P THINK FAST S

Competitive Programming

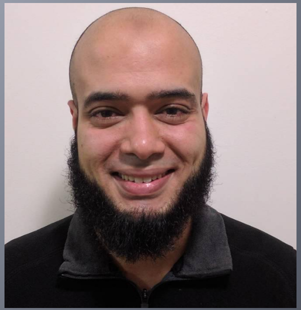
From Problem 2 Solution in O(1)

Dynamic Graphs

Heavy-light tree decomposition - 1

Mostafa Saad Ibrahim

PhD Student @ Simon Fraser University



Heavy-light tree decomposition

- Purpose: Handling queries on dynamic trees
  - Dynamic graph: Add/Remove edge or change costs
  - HLD case: **Fixed** tree structure, **but values can change**
- Algorithm (tutorial - code)
  - Simple DFS that divides the **tree** to set of **chains**
  - The real magic: A prove that # of chains from node to root is O(**log(n)**). Hence, allows **efficient querying**
  - Given the chains, use them for answering queries
- Prerequisites
  - LCA concept (not specific algorithm)
  - Segment tree: To do update/query on the chains

Recall: Range Query on arrays

- Given array of N Numbers and Q queries [Start-end], find in the range/interval:
  - range sum/max/min/average/median/lcm/gcd/xor
  - number of elements repeated K times (k = 1 = distinct)
  - **position** of 1st index with **accumulation >=** C
  - the **smallest** number < S (or their count)
  - Value repeats **exactly once** (use xor) or **most frequent**
  - Find the kth elemnt in the sorted distinct list of range
- Brute force is O(NQ), can we do better?
  - Preprocessing algorithms / Data Structures

Recall: Range Query on arrays

- Data Structures & Algorithms
  - DS: BIT, Segment Tree, heaps, BBST
  - Algos: Square root decomposition, ad hoc preprocessing
- Applying data structures / algorithms
  - Some of them will be the **easiest**
  - While others will be **harder to apply**
  - And some of them might be **impossible** to use
  - Some of them can be easy to apply, but not efficient
  - So **think** about possible **choices** before going with one

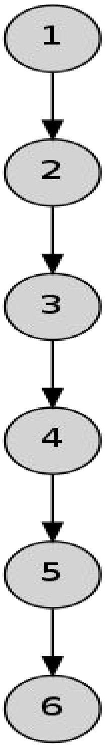
Recall: Range Query on arrays

- Static vs Dynamic arrays (update operation)
  - Sometimes we are given a static array
  - Then queries are just given ranges to compute F
  - Sometimes, you have **update operations**
  - Update position 10 with value 157
  - Some algorithms work for static case only
- Online vs offline processing
  - Sometimes you can read all queries and sort them
  - Then answering might be more efficient
  - This is always doable in competitions (read input file)
  - Sometimes **update operation** makes that impractical

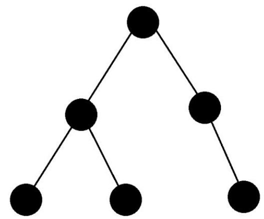
Recall: Range Query on trees

- **Basic Ways (if applicable):**
  - The idea is to convert the tree to array
  - Query 1: Compute something on subtree of given **root**?
  - Easy: **Preorder traversal** over tree flats it to array. Then any root can be identified in this array as a range
  - Query 2: Given Node A, Node B, query on path?
  - Similar to **LCA**, run Euler walk DFS to flat a tree to array
  - Then **path** (A, B) can be mapped to a **range** in the array
  - After that, just apply applicable algorithm for arrays.
  - **However**, notice, that path may have **duplicate** nodes which should be neglected. We can find solution for some algorithms (E.g. MO), but fail to others (Segment Tree).

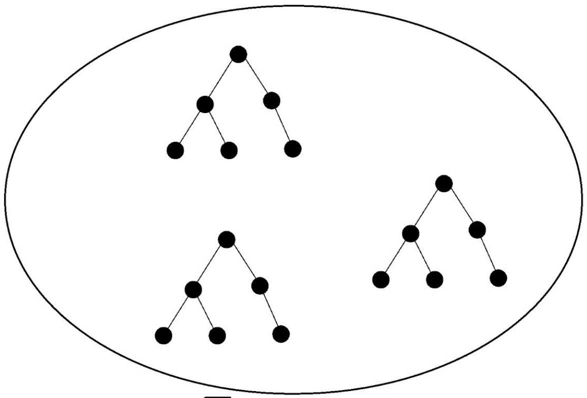
Recall: Tree and Chain



Chain



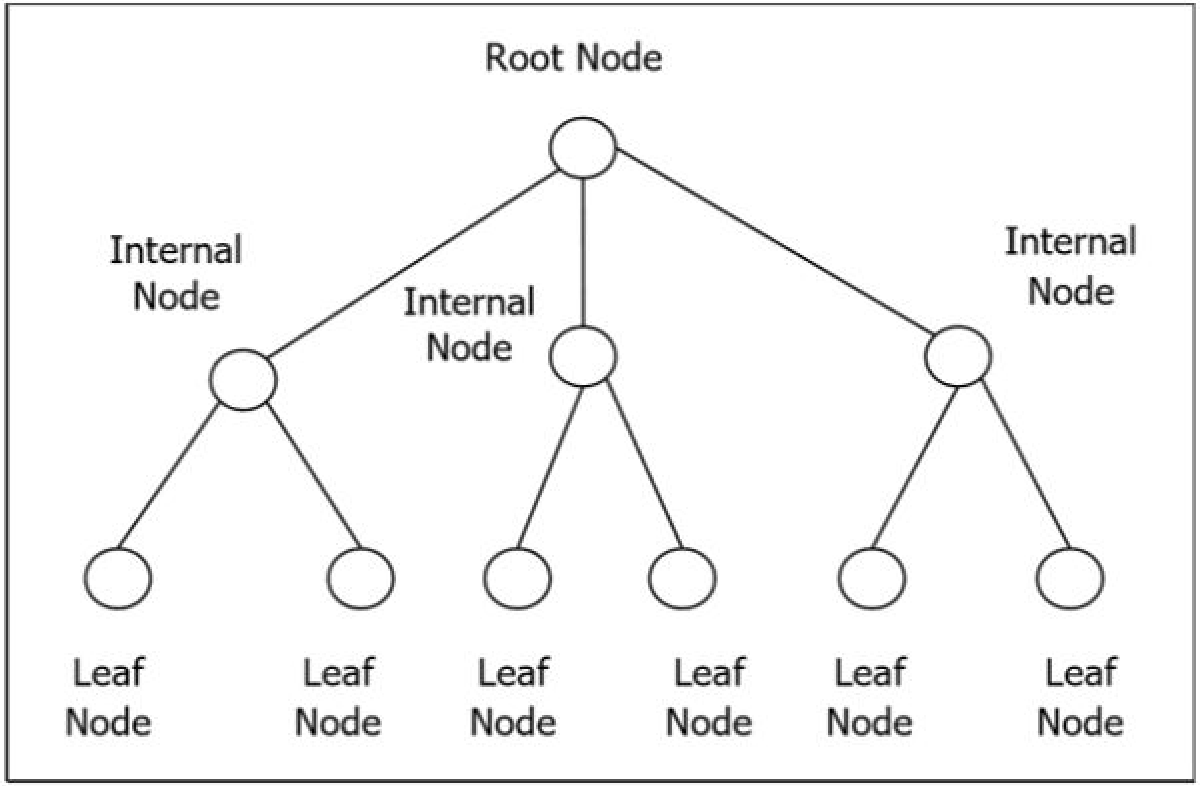
Tree



Forest

Src: https://sheridanmath.wikispaces.com/file/view/GEOMETRY_04.GIF/177066721/GEOMETRY_04.GIF https://blog.anudeep2011.com/heavy-light-decomposition/

Recall: Rooted Tree



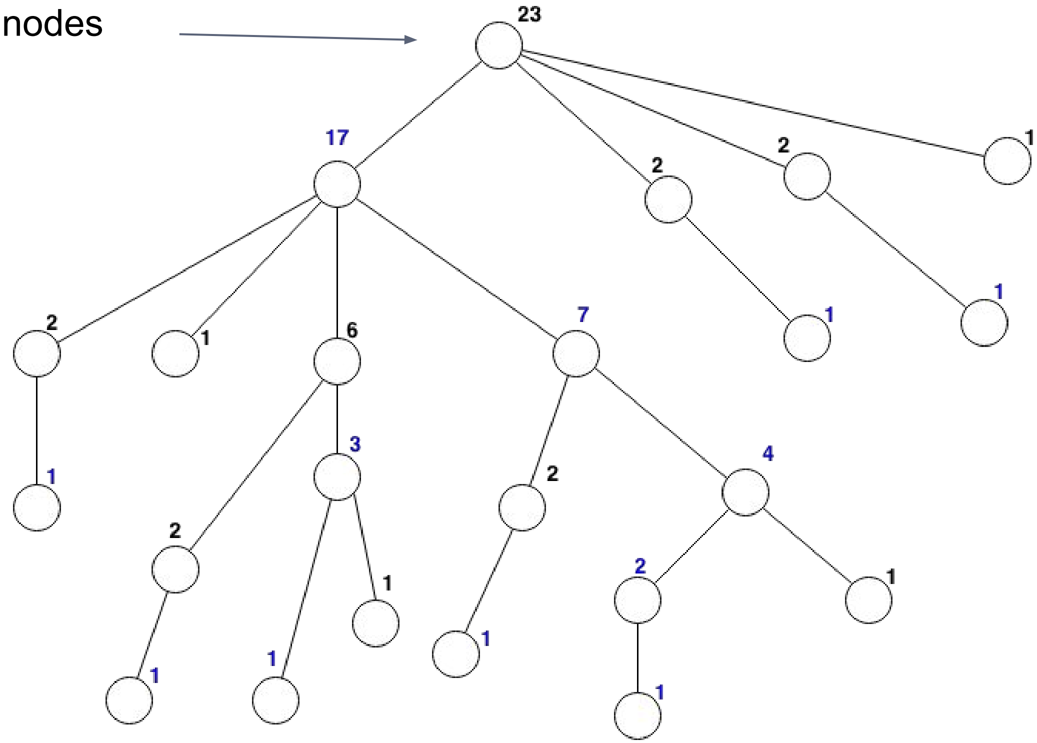
Src: https://www.tutorialspoint.com/discrete_mathematics/images/rooted_tree.jpg

Heavy-light tree decomposition

- In Summary
  - **Split** the tree into **chains** such that **every node** is in the chain with the **child** which has **biggest subtree**.
  - This can be done by a simple DFS
  - For node p, Compute childern tree sizes
  - **Connect p to child c** with biggest size
  - Other children are **independent** subtrees
  - Property: path between 2 nodes is **O(logn) chains**
  - So if each chain is a segment tree, we can do queries in O(logn) * O(segment tree per a range) , e.g. $O(\log^2 n)$

HLD: Compute node subtree size

Rooted tree of 23 nodes



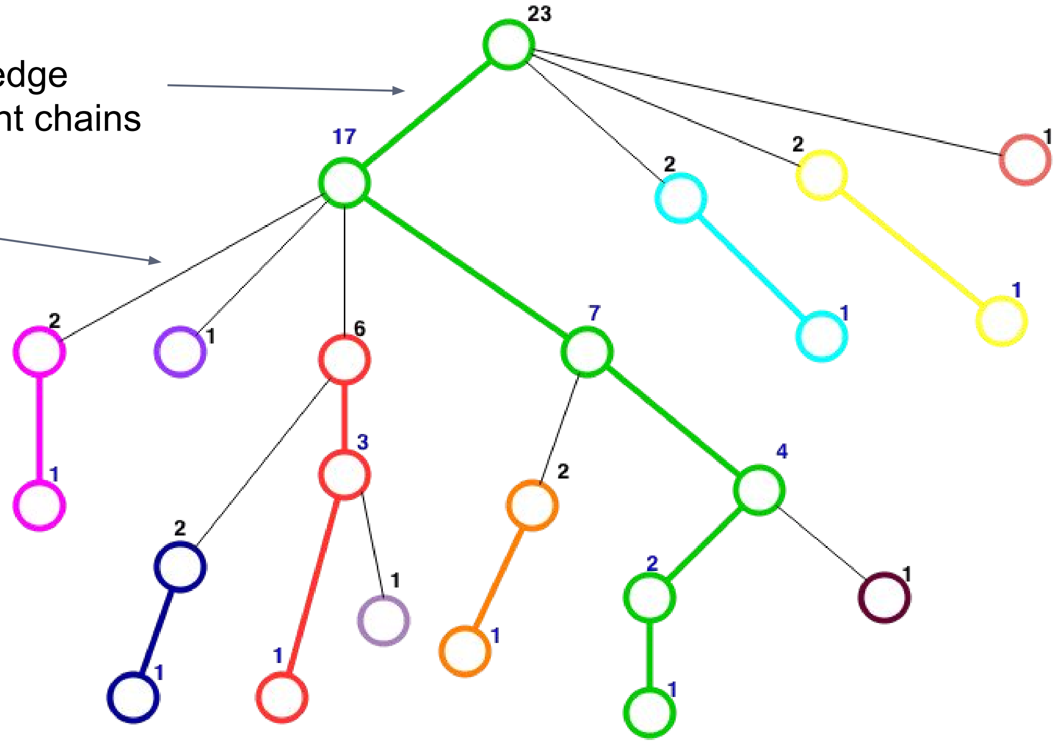
Each node has its sub-tree size written on top.
Each non-leaf node has exactly one special child whose sub-tree size is colored.
Colored child is the one with maximum sub-tree size.

Src: https://blog.anudeep2011.com/heavy-light-decomposition/

HLD: Link node with biggest child

Heavy edge
= Disjoint chains

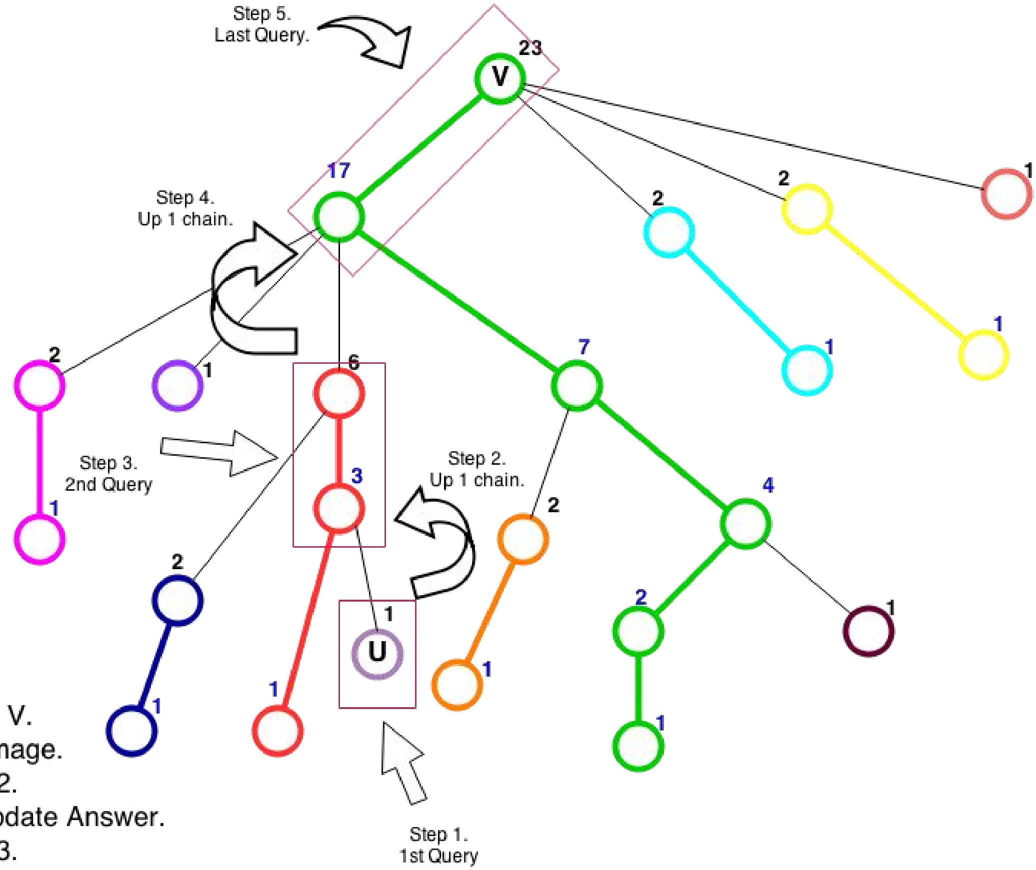
Light edge
= *Connectors*



Each Chain is represented with different color.
Thin Black lines represent the connecting edges. They connect 2 chains.

Src: https://blog.anudeep2011.com/heavy-light-decomposition/

HLD: Path U to (ancestor) V



Consider the path from U to V.
Step 1 : Query for chain 1 in image.
Step 2 : Move up to chain 2.
Step 3 : Query for chain 2 in image. Update Answer.
Step 4 : Move up to chain 3.
Step 5 : Query for chain 3 in image. Update answer.

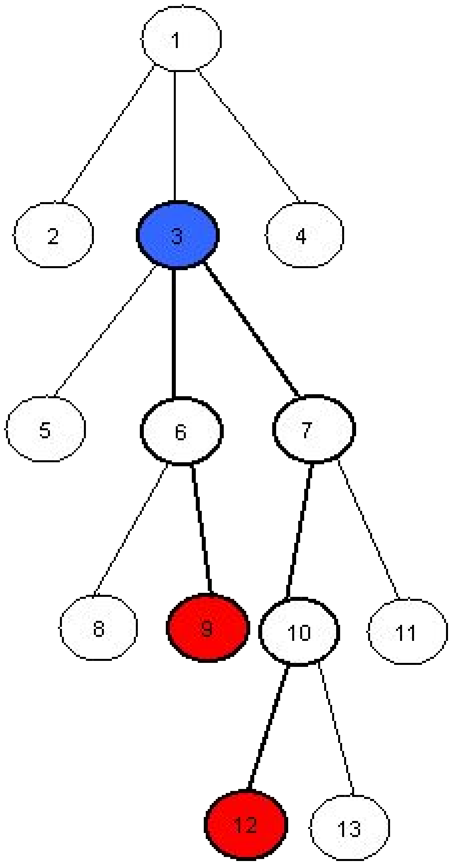
Src: https://blog.anudeep2011.com/heavy-light-decomposition/

HLD: Path U to (any) V

- To go for a child, you just keep going down
- To go an ancestor, you just keep going up
- Otherwise, some go up steps, and then go down
- **LCA**(A, B) is where you start to **flip**

- HLD computes some chains going up from node 1
- And other chains going down (or, going up too, but from the 2nd node)
- Compute V1 from first node’s chains (e.g max on path)
- Compute V2 from 2nd node’s chains (e.g max on path)
- Compute Overall V (e.g. max(v1, v2))

- What are the chains from 9 to 3 and from 12 to 3?



$LCA_T(9, 12) = 3$

Src: https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/

HLD for LCA

- Do we need an extra LCA algorithm?
  - You may think we need LCA algorithm to break path (U, V) to <U, LCA(U, V), V>
  - However, HLD itself can compute LCA
  - Let depth[i] = the computed depth of node i
  - Let root[i] = the root of the chain of node i
  - Keep **moving up** on both U and V till finding their LCA:
  - Compare depth[root[u]] with depth[root[v]]
  - The lower one, goes up to the next chain toward the root
  - E.g. if v is lower ⇒ v = parent[root[v]]
  - You just learned a new algorithm for LCA :)

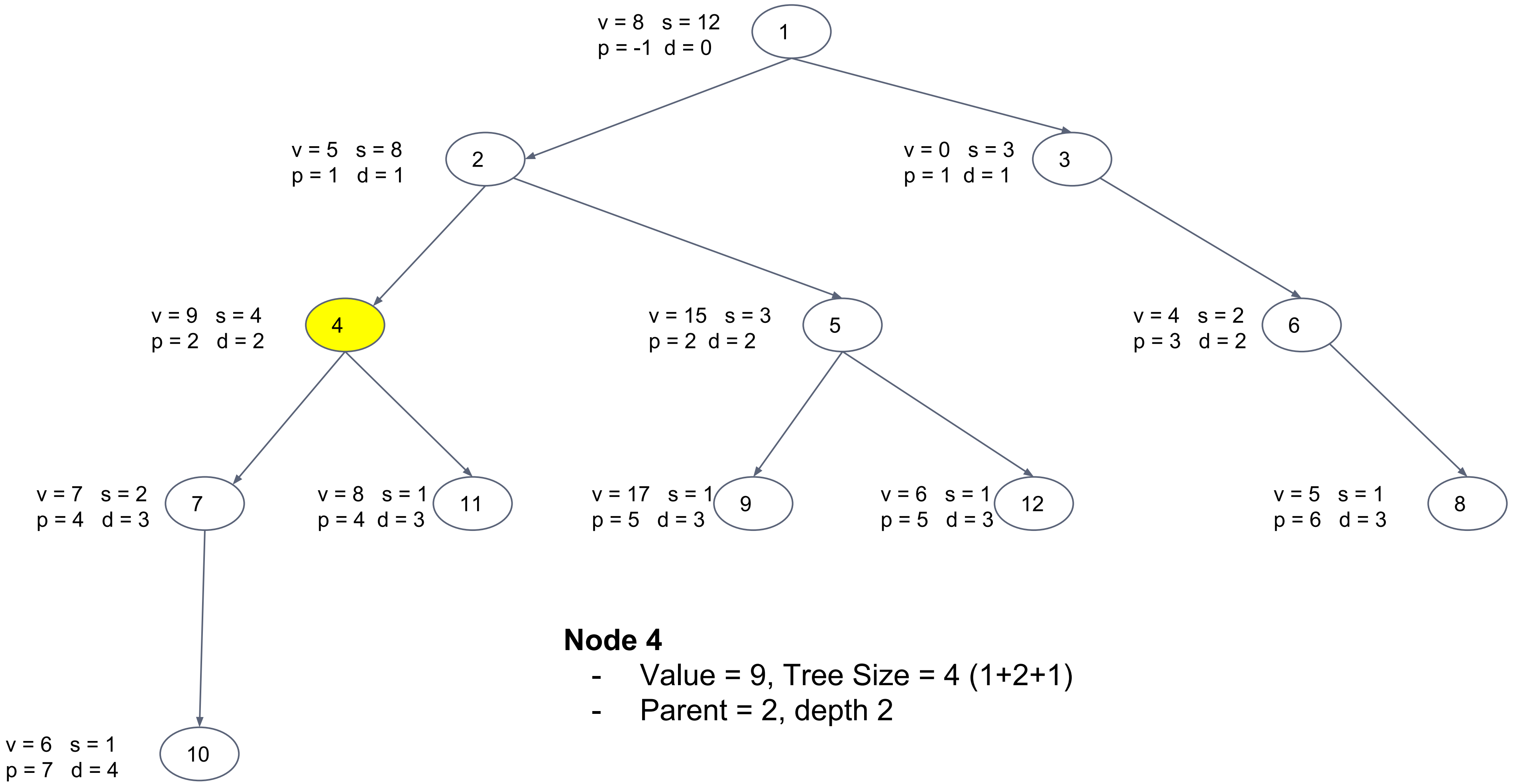
Range Query on chains

- The chain is a sequential set of values
  - So, it corresponds to an array
  - We need a range query data structure or algorithm
  - Usually used: **segment tree (ST)**
  - But others can be used based on problem nature
  - 1 segment tree for all chains vs 1 ST per a chain?
- Value changes
  - Queries to update values (node value - edge value)
  - We just change one chain value (not whole given tree)

Solving Queries on trees

- Thinking
  - First, think for the query on an **array** and solve it
  - Assume you have several arrays, can you compute their overall answer? If yes, HLD is done
  - *E.g. max of path = max over all chains*
  - *One also may just use an euler walk, given that he can handle the duplicate values on path (little scenarios?)*
  - So generally, HLD is the way to go
  - HLD also fits when the tree values changes

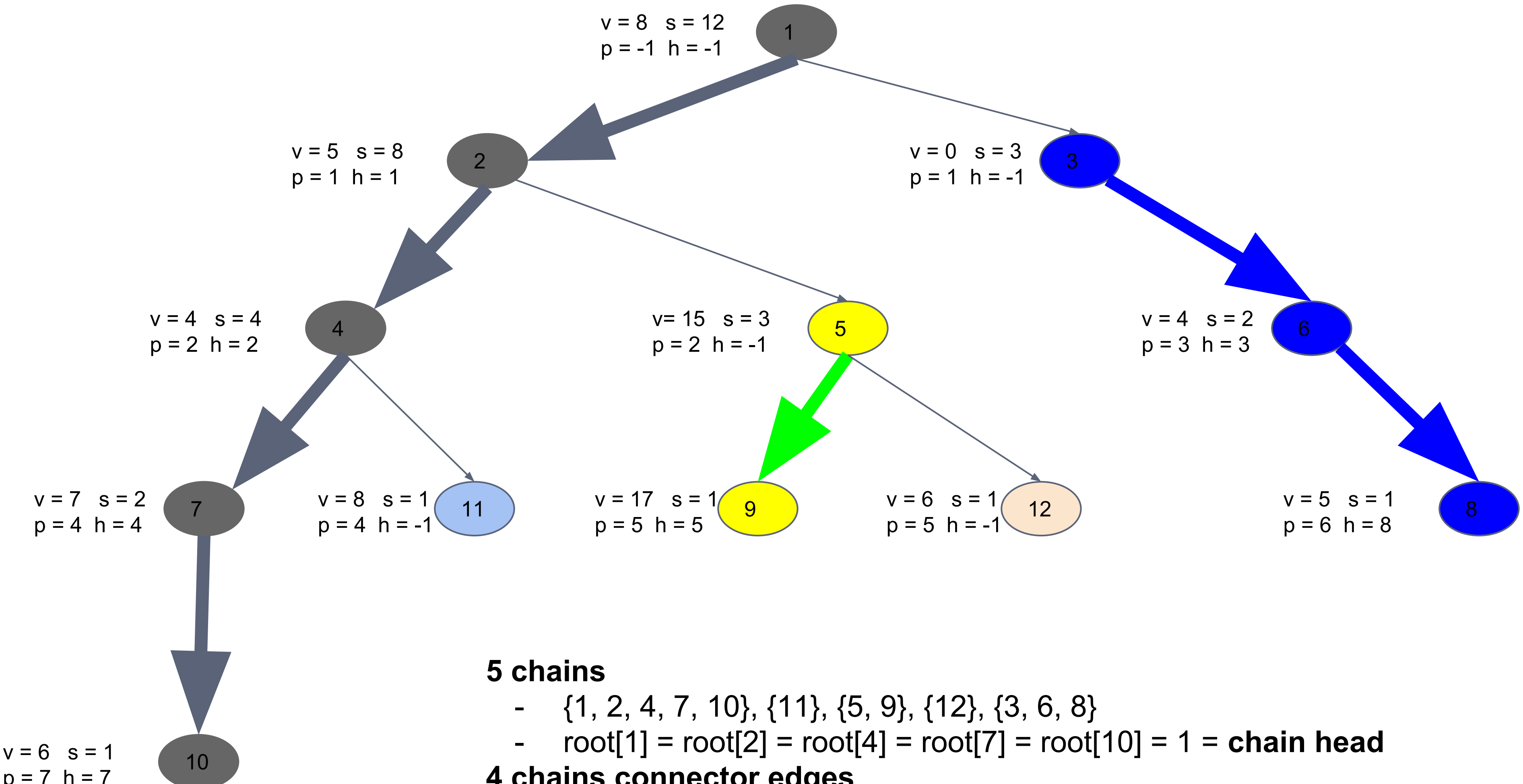
Chains to segment tree



Node 4

- Value = 9, Tree Size = 4 (1+2+1)
- Parent = 2, depth 2

Chains to segment tree



5 chains

- {1, 2, 4, 7, 10}, {11}, {5, 9}, {12}, {3, 6, 8}
- root[1] = root[2] = root[4] = root[7] = root[10] = 1 = **chain head**

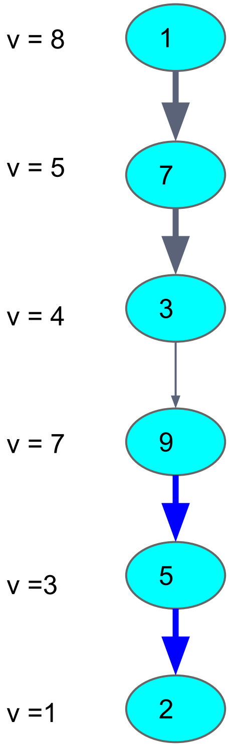
4 chains connector edges

- (4, 11) (2, 5) (5, 12) (1, 3)

Chains to segment tree

- How to map to a **single** segment tree?
  - For each chain, assign consecutive IDs mapping
  - 1 2 4 **<u>7</u>** 10 => 1 2 3 **<u>4</u>** 5 E.g. treePos[7] = 4
  - 11 => 6
  - 5 9 => 7 8
  - 12 => 9
  - 3 6 8 => 10 11 12
  - tree chains ⇒ segment tree leaves values
  - **1 2 4 7 10** 11 **5 9** 12 **3 6 8** [chains ordered]
  - **8 5 4 7 6** 8 **15 17** 6 **0 4 5** [corresponding values]

Values on nodes vs edges

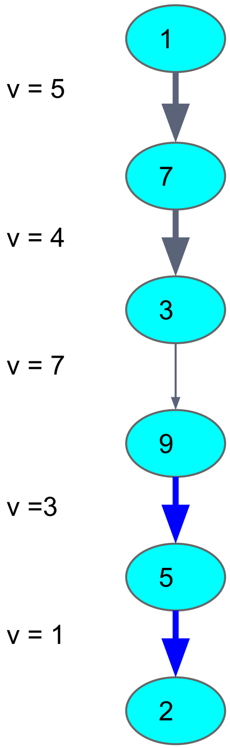


- Assume values on **nodes**
- Assume 2 chains: {1, 7, 3} and {9, 5, 2}
- Assume 1 connector (3, 9)

- tree of 6 nodes => segment tree of 6 nodes
- map to seg tree values:: **8 5 4** 7 3 1

- Path (7-2) needs 2 queries for segment tree
- Range of Query1: (2, 3) [nodes 7 to 3]
- Range of Query2: (4, 6) [nodes 9 to 2]

Values on nodes vs edges

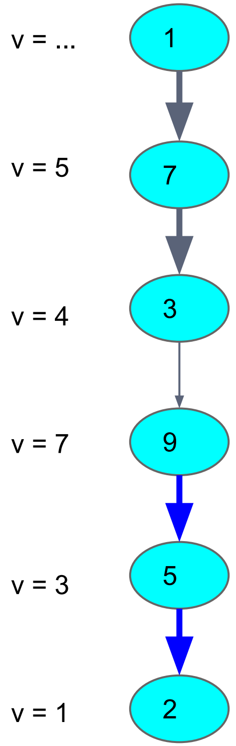


- Assume values on **edges**
- Assume 2 chains: {1, 7, 3} and {9, 5, 2}
- Assume 1 connector (3, 9)
-

- tree of 6 nodes => segment tree of 5 nodes
- mapping a chain now is not that valid
- As connector value over edge (3, 9) will be dropped!

- Trick, convert to a tree with values on nodes
- But we have N-1 values and N edges
- Don’t set value over the tree main root
- For edge (a, b) with cost C
- let node_value[b] = C

Values on nodes vs edges



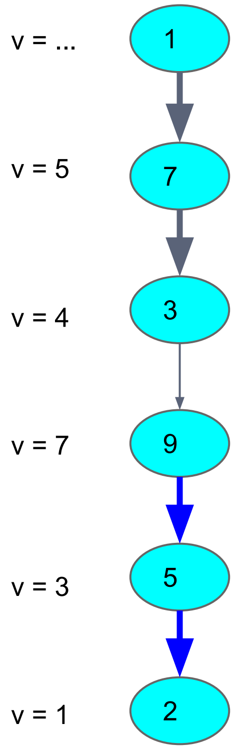
- Assume values on **edges**
- Assume 2 chains: {1, 7, 3} and {9, 5, 2}
- Assume 1 connector (3, 9)
-

- tree of 6 nodes => segment tree of 5 nodes
- mapping a chain now is valid
- But doing queries is little tricky

- Path (7-2) needs 2 queries for segment tree
- Range of Query1: (2+1, 3) [nodes 3 to 3]
    - We need +1, as value 8 is for edge (1-7)
- Range of Query2: (4, 6) [nodes 9 to 2]
    - We did not add 1 to let query consider edge (3, 9)
    - Recall, segment tree node 4 corresponds to edge (3, 9)

Implementation wise, few code changes to consider the +1 case

Values on nodes vs edges



- Assume values on **edges**
- In the LCA(A, B) algorithm, A and B keeps going up till meet at the LCA node
- So, if a node will jump to the next chain connector, then we need to cover this connector
- Range (chain root, current node) covers such a range
- If the 2 nodes A, B are in same chain, then value at A should be ignored, as it is for the previous edge
- Then (A+1, B)

- Moral of that. There is a **single difference** between values of nodes vs edges
- When the 2 nodes are on the same chain, use (A+1, B).
- The remaining **climbing up** the tree is the same for values one edges or nodes
- Code next time clarifies that

Undirected trees

- Undirected to directed
  - Sometimes the given tree is undirected
  - HLD needs a directed tree
  - Pick any node, DFS from it
  - Direct edges
  - In case **values on edges**, mark which edge direction used
  - E.g. if for edge (5, 8) u directed as (8, 5), then we put the value at node 5
  - In fact, it the same DFS for HLD can handle that

Implementing HLD

- Main HLD Algorithm
  - Dropping chains processing, it is an easy code
  - Write 1 (only) **DFS** to compute for each node: its parent, its depth and its heavy child (easy to code)
  - For every chain with root i, iterate on all its nodes j and set their **root[j]** = i (easy to code)

Implementing HLD

- HLD with range query data structure
  - This is the main purpose of **using** HLD
  - It is more sort of **implementation skills**
  - One way is **1 segment tree** to **all** tree nodes (e.g. for every tree node j, map its position to the segment tree array position: *treePos[j] = segment_tree_array_idx*
  - Then sub-chain from node **v to chain root** corresponds in the segment tree to (treePos[root[v]], treePos[v])
  - And sub-chain between 2 nodes **(u, v) on the same chain** corresponds to (treePos[u], treePos[v])
  - The other way is 1 segment tree **per a chain**
  - Other data structures might be suitable (e.g. BIT)

HLD Efficiency

- The core of the algorithm
  - HLD is a DFS decomposes the vertices of the tree into **disjoint chains** => matter of O(E+V)
  - As mentioned, the core of queries efficiency is because of O(logn) chains only from a node to the tree root
  - Can you prove that?
  - Hint: prove the connector tree size <= parent / 2
  - Read1 Read2 Read3

HLD Efficiency

- The proof
  - The number of chains is bounded b **connector edges**
  - We can prove size(a connector tree) <= size(parent)/2, and so on => O(logn)
  - Why? What is **minimum** size of the **largest** subtree?
  - If a tree has N = 100, M = 9, then a balanced children each have 99/9=11 nodes. This is the minimum: Ceil(n/m).
  - So size(any connector tree) <= Ceil(n/2) = the max child

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً