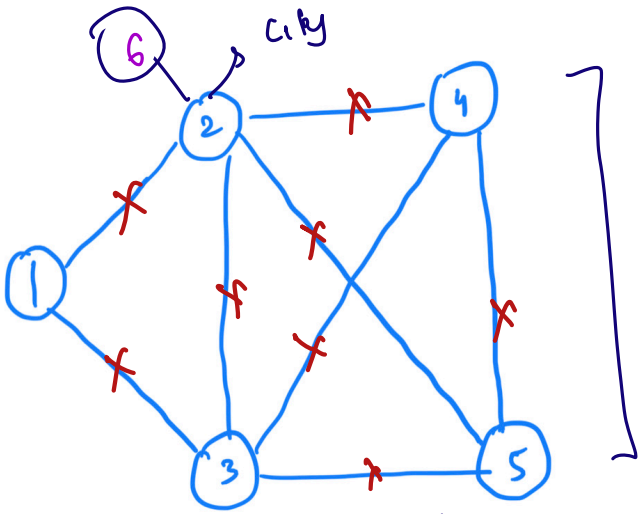
GRAPHS - 4 [Minimum Spanning Tree]

Agenda

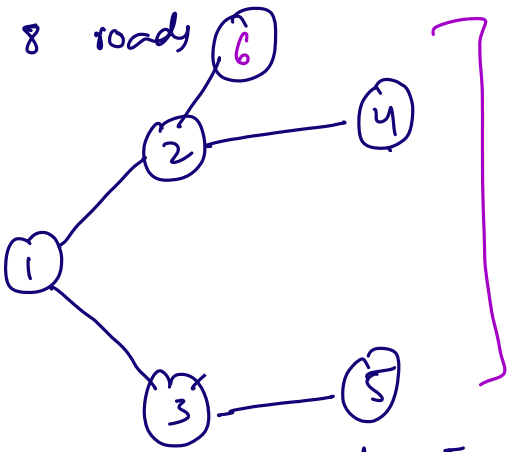
1) Prim's Algo
2) Kruskal' Algo
3) Disjoint set Union.

Spanning Tree



Map of a state

V = 5 cities
E = 8 roads



roads = 5

Tree

Special kind of graph

TREE

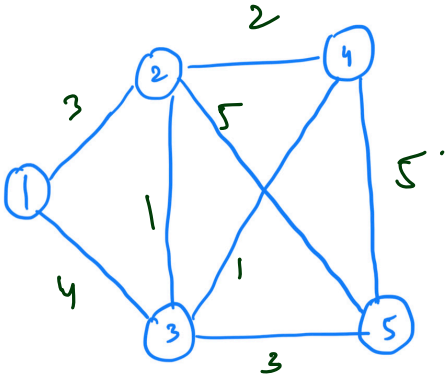
Min no. of edges ?

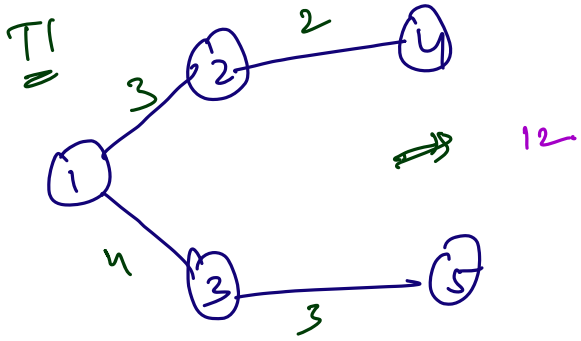
V - 1 edges

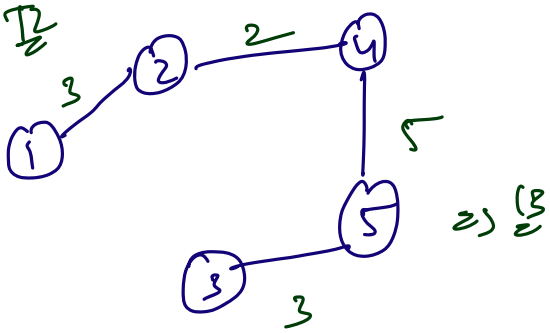
Spanning Tree

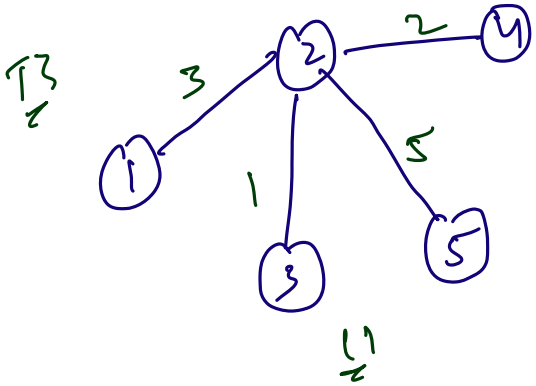
convert graph to a tree ⇒ spanningTree

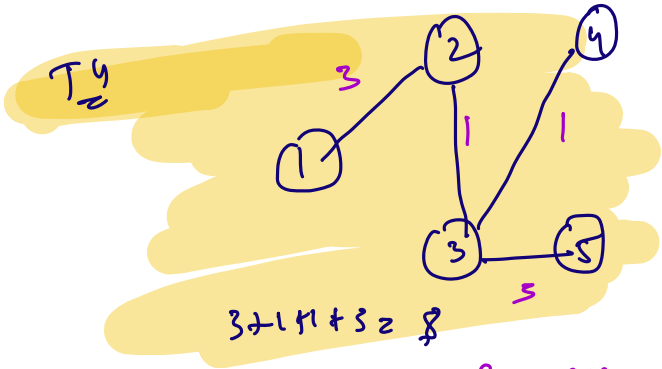
Minimum Spanning Tree



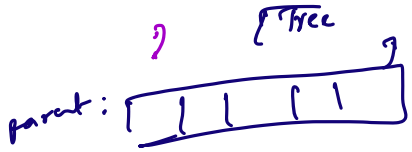


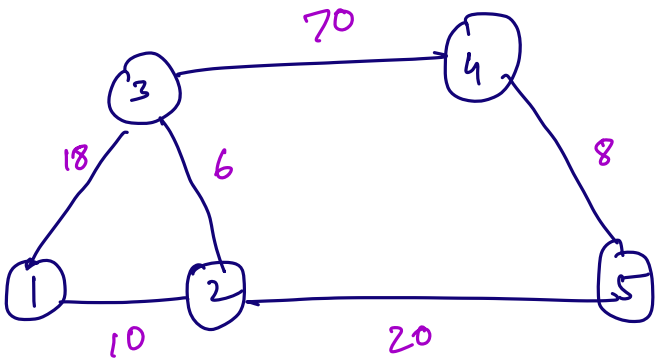


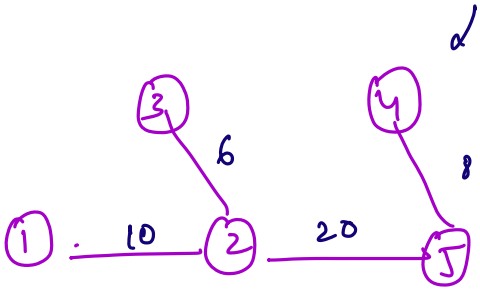




Among all the spanning tree, the spanning tree with minimum wt is called Minimum spanning Tree

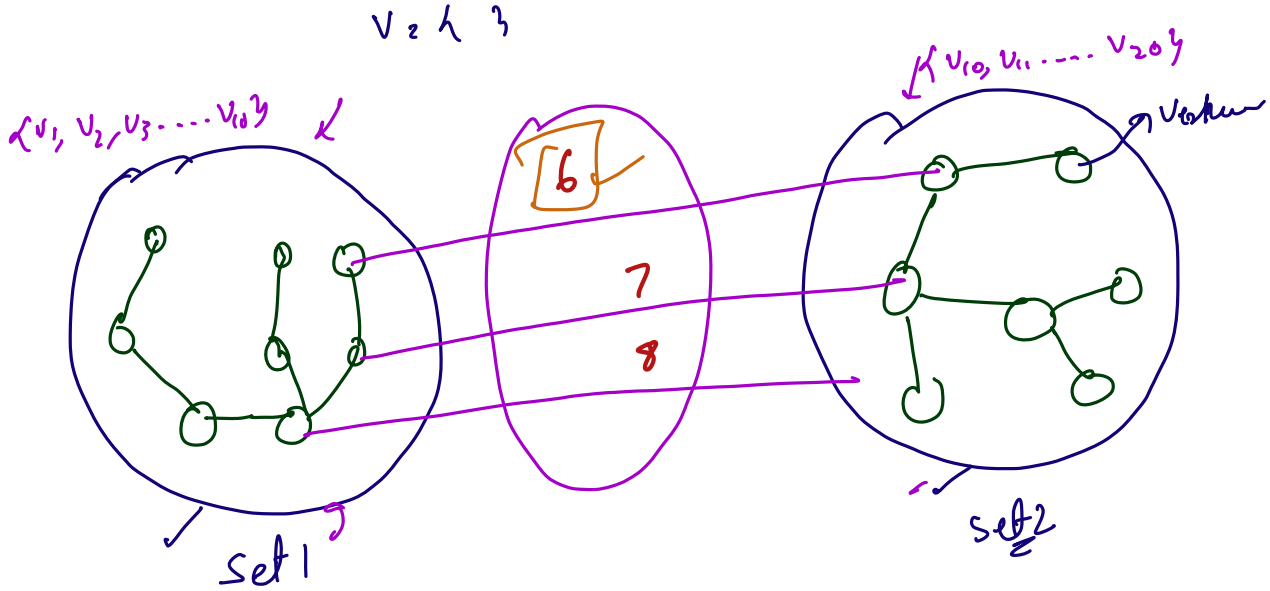






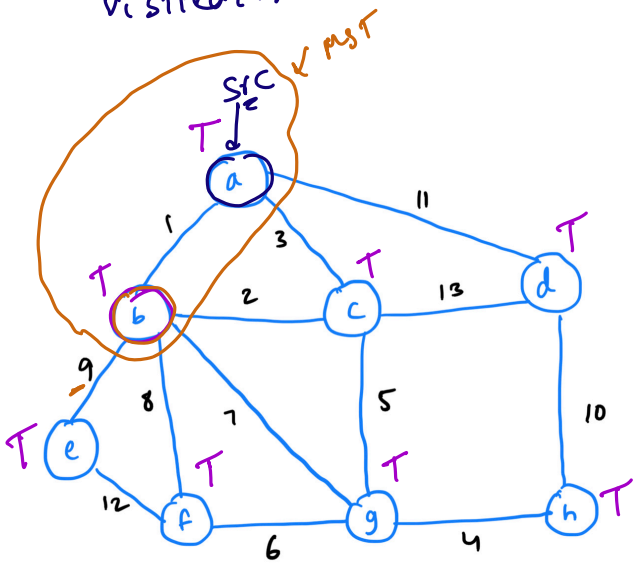
Cut in a graph

A cut is a partition of the vertices of graph into 2 disjoint subsets.

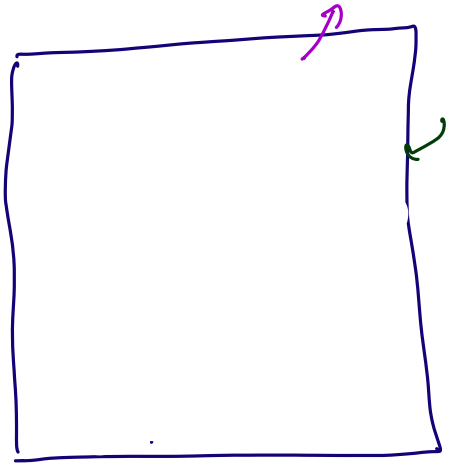


Prim's Algorithm [Greedy Algo]

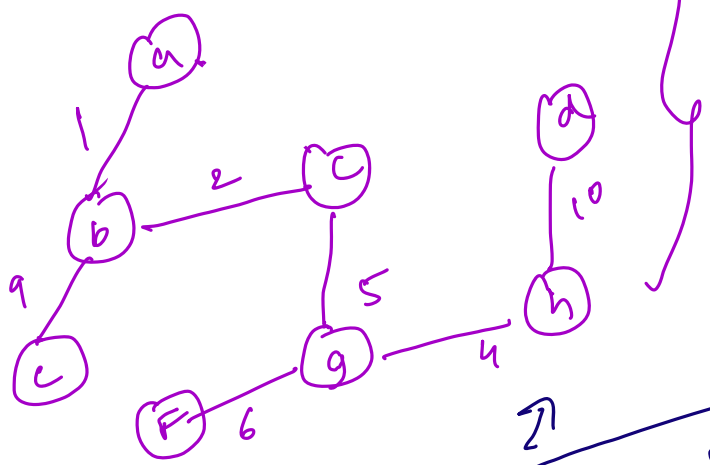
visited[i] denotes that vertex 'i' has included to MST



Min-Heap



9, 8, 7, 2, 3, 11



→ Last

C : { a, b, d, g }

any +

Minimum spanning tree

```
void prims(int N, adj[]){
    vis[N] = False;
    minHeap(pair<int, int>) pq;
    pq.push({0, 0});
    int ans = 0;

    while(!q.empty()){
        key, u = pq.top();
        pq.pop();
        if(vis[u])
            continue;
        ans += key;
        vis[u] = true;
        for(auto v : adj[u]){
            if(!visited[v]){
                pq.push({W(u, v), v});
            }
        }
    }
}
```

(E-V)

V times

→ add to cut

(vertex, Edge weight)

pq.push({src, 0});

Size of Heap: E

T.C: O(E log E)

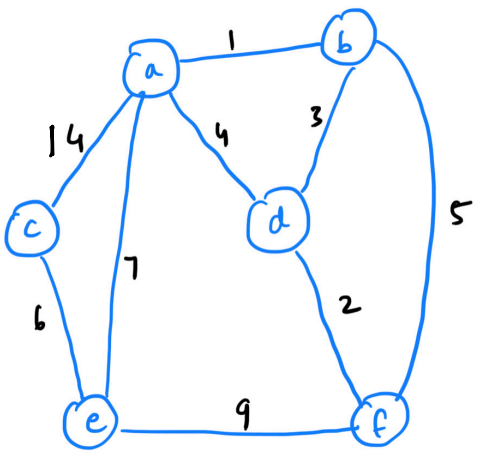
↓

O(E log V) ?

[Own implementation of heap using decrease key Method]

Kruskal's Algorithm [Greedy Algo]

1) Sort the edges in increasing order
2) If choosing an edge is forming a cycle, then ignore it.
3) Else add that edge to the MST.

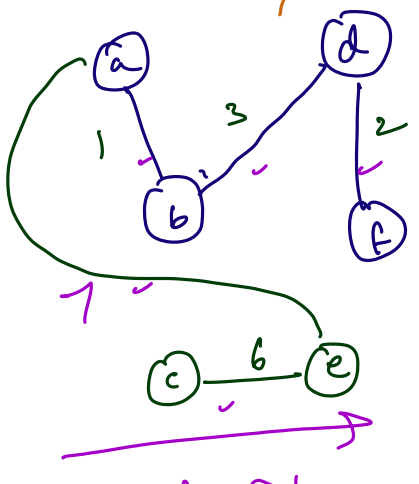


Sorted Edges

- ab, 1 ✓
- df, 2 ✓
- bd, 3 ✓
- ad, 4 ✗
- bf, 5 ✗
- ce, 6 ✓
- ae, 7 ✓
- ef, 9 ✗
- ac, 14

E

(V-1) edges



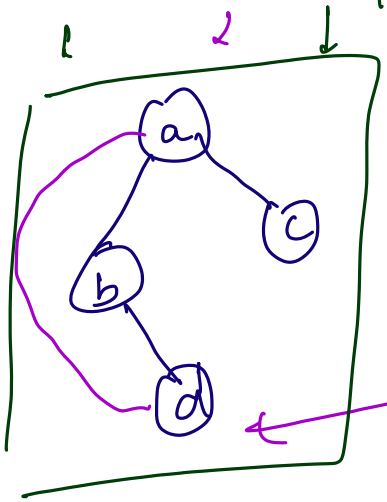
5

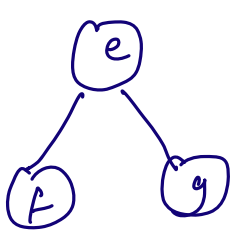
Min Spanning Tree

Cost = 12 + 7 = 19

→ Tricky Part: How do check if it forms a cycle or not.

Tree





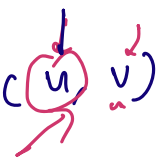
(d, e)

a - d

→ If (u, v) belong to the same component, then cycle

→ If (u, v) belong to different component then no cycle

How to check if 2 vertices belong to same component?



↓

O(V+E) }

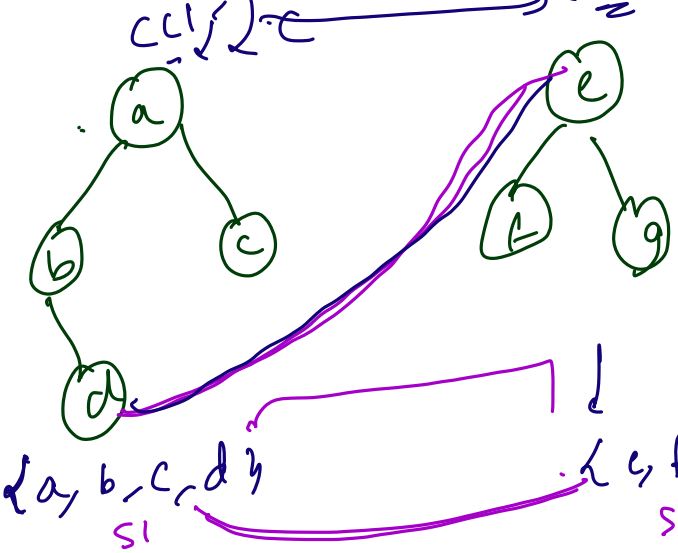
#times = E

T.C: O(E(V+E)) = O(E·V + E²)

Can we do better?

Disjoint Set Union

→ Consider vertices in a connected component to belong to one set



[d - c]

set(d) = s1 ✓
set(c) = s2 ✓

Find(u)
Find(v)

Operations

1) Find: Returns the id (set) which a vertex belong.
2) Union: Merges two sets

```
// sort Edges in Increasing order.

cost = 0;
for(i=0; i<E; i++){                          O(E)
    u,v = Edges[i];
    if( find(u) != find(v) ){
        cost += Edge weight.
        Union(u and v)        // merge the sets.
    }                          O(V)
}
```

T.C: $O(E \log E) + E\,O(Find) + V.O(Union)$

Find: 2*E

Union: V-1

T.C1: $O(E \log E + E \cdot V + V \cdot V)$

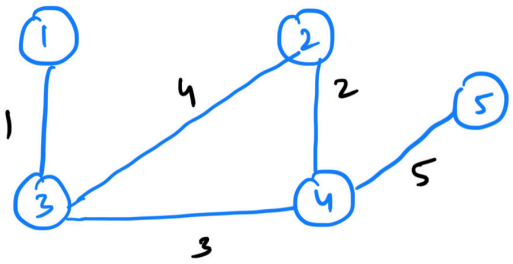
$= O(E \log E + E \cdot V + V^2)$

As worse as Brute force

TC2: $E \log E + E.\log V + V.\log V$

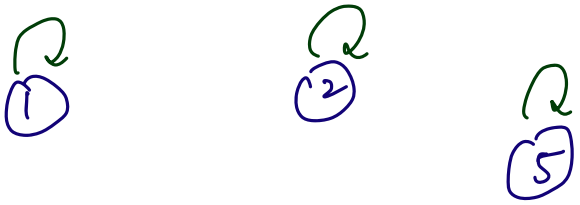
Implementation of DSU [Find & Union Functions]

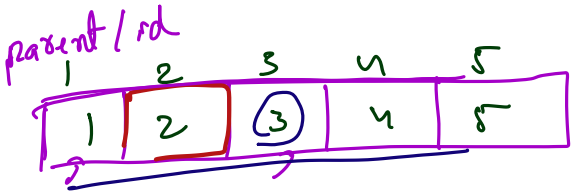
→ Each component is denoted by a set
→ Initially, there are ✓ Connected Comp [sets]



Edge
1-3, 1
2-4, 2
3-4, 3
2-3, 4
4-5, 5

parent[i] = i

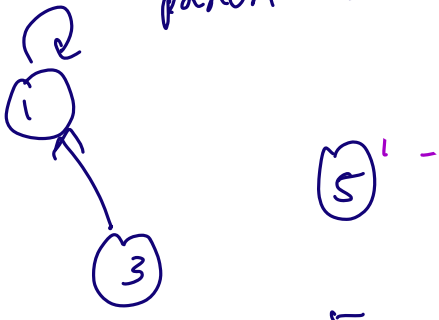


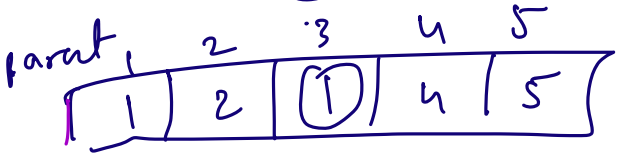


1) Edge 1-3

Way 1 ✓

parent[3] = 1

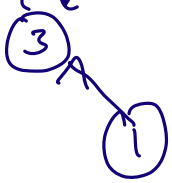


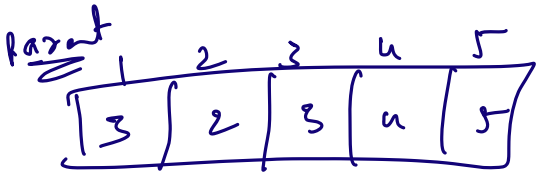


Edge 1-3

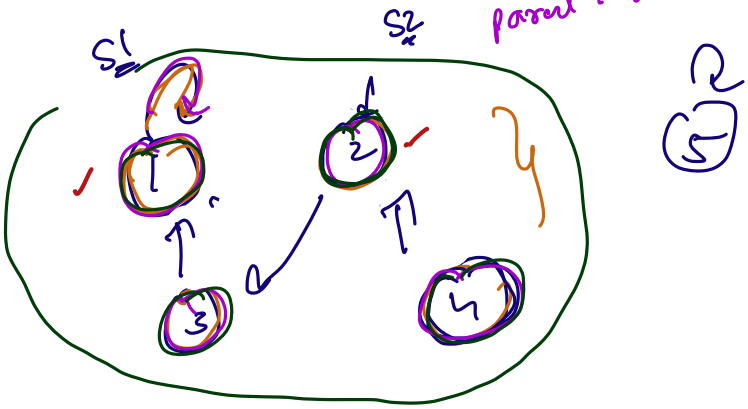
Way 2

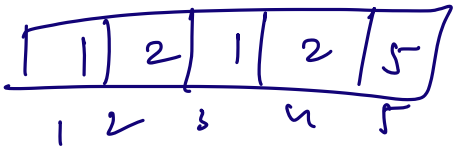
parent[1] = 3





Edge: 2 - 4





u == v
parent[u] = v
or
parent[v] = a

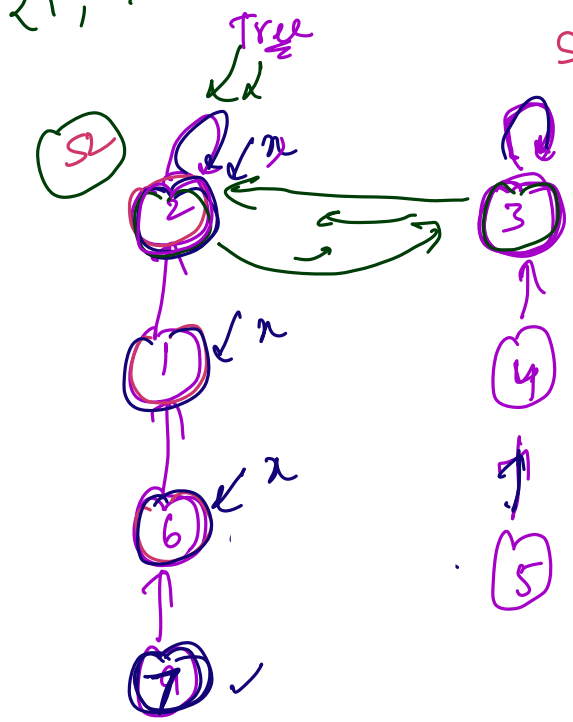
Edge: 3 - 4

Edge, u, v

Union(find(u), find(v))

parent[a] =>

{1, 2, 3, 4, 5}

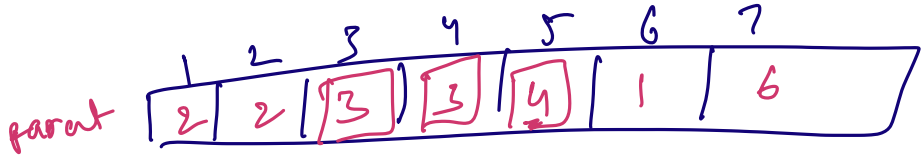


Edge (6, 5)

(6, 5)

parent[3] = 2

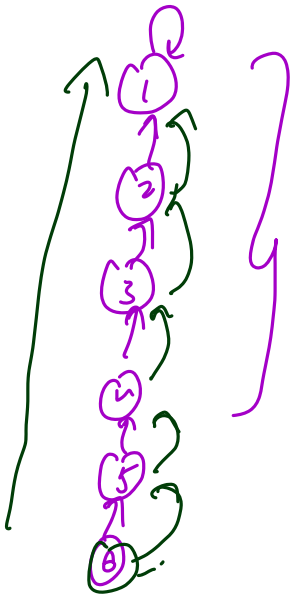
parent[2] = 2




```
int find (x) {
    while (a != parent[x]) {
        x = parent[x];
    }
    return x;
}
```

T.C: O(V)

1 - 2
2 - 3
3 - 4
4 - 5
5 - 6



O(V)

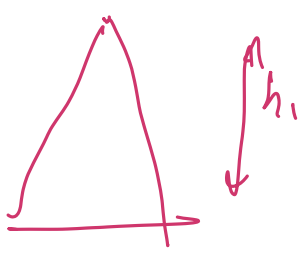
```
void Union(int u, int v) {
    x = find(u)  -> O(V)
    y = find(v)  -> O(V)
    parent[x] = y   -> O(1)
    // or  parent[y] = x
}
```

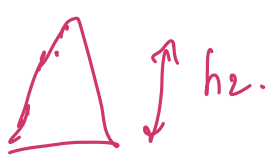
T.C. : O(V)

Union by Height [union by size] [Rank]

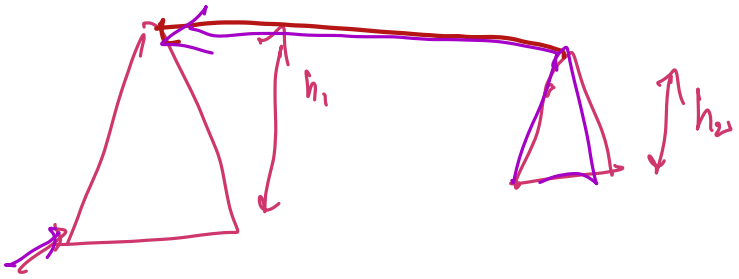
→ Tweak the union Function

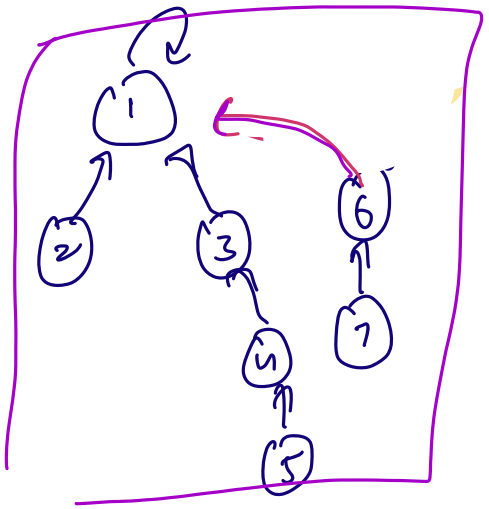
→ We want to Union 2 trees





Case 1 : $h_1 > h_2$

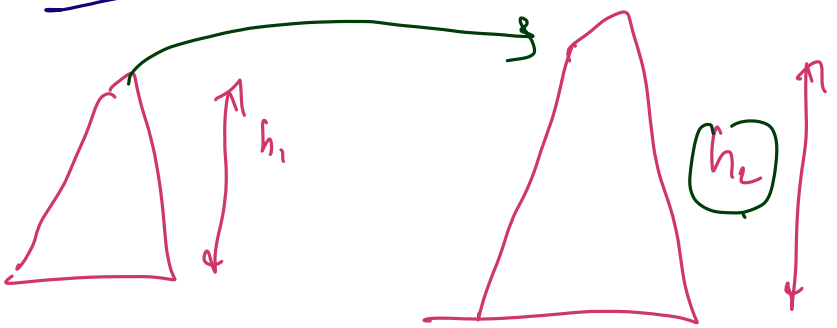




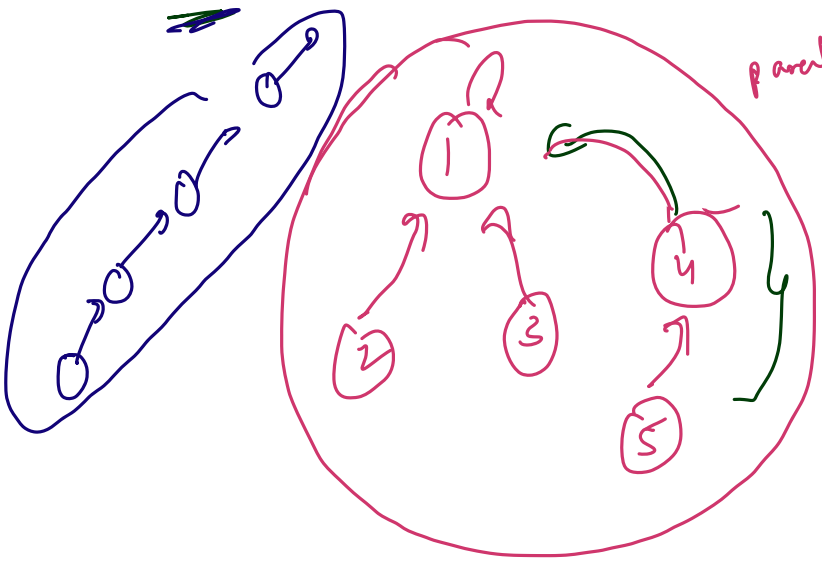
height is still h_1

(4, 7)

Case 2 : $h_1 < h_2$



Case 3 : $h_1 == h_2$



parent[u] = 1

$h_1 == h_2 == 1$

$h_1 + 1$

Increase by 1

Union by Height:

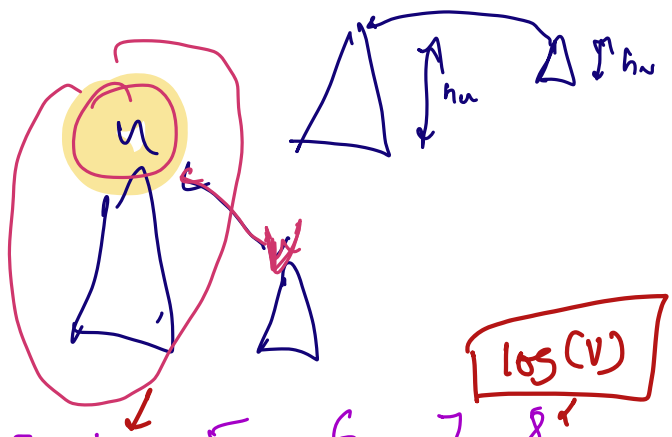
height:

```
union(int x, int y){
    u = find(x);
    v = find(y);
    if(height[u] > height[v])
        parent[v] = u;
    else if(height[v] > height[u])
        parent[u] = v;
    else{
        parent[v] = u;
        height[u] += 1;
    }
    return;
}
```

O(logV) { O(1) } O(1)

Find which set

$h_u > h_v$



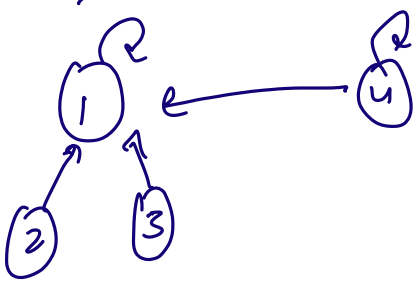
log(V)

Nodes: 0 1 2 3 4 5 6 7 8 ... N

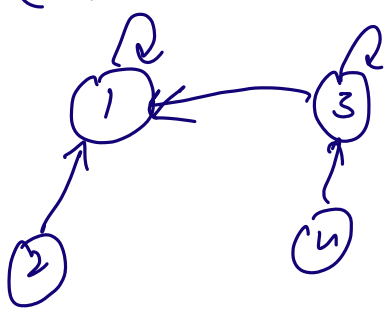
Height: 0 0 1 1 2 2 2 2 3 ... log(N)

N = 4

(3, 1)

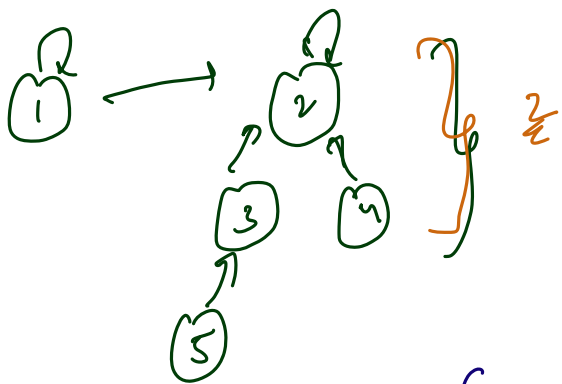


(2, 2)

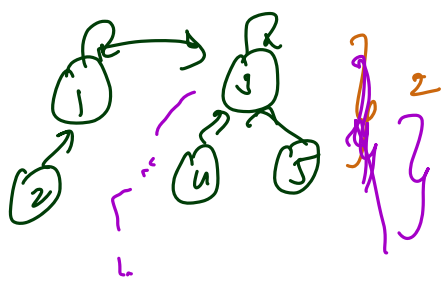


N = 5

(1, 4)



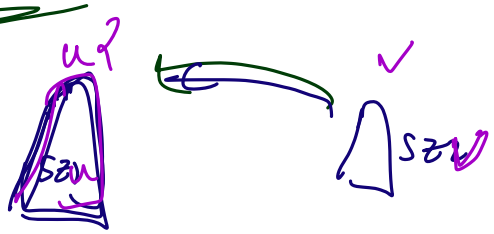
(2, 3)



T-C of Find : O(log(V))

T-C of Union : O(log V)

Union by size:



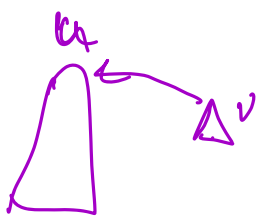
Size: No. of nodes

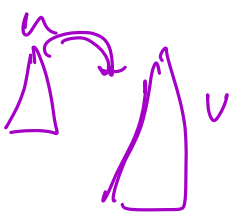
```
if ( size(u) > size(v) ) {
    parent[v] = u ;
    size[u] += size[v];
}
if( size(v) > size(u) ) {
    parent[u] = v;
    size[v] += size[u];
}
else {
}
```

O(log V) →

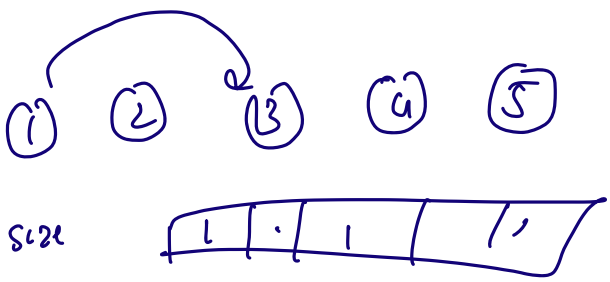
```
int find(int x){
    ...
}

union(int x, int y){
    u = find(x);
    v = find(y);
    if(size[u] > size[v]){
        parent[v] = u;
        size[u] += size[v];
    }
    else{
        parent[u] = v;
        size[v] += size[u];
    }
return;
}
```

parent



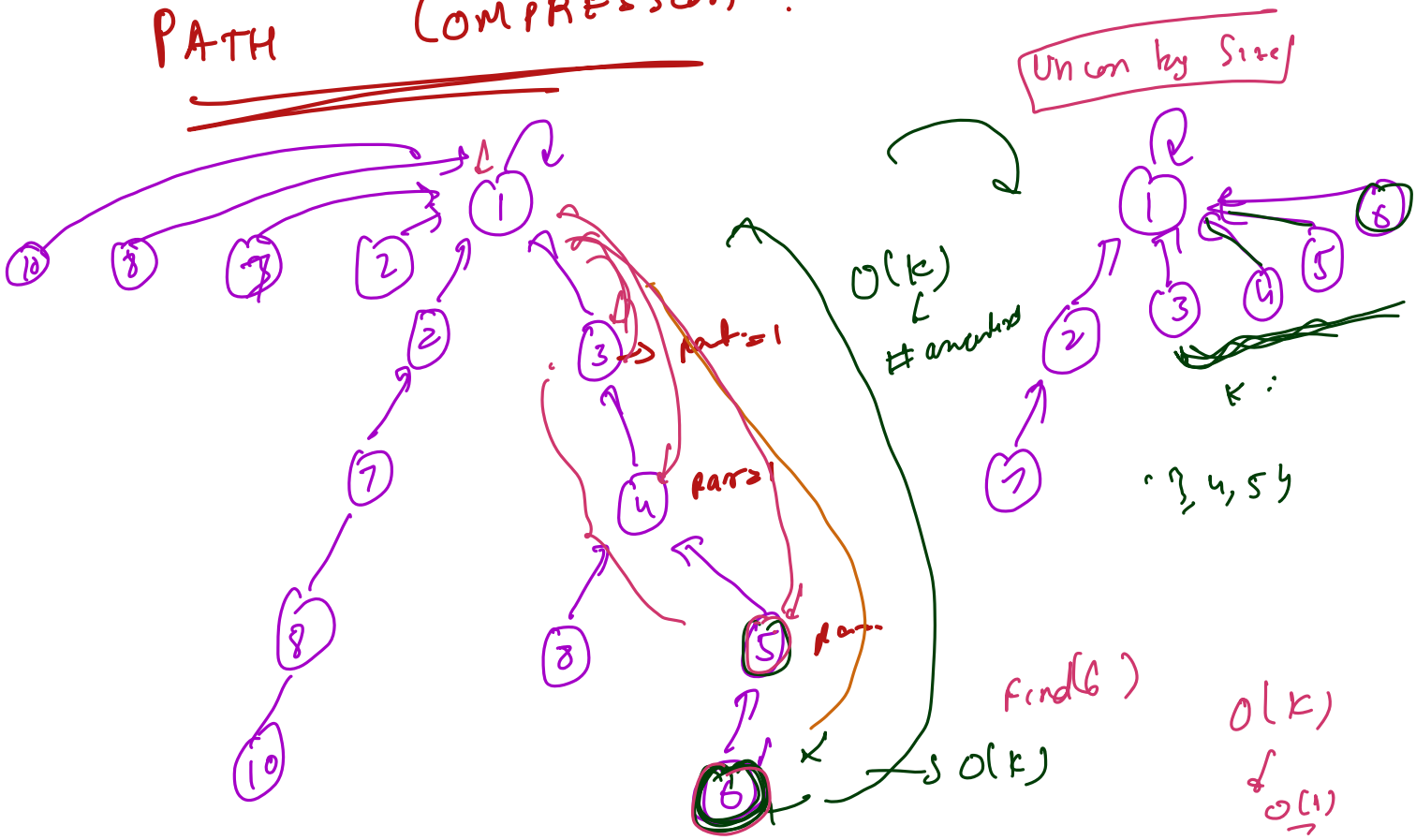


parent[v] = u



```
while( x != parent[x] ) {
    x = parent[x];
}
return x;
```

PATH COMPRESSION :



find(5) = 5 → 4 → 3 → 2 → 1

find(6) = 6 → 5 → 4 → 3 → 2 → 1

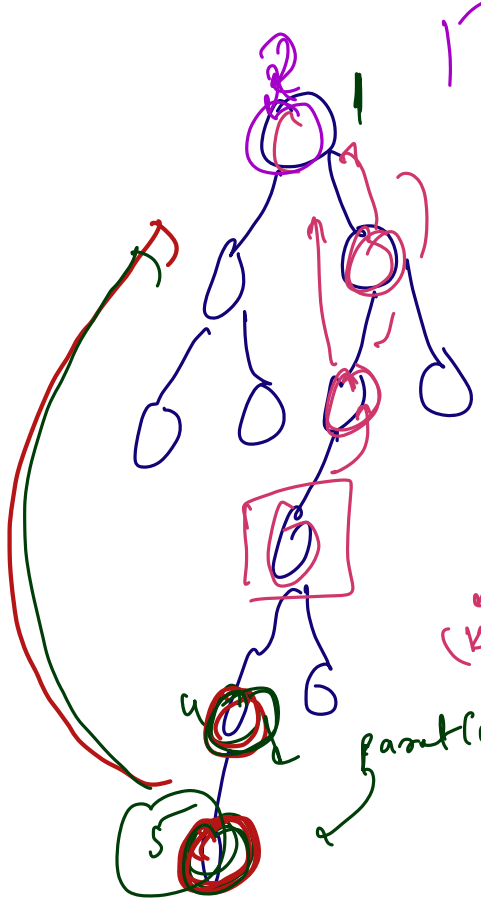
T.C : **O(K)**

K is the no. of ancestors

for remaining k ancestors, T.C: O(1)

k nodes: O(K) + K * O(1) => O(2K) => O(1)

Amortized T.C of Find Function: O(1)



find(n)

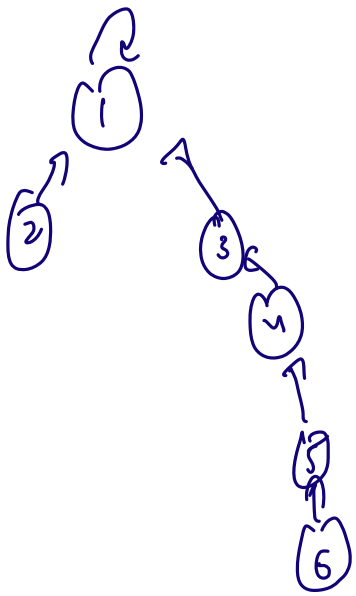
```
int find(x) {
    if(x == parent[x]) { return x; }
    parent[x] = find(parent[x]);
    return parent[x];
}
```

Dry Run

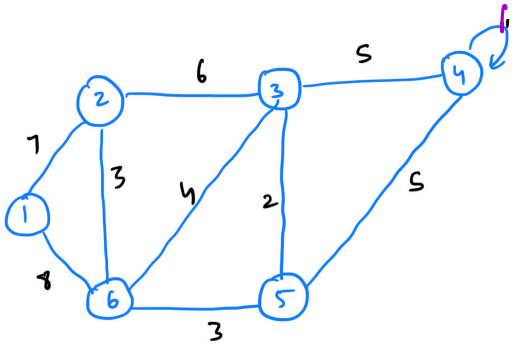
T.C: $E \log E + E \cdot O(\text{Find}) + V \cdot O(\text{Union})$

$O(1)$ $O(1)$

T.C: $E \log E$

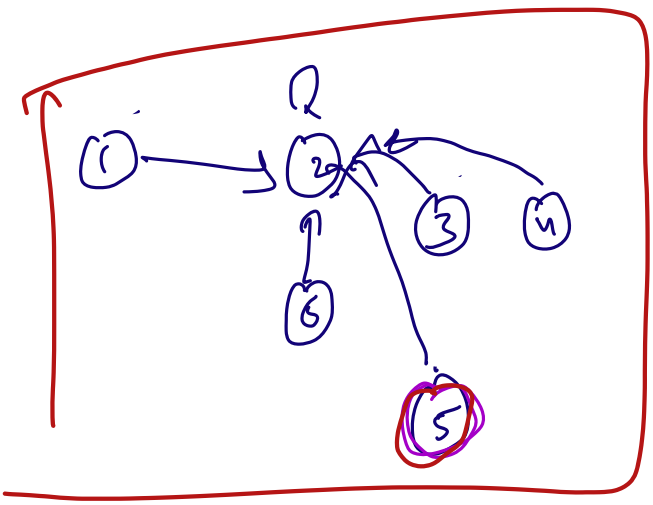


Dry Run



Cost = 2 + 3 + 3 + 5 + 7

4-4, 1 ✗
3-5, 2 ✓
2-6, 3 ✓
5-6, 3 ✓
3-6, 4 ✗
3-4, 5 ✓
4-5, 5 ✗
2-3, 6 ✗
1-2, 7 ✓
1, 6 8 ✗



parent[i] = 2

Code

Path Compression

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

Union by size

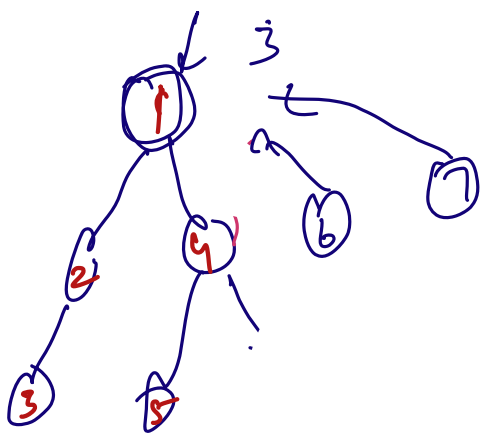
```
union(int x, int y){
      u = find(x);
      v = find(y);
      if(size[u] > size[v]){
            parent[v] = u;
            size[u] += size[v];
      }
      else{
            parent[u] = v;
            size[v] += size[u];
      }
return;
}
```

```
int kruskal(){
      ans = 0;
      sort(Edges);
      for(int i = 1 to E){
            ei = (u, v)
            if(find(u) != find(v)){
                  ans += W(ei);
                  Union(u, v);
            }
      }
}
```

S.C: Parent Array + Size Array

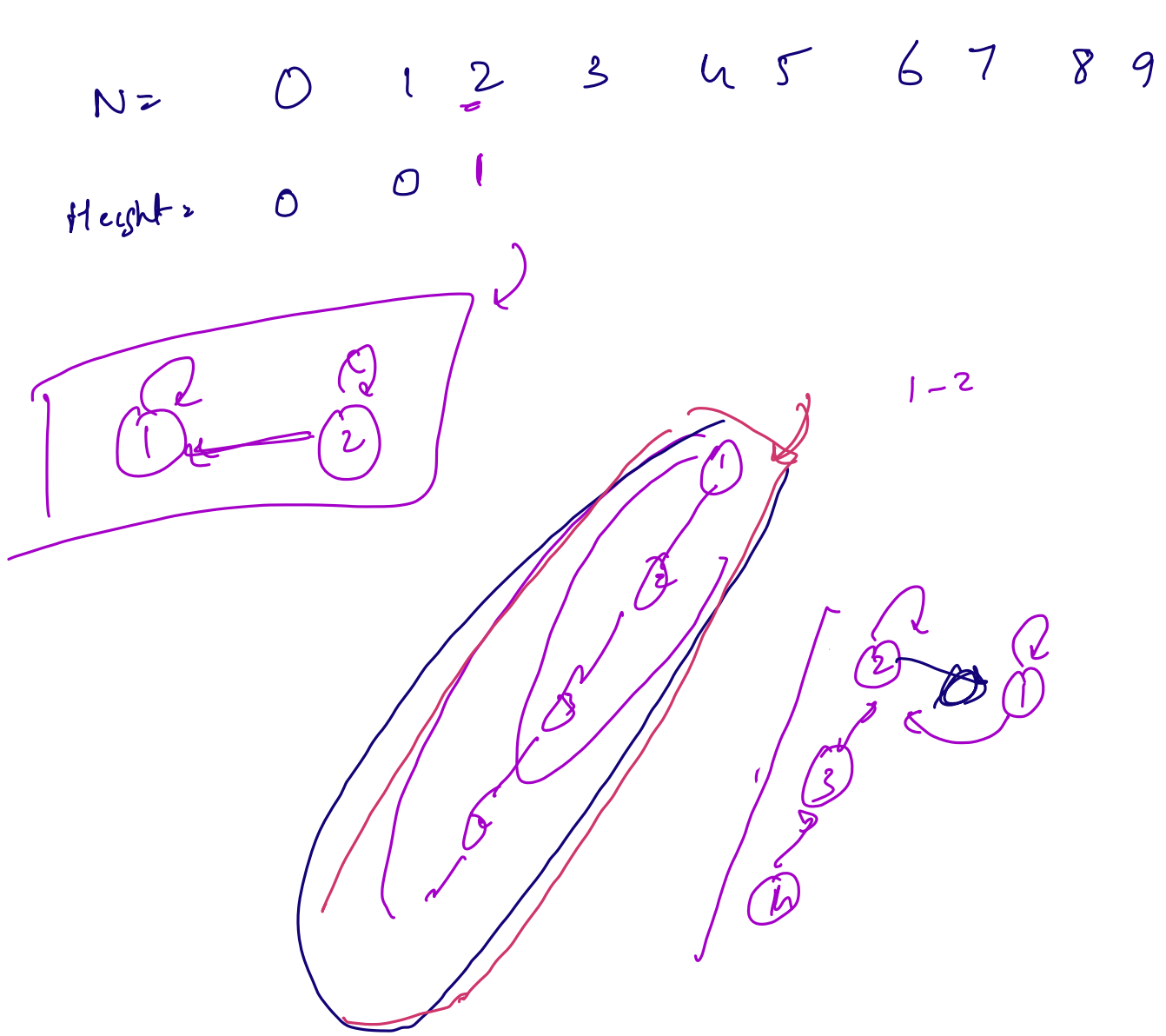
$O(V)$ + $O(V)$

$O(V)$



$H' \geq 2$

Union by Size




```
void prims(int N, adj[]){
      vis[N] = False;
      minHeap(pair<int, int>) pq;
      pq.push({0, 0});
      int ans = 0;

      while(!q.empty()){
            key, u = pq.top();
            pq.pop();
            if(vis[u])
                  continue;

            ans += key;
            vis[u] = true;
            for(auto v : adj[u]){
                  if(!visited[v]){
                        pq.push({W(u, v), v});
                  }
            }
      }
}
```

Union by Weight

```
unionByWeight(int x, int y){
      u = find(x);
      v = find(y);
      if(size[u] > size[v]){
            parent[v] = u;
            size[u] += size[v];
      }
      else if(size[v] > size[u]){
            parent[u] = v;
            size[v] += size[u];
      }

      else{
            parent[u] = v;
            size[v] += size[u];
      }
return;
}
```

Union by Height

```
union(int x, int y){
      u = find(x);
      v = find(y);
      if(height[u] > height[v])
            parent[v] = u;
      else if(height[v] > height[u])
            parent[u] = v;
      else{
            parent[v] = u;
            height[u] += 1;
      }
      return;
}
```

Kruskal

```
int kruskal(){
      ans = 0;
      sort(Edges);
      for(int i = 1 to E){
            ei = (u, v)
            if(find(u) != find(v)){
                  ans += W(ei);
                  Union(u, v);
            }
      }
}
```

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

union(int x, int y){
      u = find(x);
      v = find(y);
      if(size[u] > size[v]){
            parent[v] = u;
            size[u] += size[v];
      }
      else{
            parent[u] = v;
            size[v] += size[u];
      }
return;
}
```