



NANJING UNIVERSITY

ACM-ICPC Codebook 3
Data Structures

August 24, 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.

```

1 inline int lowbit(int x){return x&-x;}
2
3 struct bit_purq{ // point update, range query
4     int N;
5     vector<LL> tr;
6
7     void init(int n){ // fill the array with 0
8         tr.resize(N = n + 5);
9     }
10
11     LL sum(int n){
12         LL ans = 0;
13         while (n){
14             ans += tr[n];
15             n -= lowbit(n);
16         }
17         return ans;
18     }
19
20     void add(int n, LL x){
21         while (n < N){
22             tr[n] += x;
23             n += lowbit(n);
24         }
25     }
26 };
  
```

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.

```

1 inline int lowbit(int x){return x&-x;}
2
3 struct bit_rupq{ // range update, point query
4     int N;
5     vector<LL> tr;
6
7     void init(int n){ // fill the array with 0
8         tr.resize(N = n + 5);
9     }
10
11     LL query(int n){
12         LL ans = 0;
13         while (n < N){
14             ans += tr[n];
15             n += lowbit(n);
16         }
17         return ans;
18     }
19
20     void add(int n, LL x){
21         while (n){
22             tr[n] += x;
23             n -= lowbit(n);
24         }
25     }
26 };
  
```

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.

```

1 struct bit_rurq{
  
```

```

2   bit_purq d, di;
3
4   void init(int n){
5       d.init(n); di.init(n);
6   }
7
8   void add(int l, int r, LL x){
9       d.add(l, x); d.add(r+1, -x);
10      di.add(l, x*l); di.add(r+1, -x*(r+1));
11  }
12
13  LL query(int l, int r){
14      return (r+1)*d.sum(r) - di.sum(r) - l*d.sum(l-1) + di.sum(l-1);
15  }
16  };

```

2 Miscellaneous Data Structures

2.1 Union-find set

Data structure for disjoint sets with path-compression optimization.

Usage:

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

Time complexity: $O(n)$ for initialization; $O(\log n)$ for find and union.

```

1  struct ufs{
2      vector<int> p;
3
4      void init(int n){
5          p.resize(n + 1);
6          for (int i=0; i<n; i++) p[i] = i;
7      }
8
9      int find(int x){
10         if (p[x] == x) return x;
11         return p[x] = find(p[x]);
12     }
13
14     bool unite(int u, int v){

```

```

15     u = find(u); v = find(v);
16     p[u] = v;
17     return u != v;
18 }
19 };

```

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.

```

1  const int MAXN = 100007;
2  int a[MAXN];
3  int st[MAXN][32 - __builtin_clz(MAXN)];
4
5  inline int ext(int x, int y){return x>y?x:y;} // ! max
6
7  void init(int n){
8      int l = 31 - __builtin_clz(n);
9      rep (i, n) st[i][0] = a[i];
10     rep (j, l)
11         rep (i, 1+n-(1<<j))
12             st[i][j+1] = ext(st[i][j], st[i+(1<<j)][j]);
13 }
14
15 int rmq(int l, int r){
16     int k = 31 - __builtin_clz(r-l+1);
17     return ext(st[l][k], st[r-(1<<k)+1][k]);
18 }

```

3 Tree

3.1 Heavy-light decomposition

Usage:

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

```

1  const int MAXN = 100005;
2  vector<int> adj[MAXN];
3  int sz[MAXN], top[MAXN], fa[MAXN], son[MAXN], depth[MAXN], id[MAXN];
4
5  void dfs1(int x, int dep, int par){
6      depth[x] = dep;
7      sz[x] = 1;
8      fa[x] = par;
9      int maxn = 0, s = 0;
10     for (int& c: adj[x]){
11         if (c == par) continue;
12         dfs1(c, dep + 1, x);
13         sz[x] += sz[c];
14         if (sz[c] > maxn){
15             maxn = sz[c];
16             s = c;
17         }
18     }
19     son[x] = s;
20 }
21
22 int cid = 0;
23 void dfs2(int x, int t){
24     top[x] = t;
25     id[x] = ++cid;
26     if (son[x]) dfs2(son[x], t);
27     for (int& c: adj[x]){
28         if (c == fa[x]) continue;
29         if (c == son[x]) continue;
30         else dfs2(c, c);
31     }
32 }
33
34 void decomp(int root){
35     dfs1(root, 1, 0);
36     dfs2(root, root);

```



```
37 }
38
39 void query(int u, int v){
40     while (top[u] != top[v]){
41         if (depth[top[u]] > depth[top[v]]){
42             // id[top[u]] to id[u]
43             u = fa[top[u]];
44         } else {
45             // id[top[v]] to id[v]
46             v = fa[top[v]];
47         }
48     }
49     if (depth[u] > depth[v]){
50         // id[v] to id[u]
51     } else {
52         // id[u] to id[v]
53     }
54 }
```