Deleting from a data structure in $O(T(n)\log n)$

Suppose you have a data structure which allows adding elements in **true** $O(T(n))$. This article will describe a technique that allows deletion in $O(T(n)\log n)$ offline.

Algorithm

Each element lives in the data structure for some segments of time between additions and deletions. Let's build a segment tree over the queries. Each segment when some element is alive splits into $O(\log n)$ nodes of the tree. Let's put each query when we want to know something about the structure into the corresponding leaf. Now to process all queries we will run a DFS on the segment tree. When entering the node we will add all the elements that are inside this node. Then we will go further to the children of this node or answer the queries (if the node is a leaf). When leaving the node, we must undo the additions. Note that if we change the structure in $O(T(n))$ we can roll back the changes in $O(T(n))$ by keeping a stack of changes. Note that rollbacks break amortized complexity.

Notes

The idea of creating a segment tree over segments when something is alive may be used not only for data structure problems. See some problems below.

Implementation

This implementation is for the dynamic connectivity problem. It can add edges, remove edges and count the number of connected components.

```cpp
struct dsu_save {
    int v, rnkv, u, rnku;

    dsu_save() {}

    dsu_save(int _v, int _rnkv, int _u, int _rnku)
        : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};

struct dsu_with_rollbacks {
    vector<int> p, rnk;
    int comps;
    stack<dsu_save> op;
```

```cpp
        dsu_with_rollbacks() {}

        dsu_with_rollbacks(int n) {
            p.resize(n);
            rnk.resize(n);
            for (int i = 0; i < n; i++) {
                p[i] = i;
                rnk[i] = 0;
            }
            comps = n;
        }

        int find_set(int v) {
            return (v == p[v]) ? v : find_set(p[v]);
        }

        bool unite(int v, int u) {
            v = find_set(v);
            u = find_set(u);
            if (v == u)
                return false;
            comps--;
            if (rnk[v] > rnk[u])
                swap(v, u);
            op.push(dsu_save(v, rnk[v], u, rnk[u]));
            p[v] = u;
            if (rnk[u] == rnk[v])
                rnk[u]++;
            return true;
        }

        void rollback() {
            if (op.empty())
                return;
            dsu_save x = op.top();
            op.pop();
            comps++;
            p[x.v] = x.v;
            rnk[x.v] = x.rnkv;
            p[x.u] = x.u;
            rnk[x.u] = x.rnku;
        }
};

struct query {
    int v, u;
    bool united;
    query(int _v, int _u) : v(_v), u(_u) {
    }
};

struct QueryTree {
    vector<vector<query>> t;
    dsu_with_rollbacks dsu;
    int T;

    QueryTree() {}
```

```cpp
        QueryTree(int _T, int n) : T(_T) {
            dsu = dsu_with_rollbacks(n);
            t.resize(4 * T + 4);
        }

        void add_to_tree(int v, int l, int r, int ul, int ur, query& q) {
            if (ul > ur)
                return;
            if (l == ul && r == ur) {
                t[v].push_back(q);
                return;
            }
            int mid = (l + r) / 2;
            add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
            add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
        }

        void add_query(query q, int l, int r) {
            add_to_tree(1, 0, T - 1, l, r, q);
        }

        void dfs(int v, int l, int r, vector<int>& ans) {
            for (query& q : t[v]) {
                q.united = dsu.unite(q.v, q.u);
            }
            if (l == r)
                ans[l] = dsu.comps;
            else {
                int mid = (l + r) / 2;
                dfs(2 * v, l, mid, ans);
                dfs(2 * v + 1, mid + 1, r, ans);
            }
            for (query q : t[v]) {
                if (q.united)
                    dsu.rollback();
            }
        }

        vector<int> solve() {
            vector<int> ans(T);
            dfs(1, 0, T - 1, ans);
            return ans;
        }
};
```

Problems

- Codeforces - Connect and Disconnect
- Codeforces - Addition on Segments
- Codeforces - Extending Set of Points