Union-find algorithms

by Alexandra Stefan

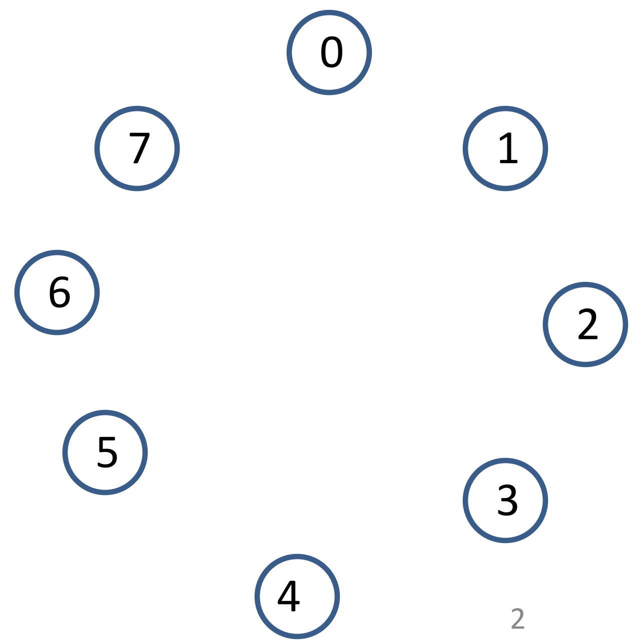
1/31/2016

Problem description

- Given N numbers/objects/nodes.
- First, assume each number is in a separate set (math interpretation of a set).
- A pair of numbers, p-q, indicates that those numbers are in the same set.

- When given a pair, p-q, we want to:
  - First, see if they are in the same set (***find***)
  - If they are, there is nothing to do
  - If they are not, (they belonged to 2 different sets) do the ***union*** of these sets (update our internal data so that now they show as one set)

- See main() in the next slide



Main()

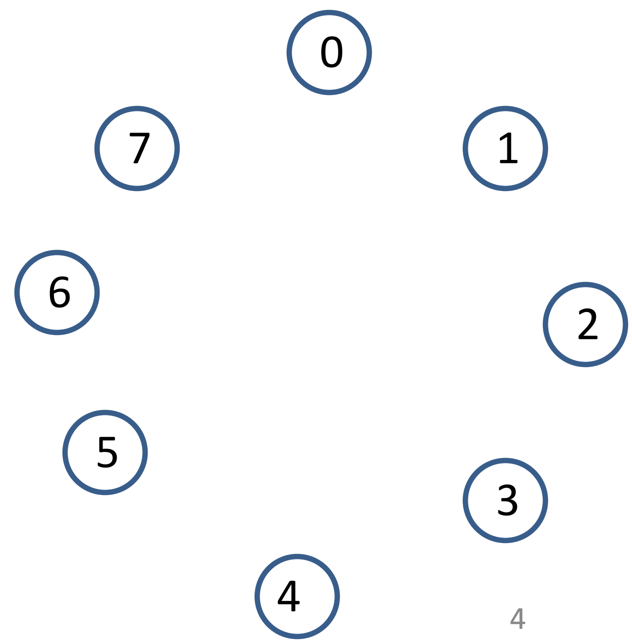
```
// We will see DIFFERENT implementations for find and set_union.

main()
{ int p, q, i, id[N], p_id, q_id;
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d", &p, &q) == 2) // while valid pairs are given
  {
    p_id = find(p, id);
    q_id = find(q, id);
    if (p_id == q_id)
    {
      printf(" %d and %d were on the same set\n", p, q);
      continue;
    }

    set_union(p_id, q_id, id, N);    // perform the union
    printf(" %d %d link led to set union\n", p, q);
} }
```

Quantities

- N - objects/numbers/nodes.
- P - pairs given to the algorithm.
- U - union operations.
  - not every pair results in a union operation

Ex.:
3-4
7-1
0-3
6-2
4-0
3-7

- Source of variance: U.
  - In the best case, U = ??? .
  - In the worst case, U = ??? .
    - What is the maximum value of U when P >= N?
      - That is, how many unions can you have at most, when you are given more pairs than objects?
      - How many unions can be done until all numbers are in the same set?



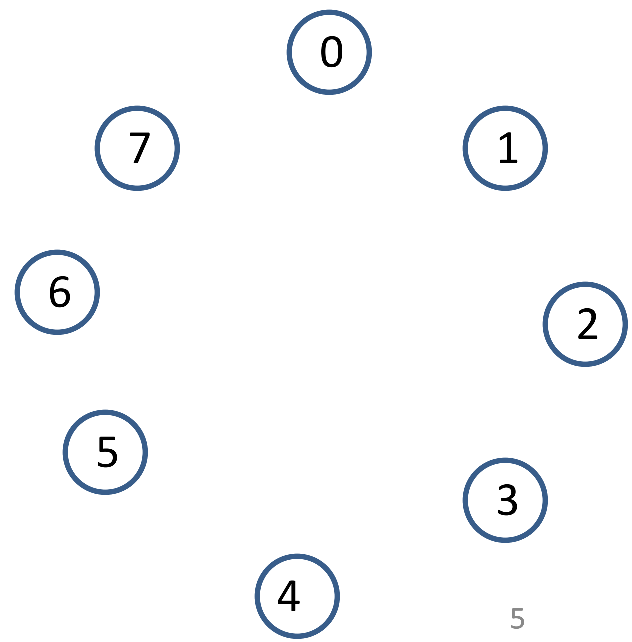
Quantities

– N - objects/numbers/nodes.
– P - pairs given to the algorithm.
– U - union operations.
  - not every pair results in a union operation

Ex.:
3-4
7-1
0-3
6-2
4-0
3-7

- Source of variance: U.
  – In the best case, U = 1 .
  – In the worst case, U = min(P, N-1) .
    - What is the maximum value of U when P >= N?
      – That is, how many unions can you have at most, when you are given more pairs than objects?

      – How many unions can be done until all numbers are in the same set?



How many find(...) and set_union(...)?

```
main()
{ int p, q, i, id[N], p_id, q_id;
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d", &p, &q) == 2) // while valid pairs are given
  {
    p_id = find(p, id);
    q_id = find(q, id);
    if (p_id == q_id)
    {
      printf(" %d and %d were on the same set\n", p, q);
      continue;
    }

    set_union(p_id, q_id, id, N);   // perform the union
    printf(" %d %d link led to set union\n", p, q);
} }
```

Union-find algorithms

- Quick find
- Quick union

- Weighted quick union

- Weighted quick union with full path compression
- Weighted quick union with path compression by halving

Union-Find: *quick find* solution

- Use: **id** array - keeps the set **id** for every number (**id[x]** is the label of the set that x belongs to):
  - Given pair: **p - q**,
  - Are they are already in the same set (***find***)?
    - Check: **id[p] == id[q]**
  - How do we do the ***union***:
    - We go through each number **i**, and if **id[i] == id[p]** (i is in the same set as p), we set **id[i] = id[q]**.

Quick-find: *find(...), set_union(...)*

```
#include <stdio.h>
#define N 10  /* Made N smaller, so we can print all ids */

/* returns the set id of the object. */
int find(int object, int id[])
{
  return id[object];
}

/* unites the two sets specified by set_id1 and set_id2*/
void set_union(int set_id1, int set_id2, int id[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    if (id[i] == set_id1) id[i] = set_id2;
}
```

Quick find

- We will build the trees produced by the quick find algorithm for the following pairs.
- Remember that for this method, the left tree points to the right tree.

3 – 5

4 – 0

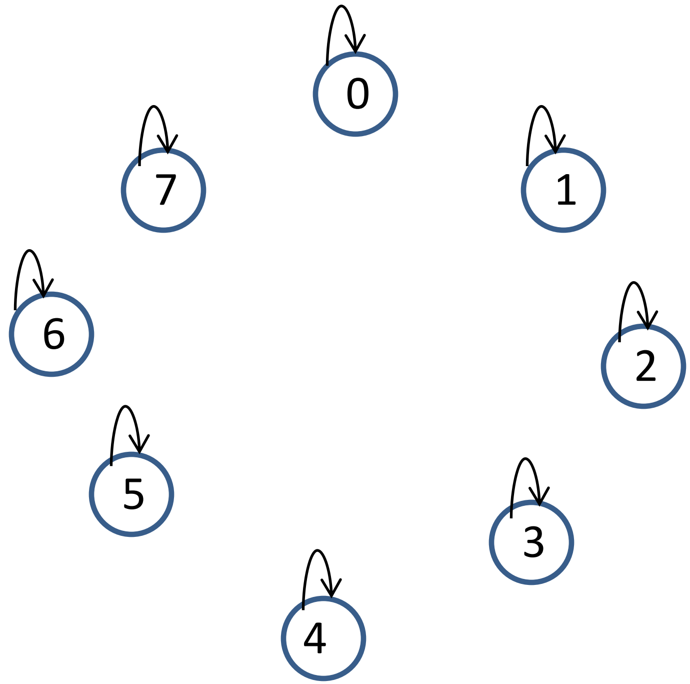
2 – 1

4 – 1

3 – 2

7 – 0

0 – 6



pos	0	1	2	3	4	5	6	7
id	0	1	2	3	4	5	6	7

Quick find

3 – 5

4 – 0

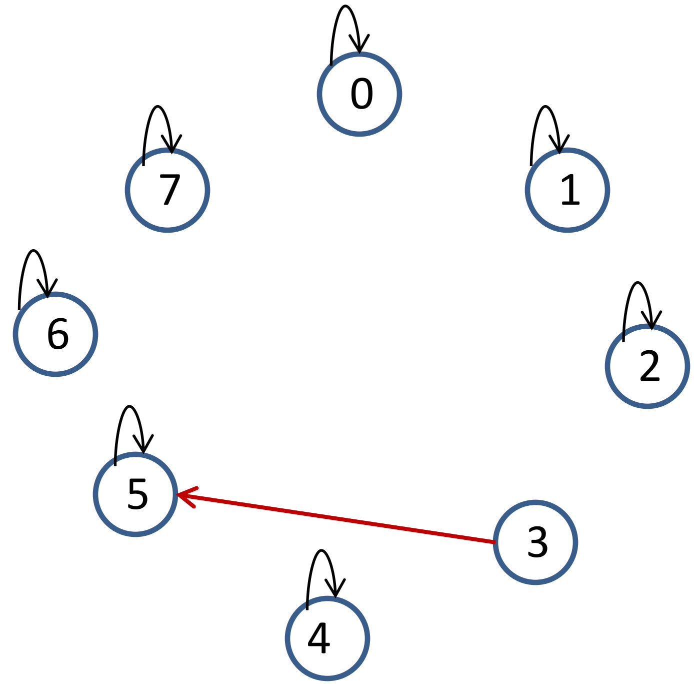
2 – 1

4 – 1

3 – 2

7 – 0

0 – 6



pos	0	1	2	3	4	5	6	7
id	0	1	2	**5**	4	5	6	7

Quick find

3 – 5

4 – 0

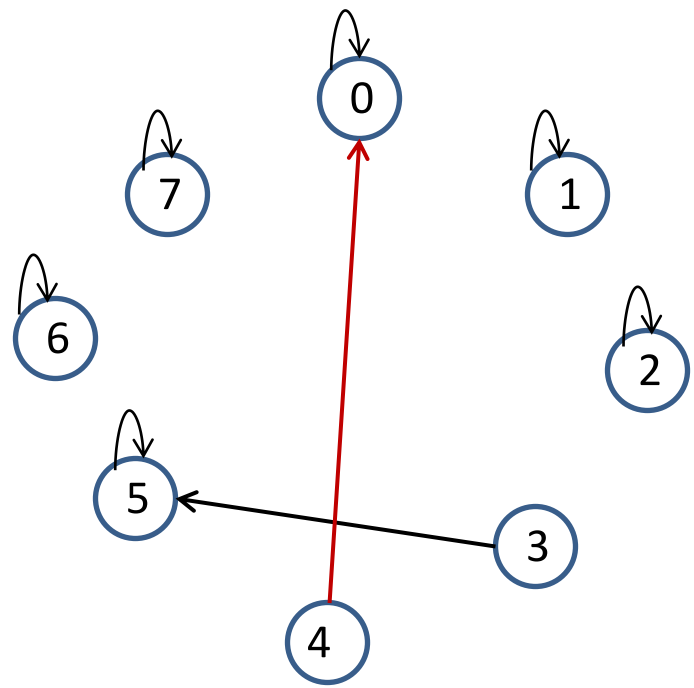
2 – 1

4 – 1

3 – 2

7 – 0

0 – 6



pos	0	1	2	3	4	5	6	7
id	0	1	2	5	**0**	5	6	7

Quick find

3 – 5

4 – 0

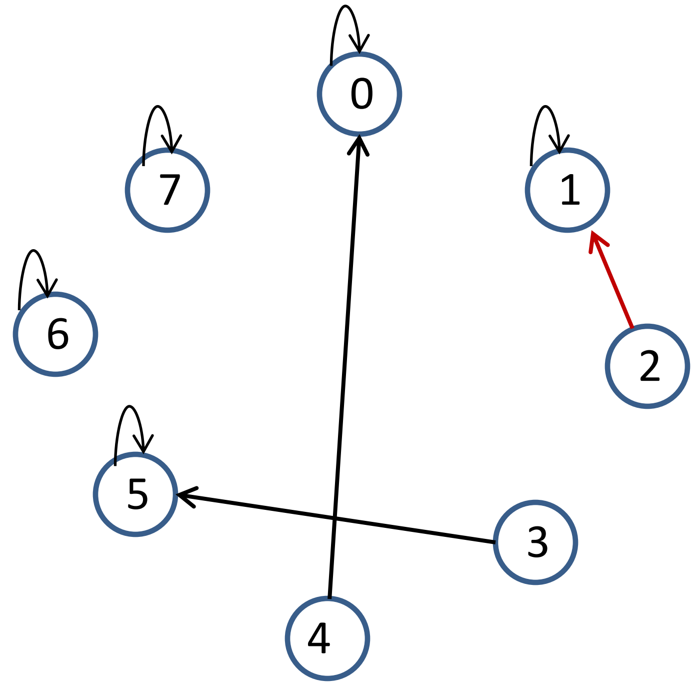
2 – 1

4 – 1

3 – 2

7 – 0

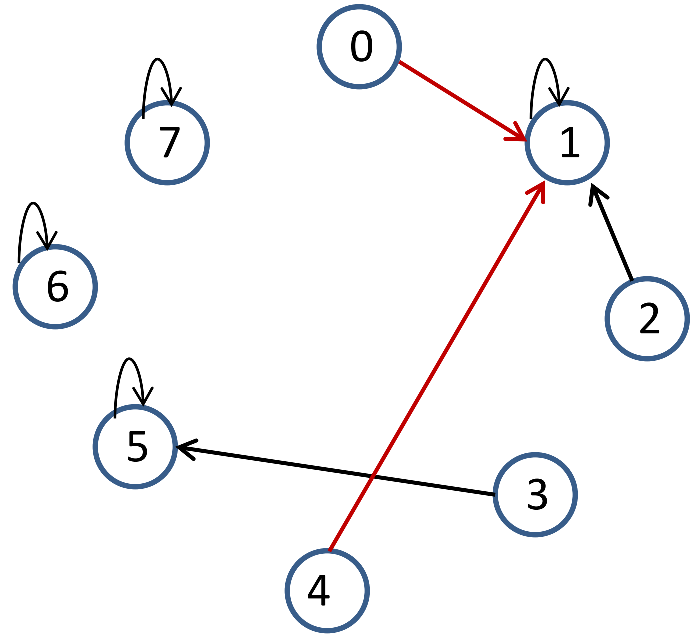
0 – 6



pos	0	1	2	3	4	5	6	7
id	0	1	**1**	5	0	5	6	7

Quick find

```
#include <stdio.h>
#define N

/* returns the set id of the object. */
int find(int object, int id[])
{
  return id[object];
}

/* unites the two sets specified by
set_id1 and set_id2*/
void set_union(int set_id1, int set_id2,
                int id[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    if (id[i] == set_id1)
        id[i] = set_id2;
}
```



3 – 5

4 – 0

2 – 1

4 – 1

3 – 2

7 – 0

0 – 6

pos	0	1	2	3	4	5	6	7
id	**1**	1	1	5	**1**	5	6	7

Note that since we do not know what other nodes are in the same set as 4, we have to look at all of the nodes, thus the loop in set_union.

Runtime: find? set_union?

Quick find - *time analysis*

```
main()
{ int p, q, i, id[N], p_id, q_id;
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d", &p, &q) == 2) // repeats P times
  {
    p_id = find(p, id);
    q_id = find(q, id); v // total:  P times * constant cost ->  O(P)
    if (p_id == q_id)
    {
      printf(" %d and %d were on the same set\n", p, q);
      continue;
    }

    set_union(p_id, q_id, id, N);//total:(U times * cost N)<= N*N-> O(N^2)
    printf(" %d %d link led to set union\n", p, q);
} }
```

Quick find

3 – 5

4 – 0

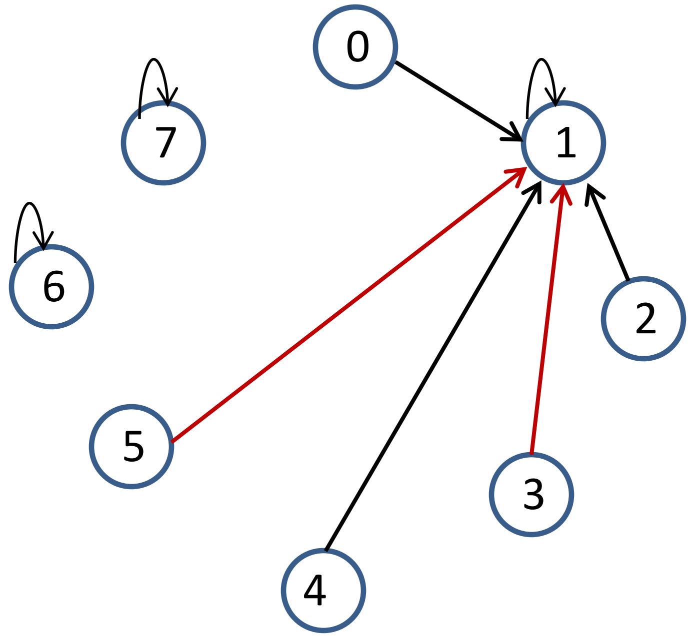
2 – 1

4 – 1

3 – 2

7 – 0

0 – 6



pos	0	1	2	3	4	5	6	7
id	1	1	1	**1**	1	**1**	6	7

Quick find

3 – 5

4 – 0

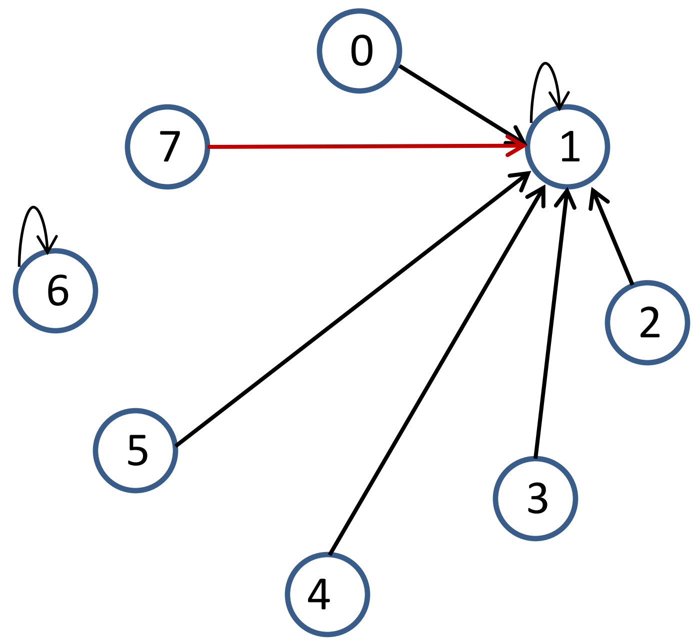
2 – 1

4 – 1

3 – 2

7 – 0

0 – 6



pos	0	1	2	3	4	5	6	7
id	1	1	1	1	1	1	6	**1**

Quick find

3 – 5

4 – 0

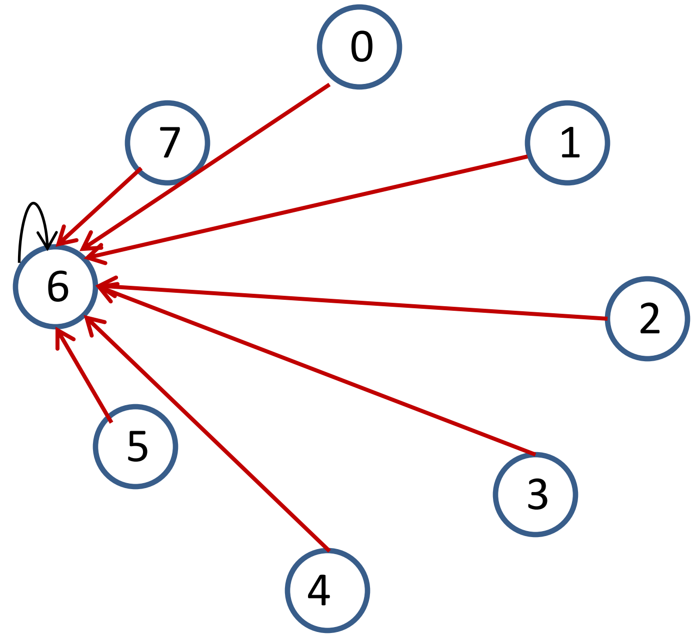
2 – 1

4 – 1

3 – 2

7 – 0

0 – 6



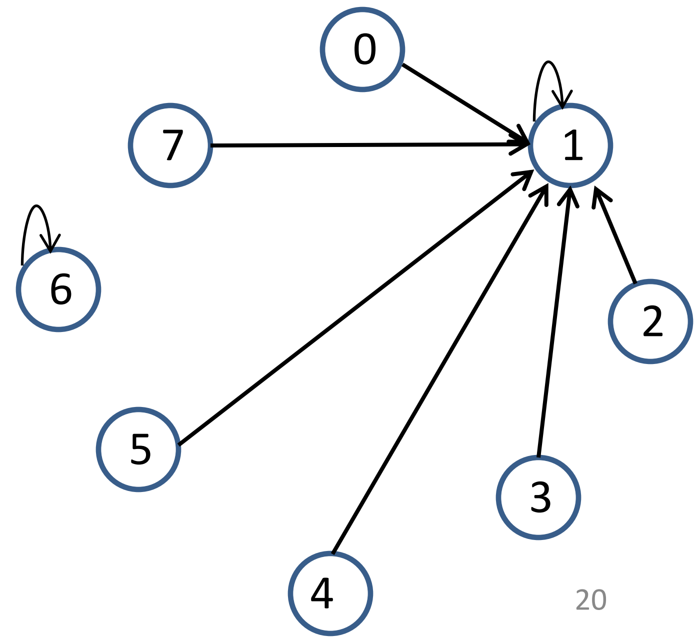
pos	0	1	2	3	4	5	6	7
id	**6**	**6**	**6**	**6**	**6**	**6**	6	**6**

Quick union

Quick Union

- Problem in quick find:
  - moving all the arrows
- Can we improve that?
  - Can we move only one arrow?
  - How will we find the set id/representative?
    - E.g. after this union, how will we know that 7 and 6 are in the same set?

Next
0 - 6



pos	0	1	2	3	4	5	6	7
id	1	1	1	1	1	1	6	1

Quick Union

- **id[p]** will NOT point directly to the set_id of p.
  - It will point to just another element of the same set.
  - Thus, **id[p]** initiates a sequence of elements:
    - **id[p] = p_2, id[p_2] = p_3, …, id[p_n] = p_n**
  - This sequence ends when we find an element $\mathbf{p_n}$ s. t.: **id[p_n] = p_n**.
  - Set id: $\mathbf{p_n}$ such that **id[p_n] = p_n**.
  - This sequence is not allowed to contain cycles.
  - As before, the representative of the left node will point to that of the right node
- We re-implement **find** and **union** to follow these rules.

Quick union

- We will build the trees produced by the **quick union** algorithm for the following pairs.
- The representative of the left node points to the representative of the right node.

3 – 5

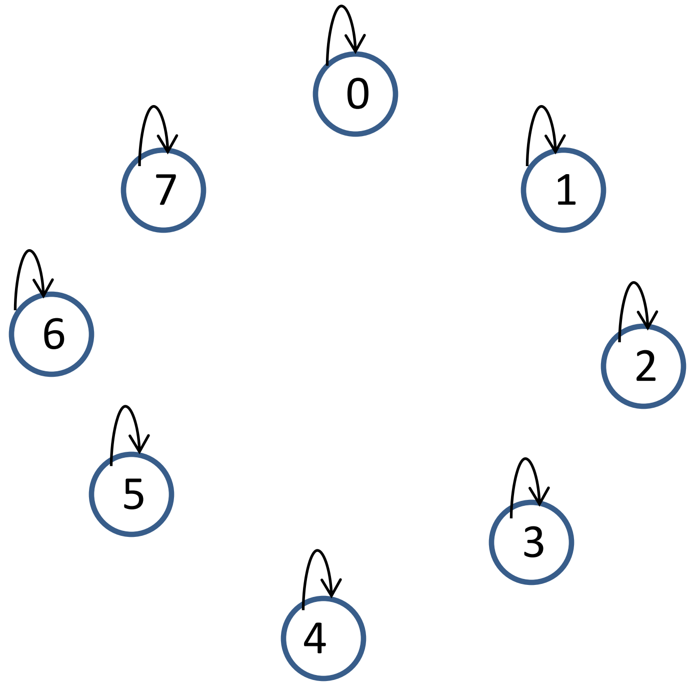
4 – 0

2 – 1

4 – 1

3 – 2

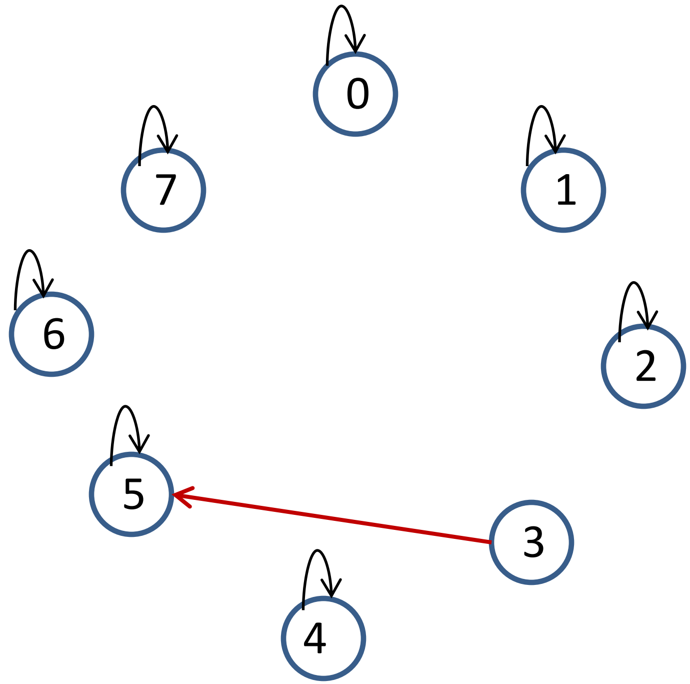
7 – 0



pos	0	1	2	3	4	5	6	7
id	0	1	2	3	4	5	6	7

Quick union

3 – 5
4 – 0
2 – 1
4 – 1
3 – 2
7 – 0



pos	0	1	2	3	4	5	6	7
id	0	1	2	**5**	4	5	6	7

Quick union

3 – 5

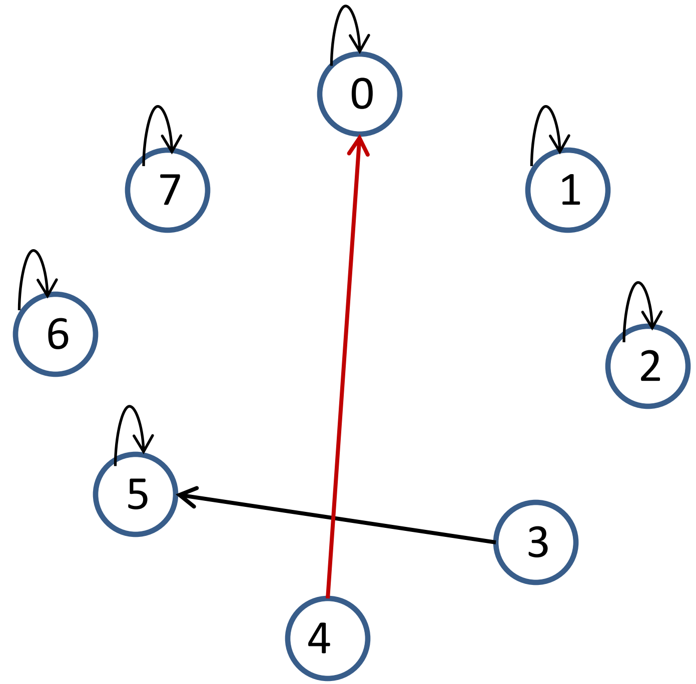
4 – 0

2 – 1

4 – 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	0	1	2	5	**0**	5	6	7

Quick union

3 – 5

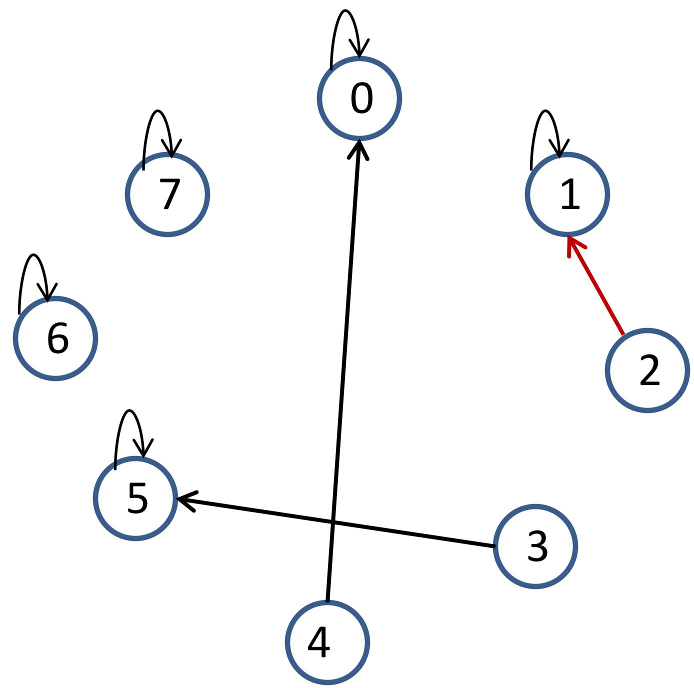
4 – 0

2 – 1

4 – 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	0	1	**1**	5	0	5	6	7

Quick union

3 – 5

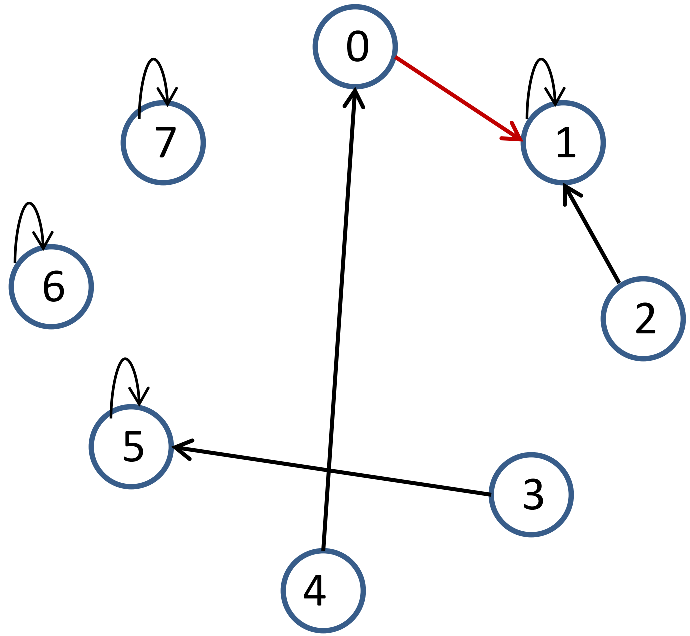
4 – 0

2 – 1

4 – 1: rep(4) is 0, rep(1) is 1, 0 will point to 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	**1**	1	1	5	0	5	6	7

Quick union

3 – 5

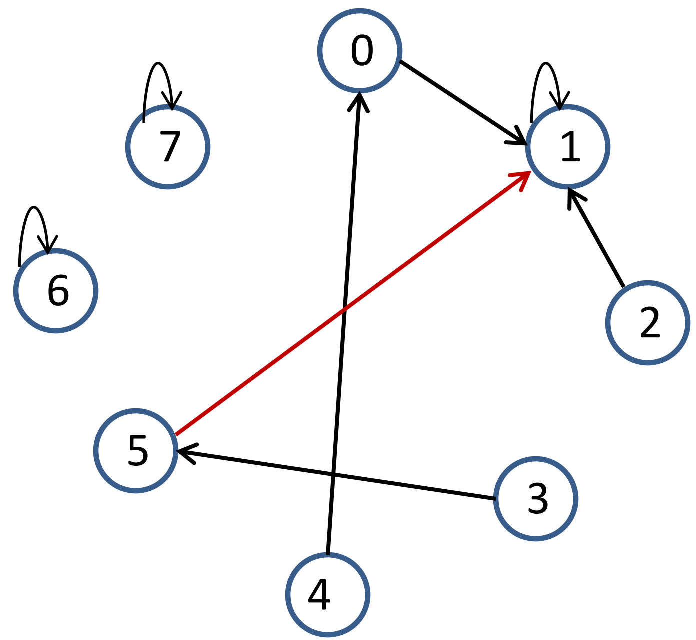
4 – 0

2 – 1

4 – 1

3 – 2:rep(3) is 5, rep(2) is 1, 5->1

7 – 0



pos	0	1	2	3	4	5	6	7
id	1	1	1	5	0	**1**	6	7

Quick union

3 – 5

4 – 0

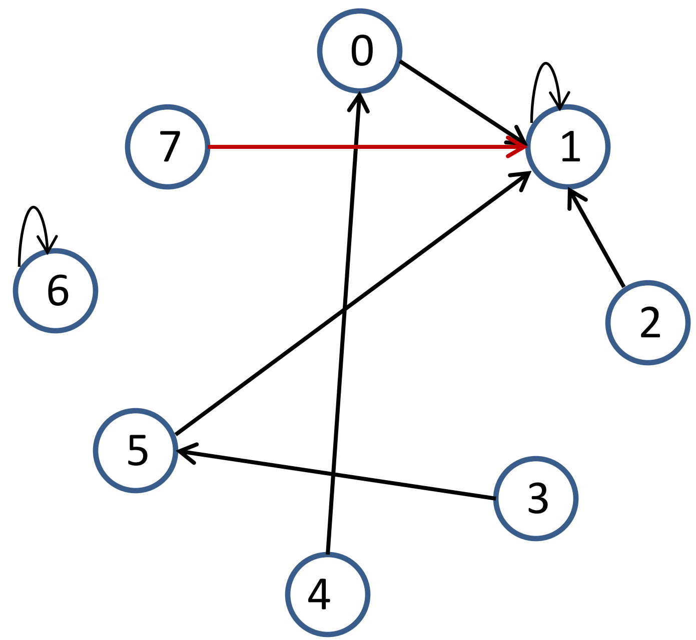
2 – 1

4 – 1

3 –2

7 – 4:rep(4) is 1, rep(7) is 7, 7->4

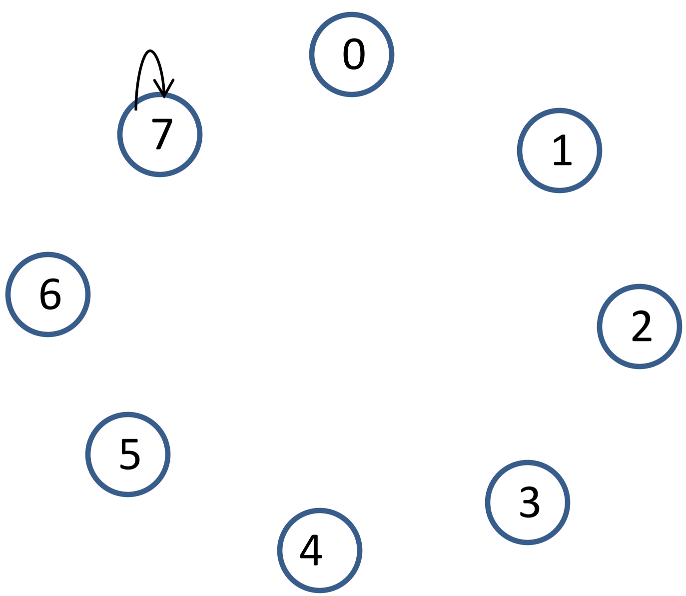
Note that the representative of 4 is 1, not 0.



pos	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
id	1	1	1	5	0	1	6	**1**

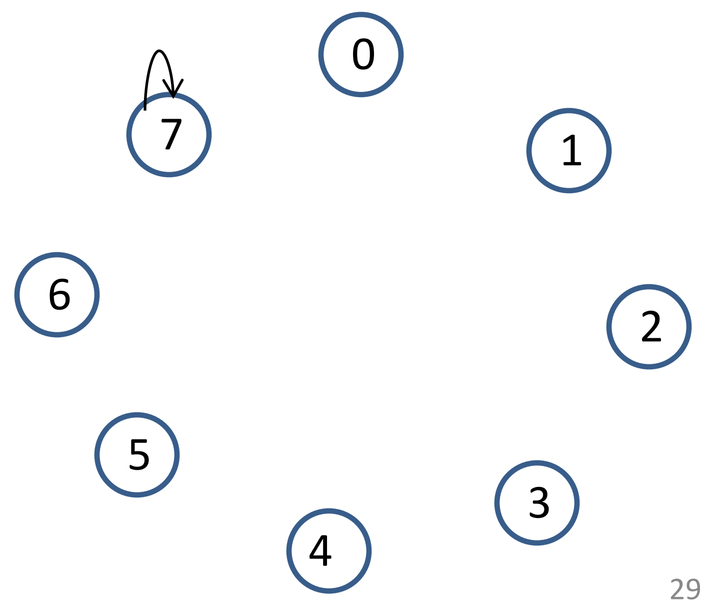
Quick union:
best, worst, average cases

- Best tree?
  - Picture
  - Sequence of pairs
  - Find cost:
  - Union cost:



- Average?

- Worst tree?
  - Picture
  - Sequence of pairs
  - Find cost
  - Union cost



Quick union – worst case analysis

- Worst case: long chain

- **Quick union** can produce chains in 2 extreme ways and the total cost of producing the chain is different in the 2 cases:
  - N - (N-1), (N-1)-(N-2), (N-2)-(N-3), …., 3 - 2, 2 - 1
    - cost proportional to N
  - N - (N-1), N-(N-2), N-(N-3), …., N - 2, N – 1
    - cost proportional to N^2

Once the chain is built, processing a pair that includes node N takes N-1 links (to find the representative of N). This leads to a cost proportional to M*N in the worst case (where M is the number of pairs and N is the number of nodes).

Case: N-(N-1), (N-1)-(N-2), …., 3- 2, 2-1

The sequence of pairs below generates a chain pointing from N to 1:

N- (N-1)

(N-1) - (N-2)

(N-2) - (N-3)

….

3 – 2

2 – 1

Note: here we are not using node 0.

Example for N = 6:

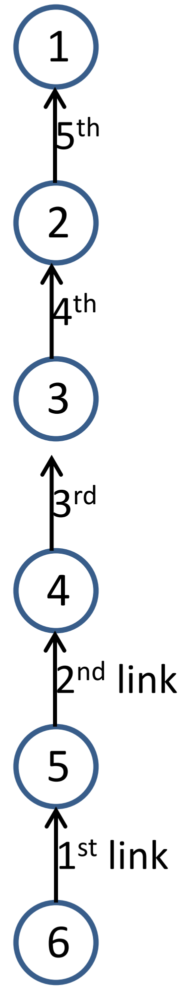
6 - 5

5 - 4

4 - 3

3 – 2

2 - 1



Note that for every pair we have representatives of current trees and such the find operation is really quick (just one check and we see we are at the representative). **We do not travel across any links/edges.**

Case: N-(N-1), N-(N-2), …., N - 2, N -1

The sequence of pairs below generates a chain pointing from N to 1:

N- (N-1)

N - (N-2)

N - (N-3)

….

N – 2

N – 1

Note: here we are not using node 0.

Example for N = 6:

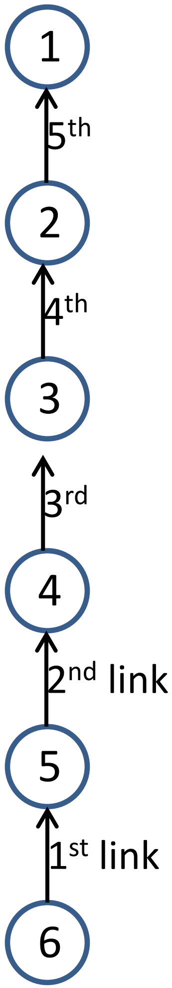
6 - 5

6 - 4

6 - 3

6 – 2

6 - 1



This sequence produces the same chain, but searching for the representative of N gets more and more expensive:

1 link for pair N – (N-2)

2 links for pair N – (N-3)

…

N-2 links for pair N – 1

Total: 1+2+3+…(N-2) = (N-2) * (N-1)/2

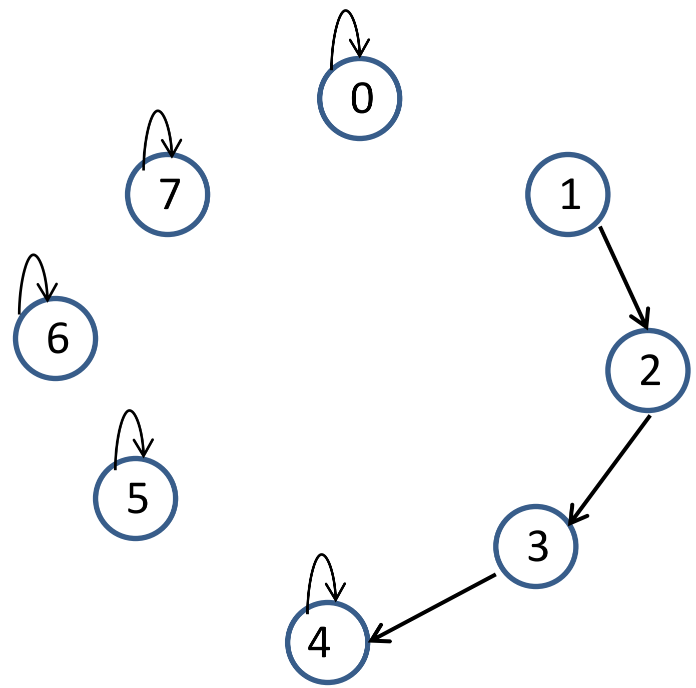
Quick union Version

```
int find(int object, int id[])
{ int next_object;
  next_object = id[object];

  while (next_object != id[next_object])
    next_object = id[next_object];

  return next_object;
}

/* unites the two sets specified by set_id1 and set_id2 */
void set_union(int set_id1, int set_id2, int id[], int size)
{
  id[set_id1] = set_id2;
}
```

Weighted quick union

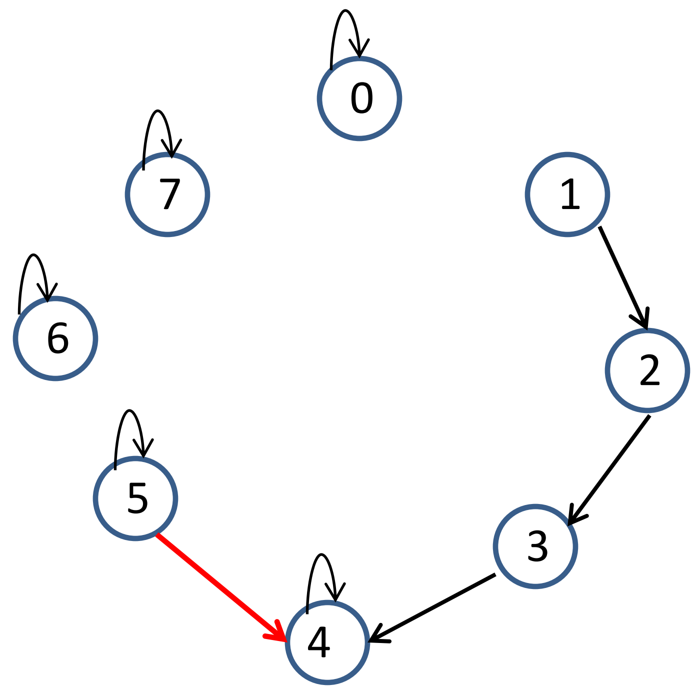
Weighted quick union

- **Choose the direction of the arrow**.
  - Tree of p -> tree of q or
  - Tree of p <- tree of q
- Assume you have the set on the right and you are given pair: **5 - 4**
  - 4 point to 5 ? or
  - 5 point to 4?
- Which arrow will give a better tree?
- What is a better tree?
- What is the best tree?



Weighted quick union

- **Choose the direction of the arrow**.
  - Tree of p -> tree of q or
  - Tree of p <- tree of q
- Assume you have the set on the right and you are given pair: **5 - 4**
  - 4 point to 5 ? or
  - 5 point to 4?
- Which of these will give you a better tree for the future?
  - How does a good tree look?
- What is the best tree?
- How will you decide which arrow direction makes a better tree?



Weighted union

- **find**: same as in second version.
- **union**: always change the id of the smaller set to that of the larger one.

```
void set_union(int set_id1, int set_id2, int id[], int sz[])
{ if (sz[set_id1] < sz[set_id2])
   {
     id[set_id1] = set_id2;
     sz[set_id2] += sz[set_id1];
   }
   else
   {
     id[set_id2] = set_id1;
     sz[set_id1] += sz[set_id2];
   }
}
```

Weighted quick union

- **Smaller tree → the larger tree**
- When the sizes are equal, the tree of the **q** will point to the tree if **p** (book implementation).

- We will build the trees produced by the weighted quick union algorithm for the following pairs.
- The size is written under the node. When we start, they all have size 1.

3 – 5

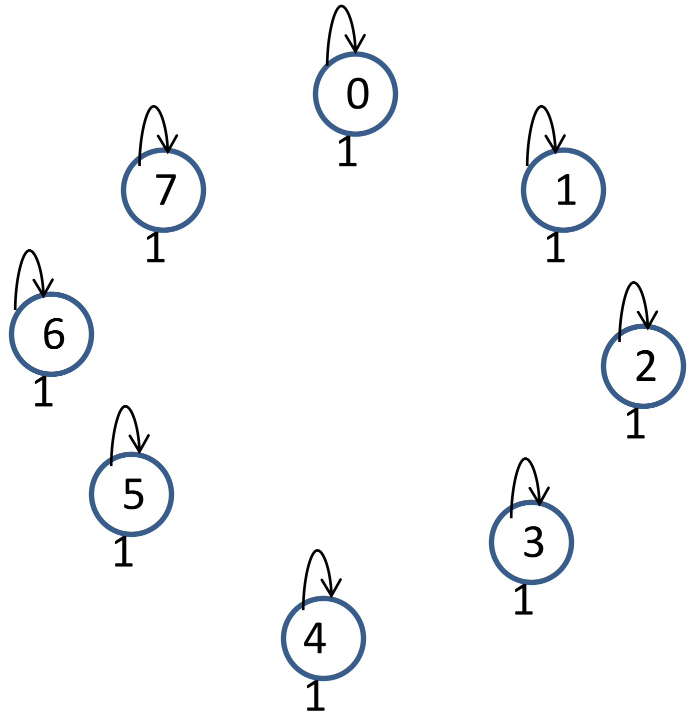
4 – 0

2 – 1

4 – 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	0	1	2	3	4	5	6	7
sz	1	1	1	1	1	1	1	1

Weighted quick union

- Remember that for this method, when the sizes are equal, the right tree points to the left tree.

3 – 5 (node 3 has size 2 now)

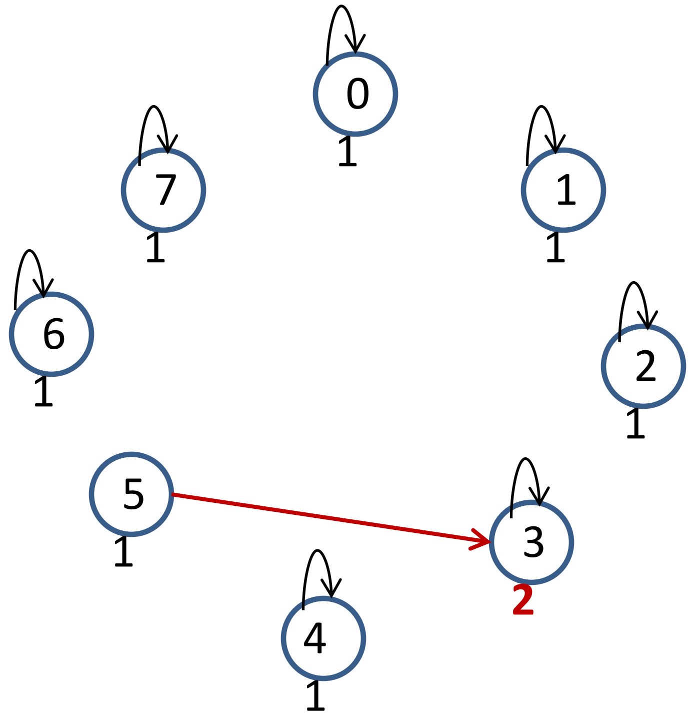
4 – 0

2 – 1

4 – 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	0	1	2	3	4	3	6	7
sz	1	1	1	2	1	1	1	1

Weighted quick union

- Remember that for this method, when the sizes are equal, the right tree points to the left tree.

3 – 5

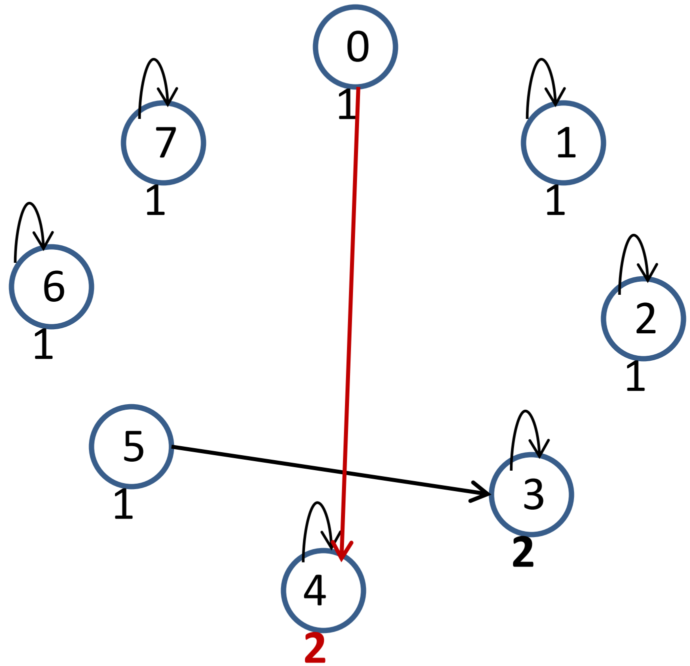
4 – 0

2 – 1

4 – 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	**4**	1	2	3	4	3	6	7
sz	1	1	1	2	**2**	1	1	1

Weighted quick union

- Remember that for this method, when the sizes are equal, the right tree points to the left tree.

3 – 5

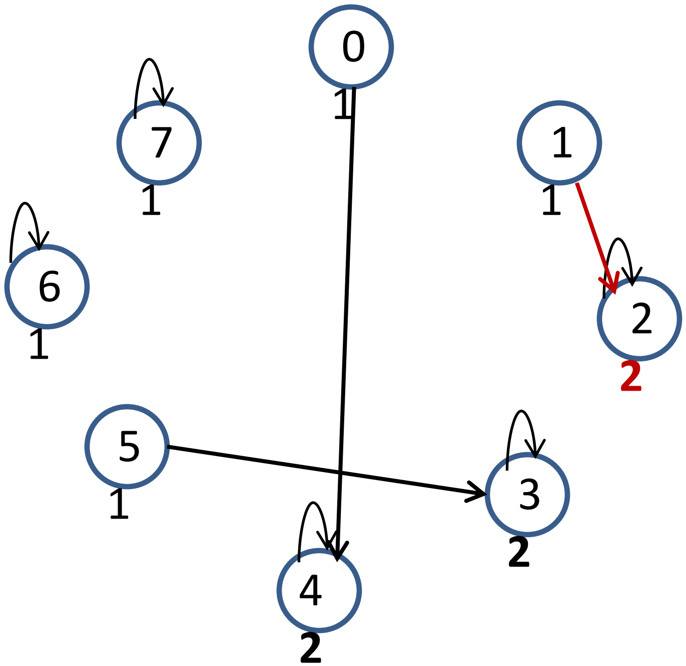
4 – 0

2 – 1

4 – 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	4	2	2	3	4	3	6	7
sz	1	1	2	2	2	1	1	1

Weighted quick union

- Remember that for this method, when the sizes are equal, the right tree points to the left tree.

3 – 5

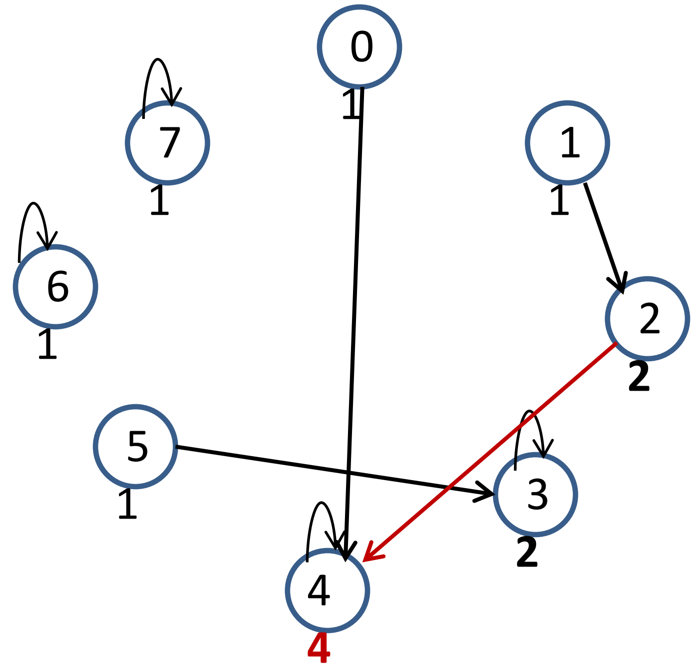
4 – 0

2 – 1

4 – 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	4	2	4	3	4	3	6	7
sz	1	1	2	2	2	1	1	1

Weighted quick union

- Remember that for this method, when the sizes are equal, the right tree points to the left tree.

3 – 5

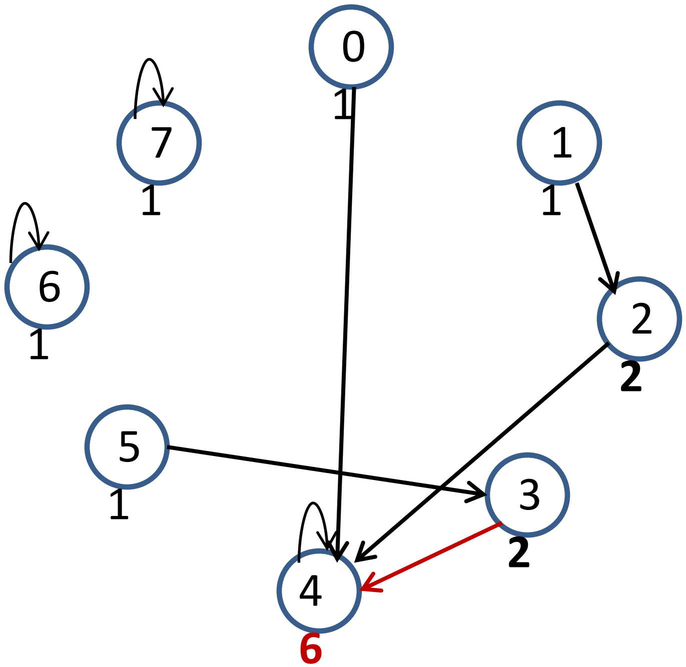
4 – 0

2 – 1

4 – 1

3 – 2: rep(3) is 3 (size 2), rep(2) is 4 (size 4)

7 – 0



pos	0	1	2	3	4	5	6	7
id	4	2	4	4	4	3	6	7
sz	1	1	2	2	6	1	1	1

Weighted quick union

- Remember that for this method, when the sizes are equal, the right tree points to the left tree.

3 – 5

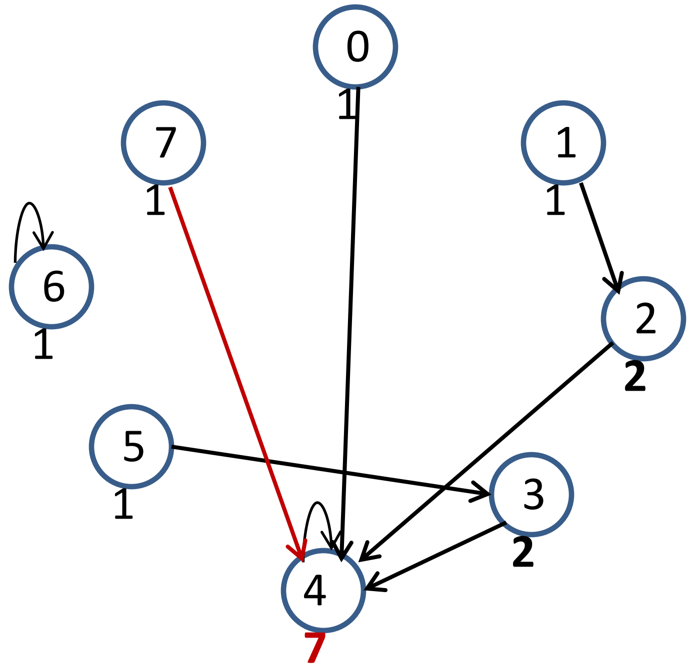
4 – 0

2 – 1

4 – 1

3 – 2

7 – 0



pos	0	1	2	3	4	5	6	7
id	4	2	4	4	4	3	6	**4**
sz	1	1	2	2	**7**	1	1	1

Weighted quick union: upper bound for the *find* operation

- **Weighted quick union** guarantees that the length of the **longest path** in any tree of size N (N nodes) is **smaller than or equal to lg(N).**
  - Property I.3, (page 16)
  - This gives an upper-bound on the cost of any ***find*** operation for this method: $\leq \lg N$

Weighted union

```
void set_union(int set_id1, int set_id2, int id[], int sz[])
{ if (sz[set_id1] < sz[set_id2])
  {
    id[set_id1] = set_id2;
    sz[set_id2] += sz[set_id1];
  }
  else
  {
    id[set_id2] = set_id1;
    sz[set_id1] += sz[set_id2];
  }
}
```

Weighted quick union - *time analysis* *find and set_union*

```
main()
{ int p, q, i, id[N], p_id, q_id;
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d", &p, &q) == 2) // repeats P times
  {
    p_id = find(p, id);
    q_id = find(q, id); v // total:  P times * cost <= P*lgN ->  O(P*lgN)
    if (p_id == q_id)
    {
      printf(" %d and %d were on the same set\n", p, q);
      continue;
    }

    set_union(p_id, q_id, id, N);//total:(U times * const cost) -> O(N)
    printf(" %d %d link led to set union\n", p, q);
} }
```

Union-Find overview - time analysis

```
main()
{ int p, q, i, id[N], p_id, q_id;

  for (i = 0; i < N; i++) id[i] = i;  // N times

  while (scanf("%d %d", &p, &q) == 2) // P times
  {
    p_id = find(p, id); // P times * cost of find
    q_id = find(q, id); // P times * cost of find
    if (p_id == q_id)
    {
      printf("%d and %d, in the same set\n",p,q);
      continue;
    }

    set_union(p_id, q_id, id, N); // U times * cost of union
    printf(" %d %d link led to set union\n", p, q);
  }
}
```

Union-Find overview - time analysis

- N - objects/numbers/nodes.
- P - pairs given to the algorithm.
- U - union operations:
  $1 \leq U \leq N-1$.

- General formula (excluding constants):
  N + P * cost(find) + U * cost(set_union)
  - The table shows the ~cost of the while loop (it excludes the initializing for loop):
    P * cost(find) + U * cost(set_union)

- In the table below we assume P > N and do not show the constants
  (The actual cost is proportional to the one shown in the table.)

Method	Cost of find	Cost of set_union	Total Best (Min)	Total Worst (Max)	Total Average	Trees (levels)
Quick find	1	N	$N*U \leq N^2$	$N*U \leq N^2$	$N*U \leq N^2$	1
Quick union	$1 \leq$ cost < N	1	P	P*N	P * lgN	≤ N-1
Weighted union	$1 \leq$ cost $\leq$ lgN	1	P	P * lgN	P * lgN	≤ lg N

Weighted quick union with full path compression

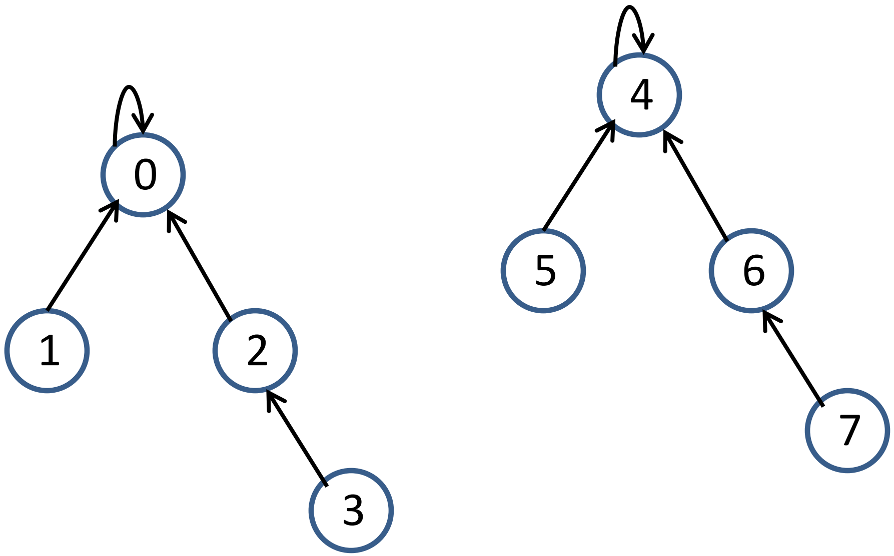
Weighted quick union with full path compression

- Full path compression is an improvement that can be added to the weighted quick union algorithm.
- After given a pair of nodes p-q, once the ***new root, r***, is determined, ***all the nodes on the path*** from p to r and from q to r will be changed to map/link directly to r.
- This update will be done in the Union operation.
  - Implicitly, if the nodes *p* and *q* are already in the same connected component, there will be no union performed and so no node will be updated.

Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6



Next we will see what happens in each of the following 3 possible cases:

a) 7 – 3
b) 5 – 3
c) 7 - 5

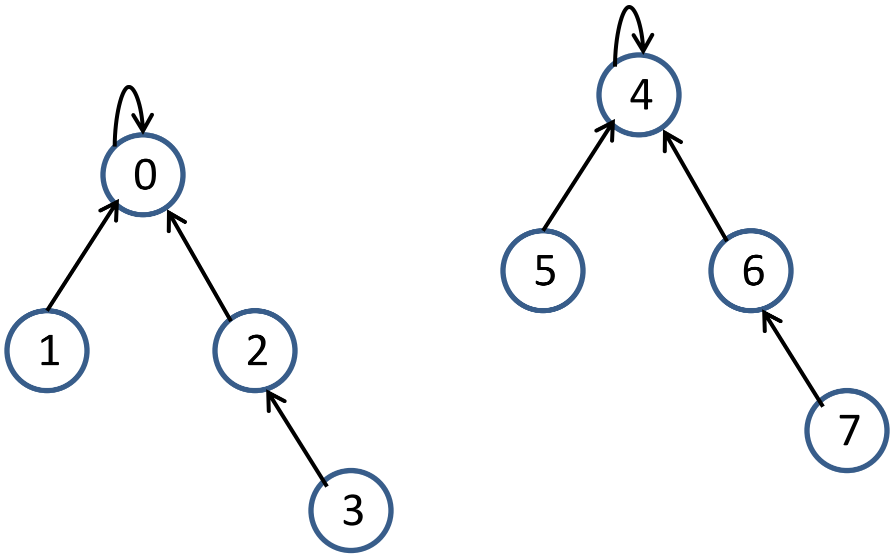
Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6

Case:

a) 7 – 3

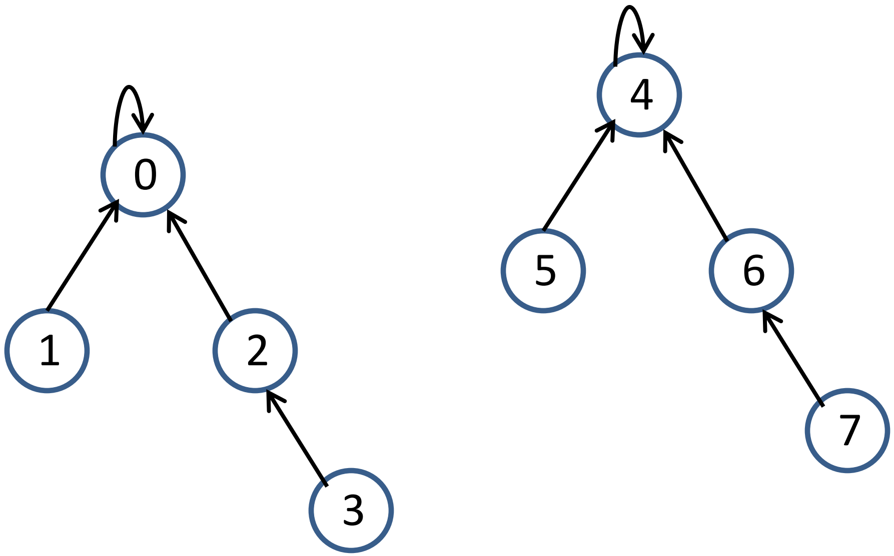


pos	0	1	2	3	4	5	6	7
id	0	0	0	2	4	4	4	6
sz	4	1	2	1	4	1	2	1

Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6



pos	0	1	2	3	4	5	6	7
id	0	0	0	2	4	4	4	6
sz	4	1	2	1	4	1	2	1

Case:

a) 7 – 3

Representative of 7:

Representative of 3:

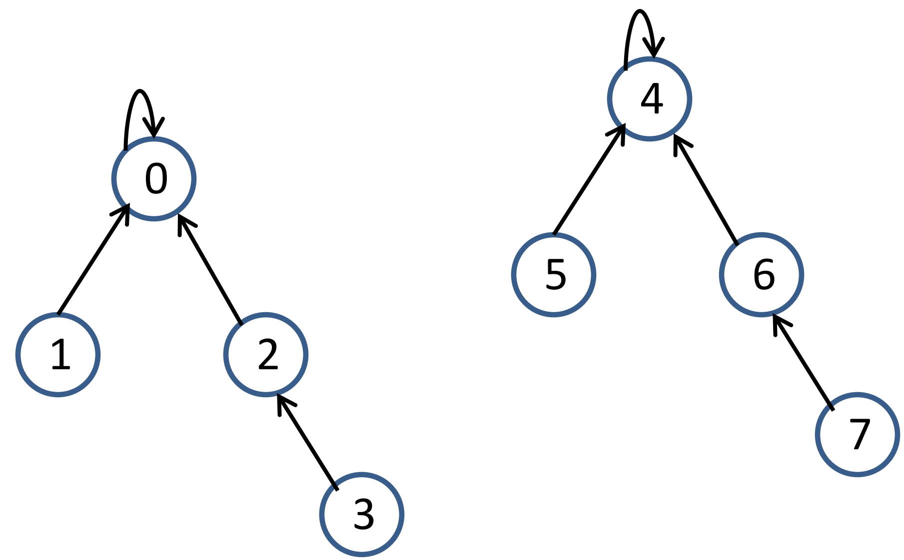
Compare sizes:

New root:

Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6



pos	0	1	2	3	4	5	6	7
id	0	0	0	2	4	4	4	6
sz	4	1	2	1	4	1	2	1

Case:

a) 7 – 3

Representative of 7: 4

Representative of 3: 0

Compare sizes: same, (right points to left)

New root: 4

Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6

Case:

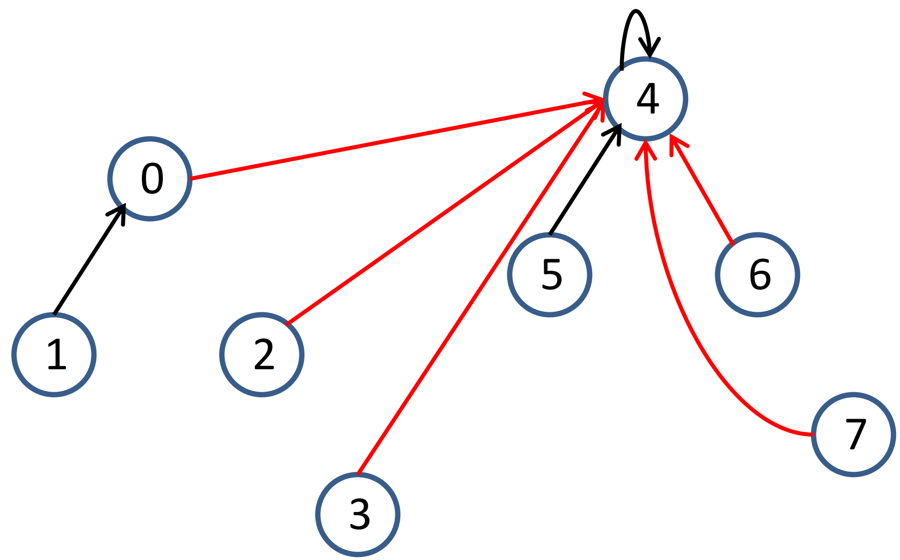
a) 7 – 3

Representative of 7: 4

Representative of 3: 0

Compare sizes: same, (right points to left)

New root: 4

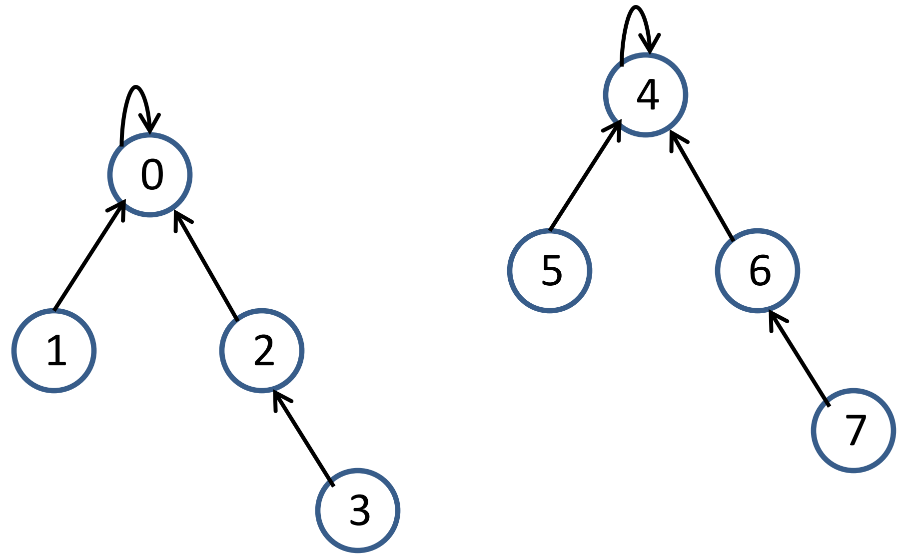


pos	0	1	2	3	4	5	6	7
id	**4**	0	**4**	**4**	4	4	**4**	**4**
sz	4	1	2	1	**8**	1	2	1

Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6



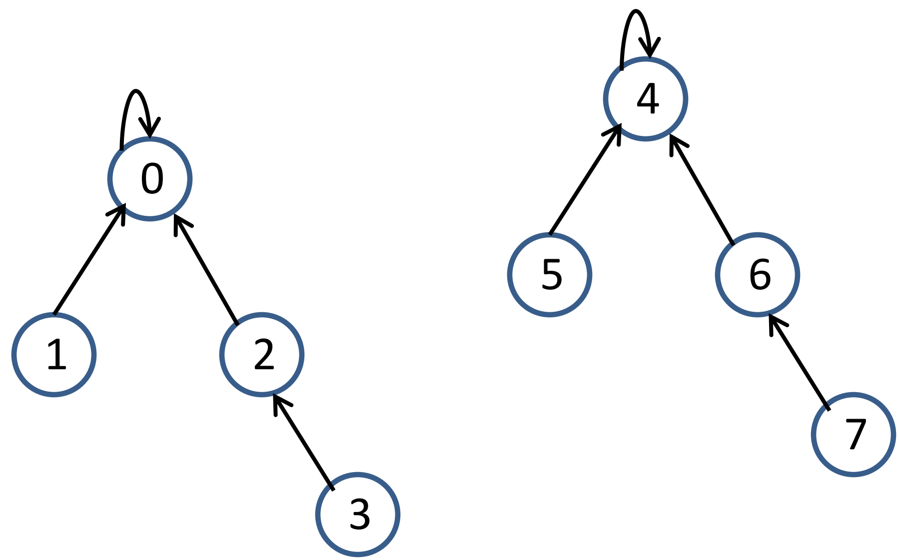
pos	0	1	2	3	4	5	6	7
id	0	0	0	2	4	4	4	6
sz	4	1	2	1	4	1	2	1

Case:
b) 5 – 3

Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6



pos	0	1	2	3	4	5	6	7
id	0	0	0	2	4	4	4	6

Case:

b) 5 – 3

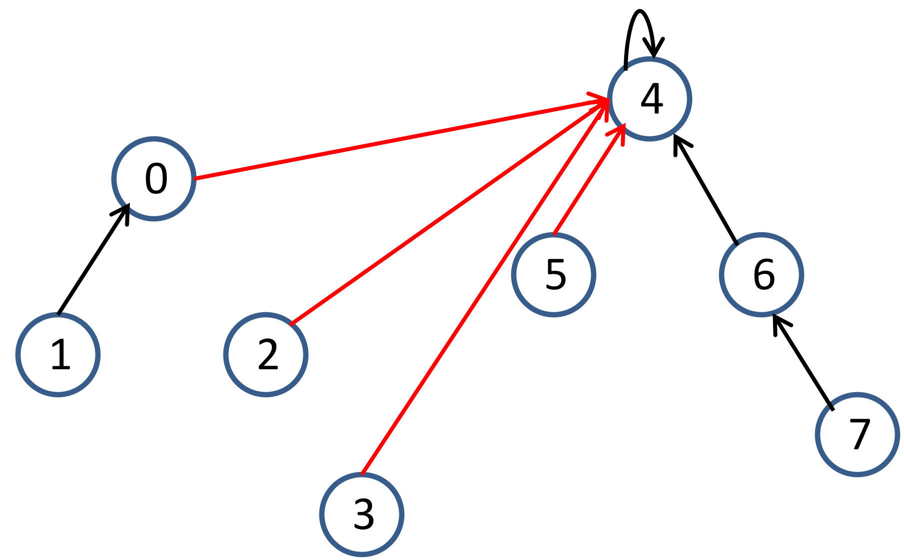
Representative of 5: 4

Representative of 3: 0

Compare sizes: same, (right points to left)

New root: 4

Weighted quick union with full path compression



The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6

pos	0	1	2	3	4	5	6	7
id	**4**	0	**4**	**4**	**4**	4	4	6
sz	4	1	2	1	**8**	1	2	1

Case:

b) 5 – 3

Representative of 5: 4

Representative of 3: 0

Compare sizes: same, (right points to left)

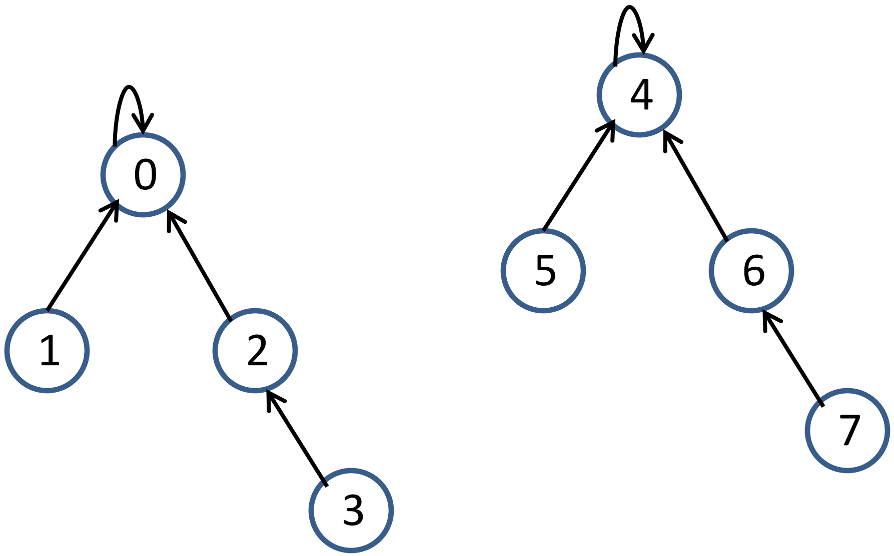
New root: 4

Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6

Case:
c) 5 – 7

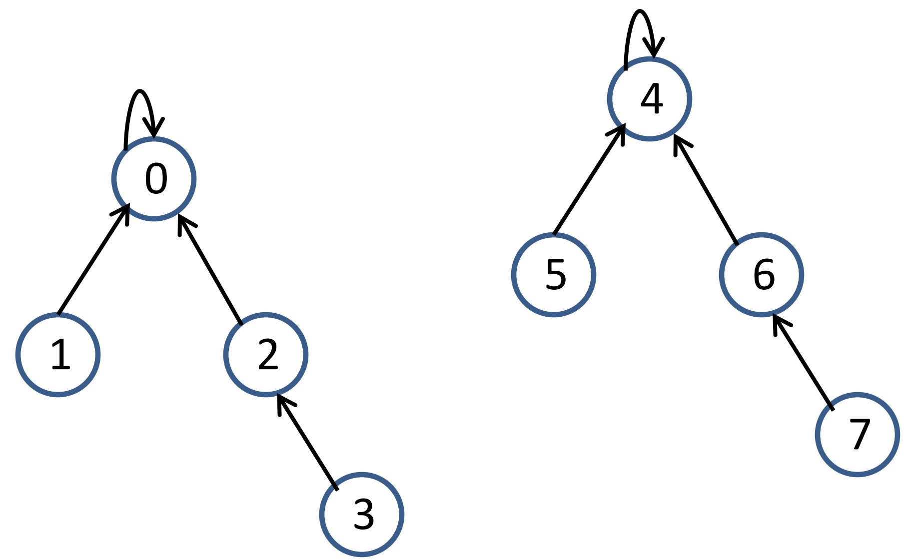


pos	0	1	2	3	4	5	6	7
id	0	0	0	2	4	4	4	6
sz	4	1	2	1	4	1	2	1

Weighted quick union with full path compression

The sequence of pairs below generated the trees to the right using weighted quick union with full path compression.

0 – 1
2 – 3
0 – 2
4 – 5
6 – 7
4 – 6



pos	0	1	2	3	4	5	6	7
id	0	0	0	2	4	4	4	6
sz	4	1	2	1	4	1	2	1

Case:

c) 5 – 7

Representative of 5: 4

Representative of 7: 4

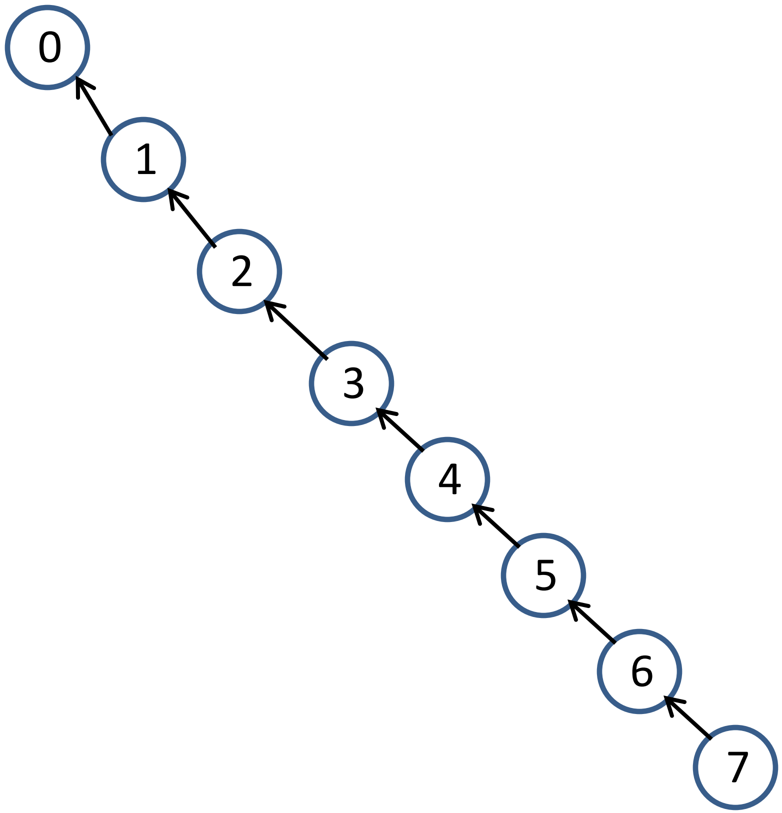
In same component: no union =>

No link gets updated.

Path compression by halving

Path compression by halving

Path compression by halving reduces the path length by making each node **reached** on the path, to skip a node. Notice in the example below that in the search for the representative of 7, the link 6->5 is not updated.



Assume we have this path in a tree and we are given the pair:

7 – 6.

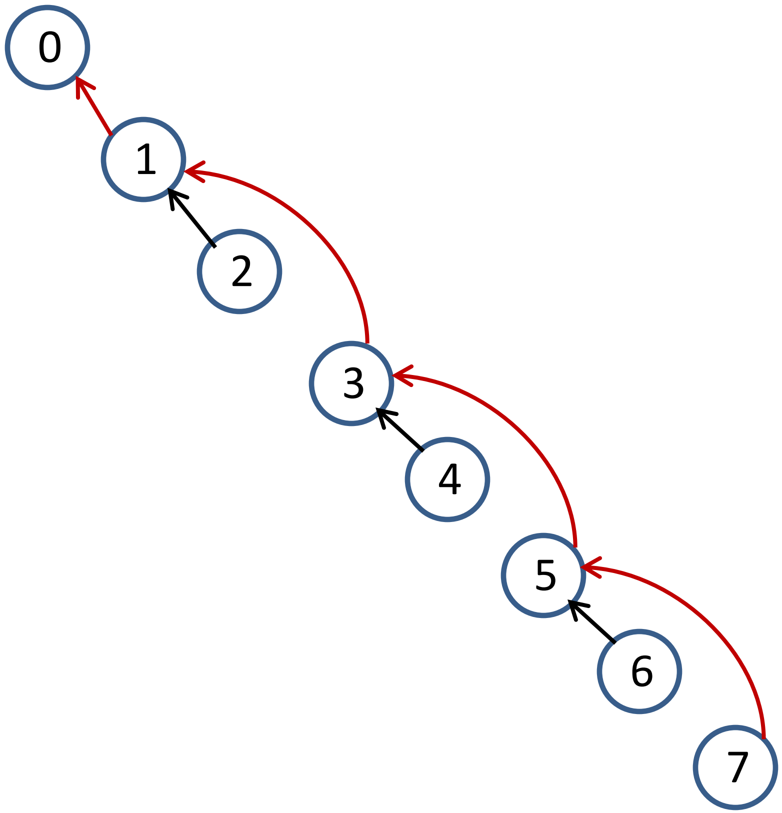
How will the tree be updated as we look up the representative of 7?

Path compression by halving

Assume we have this path in a tree and we are given the pair:

7 – 6.

How will the tree get updated as we look up the representative of **7**?



Path compression by halving

Assume we have this path in a tree and we are given the pair:

7 – 6.

How will the tree get updated as we look up the representative of **6**?