Implementation

Let us try to implement the technique of heavy-light decomposition where we will define the **heavy edge** from a node to be the edge that leads us to the child of the node with the largest subtree size. As we said that we need to build a data structure that helps to answer queries over given ranges across different chains, we can ease our implementation by building a single segment tree by keeping track of the disjoint chains(disjoint intervals in the linearized array).

The following pseudo-code also gives an example of how to handle queries for finding the maximum value of a node of the path between a given pair of nodes, however the same can be done using LCA as discussed above.

Pseudocode:

```
/*
        The function takes input:
            1.  cur_node: The current node in subsize(dfs)
            2.  adj: Adjacency list representation of the tree(passed as reference)
            3.  parent: parent array to store the parent of the nodes(passed as
                reference)
            4.  depth: depth array to store the depth of nodes in dfs(passed as reference)
            5.  heavy: heavy array to store the heavy child of a node(a heavy edge from
                node to heavy[node], passed as reference)
        The function calculates the subtree size and finds the heavy child for a node, the
        parent and depth array calculation will be useful for the decomposition of the
        tree.
*/
function subsize(cur_node, adj, parent, depth, heavy)

        //  Current Subtree size of the node
        cur_sz = 1
        //  Initializing max_size for the current node
        max_sz = 0
        for child in adj[cur_node]
                if child != parent[cur_node]
                        parent[child] = cur_node
                        depth[child] = depth[cur_node] + 1
                        child_sz = subsize(child, adj, parent, depth, heavy)
                        cur_sz = cur_sz + child_sz
                        /*
                                If the subtree size for the child is greater than the max size
                                found so far, update the max_sz and heavy child for the
                                cur_node.
```

```
                    */
                    if child_sz > max_sz
                            max_sz = child_sz
                            heavy[cur_node] = child

        return cur_sz

/*
        The function takes input:
            1. cur_node: The current node in the chain
            2. h: The head node for the current chain
            3. adj: Adjacency list representation of the tree(passed as reference)
            4. parent: parent array to store the parent of the nodes(passed as
               reference)
            5. head: The head array for storing the head of the chain for the
               node(passed as reference)
            6. pos: The linearized array for the tree storing the elements in contiguous
               form for all the nodes belonging to the same chain(passed as reference)
            7. heavy: heavy array to store the heavy child of a node(a heavy edge from
               node to heavy[node], passed as reference)
            8. cur_pos: Denotes the current position in the pos array to be filled.

        The function decomposes the tree into disjoint chains and  linearizes the tree
        into pos array in the form of disjoint contiguous segments for each different
        chain
*/
function HLD(cur_node, h, adj, parent, head, pos, heavy, cur_pos)

        //   Storing the head for the current node i.e head of the chain.
        head[cur_node] = h
        //   Storing the position for the current node in the pos array.
        pos[cur_node] = cur_pos
        cur_pos = cur_pos + 1

        /*
                Calling recursively HLD for decomposition of the heavy child of the
                current node to store the nodes belonging to the same chain in a
                contiguous manner.
        */
        if heavy[cur_node] != -1
                HLD(heavy[cur_node], h, adj, parent, head, pos, heavy, cur_pos)

        /*
```

```
            Calling recursively HLD for the light children of the current node by
            changing the head denoting the start of the new chain.
        */
        for child in adj[cur_node]
            if child != parent[cur_node] and child != heavy[cur_node]
                HLD(child, child, adj, parent, head, pos, heavy, cur_pos)



/*
        The function takes nodes a and b as the input and returns the maximum value
        of the node in the path from a to b.
        It uses a segment tree built over the linearized array pos to handle range queries
        for finding the max element.
*/
function query(a, b)

        //   Initializing answer to 0.
        answer = 0
        /*
            Jumping across different chains(different disjoint contiguous segments in
            the array) from bottom to up and finding the maximums across them.
        */
        while head[a] != head[b]
            if depth[head[a]] > depth[head[b]]
                swap(a, b)
            /*
                Using segment tree built over the linearized array to calculate
                maximum over a given segment
            */
            cur_max = segment_tree.query(pos[head[b]], pos[b])
            answer = max(answer, cur_max)
            b = parent[head[b]]

        if depth[a] > depth[b]
            swap(a, b)

        /*
            Handling the case when a and b belong to the same chain then querying
            from the segment from [pos[a], pos[b]]
        */
        cur_max = segment_tree.query(pos[a], pos[b])
        answer = max(answer, cur_max)
```

```
        return answer
```

Time Complexity: O(V + E + Q*$\log_2$V*$\log_2$V), where V is the number of vertices in the tree and E being the number of edges. Each query can be answered in $\log_2$V*$\log_2$V time which takes a total of Q*$\log_2$V*$\log_2$V time for Q queries.