Στάθης Ζάχος Άρης Παγουρτζής

# Συναρτήσεις κατακερματισμού (hashing) και δομές λεξικού (dictionary)

# Δομές ένωσης-εύρεσης (union-find)

(Σημειώσεις)


Εθνικό Μετσόβιο Πολυτεχνείο

Αθήνα 2019

# Κεφάλαιο 1

# Κατακερματισμός (hashing) και δομή λεξικού (dictionary)

Ας υποθέσουμε ότι έχουμε ένα σύνολο στοιχείων $K$ και θέλουμε να μπορούμε να καταγράφουμε ποια στοιχεία από το $K$ εμφανίζονται σε κάποια δεδομένα εισόδου. Για παράδειγμα, θέλουμε να καταγράψουμε και οργανώσουμε τις λέξεις που εμφανίζονται σε ένα κείμενο, ώστε να μπορούμε να απαντάμε γρήγορα αν κάποια λέξη υπάρχει στο κείμενο ή όχι. Χρειαζόμαστε επομένως μια δομή δεδομένων που να υποστηρίζει αποδοτικά τις λειτουργίες της εισαγωγής στοιχείου (insert) και της εύρεσης στοιχείου (find). Συνήθως (αλλά όχι πάντα) θέλουμε να υποστηρίζεται και η διαδικασία διαγραφής στοιχείου (delete), για παράδειγμα αν κάποια λέξη στο κείμενο που επεξεργαζόμαστε ήταν ορθογραφικά λανθασμένη. Ας σημειωθεί ότι αν οι διαγραφές είναι σπάνιες (όπως συμβαίνει σε πολλές εφαρμογές), η αποδοτικότητα της λειτουργίας αυτής δεν είναι τόσο κρίσιμη.

**Ορισμός 1.1.** Μια αφηρημένη δομή δεδομένων (ADT) που υποστηρίζει αποδοτικά μία ακολουθία από διαδικασίες Insert, Delete και Find ονομάζεται **λεξικό** (*dictionary*).

Αναφερόμαστε στο σύνολο $K$ ως **σύνολο κλειδιών**, καθώς συχνά αντί να αποθηκεύουμε το ίδιο το στοιχείο, προτιμούμε να χρησιμοποιούμε ένα αναγνωριστικό ή *κλειδί* που προσδιορίζει μοναδικά το στοιχείο.

Η υλοποίηση ενός λεξικού μπορεί να γίνει π.χ. με μια συνάρτηση

$$h\colon K \to \{0, \ldots, m-1\},$$

που ονομάζεται **συνάρτηση κατακερματισμού** (*hash function*). Συχνά χρησιμοποιείται και ο όρος **συνάρτηση σύνοψης** (*digest*). Φροντίζουμε ώστε η συνάρτηση $h$ που διαλέγουμε να υπολογίζεται γρήγορα για οποιοδήποτε στοιχείο του συνόλου $K$, σε χρόνο δηλαδή $O(1)$.

## 1.1 Μέθοδος διαίρεσης και αλυσίδωση

Τυπικό παράδειγμα συνάρτησης κατακερματισμού είναι η συνάρτηση mod (όπου υποθέτουμε ότι $K \subseteq \mathbb{N}$):

$$\forall k \in K : \quad h(k) = k \bmod m \tag{1.1}$$

Ας σημειωθεί ότι η συνάρτηση αυτή κατακερματίζει ομοιόμορφα το σύνολο $K$: αν $K = \{1, \ldots, N\}$ τότε σε κάθε τιμή του πεδίου τιμών απεικονίζονται περίπου $\frac{N}{m} = \frac{|K|}{m}$ στοιχεία του $K$.
Άλλα παραδείγματα είναι:

(i) η πιο γενική μορφή της μεθόδου διαίρεσης, όπου επιλέγονται 2 αριθμοί $a, b < m$ με $\gcd(a, m) = 1$ και η συνάρτηση $h$ ορίζεται ως εξής:

$$h(k) = a \cdot k + b \bmod m$$

(η επιλογή του $a$ να είναι σχετικά πρώτος με τον $m$ είναι σημαντική ώστε να η συνάρτηση να κατακερματίζει "ομοιόμορφα" το σύνολο $K$: δύο κλειδιά $k_1, k_2$ έχουν την ίδια εικόνα ανν έχουν το ίδιο υπόλοιπο $\bmod m$, όπως και στην απλή μορφή της μεθόδου)

(ii) η μέθοδος του πολλαπλασιασμού όπου χρησιμοποιείται το μήκος λέξης του υπολογιστή, έστω $w$, και ένας πολλαπλασιαστής $a < 2^w$ ($a$ περιττός αριθμός) και η συνάρτηση ορίζεται ως εξής:

$$h(k) = (a \cdot k \bmod 2^w) \operatorname{div} 2^{w-r}$$

δηλαδή αρχικά κρατούνται τα $w$ λιγότερο σημαντικά bits του γινομένου (ώστε να χωρούν σε μια λέξη του υπολογιστή) και από αυτά κρατούνται τα $r$ περισσότερο σημαντικά. Η συνάρτηση αυτή θεωρείται ότι έχει καλή συμπεριφορά στην πράξη και πολύ αποδοτική υλοποίηση. Συναντιέται και στη μορφή

$$h(k) = ((a \cdot k + b) \bmod 2^w) \operatorname{div} 2^{w-r},$$

όπου $b < 2^w$. Η μορφή αυτή θεωρείται ιδιαίτερα κατάλληλη για τον ορισμό μιας οικογένειας συναρτήσεων κατακερματισμού $h_{a,b}$ που παραμετροποιείται ως προς $a$ και $b$ και έχει την ιδιότητα της *καθολικότητας*: για δύο οποιαδήποτε διαφορετικά στοιχεία του συνόλου κλειδιών $K$, μία τυχαία και ομοιόμορφα επιλεγμένη συνάρτηση της οικογένειας έχει πιθανότητα $\leq 1/m$ να τα απεικονίσει στην ίδια τιμή (όπου η πιθανότητα λαμβάνεται ως προς όλα τα πιθανά ζεύγη $(a, b)$).

**Πίνακας κατακερματισμού και μέθοδος αλυσίδωσης.** Στη συνέχεια θα αναφερόμαστε στην μορφή 1.1 (απλή μέθοδος διαίρεσης) για τα παραδείγματά μας, αλλά όσα λέμε ισχύουν και για τις άλλες μεθόδους. Για την αποθήκευση των κλειδιών

δημιουργούμε ένα πίνακα $A$ (hash table). Κάθε στοιχείο $A[i]$ του πίνακα δείχνει σε μια λίστα η οποία περιέχει εκείνα τα κλειδιά $a \in K$ για τα οποία ισχύει $h(a) = i$. Συνεπώς για να κάνουμε *Insert, Delete* και *Find* ένα στοιχείο $a$, αρκεί να ψάξουμε μόνο τη λίστα στην οποία δείχνει το στοιχείο $A[h(a)]$.

Η μέθοδος αποθήκευσης στοιχείων με ίδια τιμή κατακερματισμού σε λίστα λέγεται **κατακερματισμός με αλυσίδωση** (*hashing with chaining*) και είναι αρκετά αποδοτικός όταν το πλήθος των εισαγόμενων στοιχείων $n$ είναι ανάλογο του $m$, καθώς τότε το αναμενόμενο πλήθος κλειδιών ανα θέση του πίνακα είναι $O(1)$.Συχνά όμως ισχύει $n >> m$ και στη χειρότερη περίπτωση, είναι πιθανόν μετά από $n$ εισαγωγές στοιχείων, να έχουμε μια λίστα μήκους σχεδόν $n$ (δηλαδή σχεδόν όλα τα στοιχεία να έχουν πάει στην ίδια θέση του πίνακα). Σε αυτή την περίπτωση, αν χρειαστεί να εκτελέσουμε $n$ φορές τις διαδικασίες *Delete* ή *Find*, ο χρόνος που απαιτείται είναι $O(n^2)$. Αν όμως φροντίσουμε ώστε η εκλογή της συνάρτησης $h$ να εξασφαλίζει μια όσο το δυνατό ομοιόμορφη κατανομή των στοιχείων στις λίστες, έτσι ώστε να μην υπάρχει συσσώρευση στοιχείων σε μια λίστα, τότε ο χρόνος αναζήτησης μπορεί να καλυτερεύσει σημαντικά.

Έστω ότι εισάγονται στον πίνακα $n$ στοιχεία. Ονομάζουμε **παράγοντα φόρτου** (*load factor*) την τιμή $\alpha = \frac{n}{m}$, που εκφράζει το αναμενόμενο (μέσο) μήκος λίστας αν υποθέσουμε ότι τα στοιχεία κατανέμονται ομοιόμορφα στις θέσεις του πίνακα. Τη στιγμή που εισάγεται το $i$-οστό στοιχείο, η λίστα στην οποία θα μπει θα έχει αναμενόμενο μήκος $\frac{i-1}{m} < \alpha$ · αυτό είναι και το πλήθος δοκιμών που θα χρειαστούν τόσο για την εισαγωγή όσο και για την εύρεση του στοιχείου αυτού. [1] Συνεπώς το αναμενόμενο πλήθος δοκιμών που θα κάνει η Find φράσσεται από την τιμή $1+\alpha$ (συνυπολογίζουμε και την τελευταία ανεπιτυχή δοκιμή, στην περίπτωση που το στοιχείο δεν υπάρχει). Με πιο προσεκτική ανάλυση, η εύρεση όταν υπάρχει το στοιχείο κοστίζει περίπου (αναμενόμενη τιμή) $\frac{1+\alpha}{2}$. Επομένως ο αναμενόμενος χρόνος για τις λειτουργίες εισαγωγής, εύρεσης και διαγραφής είναι $O(\alpha)$, και έτσι $n$ διαδικασίες θα χρειάζονται $O(n\alpha)$ χρόνο.

**Ανακατακερματισμός (Rehashing)** Είναι συνηθισμένο να μην γνωρίζουμε από πριν τον πληθικό αριθμό που μπορεί να έχει το σύνολό μας. Στην περίπτωση αυτή διαλέγουμε μια τιμή $m$ για τον πίνακα (*h*ash table, bucket table) και όταν ο αριθμός των στοιχείων γίνει μεγαλύτερος από $m$, δημιουργούμε ένα καινούργιο πίνακα-στήλη μεγέθους $2m$ και με rehashing (δηλαδή ορίζοντας μια νέα συνάρτηση με πεδίο τιμών αυτή τη φορά το $[0, 2m-1]$), βάζουμε τα στοιχεία στον καινούργιο πίνακα, καταστρέφοντας τον παλιό. Όταν τα στοιχεία γίνουν περισσότερα από $2m$, δημιουργούμε ένα άλλο πίνακα μεγέθους $4m$ κ.ο.κ. Είναι σαφές ότι κάθε φορά η κατάλληλη επιλογή της συνάρτησης κατακερματισμού παίζει σπουδαίο ρόλο προκειμένου να διατηρήσουμε τους χρόνους προσπέλασης μικρούς.

*Παράδειγμα* 1.2. Έστω ότι το σύνολό μας αποτελείται από ακεραίους που μπορούν

[1] Η εισαγωγή μπορεί να γίνει και πιο γρήγορα, σε 1 βήμα, αν βάζουμε κάθε νέο στοιχείο στην αρχή της λίστας. Αυτό φυσικά αυξάνει το κόστος εύρεσης των στοιχείων που είναι ήδη στη λίστα και έτσι ο μέσος χρόνος εύρεσης παραμένει ίδιος.

να πάρουν τιμές στο διάστημα $[0, r], r > n$. Τότε αν χρησιμοποιήσουμε τη συνάρτηση $h(a) = a \bmod m$, όπου $m$ το μέγεθος του τρέχοντα πίνακα-στήλη έχουμε τα παρακάτω: Έστω ότι εισάγουμε τους αριθμούς $1, 5, 8, 3, 9, 6$. Αρχικά επιλέγουμε $m = 2$ και έχουμε :

$$\begin{array}{ll} 0: & \\ 1: & 1, 5 \end{array}$$

Σε αυτό το σημείο επιλέγουμε $m = 4$:

$$\begin{array}{ll} 0: & 8 \\ 1: & 1, 5 \\ 2: & \\ 3: & 3 \end{array}$$

Τέλος με $m = 8$ προκύπτει το παρακάτω:

$$\begin{array}{ll} 0: & 8 \\ 1: & 1, 9 \\ 2: & \\ 3: & 3 \\ 4: & \\ 5: & 5 \\ 6: & 6 \\ 7: & \end{array}$$

## 1.2 Ανοιχτή διευθυνσιοδότηση (open addressing)

Η μέθοδος αυτή χρησιμοποιείται όταν το πλήθος των εισαγωγών $n$ είναι μικρότερο ή ίσο των διαθέσιμων θέσεων $m$. Τότε μπορεί να αποφευχθεί η χρήση λίστας με τον εξής τρόπο:

Φυλάσσεται το πολύ ένα στοιχείο σε κάθε θέση του πίνακα. Αν η θέση $A[h(a)]$ όπου πρέπει να αποθηκευθεί το στοιχείο $a$ είναι κατειλημμένη, τότε διερευνάται αν η επόμενη θέση είναι ελεύθερη, κ.ο.κ. μέχρις ότου βρεθεί ελεύθερη θέση. Για την εύρεση (Find) και τη διαγραφή, ελέγχεται πρώτα η θέση $A[h(a)]$ και αν είναι κατειλημμένη από στοιχείο διαφορετικό του $a$ τότε διερευνάται η επόμενη θέση, κ.ο.κ. μέχρις ότου είτε βρεθεί το στοιχείο είτε βρεθεί κενή θέση χωρίς να έχει βρεθεί το στοιχείο, το οποίο σημαίνει ότι το στοιχείο δεν υπάρχει στο λεξικό (αλλιώς θα έπρεπε να είχε μπει στην κενή θέση).

Η παραπάνω μέθοδος λέγεται **γραμμική διερεύνηση** (*linear probing*) αλλά δεν λειτουργεί καλά όταν $n$ είναι συγκρίσιμο με το $m$ λόγω του ότι δημιουργεί μεγάλες συστοιχίες συνεχόμενων θέσεων οι οποίες αυξάνουν την πιθανότητα κατάληψης της επόμενης ελεύθερης θέσης κ.λπ. Η μέθοδος περιγράφεται τυπικά μέσω μιας *συνάρτησης διερεύνησης θέσης* με ορίσματα το κλειδί $a$ και τον αύξοντα αριθμό

βήματος $i = 0, 1, \ldots$, που καθορίζει σε κάθε βήμα $i$ ποια θέση του πίνακα θα διερευνηθεί:

$$pos_l(a, i) = h(a) + i \bmod m, \;\; i = 0, 1, \ldots$$

Μια καλύτερη μέθοδος είναι η **τετραγωνική διερεύνηση** (*quadratic probing*) όπου αν η θέση $A[h(a)]$ είναι κατειλημμένη, διερευνάται η εξής σειρά θέσεων:

$$pos_q(a, i) = h(a) + c \cdot i + c' \cdot i^2 \bmod m, \;\; i = 0, 1, \ldots$$

όπου $i$ είναι ο αύξων αριθμός θέσης που διερευνάται, και $c, c'$ σταθερές. Και αυτή η μέθοδος μπορεί να δημιουργήσει συστοιχίες, τις λεγόμενες *δευτερεύουσες συστοιχίες*.

Μια ακόμη καλύτερη μέθοδος, που πρακτικά παράγει μια αρκετά τυχαία σειρά θέσεων, καθορίζει την απόσταση διερεύνησης σαν συνάρτηση του κλειδιού $a$, αποφεύγοντας τη δημιουργία συστοιχιών. Η μέθοδος λέγεται **διπλός κατακερματισμός** (*double hashing*) και χρησιμοποιεί δύο διαφορετικές συναρτήσεις κατακερματισμού για τον καθορισμό της θέσης διερεύνησης:

$$pos_{dh}(a, i) = h_1(a) + i \cdot h_2(a) \bmod m, \;\; i = 1, 2, \ldots$$

Αποδεικνύεται ότι στον διπλό κατακερματισμό, αν $\alpha = n/m < 1$ είναι ο παράγοντας φόρτου, το κόστος εισαγωγής και εύρεσης φράσσονται από την τιμή $\frac{1}{1-\alpha}$: η πιθανότητα εύρεσης κενής θέσης κατά την εισαγωγή ή την εύρεση είναι $1 - \alpha$ (θεωρώντας ότι η συνάρτηση θέσης επιλέγει θέσεις σχεδόν τυχαία και ανεξάρτητα από τις προηγούμενες δοκιμές) και επομένως ο αναμενόμενος αριθμός δοκιμών είναι $1/(1-\alpha)$. Μια πιο προσεκτική ανάλυση δείχνει ότι ο χρόνος εύρεσης στοιχείου που υπάρχει στον πίνακα είναι πολύ καλύτερος: $\frac{1}{\alpha} + \ln(\frac{1}{1-\alpha})$ (η ανάλυση ξεφεύγει από τους σκοπούς αυτών των σημειώσεων).

Τέλος, μια αρκετά διαφορετική μέθοδος που λέγεται **κατακερματισμός κούκου** (*cuckoo hashing*) και προτάθηκε πριν 20 έτη περίπου από τους Pagh και Rodler [3] έχει αποδειχθεί ιδιαίτερα αποτελεσματική καθώς εξασφαλίζει *σταθερό χρόνο εύρεσης* (2 δοκιμές μόνο!). Και σε αυτή τη μέθοδο χρησιμοποιούνται δύο διαφορετικές συναρτήσεις κατακερματισμού $h_1, h_2$ αλλά με τελείως διαφορετικό τρόπο απ'ότι στον διπλό κατακερματισμό. Συγκεκριμένα, δημιουργούνται δύο πίνακες $m$ θέσεων. Κάθε στοιχείο μπαίνει στην θέση $h_1(a)$ του πρώτου πίνακα και αν αυτή είναι κατειλημμένη "διώχνει" το στοιχείο $a'$ που βρίσκεται εκεί[2], το οποίο μπαίνει στη θέση $h_2(a')$ του δεύτερου πίνακα, όπου αν υπάρχει στοιχείο $a''$ "διώχνεται" και επαναλαμβάνεται η παραπάνω διαδικασία για το $a''$, κ.ο.κ. μέχρις ότου κάποιο στοιχείο να μπει σε ελεύθερη θέση ή να δημιουργηθεί κύκλος. Στην τελευταία περίπτωση πρέπει να γίνει rehashing, επιλέγοντας νέες συναρτήσεις κατακερματισμού.

[2] Ο κούκος βγάζει τα αυγά άλλων πουλιών από τη φωλιά τους για να βάλει τα δικά του, εξ ού και το όνομα της μεθόδου.

Στην μέθοδο αυτή η εισαγωγή μπορεί να κοστίζει χρονικά λίγο περισσότερο, όμως η εύρεση (και η διαγραφή επομένως) γίνεται σε 2 το πολύ δοκιμές: το στοιχείο $a$, αν βρίσκεται στην δομή θα βρίσκεται είτε στη θέση $h_1(a)$ του πρώτου πίνακα είτε στη θέση $h_2(a)$ του δεύτερου.

*Σημείωση*: σε όλες τις μεθόδους ανοιχτής διευθυνσιοδότησης, αν ο πίνακας γεμίσει πρέπει να αυξηθεί το μέγεθος (π.χ. να διπλασιαστεί το $m$) και να γίνει ανακατακερματισμός (rehashing).

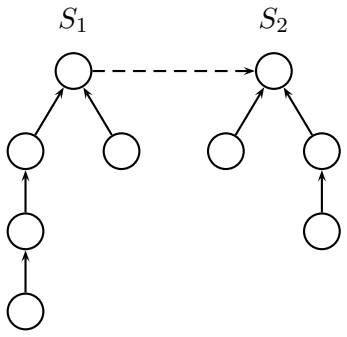
# Κεφάλαιο 2

# Δομές Ένωσης–Εύρεσης (Union–Find)

**Ορισμός 2.1.** Έστω ότι έχουμε διάφορα ξένα μεταξύ τους σύνολα και μας ενδιαφέρει η αποδοτική υλοποίηση μιας **ακολουθίας** από διαδικασίες ένωσης (*Union*) συνόλων και εύρεσης του συνόλου στο οποίο ανήκει κάποιο στοιχείο (*Find*). Μια τέτοια δομή δεδομένων ονομάζεται δομή Union-Find.

Η αναπαράσταση των συνόλων μπορεί να γίνει π.χ. με δέντρα, όπου κάθε κόμβος περιέχει ένα στοιχείο και έχει ένα pointer προς τον γονέα του. Κατά σύμβαση το όνομα του συνόλου είναι το στοιχείο που τυχαίνει να βρίσκεται στη ρίζα.

Η array Parent[i], μας δίνει τον γονέα του κόμβου $i$ και θέτουμε Parent[root]:=0. Για να βρούμε την ένωση δύο συνόλων $S_1, S_2$ αρκεί να μεταβάλλουμε την εγγραφή της Parent σε μια απ' τις δύο ρίζες των $S_1, S_2$, έτσι ώστε αντί να δείχνει 0, να δείχνει στη ρίζα του άλλου δέντρου (σχήμα 2.1). Η υλοποίηση της συνάρτησης Union φαίνεται στον αλγόριθμο 2.1.

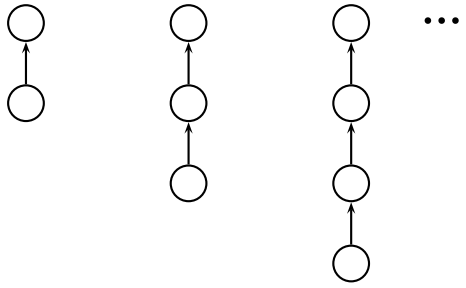



Σχήμα 2.1: Η ένωση των συνόλων $S_1$ και $S_2$

Η εκτέλεση της συνάρτησης Union γίνεται σε σταθερό χρόνο $O(1)$, ενώ για την εύρεση του συνόλου S στο οποίο ανήκει το στοιχείο i, αρκεί να βρούμε τη ρίζα του δέντρου που αναπαριστά το σύνολο S. Η υλοποίηση της συνάρτησης Find φαίνεται

στον αλγόριθμο 2.2.

Στη χειρότερη περίπτωση, ένα εκφυλισμένο (*degenarate*) δέντρο μπορεί να προκύψει από την ένωση πολλών συνόλων όπως στο σχήμα 2.2. Έτσι λοιπόν ο χρόνος που χρειάζεται η Find για να διανύσει ένα μονοπάτι του συνόλου-δέντρου με $n$-στοιχεία-κορυφές είναι στην χειρότερη περίπτωση $O(n)$. Συνεπώς $n$ εκτελέσεις της Find χρειάζονται χρόνο $O(n^2)$.



Σχήμα 2.2: Εκφυλισμένο δένδρο μετά από την ένωση πολλών συνόλων

Μπορούμε να βελτιώσουμε αυτό το χρόνο αν λάβουμε υπόψη μας τον αριθμό των στοιχείων που έχει κάθε σύνολο και συνδέουμε κάθε φορά το σύνολο-δέντρο με τα λιγότερα στοιχεία-κόμβους σε εκείνο με τα περισσότερα, αλλάζοντας τη συνάρτηση Union (Balancing). [1]

Θέτουμε την πληροφορία του πληθικού αριθμού κάθε συνόλου σαν τιμή του γονέα της ρίζας του ($Parent[root] := -\#elements$). Ο αρνητικός αριθμός ξεχωρίζει την τιμή του γονέα της ρίζας από τα υπόλοιπα στοιχεία του συνόλου. Η νέα υλοποίηση της συνάρτησης Union φαίνεται στον αλγόριθμο 2.3.

**Λήμμα 2.2.** *Ένα δέντρο που κατασκευάστηκε με τη βοήθεια του αλγόριθμου 2.3 έχει ύψος μικρότερο από* $\lfloor \log n \rfloor + 1$.

*Απόδειξη.* Επαγωγική βάση: $n = 1$. Προφανές.

Επαγωγικό βήμα: Έστω αληθές για όλα τα δέντρα με αριθμό κόμβων $\leq n-1$. Έστω ότι η τελευταία εφαρμογή της διαδικασίας ήταν $union(k, j)$ και ότι το δέντρο $j$ είχε $m$ κόμβους. Χωρίς βλάβη της γενικότητας $1 \leq m \leq \frac{n}{2}$.

[1] Παρόμοιο αποτέλεσμα επιτυγχάνεται αν αντί για το πλήθος των στοιχείων συγκρίνεται κατά την ένωση το ύψος των δένδρων και "κρεμάμε" το δένδρο μικρότερου ύψους σε αυτό με το μεγαλύτερο ύψος. Τότε κάνουμε λόγο για Union by Rank.

**Αλγόριθμος 2.1** Διαδικασία ένωσης (Union)

**function** Union ($i, j$: **integer** ($*$set$*$)): **integer** ($*$set$*$);
**begin**
  Parent[$i$]:=$j$; **return** $j$
**end**

**Αλγόριθμος 2.2** Διαδικασία εύρεσης (Find)

```
function Find (i: integer (*element*)): integer (*set*);
begin
  while Parent[i]>0 do i := Parent[i];
  return i
end
```

**Αλγόριθμος 2.3** Βελτιωμένη διαδικασία ένωσης (Balancing)

```
function Union (i, j: integer (*set*)): integer (*set*);
var x: integer; (*αρνητικός αριθμός στοιχείων του νέου συνόλου*)
begin
  x:=Parent[i]+Parent[j];
  if |Parent[i]| < |Parent[j]| then
    begin Parent[i]:=j; Parent[j]:=x; return j end
  else begin Parent[j]:=i; Parent[i]:=x; return i end
end
```

Περίπτωση 1: Νέο ύψος = ύψος του δέντρου $k$:

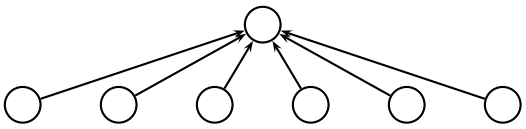
$$\text{ύψος} \leq \lfloor \log(n-m) \rfloor + 1 \leq \lfloor \log n \rfloor + 1.$$

Περίπτωση 2: Νέο ύψος $=$ ύψος του δέντρου $j+1$:

$$\text{ύψος} \leq \lfloor \log m \rfloor + 2 \leq \lfloor \log \frac{n}{2} \rfloor + 2 \leq \lfloor \log n \rfloor + 1.$$

□

Σαν αποτέλεσμα έχουμε ότι ο χρόνος που χρειάζεται μια εκτέλεση της συνάρτησης *Find* είναι $O(\log n)$, δηλαδή $n$ εκτελέσεις χρειάζονται $O(n \log n)$ χρόνο. Μπορούμε να πετύχουμε μια ακόμη καλυτέρευση του χρόνου, αλλάζοντας αυτή τη φορά τη συνάρτηση *Find* με προσθήκη της διαδικασίας *Path Compression* : ένα προς ένα τα στοιχεία-κορυφές που συναντά ο αλγόριθμος στο δρόμο για τη ρίζα, τα «ξεκρεμά» από τη θέση τους και τα «κρεμάει» από τη ρίζα, με αποτέλεσμα το σύνολο-δέντρο να τείνει προοδευτικά να μετασχηματιστεί όπως στο σχήμα 2.3. Η νέα υλοποίηση της συνάρτησης *Find* φαίνεται στον αλγόριθμο 2.4.



Σχήμα 2.3: Path compression

Είναι προφανές ότι αυτός ο τρόπος συμφέρει όταν πρόκειται να εκτελεστούν πολύ περισσότερα του ενός *Find*. Αποδεικνύεται μάλιστα ότι $n$ εκτελέσεις της συνάρτησης *Find* χρειάζονται χρόνο $O(n\alpha(n))$, όπου $\alpha(n)$ είναι ψευδοαντίστροφη της συνάρτησης Ackermann(σχεδόν σταθερά). Η ιδέα αυτή είναι παράδειγμα **αποσβετικής** αποδοτικότητας (*amortization*). Ένα βήμα της *Path Compression* κοστίζει αλλά έτσι εξοικονομείται χρόνος για κατοπινές εφαρμογές της *Find*. Η βελτίωση αυτή οφείλεται στον Tarjan [4].

**Αλγόριθμος 2.4** Βελτιωμένη διαδικασία εύρεσης (Path compression)

```
function Find(i: integer (*element*)): integer (*set*);
var j, t: integer (*element*);
begin
  j := i;
  (*Εύρεση της ρίζας*)
  while Parent[j] > 0 do j := Parent[j];
  (*Ξεκρέμασμα των φύλλων και κρέμασμα από τη ρίζα*)
  while (i ≠ j) do
    begin t := Parent[i]; Parent[i] := j; i := t end;
  return j
end
```

# Βιβλιογραφία


[1] Thomas Cormen, Charles Leiserson, Ronald Rivest and Cliff Stein, “Introduction to Algorithms”, 3rd edition, MIT Press, 2009.

[2] C.L. Liu. Στοιχεία Διακριτών Μαθηματικών (απόδοση στα Ελληνικά: Κ. Μπους και Δ. Γραμμένος). Πανεπιστημιακές Εκδόσεις Κρήτης, 2003.

[3] Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms 51, 122–144 (2004)

[4] Tarjan, Robert Endre. ”Efficiency of a Good But Not Linear Set Union Algorithm”. Journal of the ACM. 22 (2): 215–225 (1975).