
ACM TEMPLATE



UESTC_Jungle

Last build at October 23, 2018

Contents

1	Datastructure	2
1.1	Fenwick	2
1.2	BST in pb_ds	3
1.3	Segment Tree	3
1.4	Sparse Table	6
1.5	Treap	6
1.6	Splay	9

1 Datastructure

1.1 Fenwick

```

1  /* Fenwick Tree (Binary Indexed Tree), by Abreto <m@abreto.net>. */
2  #include <cstring>
3
4  using namespace std;
5
6  template <class T = int, int MAXN = 100001>
7  struct fenwick {
8      static inline int lowbit(int x) {
9          return (x&(-x));
10     }
11     int N;
12     T f[MAXN]; /* 1-based. */
13     fenwick(void):N(MAXN) {
14         init();
15     }
16     fenwick(int n):N(n) {
17         init();
18     }
19     void init(void) {
20         memset(f,0,sizeof(f));
21     }
22     void upd(int i, T dx) {
23         while(i <= N) {
24             f[i] += dx;
25             i += lowbit(i);
26         }
27     }
28     T sum(int i) {
29         T ret = 0;
30         while(i) {
31             ret += f[i];
32             i -= lowbit(i);
33         }
34         return ret;
35     }
36 };

1  /* Fenwick Tree (Binary Indexed Tree), by Abreto <m@abreto.net>. */
2
3  #define MAXN 100001
4  #define LOWBIT(x) ((x)&(-x))
5
6  int N;
7  int fen[MAXN];
8
9  void update(int i, int dx) {
10     while(i <= N) {
11         fen[i] += dx;
12         i += LOWBIT(i);
13     }
14 }
15
16 int sum(int i) {
17     int s = 0;
18     while(i > 0) {
19         s += fen[i];
20         i -= LOWBIT(i);
21     }
22     return s;
23 }

```

1.2 BST in pb_ds

```

1  /* Red-Black tree via pb_ds. */
2  #include<bits/stdc++.h>
3  #include<ext/pb_ds/assoc_container.hpp>
4  #include<ext/pb_ds/tree_policy.hpp>
5  using namespace __gnu_pbds;
6  using namespace std;
7  template <typename T>
8  using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
9
10 int main() {
11     ordered_set<int> s;
12     s.insert(1);
13     s.insert(3);
14     cout << s.order_of_key(2) << endl; // the number of elements in the s less than 2
15     cout << *s.find_by_order(0) << endl; // print the 0-th smallest number in s(0-based)
16 }

```

1.3 Segment Tree

```

1  /* Segment tree (Interval tree, range tree), by Abreto <m@abreto.net>. */
2
3  template <int STMAX = 1000000>
4  struct segment_tree {
5      struct node_t {
6          static inline node_t merge(node_t n1, node_t n2) {
7              node_t ans;
8              ans.l = n1.l;
9              ans.r = n2.r;
10             /* merge n1 and n2 to ans. */
11             return ans;
12         }
13
14         /* Data field */
15         int l,r;
16     } nodes[(STMAX+1)<<2];
17
18     struct lazy_t {
19         int marked; /* Optional */
20         /* lazy mark. */
21
22         lazy_t(void) {
23             clear();
24         }
25         void clear(void) {
26             marked=0;
27         }
28     } marks[(STMAX+1)<<2];
29
30     inline void maintain_leaf(int o, int idx) {
31         nodes[o].l = nodes[o].r = idx;
32         /* Operations to single elements ... */
33     }
34     inline void maintain(int o) {
35         nodes[o] = node_t::merge(nodes[o<<1], nodes[o<<1|1]);
36     }
37
38     /* Usage: build(1,1,n); */
39     void build(int o, int l, int r) { /* [l,r] */

```

```

40     if( r <= l ) {
41         maintain_leaf(o, l);
42     } else {
43         int mid = l+r>>1;
44         build(o<<1, l, mid);
45         build(o<<1|1, mid+1, r);
46         maintain(o);
47     }
48 }
49
50 /* Modify all elements in [l,r] */
51 void mark(lazy_t act, int o) {
52     /* do something .. */
53     marks[o].marked = 1;
54 }
55
56 /* Pass cached updates. */
57 void pushdown(int o) {
58     if( marks[o].marked ) {
59         mark(marks[o], o<<1);
60         mark(marks[o], o<<1|1);
61         marks[o].clear();
62     }
63 }
64
65 /* Do act on all elements in [L,R] */
66 void upd(int L, int R, lazy_t act, int o, int l, int r) {
67     if( L <= l && r <= R ) {
68         mark(act, o);
69     } else if (L <= R) {
70         int mid = (l+r)>>1;
71         pushdown(o);
72         if( L <= mid ) upd(L, R, act, o<<1, l, mid);
73         if( R > mid ) upd(L, R, act, o<<1|1, mid+1, r);
74         maintain(o);
75     }
76 }
77
78 node_t qry(int L, int R, int o, int l, int r) {
79     if(L <= l && r <= R)
80         return nodes[o];
81     else if (L <= R) {
82         int mid = (l+r)>>1;
83         pushdown(o);
84         if(R <= mid) return qry(L,R,o<<1,l,mid);
85         if(L > mid) return qry(L,R,o<<1|1,mid+1,r);
86         return node_t::merge(qry(L,R,o<<1,l,mid),qry(L,R,o<<1|1,mid+1,r));
87     }
88 }
89
90 int N;
91
92 segment_tree(void):N(STMAX) {}
93 segment_tree(int n):N(n) {}
94 void build(int n) {
95     N = n;
96     build(1,1,N);
97 }
98 void update(int L, int R, lazy_t act) {
99     upd(L,R,act,1,1,N);
100 }
101 node_t query(int L, int R) {
102     return qry(L,R,1,1,N);
103 }

```

```

104 |};

1  /* Segment tree (Interval tree, range tree), by Abreto <m@abreto.net>. */
2
3  #define MAXN    1000001
4
5  typedef struct {
6      int l,r;
7      /* Data field */
8  } node_t;
9
10 node_t merge(node_t n1, node_t n2) {
11     node_t ans;
12     ans.l = n1.l;
13     ans.r = n2.r;
14     /* merge n1 and n2 to ans. */
15     return ans;
16 }
17
18 typedef struct {
19     int marked; /* Optional */
20     /* lazy mark. */
21 } lazy_t;
22
23 int A[MAXN];
24 node_t nodes[MAXN<<2];
25 lazy_t marks[MAXN<<2];
26
27 void maintain_leaf(int o, int idx) {
28     nodes[o].l = nodes[o].r = idx;
29     /* Operations to single elements ... */
30 }
31 void maintain(int o) {
32     nodes[o] = merge(nodes[o<<1], nodes[o<<1|1]);
33 }
34
35 /* Usage: build(1,1,n); */
36 void build(int o, int l, int r) { /* [l,r] */
37     if( r <= l ) {
38         maintain_leaf(o, l);
39     } else {
40         int mid = l+r>>1;
41         build(o<<1, l, mid);
42         build(o<<1|1, mid+1, r);
43         maintain(o);
44     }
45     marks[o].marked = 0;
46 }
47
48 /* Modify all elements in [l,r] */
49 void mark(lazy_t act, int o) {
50     /* do something .. */
51     marks[o].marked = 1;
52 }
53
54 /* Pass cached updates. */
55 void pushdown(int o) {
56     if( marks[o].marked ) {
57         mark(marks[o], o<<1);
58         mark(marks[o], o<<1|1);
59         marks[o].marked = 0;
60     }
61 }
62

```

```

63 /* **DISCARDED** */
64 /* Set A[p]=v. Usage: modify(p, v, 1, 1, n);
65 void modify(int p, int v, int o, int l, int r)
66 {
67     if( r - l < 2 )
68     {
69         maintain_leaf(o, v);
70     } else {
71         int mid = (l+r)/2;
72         pushdown(o);
73         if( p <= mid ) modify(p, v, o*2, l, mid);
74         else modify(p, v, o*2+1, mid, r);
75         maintain(o);
76     }
77 }*/
78
79 /* Do act on all elements in [L,R] */
80 void update(int L, int R, lazy_t act, int o, int l, int r) {
81     if( L <= l && r <= R ) {
82         mark(act, o);
83     } else if (L <= R) {
84         int mid = (l+r)>>1;
85         pushdown(o);
86         if( L <= mid ) update(L, R, act, o<<1, l, mid);
87         if( R > mid ) update(L, R, act, o<<1|1, mid+1, r);
88         maintain(o);
89     }
90 }

```

1.4 Sparse Table

```

1 /* RMQ with Sparse Table, by Abreto <m@abreto.net>. */
2
3 int min(int a, int b) {
4     return (a<b)?a:b;
5 }
6
7 #define MAXN    100001
8 #define MAXLOG  32
9
10 int N;
11 int A[MAXN];    /* indexed from 0. */
12 int st[MAXN][MAXLOG];
13
14 void st_init() {
15     int i = 0, j = 0, t = 0;
16     for(i = 0; i < N; ++i) st[i][0] = A[i];
17     for(j = 1; (t=(1<<j)) <= N; ++j)
18         for(i = 0; (i+t-1) < N; ++i)
19             st[i][j] = min(st[i][j-1], st[i+(t>>1)][j-1]);
20     /* st(i,j) = min(st(i,j-1), st(i+2^(j-1),j-1)). */
21 }
22
23 int st_query(int l, int r) {
24     int k = 0;
25     while((1<<(k+1)) <= (r-l+1)) k++;
26     return min(st[l][k], st[r-(1<<k)+1][k]);
27 }

```

1.5 Treap

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define MAXN (2560000)
6
7 int __treap_mem[MAXN];
8 void init_treap_mem(void) {
9     for( int i = 1 ; i < MAXN ; i++ )
10         __treap_mem[i-1] = i;
11 }
12 int alloc_address(void) {
13     int ret = __treap_mem[0];
14     __treap_mem[0] = __treap_mem[ret];
15     return ret;
16 }
17 void free_address(int p) {
18     __treap_mem[p] = __treap_mem[0];
19     __treap_mem[0] = p;
20 }
21
22 typedef int key_t;
23 typedef int val_t;
24 struct treap {
25     key_t x;
26     val_t v;
27     int r; /* random priority */
28     int eq, s; /* number of equal ones, size of subtree (include root itself) */
29     treap *fa; /* point to its father */
30     treap *ch[2]; /* 0 for left child, 1 for right child. */
31
32     treap(void);
33     inline void maintain(void); /* update s */
34     inline void set_child(int d, treap *child);
35     inline int which(void); /* determine which child this is of its father */
36     inline int cmp(key_t ox); /* determine which child to insert ox */
37     treap *rotate(void); /* rotate this to its father, return this */
38 } treap_nodes[MAXN];
39
40 treap *new_treap(key_t x, val_t v, treap *f) {
41     treap *ret = treap_nodes + alloc_address();
42     ret->x = x;
43     ret->v = v;
44     ret->eq = ret->s = 1;
45     ret->fa = f;
46     ret->ch[0] = ret->ch[1] = NULL;
47 }
48 void free_treap(treap *p) {
49     free_address( p - treap_nodes );
50 }
51 void treap_clear(void) {
52     init_treap_mem();
53 }
54
55 treap::treap(void) {
56     r = rand();
57     eq = s = 0;
58     fa = ch[0] = ch[1] = NULL;
59 }
60 inline void treap::maintain(void) {
61     s = eq;
62     for( int i = 0 ; i < 2 ; i++ )
63         if( ch[i] )
64             s += ch[i]->s;

```



```

65 }
66 inline void treap::set_child(int d, treap *child) {
67     ch[d] = child;
68     maintain();
69     if( child ) child->fa = this;
70 }
71 inline int treap::which(void) {
72     if( NULL == fa ) return -1; /* this is not a child */
73     else return ( fa->ch[1] == this );
74 }
75 inline int treap::cmp(key_t ox) {
76     if( ox == x ) return -1; /* equal */
77     else return ( ox > x ); /* left less, right more */
78 }
79 treap *treap::rotate(void) {
80     if ( NULL == fa ) return this; /* no father, already global root. */
81     int d = which();
82     fa->set_child(d, ch[d^1]);
83     set_child(d^1, fa);
84     return this;
85 }
86
87 // — deprecated |
88 typedef int T;
89
90 struct node {
91     T v; /* value of this node */
92     int r; /* random priority */
93     int eq; /* the number of equal things */
94     int s; /* the size of subtree rooted at this */
95     node *ch[2]; /* 0 for left child, 1 for right child. */
96     node(void) {
97         r = rand();
98         ch[0] = ch[1] = NULL;
99     }
100     /* return where to insert x */
101     int cmp(T x) {
102         if(v == x) return -1;
103         else return (x < v) ? 0 : 1;
104     }
105     /* return 1 if this node is prior to other */
106     int pri(node *o) {
107         return (r > (o->r));
108     }
109     /* maintain the s field */
110     void maintain(void) {
111         s = eq;
112         if(NULL != ch[0]) s += ch[0]->s;
113         if(NULL != ch[1]) s += ch[1]->s;
114     }
115 };
116
117 /* move o to ch[d] of o->ch[d^1] */
118 void rotate(node *&o, int d) {
119     node *k = o->ch[d^1];
120     o->ch[d^1] = k->ch[d];
121     o->maintain();
122     k->ch[d] = o;
123     k->maintain();
124     o = k;
125 }

```

1.6 Splay

```

1  /* splay, by Abreto<m@abreto.net>. */
2
3  #ifndef NULL
4  #define NULL 0
5  #endif
6
7  struct node {
8      node *f, *ch[2];
9      int sz;
10     node(node *fa = NULL, node *lc = NULL, node *rc = NULL) {
11         f = fa;
12         ch[0] = lc;
13         ch[1] = rc;
14         maintain();
15     }
16     inline int szof(const int d) const {
17         return ch[d] ? ch[d]->sz : 0;
18     }
19     inline void maintain(void) {
20         sz = szof(0) + szof(1) + 1;
21     }
22     inline int which(void) {
23         if (NULL == f) return 0;
24         return (f->ch[1] == this); /* f[which()] == this */
25     }
26     inline node *setf(node *fa, int d = 0) {
27         f = fa;
28         if (f) {
29             f->ch[d] = this;
30             f->maintain();
31         }
32         return f;
33     }
34     inline node *setc(node *son, int d = 0) {
35         ch[d] = son;
36         if (son) son->f = this;
37         maintain();
38         return this;
39     }
40     /* rotate this to its fater, return this. */
41     inline node *rotate(void) {
42         if (f != NULL) {
43             node *ff = f->f;
44             int d = which(), fd = f->which();
45             setc(f->setc(ch[d ^ 1], d), d ^ 1);
46             setf(ff, fd);
47         }
48         return this;
49     }
50     /* splay this to child of target */
51     inline node *splay(node * const target = NULL) {
52         while (f != target) {
53             if (target != f->f) {
54                 ( (which() == f->which()) ? f : this )->rotate();
55             }
56             rotate();
57         }
58         return this;
59     }
60     /* 0-based rank */
61     inline node *get_k_th(unsigned k) {
62         node *p = this;

```

```

63     int rank;
64     while (k != (rank = (p->szof(0)))) {
65         if (k < rank) {
66             p = p->ch[0];
67         } else {
68             k -= (rank + 1);
69             p = p->ch[1];
70         }
71     }
72     return p->splay(f);
73 }
74 };

1  /* HDU 3487 – Play with Chain, by Abreto<m@abreto.net>. */
2  #include <bits/stdc++.h>
3
4  using namespace std;
5
6  #define MAXN    300300
7
8  int n, m;
9
10 #define LC(p)    ch[p][0]
11 #define RC(p)    ch[p][1]
12 #define TARGET(p) LC(RC(p))
13
14 int nodes;
15 int val[MAXN], ch[MAXN][2], fa[MAXN], sz[MAXN];
16 int rev[MAXN];
17
18 inline int new_node(int v, int f) {
19     int p = (++nodes);
20     val[p] = v;
21     fa[p] = f;
22     ch[p][0] = ch[p][1] = rev[p] = 0;
23     sz[p] = 1;
24     return p;
25 }
26 inline void maintain(int p) {
27     if (p) {
28         sz[p] = sz[LC(p)] + sz[RC(p)] + 1;
29     }
30 }
31 inline void make_child(int f, int d, int p) { /* make p the d-th ch of f */
32     ch[f][d] = p;
33     if (p) fa[p] = f;
34 }
35 inline void myrev(int p) {
36     if (p) {
37         rev[p] ^= 1;
38         swap(LC(p), RC(p));
39     }
40 }
41 inline void pushdown(int p) {
42     if (p && rev[p]) {
43         if (LC(p)) myrev(LC(p));
44         if (RC(p)) myrev(RC(p));
45         rev[p] = 0;
46     }
47 }
48 int build(int f = 0, int l = 0, int r = n+1) {
49     if (r < l) return 0;
50     if (l == r) return new_node(l, f);
51     int mid = l+r>>1;

```

```

52  int p = new_node(mid, f);
53  LC(p) = build(p, l, mid-1);
54  RC(p) = build(p, mid+1, r);
55  maintain(p);
56  return p;
57 }
58 inline int which(int p) { /* return 1 if p is a right child or 0 if p is a left
    child. */
59     return (RC(fa[p]) == p);
60 }
61 inline int rotate(int p) { /* rotate p to its father. [!] make sure p is not global
    root. */
62     int f = fa[p], ff = fa[f];
63     if(0 == f) return p; /* p is global root */
64     pushdown(f);
65     pushdown(p);
66     int d = which(p), df = which(f);
67     make_child(f, d, ch[p][d^1]);
68     make_child(p, d^1, f);
69     maintain(f);
70     maintain(p);
71     fa[p] = ff;
72     if(ff) ch[ff][df] = p;
73     return p;
74 }
75 inline int splay(int p, int fr) { /* splay p to the son of fr, return p. */
76     pushdown(p);
77     while(fa[p] != fr) {
78         int f = fa[p], dp = which(p);
79         if(fa[f] == fr) {
80             return rotate(p);
81         } else {
82             int df = which(f);
83             if(dp == df) {
84                 rotate(f);
85             } else {
86                 rotate(p);
87             }
88             rotate(p);
89         }
90     }
91     return p;
92 }
93 inline int get_k_th(int root, int k) {
94     int p = root;
95     int rank;
96     while(k != (rank = (sz[LC(p)] + 1))) {
97         pushdown(p);
98         if(k < rank) p = LC(p);
99         else {
100             k -= rank;
101             p = RC(p);
102         }
103     }
104     return splay(p, fa[root]);
105 }
106 inline int merge(int left, int right) {
107     pushdown(left);
108     if(RC(left)) left = get_k_th(left, sz[left]);
109     RC(left) = right;
110     maintain(left);
111     fa[right] = left;
112     return left;
113 }

```

```

114 inline int split(int root, int d) { /* split ch[root][d], return the root of splitted
    out. */
115     pushdown(root);
116     int child = ch[root][d];
117     ch[root][d] = 0;
118     maintain(root);
119     fa[child] = 0;
120     return child;
121 }
122 inline int concat(int root, int d, int p) { /* make p be ch[root][d], return root */
123     pushdown(root);
124     ch[root][d] = p;
125     fa[p] = root;
126     maintain(root);
127     return root;
128 }
129
130 void myclear(void) {
131     nodes = 0;
132 }
133
134 int ans[MAXN];
135 void inorder(int p, int &pos) {
136     if(0 == p) return;
137     pushdown(p);
138     inorder(LC(p), pos);
139     if( (0 < val[p]) && (val[p] < n+1) ) ans[pos++] = val[p];
140     inorder(RC(p), pos);
141 }
142
143 void handle() {
144     int i;
145     int root;
146     myclear();
147     root = build(0);
148     while(m--) {
149         char command[8];
150         int a, b, c;
151         int tar;
152         scanf("%s%d%d", command, &a, &b);
153         if('C' == command[0]) {
154             scanf("%d", &c);
155             root = get_k_th(root, a);
156             RC(root) = get_k_th(RC(root), b-a+2);
157             tar = split(RC(root), 0);
158             maintain(root);
159             root = get_k_th(root, c+1);
160             RC(root) = get_k_th(RC(root), 1);
161             RC(root) = concat(RC(root), 0, tar);
162             maintain(root);
163         } else {
164             root = get_k_th(root, a);
165             RC(root) = get_k_th(RC(root), b-a+2);
166             myrev(TARGET(root));
167         }
168     }
169     int pos = 0;
170     inorder(root, pos);
171     for(i = 0; i < n; i++) printf("%s%d", i ? "␣:" : "", ans[i]);
172     puts("");
173 }
174
175 int main(void) {
176     while( scanf("%d%d", &n, &m) && (n > 0) && (m > 0) )

```

```
177 |     handle();  
178 |     return 0;  
179 | }
```