Data Structures for Disjoint Sets

Manolis Koubarakis

Dynamic Sets

- Sets are fundamental for mathematics but also for computer science.
- In computer science, we usually study **dynamic sets** i.e., sets that can grow, shrink or otherwise change over time.
- The data structures we have presented so far in this course offer us ways to represent **finite, dynamic sets** and manipulate them on a computer.

Dynamic Sets and Symbol Tables

- Many of the data structures we have so far presented for symbol tables can be used to implement a dynamic set (e.g., a linked list, a hash table, a (2,4) tree etc.).

Disjoint Sets

- Some applications involve grouping n distinct elements into a collection of **disjoint sets (ξένα σύνολα)**.
- Important operations in this case are to **construct** a set, to **find** which set a given element belongs to, and to **unite** two sets.

Definitions

- A **disjoint-set data structure** maintains a collection $S = \{ S_1, S_2, \cdots, S_n\}$ of disjoint dynamic sets.
- Each set is identified by a **representative (αντιπρόσωπο)**, which is some member of the set.
- The disjoint sets might form a **partition (διαμέριση)** of a universe set U (i.e., their union is the set U).

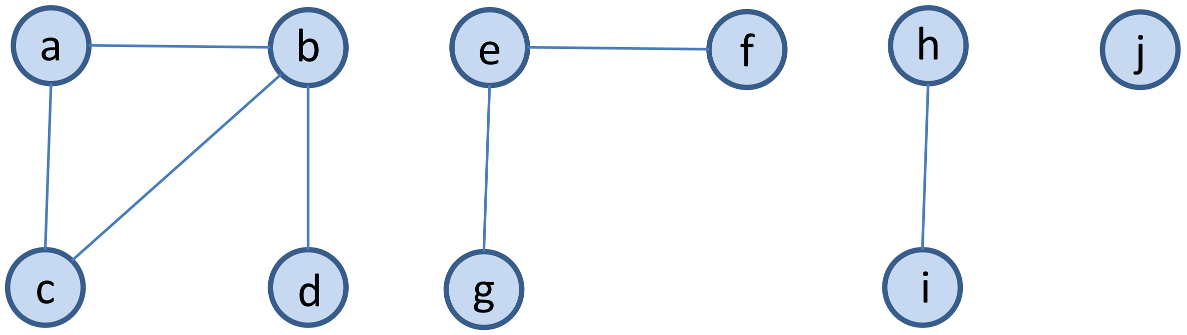
Definitions (cont'd)

- The disjoint-set data structure supports the following operations:
  - **MAKE-SET(x)**: It creates a new set whose only member (and thus representative) is pointed to by x. Since the sets are disjoint, we require that x not already be in any of the existing sets.
  - **UNION(x, y)**: It unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. One of the S_x and S_y give its name to the new set and the other set is "destroyed" by removing it from the collection S. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of $S_x \cup S_y$ (usually the representative of the set that gave its name to the union).
  - **FIND-SET(x)** returns a pointer to the representative of the unique set containing x.

Determining the Connected Components of an Undirected Graph

- One of the many applications of disjoint-set data structures is **determining the connected components (συνεκτικές συνιστώσες) of an undirected graph**.
- The implementation based on disjoint-sets that we will present here is appropriate when the edges of the graph are not static e.g., when **edges are added dynamically** and we need to maintain the connected components as each edge is added.

Example Graph



Computing the Connected Components of an Undirected Graph

- The following procedure CONNECTED-COMPONENTS uses the disjoint-set operations to **compute the connected components of an undirected graph**.

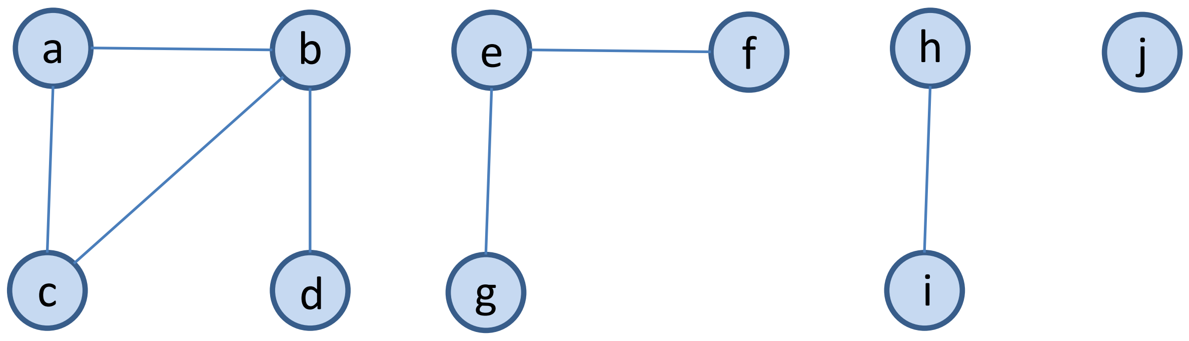
```
CONNECTED-COMPONENTS($G$)
  for each vertex $v \in V[G]$
    do MAKE-SET($v$)
  for each edge $(u, v) \in E[G]$
    do if FIND-SET($u$)$\neq$FIND-SET($v$)
         then UNION($u, v$)
```

Computing the Connected Components (cont'd)

- Once CONNECTED-COMPONENTS has been run as a preprocessing step, the procedure SAME-COMPONENT given below answers queries about whether two vertices are in the same connected component.

```
SAME-COMPONENT($u, v$)
  **if** FIND-SET($u$)=FIND-SET($v$)
    **then return** TRUE
    **else return** FALSE
```

Example Graph



The Collection of Disjoint Sets After Each Edge is Processed

Edge processed	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

Minimum Spanning Trees

- Another application of the disjoint set operations that we will see is **Kruskal’s algorithm** for computing the **minimum spanning tree** of a graph.
- We will see this algorithm in the next lecture.

Maintaining Equivalence Relations

- Another application of disjoint-set data structures is to maintain **equivalence relations.**
- **Definition**. An **equivalence relation** on a set S is relation $\equiv$ with the following properties:
  - **Reflexivity**: for all $a \in S$, we have $a \equiv a$.
  - **Symmetry**: for all $a, b \in S$, if $a \equiv b$ then $b \equiv a$.
  - **Transitivity**: for all $a, b, c \in S$, if $a \equiv b$ and $b \equiv c$ then $a \equiv c$.

Examples of Equivalence Relations

- **Equality** in some set S.
- **Connectivity** of vertices in an undirected graph.

Examples of Equivalence Relations (cont’d)

- **Equivalent type definitions in programming languages**. For example, consider the following type definitions in C:

```
struct A {
   int a;
   int b;
};
typedef A B;
typedef A C;
typedef A D;
```

- The types `A, B, C` and `D` are equivalent in the sense that variables of one type can be assigned to variables of the other types without requiring any casting.

Equivalent Classes

- If a set S has an equivalence relation defined on it, then the set S can be partitioned into disjoint subsets $S_1, S_2, \cdots, S_n$ called **equivalence classes** whose union is S.
- Each subset S_i consists of equivalent members of S. That is, $a \equiv b$ for all a and b in S_i, and $a \not\equiv b$ if a and b are in different subsets.

Example

- Let us consider the set $S = \{0,1,2,\dots,6\}$.
- Let us also consider an equivalence relation $\equiv$ on S defined by the following equivalences:

$$0 \equiv 2, 5 \equiv 6, 3 \equiv 4, 0 \equiv 4, 0 \equiv 3$$

- Note that the relation $0 \equiv 3$ follows from the others given the definition of an equivalence relation.

The Equivalence Problem

- The **equivalence problem** can be formulated as follows.
- We are given a set S and a sequence of statements of the form $a \equiv b$.
- We are to process the statements in order in such a way that, at any time, we are able to determine in which equivalence class a given element of S belongs.

The Equivalence Problem (cont'd)

- We can solve the equivalence problem by starting with each element in a different named set.
- When we process a statement $a \equiv b$, we call FIND-SET(a) and FIND-SET(b).
- If these two calls return different sets then we call UNION to unite these sets. If they return the same set then this statement follows from the other statements and can be discarded.

Example (cont'd)

- We start with each element of S in a set:

$$\{0\}\ \{1\}\ \{2\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}$$

- As the given equivalence relations are processed, these sets are modified as follows:

$0 \equiv 2$ $\{0,2\}\ \{1\}\ \{3\}\ \{4\}\ \{5\}\ \{6\}$

$5 \equiv 6$ $\{0,2\}\ \{1\}\ \{3\}\ \{4\}\ \{5,6\}$

$3 \equiv 4$ $\{0,2\}\ \{1\}\ \{3,4\}\ \{5,6\}$

$0 \equiv 4$ $\{0,2,3,4\}\ \{1\}\ \{5,6\}$

$0 \equiv 3$ follows from the other statements and is discarded

Example (cont'd)

- Therefore, the equivalent classes of S are the subsets $\{0, 2, 3, 4\}$, $\{1\}$ and $\{5, 6\}$.

Implementation in C

- Let us assume that the sets will have positive integers in the range `0` to `N-1` as their members.
- The simplest way to implement in C the disjoint sets data structure is to use an array `id[N]` of integers that take values in the range `0` to `N-1`. This array will be used to keep track of the **representative** of each set but also the **members** of each set.
- Initially, we set `id[i]=i`, for each `i` between `0` and `N-1`. This is equivalent to `N` MAKE-SET operations that create the initial versions of the sets.
- To implement the UNION operation for the sets that contain integers `p` and `q`, we scan the array `id` and change all the array elements that have the value `p` to have the value `q`. In other words, `q` becomes the representative of the union of the two sets.
- The implementation of the FIND-SET(`p`) simply returns the value of `id[p]`.
- This algorithm is called **quick-find.**

Implementation in C (cont'd)

- The program on the next slide initializes the array `id`, and then reads pairs of integers `(p,q)` and performs the operation UNION(`p,q`) if `p` and `q` are not in the same set yet.
- The program is an implementation of the equivalence problem defined earlier.

Implementation in C (cont’d)

```
#include <stdio.h>
#define N 10000
main()
  { int i, p, q, t, id[N];

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

    while (scanf("%d %d", &p, &q) == 2)
      {
        if (id[p] == id[q]) continue;
        for (t = id[p], i = 0; i < N; i++)
          if (id[i] == t) id[i] = id[q];
        printf("%d %d\n", p, q);
      }
 }
```

Example

p	q		id[0]	id[1]	id[2]	id[3]	id[4]	id[5]	id[6]
			0	1	2	3	4	5	6
0	2		2	1	2	3	4	5	6
5	6		2	1	2	3	4	6	6
3	4		2	1	2	4	4	6	6
0	4		4	1	4	4	4	6	6
0	3		4	1	4	4	4	6	6

Complexity Parameters for the Disjoint-Set Data Structures

- We will analyze the running time of our data structures in terms of two parameters:
  - n, the number of objects, and
  - m, the number of pairs of input equivalent objects to be processed.

Proposition

- The quick-find algorithm has time complexity $O(nm)$ where n is the number of objects and m is the number of input pairs of objects.
- Proof?

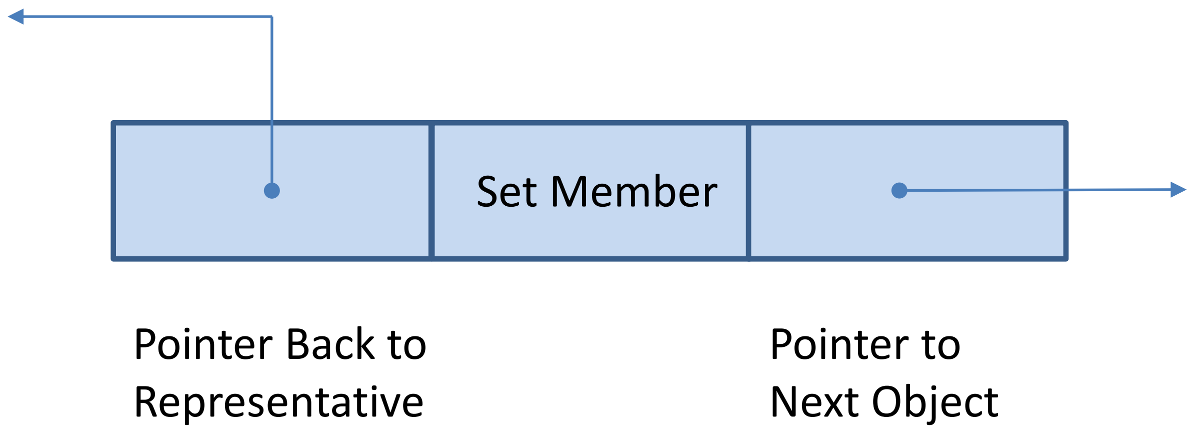
Proof

- For each of the m input pairs, we iterate the `for` loop n times.

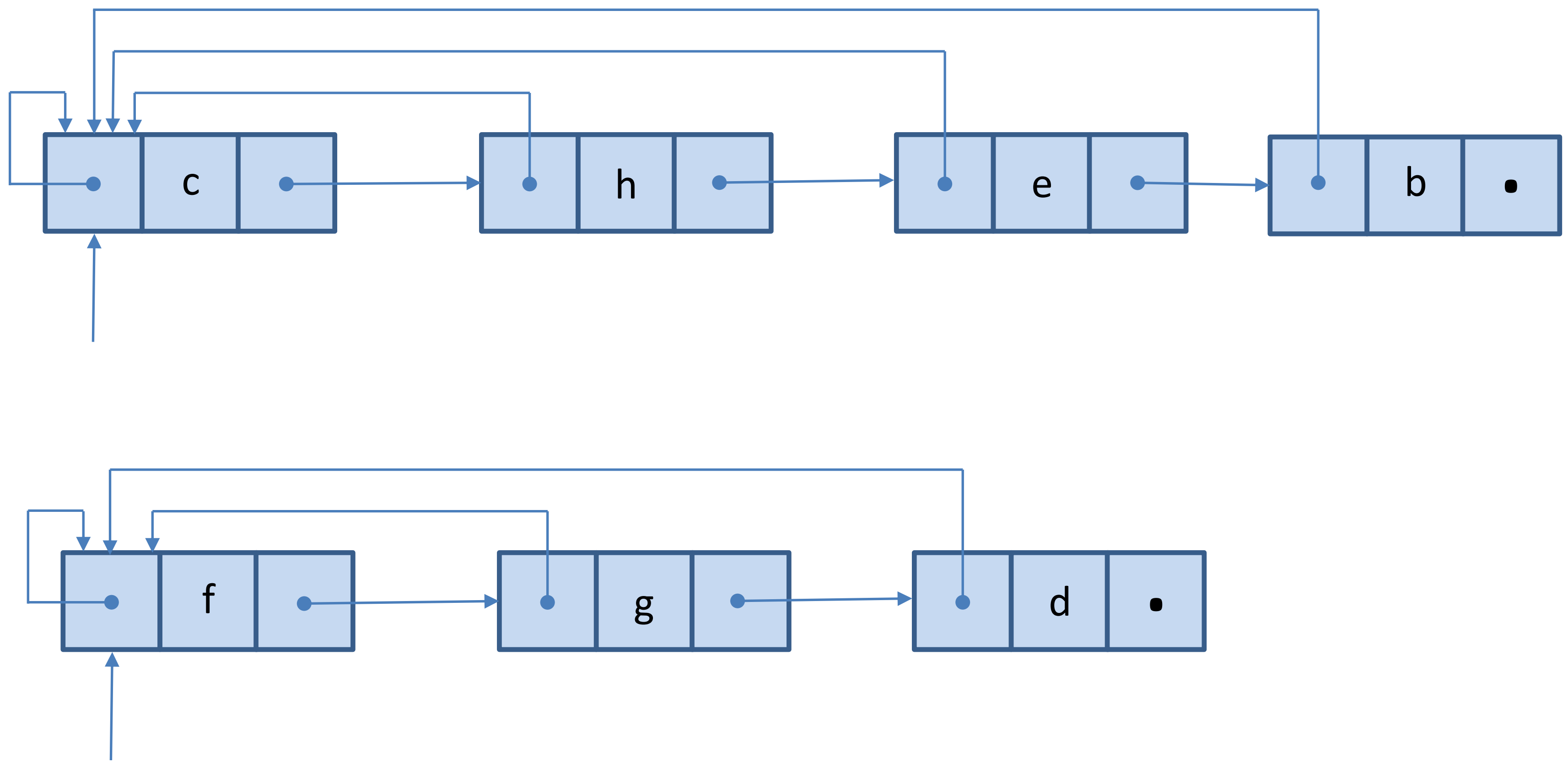
Linked-List Representation of Disjoint Sets

- Another way to implement a disjoint-set data structure is to represent each set by a **linked list.**
- The **first object** in each linked list serves as its set’s **representative**. The remaining objects can appear in the list in any order.
- Each object in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative.

The Structure of Each List Object



Example: the Sets {c, h, e, b} and {f, g, d}



The **representatives** of the two sets are c and f.

Implementation of MAKE-SET and FIND-SET

- With the linked-list representation, both MAKE-SET and FIND-SET are easy.
- To carry out MAKE-SET(x), we create a new linked list which has one object with set element x.
- To carry out, FIND-SET(x), we just return the pointer from x back to the representative.

Implementation of UNION

- To perform UNION(x, y), we can append x's list onto the end of y's list.
- The representative of the new set is the element that was originally the representative of the set containing y.
- We should also update the pointer to the representative for each object originally in x's list.

The Weighted Union Heuristic

- In the above implementation of the UNION operation we may be appending a longer list onto a shorter list, and we must update the pointer to the representative of each member of the longer list.
- If **each representative also includes the length of the list** then we can always append the smaller list onto the longer, with ties broken arbitrarily. This is called the **weighted union heuristic.**
- The word **heuristic** is used in Computer Science to mean “a rule of thumb” that allows us to improve an algorithm.

Implementation in C

- The implementation in C for the case where sets are represented by linked lists is left as an exercise.

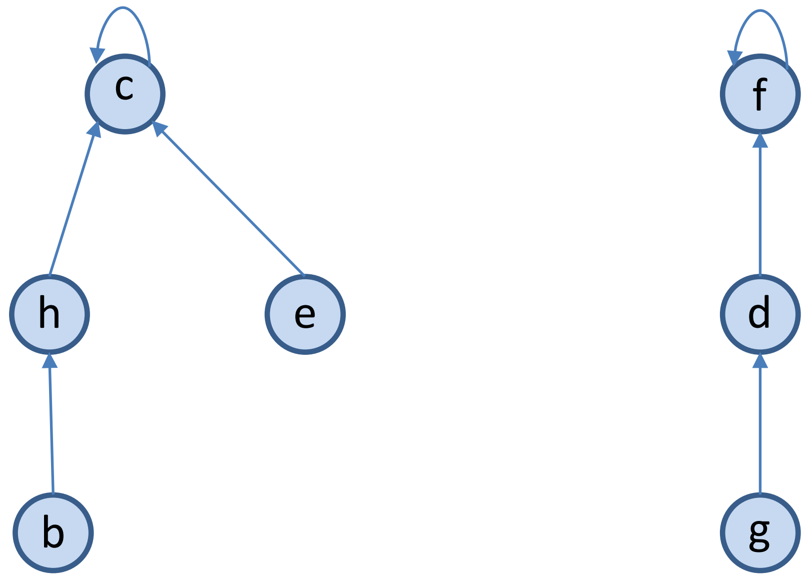
Complexity of Operations for the Linked List Representation

- MAKE-SET and FIND-SET take $O(1)$ time.
- UNION(x, y) takes time $O(|x| + |y|)$ where $|x|$ and $|y|$ denote the cardinalities of the sets that contain x and y. We need $O(|y|)$ time to reach the last object in y's list to make it point to the first object in x's list. We also need $O(|x|)$ time to update all pointers to the representative in x's list.
- If we keep a pointer to the last object in the list in each representative then we do not need to scan y's list, and we only need $O(|x|)$ time to update all pointers to the representative in x's list.
- In the worst case, the complexity of UNION is $O(n)$ since the cardinality of each set can be at most n.

Disjoint-Set Forests

- There is another interesting implementation of disjoint sets in which we represent sets by **directed rooted trees**.
- Each node of a tree represents one set member and each tree represents a set.
- In a tree, each set member points only to its parent. **The root of each tree contains the representative** of the set and is its own parent.
- For many sets, we have a **disjoint-set forest.**

Example: the Sets {b, c, e, h} and {d, f, g}

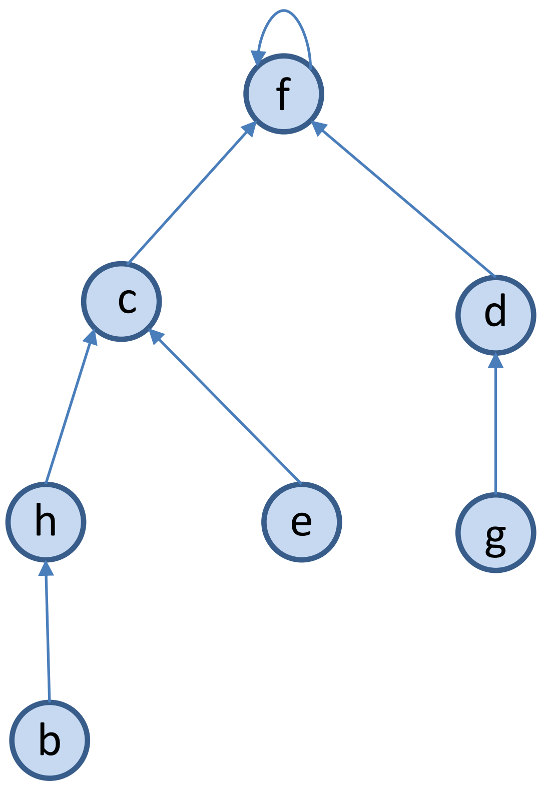


The **representatives** of the two sets are c and f.

Implementing MAKE-SET, FIND-SET and UNION

- A MAKE-SET operation simply creates a tree with just one node.
- A FIND-SET operation can be implemented by chasing parent pointers until we find the root of the tree. The nodes visited on this path towards the root constitute the **find-path**.
- A UNION operation can be implemented by making the root of one tree to point to the root of the other.

Example: the UNION of Sets {b, c, e, h} and {d, f, g}



Implementation in C

- The disjoint-forests data structure can easily be implemented by changing the meaning of the elements of array `id` in our earlier C implementation. Now each `id[i]` **represents the element `i` of a set and points to another element of that set**. **These pointers give the paths towards the root of the tree. The root element points to itself.**
- The program on the next slide illustrates this functionality. Note that after we have found the roots of the two sets, the UNION operation is simply implemented by the assignment statement `id[i]=j`. In other words, element `j` becomes the representative of the united sets and the root of the new tree.
- This algorithm is called **quick-union**.
- The implementation of the FIND-SET(`i`) operation is similar: we just follow pointers starting at `id[i]` until we find the root of the tree.

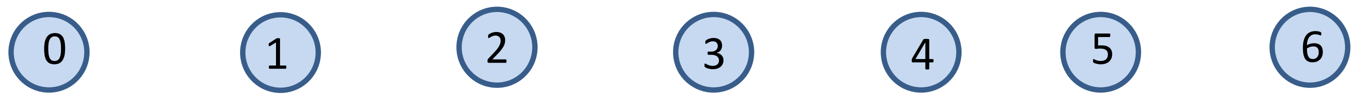
Implementation in C (cont'd)

```
#include <stdio.h>
#define N 10000
main()
  { int i, j, p, q, t, id[N];

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

    while (scanf("%d %d", &p, &q) == 2)
      {
        for (i = p; i != id[i]; i = id[i]) ;
        for (j = q; j != id[j]; j = id[j]) ;
        if (i == j) continue;
        id[i] = j;
        printf("%d %d\n", p, q);
      }
 }
```

Example

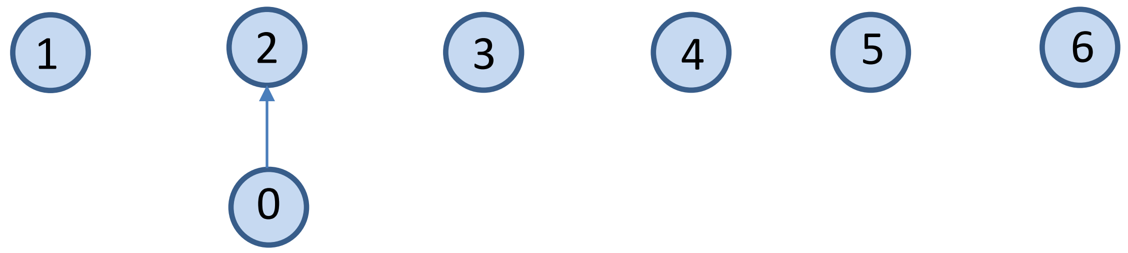
p	q		id[0]	id[1]	id[2]	id[3]	id[4]	id[5]	id[6]
			0	1	2	3	4	5	6
0	2		2	1	2	3	4	5	6
5	6		2	1	2	3	4	6	6
3	4		2	1	2	4	4	6	6
0	4		2	1	4	4	4	6	6
0	3		2	1	4	4	4	6	6
0	5		2	1	4	4	6	6	6

The Example By Showing the Trees



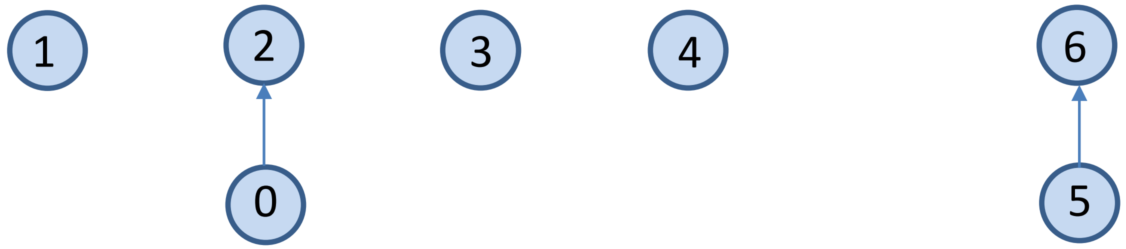
This is the initial forest. We do not show the self-loops.

Example (cont’d)



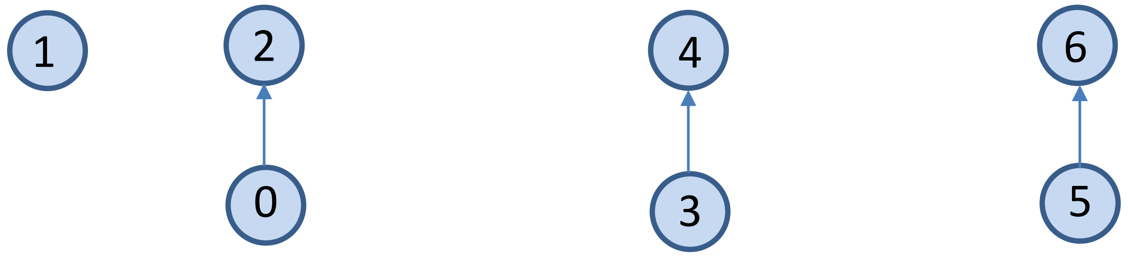
After processing the pair (0,2).

Example (cont’d)



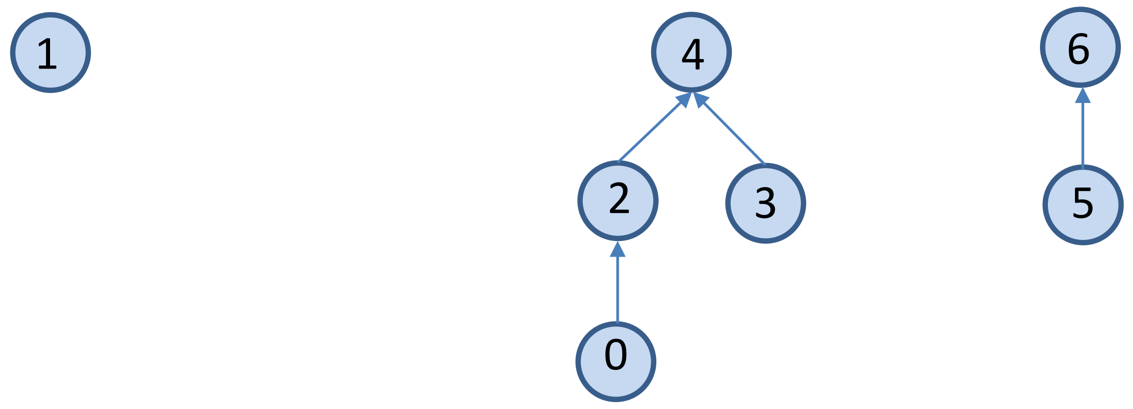
After processing the pair (5,6).

Example (cont’d)



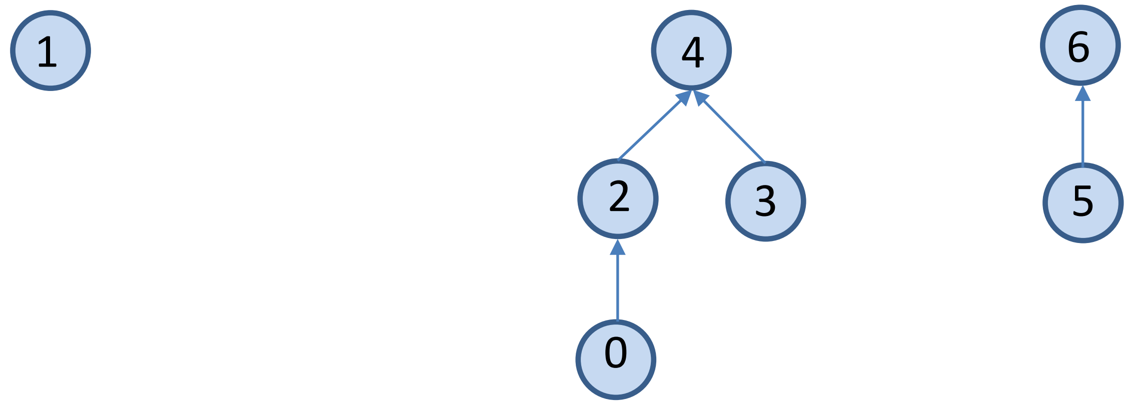
After processing the pair (3,4).

Example (cont’d)



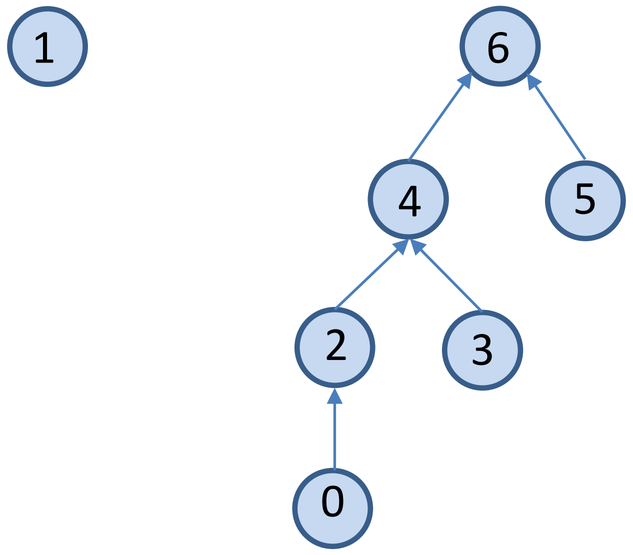
After processing the pair (0,4).

Example (cont'd)



After processing the pair (0,3), there is no change in the forest.

Example (cont’d)



After processing the pair (0,5).

Discussion

- The quick-union algorithm would seem to be faster than the quick-find algorithm, because it does not have to go through the entire array for each input pair; but how much faster is it?
- By **running empirical studies** or studying the **amortized time complexity** of the two algorithms we can show that quick-union is more efficient.
- But we **cannot guarantee** that quick-union will be substantially faster than quick-find in the worst-case because the input data could conspire to make the FIND-SET operation slow.

Proposition

- For $m > n$, the quick-union algorithm could take more than $\frac{mn}{2}$ instructions to solve a disjoint set problem with m pairs of n objects.
- Proof?

Proof

- Suppose that the input pairs come in the order $(1,2)$, then $(2,3)$, then $(3,4)$ and so forth.
- After $n-1$ such pairs, we have n objects all in the same set, and the tree that is formed by the quick-union algorithm is a chain, with n pointing to $n-1$, which points to $n-2$, which points to $n-3$, and so forth.
- To execute the FIND operation for object n, the program has to follow $n-1$ pointers.
- Thus the average number of pointers followed for the first n pairs is

$$\frac{0+1+\cdots+(n-1)}{n}=\frac{n-1}{2}.$$

Proof (cont'd)

- Now suppose that the remaining pairs all connect n to some other object.
- The FIND-SET operation for each of these pairs involves at least $(n-1)$ pointers.
- The grand total for m FIND-SET operations for this sequence of input pairs is certainly greater than $\frac{mn}{2}$.

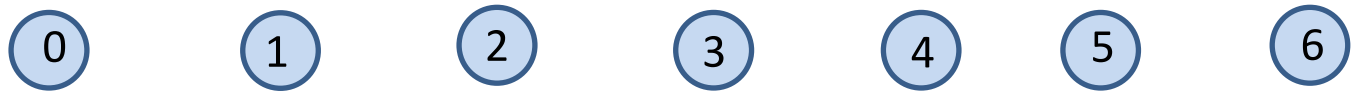
The Weighted Quick-Union Algorithm

- We can implement a **weighted version** of the UNION operation by keeping track of the size of the two trees and making the root of the smaller tree point to the root of the larger.
- The code on the next slide implements this functionality by making use of an array `sz[N]` (for size).

Implementation in C

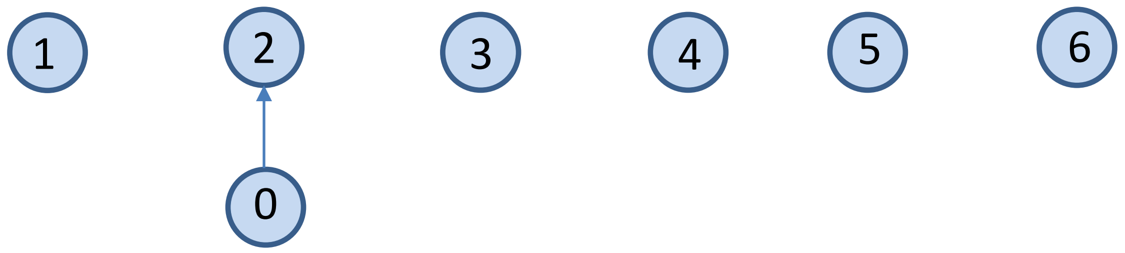
```
#include <stdio.h>
#define N 10000
main()
  { int i, j, p, q, id[N], sz[N];

    for (i = 0; i < N; i++)
      { id[i] = i; sz[i] = 1; }

    while (scanf("%d %d", &p, &q) == 2)
      {
        for (i = p; i != id[i]; i = id[i]) ;
        for (j = q; j != id[j]; j = id[j]) ;
        if (i == j) continue;
        if (sz[i] < sz[j])
             { id[i] = j; sz[j] += sz[i]; }
        else { id[j] = i; sz[i] += sz[j]; }
        printf("%d %d\n", p, q);
      }
 }
```

Example



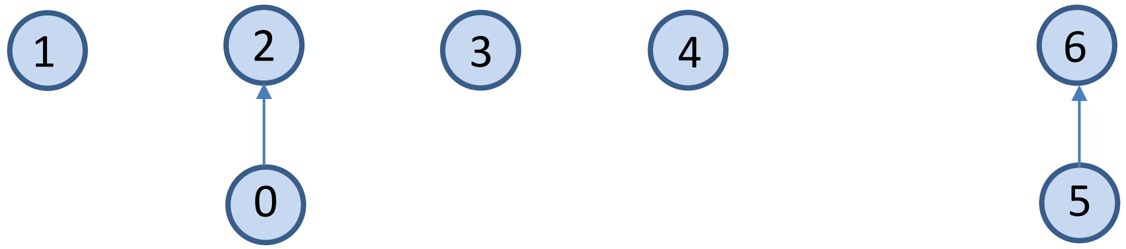
This is the initial forest. We do not show the self-loops.

Example (cont'd)



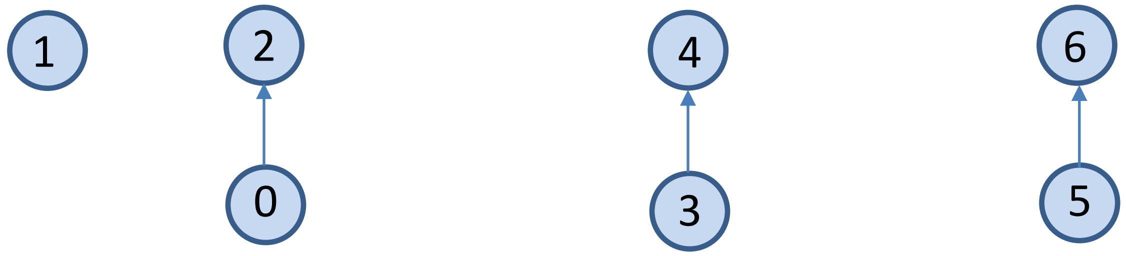
After processing the pair (0,2).

Example (cont'd)



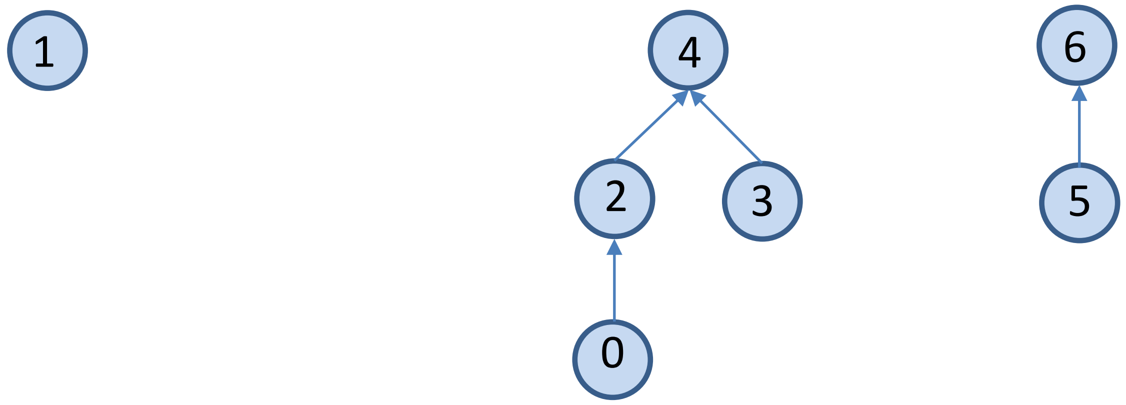
After processing the pair (5,6).

Example (cont'd)



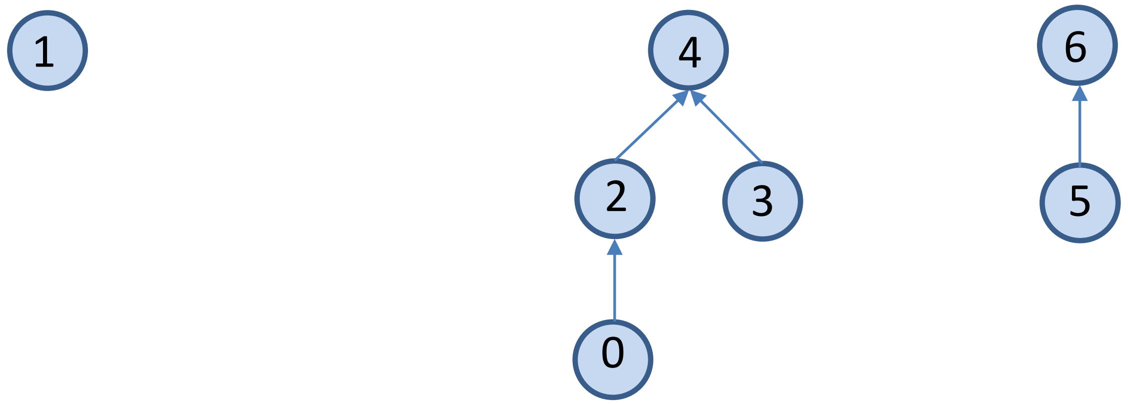
After processing the pair (3,4).

Example (cont'd)



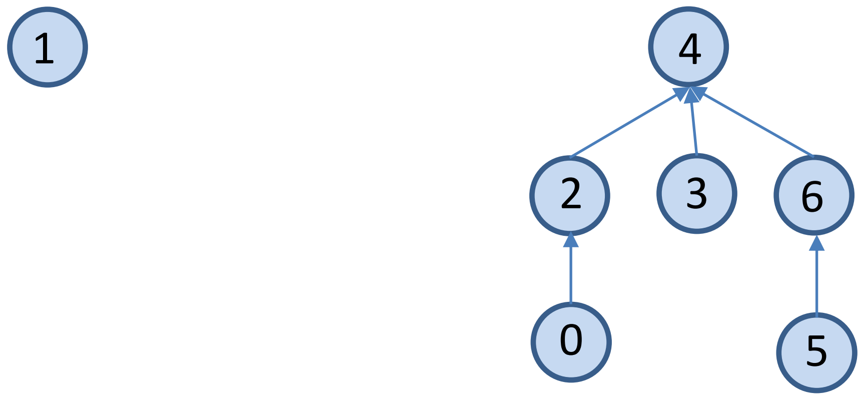
After processing the pair (0,4).

Example (cont'd)



After processing the pair (0,3), there is no change in the forest.

Example (cont'd)



After processing the pair (0,5). The shorter tree is now joined to the taller one and the paths in the resulting tree are shorter.

Proposition

- The weighted quick-union algorithm follows at most $2 \log n$ pointers to determine whether two of n objects are connected.
- Proof?

Proof

- We can prove that the Union operation preserves the property that the number of pointers followed from any node to the root in a set of k objects is no greater than $\log k$. The reason for this is as follows.
- The property holds at the beginning of the algorithm.
- When we combine a set of i nodes with a set of j nodes with $i \leq j$, we increase the number of pointers that must be followed in the smaller set by 1, but they are now in a set of size $i + j$, so the property is preserved because

$$1 + \log i = \log(i + i) \leq \log(i + j)\,.$$

Discussion

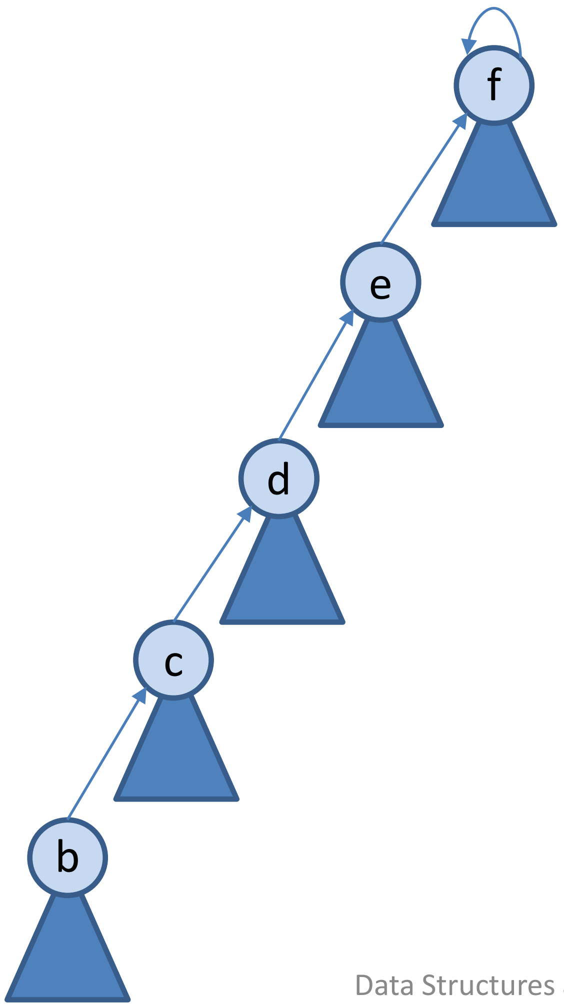
- The practical implication of the previous proposition is that the weighted quick-union algorithm uses $\boldsymbol{O(m \log n)}$ instructions to process m pairs of n objects.
- This result is in stark contrast to our finding that quick-find always (and quick union sometimes) uses at least $\frac{mn}{2}$ instructions.
- The conclusion is that, with weighted quick-union, we can guarantee that we can solve huge practical problems in a reasonable amount of time.

- Empirical studies show that the weighted quick-union algorithm can solve practical problems in **time linear in $\boldsymbol{m}$.** However, it has been shown that the problem cannot be solved in time linear in m in the worst case. The best we can hope for is $O(m \log n)$.

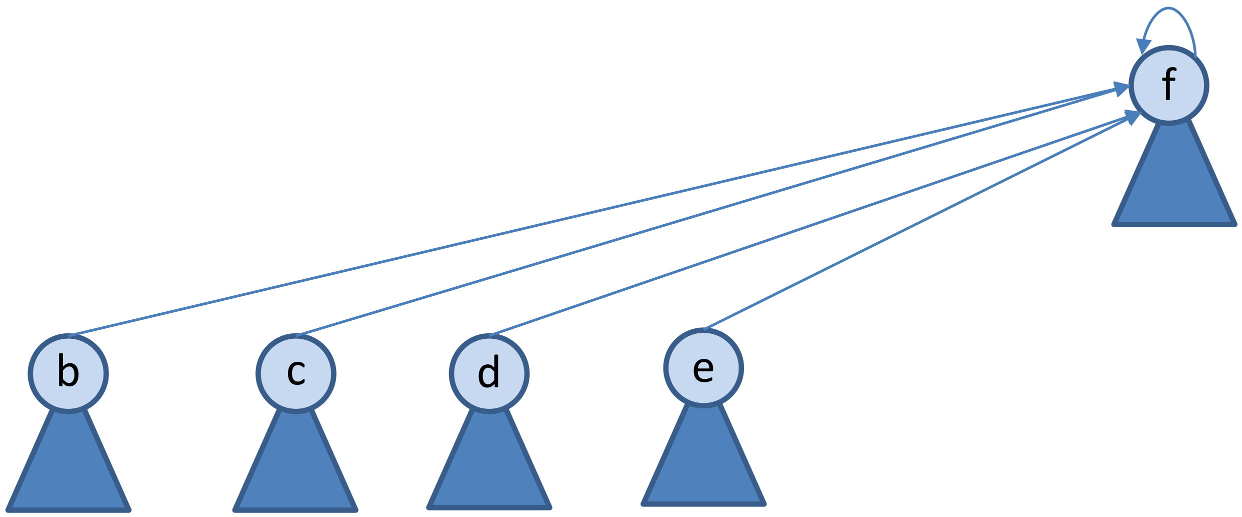
The Path Compression Heuristic

- We can extend the weighted quick-union algorithm with a second heuristic, called **path compression (συμπίεση μονοπατιού)**, which is also simple and very effective.
- This heuristic is used during FIND-SET operations to **make each node on the find-path point directly to the root.**
- In this way, trees with **small height** are constructed.
- Path compression does not change any weights.

The Path Compression Heuristic Graphically



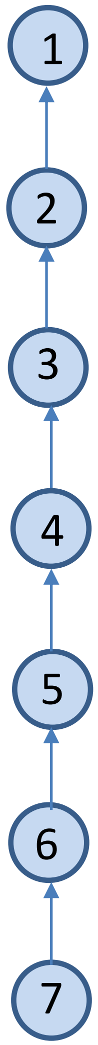
The Path Compression Heuristic Graphically (cont'd)

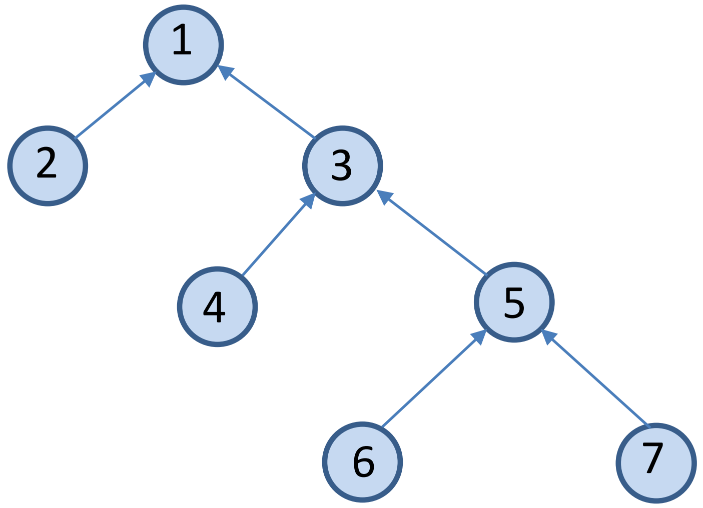


Path Compression by Halving

- An easy way to implement path compression is by **halving the length of paths on the way up the tree** by taking two links at a time, and setting the bottom one point to the same node as the top one, as shown in the next figure.

Example (cont'd)





When traversing the tree on the left from node 7 up to node 1, we can halve its height as shown on the right.

Path Compression by Halving (cont’d)

- The new algorithm is easily implemented by replacing the `for` loops of the weighted quick-union program as shown on the next slide.

Implementation in C

```
#include <stdio.h>
#define N 10000
main()
  { int i, j, p, q, id[N], sz[N];

    for (i = 0; i < N; i++)
      { id[i] = i; sz[i] = 1; }

    while (scanf("%d %d", &p, &q) == 2)
      {
        for (i = p; i != id[i]; i = id[i])
             id[i]=id[id[i]];
        for (j = q; j != id[j]; j = id[j])
             id[j]=id[id[j]];
        if (i == j) continue;
        if (sz[i] < sz[j])
             { id[i] = j; sz[j] += sz[i]; }
        else { id[j] = i; sz[i] += sz[j]; }
        printf("%d %d\n", p, q);
      }
 }
```

Readings

- Robert Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος.
  - Κεφάλαιο 1
- T.H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. MIT Press.
  - Chapter 22