BM267 - Introduction to Data Structures

1. Introduction

Ankara University

Computer Engineering Department

Bulent Tugrul

Objectives

- Learn most important basic **data structures** used in programming computers. Such as Arrays, Linked List, Queues, Stacks, Trees…
- Learn most important basic **algorithms** used on these data structures. Such as Sorting, Searching, and Hashing algorithms …
- Learn basic tools to measure/compare the **efficiency** of these algorithms

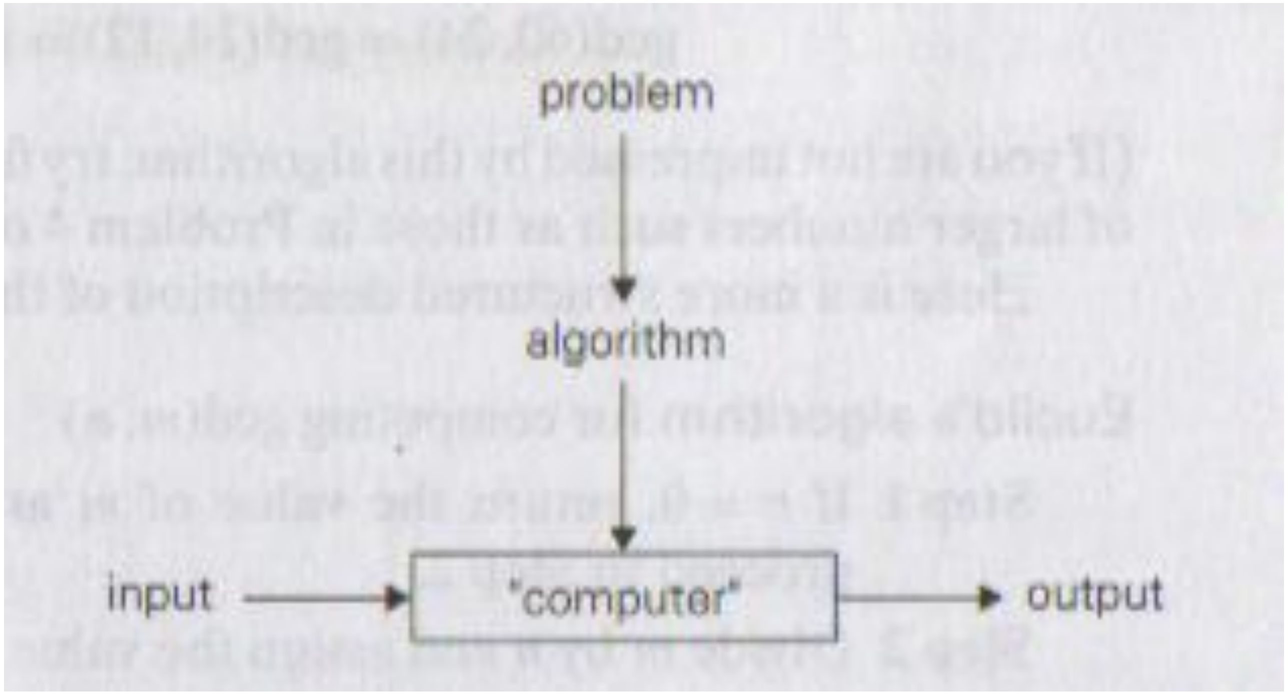
Programming Tools

There will be programming assignments and labs.

- Brush up your **C** skills, -- especially:
  - Built-in data types
  - Arrays
  - Structures, structure operations
  - Pointers, pointer operations
  - Dynamic memory allocation
  - **void** and **void*** data types

Definitions

- An **“Algorithm”** is method for solving problems that are suited for computer implementation.
- An **“Algorithm”** is a sequence of instructions for solving a problem for obtaining a required output for any legitimate input in a finite amount of time.



Definitions-2

- A “**data structure**” is a way to store and organize data in order to facilitate access and modifications.
- Algorithm + Data Structures = Program

Design process - overview

- Analyze the problem
  - Determine I / O data
  - Determine relevant data structures
  - Determine the operations needed on data structures.
  - Estimate the approximate space/time requirements
- Re-analyze the problem.

Implementation - overview

- Choose a programming Language
- Determine the data structure implementation method (Consecutive storage locations? Arrays? Dynamic memory?)
- Implement data structures
- Implement the algorithm
- Test and modify to improve the space/time requirements

Example-1: Greatest Common Divisor

- Greatest common divisor(gcd(m,n)) is defined as the largest integer that divides both m and n, where m and n are both non-zero positive integers, evenly i.e with a remainder of zero.
- 1st **Algorithm**:Calculating the common prime factors
  - Step 1 Find the prime factors of m.
  - Step 2 Find the prime factors of n.
  - Step 3 Identify all the common factors in the both sets.
  - Step 4 Compute the product of the all the common factors and return it as the gcd of the numbers given.

Example-1: Greatest Common Divisor(2)

For the numbers 60 and 24, we get

60 = 2*2*3*5

24 = 2*2*2*3

gcd(60, 24) = 2*2*3 = 12

Example-1: Greatest Common Divisor(3)

- **2nd Algorithm**: Consecutive integer checking algorithm.
  - ➢ Step 1 Assign the value of min{ m, n} to t
  - ➢ Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise go to Step 4.
  - ➢ Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise proceed to Step 4.
  - ➢ Step 4 Decrease the value of t by 1. Go to step 2.

For the numbers 60 and 24:

❖ Step 1 t= 24;

❖ Step 2 Go to Step 4

❖ Step 2 Now t=23;

Example-1: Greatest Common Divisor(4)

- Finally t becomes 12;
- Step 2 remainder of the division is zero
- Step 3 also the remainder of the division is zero and return the t value which is 12 and stop execution.

Example-1: Greatest Common Divisor(5)

3rd Algorithm:Euclid's algorithm.

- Step 1 if n=0, return the value of m as the answer and stop; otherwise proceed to step 2.
- Step 2 Divide m by n and assign the value of the remainder to r
- Step 3 Assign the value of n to m and value of r to n. Go to Step1.

- **Implementation of Euclid's algorithm**

```
int  Euclid( int m, int n){
        int r;
        while (n != 0 ){
                r = m mod n;
                m=n;
                n=r;
         }
        return m;}
```

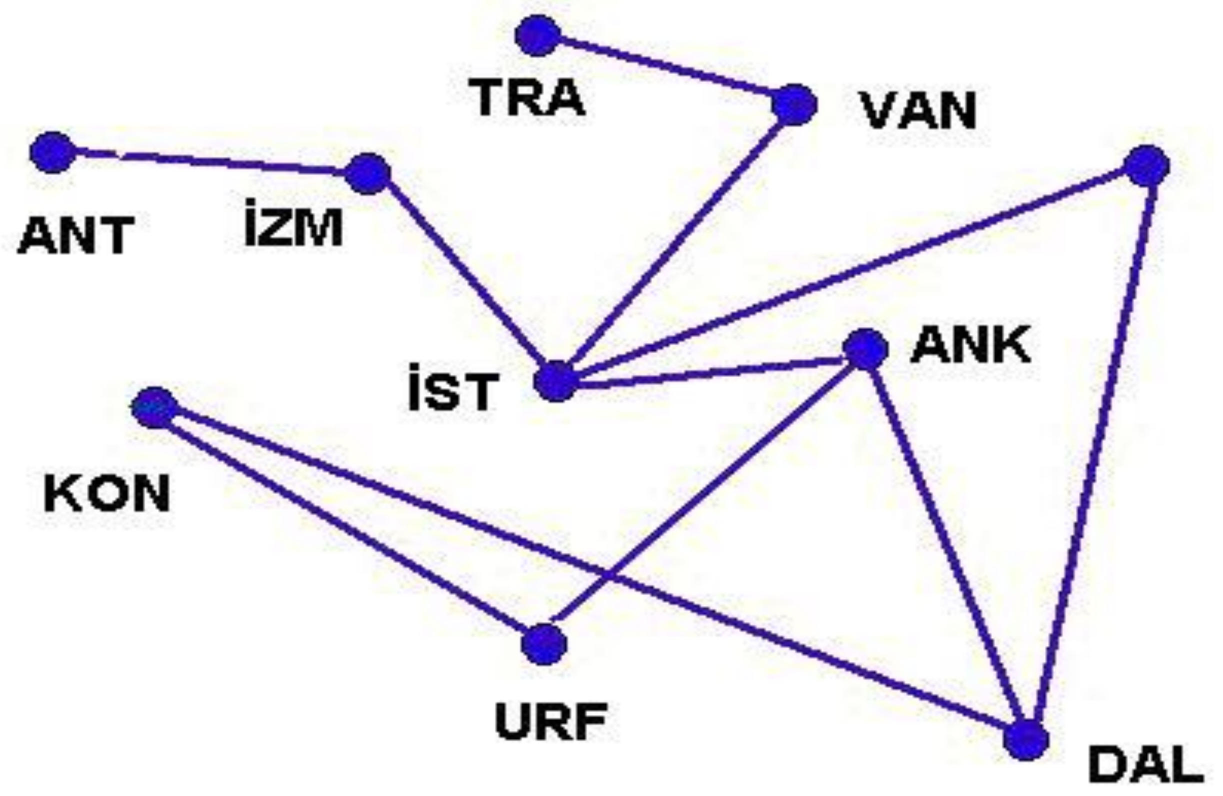
Example-2: Connectivity

- Assume: a set of **nodes** (representing cities, nets of networks, circuit board connections...)
- Given: pairs of nodes: p-q denotes a connection between p and q
- The relation '–' is symmetric:

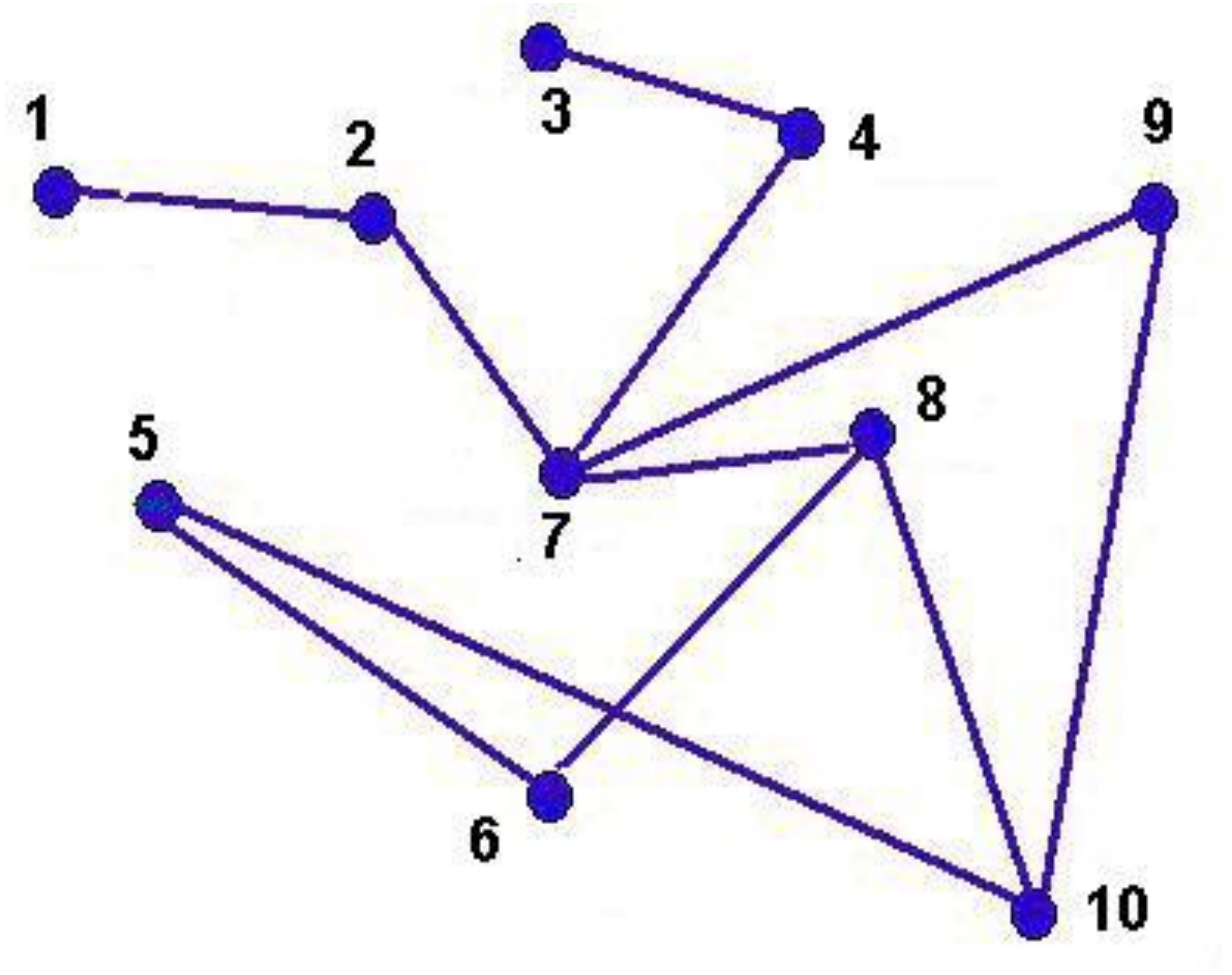
  If p-q is true, then q-p is true.
- The relation '–' is transitive:

  If p-q and q-r are true, then p-r is true.

Example-2: Connectivity (2)

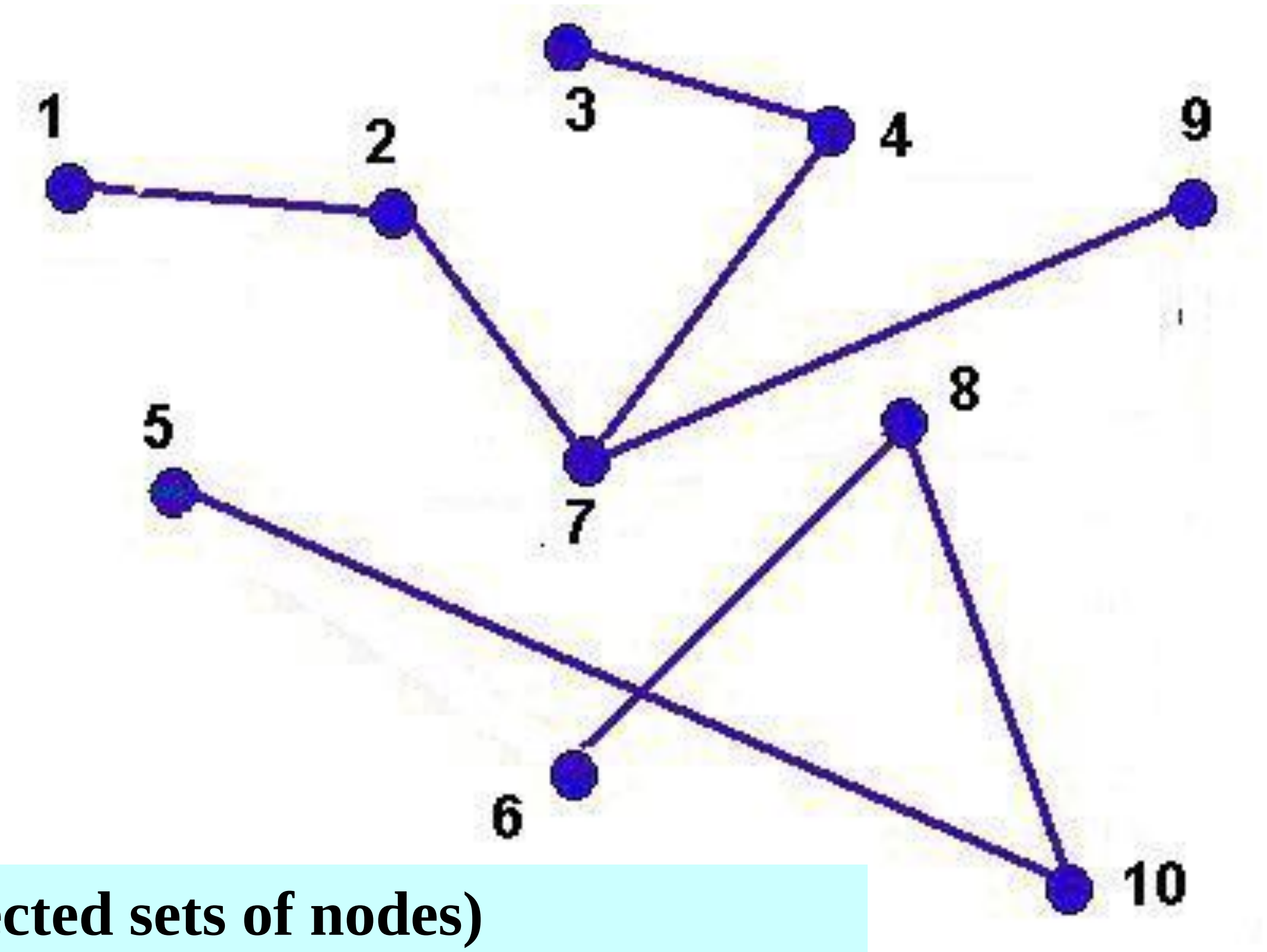


Example-2: Connectivity (3)



1-2
2-7
3-4
7-4
9-7
8-7
6-8
...

Example-2: Connectivity (4)

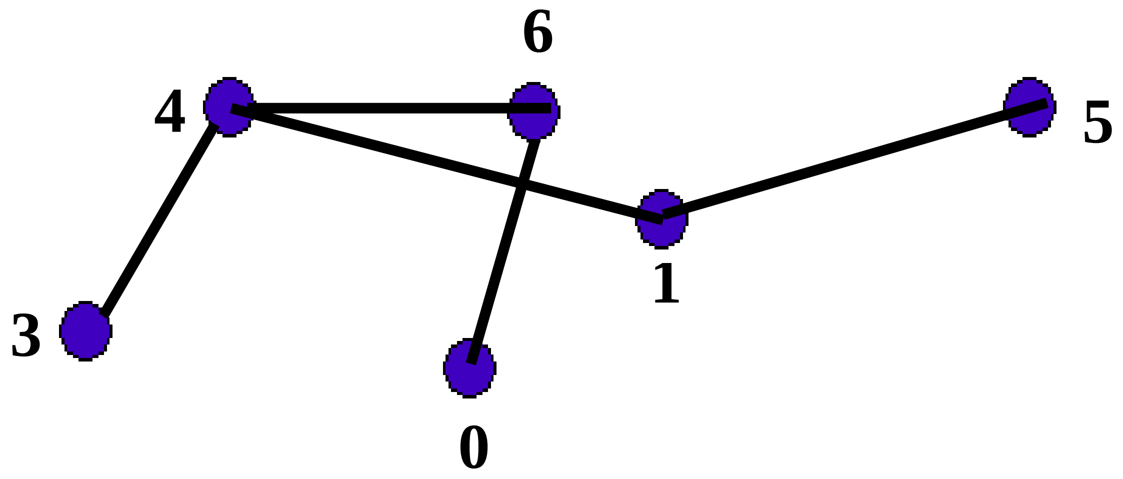


(Two disconnected sets of nodes)

Example-2: Connectivity (5)

Problem: Filter out redundant pairs – print out a pair only if it provides new information.

Input	3-4	4-6	6-0	3-6	4-0	4-1	1-3	1-5
Output	3-4	4-6	6-0			4-1		1-5



Example: Connectivity (6) - Abstract operations

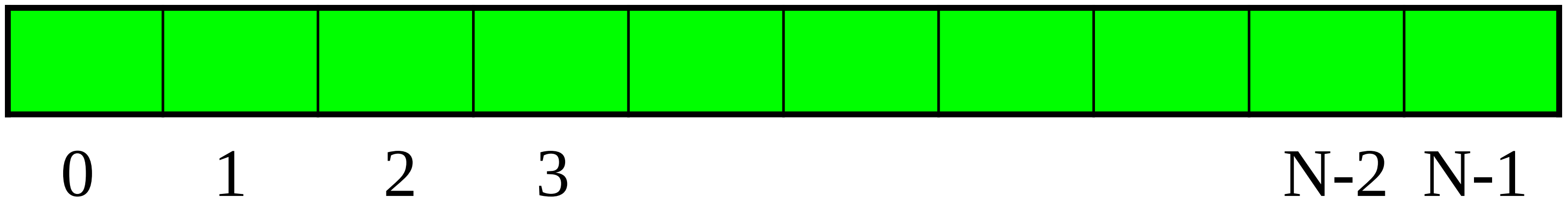
Each time we get a new pair:

- Determine whether it represents a new connection (**find** operation)
- Add the information represented by the pair to the ‘knowledge’ base (**union** operation)

Example: Connectivity (7) - Data Structure

We will:

- Assume an upper limit for the number of nodes, **N**.
- Use an array that has **N** elements (one int per node).
- Nodes are numbered (**0..N-1**), corresponding to array indices.



Example: Connectivity (8) - Representing sets

- Initially, array elements must indicate that all nodes are disconnected. (No two array elements will have the same value, each will contain its own index)
- This algorithm assumes that *p* and *q* are connected if and only if the *pth(id[p])* and *qth(id[q])* entries are equal.
- To implement the union for *p* and *q*, we go through the array, changing all the entries with the same name as *p* to have the same name as *q*.

0	1	2	3	.	.	.	.	N-2	N-1
0	1	2	3					N-2	N-1

Example: Quick-Find(1) - Initialization

```
N = 10000

ID[N]  is an array of int

FOR  (i = 0   TO  N-1)      ID[i] ← i
```

```
#define N 10000;
int i;
int ID[N];
for (i = 0; i < N; i++)
        ID[i] =  i ;
```

Example: Quick-Find(2) -Operations

```
WHILE (a new pair p, q exists)
{
        IF ID[p] == ID[q]          ← Find operation
        {
          Connection already noted, no output
        }
        ELSE                       ←Union operation
        {
          Connection not noted, add, output
        }
}
```

Example: Quick-Find(3) -Operations

```
N = 10000
ID[N]  is an array of int
FOR  (i = 0   TO  N-1)      ID[i] = i
WHILE (a new pair p-q exists)
{
    IF ID[p] == ID[q]            ← Find operation
    {   continue   }
    ELSE
    {      FOR i=0 to N-1        ←
              IF      ID[i] == ID[p]     ←Union operation
              THEN ID[i]  =  ID[q]       ←
           Output p-q
    }
}
```

Example: Quick-Find(4)

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

Example: Quick-Find(5)

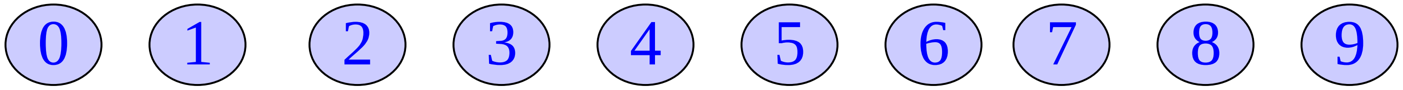
2 9	0	1	9	9	9	6	6	7	0	9
5 9	0	1	9	9	9	9	9	7	0	9
7 3	0	1	9	9	9	9	9	9	0	9
4 8	0	1	0	0	0	0	0	0	0	0
5 6	0	1	0	0	0	0	0	0	0	0
0 2	0	1	0	0	0	0	0	0	0	0

Example: Quick-Find(6)

6 1

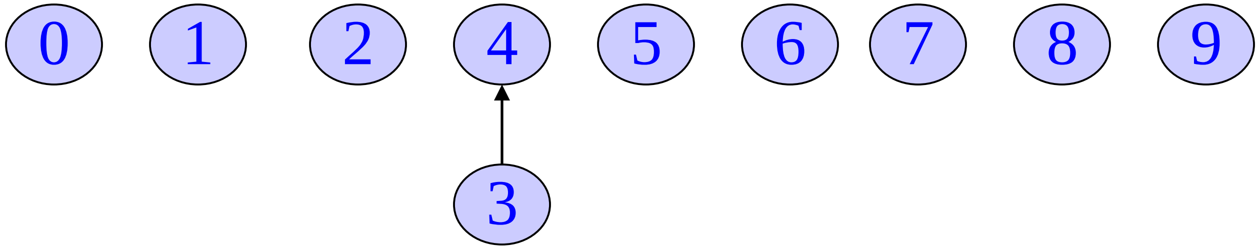
1	1	1	1	1	1	1	1	1	1

Example: Quick-Find(7)

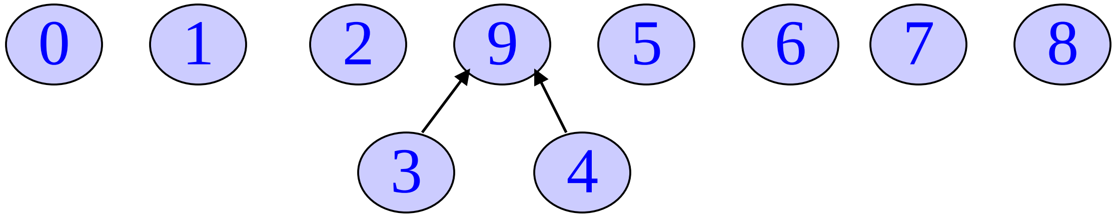


The initial state

3 4

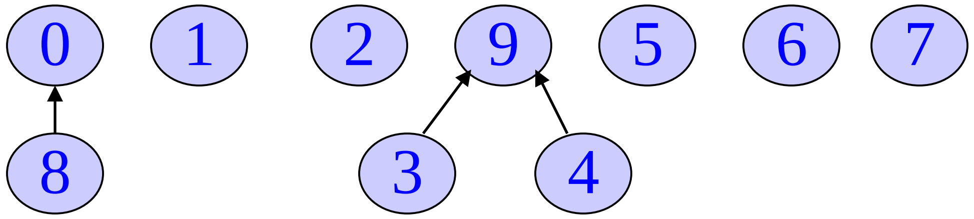


4 9

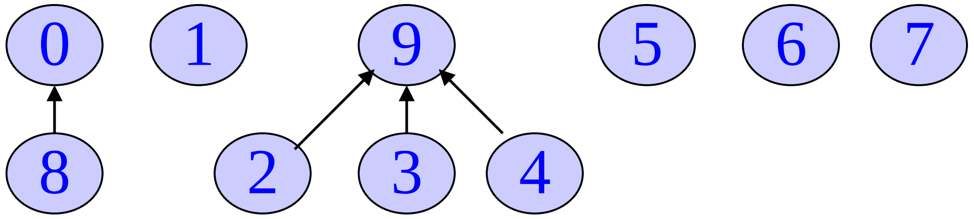


Example: Quick-Find(8)

8 0

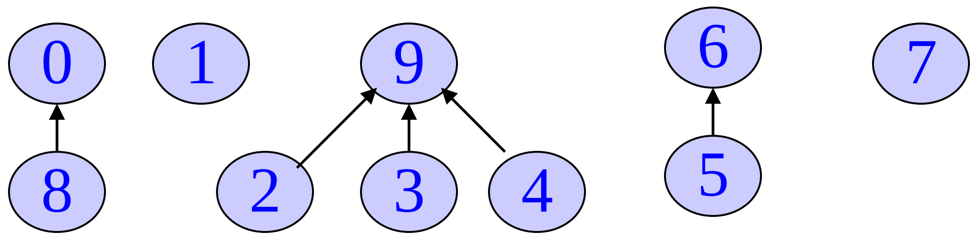


2 3



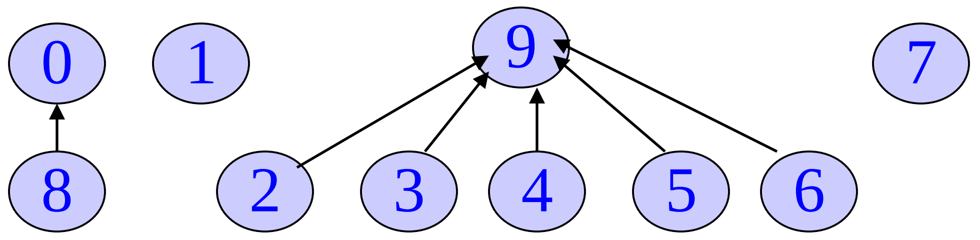
Example: Quick-Find(9)

5 6



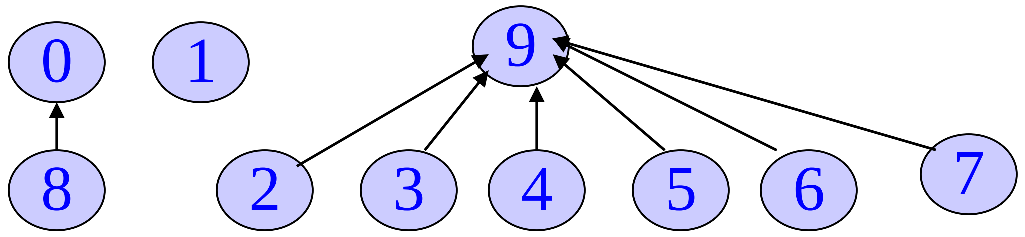
2 9 Do nothing

5 9

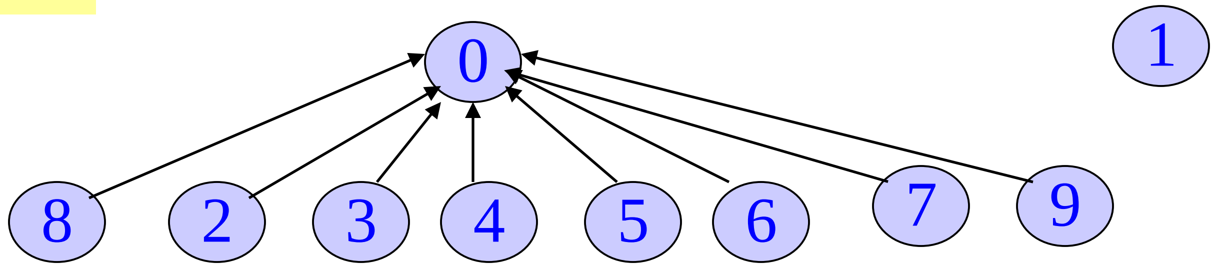


Example: Quick-Find(10)

7 3



4 8

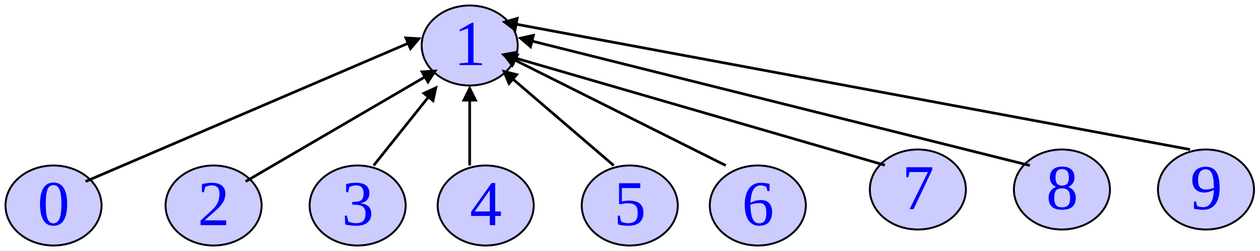


5 6 Do nothing

0 2 Do nothing

Example: Quick-Find(11)

6 1



Example: Quick-Find(12) - Analyzing Quick-find

The running time of the algorithm depends on actual data.

Find: one operation   (that's why it's called **quick-find**)

Union: N operations (at least)

Assuming a total of **M** union operations are required, the running time of the algorithm will be at least **MN**.

Example: Quick-Find(13) - Analyzing Quick-find

```
FOR  (i = 0   TO  N-1)      ID[i] = i
WHILE (a new pair p-q exists)        (once for each pair)
{
    IF ID[p]  ==  ID[q]                (once for each pair)
        continue
    Else                               (assume M unions)
    {   FOR i=0 to N-1                 (MN times)
            IF      ID[i] == ID[p]     (MN times)
            THEN ID[i]  =  ID[q]       (depends on data)
        Output p-q                     (depends on data)
    }
}
```

Example: Connectivity - Another approach

Let’s try to find a better (more efficient) algorithm that solves the same problem ...

... Without increasing space requirements.

Example: Quick–union – Data Structures

- Same data structure as previous algorithm.

0	1	2	3	.	.	.	.	N-2	N-1
0	1	2	3					N-2	N-1

- Connections are to be visualized as **tree branches**.
- Array elements indicate whether the nodes are in the same set by pointing to the **same root node**.
- Initially, array elements must indicate that all nodes are disconnected. (Each array element contains its own index)

Example: Quick–union(3) -Operations

```
N = 10000
ID[N]  is an array of int
FOR  (i = 0   TO  N-1)      ID[i] = i
WHILE (a new pair p, q exists)
{
    FOR (i=p;  i != ID[i];   i = ID[i])      ←
    FOR (j=q;  j != ID[j];  j = ID[j])       ←Find
    IF  i  !=  j                             ←
    THEN {   ID[i] = j,  output  p-q }
}
            ↑ Union operation
```

Example: Quick–union(4)

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

Example: Quick–union(5)

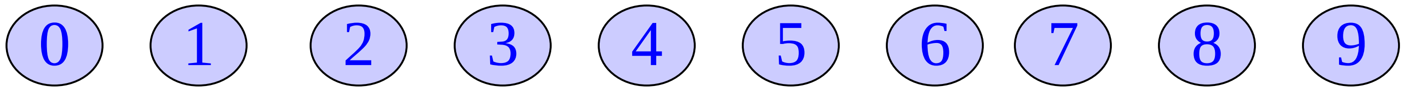
2 9	0	1	9	4	9	6	6	7	0	9
5 9	0	1	9	4	9	6	9	7	0	9
7 3	0	1	9	4	9	6	9	9	0	9
4 8	0	1	9	4	9	6	9	9	0	0
5 6	0	1	9	4	9	6	9	9	0	0
0 2	0	1	9	4	9	6	9	9	0	0

Example: Quick–union(6)

6 1

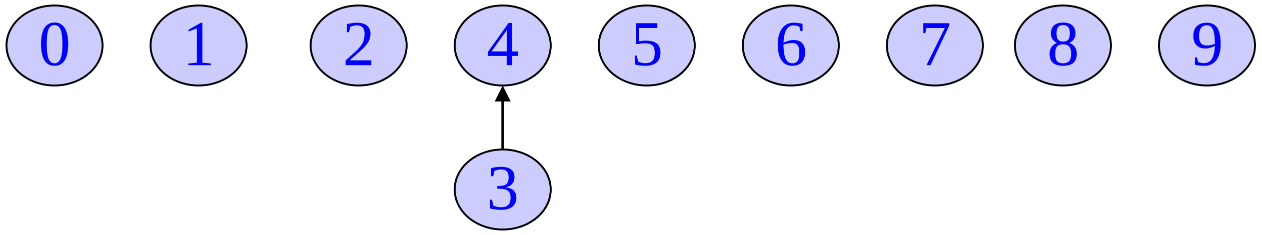
1	1	9	4	9	6	9	9	0	0

Example: Quick–union(7)

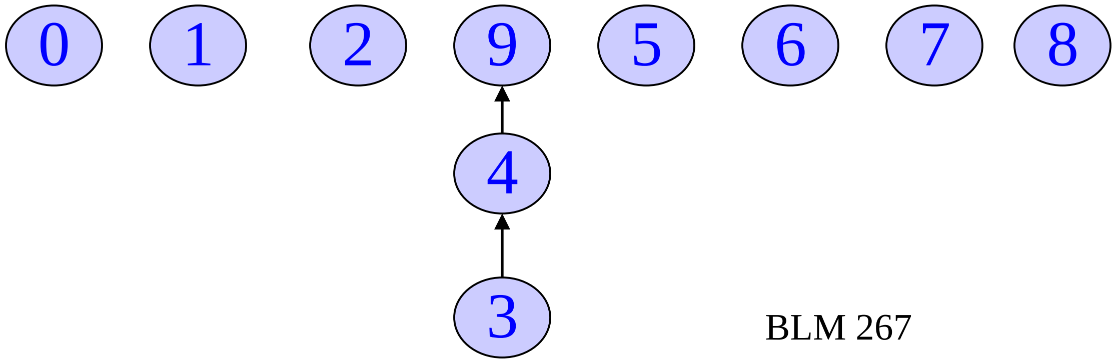


The initial state

3 4

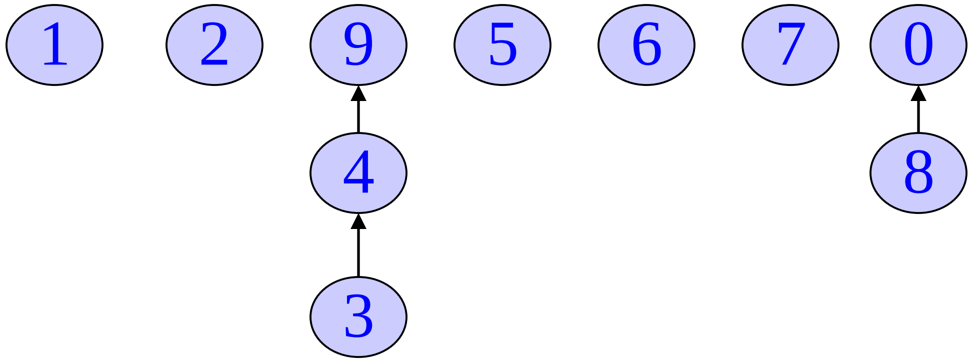


4 9

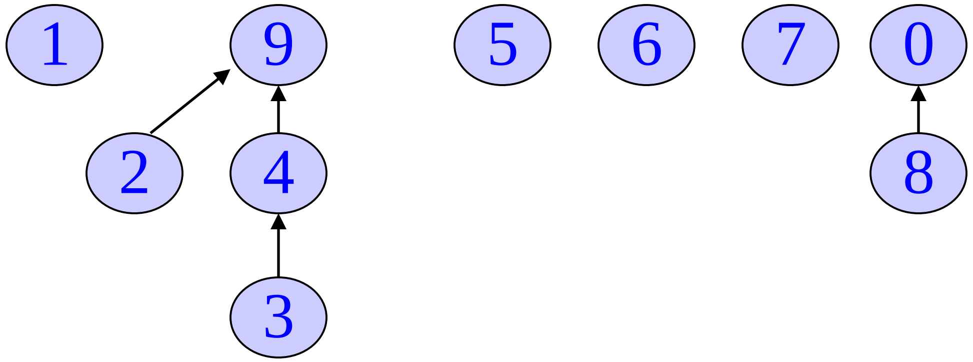


Example: Quick–union(8)

8 0

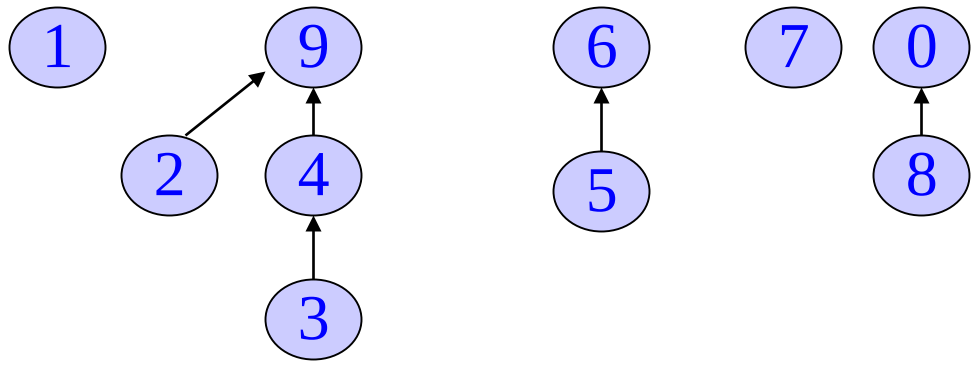


2 3



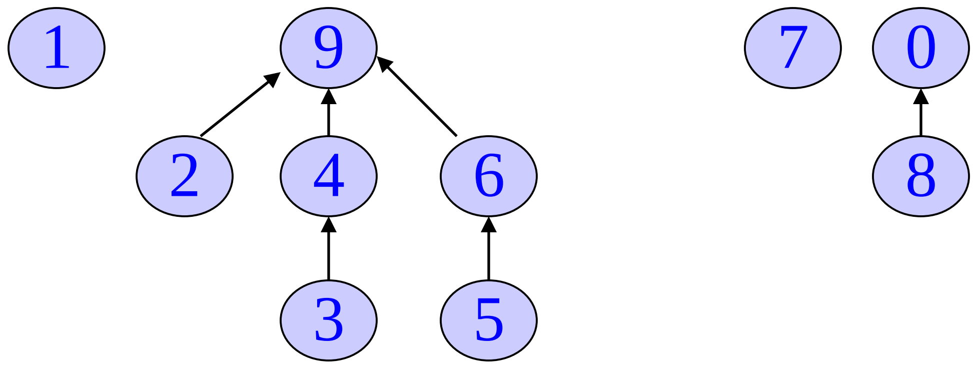
Example: Quick–union(9)

5 6



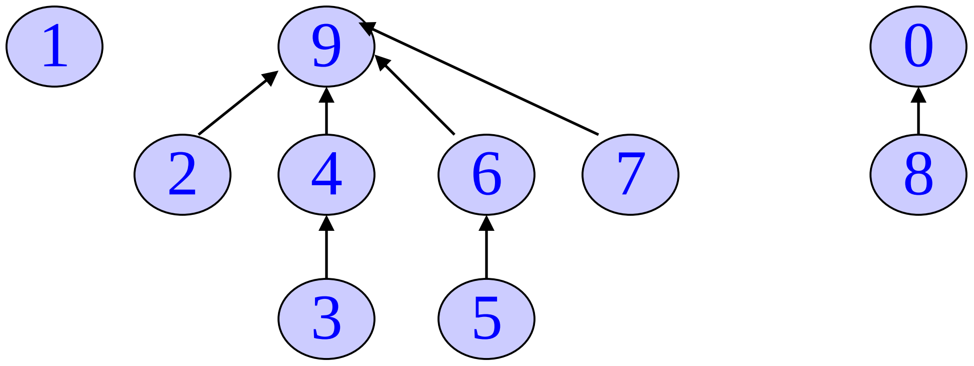
2 9 Do nothing

5 9

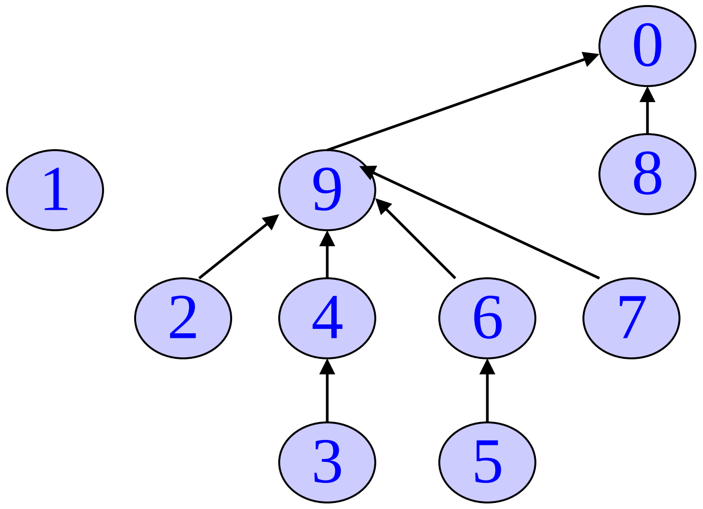


Example: Quick–union(10)

7 3



4 8

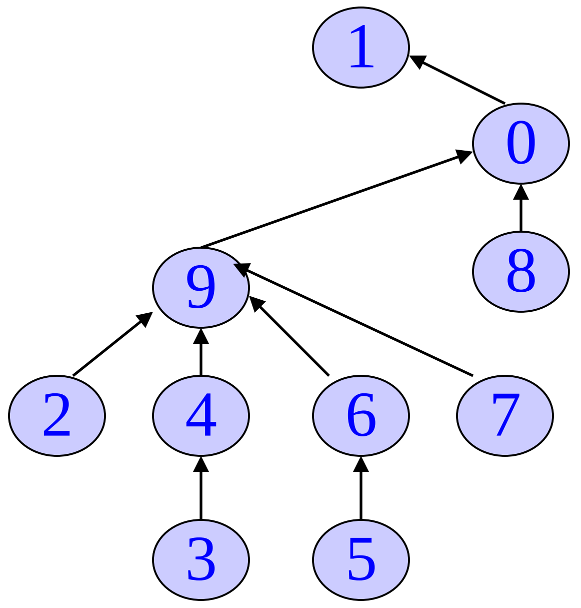


Example: Quick–union(11)

5 6 Do nothing

0 2 Do nothing

6 1



Example: Quick–union(12) - Analyzing Quick-union

The union operation does not have to go through N nodes, therefore it is faster than Quick-find.

Find operation is slower.

We can say that the algorithm is more efficient than the previous version in the average case (only after some analysis using the tools of the coming lectures).

Worst-case Behavior of Algorithms

Some input sequences will cause the algorithm to operate slower.

Such situations are called the worst-case.

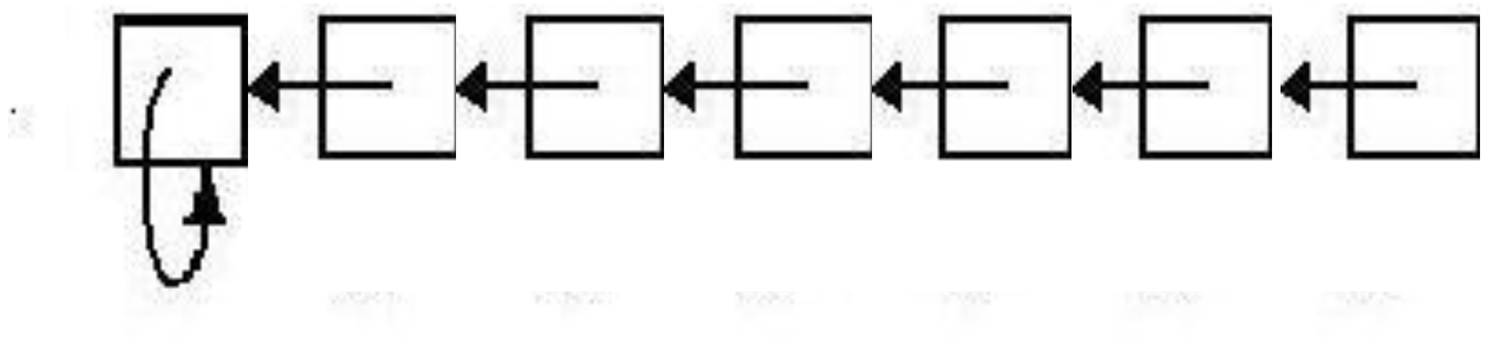
Worst-case analysis of algorithms specify the worst-case behavior of algorithms.

Algorithms are usually compared using the worst and the average case behavior.

Example: Quick–union(13) - Worst-case

Suppose input pairs arrive in the order:

0-1 , 1-2 , 2-3 , 3-4 , 4-5 ...



The find operation for object 0 has to follow N-1 pointers.

Example: Quick–union(14) - Worst-case

The average number of pointers followed for the first N pairs:

(0 + 1 + 2 .... + (N-1)) / N = (N-1) / 2

If N is large, similar situations may occur for many subsets of nodes.

Example: Quick–union(15) - Worst-case

Note that we (arbitrarily) connect the first node (p) to the second (q). This may cause long chains of nodes.

If we chose to connect the second node (q) to the first (p) the analysis result would not change (the sequence could be 1-0 , 2-1 , 3-2 , 4-3 , 5-4 ...)

We need to **flatten** the trees so that the union operation could always be completed by going through a smaller number of pointers.

Example:Weighted quick union

Use a second array to keep track of the number of nodes in each tree.

Initial:	0	1	2	3	.	.	.	.	N-2	N-1
	0	1	2	3					N-2	N-1

Size	1	1	1	1	.	.	.	.	1	1
	0	1	2	3					N-2	N-1

Example: Weighted quick union(2)

If pair p-q needs to be added:

- If size(p) < size(q) then make node p a subtree of q
- Otherwise make node q a subtree of p

(then update node counts accordingly)

0 1	0	0	2	3	.	.	.	.	N-2	N-1
	2	1	1	1	.	.	.	.	1	1

Example: Weighted quick union(3)

```
N = 10000,      ID[N]  SZ[N]  are arrays of int
FOR  (i = 0   TO  N-1)   { ID[i] = i   SZ[i] = 1 }
WHILE (a new pair p, q exists)
{
    FOR (i=p;  i != ID[i];   i = id[i])
    FOR (j=q;  j != ID[j];  j = id[j])
    IF  i  ==  j  THEN continue
    IF SZ[i] < SZ[j]
        THEN { ID[i] = j     SZ[j] += SZ[i] }
        ELSE  { ID[j] = i     SZ[i] += SZ[j] }
    output  p-q
}
```

Example: Weighted quick union(4)

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

Example: Weighted quick union(5)

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

Example: Weighted quick union(6)

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

Example: Weighted quick union(7)

4 8

8	1	3	3	3	3	5	3	3	3

1	1	1	9	1	2	1	1	2	1

5 6

8	1	3	3	3	3	5	3	3	3

1	1	1	9	1	2	1	1	2	1

0 2

8	1	3	3	3	3	5	3	3	3

1	1	1	9	1	2	1	1	2	1

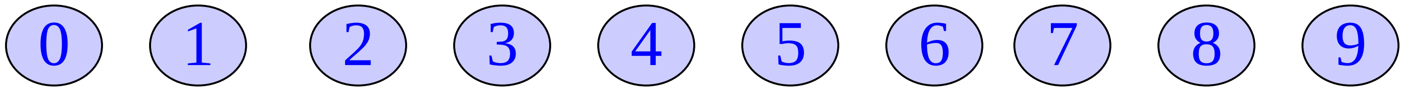
Example: Weighted quick union(8)

6 1

8	3	3	3	3	3	5	3	3	3

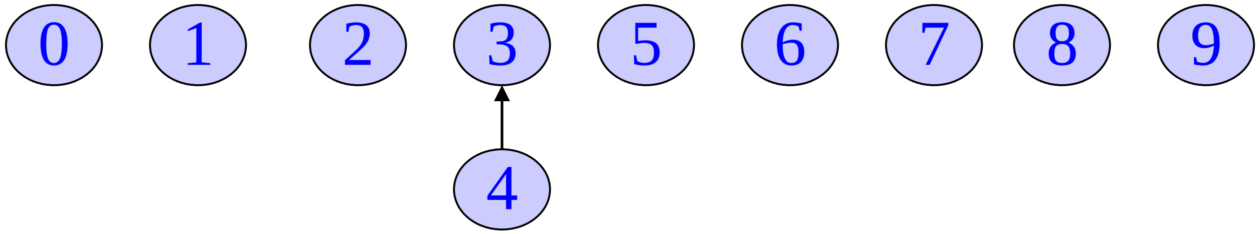
1	1	1	10	1	2	1	1	2	1

Example: Weighted quick union(9)

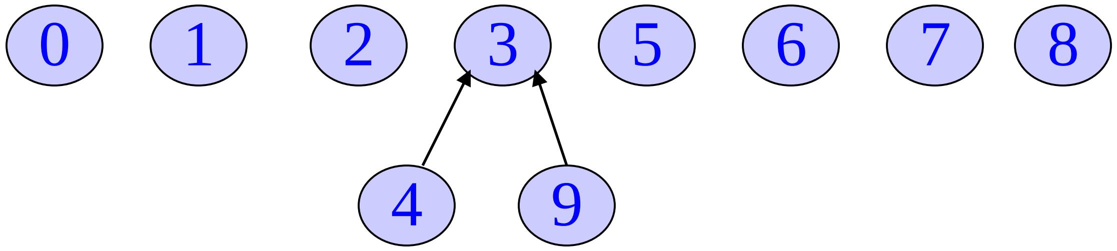


The initial state

3 4

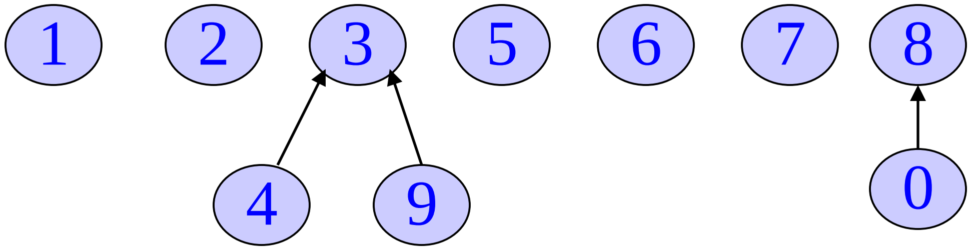


4 9

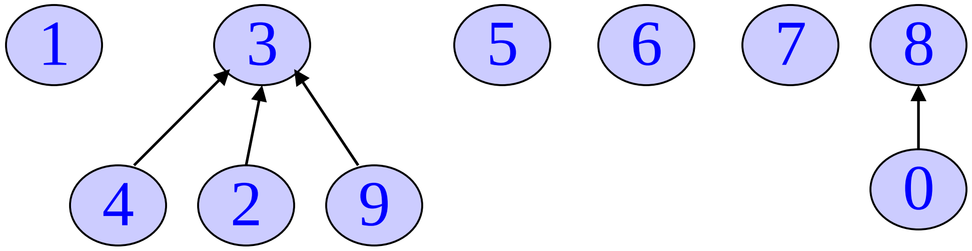


Example:Weighted quick union(10)

8 0

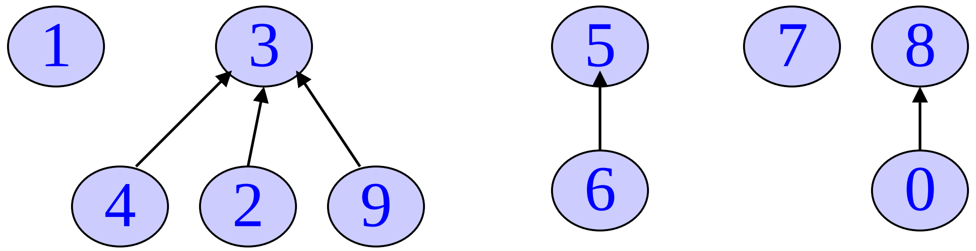


2 3



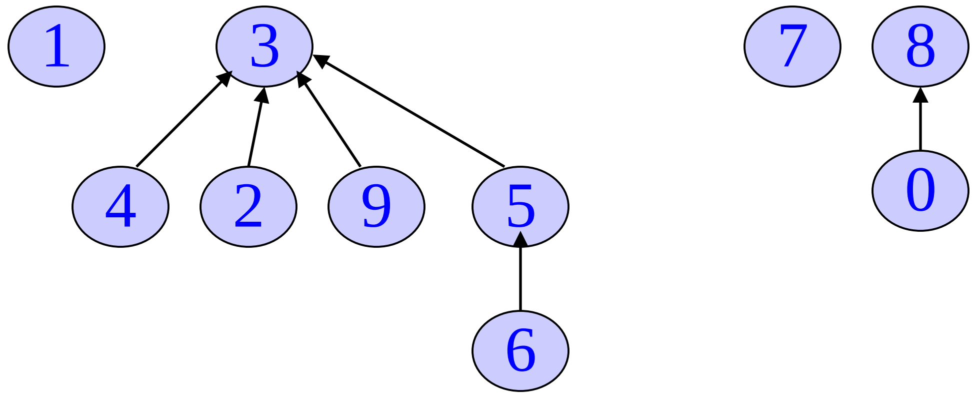
Example: Weighted quick union(11)

5 6



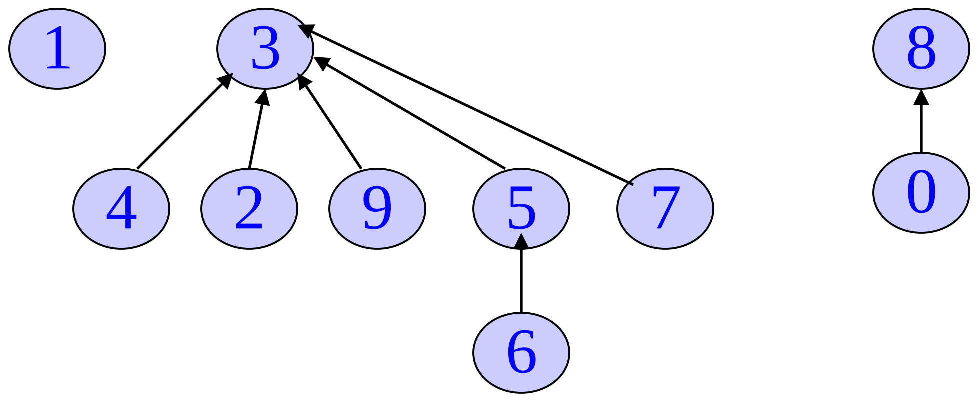
2 9 Do nothing

5 9

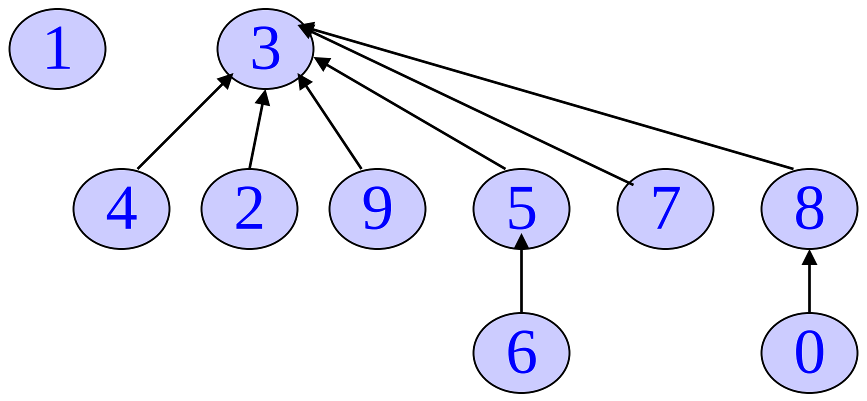


Example:Weighted quick union(12)

7 3



4 8

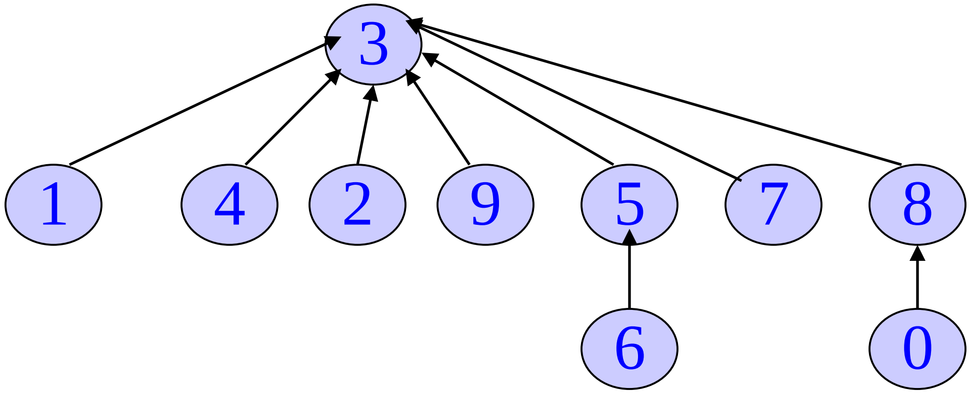


Example: Weighted quick union(13)

5 6 Do Nothing

0 2 Do Nothing

6 1



Example: Weighted quick union(14)

Maximum number of pointers that must be traversed when the tree has $\mathbf{2^n}$ nodes is **n.**

When two trees of $\mathbf{2^n}$ nodes are merged, we get a tree of $\mathbf{2^{n+1}}$ nodes, max of number of pointers that must be traversed become **n+1**.

The weighed quick-union algorithm follows at most $\mathbf{2\log_2 N} = \mathbf{2\ lg\ N}$ pointers to determine whether two of N objects are connected (much better than (N-1))