1 Algorithm

Algorithm 1: Add Node

```
if Node x not in list then
    add x to the adjacency list
end
```

Algorithm 2: Remove Node

```
Remove x from adjacency list;
for all nodes u in adjacency list do
    if Node x exists in adj[u] then
        Remove x;
    end
end
```

Algorithm 3: Add Edge

```
if both nodes exist then
    adj[node1][node2]++;
    adj[node2][node1]++;
end
```

Algorithm 4: Remove Edge

```
adj[node1][node2]–;
adj[node2][node1]–;
if adj[node1][node2]==0 then
    Remove node1 from adj[node2];
    Remove node2 from adj[node1];
end
```

Algorithm 5: DFS

```
stack s;
s.push(node);
while s is not empty do
    x ← top of s;
    s.pop();
    for all nodes u in adj[x] do
        if VisitedArray[adj[x][u]] is not eq to 1 then
            VisitedArray[adj[x][u]] = 1;
            s.push(adj[x][u]);
        end
    end
end
```

Algorithm 6: BFS

```
queue q;
q.enqueue(node);
while s is not empty do
    x ← front of s;
    s.dequeue();
    for all nodes u in adj[x] do
        if VisitedArray[adj[x][u]] is not eq to 1 then
            VisitedArray[adj[x][u]] = 1;
            q.enqueue(adj[x][u]);
        end
    end
end
```

Algorithm 7: Number of components

```
count ← 0;
for all nodes u in graph do
    if VisitedArray[u] is false then
        increment count;
        DFS(node);
    end
end
```

2 Proof of Correctness

- For the adjacency list operations, we can verify the proof of correctness from the definitions itself.
- For DFS and BFS :
  Claim 1: DFS / BFS iterates over the nodes only once. This statement can easily be proved by the fact that we keep a visited array which immediately sets the value of the node to 1 when added on the stack / queue and if the value of the visited array for the node is 0, then we do not add to the stack / queue.
  Claim 2: DFS / BFS covers each connected node. As we can see, we loop over the adjacency list of the nodes. Hence, if there exists an edge between two nodes, then both the nodes will be visited.
- For connected components, we can see that calling DFS or BFS on a node will give us the list of all the nodes connected to it (Proved earlier). Hence, as we loop over all the components, if the node was not visited by the DFS on the prior nodes, we can increment the count as it will be a disconnected component. The final count gives us the list of all the components in the graph.

3 Time Complexity

To better improve on the time complexity, we have used map. If the hash function of the map is good enough, then the time compleity for accessing each element is O(1).

- Add Node: O(1)
- Remove Node: O(n)
- Add Edge: O(1)
- Remove Edge: O(n)
- DFS and BFS: O(V + E) where V and E are the vertices and edges of the connected components respectively.
- Number of Connected Components: O(V + E) where V and E are the total number of vertices and edges.