NANJING UNIVERSITY

ACM-ICPC Codebook 3

Data Structures

September 8, 2017

Contents

1 Range Operation Structures **4**
1.1 Binary indexed tree 4
1.1.1 Point update, range query 4
1.1.2 Range update, point query 4
1.1.3 Range update, range query 5

2 Miscellaneous Data Structures **6**
2.1 Union-find set 6
2.2 Sparse table, range extremum query (RMQ) 7

3 Tree **7**
3.1 Heavy-light decomposition 7

1 Range Operation Structures

1.1 Binary indexed tree

1.1.1 Point update, range query

Usage:

`init(n)`	Initialize the tree with 0.
`add(n, x)`	Add the n-th element by x.
`sum(n)`	Return the sum of the first n elements.

Time complexity: $O(n)$ for initialization; $O(\log n)$ for each update and query.

```
inline int lowbit(int x){return x&-x;}

struct bit_purq{ // point update, range query
    int N;
    vector<LL> tr;

    void init(int n){ // fill the array with 0
        tr.resize(N = n + 5);
    }

    LL sum(int n){
        LL ans = 0;
        while (n){
            ans += tr[n];
            n -= lowbit(n);
        }
        return ans;
    }

    void add(int n, LL x){
        while (n < N){
            tr[n] += x;
            n += lowbit(n);
        }
    }
};
```

1.1.2 Range update, point query

Usage:

`init(n)`	Initialize the tree with 0.
`add(n, x)`	Add the first n element by x.
`query(n)`	Return the value of the n-th element.

Time complexity: $O(n)$ for initialization; $O(\log n)$ for each update and query.

```
inline int lowbit(int x){return x&-x;}

struct bit_rupq{ // range update, point query
    int N;
    vector<LL> tr;

    void init(int n){ // fill the array with 0
        tr.resize(N = n + 5);
    }

    LL query(int n){
        LL ans = 0;
        while (n < N){
            ans += tr[n];
            n += lowbit(n);
        }
        return ans;
    }

    void add(int n, LL x){
        while (n){
            tr[n] += x;
            n -= lowbit(n);
        }
    }
};
```

1.1.3 Range update, range query

Usage:

`init(n)`	Initialize the tree with 0.
`add(l, r, x)`	Add the elements in $[l, r]$ by x.
`query(l, r)`	Return the sum of the elements in $[l, r]$.

Requirement:
1.1.1 Point update, range query

Time complexity: $O(n)$ for initialization; $O(\log n)$ for each update and query.

```
struct bit_rurq{
```

```
    bit_purq d, di;

    void init(int n){
        d.init(n); di.init(n);
    }

    void add(int l, int r, LL x){
        d.add(l, x); d.add(r+1, -x);
        di.add(l, x*l); di.add(r+1, -x*(r+1));
    }

    LL query(int l, int r){
        return (r+1)*d.sum(r) - di.sum(r) - l*d.sum(l-1) + di.sum(l-1);
    }
};
```

2 Miscellaneous Data Structures

2.1 Union-find set

Data structure for disjoint sets with path-compression optimization.

Usage:

`init(n)`	Initialize the sets from 0 to n, each includes one element.
`find(x)`	Return the representative of the set containing x.
`unite(u, v)`	Unite the two sets containing u and v. Return `false` if u and v are already in the same set; otherwise `true`.

```
struct ufs{
    vector<int> p;

    void init(int n){
        p.resize(n + 1);
        for (int i=0; i<n; i++) p[i] = i;
    }

    int find(int x){
        if (p[x] == x) return x;
        return p[x] = find(p[x]);
    }

    bool unite(int u, int v){
        u = find(u); v = find(v);
        p[u] = v;
```

```
        return u != v;
    }
};
```

2.2 Sparse table, range extremum query (RMQ)

Usage:

`ext(x, y)`	Return the extremum of x and y. **Modify this function before use!**
`init(n)`	Calculate the sparse table for array a from `a[0]` to `a[n-1]`.
`rmq(l, r)`	Query range extremum from `a[l]` to `a[r]`.

Time complexity: $O(n \log n)$ for initialization; $O(1)$ for each query.

```
const int MAXN = 100007;
int a[MAXN];
int st[MAXN][32 - __builtin_clz(MAXN)];

inline int ext(int x, int y){return x>y?x:y;} // ! max

void init(int n){
    int l = 31 - __builtin_clz(n);
    rep (i, n) st[i][0] = a[i];
    rep (j, l)
        rep (i, 1+n-(1<<j))
            st[i][j+1] = ext(st[i][j], st[i+(1<<j)][j]);
}

int rmq(int l, int r){
    int k = 31 - __builtin_clz(r-l+1);
    return ext(st[l][k], st[r-(1<<k)+1][k]);
}
```

3 Tree

3.1 Heavy-light decomposition

Usage:

`sz[x]`	Size of subtree rooted at x.
`top[x]`	Top node of the chain that x belongs to.
`fa[x]`	Father of x if exists; otherwise 0.
`son[x]`	Child node of x in its chain if exists; otherwise 0.
`depth[x]`	Depth of x. The depth of root is 1.
`id[x]`	Index of x used in data structure.
`decomp(r)`	Perform heavy-light decomposition on tree rooted at r.
`query(u, v)`	Query the path between u and v.

Time complexity: $O(n)$ for decomposition; $O(f(n)\log n)$ for each query, where $f(n)$ is the time-complexity of data structure.

```
const int MAXN = 100005;
vector<int> adj[MAXN];
int sz[MAXN], top[MAXN], fa[MAXN], son[MAXN], depth[MAXN], id[MAXN];

void dfs1(int x, int dep, int par){
    depth[x] = dep;
    sz[x] = 1;
    fa[x] = par;
    int maxn = 0, s = 0;
    for (int& c: adj[x]){
        if (c == par) continue;
        dfs1(c, dep + 1, x);
        sz[x] += sz[c];
        if (sz[c] > maxn){
            maxn = sz[c];
            s = c;
        }
    }
    son[x] = s;
}

int cid = 0;
void dfs2(int x, int t){
    top[x] = t;
    id[x] = ++cid;
    if (son[x]) dfs2(son[x], t);
    for (int& c: adj[x]){
        if (c == fa[x]) continue;
        if (c == son[x]) continue;
        else dfs2(c, c);
    }
}

void decomp(int root){
    dfs1(root, 1, 0);
    dfs2(root, root);
```

```
}

void query(int u, int v){
    while (top[u] != top[v]){
        if (depth[top[u]] > depth[top[v]]){
            // id[top[u]] to id[u]
            u = fa[top[u]];
        } else {
            // id[top[v]] to id[v]
            v = fa[top[v]];
        }
    }
    if (depth[u] > depth[v]){
        // id[v] to id[u]
    } else {
        // id[u] to id[v]
    }
}
```