NANJING UNIVERSITY

ACM-ICPC Codebook 3

Data Structures

August 14, 2018

Contents

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:
?? ??

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]]) swap(u, v);
        // id[top[u]] to id[u]
        u = fa[top[u]];
    }
    if (depth[u] > depth[v]) swap(u, v);
    // id[u] to id[v]
}
```

3.2 Order Statistics and Splay

⚠ Like `std::set`, this structure does not support multiple equivalent elements.

Usage:

See comments in code.

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
  rkt;
// null_tree_node_update

// SAMPLE USAGE
rkt.insert(x);          // insert element
rkt.erase(x);           // erase element
rkt.order_of_key(x);    // obtain the number of elements less than x
rkt.find_by_order(i);   // iterator to i-th (numbered from 0) smallest element
rkt.lower_bound(x);
rkt.upper_bound(x);
rkt.join(rkt2);         // merge tree (only if their ranges do not intersect)
rkt.split(x, rkt2);     // split all elements greater than x to rkt2
```

3.3 Segment Tree

```
#define lson 2*
#define rson 1+2*

const int MAXN = 100005;

```

```
LL a[MAXN];
LL ans[MAXN * 4], tag[MAXN * 4];

inline void push_up(int p) {
    ans[p] = ans[lson p] + ans[rson p];
}

inline void func(int p, int l, int r, LL x) {
    tag[p] += x;
    ans[p] += (r - l + 1) * x;
}

inline void push_down(int p, int l, int r) {
    int mid = (l+r) / 2;
    func(lson p, l, mid, tag[p]);
    func(rson p, mid+1, r, tag[p]);
    tag[p] = 0;
}

void build(int p, int l, int r) {
    tag[p] = 0;
    if (l == r) {
        ans[p] = a[l];
        return;
    }
    int mid = (l+r)/2;
    build(lson p, l, mid);
    build(rson p, mid+1, r);
    push_up(p);
}

void update(int ll, int rr, int l, int r, int p, LL x) {
    if (ll <= l && r <= rr) {
        ans[p] += (r - l + 1) * x;
        tag[p] += x;
        return;
    }
    push_down(p, l, r);
    int mid = (l + r) / 2;
    if (ll <= mid) update(ll, rr, l, mid, lson p, x);
    if (rr > mid) update(ll, rr, mid+1, r, rson p, x);
    push_up(p);
}

LL query(int ll, int rr, int l, int r, int p) {
    LL ret = 0;
    if (ll <= l && r <= rr) return ans[p];
```

```
    int mid = (l+r)/ 2;
    push_down(p, l, r);
    if (ll <= mid) ret += query(ll, rr, l, mid, lson p);
    if (rr > mid) ret += query(ll, rr, mid+1, r, rson p);
    return ret;
}
```

3.4 Persistent Array

Usage:

`build(l, r)`	Build an array with indices $[l, r)$. **Remember to rewind pos when reusing memory pool!**
`access(l, r, pos)`	Access the position with index `pos`.
`update(l, r, pos, val)`	Change the value at `pos` to `val`.

Time complexity: $O(\log n)$ per operation.

```
struct node {
  union {
    int value;
    struct {
      node *left, *right;
    };
  };

  void* operator new(size_t size);


  node *update(int l, int r, int pos, int val) const {
    node* a = new node;
    if (r > l + 1) {
      int mid = (l + r) / 2;
      if (pos < mid) {
        a->left = left->update(l, mid, pos, val);
        a->right = right;
      } else {
        a->left = left;
        a->right = right->update(mid, r, pos, val);
      }
    } else {
      a->value = val;
    }
    return a;
  }

```

```
  int access(int l, int r, int pos) const {
    if (r > l + 1) {
      int mid = (l + r) / 2;
      if (pos < mid) return left->access(l, mid, pos);
      else return right->access(mid, r, pos);
    } else {
      return value;
    }
  }
} nodes[30000000];

int pos = 0;
inline void* node::operator new(size_t size) {
  return nodes + (pos++);
}

int init[1 << 21];
static node* build(int l, int r) {
  node* a = new node;
  if (r > l + 1) {
    int mid = (l + r) / 2;
    a->left = build(l, mid);
    a->right = build(mid, r);
  } else {
    a->value = init[l];
  }
  return a;
}
```