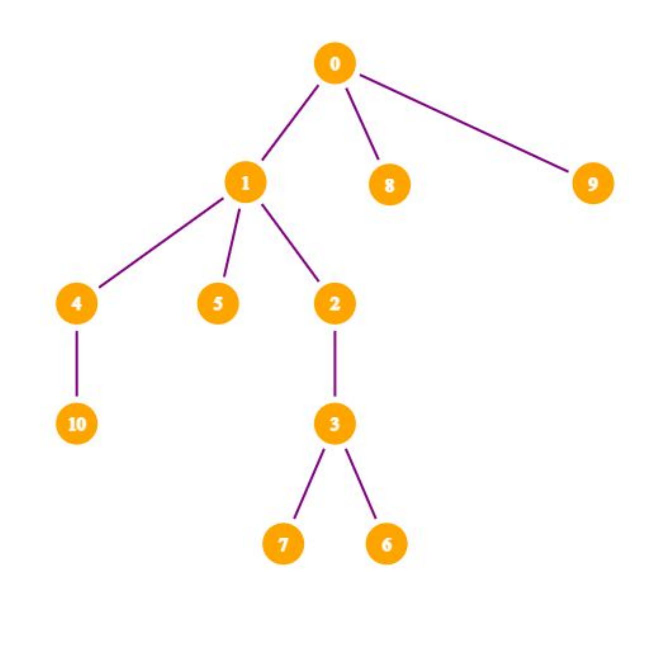
But how can we find the minimal node in this range efficiently???



RMQ = Range Minimum Query

Here we go again

EULER	0	1	4	10	4	1	5	1	2	3	7	3	6	3	2	1	0	8	0	9	0
Time in	0	1	2	3	2	1	6	1	8	9	10	9	12	9	8	1	0	17	0	19	0



3.) Build segment tree using array of *tin* values (or *depth* values)

Then call

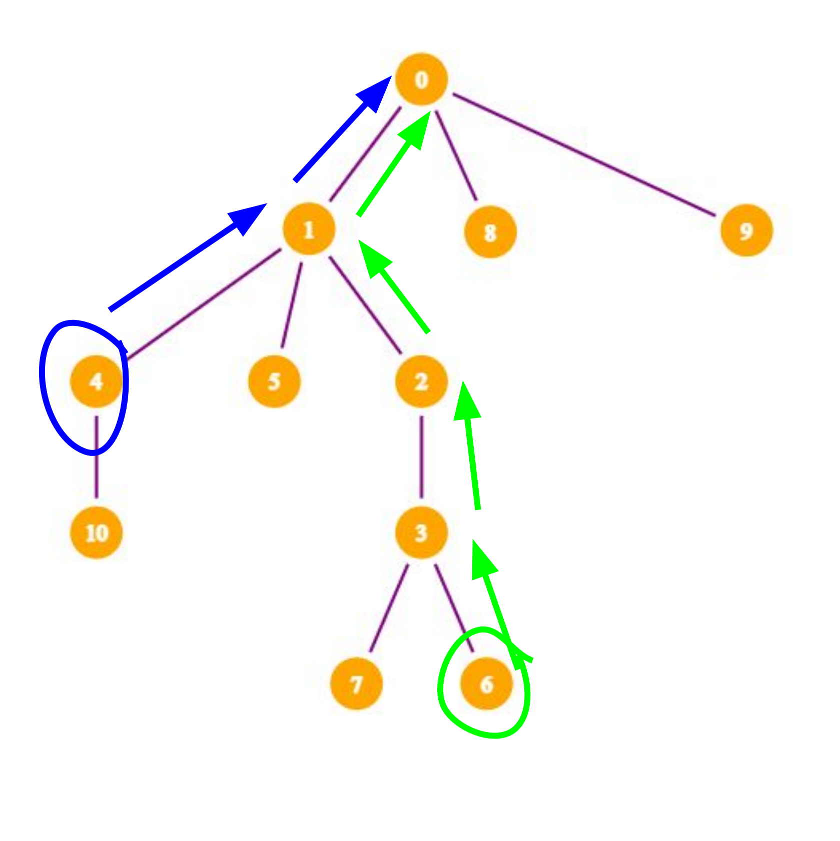
query(tin[a], tin[b])

Where query is segment tree query function.

It will return tin[LCA], which you can use to find LCA.

Each query takes O(logN) time.

One application of LCA



Distance between two nodes in a tree

If c = LCA(a, b)

ab = dist(a) + dist(b) - 2*dist(c)
Where dist(node) is distance from node to root

Thank you

References & Resources

https://codeforces.com/blog/entry/77451#:~:text=Definition%3A,edge%20or%20the%20sum%2C%20etc. (Binary Jumping)

https://saco-evaluator.org.za/presentations/2019%20Camp%202/Lowest%20Common%20Ancestor%20(Andi%20Qu).pdf (Andi's presentation)

https://saco-evaluator.org.za/presentations/2018%20Camp%202/Range%20Queries%20and%20Fenwick%20Trees%20(Yaseen%20Mowzer).pdf (Segment Trees)