Data Structures and Algorithms in Java

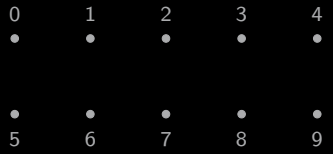
Algorithms and Data Structures: Union-Find

Outline

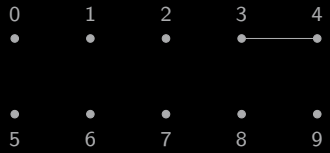
Dynamic Connectivity Problem

Dynamic Connectivity Problem

Dynamic Connectivity Problem

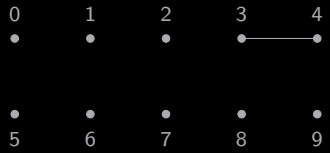


Dynamic Connectivity Problem



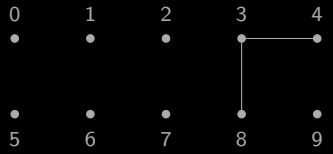
4 3

Dynamic Connectivity Problem



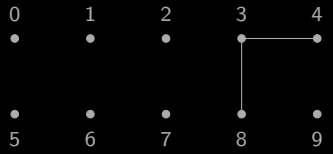
3 8

Dynamic Connectivity Problem



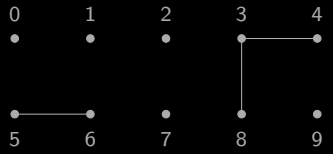
3 8

Dynamic Connectivity Problem



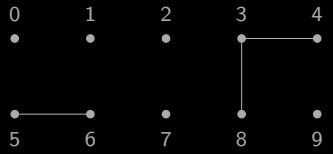
6 5

Dynamic Connectivity Problem



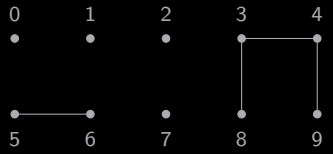
6 5

Dynamic Connectivity Problem



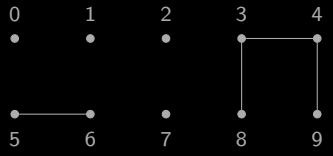
9 4

Dynamic Connectivity Problem



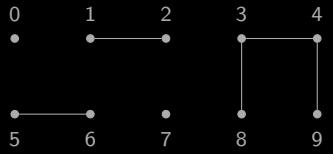
9 4

Dynamic Connectivity Problem



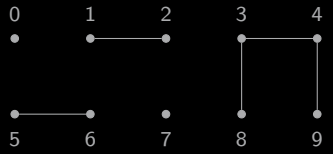
2 1

Dynamic Connectivity Problem



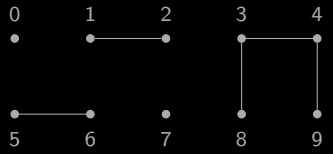
2 1

Dynamic Connectivity Problem



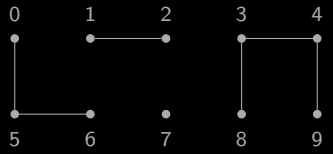
8 9

Dynamic Connectivity Problem



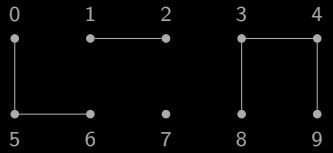
5 0

Dynamic Connectivity Problem



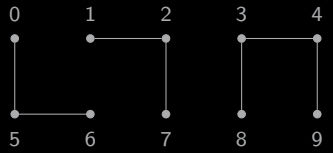
5 0

Dynamic Connectivity Problem



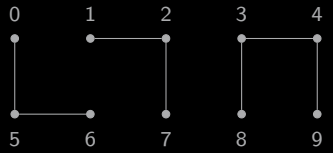
7 2

Dynamic Connectivity Problem



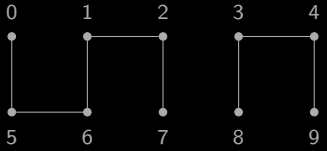
7 2

Dynamic Connectivity Problem



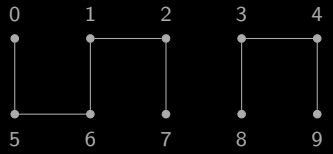
6 1

Dynamic Connectivity Problem



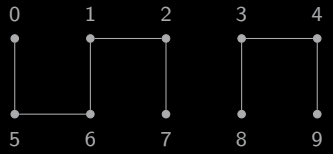
6 1

Dynamic Connectivity Problem



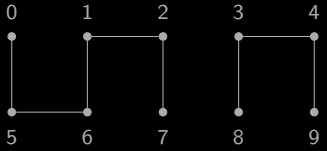
1 0

Dynamic Connectivity Problem



6 7

Dynamic Connectivity Problem



Dynamic Connectivity Problem

Dynamic Connectivity Problem

Notation

- Number of sites, n
- Site identifier, $i \in [0, n)$
- Component identifier, $i \in [0, n)$

Union-Find (UF)

Union-Find (UF)

dsa.UF

`int find(int p)`	returns the canonical site of the component containing site `p`
`int count()`	returns the number of components
`boolean connected(int p, int q)`	returns `true` if sites `p` and `q` belong to the same component, and `false` otherwise
`void union(int p, int q)`	connects sites `p` and `q`

Union-Find (UF)

Union-Find (UF)

Applications

- Percolation problem
- Kruskal's algorithm

DynamicConnectivity.java

Standard input	n (int) and a sequence of pairs of integers
Standard output	the pairs, whether they were merged, and if so, the number of components left

Union-Find (UF) · Example (Dynamic Connectivity)

DynamicConnectivity.java

Standard input	n (int) and a sequence of pairs of integers
Standard output	the pairs, whether they were merged, and if so, the number of components left

```
>_ ~/workspace/dsaj
$ _
```

DynamicConnectivity.java

Standard input	n (int) and a sequence of pairs of integers
Standard output	the pairs, whether they were merged, and if so, the number of components left

>_ ~/workspace/dsaj

```
$ cat data/tinyUF.txt
```

DynamicConnectivity.java

Standard input	n (int) and a sequence of pairs of integers
Standard output	the pairs, whether they were merged, and if so, the number of components left

~/workspace/dsaj

```
$ cat data/tinyUF.txt
10
4 3
3 8
...
6 7
$ _
```

DynamicConnectivity.java

Standard input	n (int) and a sequence of pairs of integers
Standard output	the pairs, whether they were merged, and if so, the number of components left

~/workspace/dsaj

```
$ cat data/tinyUF.txt
10
4 3
3 8
...
6 7
$ java DynamicConnectivity < data/tinyUF.txt
```

Union-Find (UF) · Example (Dynamic Connectivity)

DynamicConnectivity.java

Standard input	*n* (int) and a sequence of pairs of integers
Standard output	the pairs, whether they were merged, and if so, the number of components left

~/workspace/dsaj

```
$ cat data/tinyUF.txt
10
4 3
3 8
...
6 7
$ java DynamicConnectivity < data/tinyUF.txt
4 3 [merged, 9 components]
3 8 [merged, 8 components]
6 5 [merged, 7 components]
9 4 [merged, 6 components]
2 1 [merged, 5 components]
8 9
5 0 [merged, 4 components]
7 2 [merged, 3 components]
6 1 [merged, 2 components]
1 0
6 7
$ _
```

Union-Find (UF) · Example (Dynamic Connectivity)

</> DynamicConnectivity.java

```java
import dsa.WeightedQuickUnionUF;
import stdlib.StdIn;
import stdlib.StdOut;

public class DynamicConnectivity {
    public static void main(String[] args) {
        int n = StdIn.readInt();
        WeightedQuickUnionUF uf = new WeightedQuickUnionUF(n);
        while (!StdIn.isEmpty()) {
            int p = StdIn.readInt();
            int q = StdIn.readInt();
            StdOut.printf("%d %d", p, q);
            if (uf.connected(p, q)) {
                StdOut.println();
                continue;
            }
            uf.union(p, q);
            StdOut.println(" [merged, " + uf.count() + " components]");
        }
    }
}
```

Quick Find UF

Quick Find UF

dsa.QuickFindUF implements dsa.UF	
`QuickFindUF(int n)`	constructs an empty union-find data structure with `n` sites
`int find(int p)`	returns the canonical site of the component containing site `p`
`int count()`	returns the number of components
`boolean connected(int p, int q)`	returns `true` if sites `p` and `q` belong to the same component, and `false` otherwise
`void union(int p, int q)`	connects sites `p` and `q`

Quick Find UF

dsa.QuickFindUF implements dsa.UF	
`QuickFindUF(int n)`	constructs an empty union-find data structure with `n` sites
`int find(int p)`	returns the canonical site of the component containing site `p`
`int count()`	returns the number of components
`boolean connected(int p, int q)`	returns `true` if sites `p` and `q` belong to the same component, and `false` otherwise
`void union(int p, int q)`	connects sites `p` and `q`

Instance variables

- Component identifiers: `int[] id`
- Number of components: `int count`

Quick Find UF

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	3	4	5	6	7	8	9

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	3	4	5	6	7	8	9

4 3

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	3	3	5	6	7	8	9

4 3

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	3	3	5	6	7	8	9

3 8

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	8	8	5	6	7	8	9

3 8

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	8	8	5	6	7	8	9

6 5

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	8	8	5	5	7	8	9

6 5

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	8	8	5	5	7	8	9

9 4

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	8	8	5	5	7	8	8

9 4

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	2	8	8	5	5	7	8	8

2 1

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	1	8	8	5	5	7	8	8

2 1

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	1	8	8	5	5	7	8	8

8 9

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	1	8	8	5	5	7	8	8

5 0

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	1	8	8	0	0	7	8	8

5 0

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	1	8	8	0	0	7	8	8

7 2

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	1	8	8	0	0	1	8	8

7 2

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	1	8	8	0	0	1	8	8

6 1

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	1	1	1	8	8	1	1	1	8	8

6 1

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	1	1	1	8	8	1	1	1	8	8

1 0

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	1	1	1	8	8	1	1	1	8	8

6 7

Quick Find UF

i	0	1	2	3	4	5	6	7	8	9
id[i]	1	1	1	8	8	1	1	1	8	8

Quick Find UF

Quick Find UF

</> QuickFindUF.java 1/2

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class QuickFindUF implements UF {
    private int[] id;
    private int count;

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

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

    public int count() {
        return this.count;
    }

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

    public void union(int p, int q) {
        int pID = this.find(p);
        int qID = this.find(q);
        for (int i = 0; i < this.id.length; i++) {
            if (this.id[i] == pID) {
                this.id[i] = qID;
```

Quick Find UF

Quick Find UF

</> QuickFindUF.java 2/2

```
            }
        }
        this.count--;
    }

    public static void main(String[] args) {
        // Unit tests the data type
    }
}
```

Quick Find UF

Quick Find UF

Operation	T(n)
`QuickFindUF(int n)`	n
`int find(int p)`	1
`int count()`	1
`boolean connected(int p, int q)`	1
`void union(int p, int q)`	n

Quick Union UF

Quick Union UF

dsa.QuickUnionUF implements dsa.UF	
`QuickUnionUF(int n)`	constructs an empty union-find data structure with `n` sites
`int find(int p)`	returns the canonical site of the component containing site `p`
`int count()`	returns the number of components
`boolean connected(int p, int q)`	returns `true` if sites `p` and `q` belong to the same component, and `false` otherwise
`void union(int p, int q)`	connects sites `p` and `q`

Quick Union UF

dsa.QuickUnionUF implements dsa.UF	
`QuickUnionUF(int n)`	constructs an empty union-find data structure with `n` sites
`int find(int p)`	returns the canonical site of the component containing site `p`
`int count()`	returns the number of components
`boolean connected(int p, int q)`	returns `true` if sites `p` and `q` belong to the same component, and `false` otherwise
`void union(int p, int q)`	connects sites `p` and `q`

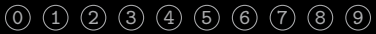
Instance variables

- Parent identifiers: `int[] parent`
- Number of components: `int count`

Quick Union UF

Quick Union UF

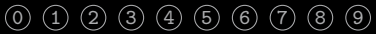
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	3	4	5	6	7	8	9



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	3	4	5	6	7	8	9

4 3



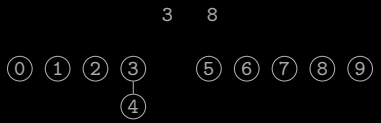
Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	3	3	5	6	7	8	9

4 3

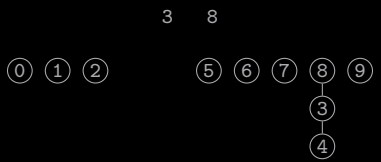
Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	3	3	5	6	7	8	9



Quick Union UF

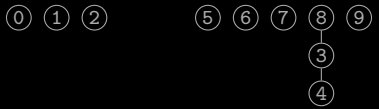
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	8	3	5	6	7	8	9



Quick Union UF

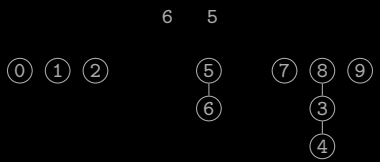
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	8	3	5	6	7	8	9

6 5



Quick Union UF

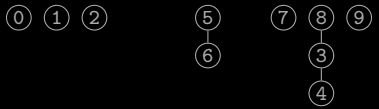
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	8	3	5	5	7	8	9



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	8	3	5	5	7	8	9

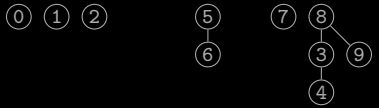
9 4



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	8	3	5	5	7	8	8

9 4



Quick Union UF

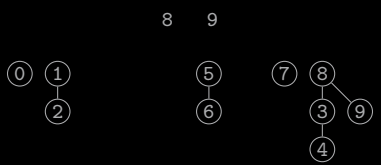
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	8	3	5	5	7	8	8

Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	1	8	3	5	5	7	8	8

Quick Union UF

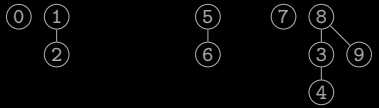
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	1	8	3	5	5	7	8	8



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	1	8	3	5	5	7	8	8

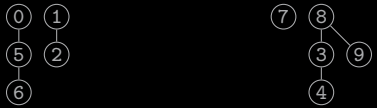
5 0



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	1	8	3	0	5	7	8	8

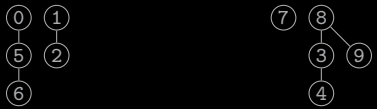
5 0



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	1	8	3	0	5	7	8	8

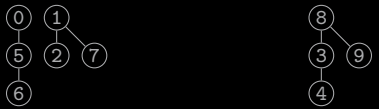
7 2



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	1	8	3	0	5	1	8	8

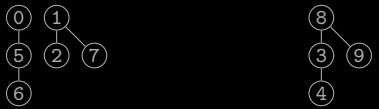
7 2



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	1	8	3	0	5	1	8	8

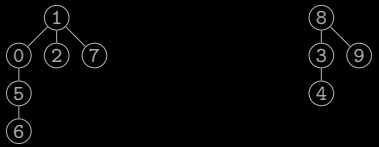
6 1



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	1	1	1	8	3	0	5	1	8	8

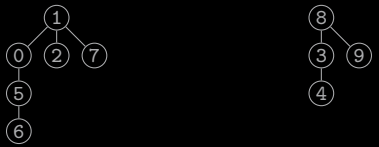
6 1



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	1	1	1	8	3	0	5	1	8	8

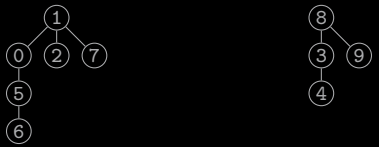
1 0



Quick Union UF

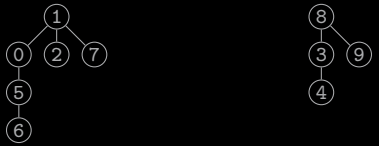
i	0	1	2	3	4	5	6	7	8	9
parent[i]	1	1	1	8	3	0	5	1	8	8

6 7



Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	1	1	1	8	3	0	5	1	8	8



Quick Union UF

Quick Union UF

</> QuickUnionUF.java 1/2

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class QuickUnionUF implements UF {
    private int[] parent;
    private int count;

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

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

    public int count() {
        return this.count;
    }

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

    public void union(int p, int q) {
        int rootP = this.find(p);
        int rootQ = this.find(q);
```

Quick Union UF

Quick Union UF

</> QuickUnionUF.java 2/2

```
36          this.parent[rootP] = rootQ;
37          this.count--;
38      }
39
40      public static void main(String[] args) {
41          // Unit tests the data type
42      }
43  }
```

Quick Union UF

Quick Union UF

Operation	T(n)
`QuickUnionUF(int n)`	n
`int find(int p)`	h
`int count()`	1
`boolean connected(int p, int q)`	h
`void union(int p, int q)`	h

h = tree height

Quick Union UF

Quick Union UF

Worst case scenario: consider $n = 5$ and pairs of sites $(0, 1), (1, 2), (2, 3)$, and $(3, 4)$

i	0	1	2	3	4
parent[i]	1	2	3	4	4

Weighted Quick Union UF

Weighted Quick Union UF

dsa.WeightedQuickUnionUF implements dsa.UF	
`WeightedQuickUnionUF(int n)`	constructs an empty union-find data structure with `n` sites
`int find(int p)`	returns the canonical site of the component containing site `p`
`int count()`	returns the number of components
`boolean connected(int p, int q)`	returns `true` if sites `p` and `q` belong to the same component, and `false` otherwise
`void union(int p, int q)`	connects sites `p` and `q`

Weighted Quick Union UF

dsa.WeightedQuickUnionUF implements dsa.UF	
`WeightedQuickUnionUF(int n)`	constructs an empty union-find data structure with `n` sites
`int find(int p)`	returns the canonical site of the component containing site `p`
`int count()`	returns the number of components
`boolean connected(int p, int q)`	returns `true` if sites `p` and `q` belong to the same component, and `false` otherwise
`void union(int p, int q)`	connects sites `p` and `q`

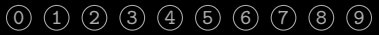
Instance variables

- Parent identifiers: `int[] parent`
- Component sizes: `int[] size`
- Number of components: `int count`

Weighted Quick Union UF

Weighted Quick Union UF

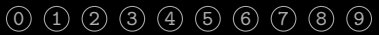
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	3	4	5	6	7	8	9
size[i]	1	1	1	1	1	1	1	1	1	1



Weighted Quick Union UF

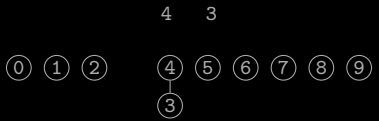
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	3	4	5	6	7	8	9
size[i]	1	1	1	1	1	1	1	1	1	1

4 3



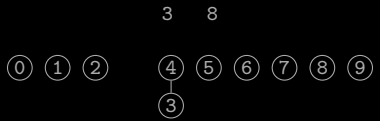
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	4	4	5	6	7	8	9
size[i]	1	1	1	1	2	1	1	1	1	1



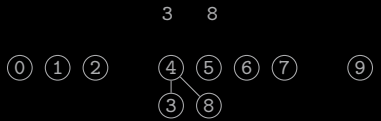
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	4	4	5	6	7	8	9
size[i]	1	1	1	1	2	1	1	1	1	1



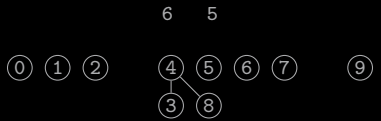
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	4	4	5	6	7	4	9
size[i]	1	1	1	1	3	1	1	1	1	1



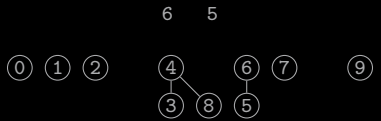
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	4	4	5	6	7	4	9
size[i]	1	1	1	1	3	1	1	1	1	1



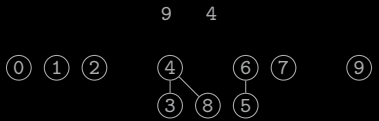
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	4	4	6	6	7	4	9
size[i]	1	1	1	1	3	1	2	1	1	1



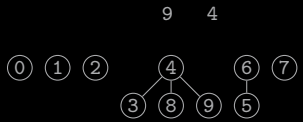
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	4	4	6	6	7	4	9
size[i]	1	1	1	1	3	1	2	1	1	1



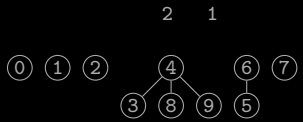
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	4	4	6	6	7	4	4
size[i]	1	1	1	1	4	1	2	1	1	1



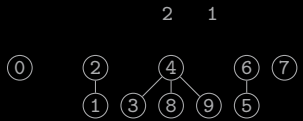
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	1	2	4	4	6	6	7	4	4
size[i]	1	1	1	1	4	1	2	1	1	1



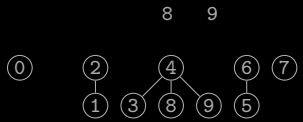
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	2	2	4	4	6	6	7	4	4
size[i]	1	1	2	1	4	1	2	1	1	1



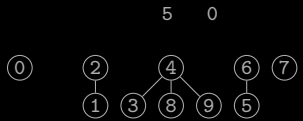
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	2	2	4	4	6	6	7	4	4
size[i]	1	1	2	1	4	1	2	1	1	1



Weighted Quick Union UF

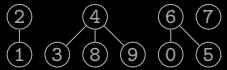
i	0	1	2	3	4	5	6	7	8	9
parent[i]	0	2	2	4	4	6	6	7	4	4
size[i]	1	1	2	1	4	1	2	1	1	1



Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	6	2	2	4	4	6	6	7	4	4
size[i]	1	1	2	1	4	1	3	1	1	1

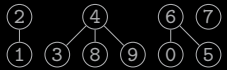
5 0



Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	6	2	2	4	4	6	6	7	4	4
size[i]	1	1	2	1	4	1	3	1	1	1

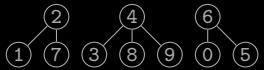
7 2



Weighted Quick Union UF

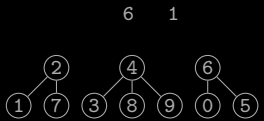
i	0	1	2	3	4	5	6	7	8	9
parent[i]	6	2	2	4	4	6	6	2	4	4
size[i]	1	1	3	1	4	1	3	1	1	1

7 2



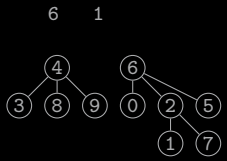
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	6	2	2	4	4	6	6	2	4	4
size[i]	1	1	3	1	4	1	3	1	1	1



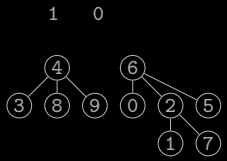
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	6	2	6	4	4	6	6	2	4	4
size[i]	1	1	3	1	4	1	6	1	1	1



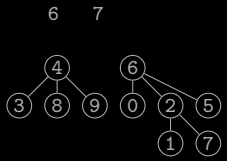
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	6	2	6	4	4	6	6	2	4	4
size[i]	1	1	3	1	4	1	6	1	1	1



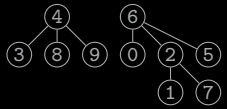
Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	6	2	6	4	4	6	6	2	4	4
size[i]	1	1	3	1	4	1	6	1	1	1



Weighted Quick Union UF

i	0	1	2	3	4	5	6	7	8	9
parent[i]	6	2	6	4	4	6	6	2	4	4
size[i]	1	1	3	1	4	1	6	1	1	1



Weighted Quick Union UF

Weighted Quick Union UF

</> WeightedQuickUnionUF.java 1/2

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class WeightedQuickUnionUF implements UF {
    private int[] parent;
    private int[] size;
    private int count;

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

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

    public int count() {
        return this.count;
    }

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

```

Weighted Quick Union UF

Weighted Quick Union UF

</> WeightedQuickUnionUF.java 2/2

```
36      public void union(int p, int q) {
37          int rootP = this.find(p);
38          int rootQ = this.find(q);
39          if (this.size[rootP] < this.size[rootQ]) {
40              this.parent[rootP] = rootQ;
41              this.size[rootQ] += this.size[rootP];
42          } else {
43              this.parent[rootQ] = rootP;
44              this.size[rootP] += this.size[rootQ];
45          }
46          this.count--;
47      }
48
49      public static void main(String[] args) {
50          // Unit tests the data type
51      }
52  }
```

Weighted Quick Union UF

Weighted Quick Union UF

Operation	T(n)
`WeightedQuickUnionUF(int n)`	n
`int find(int p)`	$\log n$
`int count()`	1
`boolean connected(int p, int q)`	$\log n$
`void union(int p, int q)`	$\log n$

Weighted Quick Union UF

Weighted Quick Union UF

Worst case scenario: consider $n = 5$ and pairs of sites $(0, 1), (1, 2), (2, 3)$, and $(3, 4)$

i	0	1	2	3	4
parent[i]	0	0	0	0	0