Advanced Computer Contest Preparation
Lecture 28

HEAVY-LIGHT DECOMPOSITION

Problem

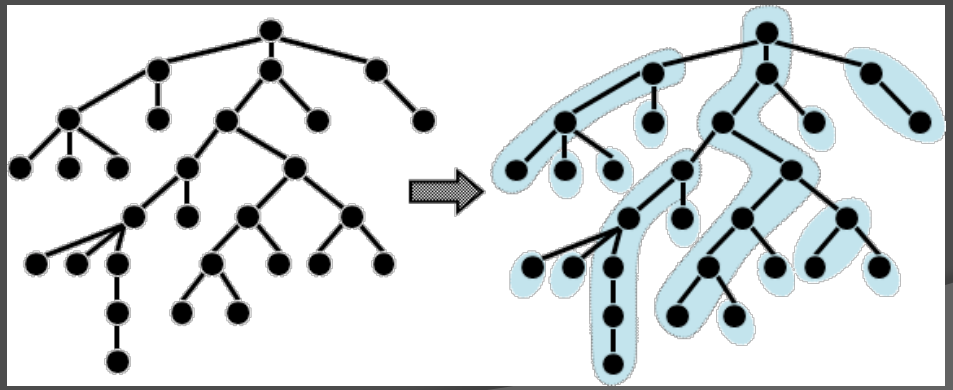
- Given a rooted, edge-weighted tree with N nodes
- We can do 2 possible operations:
  - Update: update the weight of an edge
  - Query: return the edge with the smallest weight on the path from root to query node

Solutions?

- Brute force:
  - Update: update edge in $O(1)$ time
  - Query: travel from root to query node in $O(N)$ time

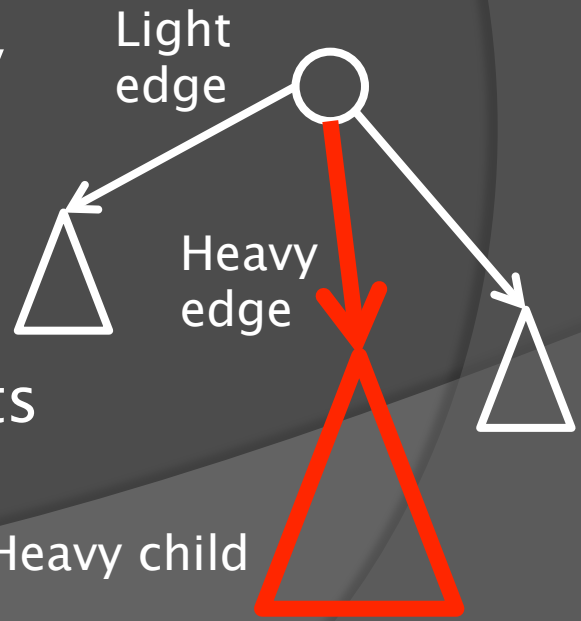
Heavy-Light Decomposition

- Heavy-Light Decomposition (HLD) is a method of partitioning the edges of a tree into heavy and light edges
- Key concept: compressing paths to allow quick tree traversal and tree queries



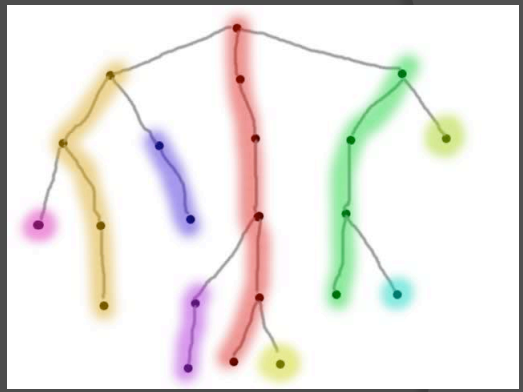
Heavy-Light Decomposition

- An edge from a node to a child is heavy if that child's subtree is larger than other children
  - This child is also called the heavy child
- If there are ties, choose 1 edge arbitrarily
- All other edges from that node to its children are light



Heavy-Light Decomposition

- A continuous sequence of heavy edges is called a heavy path
- The top of the heavy path is the highest node that does not have a heavy edge to its parent (or is the root)
- Each node stores the top node on its heavy path



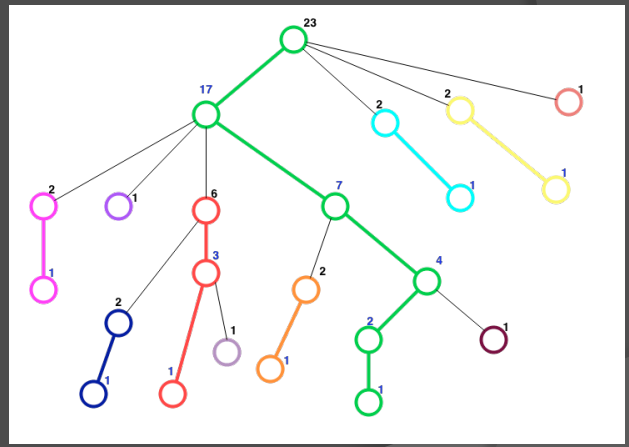
HLD – Pseudocode

```
dfs1(node)
      for each child of node
            parent[child] ← node
            dfs1(child)
      heavy[node] ← child of node with max size

dfs2(node,top_node)
      top[node] ← top_node
      dfs2(heavy[node],top_node)
      for each child of node ≠ heavy[node]
            dfs2(child,child)
```

Heavy-Light Decomposition

- When we travel from a node to the root, we use the top node of the heavy path as “shortcuts”
- Large sections of the tree can be skipped
- To go to the root, we keep going to the parent of the top node until the top node is the root



HLD Traversal – Pseudocode

```
node
while node ≠ root
      if top[node] = root
            node ← root
            break
      node ← parent[top[node]]
```

HLD Proof

- Let $size(x)$ be the number of nodes in the subtree rooted at x
- For a node nd, no light child can have $size(child) > size(nd)/2$
  - If a child has $size(child) > size(nd)/2$, it must be the the child with the largest subtree
- Therefore, there are at most $\log N$ light edges from a node to the root
- Heavy edges add no additional time; total time complexity to travel from node to root is $O(\log N)$

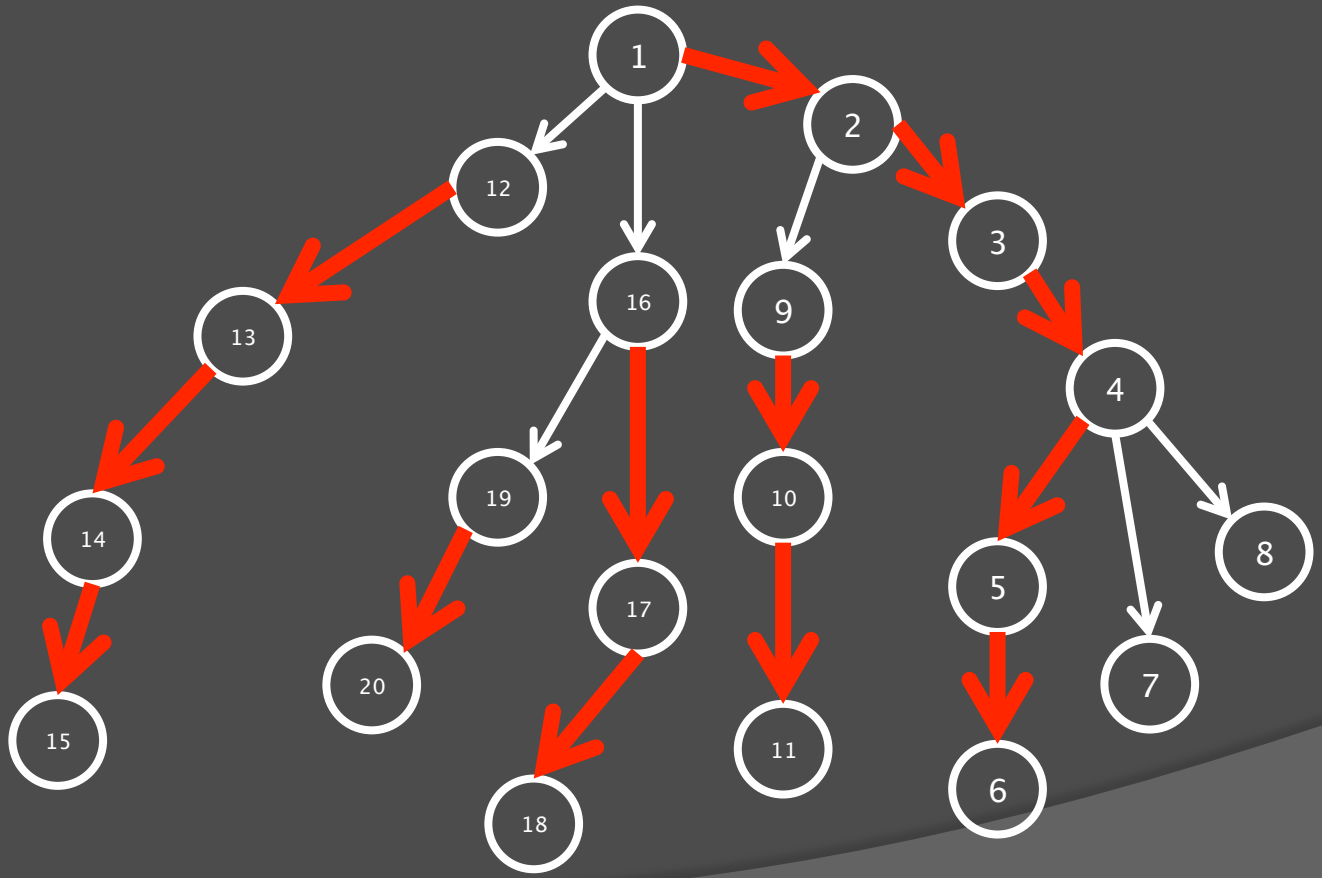
Tree Queries

- HLD works well for queries involving paths from the root to a node
- Note that there are $O(\log N)$ heavy paths from a node to the root
- We can use a data structure (PSA, segment tree, BIT, etc.) in order to quickly query all nodes/edges on the heavy path
- Therefore, the total time complexity to perform a tree query is $O(\log N) \times$ *data structure time complexity*

Tree Query Indexing

- Nodes/edges need to be indexed differently in order to use the data structure properly
- Data structures can handle range queries well
- Thus to query heavy paths easily, all heavy paths should be indexed consecutively
- By doing this, we can query a heavy path by querying the data structure from the index of the top of the path to the index of the current node

Tree Query Indexing



Tree Query Indexing

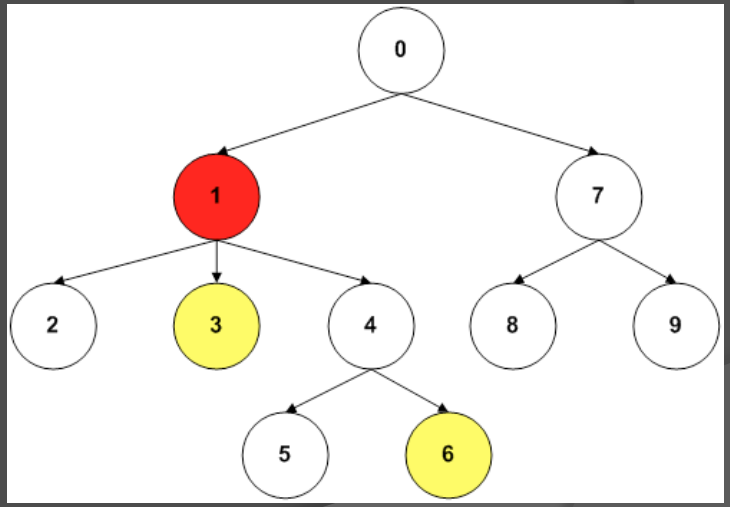
- To index nodes, perform a DFS
- Keep track of an index counter
- Set node's index equal to index counter and increment the counter
- Run the DFS on the child with a heavy edge
- Run the DFS on other children
- This DFS can be merged with the second HLD DFS

Tree Query Indexing – Pseudocode

```
index ← 1
dfs_index(node)
      id[node] ← index
      index ← index + 1
      dfs_index(heavy[node])
      for each child of node ≠ heavy[node]
            dfs_index(child)
```

Lowest Common Ancestor

- In a rooted tree, the lowest common ancestor (LCA) of two nodes is the node that is closest to the root that is common in the paths from the root to the two nodes
- Alternative explanation: The node that is closest to the root in the shortest path between two nodes



LCA – Method

- Have pointers to the two nodes
- Keep moving the deeper pointer upward
- Stop when the pointers point at the same node
- HLD modification:
  - Move the pointer whose top node is deeper
  - Stop when the pointers are on the same heavy path, higher pointer is LCA
  - Runtime: $O(\log N)$

LCA – Pseudocode

```
u,v
while top[u] ≠ top[v]
      if depth[top[u]] < depth[top[v]]
            swap(u,v)
      u ← parent[top[u]]
if depth[u] < depth[v]
      swap(u,v)
//v is the LCA
```

THANK YOU!