1. Algorithm

A. Data Structures

a. Dynamic Adjacency list (Adj)

- Adj(x) = List of mappings of all the adjacent nodes with "x"
- Adj(x)(y) = Number of Edges from "x" to "y"
- We will maintain this list using "map<T, map<T, int>>" (here T is the data type of node) structure so that we can add or remove nodes in O(log(n*n)) complexity, where n is number of nodes in graph

b. Label Mapping list (Label)

- Label(x) = Label of Component in which "x" node is
- We will maintain this list using "map<T, T>" structure so that we can add or remove nodes in O(log(n)) complexity

c. Storing Size of Components with given Label (Size)

- Size(label) = Size of component with given "label"
- We will maintain this list using "map<T, int>" structure so that we can add or remove labels in O(log(C)) complexity, where C = Number of Components in the graph

d. Storing Visited State of the node for DFS (vis)

- vis(x) = 1 if node "x" is visited else 0
- We will maintain this list using "map<T, bool>" structure so that we can add or remove labels in O(log(C)) complexity, where C = Number of Components in the graph

B. Add a Node

```
If node_in_graph(x) then
    Print "Invalid Input"
    Return

Adj(x) ← Map<T, int>()   // Initialise Adj(x) with empty map
Label(x) ← x   // Label using node itself
Size(x) ← 1   // Initially only one node in the component
```

C. Add an Edge

```
If node_not_in_graph(u) or node_not_in_graph(v) then
    Print “Invalid Input”
    Return


Adj(u)(v) ← Adj(u)(v) + 1
Adj(v)(u) ← Adj(v)(u) + 1

// If nodes are from different components then relabel the smaller component using DFS

If  Label(u) is not equal to Label(v)
    If Size(Label(u)) is greater than Size(Label(v))
        swap(u,v)
    Remove Label(u) from Size
    Remove u from Label
    DFS (u, Label(v))
```

D. Remove a Node

```
If node_not_in_graph(x) then
    Print “Invalid Input”
    Return

// Store all neighbours of “x” in set “adj”
adj ← Adj(x)

Remove x from Adj
Remove Label(x) from Size
Remove x from Label

// Remove “x” from adjacency list of neighbours
For each neighbour in adj
    Remove x from Adj(neighbour)

// Remove the node from adj list after it gets relabelled by DFS
Vis ← Map<T, int>()   // Initialise with empty map
While adj is not empty
   next ← adj.begin()
   adj ← DFS (next, next, adj)
```

E. Remove an Edge

```
If node_not_in_graph(u) or node_not_in_graph(v) or edge_not_in_graph(u,v)  then
    Print “Invalid Input”
    Return

Adj(u)(v) ← Adj(u)(v) - 1
Adj(v)(u) ← Adj(v)(u) - 1

// If two components get disconnected by removing edge, we will relabel both using DFS

If  Adj[u][v] is equal to 0

    Remove u from Adj(v)
    Remove v from Adj(u)
    Remove Label(u) from Size

    adj ← Set(u,v)
    vis ← Map<T, int>()   // Initialise with empty map
    While adj is not empty
        next ← adj.begin()
        adj ← DFS (next, next, adj)
```

F. DFS

- We will update Labels and Size Objects using DFS
- x : Current Node in DFS
- adj : We also remove the visited neighbour nodes for remove_node and remove_edge functions so they are not relabelled twice

```
DFS (x, label, adj)

    Label[x] ← label
    Size[label] ← Size[label] + 1
    vis[x] ← 1
    Remove x from adj

    For each child in Adj(x)
        If child is not visited
            adj ← DFS (child, label, adj)
    Return adj
```

G. Find Number of Components

- The "Size" Object stores the size of Component for given label
- Number of Elements in the "Size" Object will be equal to Number of labels
- Number of Components = Length (Size)

2. Proof of Correctness

- In 1.G we proved that our Result will be equal to length of the object called "Size"
- We will now prove that at any point of time during or after any of the above operations data structures remain consistent with the definition of the Result.
- Initially the length of Size object will be equal to zero, so we covered the base case.
- For all the updates we will first update Adj so the graph remains consistent

A. Adding a node

- By adding a single node to the graph, we are just creating a new label (component) which is independent of all the other elements of graphs hence Result after updating operation should will be increased by 1

$$\text{Result} \leftarrow \text{Result} + 1$$

- We make sure that x is not already in the graph and initialize Adj(x) by an empty map, hence the graph is consistent
- We assign Label(x) = x, since x not already in the graph Label is consistent
- We assign Size(x) = 1, since there is only one node itself in the component
- Now length of the size is increased by one since there was "x" is added as a key in the object

$$\text{Length(Size)} \leftarrow \text{Length(Size)} + 1$$

- Hence after adding a node Result is consistent with the changes in data structures

B. Adding an edge

- There are two possibilities when adding an edge,

a) Edge is added inside a component (Result ← Result)

a. If Label(u) is equal to Label(v) then the edge is getting added to the Connected Component
b. We will not make any changes in Label or Size objects, so the length of the Size remains constant (Length (Size) ← Length (Size))

b) Edge connects two different components (Result ← Result - 1)

a. If Label(u) is not equal to Label(v) then the edge is getting added between two Connected Components
b. We relabel all nodes in Component of "u" using the label of "v" using DFS
c. We will remove the Label(u) from the Size (Length (Size) ← Length (Size) - 1)
d. We will remove u from the Label list

- Hence all the data structures are consistent with the results after adding the edge

C. Removing a node

- We Remove all the edges incidence on the removable node and stores the neighbours in an object "adj"
- We remove Label(x) from Size
- We remove x form Label
- Now for each element in "adj", we relabel all the nodes connected it, and if the node which is getting relabelled is already in "adj" then we remove it from "adj" after relabelling so it does not get relabelled twice
- During the relabelling we increase count of Size(p) if label p is assigned to some node
- Hence the Label and Size data structures stay consistent with the update

D. Removing an edge

- If there are multiple edges between two elements we just decrement the Adj(u)(v) and Adj(v)(u) by 1, no other changes need to be done.
- If there was only one edge between the nodes then,
  - There are two possibilities when removing such an edge,

    a) After Edge removal component stays connected (Result ← Result)

    b) After Edge removal two components are separated (Result ← Result + 1)

    Lets say L = Length (Size) for the further proof

  - We remove edge from the Adj object first (L ← L - 1)
  - We remove Label(u) from the Size
  - We will now relabel all the nodes which are connected by "u" is using the label = "u"
    - We initialize Label(u) = u
    - Then assign Size (Label(u)) = 1 (L ← L + 1)
    - Relabel all nodes with u
  - Now check if "v" was visited during DFS

    a) If True then Edge removal keeps component connected

- Now we can see that L was decremented once and then incremented again, hence L = Length (Size)

b) If false then Edge removal separates the connected component
   - Now we will relabel that component using label = "v"
   - We initialize Label(v) = v
   - Then assign Size (Label(v)) = 1 (L ← L + 1)
   - Relabel all nodes with u
   - Now L is incremented once again hence L = Length (Size) + 1

- After updating the Label and Size, the Result remains consistent with Length(Size)

3. Time Complexity (worst-case)

n : Number of nodes currently present in the graph

m : Number of edges currently present in the graph

A. add_node

- We are just initializing the data structures
- O(1)

B. add_edge

- Updating the Adj matrix : O(log(n))
- DFS : O(n + m)
- Overall : O(n + m + log(n)) = O(n + m)

C. remove_node

- Updating Adj matrix:
  - We need to remove this node from adjacency lists of all connected neighbours , in the worst case which will be = n-1 = O(n)
  - At every removal complexity = O(log(n))
  - Overall Update complexity = O(n * log(n))
- DFS : O(n+m) (We make sure that every node is updated only once)
- Overall : O(n + m + n*log(n)) = O(n*log(n) + m)

D. remove_edge

- Updating the Adj matrix : O(log(n))
- DFS : O(n + m)
- Overall : O(n + m + log(n)) = O(n + m)

E. Find Connected Components

- Result = Length (Size), Here “Size” is an object used in the algorithm
- Complexity : O(1)

Overall Time Complexity after each update = O(n*log(n) + n + m + 1) = O(n*log(n) + m)

Complexity for “q” operations = O(q * (n * log(n) + m))