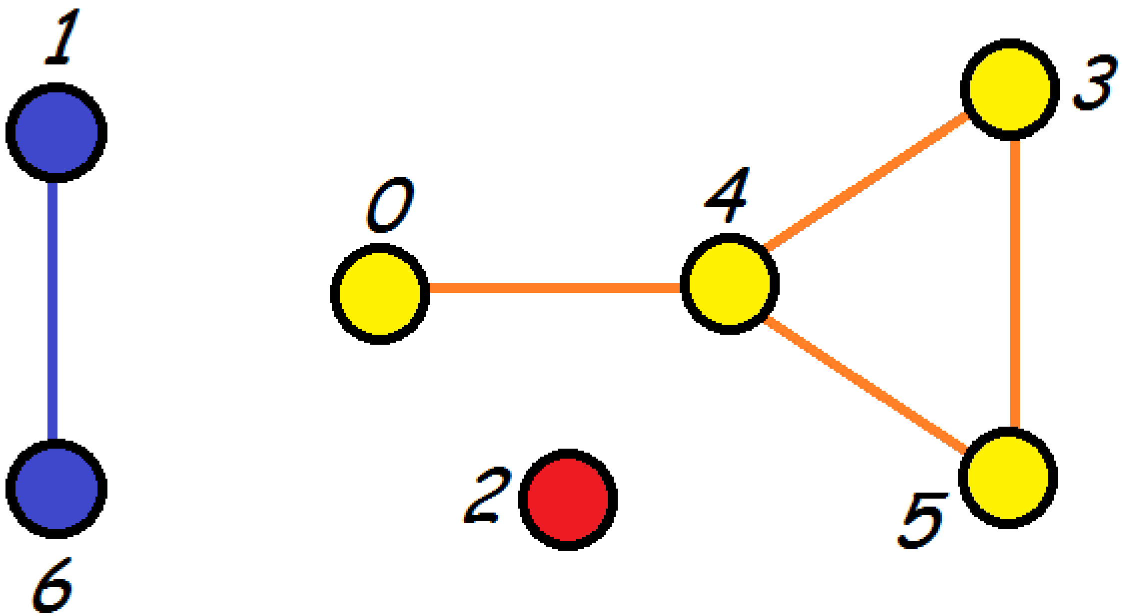
For this graph, give its

(a) adjacency matrix, (b) upper triangular adjacency matrix input format,

(c) adjacency list, and (d) adjacency list input format.



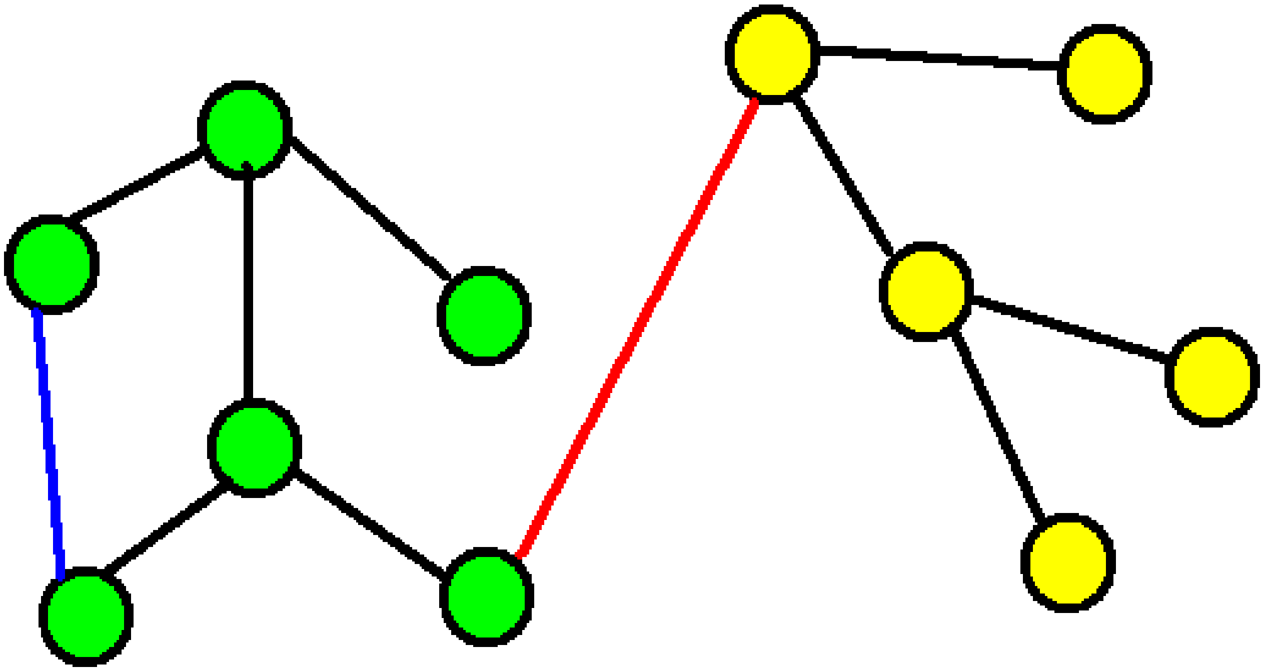
Assignment #4A is due on Fri. Nov. 15 at the beginning of class.

Assignment 4B will be available soon.

You will have to choose a union/find to implement for assignment 4B.

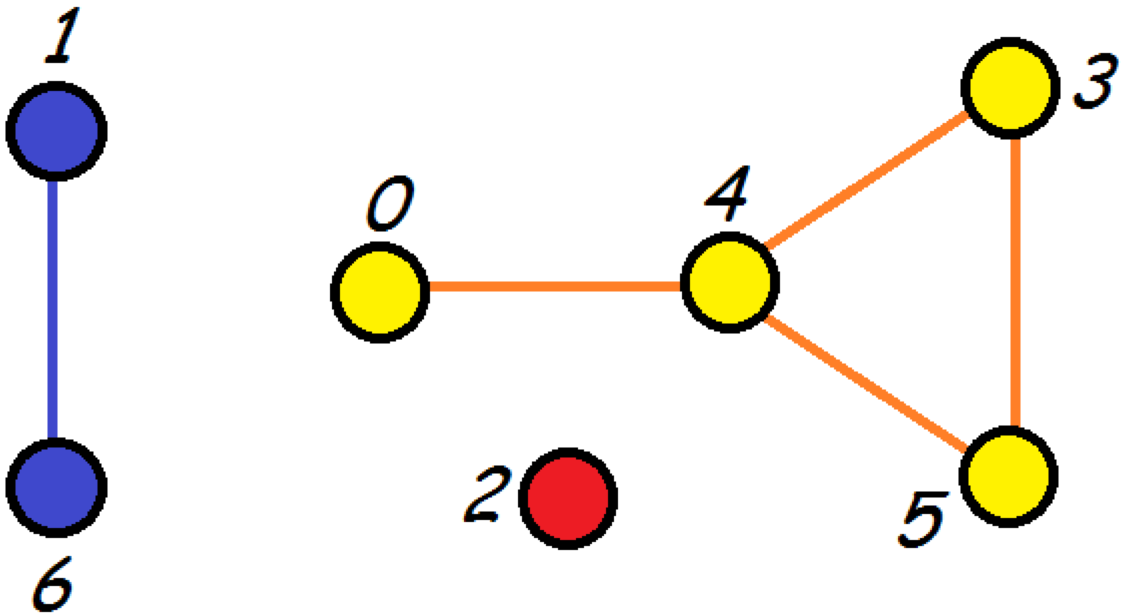
How can we determine quickly at each step whether adding a new edge creates a cycle?

Or equivalently, given an edge (u,v) are u and v in the same component?

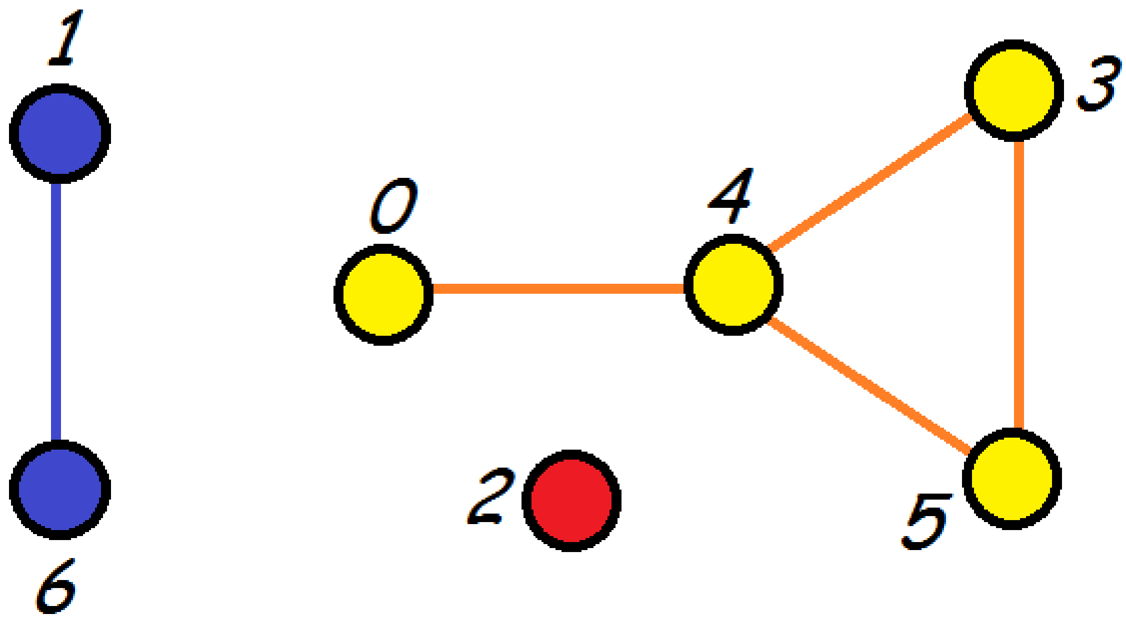


How many connected components does a graph have and which vertices are in each component?

Algorithms: BFS, DFS or UNION/FIND



Union/find: dynamic data structure for keeping track of the connected components of a graph.



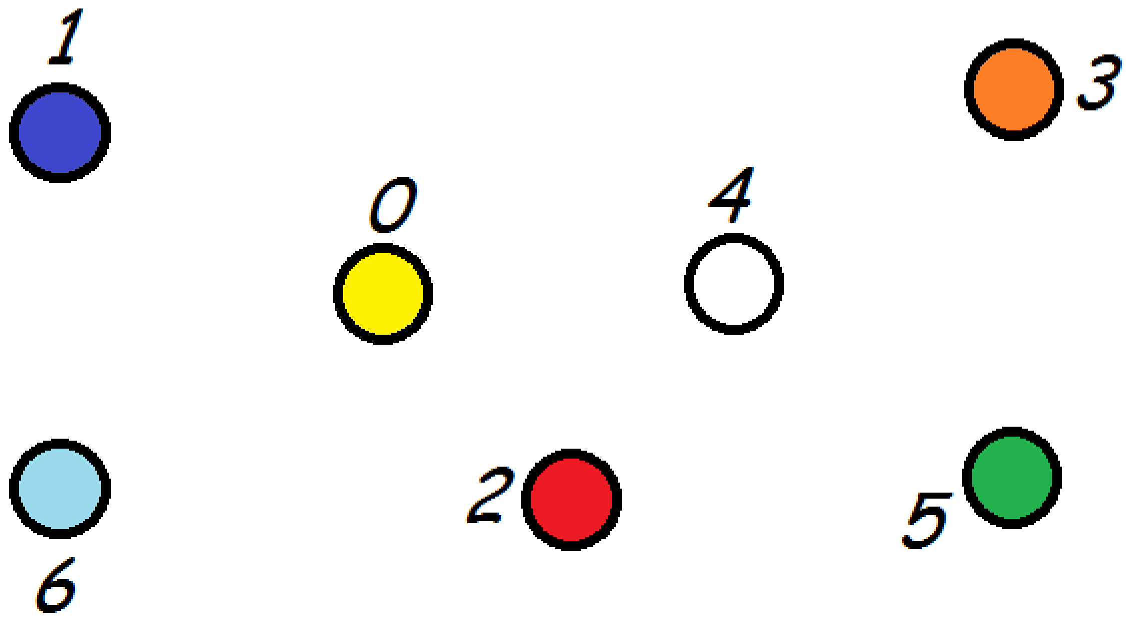
The UNION/FIND data structure is a dynamic data structure for graphs used to keep track of the connected components.

It has 2 routines:

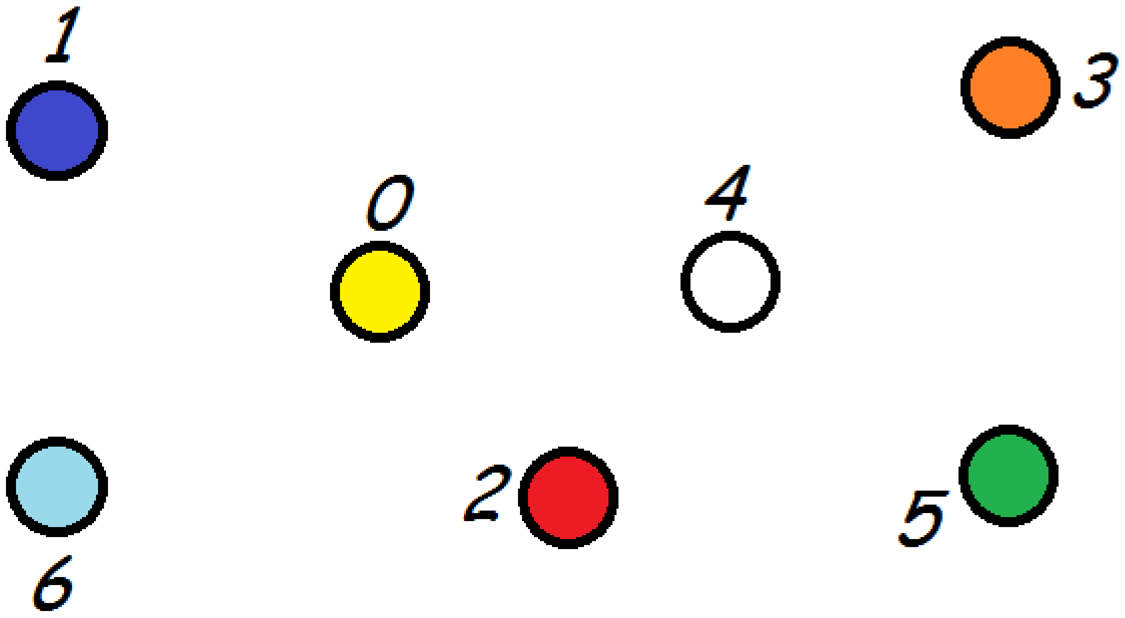
FIND(u): returns the name of the component containing vertex u

UNION(u, v): unions together the components containing u and v (corresponding to an addition of edge (u,v) to the graph).

Each vertex starts out in a component by itself:

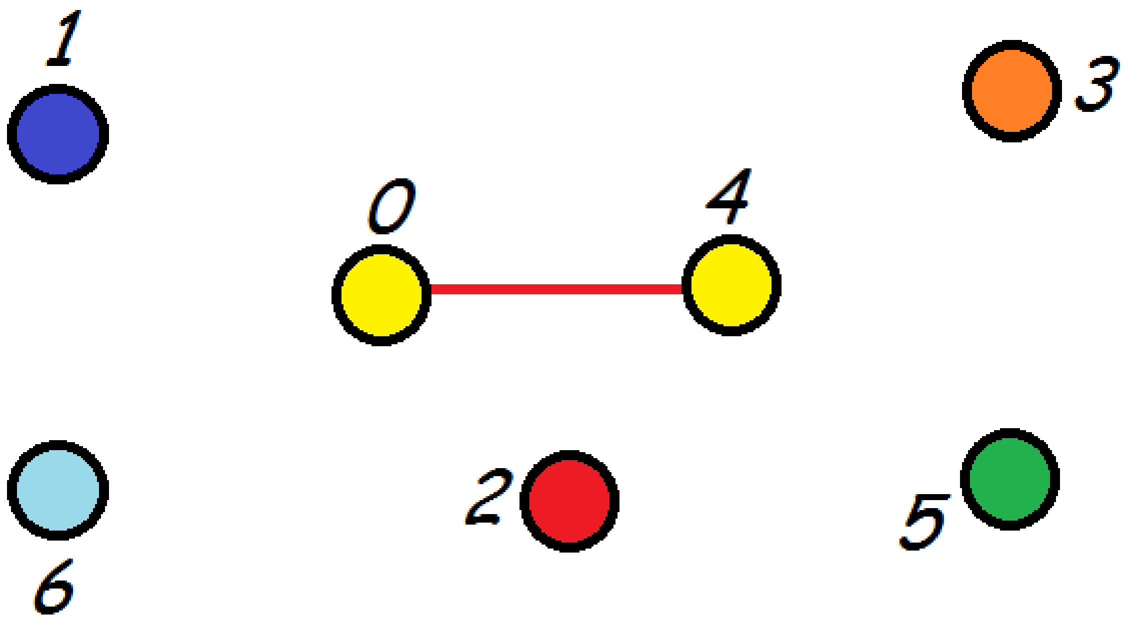


Go through adjacency list or matrix adding each edge we encounter.

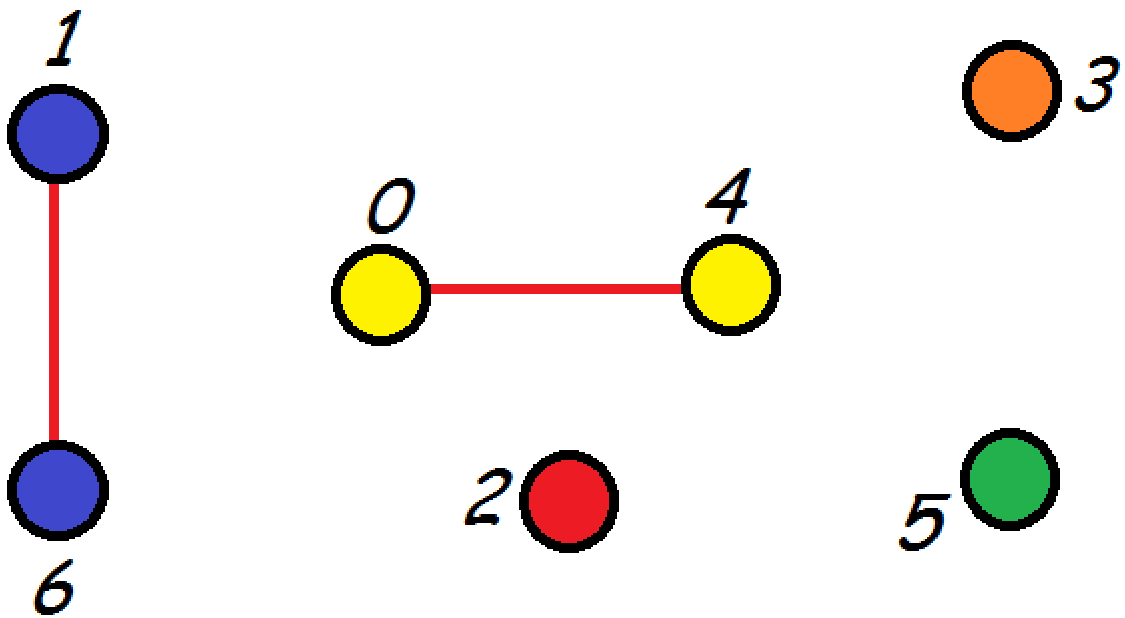


Approach 1 (Flat scheme): parent[i]= min number of vertex in same component as vertex i.

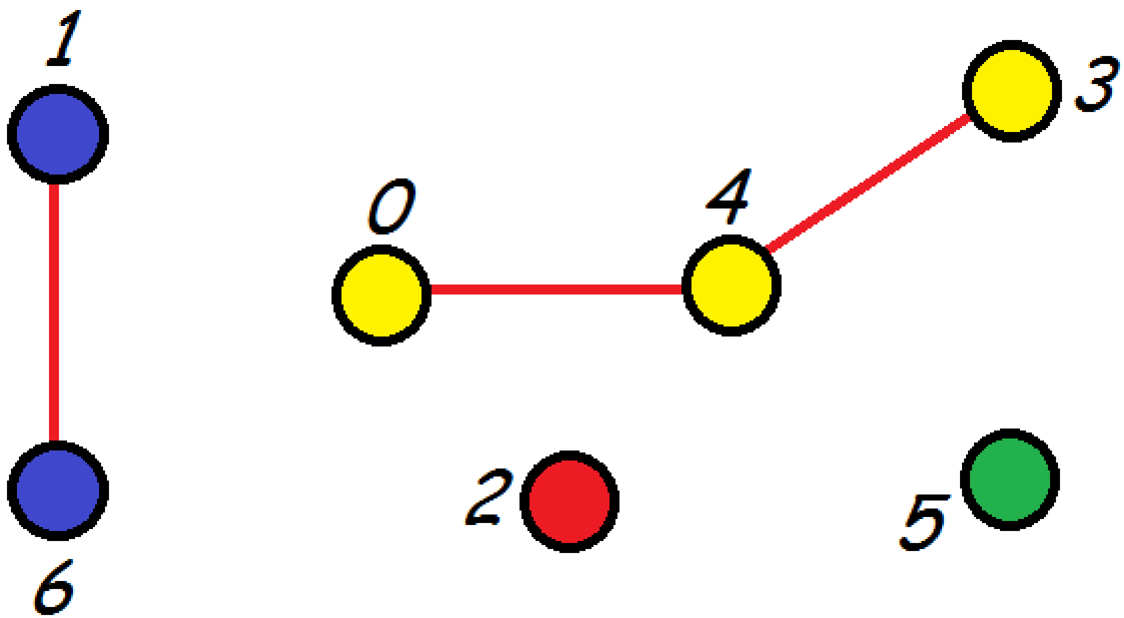
Edge (0,4): Union components with vertices 0 and 4.



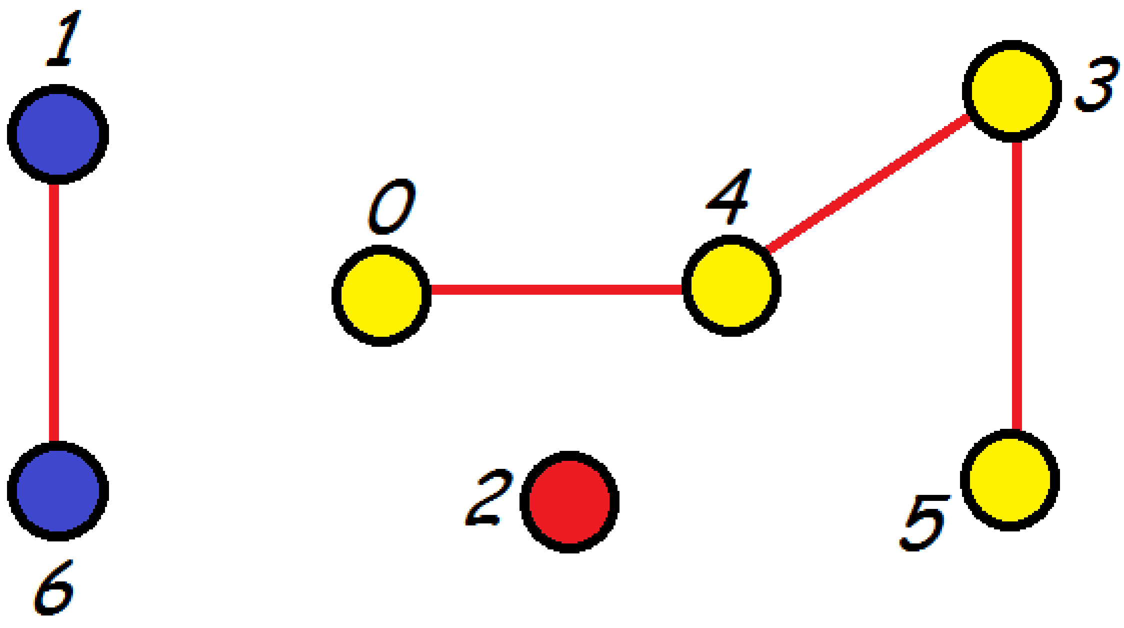
Edge (1,6): Union components with vertices 1 and 6.



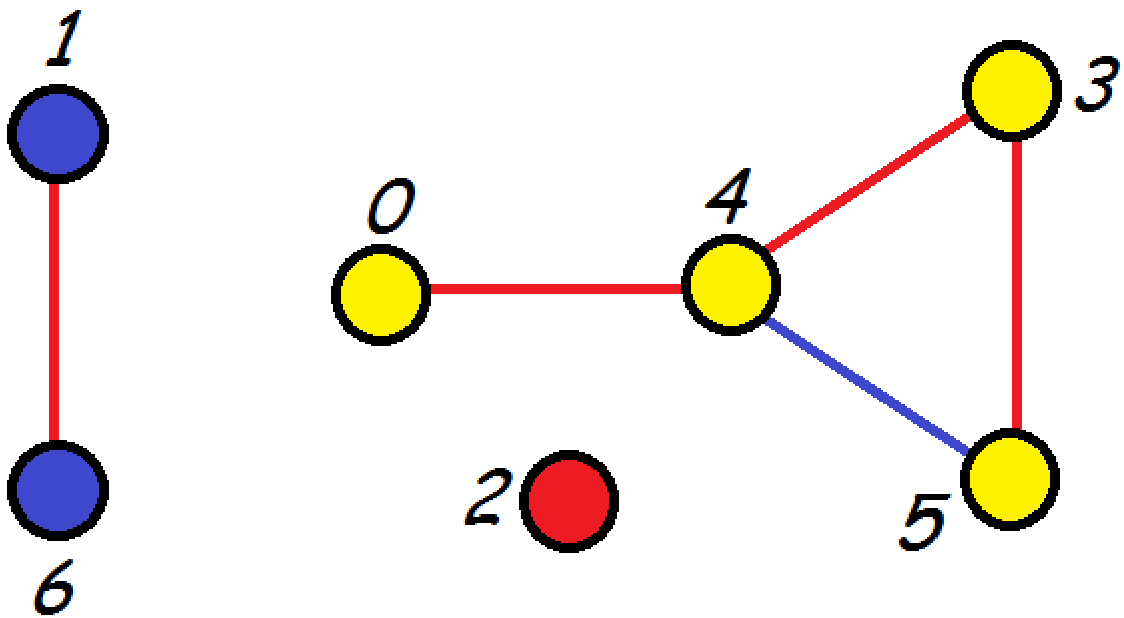
Edge (3,4): Union components with vertices 3 and 4.



Edge (3,5): Union components with vertices 3 and 5.



Edge (4,5): 4 and 5 are already in the same component (the one with vertex 0)



Draw the directed graph that represents the flat union find data structure defined by this parent array:

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	5	5	0	8	0

Show the updated parent array and also draw a picture after

flat_union(7, 4).

One algorithm that can use a union/find data structure: Kruskal's algorithm for finding a minimum weight spanning tree.

One application:

A cable company must install cable to a new neighbourhood. The cables are constrained to be buried along certain paths. The cost varies for different paths. A minimum weight spanning tree gives the cheapest way to connect everyone to cable.

Water distribution network

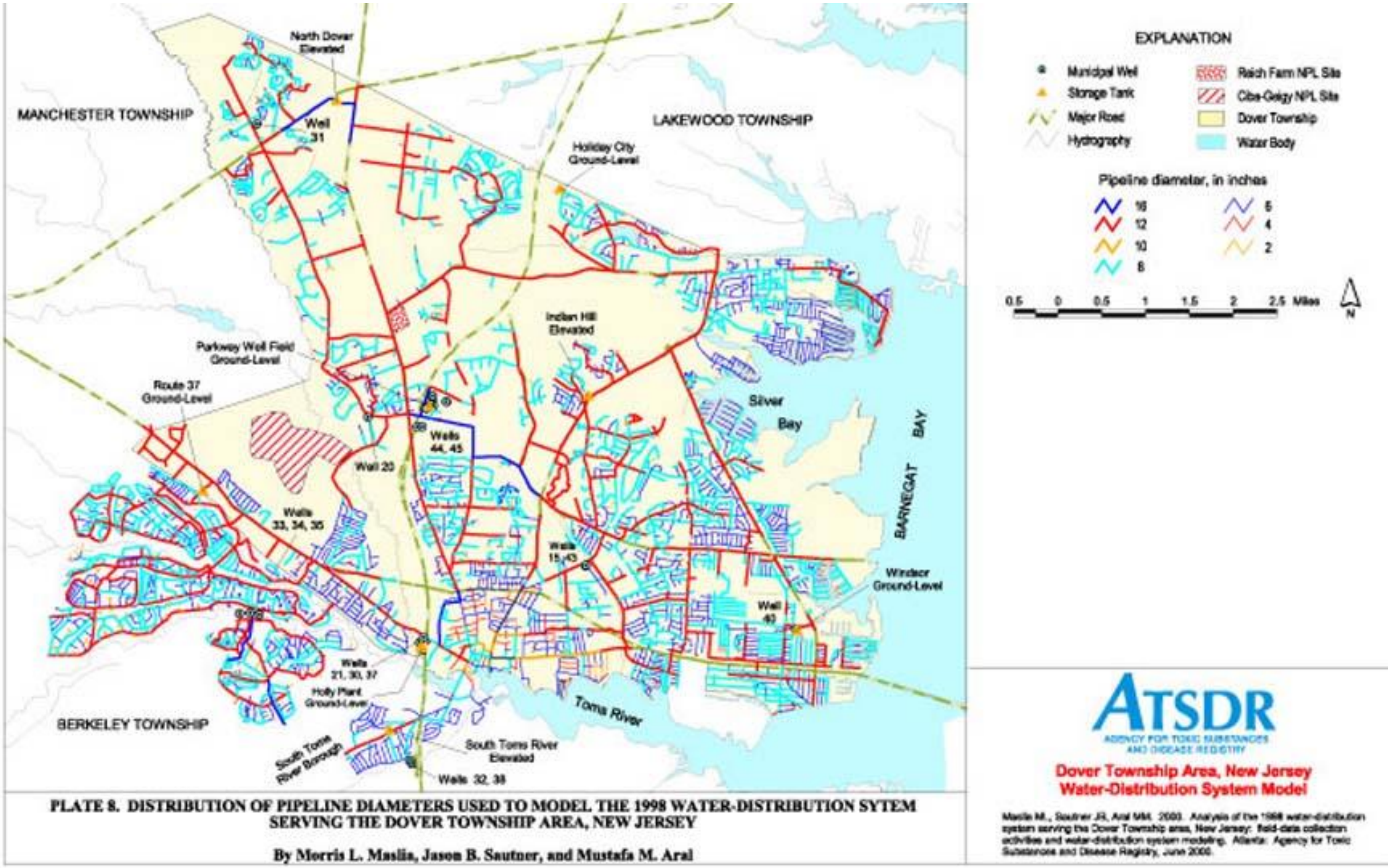


PLATE 8. DISTRIBUTION OF PIPELINE DIAMETERS USED TO MODEL THE 1998 WATER-DISTRIBUTION SYTEM SERVING THE DOVER TOWNSHIP AREA, NEW JERSEY

By Morris L. Maslia, Jason B. Sautner, and Mustafa M. Aral

The UNION/FIND data structure is a dynamic data structure for graphs used to keep track of the connected components.

It has 2 routines:

FIND(u): returns the name of the component containing vertex u

UNION(u, v): unions together the components containing u and v (corresponding to an addition of edge (u,v) to the graph).

The initialization for Approaches 1 and 2 is the same. Each vertex is in a component by itself whose name is that of the vertex.

```
public class UnionFind
{
    int n;
    int [] parent;
    public UnionFind(int nv)
    {   int  i; n= nv;
        parent= new int[n];
        for (i=0; i < n; i++)
            parent[i]=i;
    }
```

Approach 1: A Flat Scheme

The simplest scheme is to choose the vertex with minimum label to be the name of the component. We maintain an array parent which records the name of the component for each vertex.

The FIND function is:

```
public int flat_find(int u)
{
        return(parent[u]);
}
```

The UNION function:

```
public void flat_union(int u, int v)
{  int i, min, max;
   if (parent[u] == parent[v]) return;
   if (parent[u]  <  parent[v])
   { min= parent[u]; max=parent[v]; }
   else
   { max= parent[u]; min=parent[v]; }

   for (i=0; i < n; i++)
       if (parent[i]== max)
            parent[i]= min;
    }
}
```

Using the flat scheme, what are the time complexities for:

1. flat_union?

2. flat_find?

Approach 2: Slower FIND/Faster UNION

A second approach is to be lazy with the union operator:

```
public void lazy_union(int u, int v)
{
    int pu, pv;

    pu= lazy_find(u);
    pv= lazy_find(v);
    if (pu != pv)
        parent[pu]= pv;
}
```

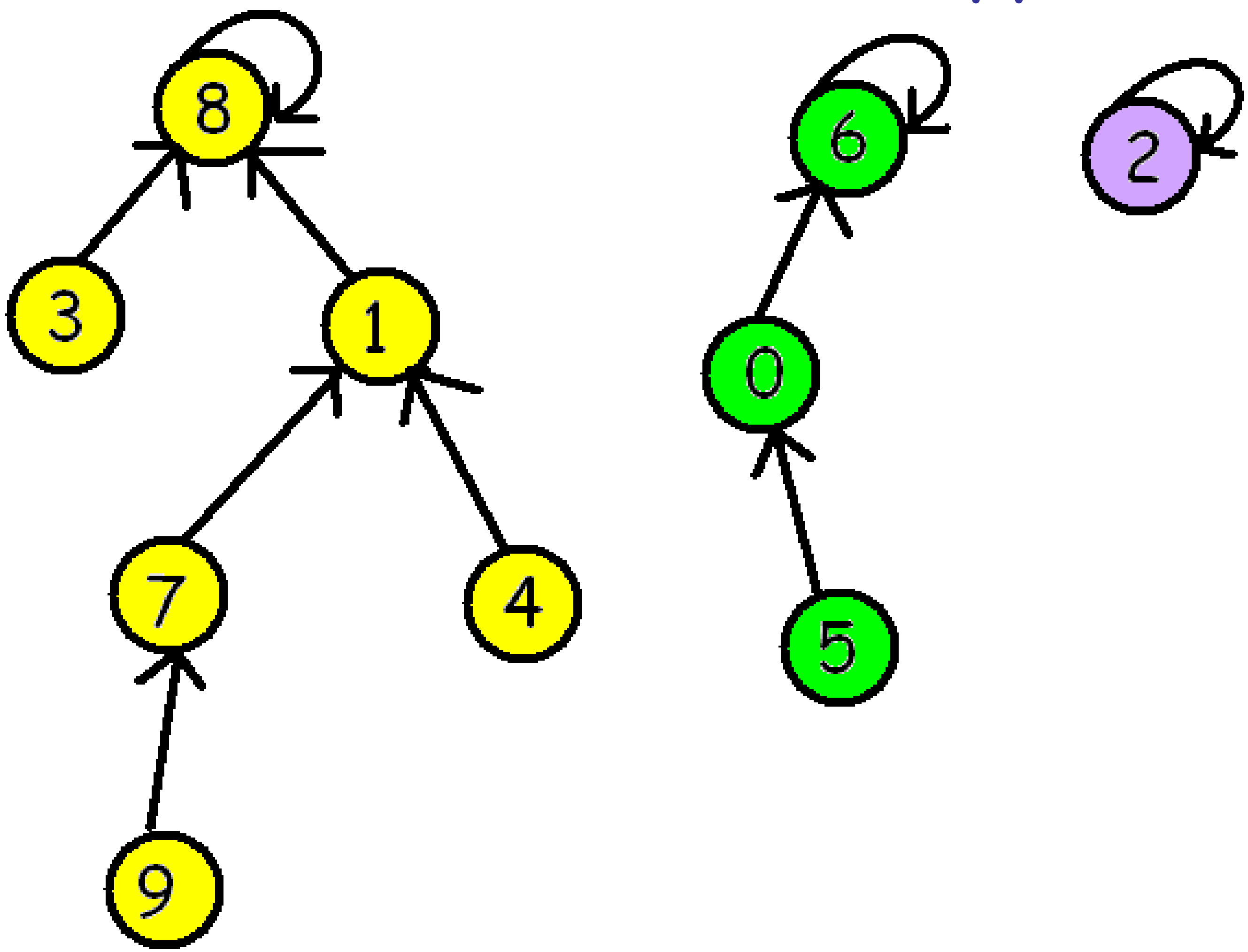
0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	5	5	0	8	0

Show the updated parent array and also draw a picture after lazy_union(7, 4).

But now, we need to traverse the structure to find the representative vertex for the component:

```
int lazy_find(int u)
{
    while (parent[u] != u)
    {
        u=parent[u];
    }
    return(u);
}
```

What is in the parent array which corresponds to this picture of a union/find data structure (Approach 2):



Approach 3: Balancing the complexities of UNION and FIND

The find for Approach 3 is similar to that for Approach 2. However, by being more careful with the UNION operation, we can reduce the complexity of the FIND.

The parent operates as before except now instead of storing parent[v]=v for a root node, we store (-1) * [the number of nodes in the component whose representative is v].

```
public UnionFind(int nv)
{   int  i;
      n= nv;
      parent= new int[n];
      for (i=0; i < n; i++)
          parent[i]= -1;
}
```

The parent operates as before except now instead of storing parent[v]=v for a root node, we store (-1) * [the number of nodes in the component whose representative is v].

The find for weighted union becomes:

```
int w_find(int u)
{
        while (parent[u] >= 0)
        {
            u=parent[u];
        }
        return(u);
}
```

WEIGHTED UNION:

```
public void w_union(int u, int v)
{  int pu, pv, nu, nv;

   pu= w_find(u);
   pv= w_find(v);
   if (pu == pv) return;

   nu= -1 * parent[pu];
   nv= -1 * parent[pv];
```

```
  if (nu <  nv)
  { // pv is the new root.
    parent[pv]+= parent[pu]; // -1*(# nodes)
    parent[pu]= pv;
  }
  else
  { // pu is the new root.
    parent[pu]+= parent[pv]; // -1 *(#nodes)
    parent[pv]= pu;
  }
}
```

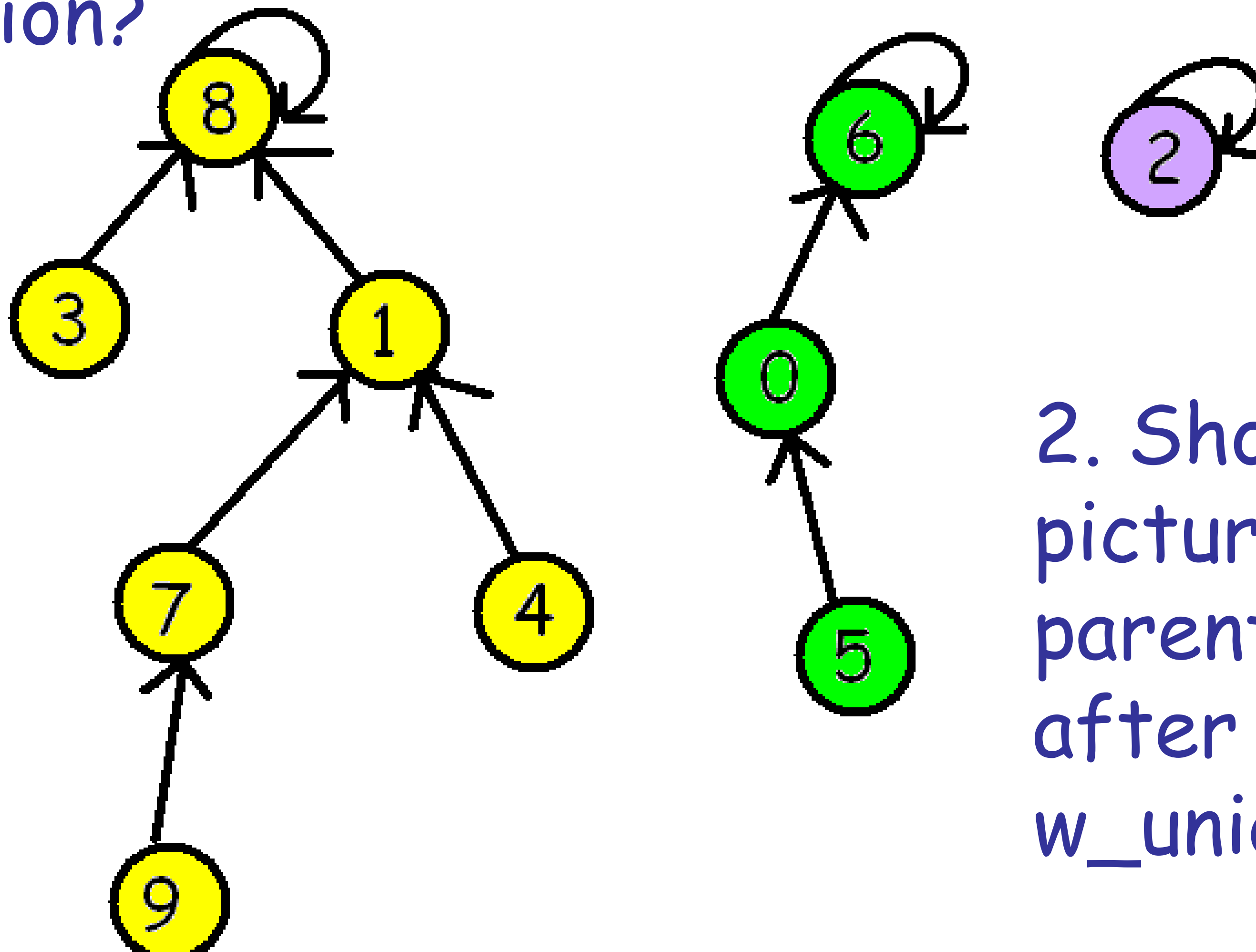
With this modification, UNION (w_union) and FIND (c_find) each take O(log n) in the worst case.

How is this changed if weighted union is used?

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	5	5	0	8	0

Show the updated parent array and also draw a picture after w_union(7, 4).

1. What is in the parent array which corresponds to this picture of a union/find data structure using weighted union?



2. Show the picture and parent array after w_union(7,5)

From wikipedia:

Path compression (collapsing find), is a way of flattening the structure of the tree whenever *Find* is used on it. The idea is that each node visited on the way to a root node may as well be attached directly to the root node; they all share the same representative. To effect this, as Find recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly

Collapsing find:

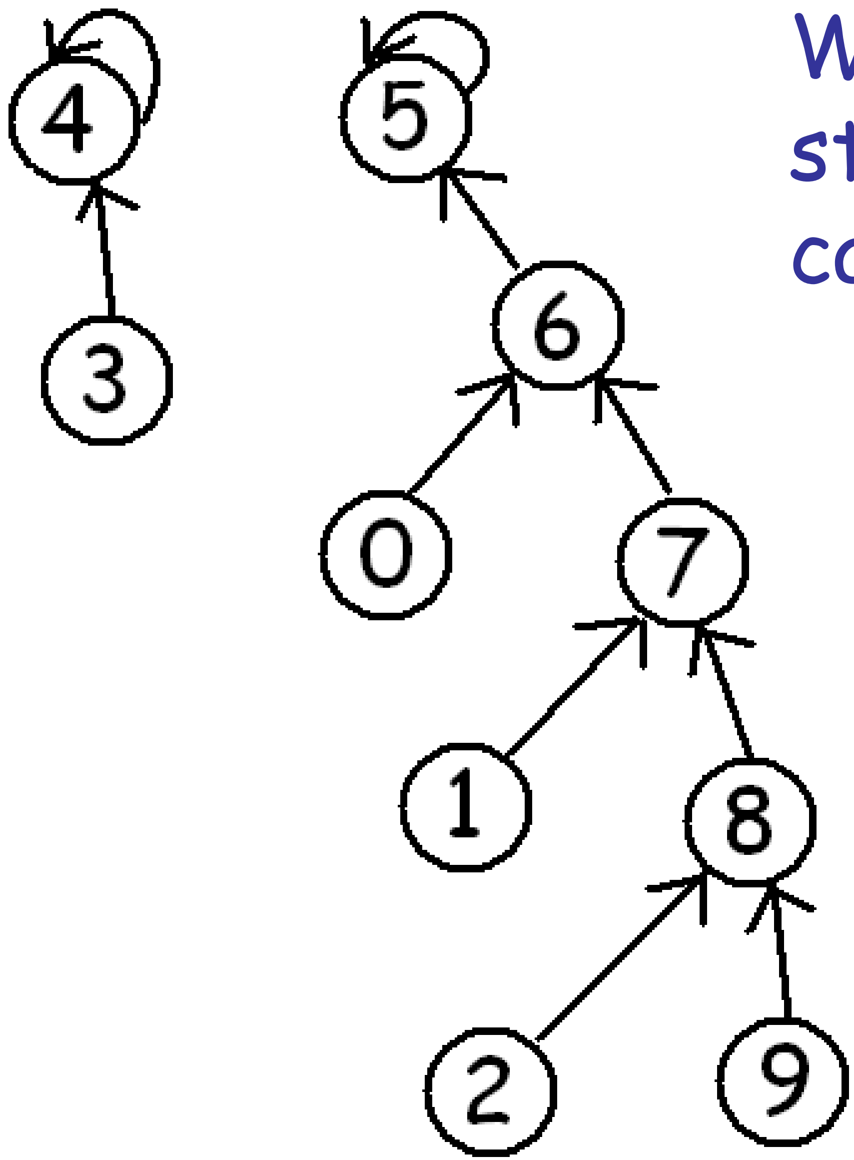
Add a stack to the class:

```
public class UnionFind
{  int n;
   int [] parent; int [] stack;
   public UnionFind(int nv)
   {
      int  i;
      parent= new int[n];
      stack= new int[n];
      for (i=0; i < n; i++)
         parent[i]= -1;
   }
```

COLLAPSING FIND

```
int c_find(int u)
{   int v, top;
    top=0;
    while (parent[u] >= 0)
    {
        stack[top]= u; top++;
        u=parent[u];
    }
    while (top > 0)
    {
        top--; v= stack[top];
        parent[v]=u;
    }
    return(u);
}
```

Weighted union and collapsing find:



What does the data structure look like after calling w-union(2,3)?

Draw a picture and give the parent array.

Note:
w-union(2,3) calls c-find(2) and c-find(3).

Time complexity (wikipedia):
Weighted union (w_union) and collapsing find (c_find) complement each other; applied together, the amortized time per operation is only $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the function $f(n) = A(n,n)$, and A is the extremely quickly-growing Ackermann function. Since $\alpha(n)$ is the inverse of this function, $\alpha(n)$ is less than 5 for all remotely practical values of n. Thus, the amortized running time per operation is effectively a small constant.

Amortized time complexity: the average time per operation over a sequence of operations.