1. Δυναμικά Σύνολα

- Τα σύνολα είναι θεμελιώδη για τα μαθηματικά αλλά και για την επιστήμη των υπολογιστών.
- Στην επιστήμη των υπολογιστών, συνήθως μελετάμε *δυναμικά σύνολα*, δηλαδή σύνολα που μπορούν να αυξάνονται, να μειώνονται ή γενικά να αλλάζουν με την πάροδο του χρόνου.
- Οι δομές δεδομένων που έχουμε παρουσιάσει μέχρι τώρα σε αυτό το μάθημα μας προσφέρουν τρόπους να αναπαριστούμε πεπερασμένα, δυναμικά σύνολα και να τα χειριζόμαστε σε έναν υπολογιστή.

---

2. Δυναμικά Σύνολα και Πίνακες Συμβόλων

- Πολλές από τις δομές δεδομένων που έχουμε παρουσιάσει μέχρι τώρα για πίνακες συμβόλων μπορούν να χρησιμοποιηθούν για την υλοποίηση ενός δυναμικού συνόλου (π.χ., συνδεδεμένη λίστα, πίνακας κατακερματισμού, δέντρο (2,4) κ.λπ.).

3. Ξένα Σύνολα (Disjoint Sets)

- Ορισμένες εφαρμογές περιλαμβάνουν την ομαδοποίηση n διαφορετικών στοιχείων σε μια συλλογή από ξένα σύνολα (δηλαδή σύνολα που δεν έχουν κοινά στοιχεία).
- Σημαντικές λειτουργίες σε αυτήν την περίπτωση είναι:
  - η δημιουργία ενός συνόλου,
  - ο εντοπισμός του συνόλου στο οποίο ανήκει ένα συγκεκριμένο στοιχείο, και
  - η ένωση δύο συνόλων.

Ορισμοί

- Μια δομή δεδομένων ξένων συνόλων (disjoint-set) διατηρεί μια συλλογή $S = \{S_1, S_2, \ldots, S_n\}$ από ξένα δυναμικά σύνολα.
- Κάθε σύνολο προσδιορίζεται από έναν *αντιπρόσωπο*, που είναι κάποιο μέλος του συνόλου.
- Τα ξένα σύνολα μπορεί να σχηματίζουν μια *διαμέριση* (partition) ενός καθολικού συνόλου U (δηλαδή η ένωσή τους να ισούται με το σύνολο U).

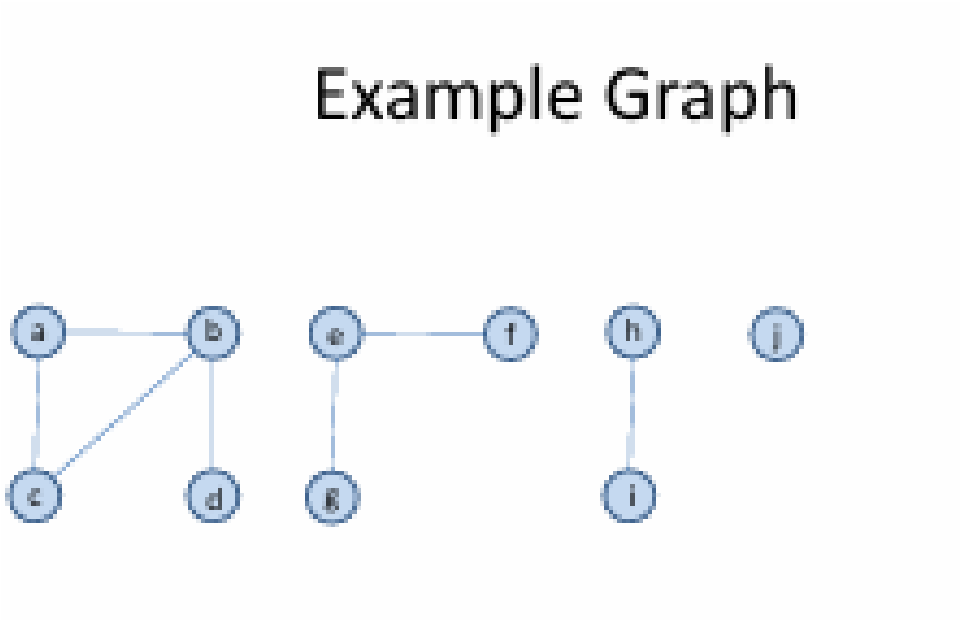
Ορισμοί (συνέχεια)

Η δομή δεδομένων ξένων συνόλων υποστηρίζει τις εξής λειτουργίες:

- **MAKE-SET(x):** Δημιουργεί ένα νέο σύνολο του οποίου μοναδικό μέλος (και συνεπώς αντιπρόσωπος) είναι το στοιχείο που δείχνει ο δείκτης x. Επειδή τα σύνολα είναι ξένα, απαιτείται το x να μην ανήκει ήδη σε κάποιο από τα υπάρχοντα σύνολα.
- **UNION(x, y):** Ενώνει τα δυναμικά σύνολα που περιέχουν τα x και y, π.χ. S_x και S_y, σε ένα νέο σύνολο που είναι η ένωση αυτών των δύο. Ένα από τα S_x ή S_y δίνει το όνομα στο νέο σύνολο και το άλλο «καταστρέφεται» αφαιρούμενο από τη συλλογή S. Τα δύο σύνολα υποτίθεται ότι είναι ξένα πριν την εκτέλεση της πράξης. Ο αντιπρόσωπος του νέου συνόλου είναι κάποιο μέλος του $S_x \cup S_y$ (συνήθως ο αντιπρόσωπος του συνόλου που έδωσε το όνομα στην ένωση).
- **FIND-SET(x):** Επιστρέφει έναν δείκτη στον αντιπρόσωπο του μοναδικού συνόλου που περιέχει το στοιχείο x.

Προσδιορισμός των Συνεκτικών Συνιστωσών σε Μη Κατευθυνόμενο Γράφο

- Μία από τις πολλές εφαρμογές των δομών ξένων συνόλων είναι ο προσδιορισμός των *συνεκτικών συνιστωσών* ενός *μη κατευθυνόμενου γράφου*.
- Η υλοποίηση που θα παρουσιάσουμε βασίζεται στα ξένα σύνολα και είναι κατάλληλη όταν οι ακμές του γράφου δεν είναι στατικές, π.χ. όταν οι ακμές προστίθενται δυναμικά και πρέπει να διατηρούμε τις συνεκτικές συνιστώσες καθώς κάθε ακμή προστίθεται.



Υπολογισμός των Συνεκτικών Συνιστωσών σε Μη Κατευθυνόμενο Γράφο

- Η παρακάτω διαδικασία **CONNECTED-COMPONENTS** χρησιμοποιεί τις λειτουργίες των ξένων συνόλων για να υπολογίσει τις συνεκτικές συνιστώσες ενός μη κατευθυνόμενου γράφου.

CONNECTED-COMPONENTS(*G*)

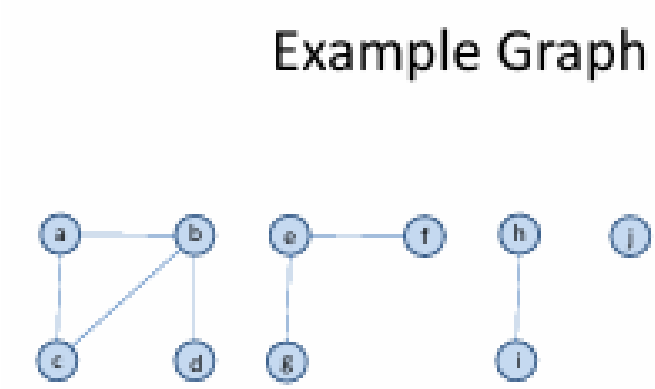
```plaintext
για κάθε κόμβο v ∈ V
    εκτέλεσε MAKE-SET(v)
για κάθε ακμή (u, v) ∈ E
    αν FIND-SET(u) ≠ FIND-SET(v)
        τότε εκτέλεσε UNION(u, v)
```

Υπολογισμός των Συνεκτικών Συνιστωσών (συνέχεια)

- Μόλις η διαδικασία **CONNECTED-COMPONENTS** εκτελεστεί ως προκαταρκτικό βήμα, η παρακάτω διαδικασία **SAME-COMPONENT** απαντά σε ερωτήματα σχετικά με το αν δύο κορυφές ανήκουν στην ίδια συνεκτική συνιστώσα.

SAME-COMPONENT(u, v)

```plaintext
αν FIND-SET(u) = FIND-SET(v)
    τότε επέστρεψε TRUE
αλλιώς επέστρεψε FALSE
```



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}

Ελάχιστα Δέντρα Εκτεταμένου Καλύμματος (Minimum Spanning Trees)

- Μια άλλη εφαρμογή των λειτουργιών των ξένων συνόλων που θα δούμε είναι ο **αλγόριθμος του Kruskal** για τον υπολογισμό του ελάχιστου εκτεταμένου δέντρου ενός γράφου.
- Τον αλγόριθμο αυτόν θα τον δούμε στο επόμενο μάθημα.

Διατήρηση Σχέσεων Ισοδυναμίας

- Μια άλλη εφαρμογή των δομών δεδομένων ξένων συνόλων είναι η διατήρηση *σχέσεων ισοδυναμίας*.
- **Ορισμός:** Μια *σχέση ισοδυναμίας* πάνω σε ένα σύνολο S είναι μια σχέση $\equiv$ με τις εξής ιδιότητες:
  - **Ανακλαστικότητα (Reflexivity):** για κάθε $a \in S$, ισχύει $a \equiv a$.
  - **Συμμετρία (Symmetry):** για κάθε $a, b \in S$, αν $a \equiv b$, τότε $b \equiv a$.
  - **Μεταβατικότητα (Transitivity):** για κάθε $a, b, c \in S$, αν $a \equiv b$ και $b \equiv c$, τότε $a \equiv c$.

Παραδείγματα Σχέσεων Ισοδυναμίας

- Η ισότητα μέσα σε κάποιο σύνολο S.
- Η συνδεσιμότητα μεταξύ κορυφών σε μη κατευθυνόμενο γράφο.

Παραδείγματα Σχέσεων Ισοδυναμίας (συνέχεια)

- Ισοδύναμοι ορισμοί τύπων σε γλώσσες προγραμματισμού.
  Για παράδειγμα, οι παρακάτω ορισμοί τύπων στη C:

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

- Οι τύποι `A`, `B`, `C` και `D` είναι ισοδύναμοι με την έννοια ότι μεταβλητές του ενός τύπου μπορούν να ανατεθούν σε μεταβλητές των άλλων τύπων χωρίς να απαιτείται μετατροπή τύπου (casting).

Ισοδύναμες Κλάσεις (Equivalence Classes)

- Αν σε ένα σύνολο S έχει οριστεί μια σχέση ισοδυναμίας, τότε το S μπορεί να διαιρεθεί σε ξένα υποσύνολα $S_1, S_2, \ldots, S_n$, που λέγονται *κλάσεις ισοδυναμίας* (equivalence classes), των οποίων η ένωση είναι το ίδιο το S.
- Κάθε υποσύνολο S_i αποτελείται από ισοδύναμα μέλη του S. Δηλαδή, για όλα τα a και b στο S_i, ισχύει $a \equiv b$, ενώ για στοιχεία σε διαφορετικά υποσύνολα, δεν ισχύει ισοδυναμία: $a \not\equiv b$.

Παράδειγμα

- Ας θεωρήσουμε το σύνολο $S = \{0, 1, 2, \ldots, 6\}$.
- Ας θεωρήσουμε επίσης μια σχέση ισοδυναμίας $\equiv$ στο S, που ορίζεται από τις παρακάτω ισοδυναμίες:
  $0 \equiv 2, \quad 5 \equiv 6, \quad 3 \equiv 4, \quad 0 \equiv 4, \quad 0 \equiv 3$
- Σημείωσε ότι η σχέση $0 \equiv 3$ προκύπτει από τις άλλες, βάσει του ορισμού της σχέσης ισοδυναμίας.

Το Πρόβλημα της Ισοδυναμίας

- Το πρόβλημα της ισοδυναμίας μπορεί να διατυπωθεί ως εξής:
- Δίνεται ένα σύνολο S και μια ακολουθία από δηλώσεις της μορφής $a \equiv b$.
- Πρέπει να επεξεργαστούμε τις δηλώσεις με τέτοιο τρόπο ώστε, οποιαδήποτε στιγμή, να μπορούμε να προσδιορίσουμε σε ποια *κλάση ισοδυναμίας* ανήκει κάποιο στοιχείο του S.

Το Πρόβλημα της Ισοδυναμίας (συνέχεια)

- Το πρόβλημα επιλύεται ξεκινώντας με κάθε στοιχείο να ανήκει σε ένα ξεχωριστό, ονομασμένο σύνολο.
- Όταν επεξεργαζόμαστε μια δήλωση $a \equiv b$, καλούμε `FIND-SET(a)` και `FIND-SET(b)`.
- Αν επιστρέφουν διαφορετικά σύνολα, καλούμε `UNION(a, b)` για να τα ενώσουμε. Αν επιστρέφουν το ίδιο σύνολο, τότε η δήλωση είναι πλεονάζουσα (συνάγεται από προηγούμενες).

Παράδειγμα (συνέχεια)

- Ξεκινάμε με κάθε στοιχείο του S σε ξεχωριστό σύνολο:
  $0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6$
- Καθώς επεξεργαζόμαστε τις δοθείσες ισοδυναμίες, τα σύνολα μεταβάλλονται ως εξής:

plaintext Αντιγραφή Επεξεργασία

```
0 ≡ 2    →   0,2   1   3   4   5   6
5 ≡ 6    →   0,2   1   3   4   5,6
3 ≡ 4    →   0,2   1   3,4   5,6
0 ≡ 4    →   0,2,3,4   1   5,6
0 ≡ 3    →   αγνοείται (ήδη προκύπτει)
```

Τελικό αποτέλεσμα:
Οι κλάσεις ισοδυναμίας του S είναι τα υποσύνολα:

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

Υλοποίηση σε C

- Έστω ότι τα σύνολα περιέχουν θετικούς ακέραιους από το 0 έως το $N - 1$.
- Η απλούστερη υλοποίηση σε C της δομής δεδομένων ξένων συνόλων είναι με έναν πίνακα `id[N]` από ακέραιους.
- Αυτός ο πίνακας κρατά τόσο τον *αντιπρόσωπο* κάθε συνόλου όσο και τα μέλη κάθε συνόλου.
- Αρχικά, θέτουμε `id[i] = i` για κάθε $i \in [0, N - 1]$ — αυτό αντιστοιχεί σε N πράξεις `MAKE-SET`.

Πράξη `UNION(p, q)`

- Για να ενώσουμε τα σύνολα των p και q, διατρέχουμε τον πίνακα `id` και αλλάζουμε όλα τα στοιχεία με τιμή `p` σε `q`.
- Δηλαδή, ο q γίνεται αντιπρόσωπος της ένωσης των δύο συνόλων.

Πράξη `FIND-SET(p)`

- Επιστρέφει απλώς την τιμή `id[p]`.

Αυτός ο αλγόριθμος λέγεται **quick-find**.

Υλοποίηση σε C (συνέχεια)

- Το πρόγραμμα της επόμενης διαφάνειας αρχικοποιεί τον πίνακα `id` και στη συνέχεια διαβάζει ζεύγη ακεραίων `(p, q)`.
- Για κάθε ζεύγος, εκτελεί την `UNION(p, q)` εφόσον τα `p` και `q` δεν ανήκουν ήδη στο ίδιο σύνολο.
- Το πρόγραμμα είναι υλοποίηση του προβλήματος της ισοδυναμίας που ορίστηκε προηγουμένως.

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

Παράμετροι Πολυπλοκότητας για τις Δομές Δεδομένων Ξένων Συνόλων

- Θα αναλύσουμε το χρόνο εκτέλεσης των δομών δεδομένων μας με βάση δύο παραμέτρους:
  - n, ο αριθμός των αντικειμένων, και
  - m, ο αριθμός των ζευγών ισοδύναμων αντικειμένων που πρέπει να επεξεργαστούμε.

Πρόταση

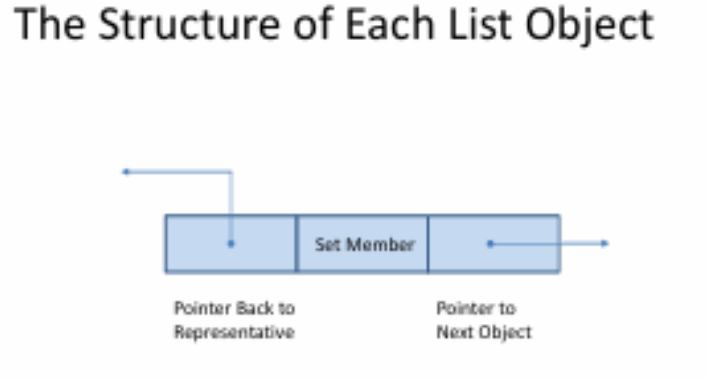
- Ο αλγόριθμος **quick-find** έχει χρονική πολυπλοκότητα $\Theta(nm)$, όπου:
  - n είναι ο αριθμός των αντικειμένων και
  - m είναι ο αριθμός των ζευγών εισόδου.
- **Απόδειξη:**

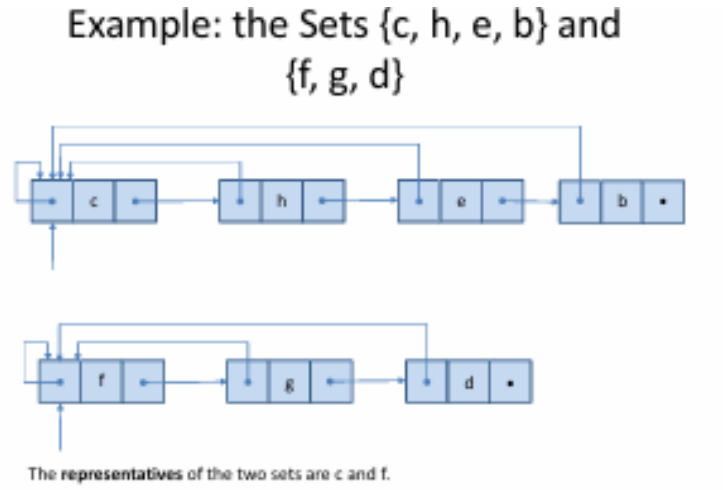
Απόδειξη

- Για κάθε ένα από τα m ζεύγη εισόδου, εκτελούμε μια επανάληψη του `for` βρόχου n φορές (για να αλλάξουμε τιμές στον πίνακα `id`).
- Άρα, συνολικά έχουμε $m \times n$ επαναλήψεις, δηλαδή πολυπλοκότητα $\Theta(nm)$.

Αναπαράσταση Ξένων Συνόλων με Συνδεδεμένες Λίστες

- Μια άλλη μέθοδος υλοποίησης της δομής ξένων συνόλων είναι μέσω **συνδεδεμένων λιστών**.
- Κάθε σύνολο αναπαρίσταται από μία συνδεδεμένη λίστα.
- Το **πρώτο αντικείμενο** κάθε λίστας λειτουργεί ως **αντιπρόσωπος** του συνόλου.
- Τα υπόλοιπα αντικείμενα μπορούν να βρίσκονται σε οποιαδήποτε σειρά μέσα στη λίστα.
- Κάθε αντικείμενο της λίστας περιέχει:
  - Ένα μέλος του συνόλου,
  - Έναν δείκτη στο επόμενο αντικείμενο της λίστας,
  - Έναν δείκτη πίσω προς τον αντιπρόσωπο του συνόλου.





Υλοποίηση των MAKE-SET και FIND-SET

- Με την αναπαράσταση μέσω **συνδεδεμένων λιστών**, τόσο η `MAKE-SET` όσο και η `FIND-SET` είναι απλές:
- Για να εκτελέσουμε MAKE-SET(x):
  - Δημιουργούμε μια νέα συνδεδεμένη λίστα η οποία περιέχει μόνο ένα αντικείμενο με στοιχείο το x.
- Για να εκτελέσουμε FIND-SET(x):
  - Επιστρέφουμε τον δείκτη από το x προς τον **αντιπρόσωπο** του συνόλου.

Υλοποίηση της UNION

- Για να εκτελέσουμε UNION(x, y):
  - Προσαρτούμε (ενώνουμε) τη λίστα του x στο τέλος της λίστας του y.
  - Ο **αντιπρόσωπος** του νέου συνόλου είναι το στοιχείο που ήταν ήδη αντιπρόσωπος του συνόλου του y.
  - Πρέπει επίσης να ενημερώσουμε τον δείκτη προς τον αντιπρόσωπο για κάθε αντικείμενο που ήταν αρχικά στη λίστα του x.

Ο Βαρυμένος Κανόνας Ένωσης (Weighted Union Heuristic)

- Στην παραπάνω υλοποίηση της `UNION`, μπορεί να συμβαίνει η μεγαλύτερη λίστα να ενώνεται με μια μικρότερη, με αποτέλεσμα να πρέπει να ενημερωθούν περισσότεροι δείκτες προς τον αντιπρόσωπο.
- Αν κάθε αντιπρόσωπος κρατάει και το **μήκος της λίστας** του, μπορούμε να **ενώνουμε πάντα τη μικρότερη λίστα με τη μεγαλύτερη**.
  - Αν έχουν ίδιο μήκος, επιλέγουμε αυθαίρετα ποια θα ενωθεί με ποια.
- Αυτή η τεχνική ονομάζεται **weighted union heuristic** (ευρετική ένωσης κατά βάρος).
- Ο όρος **ευρετική (heuristic)** στην Πληροφορική σημαίνει ένας **εμπειρικός κανόνας** που βοηθά στη βελτίωση ενός αλγορίθμου.

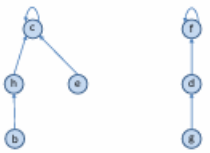
Υλοποίηση σε C

- Η υλοποίηση σε C, για την περίπτωση όπου τα σύνολα αναπαρίστανται με **συνδεδεμένες λίστες**, **αφήνεται ως άσκηση**.

Πολυπλοκότητα των Λειτουργιών στην Αναπαράσταση με Συνδεδεμένες Λίστες

- Οι συναρτήσεις MAKE-SET και FIND-SET έχουν πολυπλοκότητα $O(1)$.
- Η **UNION**(x, y) έχει πολυπλοκότητα $O(|S_x| + |S_y|)$, όπου $|S_x|$ και $|S_y|$ είναι τα πλήθη των στοιχείων των συνόλων που περιέχουν τα x και y.
  - Χρειαζόμαστε $O(|S_y|)$ χρόνο για να φτάσουμε στο τελευταίο στοιχείο της λίστας του y ώστε να το κάνουμε να δείχνει στο πρώτο στοιχείο της λίστας του x.
  - Χρειαζόμαστε επίσης $O(|S_x|)$ χρόνο για να ενημερώσουμε όλους τους δείκτες προς τον νέο αντιπρόσωπο στη λίστα του x.
- Αν διατηρούμε **δείκτη προς το τελευταίο στοιχείο της λίστας** σε κάθε αντιπρόσωπο, τότε δεν χρειάζεται να σαρώσουμε τη λίστα του y. Άρα χρειαζόμαστε μόνο $O(|S_x|)$ χρόνο.
- Στη χειρότερη περίπτωση, η πολυπλοκότητα της **UNION** είναι $O(n)$, αφού κάθε σύνολο μπορεί να περιέχει έως n στοιχεία.

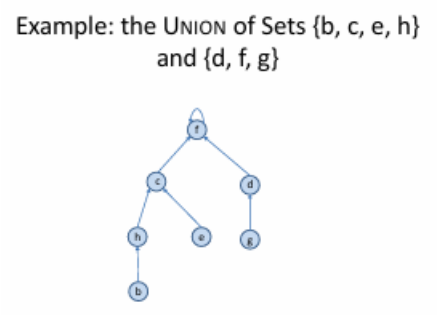
Example: the Sets {b, c, e, h} and {d, f, g}



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

Υλοποίηση των MAKE-SET, FIND-SET και UNION

- Η λειτουργία **MAKE-SET** απλώς δημιουργεί ένα δέντρο με **ένα μόνο κόμβο**.
- Η λειτουργία **FIND-SET** μπορεί να υλοποιηθεί ακολουθώντας τους δείκτες προς τους γονείς μέχρι να βρούμε τη **ρίζα του δέντρου**. Οι κόμβοι που επισκέπτονται κατά την πορεία προς τη ρίζα συνιστούν το **μονοπάτι αναζήτησης (find-path)**.
- Η λειτουργία **UNION** μπορεί να υλοποιηθεί κάνοντάς τον **γονέα της ρίζας ενός δέντρου να δείχνει στη ρίζα του άλλου δέντρου**.



Υλοποίηση σε C

- Η δομή δεδομένων **disjoint-forests** μπορεί να υλοποιηθεί εύκολα αλλάζοντας το νόημα των στοιχείων του πίνακα **id** στην προηγούμενη υλοποίησή μας σε C. Τώρα, κάθε **id[i]** αντιπροσωπεύει το στοιχείο **i** ενός συνόλου και δείχνει σε κάποιο άλλο στοιχείο του ίδιου συνόλου. Αυτοί οι δείκτες δίνουν τα μονοπάτια προς τη ρίζα του δέντρου. Το στοιχείο ρίζας δείχνει στον εαυτό του.
- Το πρόγραμμα στην επόμενη διαφάνεια δείχνει αυτή τη λειτουργικότητα. Σημειώστε ότι αφού βρούμε τις ρίζες των δύο συνόλων, η λειτουργία **UNION** υλοποιείται απλά με την εντολή **id[i] = j**. Με άλλα λόγια, το στοιχείο **j** γίνεται ο αντιπρόσωπος των ενωμένων συνόλων και η ρίζα του νέου δέντρου.
- Αυτός ο αλγόριθμος ονομάζεται **quick-union**.
- Η υλοποίηση της λειτουργίας **FIND-SET(i)** είναι παρόμοια: απλώς ακολουθούμε τους δείκτες ξεκινώντας από **id[i]** μέχρι να βρούμε τη ρίζα του δέντρου.

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.

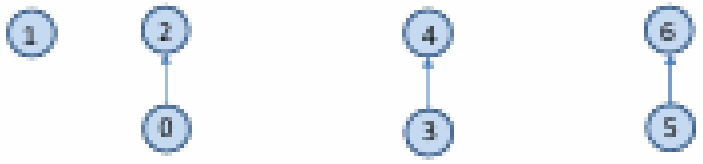
Example (cont'd)

After processing the pair (0,2).

Example (cont'd)

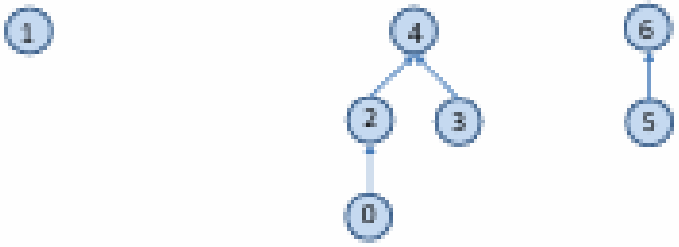
After processing the pair (5,6).

Example (cont'd)



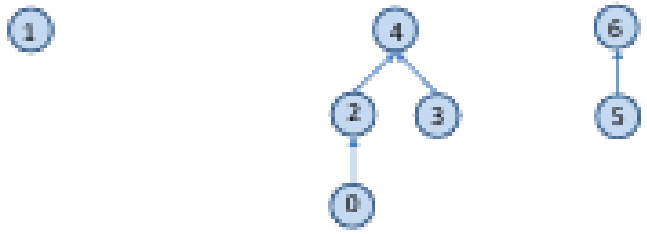
After processing the pair (3,4).

Example (cont'd)



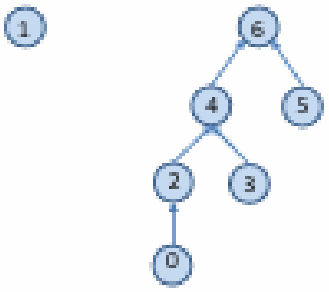
After processing the pair (0,4).

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).

Συζήτηση

- Ο αλγόριθμος **quick-union** φαίνεται να είναι πιο γρήγορος από τον αλγόριθμο **quick-find**, γιατί δεν χρειάζεται να περάσει από όλο τον πίνακα για κάθε ζεύγος εισόδου. Αλλά πόσο πιο γρήγορος είναι;
- Μέσω εμπειρικών μελετών ή μελετών της αμυντικής πολυπλοκότητας του χρόνου εκτέλεσης των δύο αλγορίθμων, μπορούμε να δείξουμε ότι ο **quick-union** είναι πιο αποδοτικός.
- Ωστόσο, δεν μπορούμε να εγγυηθούμε ότι ο **quick-union** θα είναι σημαντικά ταχύτερος από τον **quick-find** στο χειρότερο σενάριο, γιατί τα δεδομένα εισόδου μπορεί να συνεργαστούν με τρόπο που να κάνει τη λειτουργία **FIND-SET** αργή.

Πρόταση

- Για $m > n$, ο αλγόριθμος **quick-union** μπορεί να απαιτεί περισσότερο από mn οδηγίες για να λύσει ένα πρόβλημα συνόλων διαχωρισμένων με m ζεύγη από n αντικείμενα.

Απόδειξη

- Υποθέστε ότι τα ζεύγη εισόδου έρχονται με την εξής σειρά: (1,2), (2,3), (3,4), και ούτω καθεξής.
- Μετά από $n - 1$ τέτοια ζεύγη, έχουμε n αντικείμενα όλα στο ίδιο σύνολο, και το δέντρο που σχηματίζεται από τον αλγόριθμο **quick-union** είναι μια αλυσίδα, όπου το n δείχνει στο $n - 1$, που δείχνει στο $n - 2$, και ούτω καθεξής.
- Για να εκτελέσουμε την λειτουργία **FIND** για το αντικείμενο n, το πρόγραμμα πρέπει να ακολουθήσει $n - 1$ δείκτες.
- Έτσι, ο μέσος αριθμός δεικτών που ακολουθούνται για τα πρώτα $n - 1$ ζεύγη είναι:

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

Απόδειξη (συνέχεια)

- Τώρα υποθέστε ότι τα υπόλοιπα ζεύγη συνδέουν το n με κάποιο άλλο αντικείμενο.
- Η λειτουργία **FIND-SET** για κάθε από αυτά τα ζεύγη απαιτεί τουλάχιστον $(n - 1)$ δείκτες.
- Ο συνολικός αριθμός των **FIND-SET** λειτουργιών για αυτή τη σειρά εισόδων είναι σίγουρα μεγαλύτερος από $mn/2$.

Ο Αλγόριθμος Weighted Quick-Union

- Μπορούμε να υλοποιήσουμε μια "ζυγισμένη" έκδοση της λειτουργίας **UNION** παρακολουθώντας το μέγεθος των δύο δέντρων και κάνοντάς το ριζικό στοιχείο του μικρότερου δέντρου να δείχνει στο ριζικό στοιχείο του μεγαλύτερου δέντρου.
- Αυτό βοηθά στην αποφυγή της δημιουργίας πολύ βαθιών δέντρων, κάτι που θα καθυστερούσε τις λειτουργίες **FIND-SET**.

Για να το επιτύχουμε αυτό, χρησιμοποιούμε έναν πίνακα **sz[N]** (μέγεθος), όπου κάθε στοιχείο του πίνακα δείχνει το μέγεθος του δέντρου για το οποίο είναι ρίζα το αντίστοιχο στοιχείο.

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);
      }
  }
```

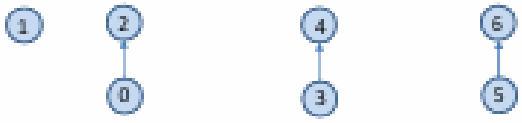
Example

Example (cont'd)

Example (cont'd)

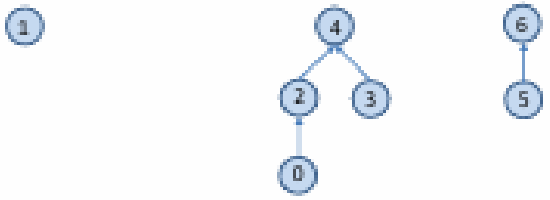
After processing the pair [5,6].

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.

Η Ευρετικη Συμπίεσης Μονοπατιού

Η **ευρετική συμπίεσης μονοπατιού** είναι μια βελτίωση που εφαρμόζεται στην επιχείρηση **FIND-SET** στην δομή δεδομένων των αποσυνδεδεμένων συνόλων. Βοηθά στο να γίνει πιο επίπεδη η δομή του δέντρου, διασφαλίζοντας ότι οι μελλοντικές επιχειρήσεις θα είναι ταχύτερες μειώνοντας το ύψος των δέντρων.

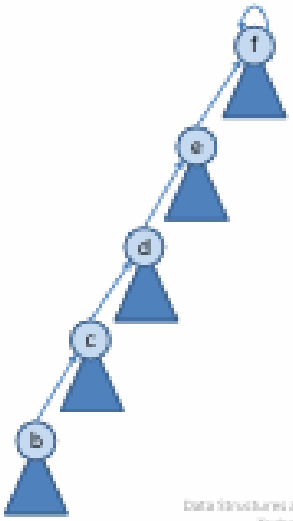
Τι κάνει η συμπίεση μονοπατιού;

- Κατά τη διάρκεια μιας επιχείρησης **FIND-SET**, αντί να βρούμε απλώς την ρίζα του δέντρου, κάνουμε κάθε κόμβο στη "διαδρομή εύρεσης" (την διαδρομή που ακολουθείται για να φτάσουμε στη ρίζα) να δείχνει απευθείας στη ρίζα. Αυτό ισοδυναμεί με την "επίπεδη" διάταξη του δέντρου και βελτιώνει την απόδοση μελλοντικών επιχειρήσεων **FIND-SET**.
- **Επίδραση**: Με την πάροδο του χρόνου, τα δέντρα γίνονται πιο επίπεδα, και η επιχείρηση **FIND-SET** απαιτεί λιγότερα βήματα, μειώνοντας τη χρονική πολυπλοκότητα σημαντικά.

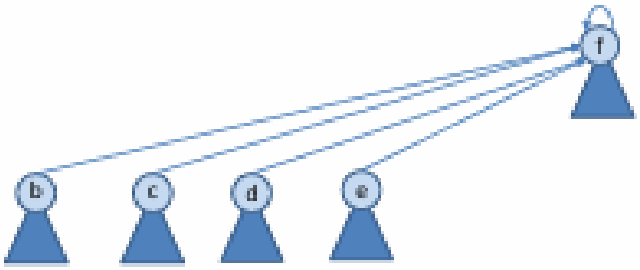
Γιατί είναι αποτελεσματικό;

- Η συμπίεση μονοπατιού βοηθά στο να κρατηθούν τα δέντρα πολύ επίπεδα, διότι συνδέει άμεσα όλους τους κόμβους ενός συνόλου με τη ρίζα, μειώνοντας την ανάγκη να διασχίσουμε πολλές επίπεδες του δέντρου για μελλοντικές επιχειρήσεις.
- **Δεν αλλάζει τα μεγέθη των δέντρων** ή επηρεάζει άμεσα την επιχείρηση ένωσης, οπότε συνεργάζεται άψογα με την **ευρετική βαρύτητας σύνδεσης** (weighted quick-union).

The Path Compression Heuristic Graphically



The Path Compression Heuristic Graphically (cont'd)



Η Ευρετική Quick-Union με Συμπίεση Μονοπατιού

Συνδυάζοντας τον αλγόριθμο **weighted quick-union** με τη **συμπίεση μονοπατιού**, αποκτούμε μια πολύ αποτελεσματική δομή δεδομένων όπου:

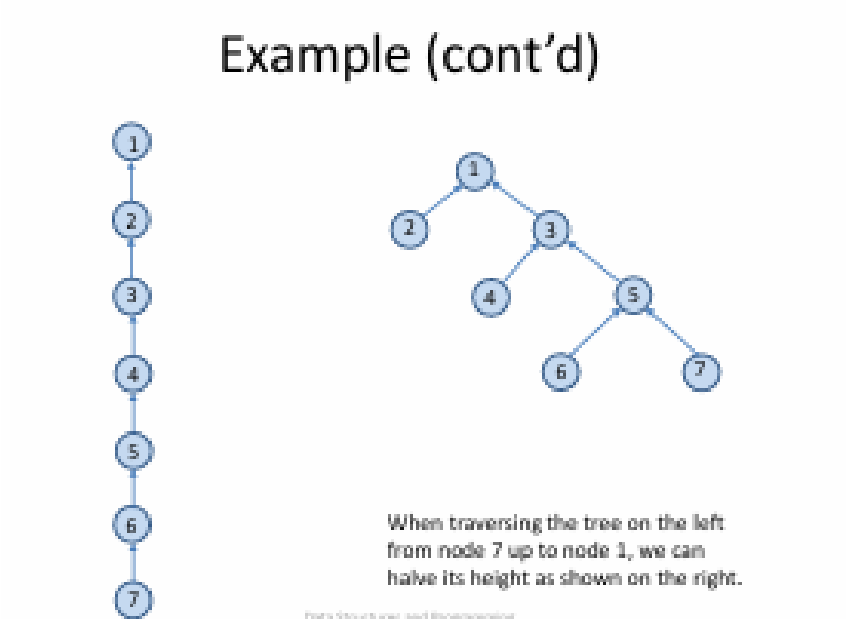
1. Οι **επιχειρήσεις ένωσης** γίνονται συνδέοντας το μικρότερο δέντρο με τη ρίζα του μεγαλύτερου (weighted union).
2. Οι **επιχειρήσεις εύρεσης** βελτιστοποιούνται κάνοντάς την διαδρομή προς τη ρίζα όσο το δυνατόν πιο σύντομη (path compression).

Θεωρητική Πολυπλοκότητα με Συμπίεση Μονοπατιού

- Η **χρονική πολυπλοκότητα** της επιχείρησης **FIND-SET** με τη **weighted quick-union** και τη **συμπίεση μονοπατιού** είναι σχεδόν σταθερή, συγκεκριμένα **O(α(n))**, όπου **α(n)** είναι η αντίστροφη συνάρτηση του Ackermann, η οποία αυξάνεται εξαιρετικά αργά (πιο αργά από την λογαριθμική ανάπτυξη).
- Ως αποτέλεσμα, για πρακτικούς σκοπούς, αυτό σημαίνει ότι ακόμα και για πολύ μεγάλα σύνολα δεδομένων, ο αλγόριθμος θα εκτελείται σχεδόν σε σταθερό χρόνο.

Πρόταση για Weighted Quick-Union με Συμπίεση Μονοπατιού

- Ο αλγόριθμος **weighted quick-union** με **συμπίεση μονοπατιού** εγγυάται ότι ο χρόνος για να επεξεργαστούν m ζεύγη από n αντικείμενα είναι $O(m\alpha(n))$, γεγονός που καθιστά τον αλγόριθμο εξαιρετικά αποτελεσματικό ακόμα και για μεγάλες εισόδους.



Η **Συμπίεση Μονοπατιού με Διχοτόμηση** (Path Compression by Halving) μπορεί να υλοποιηθεί εύκολα αντικαθιστώντας τα **βρόγχους** (for loops) του προγράμματος **weighted quick-union** με τον τρόπο που θα δείξουμε στην επόμενη διαφάνεια.

```
#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);
      }
 }
```