quwsarohi@gmail.com Section: Data Structure Page : 1

**// Union Find Disjoint Set**

vector<int>u\_list, u\_set; // u\_list[x] contains the size of a set x u\_set[x] contains the root of x

int unionRoot(int n) { // Finds the root of a point/element

if(u\_set[n] == n) // If u\_set[n] == n, then n is the root of set

return n;

else // Else keep searching for root

return u\_set[n] = unionRoot(u\_set[n]);

}

int makeUnion(int a, int b) { // Takes two points in the same set, and return the root

int x = unionRoot(a); // Find the root of x and y

int y = unionRoot(b);

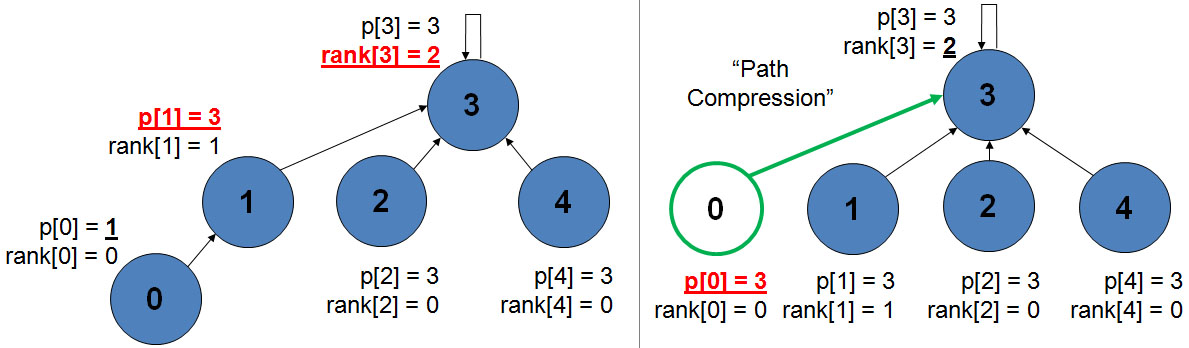
if(x == y) // If root of both points are same, then nothing to do

return x;

else if(u\_list[x] > u\_list[y]) { // If the size of x set is larger than set y (Path Compression)

u\_set[y] = x; // Make set y the subset of x

u\_list[x] += u\_list[y]; // Increase the size of set x

 return x;

}

else {

u\_set[x] = y;

u\_list[y] += u\_list[x];

return y;

} }

void unionInit(int len) { // Union Disjoint Set Initialization

u\_list.resize(len+5, 1); // Space allocation

u\_set.resize(len+5);

for(int i = 0; i <= len; i++) // At first, root of all points is the point itself

u\_set[i] = i;

}

bool isSameSet(int a, int b) { // Check if two points are in same set

if(unionRoot(a) == unionRoot(b)) // If the root of both point are same, then they are same

return 1;

return 0;

}

**// Segment Tree**

int arr[N], tree[4\*N]; // Always take the tree size 4x

// arr[] contains values starting from index 1

void segment\_build(int pos, int L, int R) { // Builts the segment tree call : segment\_build(1, 1, len\_of\_arr)

tree[pos] = 0; // Initialization of tree position

if(L==R) { // Mid point reached

tree[pos] = arr[L];

quwsarohi@gmail.com Section: Data Structure Page : 2

return; }

int mid = (L+R)/2;

segment\_build(pos\*2, L, mid);

segment\_build(pos\*2+1, mid+1, R);

tree[pos] = tree[pos\*2] + tree[pos\*2+1]; // Depends on usage (This is the main point to tweak)

}

void segment\_update(int pos, int L, int R, int i, int val) { // Val contains the value to update **SINGLE UPDATE**

if(L==R) { // Call: segment\_update(1, 1, len\_of\_arr, pos\_in\_arr new\_value)

tree[pos] = val; // If L==R then this is the midpoint that contains the single value

return; }

int mid = (L+R)/2;

if(i <= mid) segment\_update(pos\*2, L, mid, i, val); // Go for update\_position

else segment\_update(pos\*2+1, mid+1, R, i, val);

tree[pos] = tree[pos\*2] + tree[pos\*2+1]; // Depends on usage (here summation)

}

**// Query in range l-r**

int segment\_query(int pos, int L, int R, int l, int r) { // Finds value in l-r segment of arr[]

if(R < l || r < L) // Out of range l-r

return 0;

if(l <= L && R <= r) // In range l-r

return tree[pos];

int mid = (L+R)/2;

int x = segment\_query(pos\*2, L, mid, l, r);

int y = segment\_query(pos\*2+1, mid+1, R, l, r);

return x+y; // Return the total value

}

**// Toggle bit in range of [l, r]** call : update(1, 1, length\_of\_input, l, r) EX: Update lights in range l-r

void update(int pos, int L, int R, int l, int r) { **// Lazy without propagation**

if(l <= L && R <= r) { // If segment is in range l-r

arr[pos] ^= 1; // Tweak according to problem

return; }

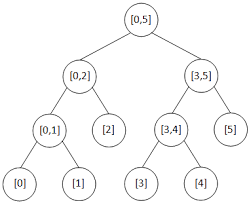
if(r < L || R < l) // Out of range l-r

return;

int mid = (L+R)/2;

update(pos\*2, L, mid, l, r);

update((pos\*2) + 1, mid+1, R, l, r);

}

**// Single position query**

bool query(int at, int L, int R, int pos) { // Query in arr position ‘pos’  
 if(pos < L || R < pos) return 0; // Range is out of pos

if(L <= pos && pos <= R) return arr[at]; // pos is in range

int mid = (L+R)/2;

if(pos < = mid)

return query(at\*2, L, mid, pos) ^ arr[at];

else

return query(at\*2+1, mid+1, R, pos) ^ arr[at];

}