Dynamic Connectivity

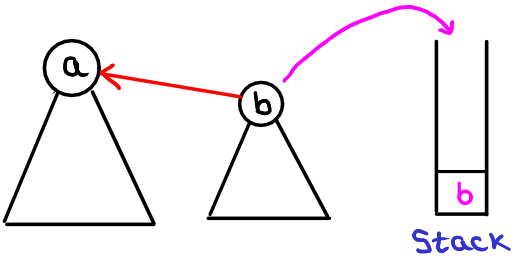
TEJAS KHAIRNAR

```cpp
// Consider an undirected graph that consists of n nodes and m edges.
// There are two types of events that can happen:
// A new edge is created between nodes a and b.
// An existing edge between nodes a and b is removed.
// Your task is to report the number of components after every event.

const int N = 2e5 + 5;
vector<pii> t[2 * N];

vector<int> queries; // Used to store the output times
map<pii, int> in;    // In time for the edges
int ans[N];

int parent[N], sz[N]; // DSU requirements
stack<int> st;        // Used for rolling back the actions

int cnt; // Number of components
int n, m, q, eu, ev;

void make()
{
    for (int i = 1; i <= n; i++)
    {
        parent[i] = i;
        sz[i] = 1;
    }
    cnt = n;
}

// No path compression as real parent is used in rollbacks
int find(int v)
{
    while (v != parent[v])
        v = parent[v];
    return v;
}

bool merge(int a, int b)
{
    a = find(a);
    b = find(b);
    if (a == b)
        return false;
    if (sz[a] < sz[b])
        swap(a, b);
    sz[a] += sz[b];
    parent[b] = a;
    st.push(b);
    cnt--;
    return true;
}

void rollback(int moment)
{
    while (st.size() > moment)
    {
        int curr = st.top();
        st.pop();
        sz[parent[curr]] -= sz[curr];
        parent[curr] = curr;
        cnt++;
    }
}

void dfs(int v, int l, int r)
{
    if (l > r)
        return;

    int moment = st.size();

    // Merge all the edges in the current node
    for (auto edge : t[v])
        merge(edge.first, edge.second);

    if (l == r)
    {
        ans[l] = cnt;
    }
    else
    {
        int mid = (l + r) / 2;
        dfs(2 * v, l, mid);
        dfs(2 * v + 1, mid + 1, r);
    }
    // Rollback once the dfs of the current node is done.
    rollback(moment);
}

void update(int v, int tl, int tr, int l, int r, int eu, int ev)
{
    // No overlap in the intervals
    if (l > tr || tl > r)
        return;
    if (l <= tl && r >= tr)
    {
        // Active range completely overlaps the segtree range
        t[v].pb({eu, ev});
        return;
    }
    int tm = (tl + tr) / 2;
    update(2 * v, tl, tm, l, r, eu, ev);
    update(2 * v + 1, tm + 1, tr, l, r, eu, ev);
}

void upd(int l, int r, int eu, int ev)
{
    update(1, 1, q + 1, l, r, eu, ev);
}
```

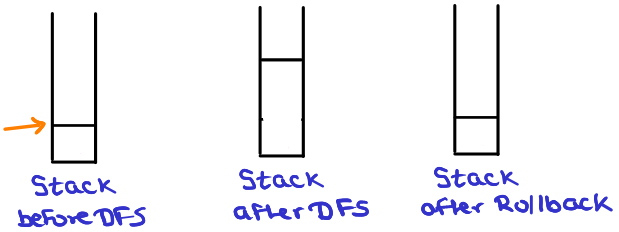
* If a edge (u,v) is added it means that (u,v) was added at first at some point of time.

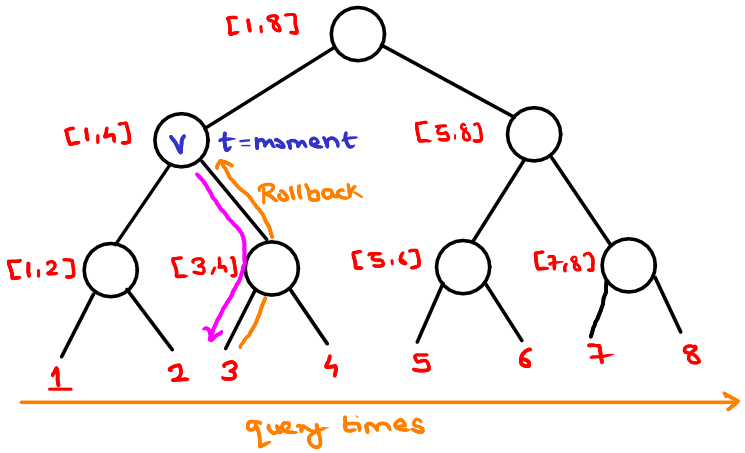
→ Initially n components
Increase at rollback
& Decrease at merge.

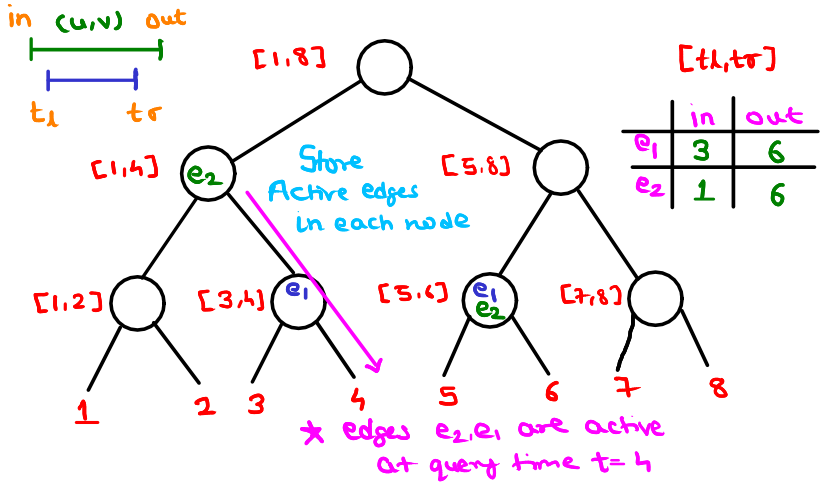
* Hence assign each edge (u,v) in time & out time



t=moment →







```cpp
int main()
{
    kira;

    cin >> n >> m >> q;
    make();

    for (int i = 0; i < m; i++)
    {
        cin >> eu >> ev;
        if (eu > ev)
            swap(eu, ev);
        in[{eu, ev}] = 1;
    }

    for (int i = 1; i <= q + 1; i++)
    {
        queries.pb(i);
    }

    int qt;
    for (int i = 2; i <= q + 1; i++)
    {
        cin >> qt >> eu >> ev;
        if (eu > ev)
            swap(eu, ev);
        if (qt == 1)
        {
            if (in.find({eu, ev}) != in.end())
                continue;
            in[{eu, ev}] = i;
        }
        else
        {
            upd(in[{eu, ev}], i - 1, eu, ev);
            in.erase({eu, ev});
        }
    }
    debug(in);
    for (auto edges : in)
    {
        upd(edges.S, q + 1, edges.F.F, edges.F.S);
    }
    dfs(1, 1, q + 1);
    for (auto x : queries)
        p0(ans[x]);
    run_time();
    return 0;
}
```

- In time of initial edges is taken as 1
- The edges which are not removed have their ending time as (q+1)