Heavy Light Decomposition | Set 2 (Implementation) - GeeksforGeeks

We strongly recommend to refer below post as a prerequisite of this.

In the above post, we discussed the Heavy-light decomposition (HLD) with the help of below example.

Suppose we have **an unbalanced tree (not necessarily a Binary Tree) of n nodes**, and we have to perform operations on the tree to answer a number of queries, each can be of one of the two types:

1. **change(a, b)**: Update weight of the a^{th} edge to b.
2. **maxEdge(a, b)**: Print the maximum edge weight on the path from node a to node b. For example maxEdge(5, 10) should print 25.

In this article implementation of same is discussed

Our line of attack for this problem is:

1. Creating the tree
2. Setting up the subtree size, depth and parent for each node (using a DFS)
3. Decomposing the tree into disjoint chains
4. Building up the segment tree
5. Answering queries

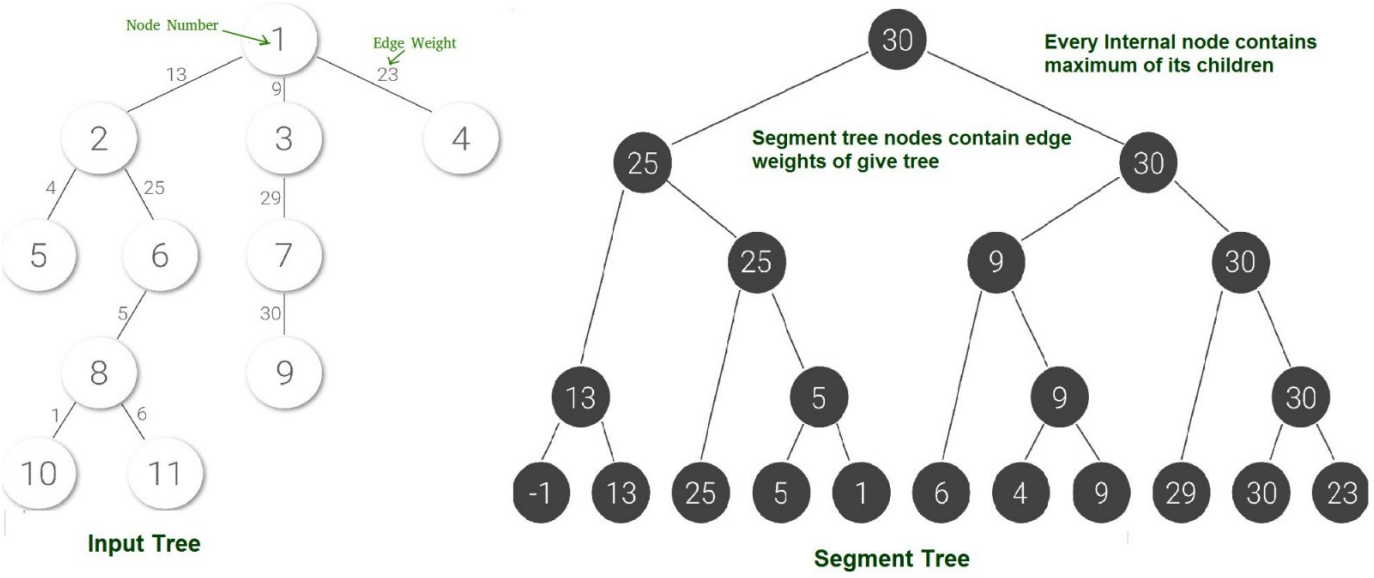
1. Tree Creation: Implementation uses adjacency matrix representation of the tree, for the ease of understanding. One can use adjacency list rep with some changes to the source. If edge number e with weight w exists between nodes u and v, we shall store e at tree[u][v] and tree[v][u], and the weight w in a separate linear array of edge weights (n-1 edges).

2. Setting up the subtree size, depth and parent for each node: Next we do a DFS on the tree to set up arrays that store parent, subtree size and depth of each node. Another important thing we do at the time of DFS is storing the deeper node of every edge we traverse. This will help us at the time of updating the tree (change() query) .

3 & 4. Decomposing the tree into disjoint chains and Building Segment Tree Now comes the most important part: HLD. As we traverse the edges and reach nodes (starting from the root), we place the edge in the segment tree base, we decide if the node will be a head to a new chain (if it is a normal child) or will the current chain extend (special child), store the chain ID to which the node belongs, and store its place in the segment tree base (for future queries). The base for segment tree is built such that all edges belonging to the same chain are together, and chains are separated by light edges.

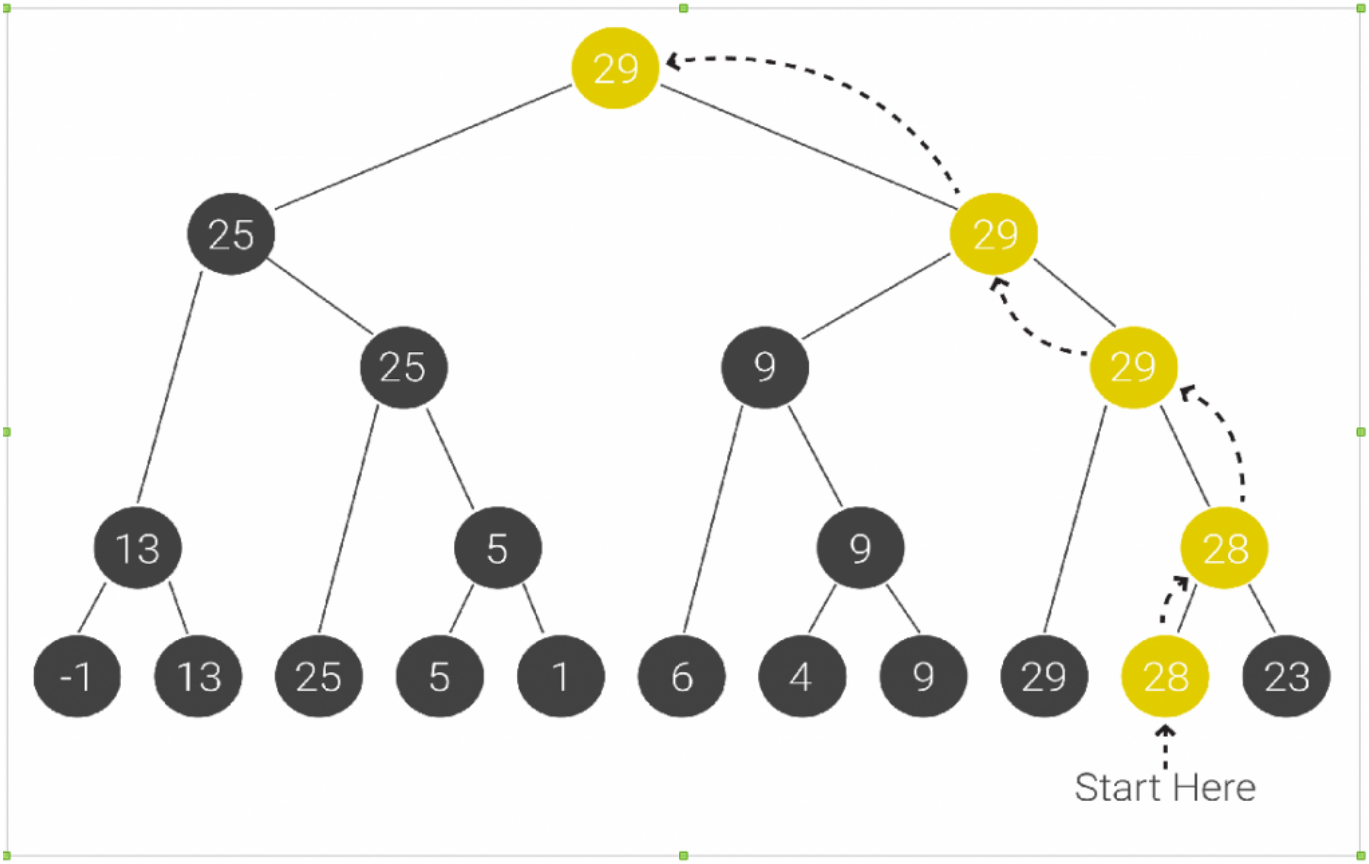
Illustration: We start at node 1. Since there wasn't any edge by which we came to this node, we insert '-1' as the imaginary edge's weight, in the array that will act as base to the segment tree.

Next, we move to node 1's special child, which is node 2, and since we traversed edge with weight 13, we add 13 to our base array. Node 2's special child is node 6. We traverse edge with weight 25 to reach node 6. We insert in base array. And similarly we extend this chain further while we haven't reached a leaf node (node 10 in our case).
Then we shift to a normal child of parent of the last leaf node, and mark the beginning of a new chain. Parent here is node 8 and normal child is node 11. We traverse edge with weight 6 and insert it into the base array. This is how we complete the base array for the segment tree.
Also remember that we need to store the position of every node in segment tree for future queries. Position of node 1 is 1, node 2 is 2, node 6 is 3, node 8 is 4, …, node 11 is 6, node 5 is 7, node 9 is 10, node 4 is 11 (1-based indexing).



5. Answering queries
We have discussed **mexEdge()** query in detail in previous post. For maxEdge(u, v), we find max weight edge on path from u to LCA and v to LCA and return maximum of two.

For **change()** query, we can update the segment tree by using the deeper end of the edge whose weight is to be updated. We will find the position of deeper end of the edge in the array acting as base to the segment tree and then start our update from that node and move upwards updating segment tree. Say we want to update edge 8 (between node 7 and node 9) to 28. Position of deeper node 9 in base array is 10, We do it as follows:



Below is C++ implementation of above steps.

```
/* C++ program for Heavy-Light Decomposition of a tree
*/#include<bits/stdc++.h>
using namespace std;
#define N 1024
int tree[N][N]; // Matrix representing the tree
/* a tree node structure. Every node has a parent, depth,
   subtree size, chain to which it belongs and a position
   in base array*/
struct treeNode
{
    int par; // Parent of this node
    int depth; // Depth of this node
    int size; // Size of subtree rooted with this node
    int pos_segbase; // Position in segment tree base
    int chain;
} node[N];/* every Edge has a weight and two ends. We store deeper end */
struct Edge
{
    int weight; // Weight of Edge
    int deeper_end; // Deeper end
} edge[N];/* we construct one segment tree, on base array */
struct segmentTree
{
```

```
    int base_array[N], tree[6*N];
} s;// A function to add Edges to the Tree matrix// e is Edge ID, u and v
are the two nodes, w is weight
void addEdge(int e, int u, int v, int w)
{
    /*tree as undirected graph*/
    tree[u-1][v-1] = e-1;
    tree[v-1][u-1] = e-1;
    edge[e-1].weight = w;
}// A recursive function for DFS on the tree// curr is the current node,
prev is the parent of curr,// dep is its depth
void dfs(int curr, int prev, int dep, int n)
{
    /* set parent of current node to predecessor*/
    node[curr].par = prev;
    node[curr].depth = dep;
    node[curr].size = 1;
    /* for node's every child */
    for (int j=0; j<n; j++)
    {
        if (j!=curr && j!=node[curr].par && tree[curr][j]!=-1)
        {
            /* set deeper end of the Edge as this child*/
            edge[tree[curr][j]].deeper_end = j;
            /* do a DFS on subtree */
            dfs(j, curr, dep+1, n);
            /* update subtree size */
            node[curr].size+=node[j].size;
        }
    }
}// A recursive function that decomposes the Tree into chains
void hld(int curr_node, int id, int *edge_counted, int *curr_chain,
        int n, int chain_heads[])
{
    /* if the current chain has no head, this node is the first node
    * and also chain head */
    if (chain_heads[*curr_chain]==-1)
        chain_heads[*curr_chain] = curr_node;
    /* set chain ID to which the node belongs */
    node[curr_node].chain = *curr_chain;
    /* set position of node in the array acting as the base to
       the segment tree */
```

```
    node[curr_node].pos_segbase = *edge_counted;
    /* update array which is the base to the segment tree */
    s.base_array[(*edge_counted)++] = edge[id].weight;
    /* Find the special child (child with maximum size)  */
    int spcl_chld = -1, spcl_edg_id;
    for (int j=0; j<n; j++)
        if (j!=curr_node && j!=node[curr_node].par && tree[curr_node][j]!=-1)
            if (spcl_chld==-1 || node[spcl_chld].size < node[j].size)
                spcl_chld = j, spcl_edg_id = tree[curr_node][j];
    /* if special child found, extend chain */
    if (spcl_chld!=-1)
        hld(spcl_chld, spcl_edg_id, edge_counted, curr_chain, n,
chain_heads);
    /* for every other (normal) child, do HLD on child subtree as separate
        chain*/
    for (int j=0; j<n; j++)
    {
        if (j!=curr_node && j!=node[curr_node].par &&
                j!=spcl_chld && tree[curr_node][j]!=-1)
        {
            (*curr_chain)++;
            hld(j, tree[curr_node][j], edge_counted, curr_chain, n,
chain_heads);
        }
    }
}// A recursive function that constructs Segment Tree for array[ss..se).//
si is index of current node in segment tree st
int construct_ST(int ss, int se, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se-1)
    {
        s.tree[si] = s.base_array[ss];
        return s.base_array[ss];
    }
    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = (ss + se)/2;
    s.tree[si] =  max(construct_ST(ss, mid, si*2),
                      construct_ST(mid, se, si*2+1));
    return s.tree[si];
```

```
}// A recursive function that updates the Segment Tree// x is the node to be updated to value val// si is the starting index of the segment tree// ss, se mark the corners of the range represented by si
int update_ST(int ss, int se, int si, int x, int val)
{
    if(ss > x || se <= x);
    else if(ss == x && ss == se-1)s.tree[si] = val;
    else
    {
        int mid = (ss + se)/2;
        s.tree[si] =  max(update_ST(ss, mid, si*2, x, val),
                          update_ST(mid, se, si*2+1, x, val));
    }
    return s.tree[si];
}// A function to update Edge e's value to val in segment tree
void change(int e, int val, int n)
{
    update_ST(0, n, 1, node[edge[e].deeper_end].pos_segbase, val);
    // following lines of code make no change to our case as we are
    // changing in ST above
    // Edge_weights[e] = val;
    // segtree_Edges_weights[deeper_end_of_edge[e]] = val;
}// A function to get the LCA of nodes u and v
int LCA(int u, int v, int n)
{
    /* array for storing path from u to root */
    int LCA_aux[n+5];
    // Set u is deeper node if it is not
    if (node[u].depth < node[v].depth)
        swap(u, v);
    /* LCA_aux will store path from node u to the root*/
    memset(LCA_aux, -1, sizeof(LCA_aux));
    while (u!=-1)
    {
        LCA_aux[u] = 1;
        u = node[u].par;
    }
    /* find first node common in path from v to root and u to
        root using LCA_aux */
    while (v)
    {
        if (LCA_aux[v]==1)break;
```

```
        v = node[v].par;
    }
    return v;
}/*  A recursive function to get the minimum value in a given range
    of array indexes. The following are parameters for this function.
    st    --> Pointer to segment tree
    index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
    ss & se  --> Starting and ending indexes of the segment represented
                 by current node, i.e., st[index]
    qs & qe  --> Starting and ending indexes of query range */
int RMQUtil(int ss, int se, int qs, int qe, int index)
{
    //printf("%d,%d,%d,%d,%d\n", ss, se, qs, qe, index);
    // If segment of this node is a part of given range, then return
    //  the min of the segment
    if (qs <= ss && qe >= se-1)
        return s.tree[index];
    // If segment of this node is outside the given range
    if (se-1 < qs || ss > qe)
        return -1;
    // If a part of this segment overlaps with the given range
    int mid = (ss + se)/2;
    return max(RMQUtil(ss, mid, qs, qe, 2*index),
               RMQUtil(mid, se, qs, qe, 2*index+1));
}// Return minimum of elements in range from index qs (quey start) to// qe
(query end).  It mainly uses RMQUtil()
int RMQ(int qs, int qe, int n)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }
    return RMQUtil(0, n, qs, qe, 1);
}// A function to move from u to v keeping track of the maximum// we move
to the surface changing u and chains// until u and v donot belong to the
same
int crawl_tree(int u, int v, int n, int chain_heads[])
{
    int chain_u, chain_v = node[v].chain, ans = 0;
```

```
    while (true)
    {
        chain_u = node[u].chain;
        /* if the two nodes belong to same chain,
        * we can query between their positions in the array
        * acting as base to the segment tree. After the RMQ,
        * we can break out as we have no where further to go */
        if (chain_u==chain_v)
        {
            if (u==v);    //trivial
            else
                ans = max(RMQ(node[v].pos_segbase+1, node[u].pos_segbase, n),
                          ans);
            break;
        }
        /* else, we query between node u and head of the chain to which
           u belongs and later change u to parent of head of the chain
           to which u belongs indicating change of chain */
        else
        {
            ans = max(ans,
                      RMQ(node[chain_heads[chain_u]].pos_segbase,
                          node[u].pos_segbase, n));
            u = node[chain_heads[chain_u]].par;
        }
    }
    return ans;
}// A function for MAX_EDGE query
void maxEdge(int u, int v, int n, int chain_heads[])
{
    int lca = LCA(u, v, n);
    int ans = max(crawl_tree(u, lca, n, chain_heads),
                  crawl_tree(v, lca, n, chain_heads));
    printf("%d\n", ans);
}// driver function
int main()
{
    /* fill adjacency matrix with -1 to indicate no connections */
    memset(tree, -1, sizeof(tree));
    int n = 11;
    /* arguments in order: Edge ID, node u, node v, weight w*/
    addEdge(1, 1, 2, 13);
```

```
    addEdge(2, 1, 3, 9);
    addEdge(3, 1, 4, 23);
    addEdge(4, 2, 5, 4);
    addEdge(5, 2, 6, 25);
    addEdge(6, 3, 7, 29);
    addEdge(7, 6, 8, 5);
    addEdge(8, 7, 9, 30);
    addEdge(9, 8, 10, 1);
    addEdge(10, 8, 11, 6);
    /* our tree is rooted at node 0 at depth 0 */
    int root = 0, parent_of_root=-1, depth_of_root=0;
    /* a DFS on the tree to set up:
     * arrays for parent, depth, subtree size for every node;
     * deeper end of every Edge */
    dfs(root, parent_of_root, depth_of_root, n);
    int chain_heads[N];
    /*we have initialized no chain heads */
    memset(chain_heads, -1, sizeof(chain_heads));
    /* Stores number of edges for construction of segment
       tree. Initially we haven't traversed any Edges. */
    int edge_counted = 0;
    /* we start with filling the 0th chain */
    int curr_chain = 0;
    /* HLD of tree */
    hld(root, n-1, &edge_counted, &curr_chain, n, chain_heads);
    /* ST of segregated Edges */
    construct_ST(0, edge_counted, 1);
    /* Since indexes are 0 based, node 11 means index 11-1,
       8 means 8-1, and so on*/
    int u = 11, v  = 9;
    cout << "Max edge between " << u << " and " << v << " is ";
    maxEdge(u-1, v-1, n, chain_heads);
    // Change value of edge number 8 (index 8-1) to 28
    change(8-1, 28, n);
    cout << "After Change: max edge between " << u << " and "
         << v << " is ";
    maxEdge(u-1, v-1, n, chain_heads);
    v = 4;
    cout << "Max edge between " << u << " and " << v << " is ";
    maxEdge(u-1, v-1, n, chain_heads);
    // Change value of edge number 5 (index 5-1) to 22
    change(5-1, 22, n);
```

```
    cout << "After Change: max edge between " << u << " and "
         << v << " is ";
    maxEdge(u-1, v-1, n, chain_heads);
    return 0;
}
```

Output:

```
Max edge between 11 and 9 is 30
After Change: max edge between 11 and 9 is 29
Max edge between 11 and 4 is 25
After Change: max edge between 11 and 4 is 23
```

This article is contributed by **Yash Varyani**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.