Introduction to algorithms and Union-Find

Overview: why to study algorithms?

Algorithms = problem solving with procedures

Data structures = means to organize data for problem solving

Programming = Algorithms + Data structures

The impact of algoriithmes spans in areas such as the internet, biology, commercial computing, graphics, multimedia, social networks and scientific applications. The impact of algorithms is broad and far-reaching. Algorithms have ancient roots (at least to Euclid, formalized by Turing and Church). Algorithms exists because there are problems that could not otherwise be addressed. For instance, the network connectivity problem. Another reason, is intellectual stimulation. In order to become an efficient programmer, one has to know algorithms and their relation with data structures. Algorithms are computational models, and they are replacing mathematical models. In summary the reasons for study algorithms are data structures are:

- Great impact in almost every major knowledge area today
- Old roots, new oportunities
- To solve problems that could not otherwise be addressed
- Intellectual stimulation
- To become a proficcient programmer
- They may unlock the secrets of life and of the universe
- For fun and profit

Union-Find: Dynamic connectivity

Steps for problem solving:

- Model the problem
- Find the algorithm
- Can we do better?
- Iterate until satisfied

The problem: Given a set of `N` objects connected in groups, find if there's a path connecting two objects.

connect 4 and 3

connect 3 and 8

connect 6 and 5

connect 9 and 4

connect 2 and 1

are 0 and 7 connected? ✗

are 8 and 9 connected? ✔

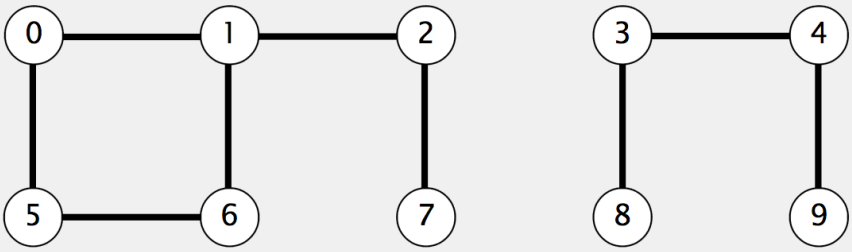
connect 5 and 0

connect 7 and 2

connect 6 and 1

connect 1 and 0

are 0 and 7 connected? ✔




```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)

connected(0, 7) leads to false
connected(8, 9) leads to true

union(5, 0)
union(7, 2)
union(6, 1)
union(1, 0)

connected(0, 7) now leads to true
```

Applications

- Pixels in a digital photo
- Computers in a network
- Friends in social networks
- Variable names in a program
- Transistors in digital chips

Tha abstraction uses integers representing objects. `connected`, is an equivalence relation:

- `connected(p, p)`
- `connected(p, q) => connected(q, p)`
- `connected(p, q) /\ connected(q, r) => connected(p, r)`

Definition. A **conected component**, is a maximal set of objects that are mutually connected.

Example:

```
{0} {1, 4, 5} {2, 3, 6, 7}
```

Algorithm:

- Find: check if two objects are in the same component
- Union: replce components containing two objects with their union

```
{0} {1, 4, 5} {2, 3, 6, 7}
union(2, 5)
{0} {1, 4, 5, 2, 3, 6, 7}
```

ADT:

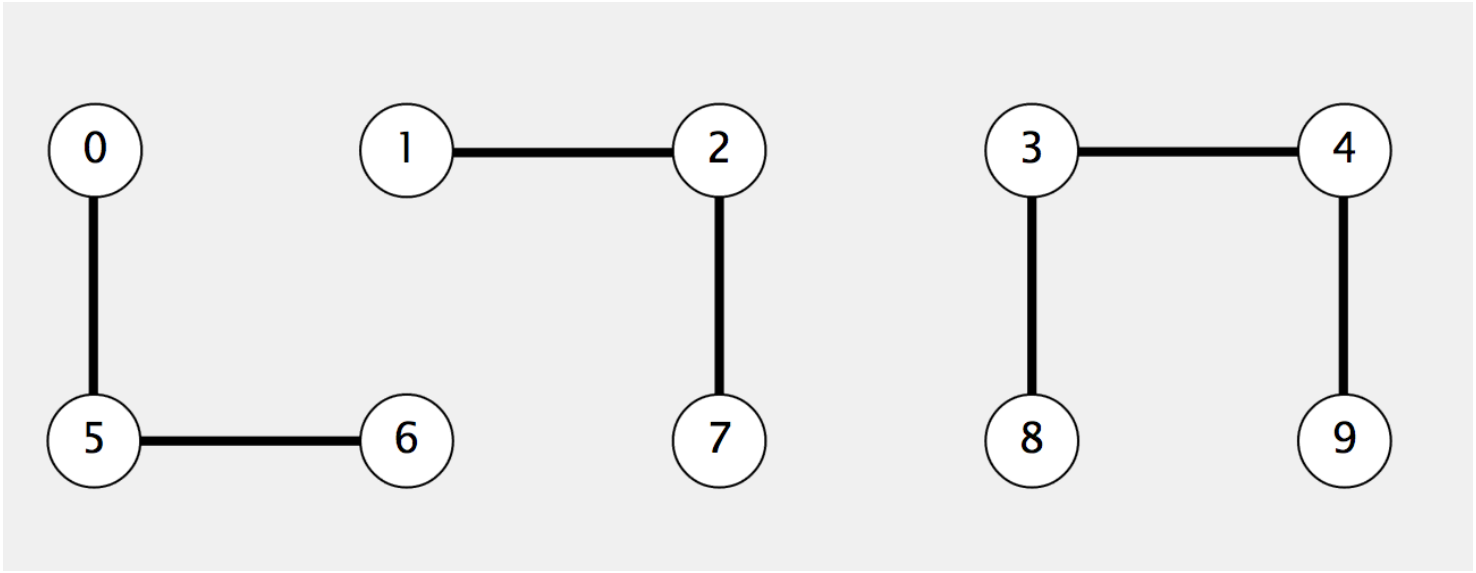
```
ADT UnionFind {

    // Initializes UF data structure with N objects (0 - N - 1)
    init(N: int): UnionFind

    // Adds a connection between p, and q
    union(p, q: int): void

    // Tells whether or not p, and q are connected
    connected(p, q: int): boolean

    // Which component does p belongs to
    find(p: int): int

    // Number of components
    count(): int
}
```

The goal for both the design of the ADT, and the algorithms, is to take into account that the number of objects can be huge, and the intensity of queries can be high. The following piece of code, receives an integer `N`, creates the `UF` object, and reads pairs of objects for proper connection, via the `union` method.

```
public static void main(String[] args) {
    int N = StdIn.readInt();
    UF uf = new UF(N);

    while(!StdIn.isEmpty()) {
        int p = StdIn.readInt();
        int q = StdIn.readInt();

        if(!uf.connected(p, q)) {
            uf.union(p, q);
            StdOut.println(p + " " + q);
        }
    }
}
```

Union-Find: Quick-Find

- `id[] = [id[0], id[1], ..., id[N-1]]`
- `connected(p, q) <=> id[p] = id[q]`



`id[] = [0, 1, 2, 8, 8, 0, 0, 1, 8, 8]`

Find: check if `p`, and `q`, have the same `id`

```java
public boolean connected(int p, int q) {
    return this.id[p] == this.id[q];
}
```

Union: to merge components containing `p` and `q`, change all entries whose `id` equals `id[p]` to `id[q]`

Example:

```
N = 10
id[] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

union(4, 3)
// forall i in [0..N-1] such that id[i] = id[4] => change it to id[3]
id[] = [0, 1, 2, 3, 3, 5, 6, 7, 8, 9]
union(3, 8)
// forall i in [0..N-1] such that id[i] = id[3] => change it to id[8]
id[] = [0, 1, 2, 8, 8, 5, 6, 7, 8, 9]
union(6, 5)
// forall i in [0..N-1] such that id[i] = id[6] => change it to id[5]
id[] = [0, 1, 2, 8, 8, 5, 5, 7, 8, 9]
union(9, 4)
// forall i in [0..N-1] such that id[i] = id[9] => change it to id[4]
id[] = [0, 1, 2, 8, 8, 5, 5, 7, 8, 8]
union(2, 1)
// forall i in [0..N-1] such that id[i] = id[2] => change it to id[1]
id[] = [0, 1, 1, 8, 8, 5, 5, 7, 8, 8]

connected(8, 9) // id[8] == id[9]? => true
connected(5, 0) // id[5] == id[0]? => false

union(5, 0)
// forall i in [0..N-1] such that id[i] = id[5] => change it to id[0]
id[] = [0, 1, 1, 8, 8, 0, 0, 7, 8, 8]
```

The whole implementation:

```java
public class QuickFindUF {

    private int[] id;
    private int N;

    public QuickFindUF(int N) {
        this.N = N;
        this.id = new int[this.N];
        for(int i = 0; i < this.N; i++)
            id[i] = i;
    }

    public void union(int p, int q) {
        int pid = id[p];
        int qid = id[q];
        for(int i = 0; i < this.N; i++)
            if(id[i] == pid) id[i] = qid;
    }

    public boolean connected(int p, int q) {
        return this.id[p] == this.id[q];
    }

}
```

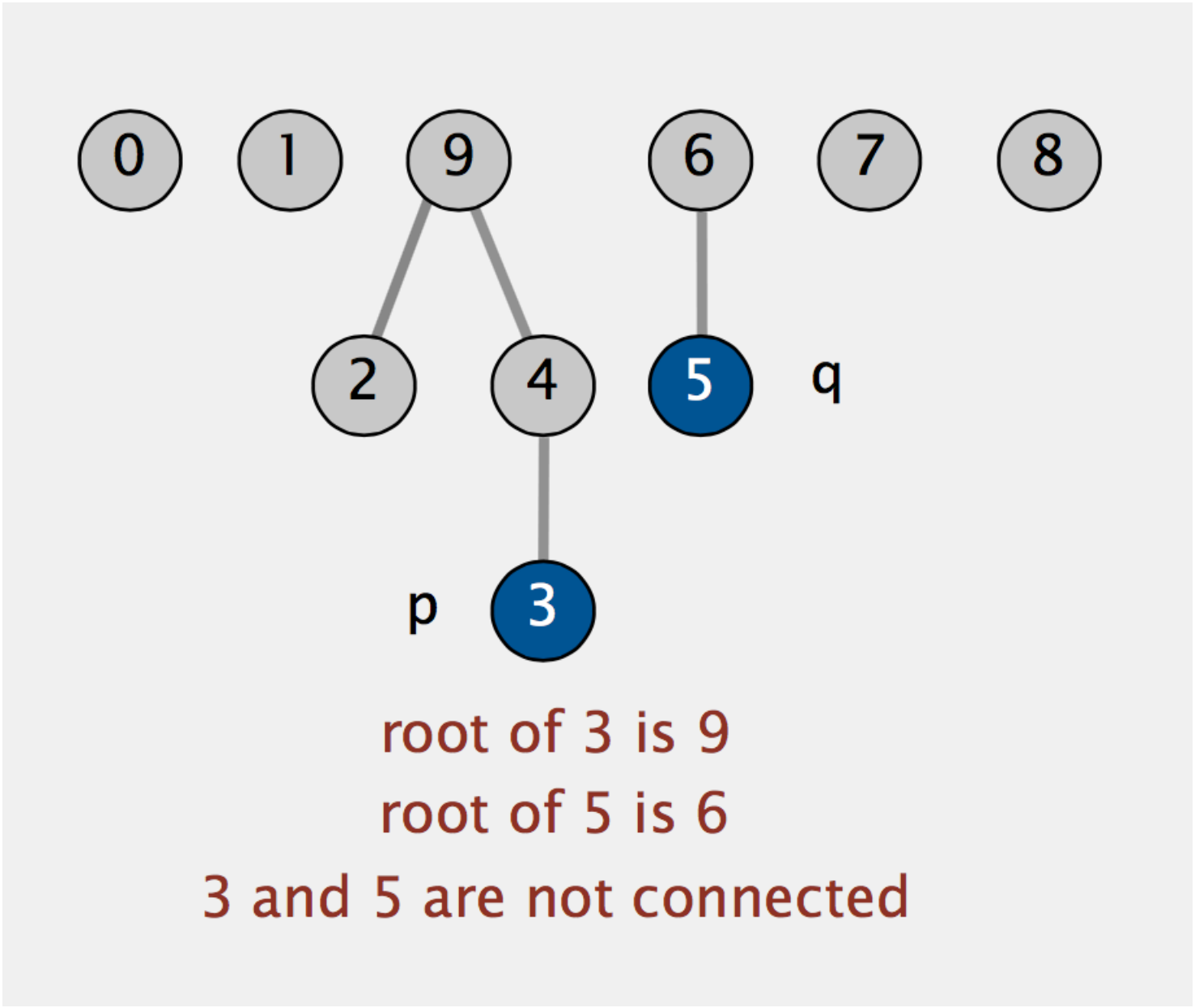
Takes N^2 array accesses to process sequences of N union commands on N objects. N^2, does not scale on large input.

Union-Find: Quick-Union

- `id[] = [id[0], id[1], ..., id[N-1]]`
- `id[i]`, is parent of `i`
- **Root** of `i` is `id[id[id[...id[i]...]]]`

`id[] = [0, 1, 9, 4, 9, 6, 6, 7, 8, 9]`
`parent(3) = 4 /\ parent(4) = 9 /\ parent[9] = 9`

Roots point to themselves.

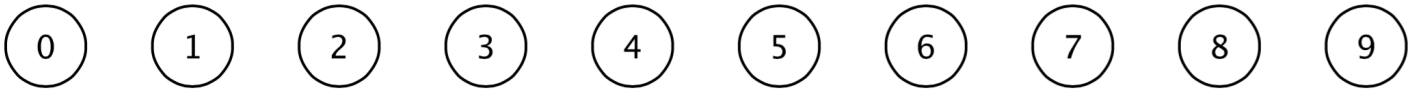


Find: check if `p`, and `q`, have the same **root**

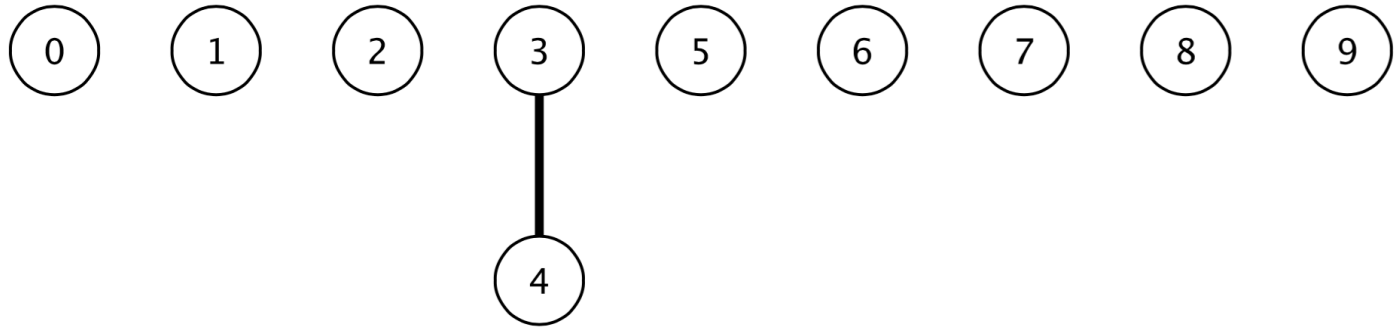
Union: to merge components containing `p` and `q`, set the `id` of `p's` root to the `id` of `q's` root.

Example:

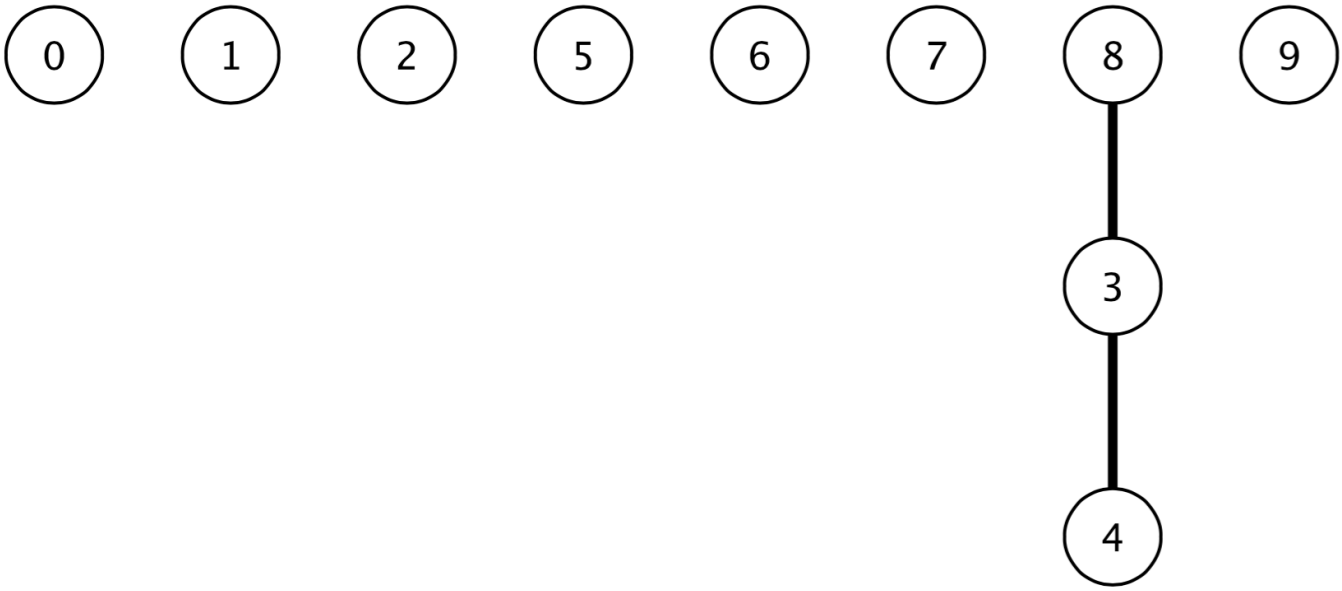
```
id[] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```



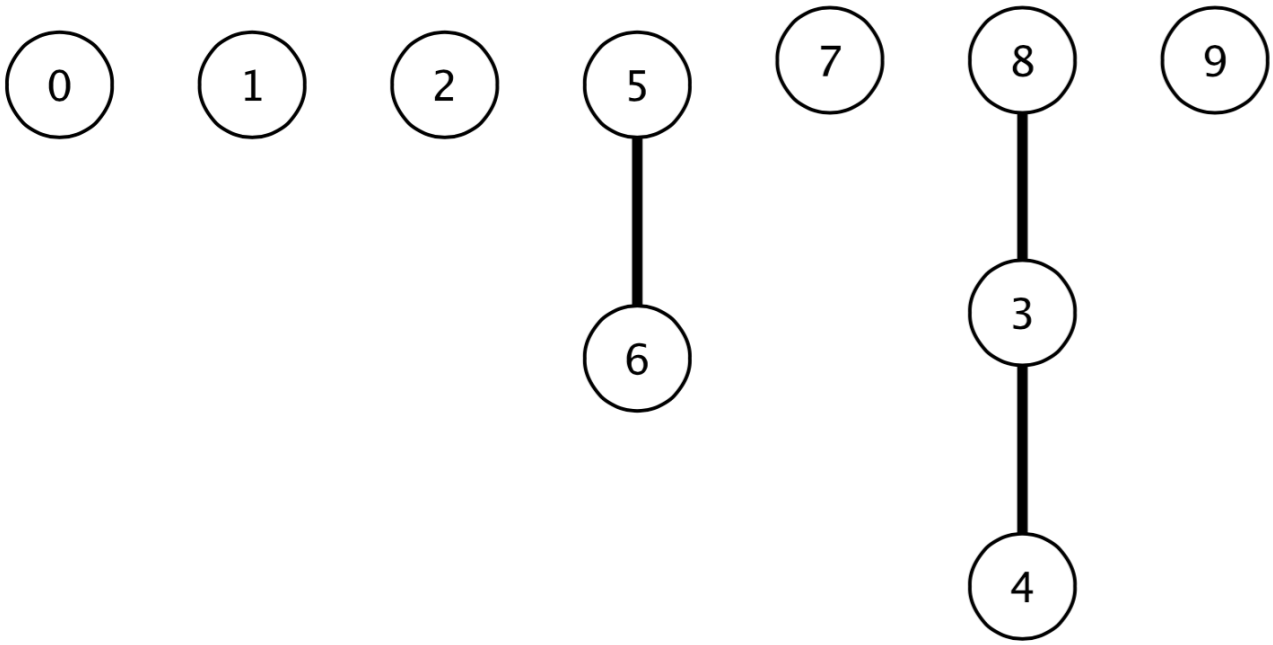
```
id[] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
union(4, 3)
parent(4) = 4 => Root

parent(3) = 3 => Root
id[] = [0, 1, 2, 3, 3, 5, 6, 7, 8, 9]
```



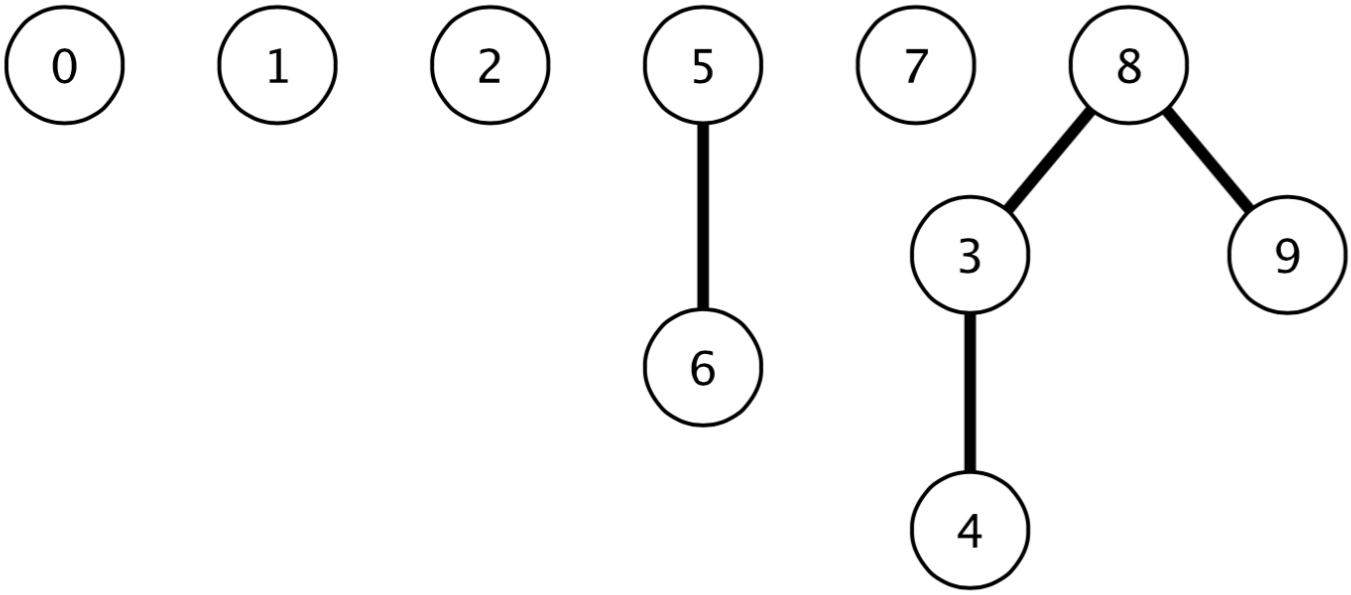

```
id[] = [0, 1, 2, 3, 3, 5, 6, 7, 8, 9]
union(3, 8)
parent(3) = 3 => Root

parent(8) = 8 => Root
id[] = [0, 1, 2, 8, 3, 5, 6, 7, 8, 9]
```



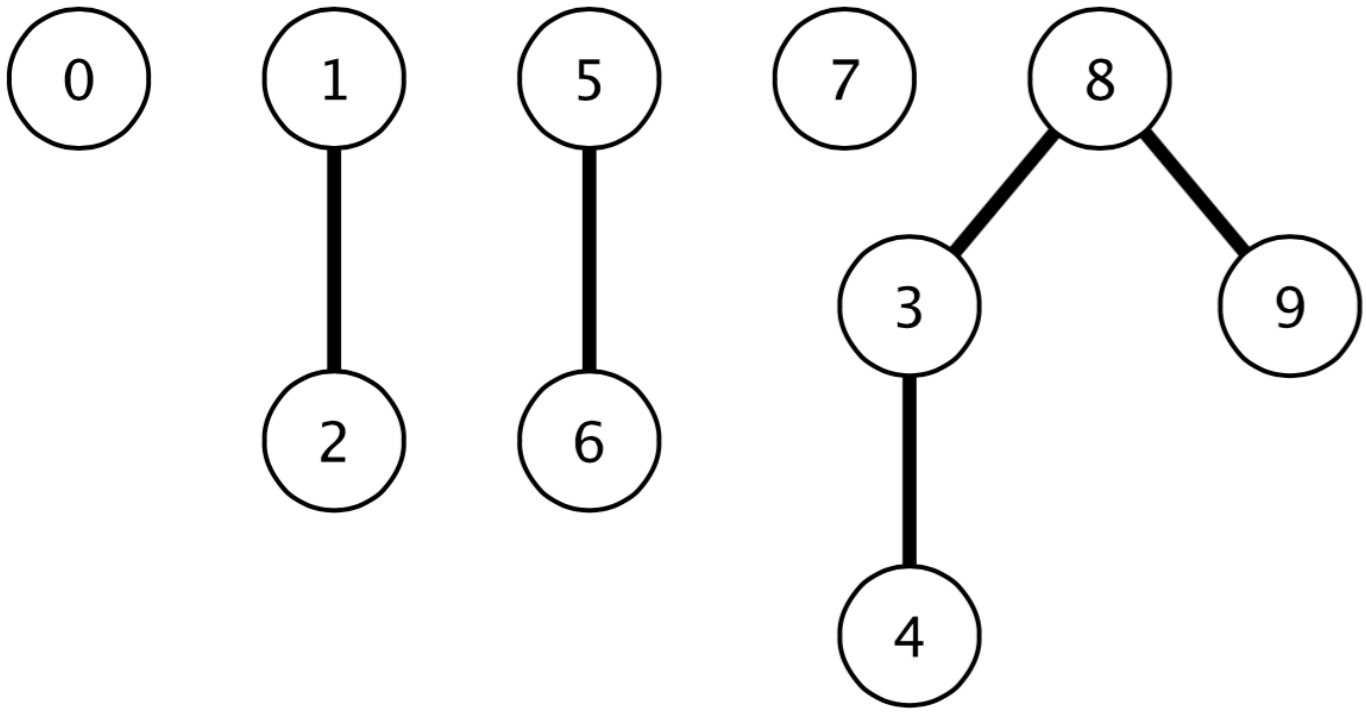
```
id[] = [0, 1, 2, 8, 3, 5, 6, 7, 8, 9]
union(6, 5)
parent(6) = 6 => Root

parent(5) = 5 => Root
id[] = [0, 1, 2, 8, 3, 5, 5, 7, 8, 9]
```




```
id[] = [0, 1, 2, 8, 3, 5, 5, 7, 8, 9]
union(9, 4)
parent(9) = 9 => Root

parent(4) = 3
parent(3) = 8
parent(8) = 8 => Root
id[] = [0, 1, 2, 8, 3, 5, 5, 7, 8, 8]
```

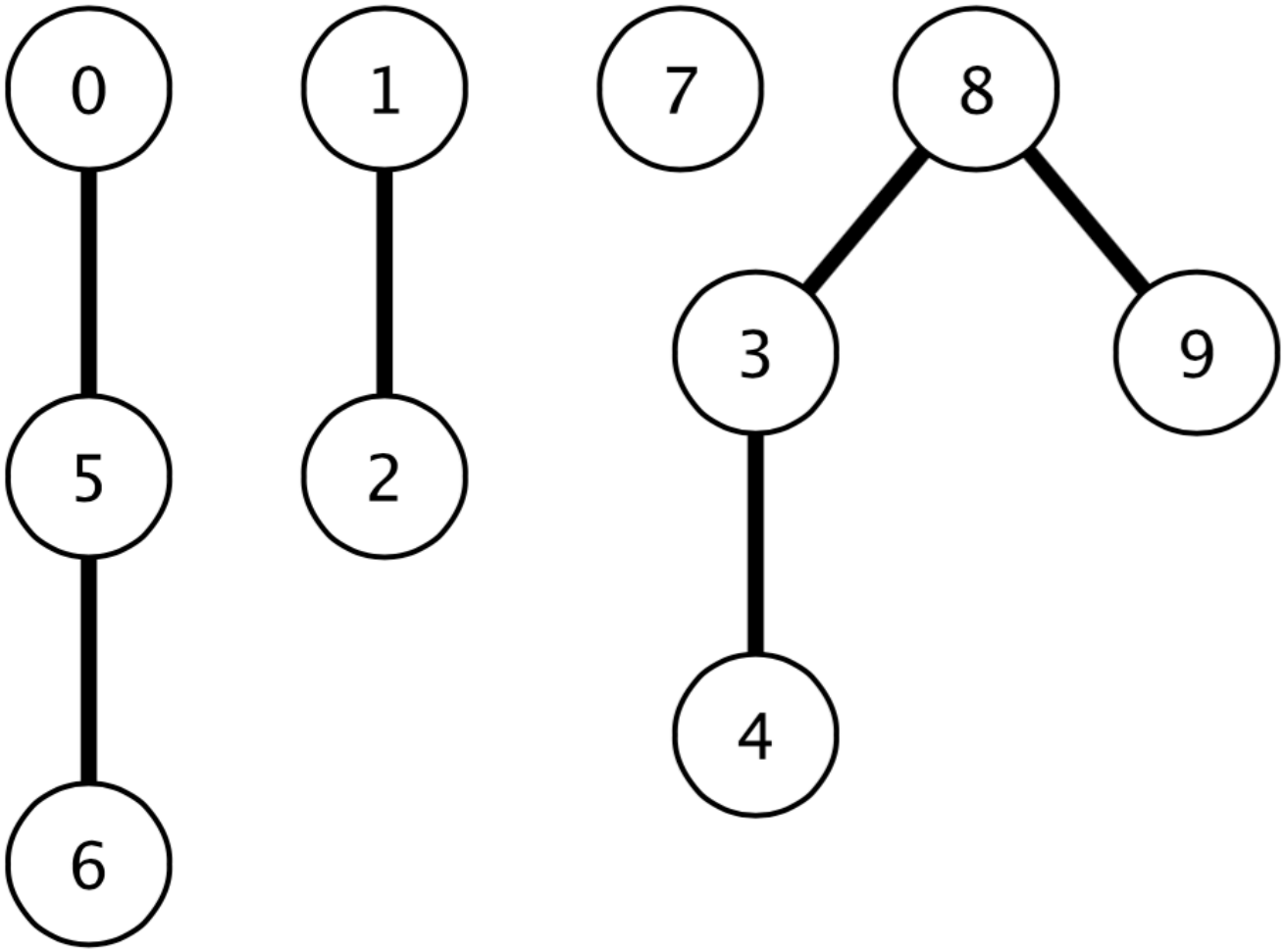



```
id[] = [0, 1, 2, 8, 3, 5, 5, 7, 8, 8]
union(2, 1)
parent(2) = 2 => Root

parent(1) = 1 => Root
id[] = [0, 1, 1, 8, 3, 5, 5, 7, 8, 8]
```

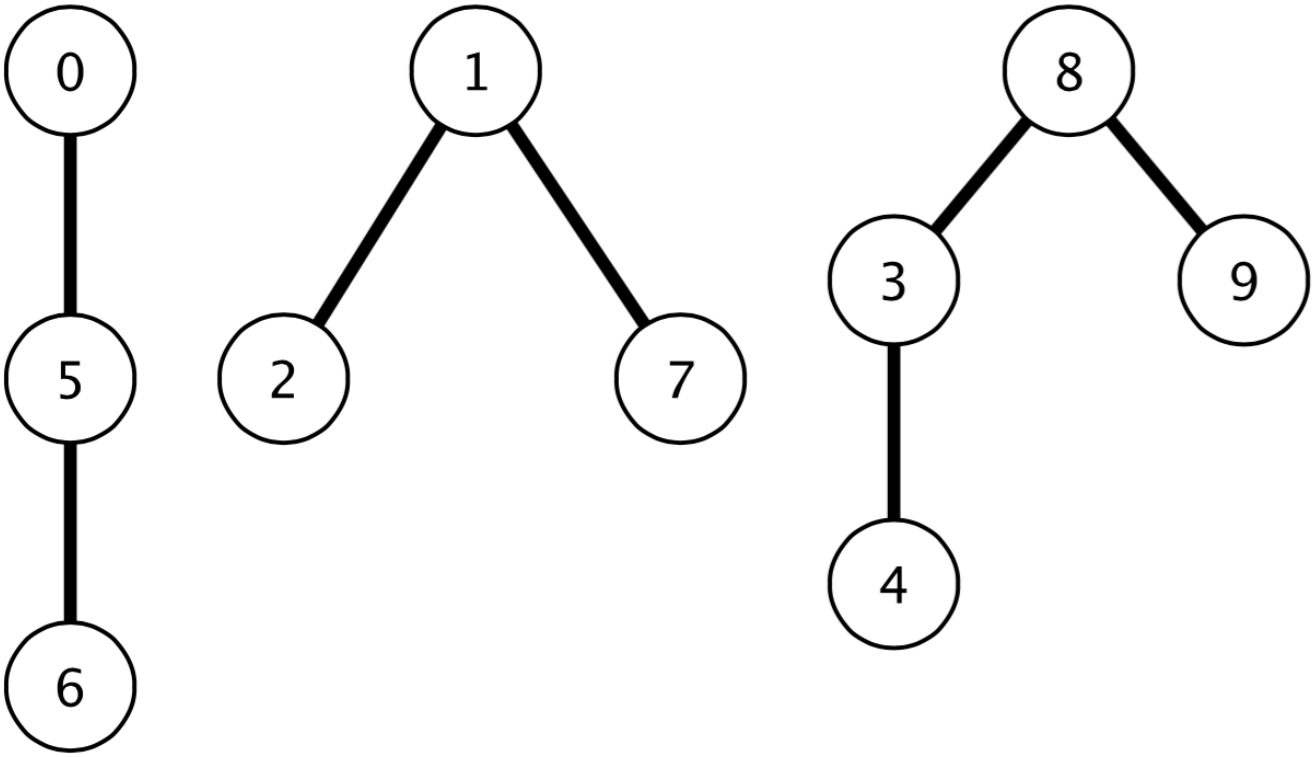



```
connected(8, 9) now leads to true
connected(5, 4) now leads to false
```

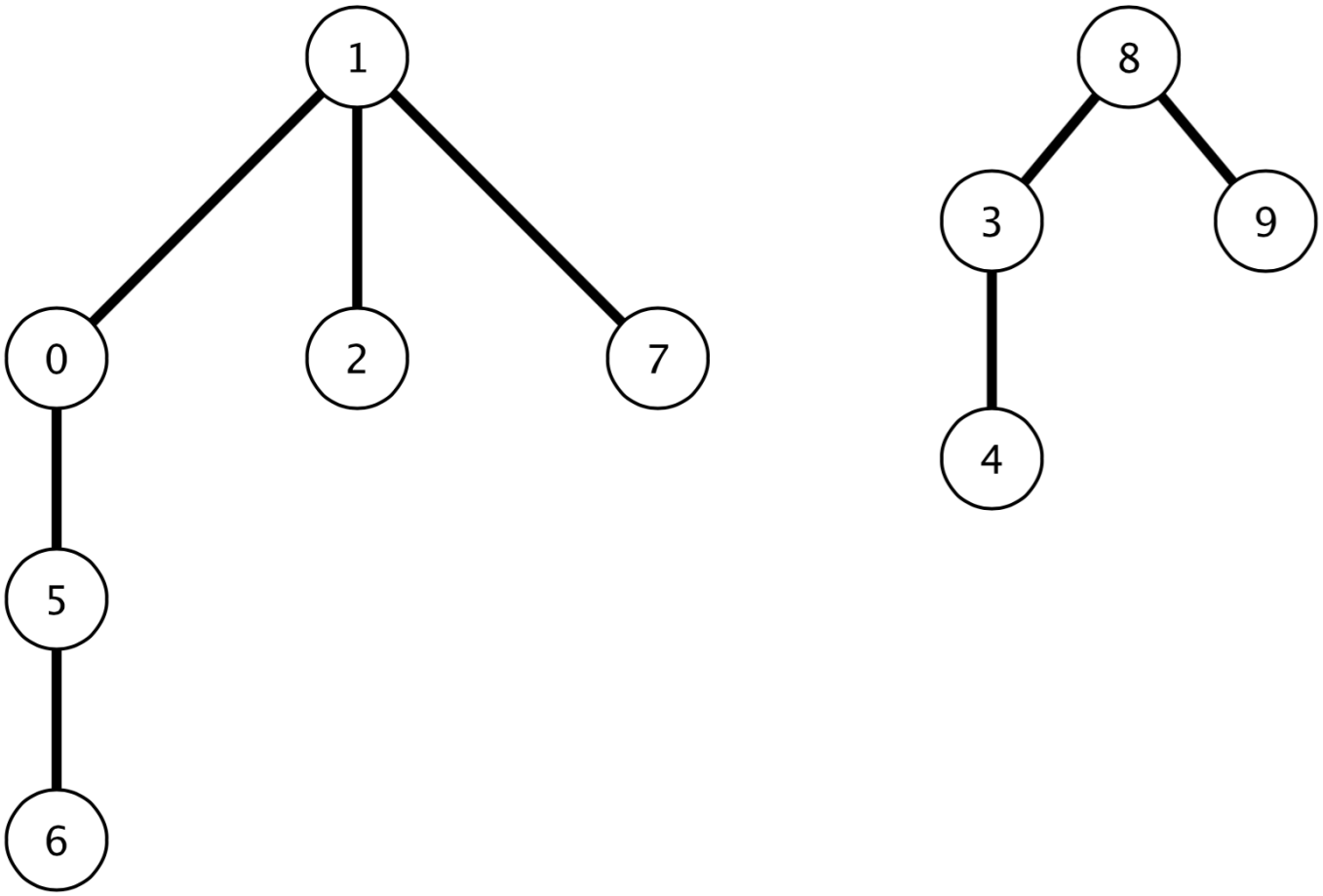
```
id[] = [0, 1, 1, 8, 3, 5, 5, 7, 8, 8]
union(5, 0)
parent(5) = 5 => Root

parent(0) = 0 => Root
id[] = [0, 1, 1, 8, 3, 0, 5, 7, 8, 8]
```



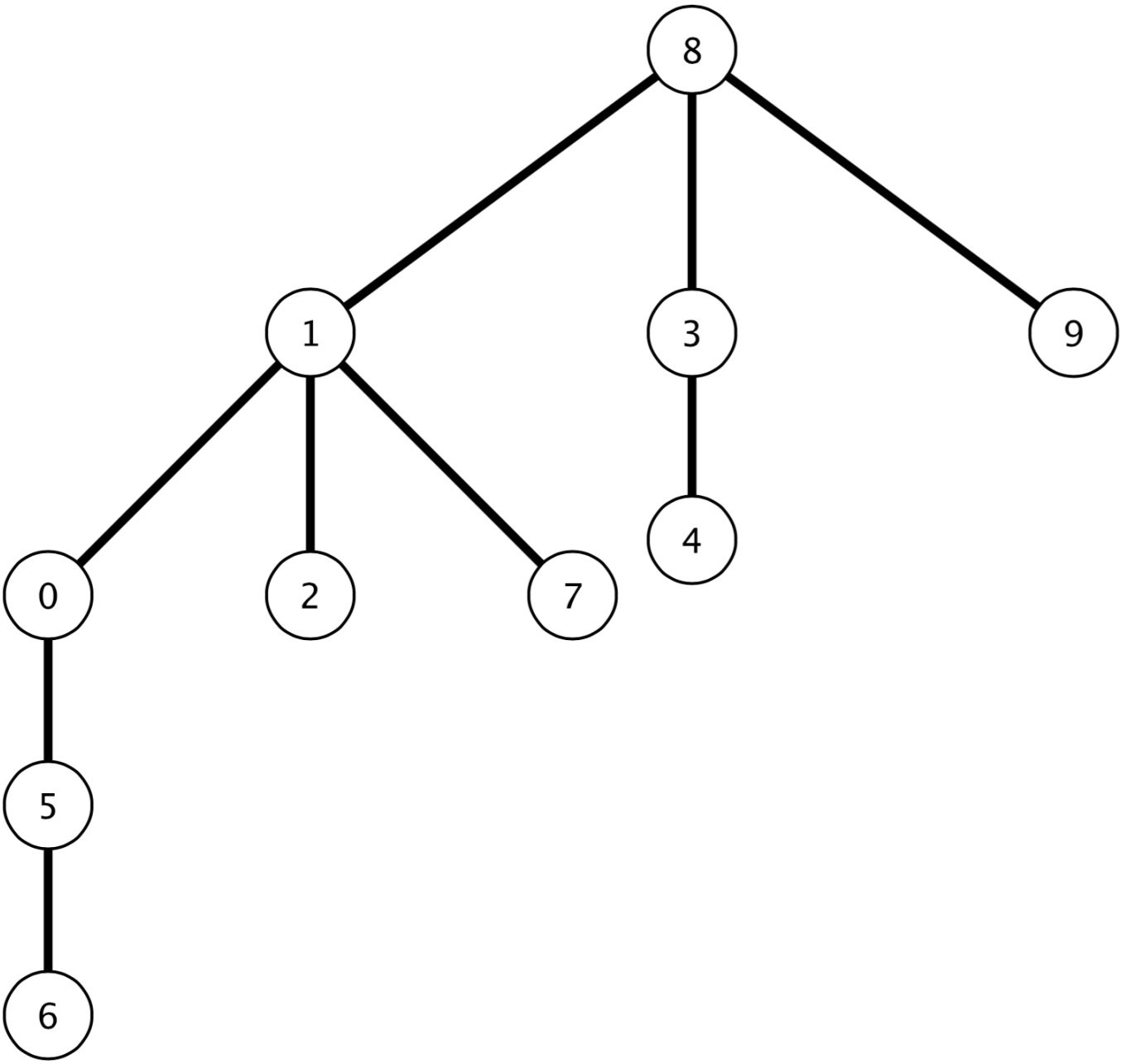

```
id[] = [0, 1, 1, 8, 3, 0, 5, 7, 8, 8]
union(7, 2)
parent(7) = 7 => Root

parent(2) = 1
parent(1) = 1 => Root
id[] = [0, 1, 1, 8, 3, 0, 5, 1, 8, 8]
```




```
id[] = [0, 1, 1, 8, 3, 0, 5, 1, 8, 8]
union(6, 1)
parent(6) = 5
parent(5) = 0
parent(0) = 0 => Root

parent(1) = 1 => Root
id[] = [1, 1, 1, 8, 3, 0, 5, 1, 8, 8]
```




```
id[] = [1, 1, 1, 8, 3, 0, 5, 1, 8, 8]
union(7, 3)
parent(7) = 1
parent(1) = 1 => Root

parent(3) = 8
parent(8) = 8 => Root
id[] = [1, 8, 1, 8, 3, 0, 5, 1, 8, 8]
```



The root of the component containing the first item, is now a child of the root of the component containing second item.

```
union(p, q) is
    rp = root(p)
    rq = root(q)

    id[rp] = rq
```

The final code is:

```
public class QuickUnionUF {

    private int[] id;
    private int N;

    public QuickUnionUF(int N) {
        this.N = N;
        this.id = new int[this.N];
        for(int i = 0; i < this.N; i++)
            id[i] = i;
    }

    private int root(int i) {
        while(id[i] != i) i = id[i];
        return i;
    }

    public boolean connected(int p, int q) {
        return root(p) == root(q);
    }

    public void union(int p, int q) {
        int rp = root(p);
        int rq = root(q);
        id[rp] = rq;
    }

}
```

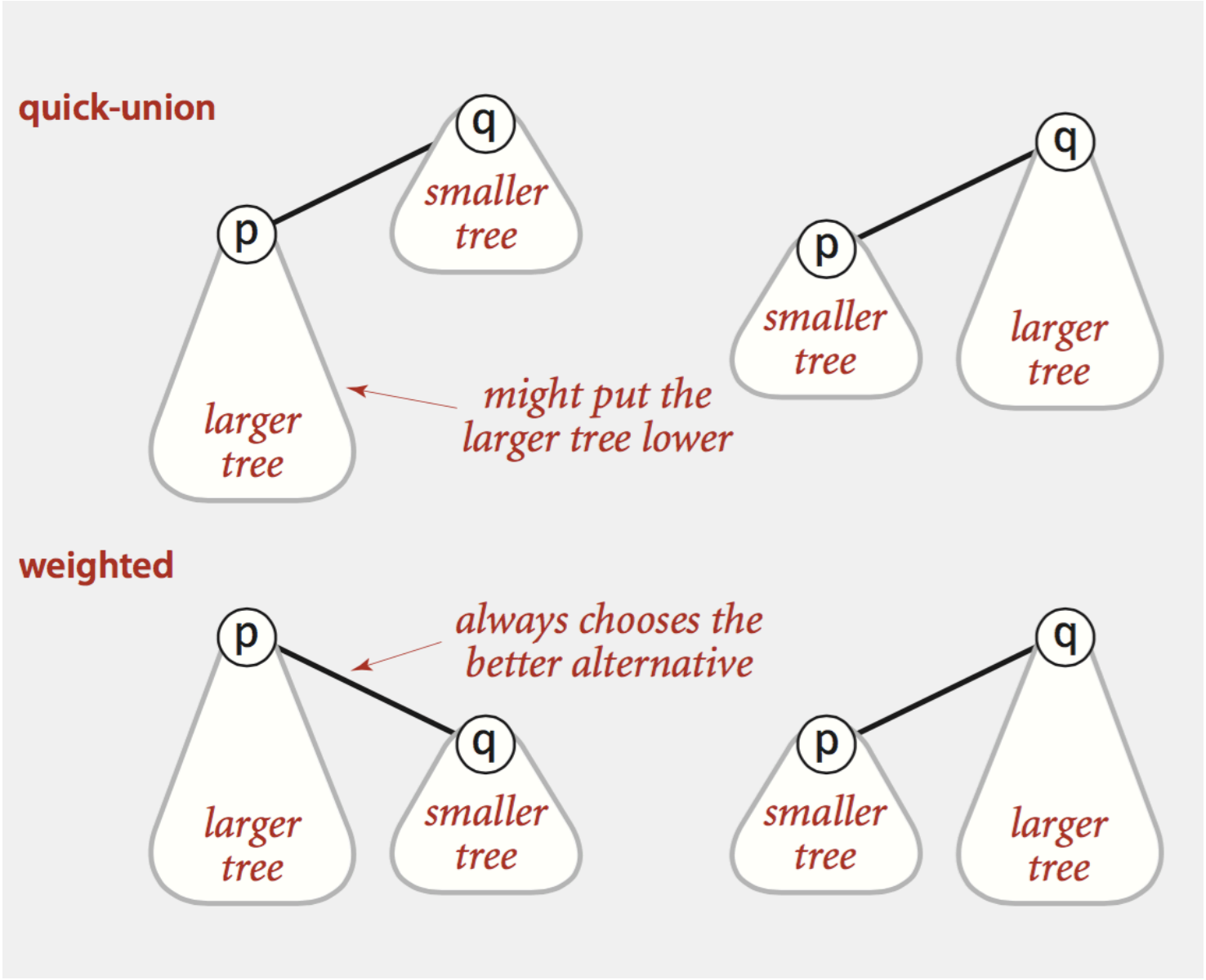
The tree can get too tall. So can we do better?

Union-Find: Improvements

Weighting (the smaller tree goes below)

When implementing quick-unions, take steps to avid large trees:

- Keep track of size of each tree
- Balance by linking root of smaller tree to root of the larger tree



Data structure: Same as quick-union, but mantain extra array `sz[i]`, to count number of objects in the tree rooted at `i`.

Find: Identical to quick-union.

```
return root(p) == root(q);
```

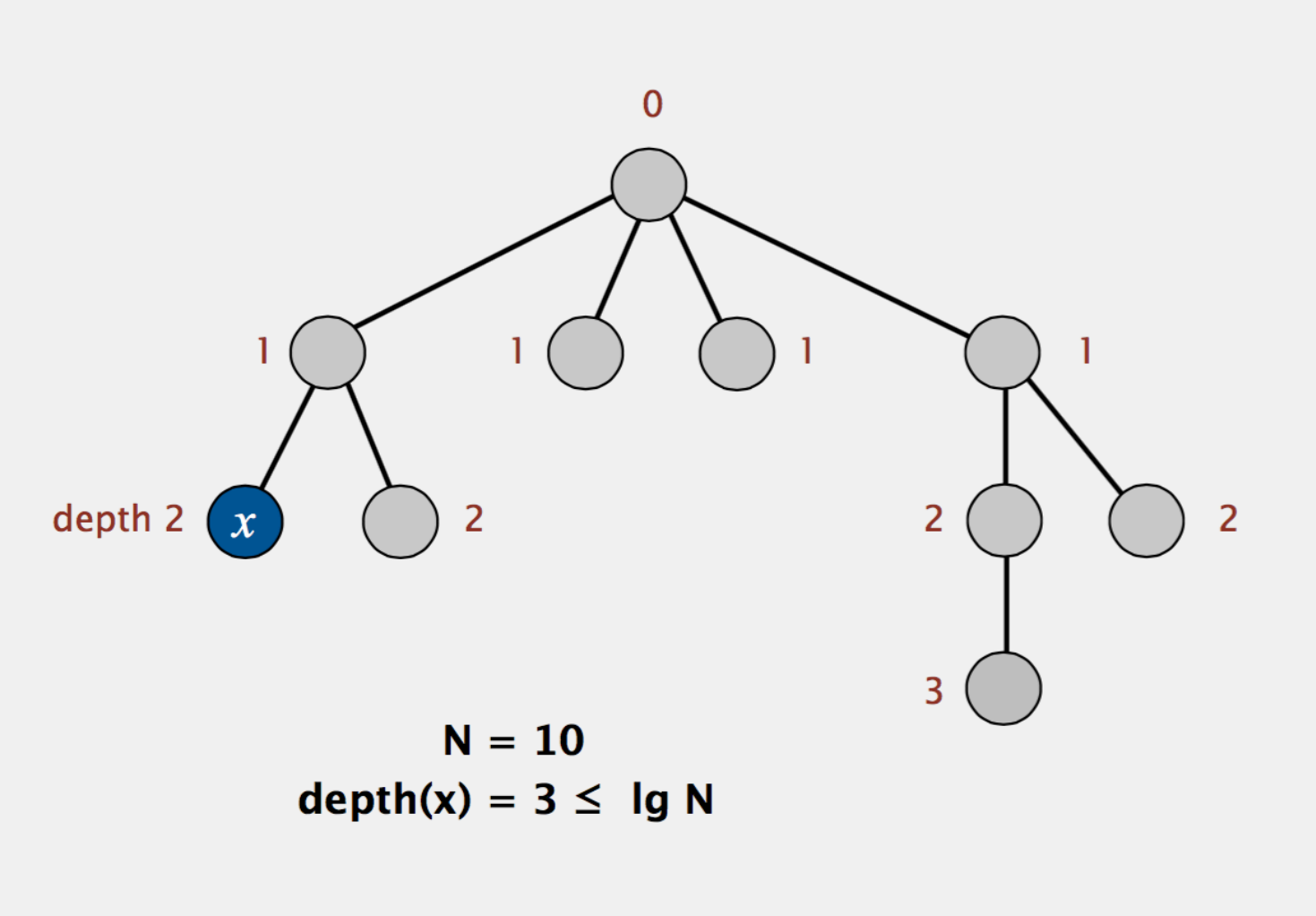
Takes time proportional to depth of `p` and `q`.

Union: Check the sizes and link the root of the smaller tree to the root of the largest tree. And update the size of the array.

```
int i = root(p);
int j = root(q);

if (i == j) return;

if (sz[i] < sz[j]) {
    id[i] = j;
    sz[j] += sz[i];
} else {
    id[j] = i;
    sz[i] += sz[j];
}
```

Takes constant time, given roots.

Proposition: Depth of any node `x` is at most `log(N)`



Proof: When does the depth of `x` increase?

Increase by 1 when tree `T1` containing `x` is merged into another tree `T2`.

- The size of the tree containing `x` at least doubles since `|T2| >= |T1|`.
- Size of tree containing `x`, can double at most `log(N)` times. Why?

If you start with one, and doubles `log(N)` times, you get `N`, and there's only `N` nodes in the tree.

```
A = {1, 2, 4, 8, 16, ..., N} => |A| = log(N)
```

Algorithm	Initialize	Union	Connected
quick-find	`N`	`N`	1
quick-union	`N`	`N`	`N`
weighted QU	`N`	`log(N)`	`log(N)`

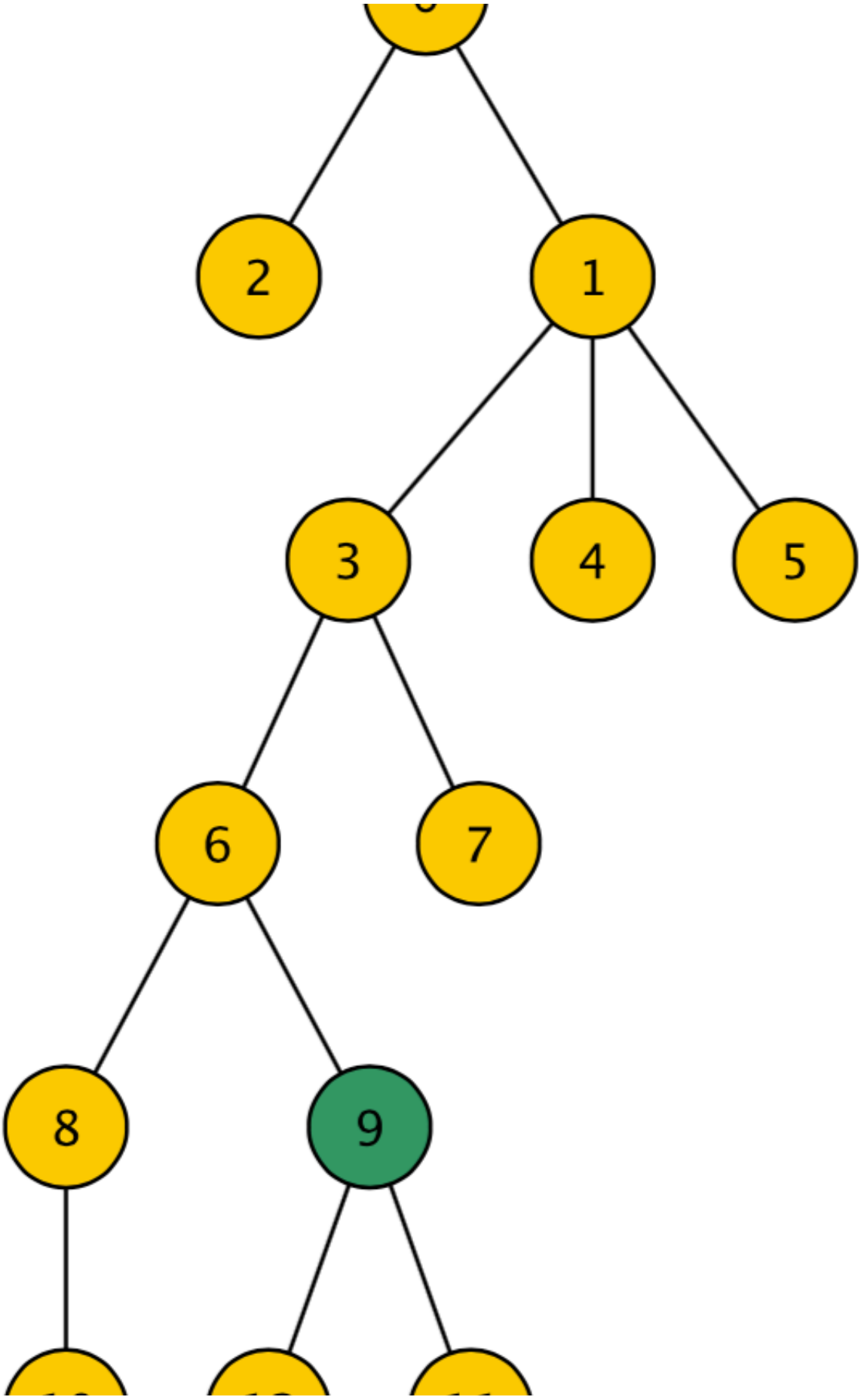
Path compression (keeps tree completely flat)

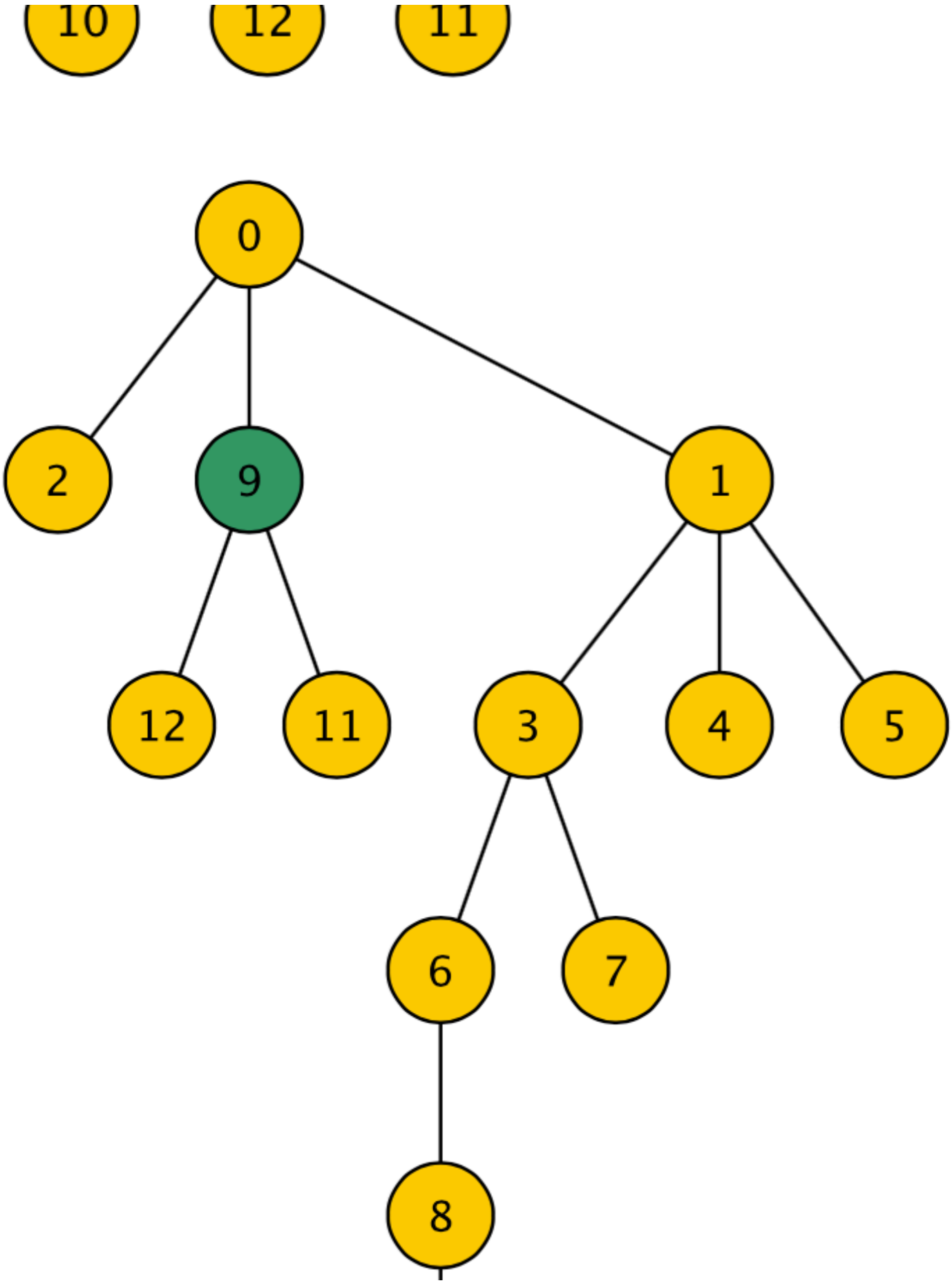
Just after computing the root of `p`, set the `id` of each examined node to point to that root. When finding the root of `p`, make every node in that path, point to the root.

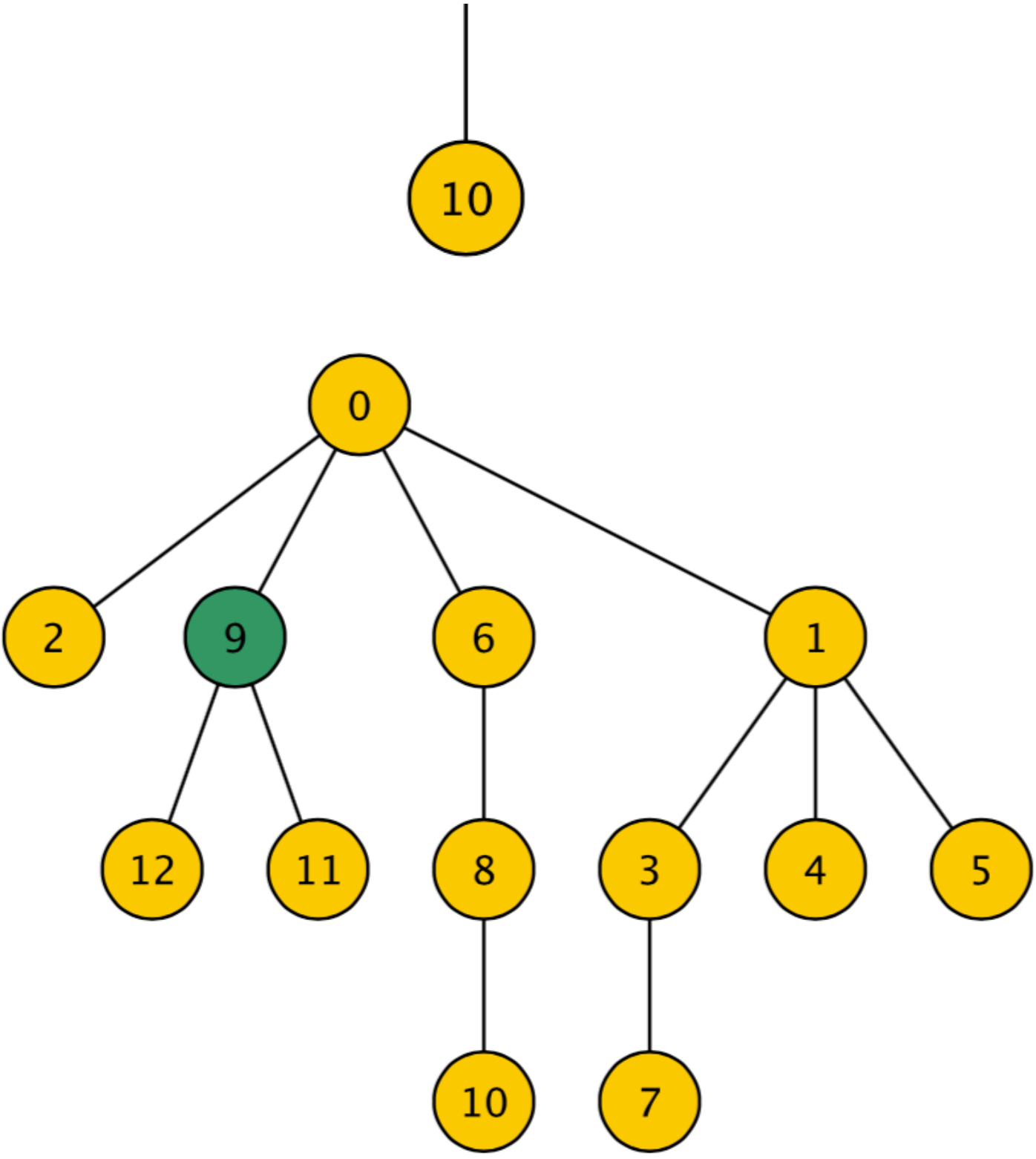
```
// Two-pass implementation
private int root(int i) {
    List<Integer> path = new ArrayList<Integer>();
    while(id[i] != i) {
        path.append(id[i]);
        i = id[i];
    }

    for(Integer node : path)
        id[node] = i;

    return i;
}

// Simpler one-pass variant
private int root(int i) {
    while(id[i] != i) {
        id[i] = id[id[i]];
        i = id[i];
    }
    return i;
}
```

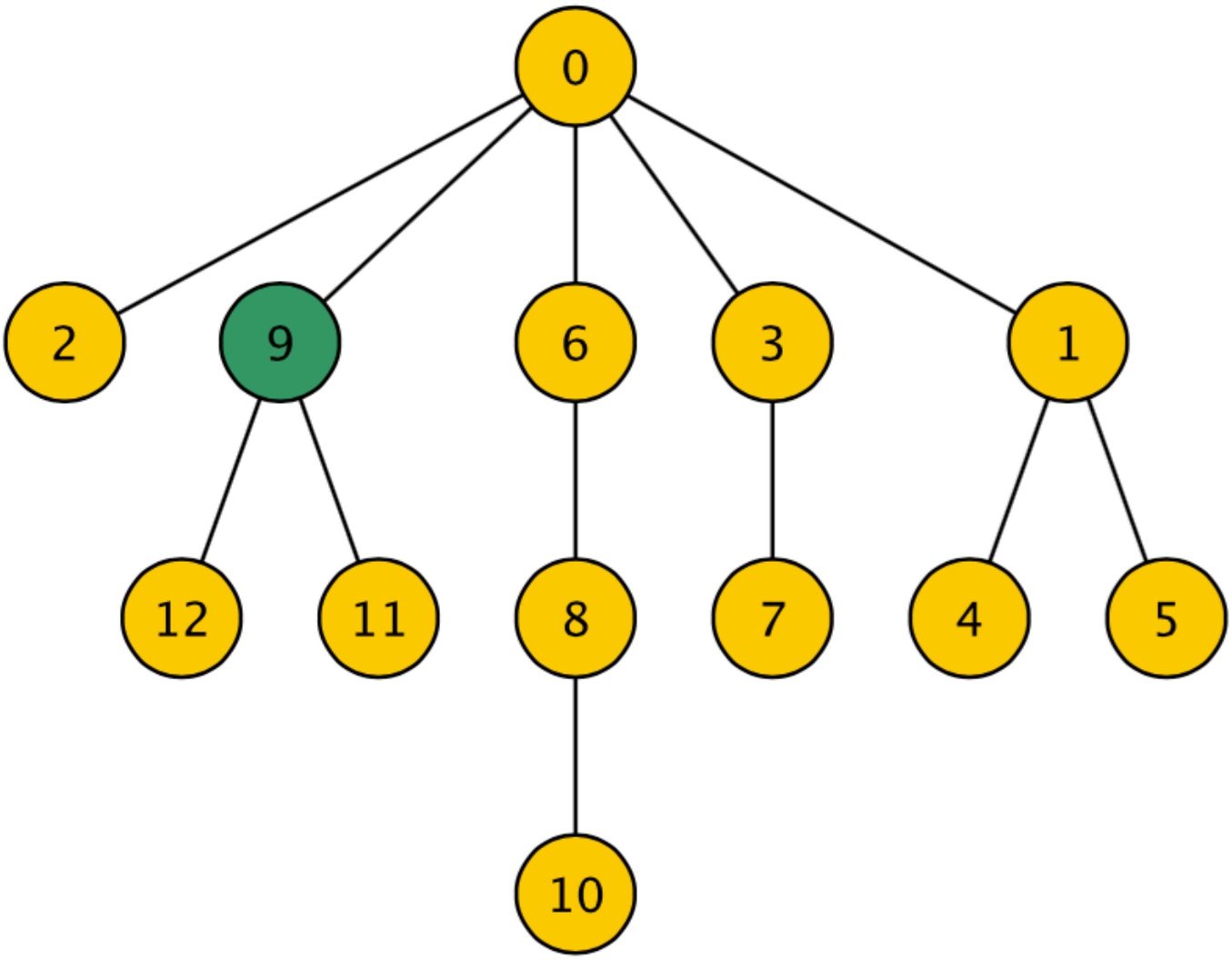
Example:

To find the root of the node labeled with 9, we compress its path up to the root.


2
1
3
4
5
6
7
8
9


10
12
11
0
2
9
1
12
11
3
4
5
6
7
8


10
0
2
9
6
1
12
11
8
3
4
5
10
7



Implementations

```java
public class QuickFindUF {
    public int[] id;
    public int count;

    public QuickFindUF(int N) {
        count = N;
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }

    public int find(int p) {
        return id[p];
    }

    public boolean connected(int p, int q) {
        return id[p] == id[q];
    }

    public void union(int p, int q) {
        int pID = id[p];   // needed for correctness
        int qID = id[q];   // to reduce the number of array accesses

        // p and q are already in the same component
        if (pID == qID) return;

        for (int i = 0; i < id.length; i++)
            if (id[i] == pID) id[i] = qID;
        count--;
    }
}
```

```java
public class QuickUnionUF {
    public int[] parent;
    public int count;

    public QuickUnionUF(int N) {
        parent = new int[N];
        count = N;
        for (int i = 0; i < N; i++)
            parent[i] = i;
    }

    public int find(int p) {
        while (p != parent[p])
            p = parent[p];
        return p;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) return;
        parent[rootP] = rootQ;
        count--;
    }
}
```

```java
public class WeightedQuickUnionUF {
    public int[] parent;
    private int[] size;
    public int count;

    public WeightedQuickUnionUF(int N) {
        count = N;
        parent = new int[N];
        size = new int[N];
        for (int i = 0; i < N; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    public int find(int p) {
        while (p != parent[p])
            p = parent[p];
        return p;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) return;

        // make smaller root point to larger one
        if (size[rootP] < size[rootQ]) {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        } else {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        }
        count--;
    }
}
```

```java
public class WeightedQuickUnionPathCompressionUF {
```

```java
    public int[] parent;
    private int[] size;
    public int count;

    public WeightedQuickUnionPathCompressionUF(int N) {
        count = N;
        parent = new int[N];
        size = new int[N];
        for (int i = 0; i < N; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    public int find(int p) {
        validate(p);
        int root = p;
        while (root != parent[root])
            root = parent[root];
        while (p != root) {
            int newp = parent[p];
            parent[p] = root;
            p = newp;
        }
        return root;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) return;

        // make smaller root point to larger one
        if (size[rootP] < size[rootQ]) {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        } else {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        }
        count--;
```

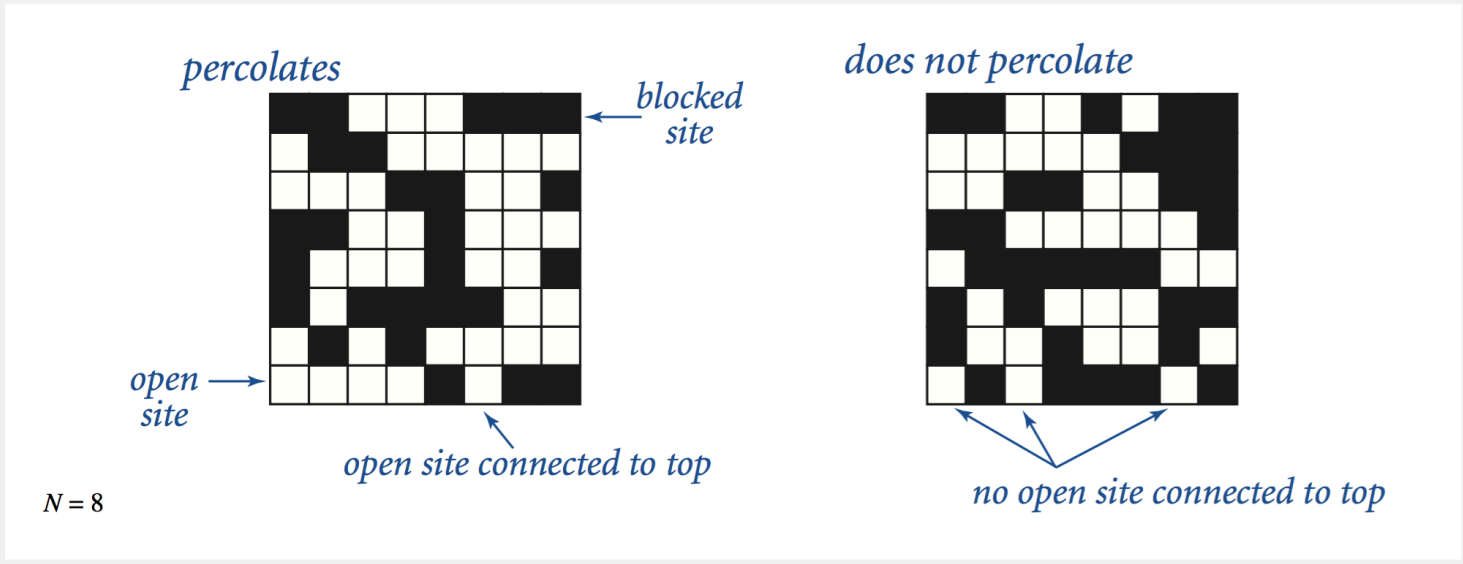
```
    }
}
```

Applications

- Percolation
- Games (Go, Hex)
- Dynamic connectivity
- Least common ancestor
- Equivalence of finite automata
- Hoshen-Kopelman algorithm
- Hinley-Milner polymorphic type inference
- Kruskal's minimum spanning tree algorithm
- Compiling equivalence statements in FORTRAN
- Morphological attribute openings and closings
- Matlab's `bwlabel()` function in image processing

Percolation

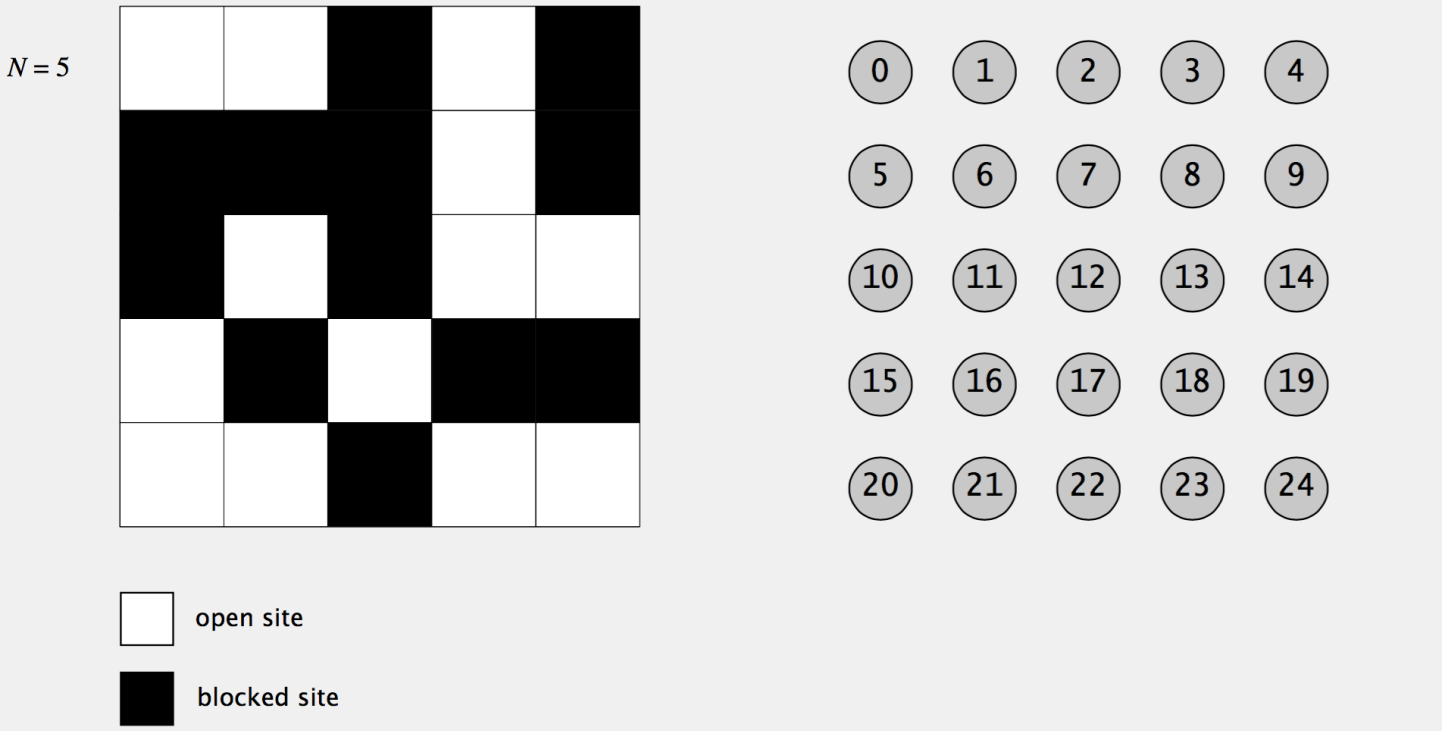
- `N` by `N` grid of sites
- Each site is open with probability `p`
- The system **percolates** iff top and bottom are connected by open sites

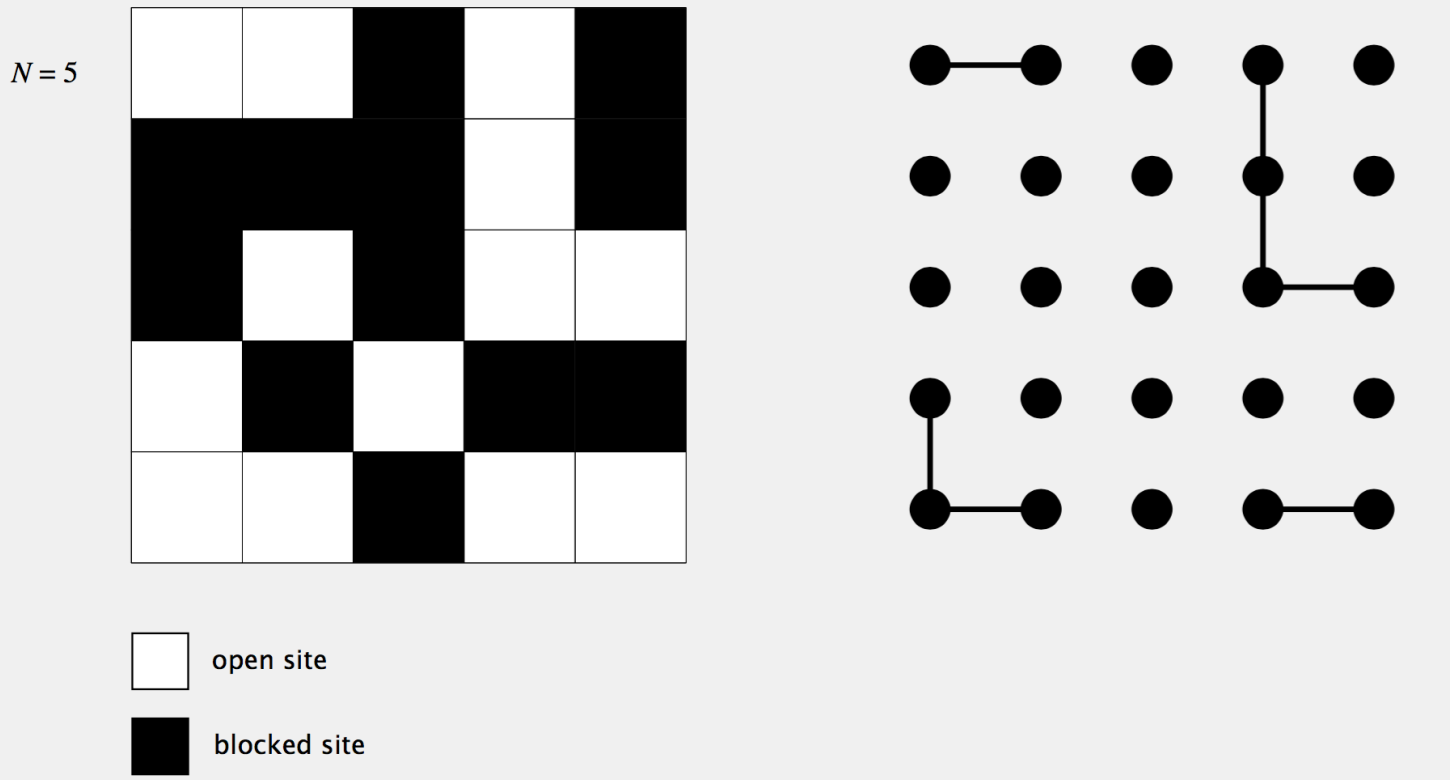


Likelihood of percolations, depends on site vacancy probability `p`. How do we know wheter is it going to percolate, to find the threshold.

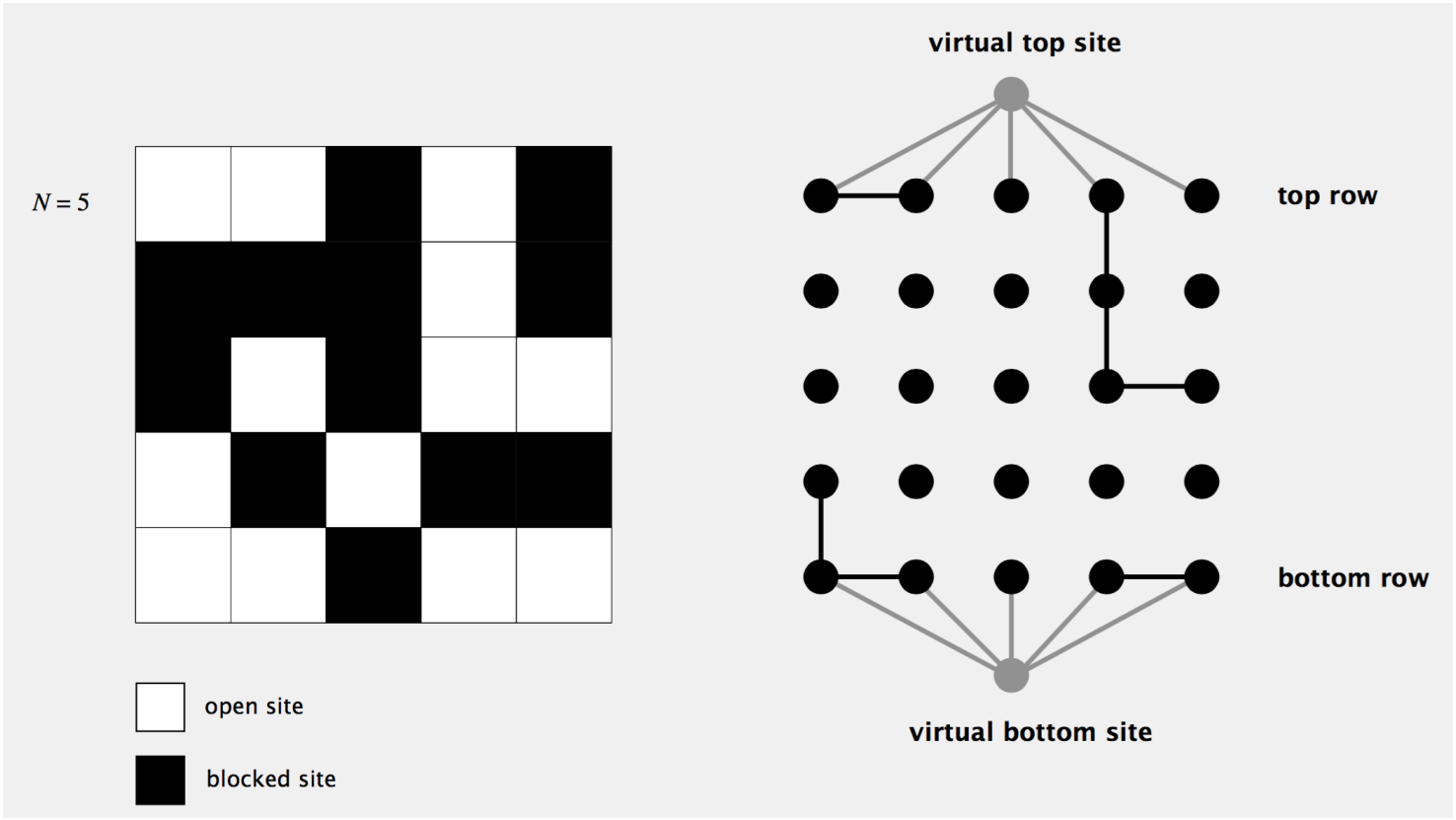
Monte Carlo simulation

- Initialize the whole grid to be blocked
- Declare random sites open until top cpnnected to bottom
- Vacancy percentage estimates `p*`

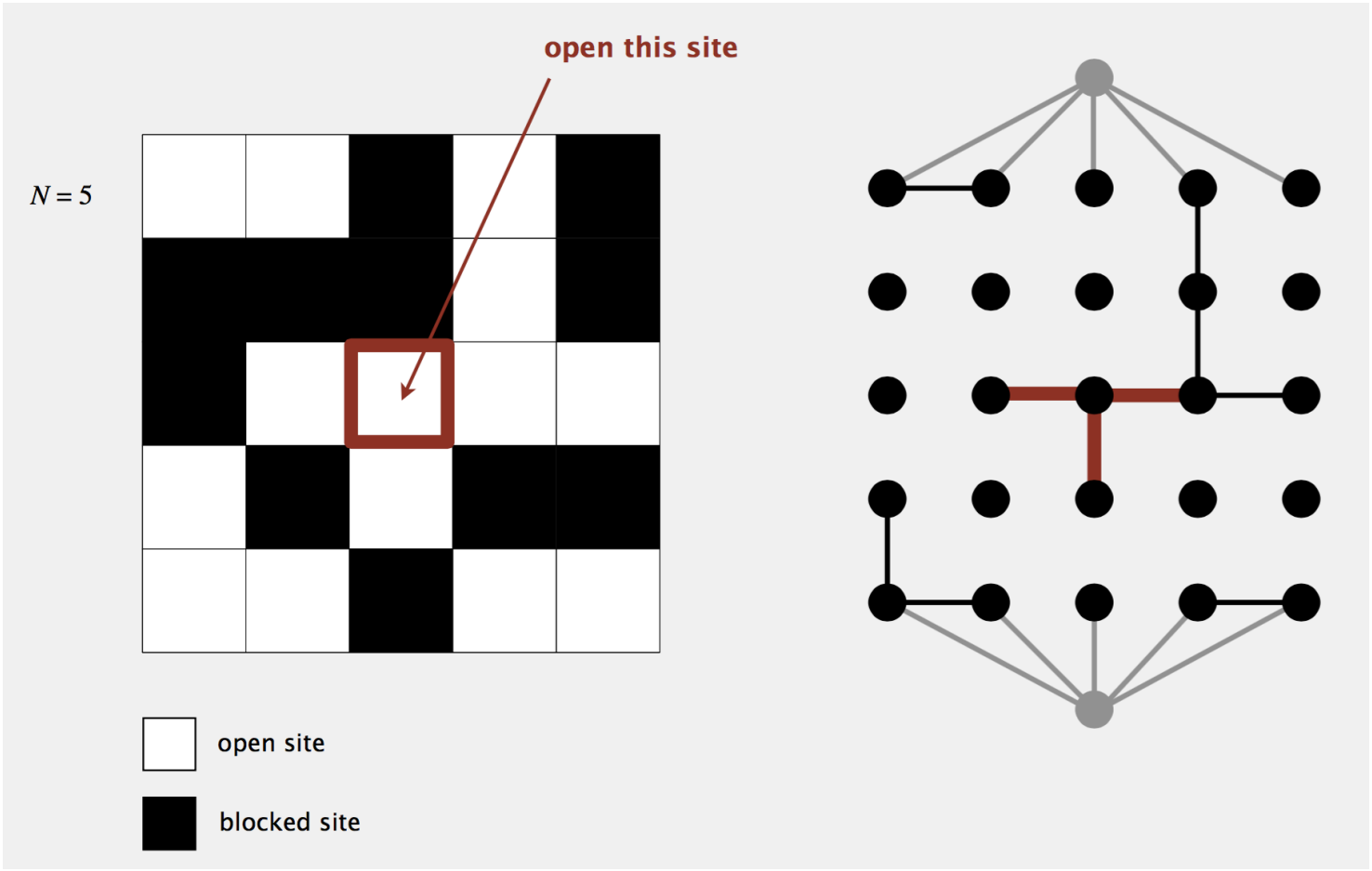




Percolates iff any site on bottom row is connected to site on topb row. `N^2` calls to `connected()`. Instedad, we create virtual sites at the top and at the botton. So it perculates iff the top virtual node, is connected to the top botton site.



To model opening a new site, we connect it to its adjacent open sites



The percolation threshold, is about 0.592746.