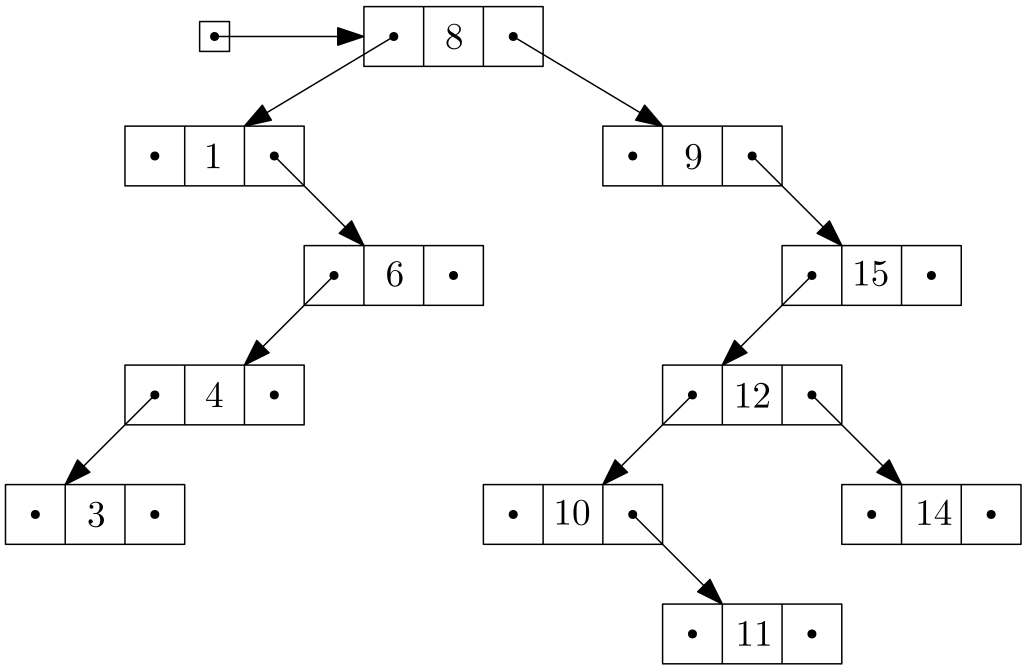
(Πρόχειρες) σημειώσεις στις ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ



Δημήτρης Ζώρος

Οι σημειώσεις αυτές γράφτηκαν για τις ανάγκες του μαθήματος Δομές Δεδομένων του τμήματος Μαθηματικών του Ε.Κ.Π.Α. και περιέχουν τα απολύτως απαραίτητα στοιχεία που θα πρέπει να γνωρίζει ένας μαθηματικός προτού ασχοληθεί σοβαρά με κάποιο μάθημα Αλγορίθμων. Η ανάγκη να γραφτούν προέκυψε από την έλλειψη κάποιου συγγράμματος που να είναι γραμμένο έτσι ώστε να μην απαιτεί από τον αναγνώστη προηγούμενες γνώσεις (φυσικά μια σχετική εξοικείωση με τους Αλγορίθμους και τον Προγραμματισμό πάντα είναι επιθυμητή), ούτε τη γνώση κάποιας συγκεκριμένης γλώσσας προγραμματισμού. Για όσους ζητούν μία αναλυτικότερη, πληρέστερη ή πιο εξειδικευμένη παρουσίαση της ύλης, παρατίθεται στο τέλος των σημειώσεων σχετική βιβλιογραφία.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΕΦΑΛΑΙΟ 0

ΕΙΣΑΓΩΓΗ

0.1 Δεδομένα και προγράμματα

Ενδιαφερόμαστε να επιλύσουμε *προβλήματα* της ακόλουθης μορφής:

- Μας δίνεται κάποια *πληροφορία*, η πληροφορία αυτή μπορεί να είναι είτε *στατική*, δηλαδή να μην αλλάξει ποτέ, είτε *δυναμική*, να αλλάζει δηλαδή με το πέρασμα του χρόνου,
- το ζητούμενο είναι η διεκπεραίωση κάποιας διαδικασίας, η εύρεση κάποιας απάντησης (Ναι ή Όχι) ή ακόμα και ο υπολογισμός της τιμής κάποιας συνάρτησης.

Ένα παράδειγμα προβλήματος περιγράφετε παρακάτω.

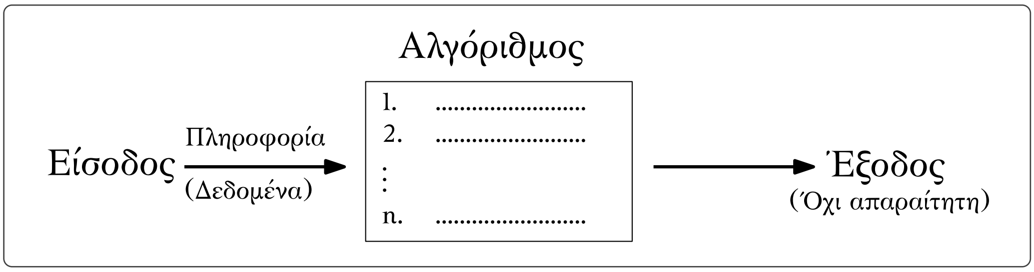
Παράδειγμα 0.1.1. Μας δίνεται ένας κατάλογος με έναν πολύ μεγάλο αριθμό ονομάτων, π.χ. ο κατάλογος με τους ενεργούς φοιτητές του τμήματος Μαθηματικών. Η πληροφορία αυτή είναι δυναμική καθώς καινούρια ονόματα προστίθενται και διαγράφονται κάθε χρόνο.

Η διαδικασία που πρέπει να διεκπεραιώσουμε είναι να ελέγξουμε αν στον κατάλογο αυτό υπάρχει ένα συγκεκριμένο όνομα. Τέτοιου είδους αναζητήσεις θα χρειαστεί να επαναληφθούν πολλές φορές σε αυτόν τον κατάλογο, για παράδειγμα η γραμματεία του τμήματος ψάχνει δεκάδες ονόματα κάθε μέρα.

Ας κάνουμε ακόμα πιο ρεαλιστικό το παράδειγμα: Ο κατάλογος αποθηκεύει όλη την πληροφορία που συνοδεύει έναν φοιτητή (πατρώνυμο, αριθμός μητρώου, έτος εισαγωγής, βαθμολογία μαθημάτων κλπ.) και κάθε φορά που η αναζήτηση μας βρίσκει κάποιο όνομα του καταλόγου η συνοδευτική πληροφορία ανακτάτε από τον κατάλογο.

Πολλά είναι τα ερωτήματα που προκύπτουν από το παραπάνω παράδειγμα, το σημαντικότερο από αυτά είναι:

> *Πως θα οργανώσουμε την πληροφορία που περιέχει ο κατάλογος έτσι ώστε η αναζήτηση αυτή να γίνεται όσο ποιο γρήγορα γίνεται;*



Σχήμα 0.1.1: *Σχηματική αναπαράσταση Αλγορίθμου.*

Προτού επιχειρήσουμε να δώσουμε απάντηση σε αυτό το ερώτημα, και να αρχίσουμε επισήμως να ερευνούμε τις *Δομές Δεδομένων*, θα πρέπει να συλλογιστούμε λίγο πως θα επιλύσουμε το παραπάνω πρόβλημα. Φυσικά προτιθέμεθα να κάνουμε τις απαραίτητες αναζητήσεις με τη χρήση ενός ηλεκτρονικού υπολογιστή, συνεπώς α) θα πρέπει η πληροφορία του καταλόγου να μεταφραστεί (πιο σωστά να *κωδικοποιηθεί*) σε κάποια συμβολική γλώσσα που μπορεί να αντιληφθεί και να επεξεργαστεί ο Η/Υ (συγκεκριμένα σε μία ακολουθία από *δυαδικά ψηφία*), και β) θα πρέπει να σχεδιάσουμε έναν *Αλγόριθμο*, να τον συντάξουμε στη γλώσσα προγραμματισμού που προτιμάμε (Python, Java, C κλπ.) και να τον τρέξουμε στον Η/Υ με *είσοδο* την κωδικοποίηση του καταλόγου. Το α) δεν θα μας απασχολήσει καθόλου στις σημειώσεις αυτές. Το β) περιέχει μία έννοια που, παρόλο που εμφανίστηκε στα μαθηματικά ήδη από την αρχαιότητα, διαφεύγει ακόμα τυπικού, αυστηρά μαθηματικού ορισμού: η έννοια του Αλγόριθμου. Πως αντιλαμβανόμαστε διαισθητικά την έννοια του Αλγορίθμου;

Ορισμός 0.1.2 (Διαισθητικός ορισμός Αλγορίθμου). *Αλγόριθμος* είναι μία πεπερασμένη ακολουθία (αυστηρά καθορισμένων) απλών οδηγιών που διεκπεραιώνουν κάποια εργασία (δες και Σχήμα 0.1.1).

Παράδειγμα 0.1.3. Ας επανέλθουμε στο προηγούμενο παράδειγμα. Θέλουμε να εξετάσουμε αν υπάρχει φοιτητής του τμήματος Μαθηματικών με το όνομα «Ευκλείδης Γεωμέτρης».Ένας αλγόριθμος που διεκπεραιώνει αυτή την αναζήτηση είναι και ο ακόλουθος:

> *Διάβασε ένα-ένα κάθε όνομα του καταλόγου ελέγχοντας αν το τρέχον όνομα ταυτίζεται με το «Ευκλείδης Γεωμέτρης». Αν ταυτίζεται επέστρεψε* **ΝΑΙ**, *καθώς και την πληροφορία που συνοδεύεται με το όνομα, και έπειτα σταμάτα. Αν διαβάσεις όλα τα ονόματα του καταλόγου και δεν βρεις το όνομα «Ευκλείδης Γεωμέτρης» επέστρεψε* **ΟΧΙ**.

Τώρα που έχουμε έναν αλγόριθμο που λύνει το πρόβλημά μας μπορούμε να εστιάσουμε την προσοχή μας στα δεδομένα-πληροφορία που έχουμε να διαχειριστούμε. Ο κατάλογος μας πρέπει να υποστηρίζει τις ακόλουθες απολύτως βασικές λειτουργίες:

1. Εισαγωγή ονόματος (δημιουργία καινούριας *καρτέλας* δηλαδή και εισαγωγή σε αυτήν των απαραίτητων στοιχείων)

2. Διαγραφή ονόματος

3. Αναζήτηση ονόματος

Ο κατάλογος αυτός ουσιαστικά είναι μία δομή δεδομένων, ένα σύνολο δηλαδή από δεδομένα-πληροφορία οργανωμένα κατάλληλα ώστε να υποστηρίζονται κάποιες απαραίτητες λειτουργίες (ή *πράξεις*). Κάθε μία από αυτές τις λειτουργίες αποτελεί ένα ξεχωριστό πρόβλημα το οποίο καλούμαστε να λύσουμε αλγοριθμικά.

Η *οργάνωση* της πληροφορίας αφορά τον τρόπο που την αποθηκεύουμε στη μνήμη του Η/Υ. Θα δούμε στη συνέχεια ότι αν οργανώσουμε την πληροφορία κατάλληλα θα μπορέσουμε να κάνουμε τις λειτουργίες που χρειαζόμαστε πολύ πιο *«αποδοτικά»*.

Κλείνοντας αυτή την παράγραφο ας καταγράψουμε τις εκκρεμότητες μας, τα πράγματα που θα πρέπει να συζητήσουμε προτού ξεκινήσουμε να μελετάμε τις Δομές Δεδομένων:

- *Αλγοριθμική γλώσσα:* Θα πρέπει να συμφωνήσουμε σε μία στοιχειώδη *«ψευδογλώσσα»* που θα περιέχει τις απολύτως απαραίτητες εντολές για να υλοποιήσουμε τις δομές δεδομένων που θα μας απασχολήσουν.

- *Καταχώρηση στη μνήμη:* Θα πρέπει να κατανοήσουμε (σε εντελώς αφηρημένο επίπεδο) τον τρόπο που καταχωρεί ο Η/Υ τα δεδομένα στη μνήμη του.

- *Αποδοτικότητα Αλγορίθμων:* Θα πρέπει να ορίσουμε ένα «μέτρο» της αποδοτικότητας των αλγορίθμων που θα σχεδιάσουμε ως προς τους *υπολογιστικούς πόρους* που έχουν μεγαλύτερη σημασία για εμάς (για τους σκοπούς αυτών των σημειώσεων θα ενδιαφερθούμε μόνο για τον χρόνο που χρειάζονται οι αλγόριθμοι για να επιστρέψουν κάποια απάντηση).

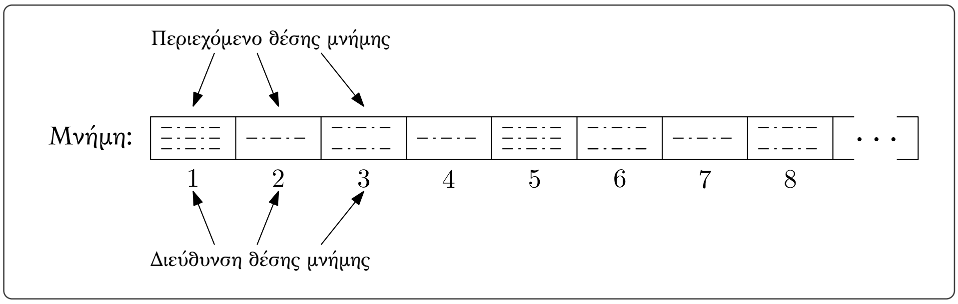
- *Κατάλληλη οργάνωση:* Αφού δούμε όλα τα παραπάνω θα δούμε πως η οργάνωση των δεδομένων μπορεί να μας βοηθήσει να διεκπεραιώσουμε τις λειτουργίες μίας δομής δεδομένων πιο αποδοτικά.

0.2 Καταχώρηση στη μνήμη

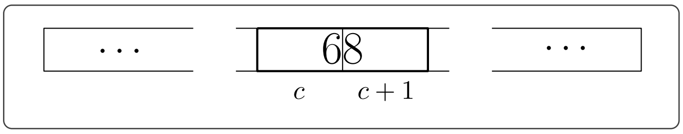
Ας ξεκινήσουμε από την εκκρεμότητα με την πιο «τεχνολογική χροιά». Σε όσα ακολουθούν θα κάνουμε μια υπεραπλούστευση της αρχιτεκτονικής των Η/Υ καθώς αυτή δεν αποτελεί (άμεσο) αντικείμενο των δομών δεδομένων.

Θεωρούμε ότι η *κύρια μνήμη* του Η/Υ αποτελείται από ένα σύνολο *θέσεων μνήμης*, κάθε μία από τις οποίες έχει μία *διεύθυνση* που παριστάνεται με ένα φυσικό αριθμό (π.χ. θέση 1871). Οι θέσεις αυτές έχουν συγκεκριμένη χωρητικότητα από δυαδικά ψηφία πληροφορίας που μπορούν να αποθηκεύσουν (π.χ. μπορούν να αποθηκεύσουν 1 byte, δηλαδή 8 δυαδικά ψηφία). Μια σχηματική αναπαράσταση βρίσκεται στο Σχήμα 0.2.1.

Οι Η/Υ χρησιμοποιούν κύρια μνήμη *τυχαίας προσπέλασης*, το λεγόμενο *μοντέλο RAM* (Random Access Memory). Ο όρος τυχαία προσπέλαση πηγάζει από το γεγονός ότι ο χρόνος που δαπανά ο Η/Υ για να *αποκτήσει πρόσβαση* σε μια θέση μνήμης (να βρει και να διαβάσει το περιεχόμενο δηλαδή) είναι ίδιος για κάθε θέση μνήμης.



Σχήμα 0.2.1: *Σχηματική αναπαράσταση της κύριας μνήμης του Η/Υ.*



Σχήμα 0.2.2: *Το Παράδειγμα 0.2.1.*

Κατά την εκτέλεση ενός προγράμματος όποτε *δηλώνουμε* κάποια μεταβλητή ο Η/Υ *δεσμεύει* κατάλληλο πλήθος θέσεων μνήμης και «θυμάται» ότι η τιμή της μεταβλητής θα βρίσκεται σε αυτές τις θέσεις μνήμης. Για να το πετύχει αυτό θυμάται τη διεύθυνση της πρώτης θέσης μνήμης και τον *τύπο* της μεταβλητής. Όταν στη συνέχεια *αναθέσουμε τιμή* στη μεταβλητή ο Η/Υ θα τοποθετήσει την τιμή στις θέσεις μνήμης που δέσμευσε για τη μεταβλητή.

Παράδειγμα 0.2.1. Θεωρήστε ότι τρέχουμε ένα πρόγραμμα στον Η/Υ μας που περιέχει την παρακάτω γραμμή κώδικα:

$$\textbf{int } x \leftarrow 68$$

η οποία δηλώνει ότι η μεταβλητή με όνομα x είναι τύπου **int** (ακεραίου δηλαδή) και αναθέτει σε αυτή την τιμή 68.

Ας υποθέσουμε ότι για να αποθηκεύσουμε τον ακέραιο 68 χρειάζονται 2 θέσεις μνήμης και ότι οι πρώτες δύο διαδοχικές ελεύθερες θέσεις μνήμης έχουν διεύθυνση c και $c+1$. Ο Η/Υ δεσμεύει τις δύο αυτές θέσεις μνήμης, αποθηκεύει σε αυτές τον ακέραιο 68 και από εδώ και στο εξής (μέχρι να τερματίσει το πρόγραμμα και να *αδειάσουν* οι θέσεις μνήμης που έχει χρησιμοποιήσει), όποτε χρειάζεται την τιμή της μεταβλητής με όνομα x (είτε για να την αλλάξει είτε απλά για να ανακτήσει την τιμή της), θυμάται ότι βρίσκεται στις θέσεις μνήμης με διευθύνσεις c και $c+1$ (δες Σχήμα 0.2.2) [1].

Σύμβαση 0.2.2. Χάριν απλότητας θα θεωρήσουμε ότι η τιμή κάθε μεταβλητής, οποιουδήποτε τύπου και αν είναι (ακέραιος, πραγματικός, χαρακτήρας κλπ.), χωράει να αποθηκευτεί

[1] Στην πραγματικότητα χρειάζεται να θυμάται μόνο τη διεύθυνση της πρώτης θέσης μνήμης και τον τύπο της μεταβλητής.

σε ακριβώς μία θέση μνήμης. Συνεπώς από εδώ και στο εξής δεν θα μας ενδιαφέρει τι τύπου είναι κάθε μεταβλητή. Σε αυτό το πνεύμα η ψευδογλώσσα που θα ορίσουμε αργότερα δεν θα περιέχει *δηλώσεις τύπου*.

Κάθε φορά που στο πρόγραμμα δηλώνεται κάποια καινούρια μεταβλητή ο Η/Υ θα βρίσκει την πρώτη κενή θέση μνήμης και θα τη δεσμεύει για τη μεταβλητή αυτή (όπως κάναμε στο Παράδειγμα 0.2.1 μόνο που πλέον, λόγω της Σύμβασης 0.2.2, ο ακέραιος 68 θα χωράει σε μία θέση μνήμης).

Σε τακτά χρονικά διαστήματα ο Η/Υ αδειάζει τις θέσεις μνήμης που δεν χρειάζονται πια στα προγράμματα που τρέχει.

Παρατήρηση 0.2.3. Σωστή πρακτική θα ήταν ένα πρόγραμμα να αδειάζει τις θέσεις μνήμης των μεταβλητών που δεν χρειάζεται πια. Αυτό θα μπορούσαμε να το δηλώνουμε ρητά όταν σχεδιάζουμε το πρόγραμμα χρησιμοποιώντας μία ειδική εντολή στη ψευδογλώσσα μας (π.χ. **free** x). Δεν συντρέχει όμως λόγος να κάνουμε κάτι τέτοιο, θα θεωρήσουμε ότι το άδειασμα της μνήμης είναι δουλειά με την οποία έχει επιφορτιστεί ο Η/Υ μας.

0.2.1 Ομαδική καταχώρηση στη μνήμη

Συχνά χρειάζεται να καταχωρηθούν *ομαδικά* πολλά δεδομένα στη μνήμη καθώς αυτά συσχετίζονται με κάποιον τρόπο και θα θέλαμε οι θέσεις μνήμης που θα τους δοθούν να έχουν κάποια οργάνωση ώστε η ανάκτηση των τιμών να γίνεται πιο άμεσα. Το κλασικό παράδειγμα, με το οποίο θα ασχοληθούμε σε αυτή την παράγραφο, είναι η καταχώρηση ενός πίνακα. Ο πίνακας περιέχει πολλές τιμές, μία για κάθε θέση του, τις οποίες θα πρέπει ο Η/Υ να καταχωρήσει στη μνήμη ομαδικά και οργανωμένα.

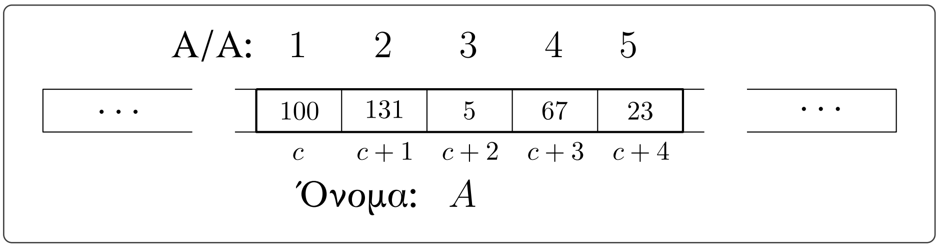
Η ομαδική καταχώρηση μπορεί να γίνει με δύο τρόπους:

Στατική καταχώρηση: Αν γνωρίζουμε εξ αρχής το ανώτερο πλήθος θέσεων που θα χρειαστούμε για την καταχώρηση της πληροφορίας (και δηλώσουμε φυσικά τη γνώση αυτή μέσα στο πρόγραμμά μας) ο Η/Υ δεσμεύσει τις πρώτες ελεύθερες διαδοχικές θέσεις μνήμης που θα βρει, αναθέτοντας τους ένα όνομα και διακρίνοντας τις θέσεις μνήμης που έχει αποθηκεύσει τις επιμέρους τιμές μέσω ενός αύξοντα αριθμού (έναν *ακέραιο δείκτη*). Ας δούμε ένα συγκεκριμένο παράδειγμα.

Παράδειγμα 0.2.4. Ας υποθέσουμε ότι θέλουμε να αποθηκεύσουμε τον μονοδιάστατο πίνακα

$$[100 \;\; 131 \;\; 5 \;\; 67 \;\; 23]$$

και σύμφωνα με τη σύμβαση μας κάθε ένας από αυτούς τους ακέραιους μπορεί να αποθηκευτεί σε μία θέση μνήμης. Ο Η/Υ θα ψάξει να βρει 5 συνεχόμενες θέσεις μνήμης, θα τους δώσει ένα όνομα (αν δεν το έχουμε κάνει ήδη μέσα στο πρόγραμμα), έστω A, και θα διακρίνει τα στοιχεία του πίνακα χρησιμοποιώντας έναν αύξοντα αριθμό (ξεκινώντας από το 1). Γνωρίζοντας ότι ο πίνακα A είναι αποθηκευμένος σε 5 θέσεις μνήμης με πρώτη αυτή που έχει διεύθυνση έστω τη c μπορούμε άμεσα να ανακτήσουμε την τρίτη τιμή του πίνακα (που θα τη συμβολίζουμε με $A[3]$) πηγαίνοντας στη διεύθυνση $c + 2$ και διαβάζοντας το περιεχόμενο αυτής της θέσης μνήμης (δες Σχήμα 0.2.3).



Σχήμα 0.2.3: *Στατική καταχώρηση ενός πίνακα.*

Αφού κατανοήσαμε πως γίνεται αυτό το είδος καταχώρησης θα πρέπει να εξηγήσουμε γιατί την ονομάζουμε *στατική*. Είδαμε ότι η βασική προϋπόθεση για τη στατική καταχώρηση είναι η a priori γνώση του «όγκου» των δεδομένων που πρέπει να καταχωρήσουμε στη μνήμη. Πέραν αυτού, αν κατά τη διάρκεια της εκτέλεσης του προγράμματός μας αυξομειωθεί ο όγκος των δεδομένων αυτών, είτε αλλάξουν κάποιες τιμές τους, δεν θα μπορούμε να τροποποιήσουμε τις θέσεις μνήμης για να καταχωρήσουμε αυτές τις αλλαγές. Θα πρέπει να κάνουμε την (ομαδική) καταχώρηση των δεδομένων εξ αρχής.

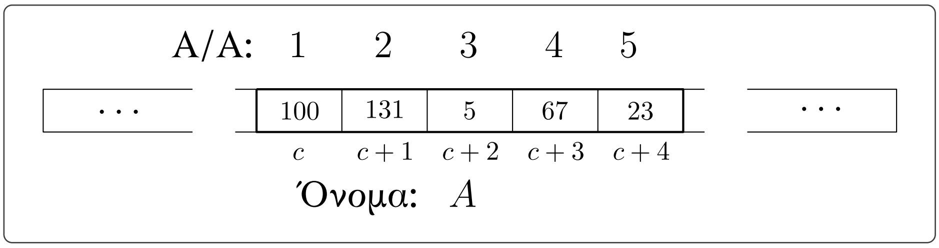
Δυναμική καταχώρηση: Στην περίπτωση όπου δεν γνωρίζουμε εξ αρχής τον όγκο των δεδομένων που θα χρειαστεί να αποθηκεύσουμε ή όπου τον γνωρίζουμε αλλά τα δεδομένα θα υποστούν αλλαγές που θα επηρεάσουν τη χωρητικότητα στη μνήμη που καταλαμβάνουν, θα χρειαστεί να δεσμεύσουμε τις θέσεις στη μνήμη *δυναμικά*. Ας δούμε πάλι ένα αντιπροσωπευτικό παράδειγμα.

Παράδειγμα 0.2.5. Ας υποθέσουμε ότι θέλουμε να αποθηκεύσουμε τη λίστα με τα ψώνια που έχουμε να κάνουμε:

1. Γάλα
2. Πορτροκάλια
3. Αυγά

όπως συμβαίνει πολύ συχνά η λίστα αυτή δέχεται συνεχώς μεταβολές. Θα θέλαμε να την αποθηκεύσουμε στη μνήμη του Η/Υ μας καταλαμβάνοντας θέσεις μνήμης με δυναμικό τρόπο. Θα μπορούσαμε όποτε δημιουργείται μία εγγραφή στη λίστα να δεσμεύουμε κατάλληλες θέσεις στη μνήμη για να την αποθηκεύσουμε (σύμφωνα με τη Σύμβαση 0.2.2 μας αρκεί μία μόνο θέση) και να τις «συνδέσουμε» με τις υπόλοιπες εγγραφές έτσι ώστε να διατηρηθεί η σειρά τους (δες Σχήμα 0.2.4).

Για να μπορέσει να γίνει αυτή η «σύνδεση» πολλές γλώσσες προγραμματισμού χρησιμοποιούν έναν ειδικό *τύπο δεδομένων* που ονομάζεται *δείκτης*. Αυτόν θα χρησιμοποιήσουμε και στη δική μας ψευδογλώσσα. Πρώτα όμως θα χρειαστεί να κάνουμε μία παρένθεση και να συζητήσουμε λίγο για τους τύπους δεδομένων που χρησιμοποιούν οι γλώσσες προγραμματισμού.



Σχήμα 0.2.4: *Δυναμική καταχώρηση μίας λίστας.*

Τύποι δεδομένων

Έχουμε ήδη αναφερδεί στον τύπο δεδομένων **int** (δες Παράδειγμα 0.2.1). Τα δεδομένα ενός προγράμματος (οι τιμές των μεταβλητές, οι σταθερές κλπ.) ανήκουν σε κάποιους τύπους που έχουν συγκεκριμένο *πεδίο τιμών* και συγκεκριμένους τρόπους επεξεργασίας (πράξεις). Ας δούμε μερικούς από τους πιο συνηδισμένους τύπους δεδομένων:

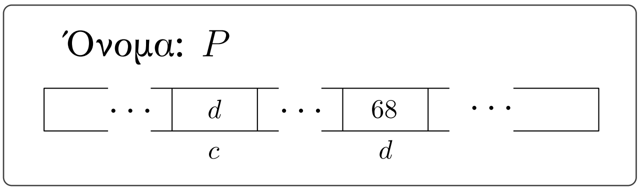
Τύπος Δεδομένων	**Πεδίο Τιμών**	**Παραδείγματα πράξεων**
Ακέραιος (int)	Οι ακέραιοι από το -maxint έως το maxint [1]	Ανάθεση τιμής ($\leftarrow$) Αριθμητικές πράξεις ($+, \cdot$, κλπ.) Συγκρίσεις ($=, <, \leq$)
Πραγματικός (float) [2]	Εξαρτάται από την ακρίβεια που ζητάμε	Ανάθεση τιμής ($\leftarrow$) Αριθμητικές πράξεις ($+, \cdot$, κλπ.) Συγκρίσεις ($=, <, \leq$)
Αληθοτιμές (bool)	$\{\mathrm{True}, \mathrm{False}\}$	Ανάθεση τιμής ($\leftarrow$) Λογικές πράξεις ($\wedge, \vee, \neg$)
Χαρακτήρας (char)	Οι ακέραιοι από 0 έως 127 [3]	Ανάθεση τιμής ($\leftarrow$)

Σε πολλές γλώσσες προγραμματισμού όταν ορίζουμε μία μεταβλητή πρέπει να δηλώνουμε ρητά τον τύπο δεδομένων της (όπως κάναμε στο Παράδειγμα 0.2.1). Ο λόγος που γίνεται αυτό είναι για να δεσμευτεί το κατάλληλό πλήθος θέσεων στη μνήμη και για να ενημερώσουμε τον Η/Υ για τις αποδεκτές πράξεις που μπορούμε να εφαρμόσουμε σε αυτή την μεταβλητή.

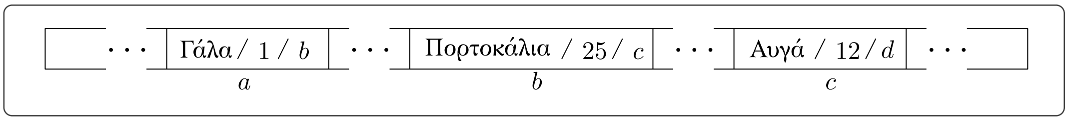
Ο τύπος δεδομένων *δείκτης* που θα χρησιμοποιήσουμε στη δική μας ψευδογλώσσα θα έχει πεδίο τιμών τις διευθύνσεις των θέσεων μνήμης. Αφού η τιμή του δείκτη θα αποθηκεύεται και αυτή με τη σειρά της στη μνήμη θα έχουμε ένα όνομα για τον δείκτη και μία διεύθυνση όπου το περιεχόμενο της θα είναι η διεύθυνση μίας άλλης θέσης μνήμης (ένας αριθμός στην ουσία που θα αντιστοιχεί στην τιμή του δείκτη). Το Σχήμα 0.2.5 θα βοηθήσει στην κατανόηση.

Για να κατασκευάσουμε την αλυσίδα που περιγράψαμε πιο πριν στη δυναμική καταχώρηση δεδομένων πέρα από τον δείκτη θα χρειαστεί να εισάγουμε και τους *σύνθετους τύπους*

[1] Όπου maxint ο μέγιστος ακέραιος αριθμός που μπορεί να χρησιμοποιήσει ο Η/Υ.
[2] Οι πραγματικοί αριθμοί αποκαλούνται *αριθμοί κινητής υποδιαστολής*.
[3] Οι αριθμοί αυτοί αντιστοιχούν σε χαρακτήρες του αλφαβήτου ASCII.



Σχήμα 0.2.5: *Η τιμή του δείκτη P που είναι αποθηκευμένος στη θέση μνήμης με διεύθυνση c ισούται με d. Στη θέση μνήμης με διεύθυνση d είναι αποθηκευμένος ο ακέραιος 68. Χαλαρά μιλώντας θα μπορούσαμε να πούμε ότι ο δείκτης P δείχνει στην τιμή 68.*



Σχήμα 0.2.6: *Παράδειγμα καταχώρησης στη μνήμη της λίστας με τα ψώνια μας.*

*δεδομένων.*Ένας σύνθετος τύπος δεδομένων αποτελείται από επιμέρους *πεδία*, σε αντίθεση με τους απλούς τύπους δεδομένων που αποτελούνται από ένα μόνο πεδίο. Τα πεδία αυτά σχετίζονται μεταξύ τους και ακολουθούν κάποια συγκεκριμένη οργάνωση (αυτή η οργάνωση δεν θα μας απασχολήσει ιδιαίτερα).

Παράδειγμα 0.2.6. Ας επανέλθουμε στο παράδειγμα με τη λίστα για τα ψώνια. Στη συνέχεια για να υλοποιήσουμε τις λίστες θα χρησιμοποιούμε έναν σύνθετο τύπο δεδομένων που θα τον ονομάζουμε *κόμβο.*Ένας κόμβος θα μπορούσε να αποτελείται από μία συμβολοσειρά (το όνομα), έναν φυσικό αριθμό (την ποσότητα) και έναν δείκτη. Κάθε εγγραφή στη λίστα για τα ψώνια μας θα αντιστοιχεί σε μια μεταβλητή τύπου κόμβου και θα αποτελείται από το όνομα του προϊόντος και την ποσότητα που πρέπει να αγοράσουμε. Ο δείκτης θα χρησιμοποιηθεί για να «συνδέσουμε» τη λίστα μας δίνοντάς μας τη θέση μνήμης που είναι αποθηκευμένος ο επόμενος κόμβος της λίστας (δες Σχήμα 0.2.6).

Θα επανέλθουμε σε αυτό το θέμα στη συνέχεια και θα το συζητήσουμε με μεγαλύτερη λεπτομέρεια. Κλείνουμε αυτή την παράγραφο κάνοντας μια σημαντική παρατήρηση.

Παρατήρηση 0.2.7. Αν θέλουμε να προσπελάσουμε κάποιον συγκεκριμένο κόμβο της λίστας θα χρειαστεί να επισκεφτούμε όλους τους προηγούμενους κόμβους! Αυτό οφείλεται στο γεγονός ότι δεν έχουμε καταχωρήσει τους κόμβους σε συνεχόμενες θέσεις μνήμης (λόγω της απαίτησης να έχουμε δυναμική καταχώρηση). Αν είχαμε στατική καταχώρηση, και κατ' επέκταση αποθήκευση σε συνεχόμενες θέσεις μνήμης, θα μπορούσαμε να πάμε κατευθείαν στον ζητούμενο κόμβο (δες Παράδειγμα 0.2.4).

0.3 Αλγοριθμική γλώσσα

Σκοπός μας σε αυτήν την παράγραφο είναι να συμφωνήσουμε στο σύνολο *εντολών* που θα απαρτίζουν την –εντός εισαγωγικών– γλώσσα προγραμματισμού που θα χρησιμοποιούμε για να συντάσσουμε τους αλγορίθμους μας. Στόχος μας είναι αυτό το σύνολο εντολών να είναι το μικρότερο δυνατό ώστε η ψευδογλώσσα μας, ναι μεν να είναι συνοπτική, αλλά να είναι και κατανοητή. Επιπλέον, όπως σε κάθε γλώσσα προγραμματισμού, οι εντολές θα πρέπει να είναι όσο το δυνατόν πιο αυστηρά ορισμένες.

Προτού ορίσουμε ρητά τις εντολές της ψευδογλώσσας μας ίσως ήταν χρήσιμο να δούμε ένα παράδειγμα αλγορίθμου γραμμένου σε αυτή.

MinElement(A, n)

Είσοδος: Μη κενός πίνακας A με n ακέραιους αριθμούς
Έξοδος : Η θέση x και η τιμή min του μικρότερου στοιχείου του πίνακα

$i \leftarrow 0, x \leftarrow 1, min \leftarrow A[1]$ *%Όπου $A[1]$ το πρώτο στοιχείο του πίνακα A*
while $i \leq n$ *%Αλλιώς:* **for** $i \leftarrow 1$ **to** n
  if $min > A[i]$ **then** *%$A[i]$ το i-οστό στοιχείο του A*
    $x \leftarrow i, min \leftarrow A[i]$
  $i \leftarrow i + 1$ *%Με το* **for** *δεν χρειάζεται*
return x, min

Ένας αλγόριθμος στη ψευδογλώσσα μας θα αποτελείται από τα ακόλουθα:

- *Τίτλος* (π.χ. MinElement)
- *Είσοδος:* Οι παράμετροι εισόδου (π.χ. MinElement(A, n)). Πέρα από τη δήλωσή τους δίπλα στο όνομα το αλγορίθμου καλό θα ήταν να προσθέσουμε και μια γραμμή που θα εξηγεί την κάθε παράμετρο (δες τον αλγόριθμο MinElement).
- *Έξοδος:* Οι τιμές που θέλουμε να *επιστρέφει* ο αλγόριθμός μας [1]. Επιστρέφονται με την εντολή **return** (π.χ. **return** x, min). Ο αλγόριθμος δεν θα εκτελέσει τις εντολές που βρίσκονται μετά από το **return** (θα τερματίσει).
- *Μεταβλητές* που θα φέρουν κάποιο όνομα (π.χ. i, x, min) [2]
- *Σταθερές* (π.χ. αριθμοί, True ή False, λέξεις κλπ.)
- *Τελεστές και πράξεις:* Αριθμητικοί τελεστές (π.χ. $+, -, \cdot, /$ κλπ.), τελεστές σύγκρισης (π.χ. $<, \leq, =$ κλπ.), λογικοί τελεστές (π.χ. **and**, **or**, **not** κλπ.)
- *Έκφραση ανάθεσης τιμής* (π.χ. $i \leftarrow 1$)
- *Εκφράσεις ελέγχου ροής προγράμματος:*
    - if-then-else (έλεγχος με συνθήκες):

[1] Δεν είναι απαραίτητο να επιστρέφουν πάντα κάποια τιμή.
[2] Όπως είπαμε και πριν δεν θα αναφέρουμε τι τύπου είναι η κάθε μεταβλητή, αυτό θα γίνεται κατανοητό από τα συμφραζόμενα.

```
if <Λογική συνθήκη> then
    <Κώδικας>
```

ή, με εναλλακτικό κώδικα:

```
if <Λογική συνθήκη> then
    <Κώδικας>
else if <Λογική συνθήκη> then
    <Κώδικας>
else
    <Εναλλακτικός κώδικας>
```

- while (έλεγχος επανάληψης):

```
while <Λογική συνθήκη>
    <Κώδικας που επαναλαμβάνεται>
```

- for (έλεγχος επανάληψης όταν γνωρίζουμε εκ των προτέρων το πλήθος των επαναλήψεων):

```
for <Μετρητής>←<Αρχική τιμή> to <Τελική τιμή>
    <Κώδικας που επαναλαμβάνεται>
```

- *Σχόλια:* Θα ξεκινάνε με το σύμβολο % (δες τον αλγόριθμο MinElement)

Παραδείγματα χρήσης αυτής της υποτυπώδης γλώσσας προγραμματισμούς θα δούμε και άλλα σε αυτό το κεφάλαιο. Αργότερα θα χρειαστεί να εμπλουτίσουμε λίγο ακόμα τη γλώσσα μας για να μπορέσουμε να υλοποιήσουμε (και να χρησιμοποιήσουμε) τις δομές δεδομένων που θα μας απασχολήσουν.

0.4 Αποδοτικότητα αλγορίθμων

Για να μπορέσουμε να διακρίνουμε πότε ένας αλγόριθμος είναι «προτιμητέος» για τις εφαρμογές μας θα χρειαστεί να ορίσουμε ένα κατάλληλο «μέτρο» για να εκτιμήσουμε και να συγκρίνουμε την (κάπως αφηρημένη) έννοια της *αποδοτικότητας των αλγορίθμων.*

Όταν τρέχουμε ένα πρόγραμμα στον Η/Υ μας θα θέλαμε να ελαχιστοποιήσουμε πολλές παραμέτρους ώστε το πρόγραμμά μας να είναι όσο το δυνατό πιο αποδοτικό. Οι βασικότερες είναι:

- Χρόνος μέχρι να τερματίσει το πρόγραμμά μας (*Χρόνος εκτέλεσης*).
- Το μέγεθος της μνήμης που θα χρειαστεί για να αποθηκεύσει τα δεδομένα που θα παράγει το πρόγραμμά μας (*Χώρος*).

Φυσικά θα μπορούσαν να μας απασχολούν και άλλες παράμετροι, όπως για παράδειγμα το πλήθος των παράλληλων επεξεργαστών που χρησιμοποιεί το πρόγραμμά μας, ή ακόμα και το ρεύμα που θα χρειαστεί να καταναλώσει ο Η/Υ για να τρέξει το πρόγραμμα αυτό ή η θερμοκρασία που θα αναπτύξει ο Η/Υ.

Εμείς θα επικεντρωθούμε μόνο στον χρόνο εκτέλεσης των προγραμμάτων που θα σχεδιάσουμε, ή πιο σωστά των αλγορίθμων. Μπορούμε να μετρήσουμε (ή να εκτιμήσουμε) τον χρόνο αυτό με δύο τρόπους:

1. *Εμπειρικά-Πειραματικά:* Υλοποιούμε τον αλγόριθμό μας σε κάποια γλώσσα προγραμματισμού. Τον εκτελούμε σε έναν συγκεκριμένο Η/Υ με πολλές διαφορετικές εισόδους (που τις επιλέγουμε συνήθως τυχαία, ακολουθώντας κάποια κατανομή) και μετράμε τον χρόνο που χρειάζεται για να τερματίσει για κάθε μία από αυτές (π.χ. σε δευτερόλεπτα). Έπειτα υπολογίζουμε τον μέσο όρο του χρόνου που έτρεξε ο αλγόριθμός μας.

Αυτός ο τρόπος εκτίμησης του χρόνου εκτέλεσης μπορεί να μας δώσει μια ρεαλιστική εικόνα της αποδοτικότητας του αλγορίθμου, έχει όμως ένα πολύ σοβαρό μειονέκτημα. Αν τρέξουμε τον αλγόριθμό μας σε κάποιον άλλο Η/Υ, για κάποιες άλλες εισόδους ή ακόμα κι αν επιλέξουμε κάποια άλλη γλώσσα προγραμματισμού, τα αποτελέσματα θα είναι διαφορετικά (ενδεχομένως και αντιφατικά).

2. *Θεωρητικά-Μαθηματικά:* Υπολογίζουμε με μαθηματικές μεθόδους (σαφώς πιο σύνθετες από την απλή χρονομέτρηση) τον χρόνο εκτέλεσης σαν συνάρτηση του *μεγέθους της εισόδου του αλγορίθμου.*

Καταρχάς, με τον όρο μέγεθος της εισόδου θεωρούμε μία μετρήσιμη ποσότητα που σχετίζεται με την είσοδό μας. Στις περισσότερες περιπτώσεις την ποσότητα αυτή θα μας την «επιβάλει» το ίδιο το πρόβλημα που επιλύουμε. Μερικά παραδείγματα που θα δούμε στη συνέχεια είναι η *διάσταση* ενός πίνακα, το *ύψος* ενός δέντρου, το *πλήθος των κορυφών* ενός γραφήματος κ.α. [1].

Δεν θα μετράμε *την χρονική πολυπλοκότητα* του αλγορίθμου π.χ. σε δευτερόλεπτα (καθώς δεν μας ενδιαφέρει ο πραγματικός χρόνος εκτέλεσης) αλλά σε πλήθος εκτελέσεων των *βασικών υπολογιστικών πράξεων*, όπως για παράδειγμα οι αναθέσεις τιμών, οι συγκρίσεις δύο αριθμών, οι αριθμητικές πράξεις κλπ. [2].

Ο θεωρητικός τρόπος υπολογισμού του χρόνου εκτέλεσης δεν πάσχει από τα μειονεκτήματα του εμπειρικού, γι' αυτό και θα τον επιλέξουμε για την ανάλυση των αλγορίθμων μας. Παρόλο που η ανάλυση των αλγορίθμων που θα συναντήσουμε θα είναι (στις περισσότερες περιπτώσεις) στοιχειώδης, θα χρειαστεί να εμβαθύνουμε περισσότερο στα βασικά χαρακτηριστικά του θεωρητικού τρόπου ανάλυσης. Ας δούμε ένα παράδειγμα.

Παράδειγμα 0.4.1. Ο παρακάτω αλγόριθμος υπολογίζει το *εσωτερικό γινόμενο* δύο διανυσμάτων:

[1] Στην Υπολογιστική Πολυπλοκότητα το μέγεθος της εισόδου ορίζεται τυπικά ως το μήκος της κωδικοποίησης της εισόδου, οποιαδήποτε μορφή και αν έχει αυτή.

[2] Στην πραγματικότητα ο χρόνος σε δευτερόλεπτα που χρειάζεται κάθε μία από αυτές τις πράξεις είναι συγκεκριμένος για τον Η/Υ μας (ή έστω μπορούμε να βρούμε μία σταθερά που τον φράσει από πάνω), έτσι θα μπορούσαμε να εκτιμήσουμε και τον πραγματικό χρόνο εκτέλεσης (δες Παράδειγμα 0.4.1).

InnerProduct(X, Y, n)

Είσοδος: Μονοδιάστατοι πίνακες X, Y με n ακέραιους αριθμούς
Έξοδος : Το εσωτερικό γινόμενό τους

```
z ← 0
for i ← 0 to n
    t ← X[i] · Y[i]
    z ← z + t
return z
```

Ας υποθέσουμε ότι το μέγεθος της εισόδου είναι το n, το πλήθος δηλαδή των στοιχείων του κάθε διανύσματος. Ο αλγόριθμος αυτός εκτελεί:

- $3n + 1$ αναθέσεις τιμών

- n πολλαπλασιασμούς ακέραιων

- n προσθέσεις ακέραιων

Συνεπώς, αν οι αναθέσεις (στον Η/Υ μας) χρειάζονται c_1 δευτερόλεπτα, οι πολλαπλασιασμοί ακέραιων c_2 και οι προσθέσεις c_3 ο πραγματικός χρόνος εκτέλεσης του αλγορίθμου για είσοδο μεγέθους n είναι:

$$
\begin{aligned}
T(n) &= (3n+1) \cdot c_1 + n \cdot c_2 + n \cdot c_3 \\
&= c_4 \cdot n + c_1
\end{aligned}
$$

όπου $c_4 = 3c_1 + c_2 + c_3$.

Δεν θα μας απασχολήσουν οι σταθερές στη συνάρτηση του χρόνο υπολογισμού – παρόλο που σε πραγματικά σενάρια έχουν πολύ μεγάλη σημασία – για αυτό και θα τις αγνοούμε. Αυτό που θα μας ενδιαφέρει είναι η *τάξη μεγέθους* της συνάρτησης αυτής. Στο Παράδειγμα 0.4.1 το μόνο που μας ενδιαφέρει είναι το γεγονός ότι η χρονική πολυπλοκότητα του InnerProduct περιγράφεται από μία *γραμμική συνάρτηση*.

Είναι πολύ συχνό το φαινόμενο ένας αλγόριθμος για κάποιες εισόδους να χρειάζεται να εκτελέσει πολλές περισσότερες βασικές πράξεις από όσες χρειάζεται για κάποιες άλλες εισόδους. Ένα πολύ απλό παράδειγμα είναι η αναζήτηση στοιχείου σε πίνακα. Ας υποθέσουμε ότι θέλουμε να ελέγξουμε αν ο αριθμός 0 βρίσκεται σε έναν μονοδιάστατο πίνακα με n αριθμούς. Για τις εισόδους που έχουν σαν πρώτο στοιχείο το 0 ο στοιχειώδης αλγόριθμος που ελέγχει στοιχείο-στοιχείο τον πίνακα θα κάνει έναν μόνον έλεγχο, ενώ για τις εισόδους που δεν περιέχουν 0 θα κάνει n ελέγχους (δες και Παράδειγμα 0.4.3). Θα πρέπει συνεπώς να αποφασίσουμε πολύ προσεκτικά ποια θα είναι η συνάρτηση που θα μας δώσει τη χρονική πολυπλοκότητα του αλγορίθμου: *Θα είναι μία σταθερή συνάρτηση ή μία γραμμική συνάρτηση;*

Ανάλυση χειρότερης περίπτωσης

Ορισμός 0.4.2. Έστω D_n το σύνολο όλων των εισόδων μεγέθους n. Έστω $I \in D_n$ είσοδος για την οποία ο αλγόριθμος θα χρειαστεί να εκτελέσει $t(I)$ βασικές πράξεις προτού τερματίσει. Η *πολυπλοκότητα χειρότερης περίπτωσης* του αλγορίθμου δίνεται από τη συνάρτηση:

$$T(n) = \max\{t(I) \in \mathbb{N} \mid I \in D_n\},\ n \in \mathbb{N}$$

Παράδειγμα 0.4.3. Ας δούμε πιο αναλυτικά τον αλγόριθμο αναζήτησης στοιχείου που αναφέραμε πριν:

LinearSearch(A, n, x)

Είσοδος: Πίνακας A με n ακέραιους και ακέραιος αριθμός x
Έξοδος : Η θέση της πρώτης εμφάνισης του x στον A (αν αυτός εμφανίζεται στον A) ή το μήνυμα "Δεν υπάρχει" σε αντίθετη περίπτωση

```
i ← 1
while i ≤ n and A[i] ≠ x
    i ← i + 1
if i > n then
    return "Δεν υπάρχει"
return i
```

Αν το στοιχείο x βρίσκεται στην πρώτη θέση του πίνακα (δηλαδή $A[1] = x$) τότε ο αλγόριθμος θα εκτελέσει:

- 1 ανάθεση τιμής
- 3 συγκρίσεις

αν όμως δεν βρίσκεται στον πίνακα (χειρότερη περίπτωση) ο αλγόριθμος θα εκτελέσει:

- n+1 αναθέσεις τιμής
- 2n+1 συγκρίσεις

Στην πολυπλοκότητα χειρότερης περίπτωσης μας ενδιαφέρει μόνο το γεγονός ότι για τη χειρότερη δυνατή είσοδο μεγέθους n για τον αλγόριθμό μας αυτός χρειάζεται γραμμικές το πλήθος (ως προς το n) βασικές πράξεις.

Ανάλυση αναμενόμενης περίπτωσης

Ορισμός 0.4.4. Έστω D_n το σύνολο όλων των εισόδων μεγέθους n. Έστω ότι η είσοδος $I \in D_n$ έχει πιθανότητα $p(I)$ να εμφανιστεί και ο αλγόριθμος θα χρειαστεί να εκτελέσει $t(I)$ βασικές πράξεις προτού τερματίσει. Η *πολυπλοκότητα μέσης περίπτωσης* του αλγορίθμου δίνεται από τη συνάρτηση:

$$T(n) = \sum_{I \in D_n} p(I) \cdot t(I),\ n \in \mathbb{N}$$

Η πολυπλοκότητα μέσης περίπτωσης προϋποθέτει ότι είναι γνωστή η κατανομή των εισόδων n για κάθε $n \in \mathbb{N}$, γεγονός που την κάνει πολύ δύσκολη στην εφαρμογή.

Συνήθως είναι προτιμότερο ένας αλγόριθμος να έχει καλύτερη πολυπλοκότητα χειρότερης περίπτωσης παρά πολυπλοκότητα μέσης περίπτωσης. Αυτό συμβαίνει γιατί η πολυπλοκότητα χειρότερης περίπτωσης μας δίνει μία «εγγύηση ταχύτητας» για *όλες* τις εισόδους. Η πιο γνωστή εξαίρεση σε αυτό είναι οι αλγόριθμοι Simplex και Karmakar για την επίλυση *γραμμικών προγραμμάτων*. Παρόλο που ο αλγόριθμος Simplex έχει εκθετική πολυπλοκότητα χειρότερης περίπτωσης, σε αντίθεση με τον Karmakar που έχει πολυωνυμική, προτιμάται στην πράξη καθώς έχει πολύ καλύτερη πολυπλοκότητα μέσης περίπτωσης [1]. Στις σημειώσεις αυτές θα αρκεστούμε στην ανάλυση χειρότερης περίπτωσης για τους αλγορίθμους που θα παρουσιάσουμε.

Κλείνοντας την παράγραφο αυτή ας αναφέρουμε ξανά τα βασικά πλεονεκτήματα του θεωρητικού τρόπου υπολογισμού της χρονικής πολυπλοκότητας:

- Δεν επηρεάζεται από την υπολογιστική ισχύ του εκάστοτε Η/Υ.
- Δεν επηρεάζεται από την επιλογή εισόδων.
- Δεν επηρεάζεται από τη γλώσσα προγραμματισμού ή την ικανότητα του προγραμματιστή (λόγω και της *ασυμπτωτικής ανάλυσης* που θα δούμε ευθύς αμέσως).

0.5 Ασυμπτωτική ανάλυση

Αναφέραμε πριν το γεγονός ότι όταν μετράμε τη χρονική πολυπλοκότητα ενός αλγορίθμου θα εξαλείφουμε τους «μη-σημαντικούς παράγοντες», όπως είναι οι σταθερές ή οι είσοδοι με μικρό μέγεθος. Αυτό το κάνουμε για δύο λόγους: Πρώτον γιατί ενδιαφερόμαστε μόνο για την τάξη μεγέθους και δεύτερον χάριν ευκολίας. Ας ορίσουμε τυπικά αυτήν τη «σύμβαση».

Ορισμός 0.5.1 (*Ο ασυμπτωτικό άνω φράγμα).* Έστω συνάρτηση $g : \mathbb{N} \to \mathbb{R}^+$. Ορίζουμε το σύνολο:

$$O(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \text{Υπάρχουν } c \in \mathbb{R}^+ \text{ και } n_0 \in \mathbb{N} \text{ τέτοια ώστε για κάθε } n \geq n_0\text{:} \\ f(n) \leq c \cdot g(n)\}$$

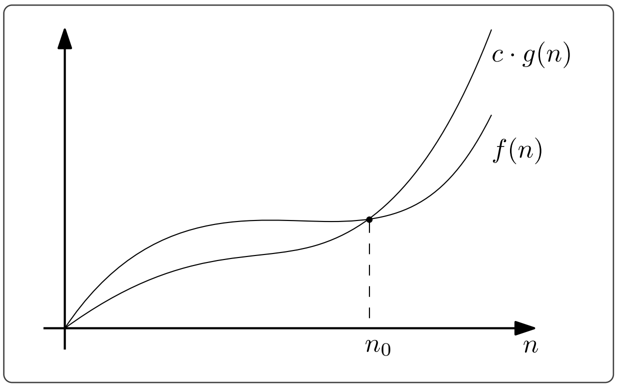
των συναρτήσεων που ασυμπτωτικά φράσσονται άνω από την g (δες Σχήμα 0.5.1).

Σύμβαση 0.5.2. Για να συμφωνήσουμε με τον καθιερωμένο τρόπο συμβολισμού θα γράφουμε $f = O(g)$ ή και $f(n) = O(g(n))$ αντί για $f \in O(g)$.

Παράδειγμα 0.5.3. Ας δούμε μερικά παραδείγματα του Ορισμού 0.5.1:

- $5\sqrt{n} = O(n)$

[1] Με απλά λόγια, παρουσιάζει εκθετική πολυπλοκότητα για πολύ σπάνιες εισόδους.



Σχήμα 0.5.1: *$f \in O(g)$ (ή αλλιώς $f = O(g)$). Το γεγονός αυτό το βεβαιώνουν οι σταθερές $c \in \mathbb{R}^+$ και $n_0 \in \mathbb{N}$.*

- $2n^3 + 5n^2 \log n = O(n^3)$
- $\frac{n}{2 \log n} = O(n)$

Παράδειγμα 0.5.4. Έστω ότι η χρονική πολυπλοκότητα του αλγορίθμου LinearSearch (στη χειρότερη περίπτωση που μας ενδιαφέρει) δίνεται από τη συνάρτηση $T(n)$. Για τη συνάρτηση αυτή ισχύει ότι $T(n) = O(n)$, έχει όπως λέμε *γραμμική τάξη μεγέθους*.

Οι βασικότερες τάξεις μεγέθους της χρονικής πολυπλοκότητας των αλγορίθμων δίνονται στον ακόλουθο πίνακα:

Χρόνος αλγορίθμου	**Συμβολισμός**
Σταθερός	$O(1)$
Πολυλογαριθμικός	$O(\log^k n),\ k \in \mathbb{N}$
Γραμμικός	$O(n)$
Πολυωνυμικός	$O(n^k),\ k \in \mathbb{N}$
Εκθετικός	$O(2^{n^k}),\ k \in \mathbb{N}$

Υπάρχουν κάποιοι βασικοί «κανόνες» που ακολουθούμε όταν θέλουμε να υπολογίσουμε την τάξη μεγέθους κάποιας συναρτήσεις [1]:

- Οι σταθερές είναι $O(1)$.
- Αν c σταθερά τότε $c \cdot f(n) = O(f(n))$.
- Αν $h(n) = O(g(n))$ και $g(n) = O(f(n))$ τότε $h(n) = O(f(n))$.
- Αν $g_1(n) = O(f_1(n))$ και $g_2(n) = O(f_2(n))$ τότε:

[1] Οι κανόνες αυτοί περιγράφονται εδώ διαισθητικά, μπορούν φυσικά να αποδειχθούν και τυπικά βάση του Ορισμού 0.5.1.

$$g_1(n) + g_2(n) = \begin{cases} O(f_1(n)) & ,\text{Αν } f_2(n) = O(f_1(n)) \\ O(f_2(n)) & ,\text{Αλλιώς} \end{cases}, \text{ και}$$
$$g_1(n) \cdot g_2(n) = O(f_1(n) \cdot f_2(n))$$

Όπως γίνεται εύκολα αντιληπτό, όταν υπολογίζουμε τη χρονική πολυπλοκότητα ενός αλγορίθμου προσπαθούμε να βρούμε το χαμηλότερο δυνατό άνω φράγμα. Για παράδειγμα για τη συνάρτηση $T(n)$ που περιγράφει την πολυπλοκότητα του LinearSearch προφανώς ισχύει επίσης ότι $T(n) = O(n^2)$, αλλά το $O(n)$ είναι πιο «σφιχτό» άνω φράγμα.

0.6 Παράδειγμα δομής δεδομένων: *Κατάλογοι* και *Ευρετήρια*

Η μόνη εκκρεμότητα που δεν έχουμε διευθετήσει ακόμα αφορά την οργάνωση μίας δομής δεδομένων και το πως αυτή δύναται να επηρεάσει τη χρονική πολυπλοκότητα των αλγορίθμων που χρησιμοποιούν τη δομή. Ας επανέλθουμε μία ακόμα φορά στο Παράδειγμα 0.1.1. Στο παράδειγμα αυτό ορίσαμε περιγραφικά μία δομή δεδομένων, τον *κατάλογο*, που υποστηρίζει κάποιες πολύ βασικές λειτουργίες (εισαγωγή, διαγραφή και αναζήτηση ονόματος) αλλά δεν έχει ιδιαίτερη εσωτερική δομή καθώς τα ονόματα (και όλα τα συνημμένα στοιχεία) είναι αποθηκευμένα στη μνήμη χωρίς να ακολουθούν κάποια συγκεκριμένη σειρά (ακολουθούν ουσιαστικά τη σειρά με την οποία εισήχθησαν στον κατάλογο).

Χωρίς να παρουσιάσουμε αναλυτικά τους αλγορίθμους που υλοποιούν τις τρεις λειτουργίες που υποστηρίζει ο κατάλογος [1] ας υπολογίσουμε (διαισθητικά) την χρονική πολυπλοκότητά τους:

Εισαγωγή:	$O(1)$	(προσθέτουμε τις νέες εγγραφές στο τέλος του καταλόγου [2])
Διαγραφή:	$O(n)$	(διαγράφουμε το i-οστό στοιχείο του καταλόγου [3])
Αναζήτηση:	$O(n)$	(στη χειρότερη περίπτωση)

όπου το μέγεθος της εισόδου n αντιπροσωπεύει το πλήθος των εγγραφών του καταλόγου.

Ας υποθέσουμε τώρα ότι έχουμε έναν κατάλογο στον οποίο δεν κάνουμε πολλές εισαγωγές και διαγραφές (π.χ. ο τηλεφωνικός μας κατάλογος). Το μεγαλύτερο μέρος των λειτουργιών που εκτελούμε πάνω σε αυτόν τον κατάλογο είναι η αναζήτηση ονόματος. Αντί να διαβάζουμε κάθε φορά (ενδεχομένως) ολόκληρο τον κατάλογο για να ελέγξουμε αν κάποιο όνομα εμφανίζεται σε αυτόν θα ήταν λογικό να κρατάμε τα ονόματα σε αλφαβητική σειρά. Ας δώσουμε ένα διαφορετικό όνομα σε αυτήν τη δομή και ας την πούμε *Ευρετήριο*.

Παρόλο που από μόνη της η επιπλέον οργάνωση που επιβάλαμε στον τρόπο που καταχωρούμε τα ονόματα στο ευρετήριο – τοποθέτηση στη σωστή θέση σύμφωνα με αλφαβητική σειρά – δεν κάνει τον αλγόριθμο αναζήτησης που περιγράψαμε (τον LinearSearch στην ουσία) πιο γρήγορο [4], μας δίνει τη δυνατότητα να σχεδιάσουμε αλγορίθμους που ουσιαστικά

[1] Δεν θα μπορούσαμε να το κάνουμε άλλωστε διότι δεν έχουμε ορίσει τυπικά αυτήν τη δομή δεδομένων.
[2] Υποθέτουμε ότι γνωρίζουμε ποια είναι η τρέχουσα τελευταία εγγραφή στον κατάλογο.
[3] Αν δεν είχαμε δυναμική καταχώρηση θα ήταν $O(1)$.
[4] Στη χειρότερη περίπτωση πάλι χρειάζεται χρόνο $O(n)$.

εκμεταλλεύονται αυτήν την οργάνωση και επιτυγχάνουν γρηγορότερους χρόνους. Ας δούμε έναν από αυτούς.

BinarySearch(E, n, x)

Είσοδος: Ευρετήριο E (αποθηκευμένο στη μνήμη ως πίνακας) με n εγγραφές και όνομα x

Έξοδος : Η θέση της πρώτης εμφάνισης του x στο E (αν αυτό εμφανίζεται στο E) ή το μήνυμα "Δεν υπάρχει" σε αντίθετη περίπτωση

1 $l \leftarrow 1,\ r \leftarrow n$
2 $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ *%$\lfloor\ \rfloor$ είναι το κάτω ακέραιο μέρος*
3 **while** $E[m] \neq x$ **and** $l \leq r$
4   **if** $E[m] < x$ **then** *%$<$ η σχέση του μικρότερου στη λεξικογραφική διάταξη*
5         $l \leftarrow m + 1$
6   **else**
7         $r \leftarrow m - 1$
8     $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
9 **if** $E[m] = x$ **then**
10   **return** m
11 **else**
12   **return** "Δεν υπάρχει"

Τα δύο πρώτα βήματα του αλγορίθμου (γραμμές 1 και 2) χρειάζονται σταθερό χρόνο, $O(1)$ σύμφωνα με τον συμβολισμό μας. Τα βήματα που βρίσκονται εντός του βρόγχου επανάληψης **while** (γραμμές 3–9) χρειάζονται επίσης χρόνο $O(1)$. Τα βήματα αυτά θα επαναληφθούν $O(\log n)$ φορές καθώς σε κάθε επανάληψη η τιμή του m υποδιπλασιάζεται (άρα το r θα γίνει μεγαλύτερο του l σε $O(\log n)$ επαναλήψεις). Τέλος τα βήματα 10–13 χρειάζονται και αυτά χρόνο $O(1)$, οπότε ο συνολικός χρόνος του αλγορίθμου περιγράφεται από τη συνάρτηση:

$$T(n) = O(1) + O(\log n) \cdot O(1) + O(1) = O(\log n)$$

Σε ένα ευρετήριο λοιπόν η λειτουργία αναζήτησης γίνεται (εκθετικά) πιο γρήγορα απ' ότι σε έναν κατάλογο. Η διαγραφή του i-οστού στοιχείου χρειάζεται πάλι χρόνο $O(n)$ (στη χειρότερη περίπτωση). Παρατηρήστε όμως ότι αν θέλαμε να διαγράψουμε ένα συγκεκριμένο στοιχείο, χωρίς να ξέρουμε ποια είναι η θέση του στο ευρετήριο, θα χρειαστούμε χρόνο $O(\log n)$, όσο χρόνο χρειαζόμαστε δηλαδή για να το βρούμε [1].

Στον αντίποδα όμως, η λειτουργία της εισαγωγής θα χρειαστεί παραπάνω χρόνο απ' ότι σε έναν κατάλογο ($O(n)$ στη χειρότερη περίπτωση), καθώς η κάθε εγγραφή θα πρέπει να μπαίνει στη σωστή σειρά (ως προς τη λεξικογραφική διάταξη) κατά την εισαγωγή της στο ευρετήριο.

Κλείνοντας το κεφάλαιο αυτό ας θέσουμε κάποιους στόχους για τη συνέχεια:

[1] Σε έναν κατάλογο ο χρόνος θα ήταν πάλι $O(n)$ καθώς θα έπρεπε να κάνουμε γραμμική αναζήτηση για να βρούμε το στοιχείο.

1. Θα παρουσιάσουμε τις βασικότερες δομές δεδομένων: *Πίνακες*, *Λίστες*, *Στοίβες*, *Ουρές*, *Δέντρα*, *Γραφήματα*.
2. Θα αναλύσουμε τον χρόνο που χρειάζονται οι βασικές λειτουργίες που υποστηρίζει κάθε μία από αυτές, κάνοντας μια εισαγωγή στη σχετική μεθοδολογία.
3. Θα δούμε κάποια παραδείγματα εφαρμογών τους που υποδεικνύουν τη χρησιμότητά τους.
4. Θα προσπαθήσουμε να κατηγοριοποιήσουμε τις δομές αυτές σύμφωνα με τις ομοιότητές τους.
5. Θα προσπαθήσουμε να τις αξιολογήσουμε ως προς την καταλληλότητά τους για συγκεκριμένες εργασίες και εφαρμογές.

ΚΕΦΑΛΑΙΟ 1

ΠΙΝΑΚΕΣ

Ήδη από το πρώτο κεφάλαιο χρησιμοποιήσαμε πίνακες προκειμένου τα παραδείγματά μας να είναι λίγο πιο ρεαλιστικά. Όποιος έχει ασχοληθεί με κάποιο μάθημα πληροφορικής σίγουρα θα έχει συναντήσει αλγορίθμους που χρησιμοποιούν πίνακες. Παρόλο που οι πίνακες είναι λίγο έως πολύ σε όλους γνωστοί, αποτελούν την απλούστερη στην κατανόηση δομή δεδομένων γι' αυτό συνιστούν την καλύτερη αφετηρία για τη μελέτη μας (πόσο μάλλον για αυτές τις σημειώσεις που επιχειρούν την παρουσίαση της ύλης χωρίς να βασίζονται σε πρότερες γνώσεις).

1.1 Μονοδιάστατοι πίνακες

Ο *(μονοδιάστατος) πίνακας* είναι μία στοιχειώδης δομή δεδομένων που υπάρχει ενσωματωμένη σε όλες σχεδόν τις γλώσσες προγραμματισμού και αποτελεί τη βάση υλοποίησης πολλών άλλων δομών. Ένας πίνακας αποθηκεύει δεδομένα (τα λεγόμενα *στοιχεία* του πίνακα):

α. που είναι του ίδιου τύπου (π.χ. πίνακας ακέραιων, πίνακας δεικτών κ.λπ.),

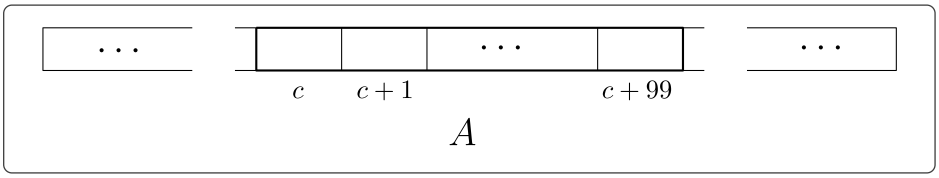
β. με στατικό τρόπο και σε συνεχόμενες θέσεις μνήμης.

Η αναφορά στα στοιχεία ενός πίνακα θα γίνεται με τη χρήση ενός ακέραιου δείκτη (π.χ. αν A είναι το όνομα ενός πίνακα με $A[3]$ συμβολίζουμε το στοιχείο που βρίσκεται στην τρίτη θέση από τα αριστερά, δες Παράδειγμα 0.2.4).

Ο πίνακας είναι μία *δομή τυχαίας προσπέλασης* καθώς ο χρόνος προσπέλασης ενός στοιχείου του είναι ανεξάρτητος από τη θέση του στον πίνακα (δες Παράδειγμα 0.2.4). Αυτό φυσικά οφείλεται στην αρχιτεκτονική των Η/Υ που χρησιμοποιούμε (μοντέλο RAM).

Η δομή δεδομένων πίνακας υποστηρίζει τις ακόλουθες τρεις λειτουργίες:

1. *Αρχικοποίηση (ή δημιουργία) πίνακα*



Σχήμα 1.1.1: *Με την εντολή $A \leftarrow$* **new matrix**(100) *ο Η/Υ δεσμεύει τις πρώτες 100 συνεχόμενες ελεύθερες θέσεις στη μνήμη του.*

2. *Προσπέλαση (ή ανάκτηση) του στοιχείου με δείκτη i*

3. *Καταχώρηση (ή ενημέρωση) του στοιχείου με δείκτη i*

Ας τις δούμε αναλυτικά:

1. *Αρχικοποίηση πίνακα:* Κατά την αρχικοποίηση ενός πίνακα ο Η/Υ δεσμεύει τον απαραίτητο χώρο στη μνήμη του. Για παράδειγμα στη Java για να αρχικοποιήσουμε έναν πίνακα A με 100 ακέραιους θα γράφαμε:

   ```
   int [] A;
   A = new int [100];
   ```

   Στην πρώτη γραμμή δηλώνουμε ότι η μεταβλητή A είναι τύπου πίνακα ακέραιων και στη δεύτερη ενημερώνουμε τον Η/Υ ότι ο A θα έχει 100 ακέραιους ώστε να δεσμεύσει τις απαιτούμενες θέσεις στη μνήμη.

   Στη δική μας ψευδογλώσσα όταν θέλουμε να αρχικοποιήσουμε έναν πίνακα θα γράφουμε:

   $A \leftarrow$ **new matrix**(n)

   όπου n το (ήδη γνωστό) μέγεθος του πίνακα (δες Σχήμα 1.1.1). Όπως προείπαμε, δεν θα χρειάζεται να αναφέρουμε τι στοιχεία θα περιέχει ο πίνακας καθώς αυτό θα γίνεται άμεσα αντιληπτό από τον υπόλοιπο αλγόριθμο.

   Ο χρόνος που απαιτείται για να εκτελεστεί η αρχικοποίηση θεωρούμε ότι είναι $O(1)$ (σταθερός), καθώς δεν μας απασχολεί το πως ο Η/Υ εκτελεί εσωτερικά τη δέσμευση θέσεων στη μνήμη.

2. *Προσπέλαση:* Για να προσπελάσουμε το στοιχείο με δείκτη i (να ανακτήσουμε δηλαδή την τιμή του) αρκεί να γράψουμε:

   $x \leftarrow A[i]$

όπου A ο πίνακας και το i είναι μικρότερο είτε ίσο από τη διάστασή του A [1]. Παράλληλα εκχωρούμε την τιμή στη μεταβλητή x. Ο χρόνος προσπέλασης ενός στοιχείου (πάλι εξαιτίας του μοντέλου RAM των H/Y μας) είναι $O(1)$.

3. *Καταχώρηση:* Αν θέλουμε να καταχωρήσουμε νέα τιμή ή να αλλάξουμε την ήδη υπάρχουσα στο στοιχείο με δείκτη i θα γράφουμε:

   $A[i] \leftarrow c$

   όπου c η τιμή που θέλουμε να καταχωρήσουμε στη θέση i του πίνακα A. Και αυτή η λειτουργία γίνεται σε σταθερό χρόνο.

Ενδεχομένως να μας φανεί χρήσιμο να έχουμε και μία εντολή που επιστρέφει το μέγεθος ενός πίνακα:

$x \leftarrow$ **length**(A)

Μπορούμε να υποθέσουμε επιπλέον ότι η **length** χρειάζεται σταθερό χρόνο και ότι δεν χρειάζεται να «διαβάσουμε» ολόκληρο τον πίνακα για να υπολογίσουμε το μέγεθός του (οπότε και θα σπαταλάγαμε γραμμικό ως προς το πλήθος στοιχείων χρόνο για να το υπολογίσουμε).

Ας δούμε ένα παράδειγμα εφαρμογής των παραπάνω.

Παράδειγμα 1.1.1. Θα δούμε έναν αλγόριθμο *ταξινόμησης* των στοιχείων ενός πίνακα ακέραιων ως προς αύξουσα τιμή. Ο αλγόριθμος αυτός είναι πολύ αποδοτικός όταν ο πίνακας εισόδου περιέχει αριθμούς που ανήκουν σε ένα μικρό σύνολο ακέραιων αριθμών. Αυτό θα γίνει εμφανές όταν θα αναλύσουμε τη χρονική πολυπλοκότητα του. Πέρα από τον πίνακα εισόδου A θα χρησιμοποιήσουμε και ένα βοηθητικό πίνακα, τον B.

BucketShort(A, n, m)

Είσοδος: Μη κενός πίνακας A με n ακέραιους από το 1 μέχρι τον ακέραιο m
Έξοδος : Ο πίνακας A με τα στοιχεία του ταξινομημένα κατά αύξουσα τιμή

```
B ← new matrix(m)
for i ← 1 to m                     % Γεμίζουμε τον βοηθητικό πίνακα B με μηδενικά
    B[i] ← 0
for j ← 1 to n
    B[A[j]] ← B[A[j]] + 1
k ← 1, i ← 1
```

[1] Δεν θα λάβουμε ειδική μέριμνα για τις περιπτώσεις όπου ο χρήστης χρησιμοποιεί τις εντολές με εσφαλμένο τρόπο, όπως για παράδειγμα αν έγραφε $A[5]$ για έναν πίνακα με λιγότερες από 5 στοιχεία. Σε μία ρεαλιστική γλώσσα προγραμματισμού θα έπρεπε να είχαμε λάβει υπόψιν μας τυχόν λάθη αυτού του τύπου ώστε να ενημερώσουμε τον χρήστη κατά τη φάση της μεταγλώττισης του προγράμματος.

```
7  while i ≤ m
8      if B[i] = 0 then
9          i ← i + 1
10     else
11         for j ← 1 to B[i]  %Συνολικά και για τα ≤ m for θα έχουμε n επαναλήψεις
12             A[k] ← i
13             k ← k + 1
14         i ← i + 1
15 return A
```

Προτού αναλύσουμε τον αλγόριθμο αυτό ας δούμε ένα παράδειγμα. Δίνοντας σαν είσοδο στον BucketShort τον πίνακα $A = \begin{bmatrix} 12 & 8 & 9 & 10 & 5 & 12 & 9 & 3 & 12 & 15 \end{bmatrix}$, τον ακέραιο 10 (πλήθος στοιχείων A) και τον ακέραιο 15 (μεγαλύτερη τιμή στον A) θα δημιουργήσει έναν πίνακα B με 15 μηδενικά, ο οποίος μετά το βήμα 5 θα διαμορφωθεί ως εξής:

$$B = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 2 & 1 & 0 & 3 & 0 & 0 & 1 \end{bmatrix}$$

Κάθε θέση του B αντιστοιχεί σε έναν ακέραιο αριθμό από το 1 έως το 15. Το γεγονός ότι οι δύο πρώτες θέσεις του έχουν μηδενικά σημαίνει ότι στον A δεν εμφανίζονται οι αριθμοί 1 και 2, ενώ το ότι έχει στην τρίτη θέση 1 δείχνει ότι στον A εμφανίζεται μία μόνο φορά ο αριθμός 3 (ο 9 εμφανίζεται δύο φορές γι' αυτό η ένατη θέση του B έχει τιμή 2). Αφού συμπληρωθεί ο B το μόνο που μένει να γίνει είναι να τοποθετηθούν στον A οι αριθμοί από το 1 μέχρι το 15 ανάλογα με το πλήθος εμφάνισής τους που καταγράψαμε στον B (γραμμές 6–14). Στο τέλος ο A θα έχει την μορφή:

$$A = \begin{bmatrix} 3 & 5 & 8 & 9 & 9 & 10 & 12 & 12 & 15 \end{bmatrix}$$

Ας βρούμε τώρα τη συνάρτηση που δίνει τη χρονική πολυπλοκότητα του BucketShort. Στα βήματα 1–6 έχουμε $O(m+n)$ βασικές πράξεις. Ο έλεγχος στη γραμμή 8 του **while** θα επαναληφθεί m φορές, η γραμμή 9 $m - r$ φορές, όπου r το πλήθος των διαφορετικών αριθμών από το 1 έως το 15 που εμφανίζονται στον A, θα εκτελεστούν r **for** στη γραμμή 11 και οι γραμμές 12 και 13 συνολικά θα επαναληφθούν n φορές (μία για κάθε στοιχείο του A). Επομένως στις γραμμές 7–14 γίνονται $O(m+n)$ βασικές πράξεις και άρα η συνολική χρονική πολυπλοκότητα του BucketShort είναι $O(m+n)$ [1].

Παρατηρήστε ότι αν το m είναι πολύ μικρότερο από το n (π.χ. αν έχουμε έναν πίνακα με 1.000.000 ακέραιους από το 1 μέχρι το 10) τότε ο BucketShort αποτελεί έναν πολύ γρήγορο αλγόριθμο ταξινόμησης.

Παρατήρηση 1.1.2. Μπορούμε με διάφορα τεχνάσματα να έχουμε δυναμική καταχώρηση στη μνήμη –χρησιμοποιώντας πίνακες– ούτως ώστε να αποκομίσουμε τα «οφέλη» και από

[1] Παρατηρήστε ότι εδώ υπολογίζουμε την χρονική πολυπλοκότητα ως συνάρτηση δύο παραμέτρων!

$$\begin{array}{cc} & \text{Στήλες} \\ A = & \begin{bmatrix} A[1][1] & A[1][2] & A[1][3] \\ A[2][1] & A[2][2] & A[2][3] \end{bmatrix} \text{ Γραμμές} \end{array}$$

Σχήμα 1.2.1: *Παράδειγμα δισδιάστατου πίνακα με 2 γραμμές και 3 στήλες.*

τους δύο τρόπους καταχώρησης (ταχύτητα προσπέλασης αλλά και ευελιξία ως προς το μέγεθος). Ένα από αυτά είναι το εξής: Όποτε προκύπτει η ανάγκη για νέες θέσεις στον πίνακα διπλασιάζουμε το μέγεθός του, ενώ όταν οι άδειες θέσεις του γίνουν περισσότερες από τις μισές υποδιπλασιάζουμε το μέγεθός του. Η απόδειξη ότι αυτή η στρατηγική είναι αποδοτική προκύπτει μέσω *αντισταθμιστικής ανάλυσης* όπου μετράμε τον μέσο χρόνο που χρειάζεται για να ολοκληρωθεί μια ακολουθία από λειτουργίες (π.χ. n εισαγωγές και n διαγραφές).

1.2 Δισδιάστατοι πίνακες (και πίνακες μεγαλύτερης διάστασης)

Όπως και στους μονοδιάστατους πίνακες τα στοιχεία ενός *δισδιάστατου πίνακα*:

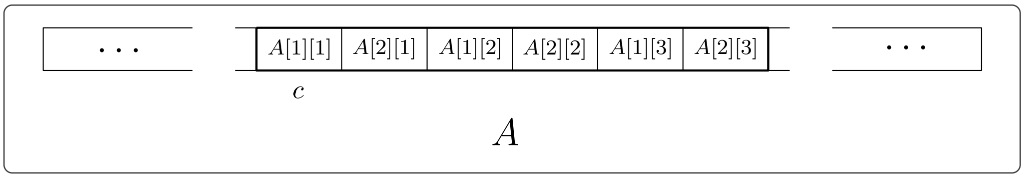
α. είναι του ίδιου τύπου,

β. καταχωρούνται στη μνήμη με στατικό τρόπο και σε συνεχόμενες θέσεις.

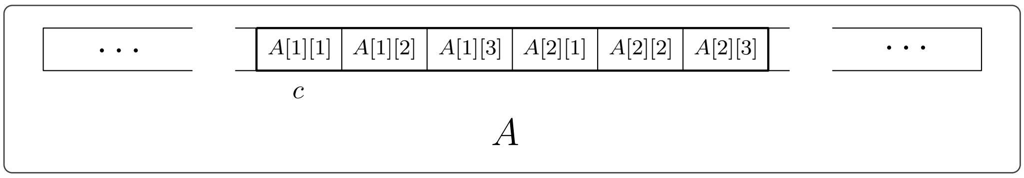
Καθώς ο πίνακας πλέον έχει δύο διαστάσεις μπορούμε να διακρίνουμε τα στοιχεία του σε *στήλες* και *γραμμές*. Η αναφορά στα στοιχεία γίνεται με τη χρήση δύο ακέραιων δεικτών (π.χ. αν A είναι το όνομα του πίνακα με $A[2][3]$ συμβολίζουμε το στοιχείο που βρίσκεται στη δεύτερη γραμμή και την τρίτη στήλη (δες Σχήμα 1.2.1). Για το μέγεθος του πίνακα θα γράφουμε $p \times q$ και θα εννοούμε ότι ο πίνακας έχει p γραμμές και q στήλες.

Η καταχώρηση στη μνήμη των στοιχείων ενός δισδιάστατου πίνακα (η *αναπαράστασή* του) γίνεται με τη χρήση ενός μονοδιάστατου πίνακα και μπορεί να ακολουθεί δύο τρόπους:

1. *Διάταξη κατά στήλες:* Αποθηκεύουμε τα στοιχεία του στήλη-στήλη (δες Σχήμα 1.2.2). Έτσι, αν γνωρίζει ο Η/Υ τη διεύθυνση c που έχει αποθηκευτεί το πρώτο στοιχείο του πίνακα και το μέγεθος του, έστω $p \times q$, για να προσπελάσει το στοιχείο $A[i][j]$ αρκεί να πάει στη διεύθυνση:
$$c + (j-1) \cdot p + i - 1$$
(Τονίζουμε για μία ακόμα φορά τη σύμβαση που έχουμε κάνει ότι κάθε στοιχείο, οποιουδήποτε τύπου και αν είναι, «χωράει» να αποθηκευτεί σε μία και μόνο θέση μνήμης.)



Σχήμα 1.2.2: *Η καταχώρηση στη μνήμη του πίνακα στο Σχήμα 1.2.1 στήλη-στήλη.*



Σχήμα 1.2.3: *Η καταχώρηση στη μνήμη του πίνακα στο Σχήμα 1.2.1 γραμμή-γραμμή.*

2. *Διάταξη κατά γραμμές:* Αποθηκεύουμε τα στοιχεία του γραμμή-γραμμή (Σχήμα 1.2.3). Τώρα το στοιχείο $A[i][j]$ βρίσκεται στη διεύθυνση:

$$c + (i-1) \cdot q + j - 1$$

Στο παράδειγμα 1.2.3 θα δούμε ακόμα έναν τρόπο καταχώρησης στη μνήμη που αφορά όμως πίνακες ειδικής μορφής.

Παρατήρηση 1.2.1. Για πίνακες μεγαλύτερης διάστασης δουλεύουμε με ανάλογο τρόπο [1].

Οι λειτουργίες της προσπέλασης και της ενημέρωσης για τους δισδιάστατους πίνακες είναι ίδιες με αυτές των μονοδιάστατων (μόνο που πλέον έχουμε μία ακόμα διάσταση στα στοιχεία), ενώ κατά την αρχικοποίηση πρέπει να εισάγουμε και τη δεύτερη διάσταση του πίνακα:

$A \leftarrow$ **new matrix**(p, q) *%p είναι οι γραμμές και q οι στήλες*

Τέλος για να πάρουμε το πλήθος γραμμών και στηλών ενός πίνακα θα χρησιμοποιούμε τις ακόλουθες εντολές:

$x \leftarrow$ **rows**(A) *%Επιστρέφει το πλήθος γραμμών*
$y \leftarrow$ **cols**(A) *%Επιστρέφει το πλήθος στηλών*

Παράδειγμα 1.2.2. Σαν πρώτο παράδειγμα εφαρμογής των δισδιάστατων πινάκων θα δούμε πως υλοποιούμε τον πολλαπλασιασμό μεταξύ δύο πινάκων (με κατάλληλες διαστάσεις).

[1] Για παράδειγμα, αν έχουμε πίνακα τριών διαστάσεων τον χωρίζουμε πρώτα σε δισδιάστατους πίνακες και έπειτα τους αποθηκεύουμε σύμφωνα με όσα είπαμε πριν.

MatrixProduct(A, B)

Είσοδος: Μη κενοί πίνακες ακέραιων A, B
Έξοδος : Ο πίνακας C που αποτελεί το γινόμενο των A και B

```
if cols(A) ≠ rows(B) then
    return "Λάθος διαστάσεις"
else
    p ← rows(A), q ← cols(A), r ← cols(B)
    C ← new matrix(p, r)
    for i ← 1 to p
        for j ← 1 to r
            C[i][j] ← 0
            for k ← 1 to q
                C[i][j] ← C[i][j] + A[i][k] · B[k][j]
    return C
```

Ο χρόνος του MatrixProduct είναι $O(p \cdot q \cdot r)$ ($O(n^3)$ αν έχουμε να κάνουμε με τετραγωνικούς $n \times n$ πίνακες).

Σαν δεύτερο παράδειγμα θα δούμε πως μπορούμε να χρησιμοποιήσουμε με έξυπνο τρόπο μονοδιάστατους πίνακες για να καταχωρήσουμε στη μνήμη δισδιάστατους πίνακες που έχουν κάποια ειδική μορφή (όπως για παράδειγμα *συμμετρικούς πίνακες*, *άνω τριγωνικούς* κλπ.). Σε αυτές τις ειδικές κατηγορίες πινάκων δεν χρειάζεται να «σπαταλήσουμε» μνήμη για να αποθηκεύσουμε το περιεχόμενο των θέσεων που δεν περιέχουν «ουσιαστική πληροφορία» (που π.χ. περιέχουν μηδενικά).

Παράδειγμα 1.2.3. Ένας πίνακας ακέραιων καλείται *αραιός* όταν το πλήθος των μη μηδενικών στοιχείων του είναι πολύ μικρό σε σχέση με το μέγεθός του. Για παράδειγμα στον πίνακα A που ακολουθεί τα μη-μηδενικά στοιχεία είναι μόνο 8 από τα 64.

$$A = \begin{bmatrix} 12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\ -6 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Προκειμένου να αποθηκεύσουμε τον A στη μνήμη αποθηκεύουμε έναν πίνακα $DATA$ που περιέχει τα μη-μηδενικά στοιχεία σειρά-σειρά:

$$DATA = \begin{bmatrix} 12 & 3 & 6 & 5 & 2 & 8 & -6 & 3 \end{bmatrix}$$

και δύο ακόμα πίνακες ROW και COL που μας δίνουν τη γραμμή και τη στήλη του κάθε στοιχείου του πίνακα A:

$$ROW = \begin{bmatrix} 1 & 2 & 2 & 3 & 3 & 4 & 5 & 5 \end{bmatrix}$$

$$COL = \begin{bmatrix} 1 & 1 & 3 & 3 & 6 & 3 & 1 & 3 \end{bmatrix}$$

Έτσι αντί για 64 θέσεις στη μνήμη χρησιμοποιούμε μόνο 24 [1].

Για να ενημερώσουμε ένα στοιχείο σύμφωνα με αυτόν τον τρόπο αναπαράστασης θα χρειαστεί να υπολογίσουμε εκ νέου τους τρεις πίνακες. Όπως θα δούμε ο χρόνος που θα χρειαστούμε είναι $O(m)$, όπου m το πλήθος των μη μηδενικών στοιχείων. Ας δούμε τον αλγόριθμο που υλοποιεί αυτήν τη λειτουργία (θα υποθέσουμε ότι η καινούργια τιμή είναι μη μηδενική [2]).

Update$(DATA, ROW, COL, x, i, j)$

Είσοδος: Οι πίνακες $DATA, ROW, COL$ που αναπαριστούν έναν δισδιάστατο πίνακα και η τιμή x που πρέπει να μπει στη θέση i, j του πίνακα

Έξοδος : Οι πίνακας $DATA', ROW', COL'$ που προκύπτουν μετά την ενημέρωση του στοιχείου στη θέση i, j

$m \leftarrow$ **length**$(DATA)$
for $k \leftarrow 1$ **to** m
    if $ROW[k] = i$ **and** $COL[k] = j$ **then** *% Το στοιχείο που ενημερώνουμε είναι μη-μηδενικό*
        $DATA[k] \leftarrow x$
        return $DATA, ROW, COL$
$DATA' \leftarrow$ **new matrix**$(m+1)$ *% Αν το στοιχείο i, j ήταν τελικά μηδενικό*
$ROW' \leftarrow$ **new matrix**$(m+1)$
$COL' \leftarrow$ **new matrix**$(m+1)$
$k \leftarrow 1$
while $(ROW[k] < i)$ **or** $(ROW[k] = i$ **and** $COL[k] < j)$
    $DATA'[k] \leftarrow DATA[k]$ *% Δεν αλλάζουν αυτά τα στοιχεία του $DATA$*
    $ROW'[k] \leftarrow ROW[k]$
    $COL'[k] \leftarrow COL[k]$
    $k \leftarrow k+1$
$DATA'[k] \leftarrow x$ *% Προσθέτουμε το νέο στοιχείο*
$ROW'[k] \leftarrow i$
$COL'[k] \leftarrow j$

[1] Φανταστείτε ότι σε έναν πολύ μεγαλύτερο αραιό πίνακα η διαφορά θα ήταν αρκετά πιο αισθητή.

[2] Σαν άσκηση μπορείτε να προσθέσετε την περίπτωση όπου αλλάζουμε μη μηδενικό στοιχείο με μηδενικό (Άσκηση 1.1).

while $k \leq m$ *% Προσέξτε ότι χρησιμοποιούμε το ίδιο k με πριν*
    $DATA'[k+1] \leftarrow DATA[k]$ *% Τοποθετούμε τα υπόλοιπα στοιχεία του $DATA$ στον $DATA'$*
    $ROW'[k+1] \leftarrow ROW[k]$
    $COL'[k+1] \leftarrow COL[k]$
    $k \leftarrow k+1$
return $DATA', ROW', COL'$

Παρατηρήστε ότι όλες οι επαναλήψεις γίνονται το πολύ m φορές και σε κάθε επαναληπτικό βρόγχο γίνεται σταθερό πλήθος πράξεων. Συνεπώς ο χρόνος του αλγορίθμου είναι $O(m)$.

Μπορεί η ενημέρωση στοιχείου (και αντίστοιχα η διαγραφή στοιχείου) να είναι πολύ πιο αργή από τον κλασικό τρόπο καταχώρησης του πίνακα, αλλά η αναζήτηση στοιχείου (δες την Παράγραφο 1.3) γίνεται σε χρόνο $O(m)$ αντί για $O(p \cdot q)$ καθώς αρκεί να ψάξουμε το στοιχείο στον πίνακα $DATA$ και όχι σε ολόκληρο τον πίνακα.

1.3 Αναζήτηση σε πίνακα

Από το πρώτο κιόλας κεφάλαιο έχουμε δει τρόπους να αναζητήσουμε ένα στοιχείο σε έναν πίνακα [1]:

1. *Γραμμική αναζήτηση*: Ο αλγόριθμος LinearSearch με χρόνο $O(n)$ και παρουσιάζεται στο Παράδειγμα 0.4.3,
2. *Δυαδική αναζήτηση* σε ταξινομημένο πίνακα (ευρετήριο): Ο αλγόριθμος BinarySearch με χρόνο $O(\log n)$ παρουσιάζεται στη Παράγραφο 0.6,

όπου n το πλήθος στοιχείων του πίνακα.

Ασκήσεις

1.1. Προσθέστε στον αλγόριθμο Update (Παράδειγμα 1.2.3) την περίπτωση όπου αλλάζουμε μη μηδενικό στοιχείο με μηδενικό.

1.2. Θυμηθείτε την αναπαράσταση αραιού πίνακα με τρεις πίνακας που είδαμε στο Παράδειγμα 1.2.3. Δώστε αλγόριθμο Remove$(DATA, ROW, COL, i, j)$ που «αντικαδιστά» το στοιχείο i, j του πίνακα με μηδέν και επιστρέφει τους τρεις πίνακες ανανεωμένους.

[1] Χάριν απλότητας τα παραδείγματα που είδαμε αφορούν μονοδιάστατους πίνακες, οι αλγόριθμοι όμως προσαρμόζονται εύκολα και σε πολυδιάστατους πίνακες.

1.3. Υλοποιήστε τον αλγόριθμο γραμμικής αναζήτησης (Παράδειγμα 0.4.3) για έναν δισδιάστατο πίνακα (μπορείτε να επιλέξετε αν τα στοιχεία θα ελέγχονται σειρά-σειρά ή στήλη στήλη).

1.4. Ένας $n \times n$ πίνακας A είναι ταξινομημένος αν κάθε γραμμή του είναι ταξινομημένη και $A[i][n] \leq A[i+1][1]$ για κάθε $i = 1, \ldots, n-1$. Σκεφτείτε (και υλοποιήστε) αλγόριθμο αναζήτησης αποδοτικότερο της απλής γραμμικής αναζήτησης (Άσκηση 1.3) και εκτιμήστε τον χρόνο που χρειάζεται ως προς το n.

1.5. Θυμηθείτε την ακολουθία Fibonacci:

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n) = f(n-1) + f(n-2)) \text{ , για } n \geq 2 \end{cases}$$

Δώστε αλγόριθμο που δέχεται ως είσοδο φυσικό αριθμό n και υπολογίζει τον n-οστό όρο της, υπολογίζοντας όλες τις ενδιάμεσες τιμές και αποθηκεύοντάς τες σε έναν πίνακα.

ΚΕΦΑΛΑΙΟ 2

ΛΙΣΤΕΣ

Η δομή δεδομένων *(συνδεδεμένη) Λίστα* αποθηκεύει δεδομένα (του ίδιου τύπου, αν και αυτό δεν είναι απαραίτητο) που ακολουθούν κάποια διάταξη, έτσι ώστε κάθε στοιχείο της να έχει συγκεκριμένη θέση. Ίσως το πιο αντιπροσωπευτικό παράδειγμα λίστας να ήταν η λίστα με τα ψώνια μας όπου παρόλο που δεν έχει μεγάλη σημασία η σειρά των προϊόντων δεν παύει να υπάρχει (δες Παράδειγμα 0.2.5). Φυσικά θα θέλαμε η σειρά των στοιχείων της λίστας να σηματοδοτεί κάτι πιο ουσιαστικό (π.χ. να έχουμε τα προϊόντα σε αλφαβητική σειρά) για να την εκμεταλλευτούμε και να κατασκευάσουμε πιο αποδοτικούς αλγόριθμους.

Ως εδώ δεν προκύπτει κάποια διαφοροποίηση με τους πίνακες. Η ουσιαστική διαφορά είναι ότι τις λίστες τις καταχωρούμε στη μνήμη με δυναμικό τρόπο και έτσι δεν χρειάζεται εκ των προτέρων να ξέρουμε το τελικό μέγεθός τους. Αυτό φυσικά κάνει τις λίστες να μην αποτελούν δομή τυχαίας προσπέλασης. Αποτελεί μία δομή *σειριακής προσπέλασης*, έτσι ο χρόνος για την προσπέλαση ενός στοιχείου είναι μεγαλύτερος από αυτόν των πινάκων [1].

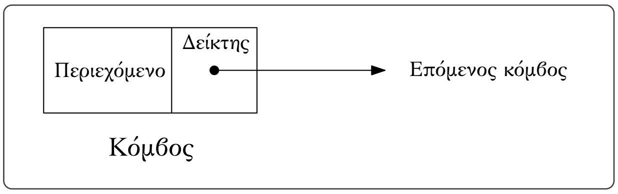
Οι λίστες που θα δούμε χωρίζονται σε δύο βασικές κατηγορίες:

- *Γενικές λίστες:* Επιτρέπουν την εισαγωγή και τη διαγραφή στοιχείου από οποιαδήποτε θέση (π.χ. *Απλά συνδεδεμένες λίστες*, *Διπλά συνδεδεμένες*, *Κυκλικά συνδεδεμένες* κ.λπ.).

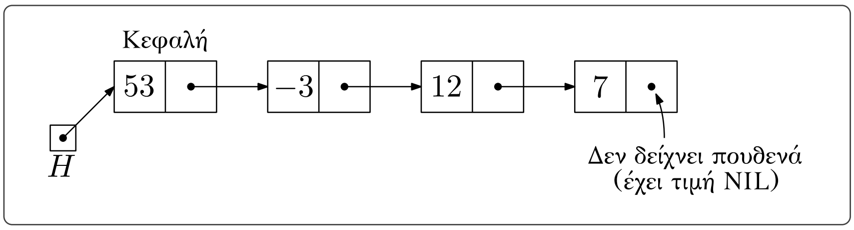
- *Ειδικές λίστες:* Μπορούμε να εισάγουμε ή/και να διαγράψουμε στοιχείο μόνο από συγκεκριμένη θέση (π.χ. *Στοίβες*, *Ουρές* κ.λπ.).

Σε αυτό το κεφάλαιο θα αναφερθούμε μόνο στις γενικές λίστες. Οι ειδικές λίστες θα μας απασχολήσουν στο κεφάλαιο που ακολουθεί.

[1] Σε συγκεκριμένες όμως περιπτώσεις μπορούμε να εισάγουμε ή να διαγράψουμε στοιχεία πιο γρήγορα από ότι σε έναν πίνακα. Για παράδειγμα όταν έχουμε ταξινομημένο πίνακα και θέλουμε να εισάγουμε στοιχείο στην αρχή του, θα πρέπει να μετακινήσουμε όλα τα στοιχεία του πίνακα μία θέση δεξιά (χρόνος $O(n)$, ενώ σε μία ταξινομημένη λίστα αυτό μπορεί να γίνει σε σταθερό χρόνο καθώς απλά εισάγουμε το στοιχείο στην αρχή της.)



Σχήμα 2.1.1: *Παράδειγμα κόμβου.*



Σχήμα 2.1.2: *Παράδειγμα σχηματικής αναπαράστασης της λίστας ακέραιων* $53, -3, 12, 7$. *Ο δείκτης με όνομα* H *είναι ο δείκτης κεφαλής της λίστας.*

2.1 Απλά συνδεδεμένες λίστες

Για την αναπαράσταση μίας (απλά) συνδεδεμένης λίστας θα χρησιμοποιήσουμε έναν σύνθετο τύπο δεδομένων που θα τον ονομάσουμε *κόμβο*. Ο κόμβος περιέχει τα ακόλουθα πεδία:

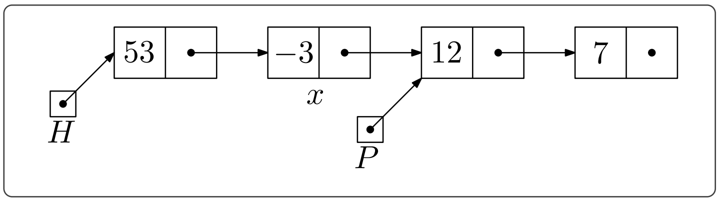
1. Περιεχόμενο στοιχείου λίστας (π.χ. ακέραιος, συμβολοσειρά κ.λπ., ή ακόμα και συνδυασμός περισσότερων πραγμάτων).

2. Δείκτης με τιμή τη διεύθυνση μνήμης που έχει αποθηκευτεί ο κόμβος που περιέχει το επόμενο στοιχείο της λίστας (δες Σχήμα 2.1.1). Τον δείκτη αυτόν θα τον αποκαλούμε *δείκτη επόμενου*.

Πέρα από τους δείκτες που χρησιμοποιούμε εντός των κόμβων, θα χρησιμοποιήσουμε έναν ακόμα δείκτη που θα «δείχνει» στον πρώτο κόμβο της λίστας, τη λεγόμενη *κεφαλή*. Τον δείκτη αυτόν θα τον αποκαλούμε *δείκτη κεφαλής*. Τέλος, θα θεωρούμε ότι η τιμή του δείκτη του τελευταίου κόμβου της λίστας δεν είναι ορισμένη, θα έχει δηλαδή κάποια «ασυνάρτητη» τιμή. Την τιμή αυτή θα την συμβολίσουμε με NIL (δες Σχήμα 2.1.2). Σύμφωνα με τη Σύμβαση 0.2.2 θεωρούμε ότι το περιεχόμενο του κόμβου αποθηκεύεται σε μία θέση μνήμης. Ολόκληρη η λίστα προφανώς θα αποθηκεύεται σε παραπάνω από μία θέσεις μνήμης [1].

Σε μια αντικειμενοστραφή γλώσσα προγραμματισμού (Python, C++, Java κ.λπ.) για να ορίσουμε έναν κόμβο θα έπρεπε πρώτα να ορίσουμε έναν σύνθετο τύπο δεδομένων, έστω **node**. Για παράδειγμα στην Java θα γράφαμε:

[1] Θα χρειαστούμε τόσες θέσεις όσα και τα στοιχεία, συν ακόμα μία για τον δείκτη κεφαλής.



Σχήμα 2.1.3: *Η λίστα του Παραδείγματος 2.1.1.*

```
Private class Node
{
   Item item;   // Περιεχόμενο κόμβου
   Node next;   // Δείκτης για τον επόμενο κόμβο
}
```

Αν τώρα είχαμε μία μεταβλητή τύπου **node**, έστω την x, και θέλαμε να αναφερθούμε στο πεδίο item της (στο περιεχόμενο του κόμβου δηλαδή) θα γράφαμε `x.item`, ενώ για το πεδίο next θα γράφαμε `x.next`.

Θα υιοθετήσουμε αυτόν τον συμβολισμό και στη δική μας ψευδογλώσσα. Έτσι αν η x είναι μεταβλητή τύπου κόμβου θα γράφουμε:

- $x.item$ για να αναφερθούμε στο περιεχόμενο του κόμβου με όνομα x,

- $x.next$ για να αναφερθούμε στον δείκτη επόμενου του κόμβου με όνομα x.

Παράδειγμα 2.1.1. Στη λίστα που δείχνει το Σχήμα 2.1.3 αν μέσα στον κώδικα παραδείγματος χάρη γράψουμε:

$$y \leftarrow x.item$$

θα δώσουμε στην μεταβλητή y σαν τιμή το περιεχόμενο του κόμβου x, δηλαδή την τιμή -3. Επίσης θα μπορούσαμε να γράψουμε:

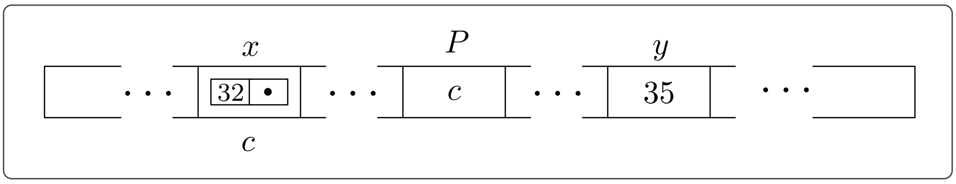
$$P \leftarrow x.next$$

αρχικοποιώντας την μεταβλητή τύπου δείκτη P και δίνοντάς της την τιμή του δείκτη $x.next$ (δες Σήμα 2.1.3).

Θα χρειαστεί να εφοδιάσουμε τη γλώσσα μας με μερικές επιπλέον εντολές. Θα γράφουμε:

- $x \leftarrow$**new node**(y) για να αρχικοποιήσουμε μία μεταβλητή τύπου κόμβου με περιεχόμενο y (δηλαδή $x.item = y$) που δεν συνδέεται με κάποιον άλλο κόμβο (δηλαδή $x.next =$ NIL),

- $P \leftarrow$ **address**(x), όπου x μεταβλητή τύπου κόμβου, για να δώσουμε στη μεταβλητή (τύπου δείκτη) P την τιμή της διεύθυνσης που έχει αποθηκευτεί ο x,



Σχήμα 2.1.4: *Το περιεχόμενο της μνήμης (που μας αφορά) στο Παράδειγμα 2.1.2.*

- $x \leftarrow$**node**(P), όπου P μεταβλητή τύπου δείκτη, για να δώσουμε στην μεταβλητή x ως τιμή τον κόμβο που είναι αποθηκευμένος στη διεύθυνση που δείχνει ο P.

Παράδειγμα 2.1.2. Ας δούμε ένα παράδειγμα χρήσης αυτών των εντολών:

$x \leftarrow$ **new node**(32) *%Δες Σχήμα 2.1.4*
$P \leftarrow$ **address**(x) *%Η τιμή που θα πάρει ο P είναι c*
$y \leftarrow$ **node**$(P).item + 3$ *%Η τιμή της y θα είναι 35*

Λειτουργίες απλά συνδεδεμένων λιστών

Οι συνδεδεμένες λίστες θέλουμε να υποστηρίζουν τις ακόλουθες λειτουργίες:

1. *Αρχικοποίηση λίστας*

2. *Έλεγχος για το αν η λίστα είναι κενή*

3. *Διαπέραση λίστας*

4. *Εισαγωγή στοιχείου*

5. *Διαγραφή στοιχείου*

6. *Αναζήτηση στοιχείου*

Ας δούμε αναλυτικά πως υλοποιούνται αυτές οι 6 λειτουργίες:

1. *Αρχικοποίηση λίστας:* Για να αρχικοποιήσουμε μία λίστα (να δημιουργήσουμε δηλαδή μια *κενή* λίστα) το μόνο που χρειάζεται να κάνουμε είναι να ορίσουμε μία μεταβλητή τύπου δείκτη, έστω την H, η οποία θα αποτελεί τον δείκτη κεφαλής της λίστας. Θα πρέπει να δώσουμε στην H την τιμή NIL καθώς η λίστα δεν περιέχει κάποιο κόμβο (και προφανώς δεν υπάρχει κόμβος που να αποτελεί την κεφαλή της). Αντί να γράφουμε $H \leftarrow$ NIL, για λόγους συμβατότητας θα γράφουμε:

   $H \leftarrow$ **new list**

   Προφανώς ο χρόνος αρχικοποίησης μίας λίστας είναι $O(1)$.

2. *Έλεγχος για το αν η λίστα είναι κενή:* Αν κάποιος μας δώσει τον δείκτη κεφαλής H μιας λίστας μπορούμε εύκολα να διαπιστώσουμε αν η λίστα είναι κενή, ελέγχοντας αν η τιμή του H είναι NIL. Αυτός ο υποτυπώδης αλγόριθμος υλοποιείται ως εξής:

   IsEmpty(H)

   Είσοδος: Ο δείκτης κεφαλής H μιας (απλά συνδεδεμένης) λίστας
   Έξοδος : True αν η λίστα είναι κενή, False διαφορετικά

   ```
   if H = NIL then
       return True
   else
       return False
   ```

   Εφόσον δεν χρειάζεται να «διαβάσουμε» ολόκληρη τη λίστα και κάνουμε μόνο ένα σταθερό πλήθος ελέγχων (μόνο έναν για την ακρίβεια), ο χρόνος που χρειάζεται ο IsEmpty είναι $O(1)$.

3. *Διαπέραση λίστας:* Αν εξαιρέσουμε την εισαγωγή και τη διαγραφή στοιχείου η διαπέραση είναι η πιο σημαντική πράξη σε μία δομή δεδομένων. Κατά τη διαπέραση περνάμε απ' όλα τα στοιχεία της λίστας και εφαρμόζουμε σε αυτά κάποια διαδικασία (π.χ. αύξηση της τιμής τους κατά ένα). Ας υποθέσουμε ότι θέλουμε να εφαρμόσουμε στα στοιχεία μιας λίστας (το περιεχόμενο των κόμβων δηλαδή) τη διαδικασία με όνομα Process. Ο αλγόριθμος που ακολουθεί υλοποιεί την πράξη της διαπέρασης:

   Traversal(H)

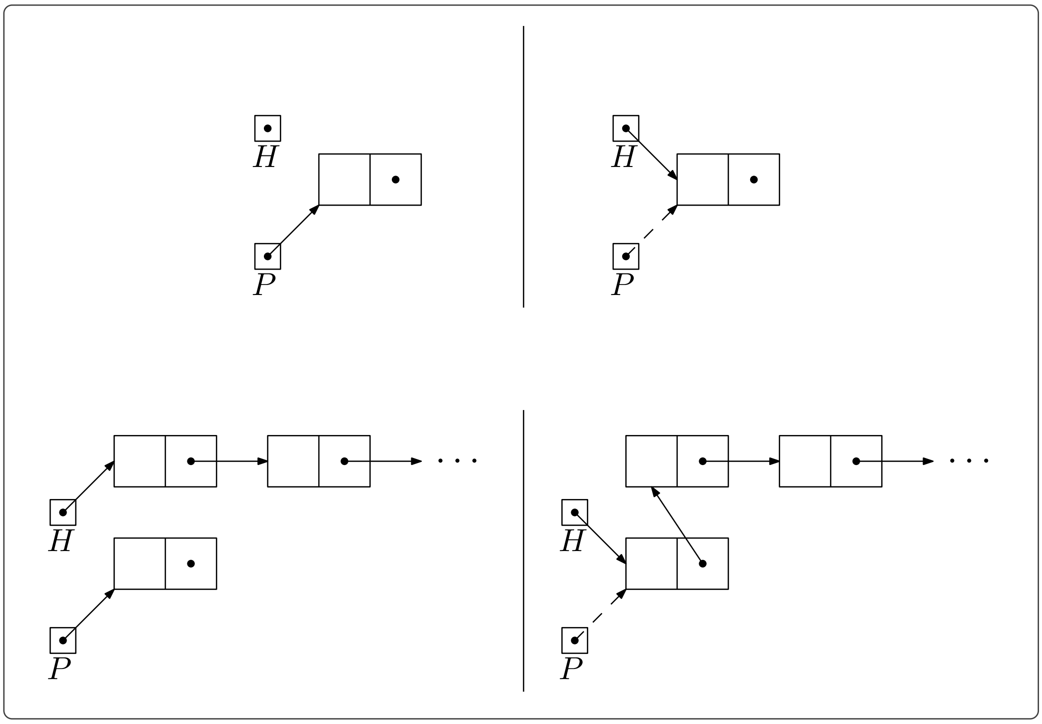
   Είσοδος: Ο δείκτης κεφαλής H μιας (απλά συνδεδεμένης) λίστας
   Έξοδος : Τίποτα (εσωτερικά ο αλγόριθμος εφαρμόζει την Process σε κάθε στοιχείο της λίστας)

   ```
   P ← H
   while P ≠ NIL                    %Όσο δεν έχουμε φτάσει στο τέλος της λίστας
       Process(node(P).item)
       P ← node(P).next
   ```

   Έστω ότι η λίστα περιέχει n κόμβους. Ο χρόνος που χρειάζεται αυτός ο αλγόριθμος είναι $O(n \cdot T(n))$ όπου $T(n)$ ο χρόνος που χρειάζεται η Process [1].

4. *Εισαγωγή στοιχείου:* Θα τη χωρίσουμε σε δύο ξεχωριστούς αλγόριθμους. Ο πρώτος θα κάνει εισαγωγή στην αρχή της λίστας (και σε κενή λίστα) και ο δεύτερος θα κάνει εισαγωγή μετά από συγκεκριμένο κόμβο της λίστας.

[1] Λογικά ο χρόνος της Process θα είναι σταθερός καθώς εφαρμόζεται σε ένα μόνο στοιχείο της λίστας. Αν θέλαμε να εφαρμόσουμε μία πιο πολύπλοκη διαδικασία στη λίστα μας, καλό θα ήταν να την «ενσωματώσουμε» στον αλγόριθμο Traversal.



Σχήμα 2.1.5: *Αριστερά βλέπουμε τη λίστα πριν την εισαγωγή και δεξιά τη λίστα μετά την εισαγωγή. Στην πρώτη περίπτωση η λίστα είναι κενή. Παρατηρήστε ότι ο δείκτης P εξακολουθεί να δείχνει στον κόμβο που εισάγαμε αλλά αυτό δεν θα μας απασχολεί.*

InsertAtHead(P, H)

Είσοδος: Ο δείκτης κεφαλής H μιας (απλά συνδεδεμένης) λίστας και ο δείκτης P του προς εισαγωγή κόμβου [1]

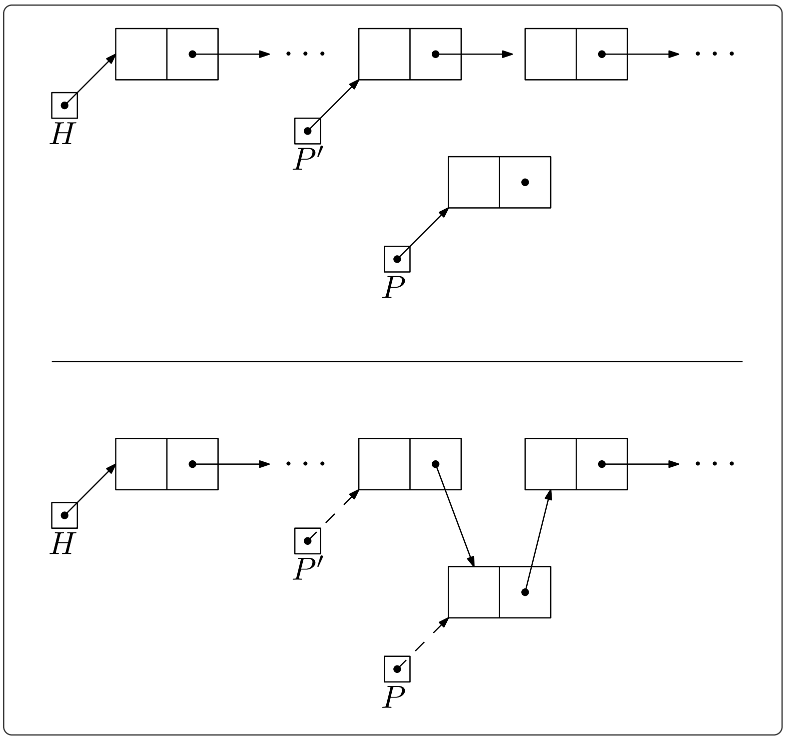
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

if $P =$ NIL **then** *%Αν ο P δεν δείχνει σε κόμβο*
    return "Σφάλμα"
else
    node$(P).next \leftarrow H$ *%NIL αν η λίστα είναι κενή*
    $H \leftarrow P$

Στο Σχήμα 2.1.5 παρουσιάζεται σχηματικά το πως γίνεται η εισαγωγή ενός κόμβου στην αρχή μιας λίστας (κενής και μη-κενής). Ο χρόνος του InsertAtHead είναι σταθερός.

Ο ακόλουθος αλγόριθμος υλοποιεί την εισαγωγή *μετά* από συγκεκριμένο κόμβο της

[1] Προς αποφυγήν παρανοήσεων τονίζουμε το γεγονός ότι ο δείκτης αυτός δεν είναι ο δείκτης επόμενου του κόμβου, αλλά ένας δείκτης που «δείχνει» τον προς εισαγωγή κόμβο. Στην ουσία δεχόμαστε ως είσοδο τη διεύθυνση της θέσης μνήμης που είναι αποθηκευμένος ο προς εισαγωγή κόμβος.



Σχήμα 2.1.6: *Πάνω βλέπουμε τη λίστα πριν την εισαγωγή και κάτω τη λίστα μετά την εισαγωγή. Η εισαγωγή γίνεται μετά τον κόμβο που δείχνει ο δείκτης P'.*

λίστας. Σε αυτήν την περίπτωση στην ουσία παρεμβάλουμε τον προς εισαγωγή κόμβο μεταξύ δύο κόμβων [1] (δες Σχήμα 2.1.6).

InsertAfter(P, P')

Είσοδος: Ο δείκτης P' που δείχνει τον κόμβο μετά από τον οποίο θέλουμε να εισάγουμε τον κόμβο που δείχνει ο P
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

if $P =$ NIL **or** $P' =$ NIL **then** *%Αν ο P ή ο P' δεν δείχνουν σε κόμβο*
  return "Σφάλμα"
else
  node$(P).next \leftarrow$ **node**$(P').next$ *%Αν ο P' δείχνει στον τελευταίο κόμβο τότε* **node**$(P').next = NIL$
  node$(P').next \leftarrow P$

Και σε αυτή την περίπτωση η εισαγωγή χρειάζεται σταθερό χρόνο.

[1] Αν η εισαγωγή γίνεται μετά από τον τελευταίο κόμβο της λίστας προφανώς δεν υπάρχει δεύτερος κόμβος για να έχουμε «σωστή» παρεμβολή, αλλά η φιλοσοφία του αλγορίθμου παραμένει ίδια.

Προτού παρουσιάσουμε τις υπόλοιπες λειτουργίες των συνδεδεμένων λιστών ίσως ήταν χρήσιμο να δούμε ένα παράδειγμα εφαρμογής όσων έχουμε δει μέχρι τώρα. Θα δούμε πως δημιουργούμε μία λίστα ακέραιων και πως τη «γεμίζουμε».

Παράδειγμα 2.1.3. Οι παρακάτω γραμμές κώδικα δημιουργούν μία λίστα που περιέχει τους ακέραιους από το 0 μέχρι το n:

1 $H \leftarrow$ **new list**
2 $x \leftarrow$ **new node**(0)
3 $P \leftarrow$ **address**(x)
4 InsertAtHead(P, H)
5 $P' \leftarrow H$
6 **for** $i \leftarrow 1$ **to** n
7     $x \leftarrow$ **new node**(i)
8     $P \leftarrow$ **address**(x)
9   InsertAfter(P, P')
10     $P' \leftarrow P$ [1]

Αν τώρα θεωρήσουμε πως η διαδικασία Process διπλασιάζει την τιμή του στοιχείου:

Process(x)

Είσοδος: Ακέραιος x
Έξοδος : Τίποτα (εσωτερικά διπλασιάζεται η τιμή του x)

1 $x \leftarrow 2 \cdot x$

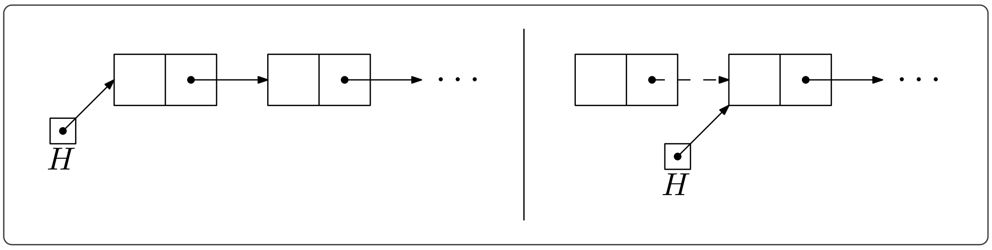
και προσθέσουμε στο τέλος του παραπάνω κώδικα την εντολή Traversal(H) θα προκύψει η λίστα $0, 2, 4, 6, \ldots, 2n$. Σε αυτήν την περίπτωση η Traversal θα χρειαστεί γραμμικό χρόνο ως προς το πλήθος των στοιχείων καθώς η Process χρειάζεται χρόνο $O(1)$.

4. *Διαγραφή στοιχείου:* Και τη διαγραφή στοιχείου θα τη χωρίσουμε σε δύο περιπτώσεις. Πρώτα θα δούμε πως διαγράφουμε τον πρώτο κόμβο στη λίστα και έπειτα πως διαγράφουμε τον κόμβο που βρίσκεται *μετά* από συγκεκριμένο κόμβο της λίστας.

   Αν θέλουμε να διαγράψουμε τον πρώτο κόμβο της λίστας αρκεί να αλλάξουμε τον δείκτη κεφαλής H ώστε να δείχνει τον δεύτερο κόμβο της λίστας (δες Σχήμα 2.1.7). Έχουμε ήδη αναφέρει πως η σωστή πρακτική θα ήταν μετά τη διαγραφή του κόμβου να αδειάζουμε και τη θέση μνήμης που τον περιέχει (περιλαμβάνοντας στη γλώσσα μας σχετικές εντολές). Δεν θα μας απασχολήσει όμως αυτό [2]. Ας δούμε τον αλγόριθμο:

[1] Εδώ κάνουμε μια μικρή παρατυπία. Είπαμε ότι ο δείκτης P μετά την εισαγωγή μας είναι άχρηστος (δες Σχήμα 2.1.6) εδώ όμως τον χρησιμοποιούμε... Για να είμαστε πιστοί στα λεγόμενά μας θα έπρεπε να κρατήσουμε την τιμή του P σε έναν άλλον δείκτη, έστω στον P'', και στη γραμμή 10 να είχαμε $P' \leftarrow P''$. Θα υποπέσουμε στην παρατυπία αυτή και άλλες φορές στο μέλλον.

[2] Ο κόμβος θα συνεχίσει να υπάρχει και ο δείκτης επομένου του θα συνεχίσει να δείχνει την καινούρια κεφαλή της λίστας.



Σχήμα 2.1.7: *Αριστερά βλέπουμε τη λίστα πριν την διαγραφή και δεξιά τη λίστα μετά την διαγραφή.*

DeleteFirst(H)

Είσοδος: Ο δείκτης κεφαλής H μιας (απλά συνδεδεμένης) λίστας
Έξοδος : Τίποτα (εσωτερικά γίνεται η διαγραφή του πρώτου κόμβου)

$H \leftarrow \mathbf{node}(H).next$

Παρατηρήστε ότι αν η λίστα περιέχει έναν μόνο κόμβο τότε μετά τη διαγραφή του ο δείκτης κεφαλής θα έχει τιμή NIL (όπως έχει και κατά την αρχικοποίηση της λίστας), συνεπώς το αποτέλεσμα θα είναι μία κενή λίστα. Ο χρόνος διαγραφής του πρώτου στοιχείου μιας λίστας είναι $O(1)$.

Ας υποθέσουμε τώρα ότι μας δίνουν έναν δείκτη που δείχνει στον *προηγούμενο* κόμβο από αυτόν που θέλουμε να διαγράψουμε. Οι ενέργειες που πρέπει να κάνουμε σε αυτήν την περίπτωση είναι να αλλάξουμε τον δείκτη επομένου αυτού του κόμβου ώστε να δείχνει τον μεθεπόμενο κόμβο (αν φυσικά υπάρχει), «παρακάμπτοντας» έτσι τον κόμβο που θέλουμε να διαγράψουμε (δες Σχήμα 2.1.8).

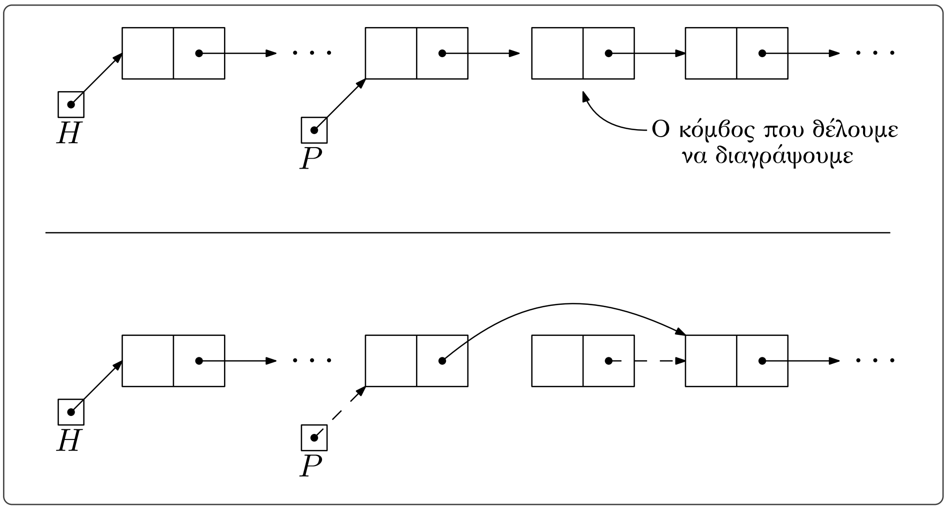
DeleteNext(P)

Είσοδος: Ο δείκτης P που δείχνει τον προηγούμενο κόμβο από αυτόν που θα διαγράψουμε
Έξοδος : Τίποτα (εσωτερικά γίνεται η διαγραφή)

if $P =$ NIL **or** $\mathbf{node}(P).next =$ NIL **then** *%Δεν υπάρχει κόμβος να διαγράψουμε*
    return “Σφάλμα”
else
    $\mathbf{node}(P).next \leftarrow \mathbf{node}(\mathbf{node}(P).next).next$ *%NIL αν ο P δείχνει τον προτελευταίο κόμβο*

Και αυτού του τύπου η διαγραφή χρειάζεται χρόνο $O(1)$.

Παράδειγμα 2.1.4. Ας δούμε ακόμα ένα παράδειγμα διαγραφής στοιχείου από συνδεδεμένη λίστα. Υποθέστε ότι θέλουμε να διαγράψουμε το k-οστό στοιχείο της λίστας για κάποιο



Σχήμα 2.1.8: *Πάνω βλέπουμε τη λίστα πριν την διαγραφή και κάτω τη λίστα μετά την διαγραφή.*

φυσικό αριθμό $k > 0$ (για $k = 1$ έχουμε διαγραφή της κεφαλής της λίστας). Ο αλγόριθμος που ακολουθεί υλοποιεί αυτήν τη διαγραφή.

kDelete(H, k)

Είσοδος: Ο δείκτης κεφαλής H μιας (απλά συνδεδεμένης) λίστας και ακέραιος k
Έξοδος : Τίποτα (εσωτερικά γίνεται η διαγραφή του k-οστού στοιχείου της λίστας, εφόσον αυτό υπάρχει)

if $k = 1$ **then** *%Διαγραφή πρώτου στοιχείου*
  DeleteFirst(H)
else
  $P \leftarrow H, i \leftarrow 1$
  while $P \neq$ NIL **and** $i \leq k - 2$ *%Θέλουμε να σταματήσουμε στον $(k-1)$-οστό κόμβο*
    $P \leftarrow$ **node**$(P).next$
    $i \leftarrow i + 1$
  DeleteNext(P) *%Αν η λίστα έχει $\leq k - 1$ στοιχεία θα έχουμε $P = NIL$ και ο DeleteNext θα επιστρέψει "Σφάλμα"*

Ο kDelete (στη χειρότερη περίπτωση) θα χρειαστεί να διασχίσει ολόκληρη τη λίστα (καθώς μπορεί να ζητηθεί να διαγραφεί το στοιχείο που βρίσκεται στην τελευταία θέση), συνεπώς ο χρόνος που χρειάζεται είναι $O(n)$, όπου n το πλήθος στοιχείων της λίστας [1].

[1] Με αντίστοιχο τρόπο θα υλοποιήσαμε τη διαγραφή συγκεκριμένου κόμβου από τη λίστα. Και σε αυτή την περίπτωση θα πρέπει να βρούμε τον προηγούμενο κόμβο για να εφαρμόσουμε σε αυτόν τον αλγόριθμο DeleteNext.

5. *Αναζήτηση στοιχείου:* Για να αναζητήσουμε ένα στοιχείο σε μια απλά συνδεδεμένη λίστα θα πρέπει να εφαρμόσουμε γραμμική αναζήτηση (δες Παράδειγμα 0.4.3), ακόμα και στην περίπτωση που έχουμε ταξινομημένη λίστα [1].Έτσι η χρονική πολυπλοκότητα του αλγορίθμου αναζήτησης θα είναι γραμμικός ($O(n)$ σε λίστα με n στοιχεία).

Find(H, x)

Είσοδος: Ο δείκτης κεφαλής H μιας (απλά συνδεδεμένης) λίστας και στοιχείο x
Έξοδος : Η διεύθυνση στη μνήμη που είναι αποθηκευμένος ο κόμβος που περιέχει το x

$P \leftarrow H$
while $P \neq$ NIL **and** **node**$(P).item \neq x$
  $P \leftarrow$ **node**$(P).next$
if $P =$ NIL **then** *% «Τελείωσε» η λίστα και δεν το βρήκαμε*
  return "Δεν υπάρχει"
else
  return P

Αν το x περιέχεται σε παραπάνω από έναν κόμβο της λίστας ο Find επιστρέφει την πρώτη εμφάνιση, ενώ όταν το x δεν εμφανίζεται καθόλου στη λίστα επιστρέφει μήνυμα «σφάλματος».

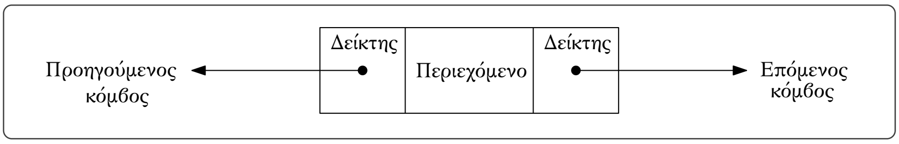
2.2 Άλλοι τύποι συνδεδεμένων λιστών

Σε αυτή την παράγραφο θα δούμε τους δύο βασικότερους τύπους συνδεδεμένης λίστας, πέρα από τις απλά συνδεδεμένες.

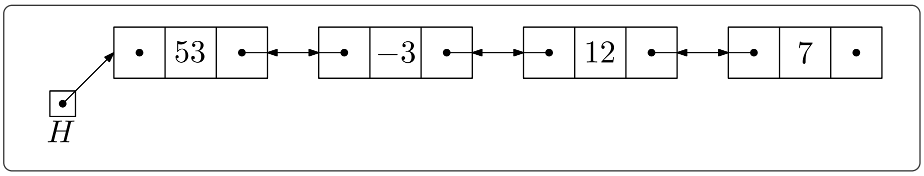
2.2.1 Διπλά συνδεδεμένες λίστες

Όπως υποδεικνύει και το όνομα, στις *διπλά συνδεδεμένες λίστες*, πέρα από τον δείκτη επόμενου κόμβου, κάθε κόμβος περιέχει και ένα δείκτη που δείχνει στον προηγούμενο κόμβο. Τον δείκτη αυτόν θα τον αποκαλούμε *δείκτη προηγούμενου* (δες Σχήμα 2.2.1). Πέραν αυτών των δεικτών θα χρησιμοποιήσουμε και εδώ έναν επιπλέον δείκτη που θα δείχνει την κεφαλή της λίστας. Καθώς η κεφαλή της λίστας δεν έχει προηγούμενο κόμβο η τιμή του δείκτη προηγούμενού της θα είναι NIL. Για να μπορέσουμε να χρησιμοποιήσουμε την τιμή του δείκτη προηγούμενου θα χρειαστεί να προσθέσουμε την ακόλουθη εντολή στη γλώσσα μας:

[1] Για να εφαρμόσουμε δυαδική αναζήτηση θα χρειαστεί να ξέρουμε εξ αρχής το μέγεθος της λίστας για να βρούμε στη συνέχεια το μεσαίο στοιχείο κ.λπ.. Για να μετρήσουμε όμως τους κόμβους της λίστας θα χρειαστεί να κάνουμε διαπέραση, οπότε τελικά δεν θα έχουμε κάποια βελτίωση στον χρόνο. Ακόμα όμως και αν παίρναμε αυτήν την πληροφορία «τζάμπα» (προσθέτοντας π.χ. έναν ακέραιο στον ορισμό της λίστας που θα κρατάει το μήκος της), θα έπρεπε να κάνουμε σειριακή προσπέλαση για να φτάσουμε στον μεσαίο κόμβο. Οπότε ο χρόνος και πάλι θα ήταν γραμμικός (αφού $n/2 = O(n)$).



Σχήμα 2.2.1: *Παράδειγμα κόμβου μιας διπλά συνδεδεμένης λίστας.*



Σχήμα 2.2.2: *Παράδειγμα διπλά συνδεδεμένης λίστας που περιέχει τους ακέραιους* $53, -3, 12, 7$.

- $x.previous$: Αναφορά στον δείκτη προηγούμενου κόμβου του κόμβου με όνομα x.

Για λόγους πληρότητας θα εισάγουμε και μία διαφορετική εντολή για τη δημιουργία κόμβου διπλά συνδεδεμένης λίστας:

- $x \leftarrow$**new double node**(y): Αρχικοποίηση μεταβλητής τύπου κόμβου διπλά συνδεδεμένης λίστας με περιεχόμενο y και τιμή των δύο δεικτών (επομένου και προηγούμενου) NIL,

Στο Σχήμα 2.2.2 φαίνεται ένα παράδειγμα διπλά συνδεδεμένης λίστας. Οι λειτουργίες που υποστηρίζουν οι διπλά συνδεδεμένες λίστες είναι ίδιες με αυτές των απλά συνδεδεμένων, μόνο που πλέον έχουμε περισσότερες δυνατότητες να «κινηθούμε» πάνω στην λίστα.

Λειτουργίες διπλά συνδεδεμένων λιστών

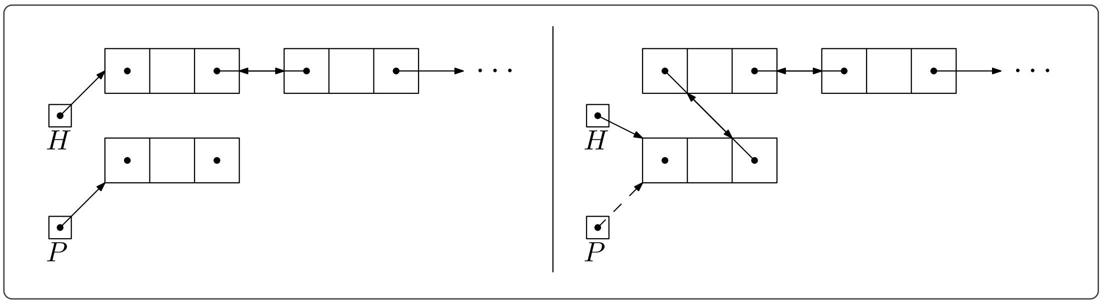
1. *Αρχικοποίηση λίστας:* Η αρχικοποίηση μιας διπλά συνδεδεμένης λίστας γίνεται ακριβώς με τον ίδιο τρόπο με τις απλά συνδεδεμένες λίστες, για να τονίσουμε όμως ότι η λίστα είναι διπλά συνδεδεμένη θα γράφουμε:

   $H \leftarrow$ **new double list**

2. *Έλεγχος για το αν η λίστα είναι κενή:* Πάλι ελέγχουμε αν ο δείκτης κεφαλής έχει τιμή NIL (ακριβώς όπως στον IsEmpty).

3. *Διαπέραση λίστας:* Η διαπέραση μπορεί να γίνει με δύο τρόπους: Από την κεφαλή προς το τέλος της λίστας ή από έναν κόμβο προς την κεφαλή της λίστας [1]. Ο αλγόριθμος για το πρώτο είδος διαπέρασης είναι ακριβώς ίδιος με τον Traversal. Τον δεύτερο

[1] Μπορούμε φυσικά να κάνουμε και διαπέραση από έναν κόμβο προς το τέλος της λίστας. Ο αλγόριθμος είναι εντελώς αντίστοιχο με τον Traversal.



Σχήμα 2.2.3: *Αριστερά βλέπουμε τη λίστα πριν την εισαγωγή και δεξιά τη λίστα μετά την εισαγωγή.*

τρόπο τον υλοποιεί ο παρακάτω αλγόριθμος (ας υποθέσουμε ξανά ότι θέλουμε να εφαρμόσουμε τη διαδικασία Process στα στοιχεία της λίστας):

BackwardTraversal(P)

Είσοδος: Δείκτης P που δείχνει τον κόμβο από τον οποίο θα αρχίσει η *οπισθόδρομη διαπέραση*

Έξοδος : Τίποτα (εσωτερικά ο αλγόριθμος εφαρμόζει την Process στα στοιχεία της λίστας από την κεφαλή μέχρι και τον κόμβο που δείχνει ο P)

```
while P ≠ NIL                %Όσο δεν έχουμε φτάσει στη κεφαλή της λίστας
    Process(node(P).item)
    P ← node(P).previous
```

Ο χρόνος που χρειάζεται η οπισθόδρομη διαπέραση της λίστας είναι $O(n \cdot T(n))$ όπου $T(n)$ ο χρόνος που χρειάζεται η Process και n το πλήθος κόμβων.

4. *Εισαγωγή στοιχείου:* Όσον αφορά την εισαγωγή κόμβου στη λίστα υπάρχουν τρεις τρόποι: Οι δύο που είδαμε στις απλά συνδεδεμένες λίστες (στην αρχή της λίστας και *μετά* από κόμβο) και η εισαγωγή στοιχείου *πριν* από συγκεκριμένο κόμβο. Θα δούμε και τους τρεις αλγορίθμους καθώς υπάρχουν μερικές διαφορές από αυτούς των απλά συνδεδεμένων λιστών (λόγω του επιπλέον δείκτη που περιέχουν οι κόμβοι).

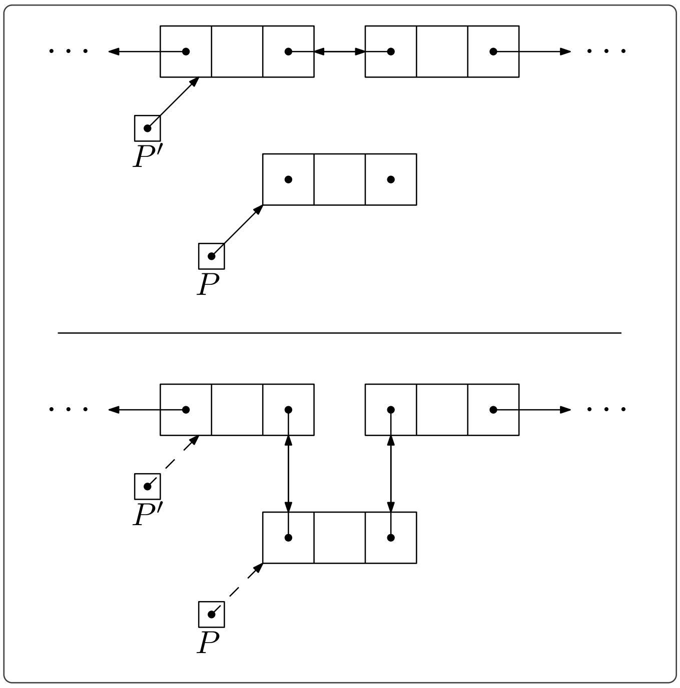
   Ας ξεκινήσουμε από την εισαγωγή κόμβου στην αρχή της λίστας (ή σε κενή λίστα). Μία σχηματική αναπαράσταση των αλλαγών που πρέπει να συμβούν στους δείκτες επόμενου και προηγούμενου των κόμβων παρουσιάζεται στο Σχήμα 2.2.3.

InsertAtHead(P, H)

Είσοδος: Ο δείκτης κεφαλής H μιας διπλά συνδεδεμένης λίστας και ο δείκτης P του προς εισαγωγή κόμβου

Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

Τελευταία ενημέρωση 28/1/2025, στις 09:53.



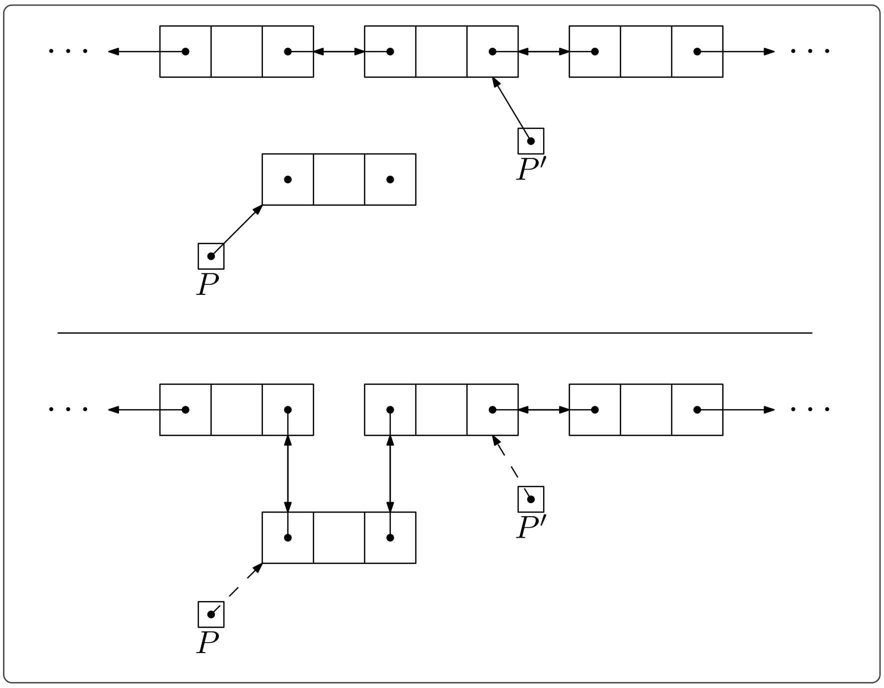
Σχήμα 2.2.4: *Πάνω βλέπουμε τη λίστα πριν την εισαγωγή και κάτω τη λίστα μετά την εισαγωγή.*

if $P =$ NIL **then** *%Αν ο P δεν δείχνει σε κόμβο*
    return "Σφάλμα"
else
    node$(P).next \leftarrow H$ *%NIL αν η λίστα είναι κενή*
    node$(P).previous \leftarrow$ NIL *%Μπορεί να μην έχει τιμή NIL*
    if not IsEmpty(H) **then** *%Αν η λίστα δεν είναι κενή*
        node$(H).previous \leftarrow P$
    $H \leftarrow P$

Η εισαγωγή *μετά* από στοιχείο παρόλο που έχει την ίδια φιλοσοφία με τον αλγόριθμο InsertAfter για τις απλά συνδεδεμένες λίστες, χρειάζεται μεγαλύτερη προσοχή. Το πρόβλημα είναι ότι πρέπει να προσέξουμε τη σειρά με την οποία θα αλλάξουμε τους δείκτες (δες Σχήμα 2.2.4).

InsertAfter(P, P')

Είσοδος: Ο δείκτης P' που δείχνει τον κόμβο μετά από τον οποίο θέλουμε να εισάγουμε τον κόμβο που δείχνει ο P
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)



Σχήμα 2.2.5: *Πάνω βλέπουμε τη λίστα πριν την εισαγωγή και κάτω τη λίστα μετά την εισαγωγή.*

if $P = \text{NIL}$ **or** $P' = \text{NIL}$ **then**
  return "Σφάλμα"
else
  node$(P).next \leftarrow$ **node**$(P').next$
  node$(P).previous \leftarrow P'$
  if node$(P').next \neq \text{NIL}$ **then** *%Αν κάνουμε εισαγωγή στο τέλος της λίστας δεν χρειάζεται*
    node(**node**$(P').next).previous \leftarrow P$
  node$(P').next \leftarrow P$

Τέλος, μπορούμε να εισάγουμε στοιχείο και *πριν* από δοσμένο κόμβο [1]. Πριν δούμε τον αλγόριθμο καλό θα ήταν να δούμε τη σχηματική αναπαράσταση αυτής της εισαγωγής (Σχήμα 2.2.5). Να τονίσουμε ότι δεν μπορούμε να χρησιμοποιήσουμε αυτόν τον τύπο εισαγωγής για να εισάγουμε στοιχείο στην αρχή της λίστας. Αυτό είναι δουλειά του αλγόριθμου InsertAtHead.

[1] Παρατηρήστε ότι στις απλά συνδεδεμένες λίστες για να το κάνουμε αυτό θα έπρεπε να βρούμε τον προ-προηγούμενο κόμβο κάνοντας διαπέραση της λίστας από την κεφαλή μέχρι τον δοσμένο κόμβο. Συνεπώς θα χρειαζόμασταν χρόνο $O(n)$, όπου n το μέγεθος της λίστας.

InsertBefore(P, P')

Είσοδος: Ο δείκτης P' που δείχνει τον κόμβο πριν από τον οποίο θέλουμε να εισάγουμε τον κόμβο που δείχνει ο P
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

if $P =$ NIL **or** $P' =$ NIL **then**
  return “Σφάλμα”
else if node$(P').previous =$ NIL **then** *%Άρα ο P' δείχνει στην κεφαλή*
  return “Εισαγωγή στην αρχή”
else
  node$(P).next \leftarrow P'$
  node$(P).previous \leftarrow$ **node**$(P').previous$
  node(**node**$(P').previous).next \leftarrow P$
  node$(P').previous \leftarrow P$

Ο χρόνος του και των τριών αλγορίθμων εισαγωγής κόμβου είναι σταθερός.

5. *Διαγραφή στοιχείου:* Μπορεί να γίνει με τέσσερις τρόπους. Στη διαγραφή της κεφαλής πάλι μετακινούμε τον δείκτη κεφαλής και «ξεχνάμε» την παλιά κεφαλή. Θα πρέπει όμως επιπλέον να αλλάξουμε τον δείκτη προηγούμενου στην καινούρια κεφαλή και να του δώσουμε την τιμή NIL.

DeleteFirst(H)

Είσοδος: Ο δείκτης κεφαλής H μιας διπλά συνδεδεμένης λίστας
Έξοδος : Τίποτα (εσωτερικά γίνεται η διαγραφή του πρώτου κόμβου)

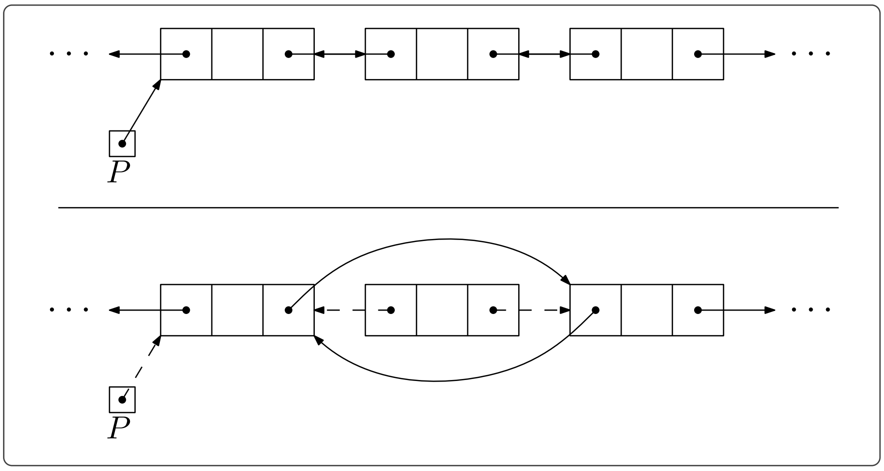
$H \leftarrow$ **node**$(H).next$
if H $\neq$ NIL **then**
  node$(H).previous \leftarrow$ NIL

Οι επόμενοι δύο τύποι διαγραφής είναι η διαγραφή του *επόμενου* κόμβου από τον δοσμένο (Σχήμα 2.2.6) και του *προηγούμενου.*

DeleteNext(P)

Είσοδος: Ο δείκτης P που δείχνει τον προηγούμενο κόμβο από αυτόν που θα διαγράψουμε
Έξοδος : Τίποτα (εσωτερικά γίνεται η διαγραφή)

if $P =$ NIL **or node**$(P).next =$ NIL **then**
  return “Σφάλμα”
else if node(**node**$(P).next).next =$ NIL **then** *%Διαγράφουμε τον τελευταίο κόμβο*
  node$(P).next \leftarrow$ NIL *%Απλά τον «ξεχνάμε»*



Σχήμα 2.2.6: *Πάνω βλέπουμε τη λίστα πριν τη διαγραφή και κάτω τη λίστα μετά τη διαγραφή.*

---

else
  node(**node**(**node**(P).$next$).$next$).$previous \leftarrow P$
  node(P).$next \leftarrow$ **node**(**node**(P).$next$).$next$

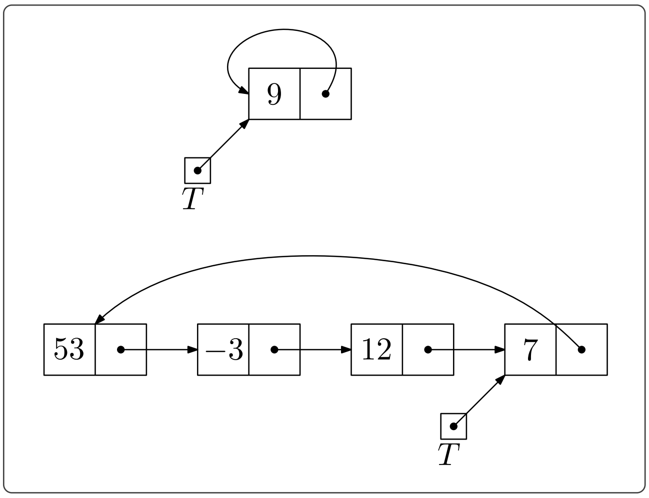
DeletePrevious(P)

Είσοδος: Ο δείκτης P που δείχνει τον επόμενο κόμβο από αυτόν που θα διαγράψουμε
Έξοδος : Τίποτα (εσωτερικά γίνεται η διαγραφή)

if $P =$ NIL **or** **node**(P).$previous =$ NIL **then**
  return "Σφάλμα"
else if **node**(**node**(P).$previous$).$previous =$ NIL **then** *%Δεν υπάρχει προηγούμενος κόμβος*
  return "Διαγραφή κεφαλής"
else
  node(**node**(**node**(P).$previous$).$previous$).$next \leftarrow P$
  node(P).$previous \leftarrow$ **node**(**node**(P).$previous$).$previous$

Τέλος, μπορούμε να διαγράψουμε και κάποιο συγκεκριμένο κόμβο. Αρκεί αν βρούμε τον προηγούμενό (ή τον επόμενο) κόμβο και να εφαρμόσουμε τον DeleteNext (ή τον DeletePrevious αντίστοιχα) για αυτόν τον κόμβο (οι λεπτομέρειες αφήνονται ως άσκηση). Ο αλγόριθμος αυτός, όπως και οι υπόλοιποι τρεις, χρειάζεται χρόνο $O(1)$ [1].

[1] Παρατηρήστε ότι αυτού του είδους η διαγραφή είναι δυνατή και με απλά συνδεδεμένες λίστες, χρειάζεται όμως γραμμικό χρόνο καθώς θα χρειαστεί να κάνουμε διαπέραση για να βρούμε τον προηγούμενο κόμβο.



Σχήμα 2.2.7: *Παράδειγμα δύο κυκλικά συνδεδεμένων λιστών. Η πρώτη από αυτές περιέχει μόνο ένα στοιχείο.*

6. *Αναζήτηση στοιχείου:* Δεν έχει καμία διαφορά από τις απλά συνδεδεμένες λίστες. Υλοποιείται από τον αλγόριθμο Find.

2.2.2 Κυκλικά συνδεδεμένες λίστες

Για πολλές εφαρμογές, καθώς και για την υλοποίηση πιο σύνθετων δομών δεδομένων (π.χ. για κάποιες *ουρές*), χρειαζόμαστε λίστες που είναι συνδεδεμένες *κυκλικά*, δηλαδή ο τελευταίος κόμβος της λίστας συνδέεται με τον πρώτο (δες Σχήμα 2.2.7). Συνέπεια αυτού είναι το γεγονός ότι στις *κυκλικά συνδεδεμένες λίστες* δεν θα υπάρχει δείκτης επόμενου που να έχει τιμή NIL [1].

Παρατηρήστε ότι πλέον είναι κάπως ασαφές ποιος είναι ο πρώτος κόμβος στη λίστα, καθώς οι κόμβοι είναι συνδεδεμένοι κυκλικά. Παρόλα αυτά (χάριν ευκολίας) εμείς θα συντηρούμε έναν δείκτη (ο δείκτης T στο Σχήμα 2.2.7) που σε αυτήν την περίπτωση θα δείχνει τον *τελευταίο* κόμβο της λίστας [2], τη λεγόμενη *ουρά της λίστας*.

Λειτουργίες κυκλικά συνδεδεμένων λιστών

Ας περάσουμε να δούμε πως υλοποιούνται οι έξι λειτουργίες στις κυκλικά συνδεδεμένες λίστες.

1. *Αρχικοποίηση λίστας:* Υλοποιείται πάλι αρχικοποιώντας έναν δείκτη στην τιμή NIL. Θα γράφουμε:

[1] Όσα θα πούμε σε αυτή την παράγραφο αφορούν απλά συνδεδεμένες λίστες, εύκολα όμως μεταφέρονται και σε διπλά συνδεδεμένες λίστες.

[2] Φυσικά ο πρώτος κόμβος της λίστας θα είναι ο επόμενός του.

$T \leftarrow$ **new circular list**

2. *Έλεγχος για το αν η λίστα είναι κενή:* Ελέγχουμε αν ο δείκτης ουράς έχει τιμή NIL.

3. *Διαπέραση λίστας:* Στις κυκλικά συνδεδεμένες λίστες μπορούμε να κάνουμε διαπέραση ξεκινώντας από οποιονδήποτε κόμβο, καθώς όμως δεν υπάρχει κόμβος με δείκτη επομένου NIL (που να σηματοδοτεί το τέλος της λίστας) θα πρέπει να «θυμόμαστε» από που ξεκινήσαμε για να ξέρουμε πότε πρέπει να σταματήσει η διαπέραση (υποθέτουμε κατά τα γνωστά ότι θέλουμε να εφαρμόσουμε τη διαδικασία Process στα στοιχεία της λίστας).

Traversal(P)

Είσοδος: Δείκτης P που δείχνει τον κόμβο από τον οποίο θα αρχίσει η διαπέραση
Έξοδος : Τίποτα (εσωτερικά ο αλγόριθμος εφαρμόζει την Process στα στοιχεία της λίστας)

Process(**node**$(P).item$)
$P' \leftarrow$ **node**$(P).next$
while $P' \neq P$ *%Όσο δεν φτάσαμε ξανά εκεί απ' όπου ξεκινήσαμε*
  Process(**node**$(P').item$)
  $P' \leftarrow$ **node**$(P').next$

Ο χρόνος της διαπέραση μιας κυκλικά συνδεδεμένης λίστας με n κόμβους είναι $O(n \cdot T(n))$, όπου $T(n)$ ο χρόνος της Process.

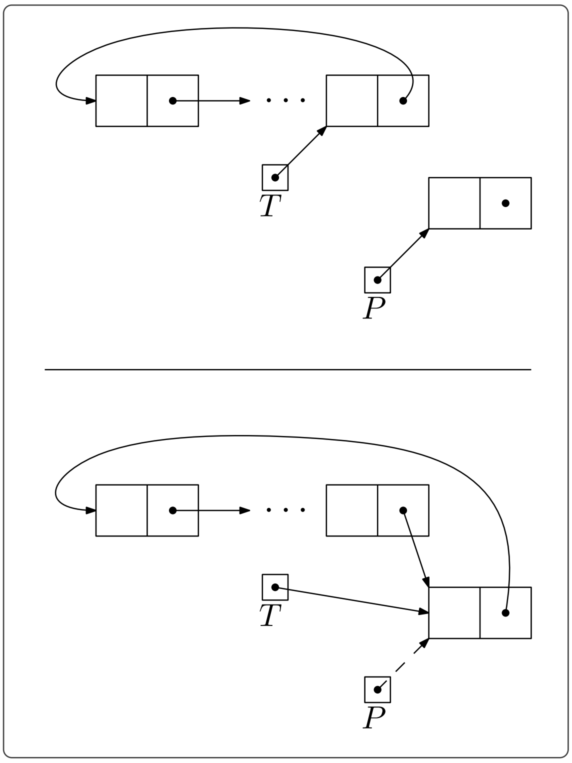
4. *Εισαγωγή στοιχείου:* Χωρίζουμε την εισαγωγή σε τρεις αλγορίθμους: εισαγωγή σε *κενή λίστα*, *μετά* από κόμβο και *στην ουρά* της λίστας. Ο πρώτος και ο τρίτος αλγόριθμος έχουν κάποιες λεπτομέρειες που θα πρέπει να προσέξουμε, παρόλο που στην ουσία αντιστοιχούν στον αλγόριθμο InsertAtHead και τον InsertAfter των απλά συνδεδεμένων λιστών.

   Η εισαγωγή σε *κενή λίστα* έχει την ιδιαιτερότητα ότι, καθώς η λίστα περιέχει μόνο έναν κόμβο ο δείκτης επόμενού του θα πρέπει να δείχνει στον εαυτό του. Ένα παράδειγμα λίστας με μόνο ένα στοιχείο φαίνεται στο Σχήμα 2.2.7

InsertAtEmptyList(P, T)

Είσοδος: Ο δείκτης ουράς T μιας άδειας κυκλικά συνδεδεμένης λίστας και ο δείκτης P του προς εισαγωγή κόμβου
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

if $P =$ NIL **then**
  return "Σφάλμα"



Σχήμα 2.2.8: *Πάνω βλέπουμε τη λίστα πριν την εισαγωγή και κάτω τη λίστα μετά την εισαγωγή.*

---

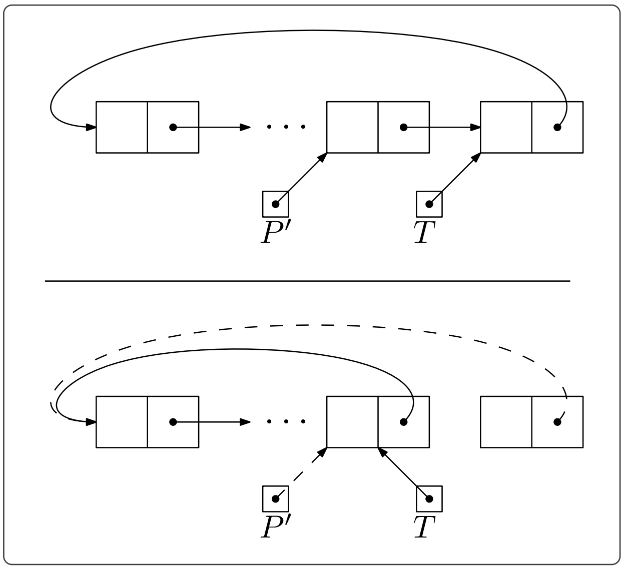
else if $T = \mathrm{NIL}$ **then** *%Αν η λίστα είναι όντως κενή*
  $T \leftarrow P$
  node$(P).next \leftarrow P$
else
  return "Μη κενή λίστα"

Η εισαγωγή *μετά* από κόμβο υλοποιείται από τον αλγόριθμο InsertAfter για τις απλά συνδεδεμένες λίστες.

Για την εισαγωγή κόμβου στην *ουρά* της λίστας αρκεί να εισάγουμε τον κόμβο μετά από τον κόμβο που δείχνει ο δείκτης ουράς, και έπειτα να διορθώσουμε τον δείκτη αυτόν ώστε να δείχνει τον καινούργιο τελευταίο κόμβο (δες Σχήμα 2.2.8).

InsertAtTail(P, T)

Είσοδος: Ο δείκτης ουράς T μιας κυκλικά συνδεδεμένης λίστας και ο δείκτης P του προς εισαγωγή κόμβου
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)



Σχήμα 2.2.9: *Πάνω βλέπουμε τη λίστα πριν τη διαγραφή της ουράς και κάτω τη λίστα μετά τη διαγραφή.*

---

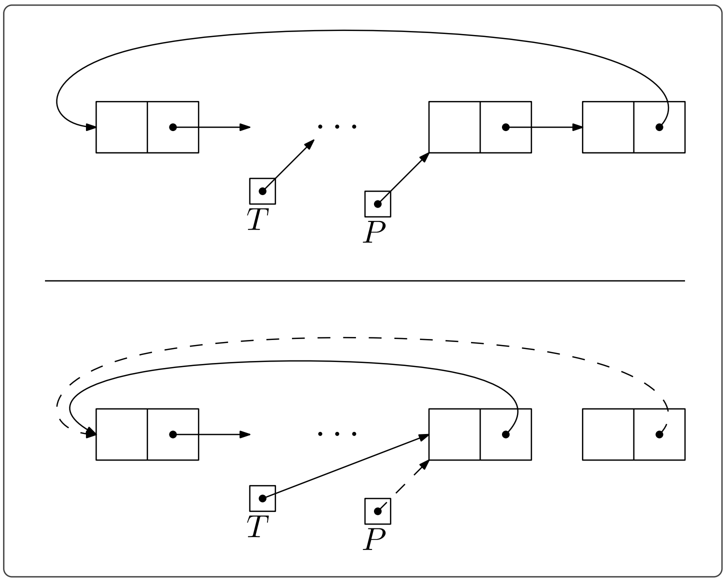
if $P = \text{NIL}$ **then**
  return "Σφάλμα"
else if $T = \text{NIL}$ **then**
  return "Κενή λίστα"
else
  InsertAfter(P, T) *%Ο αλγόριθμος για τις απλά συνδεδεμένες λίστες*
  $T \leftarrow P$

Ο χρόνος εισαγωγής κόμβου σε κυκλικά συνδεδεμένη λίστα είναι $O(1)$.

5. *Διαγραφή στοιχείου:* Η διαγραφή στοιχείου παρουσιάζει μία ιδιαιτερότητα που αφορά τη διαγραφή της ουράς. Όταν διαγράφουμε την ουρά πρέπει να ενημερώσουμε και τον δείκτη ουράς, που πλέον πρέπει να δείχνει στον (παλιό) προτελευταίο κόμβο της λίστας (τον προηγούμενο από αυτόν που έδειχνε ο δείκτης ουράς). Στις απλά κυκλικά συνδεδεμένες λίστες για να βρούμε αυτόν τον κόμβο θα χρειαστεί να κάνουμε διαπέραση [1]. Για να αποφύγουμε τη διαπέραση (που θα έκανε τη διαγραφή πολύ χρονοβόρα) θα μπορούσαμε να απαιτήσουμε να ξέρουμε εκ των προτέρων ποιος είναι ο προτελευταίος κόμβος της λίστας (δες Σχήμα 2.2.9).

   Με αυτόν τον τρόπο η διαγραφή της ουράς φαινομενικά χρειάζεται σταθερό χρόνο. Καθώς όμως μετά τη διαγραφή αλλάζει ο προτελευταίος κόμβος, και για να τον βρούμε

[1] Προφανώς στις διπλά κυκλικά συνδεδεμένες λίστες η διαπέραση δεν είναι απαραίτητη.



Σχήμα 2.2.10: *Ο δεύτερος τρόπος διαγραφής στοιχείου από την λίστα. Ο κόμβος που διαγράφεται είναι ο επόμενος από αυτόν που δείχνει ο P.*

σε μία απλά κυκλικά συνδεδεμένη λίστα ξανά θα πρέπει να κάνουμε διαπέραση, τελικά θα χρειαστούμε γραμμικό χρόνο.

Ένας πιο κομψός τρόπος (και σίγουρα πιο εύχρηστος) είναι να θεωρήσουμε ότι μπορούμε να διαγράψουμε μόνο τον *επόμενο* κόμβο από έναν δοσμένο κόμβο (με την DeleteNext για τις απλά συνδεδεμένες λίστες) και ότι ο δοσμένος κόμβος μετά τη διαγραφή θα χρίζεται αυτομάτως ουρά της λίστας, ακόμα και αν δεν ήταν ο προτελευταίος κόμβος (δες Σχήμα 2.2.10). Στην ουσία πάντα θα θεωρούμε ότι διαγράφουμε την ουρά, αδιαφορώντας ποια είναι η πραγματική της θέση. Γι' αυτό και τον αλγόριθμο που υλοποιεί αυτή την διαγραφή τον ονομάζουμε DeleteTail.

DeleteTail(P, T)

Είσοδος: Ο δείκτης P που δείχνει τον προηγούμενο κόμβο από αυτόν που θα διαγράψουμε και ο δείκτης ουράς T
Έξοδος : Τίποτα (εσωτερικά γίνεται η διαγραφή)

if $P = \text{NIL}$ **then**
  return "Σφάλμα"
else if **node**$(P).next = P$ **then** *%Έχουμε λίστα με μόνο ένα στοιχείο*
  $T \leftarrow \text{NIL}$
else
  DeleteNext(P)
  $T \leftarrow P$

Στην γραμμή 3 του κώδικα ξεχωρίζουμε την περίπτωση που η λίστα περιέχει μόνο έναν κόμβο (*άδειασμα λίστας*). Ο χρόνος του DeleteTail είναι $O(1)$.

6. *Αναζήτηση στοιχείου:* Μπορούμε να αρχίσουμε την αναζήτηση από οποιονδήποτε κόμβο μας δοθεί, προσέχοντας πότε θα τον επισκεφτούμε ξανά. Αυτό σηματοδοτεί το τέλος της αναζήτησης.

Find(P, x)

Είσοδος: Δείκτης P που δείχνει σε κόμβο μιας κυκλικά συνδεδεμένης λίστας και στοιχείο x

Έξοδος : Η διεύθυνση στη μνήμη που είναι αποθηκευμένος ο κόμβος που περιέχει το x [1]

if **node**$(P).item = x$ **then**
  return P
$P' \leftarrow$ **node**$(P).next$
while $P' \neq P$ **and** **node**$(P').item \neq x$
  $P' \leftarrow$ **node**$(P').next$
if $P' = P$ **then**
  return "Δεν βρέδηκε"
else
  return P'

Ο χρόνος αναζήτησης στοιχείου είναι γραμμικός ως προς το πλήθος κόμβων της λίστας.

Ασκήσεις

2.1. Θεωρήστε την ακόλουθη αναπαράσταση μίας λίστας με το πολύ n-στοιχεία:

- Τα στοιχεία της λίστας αποθηκεύονται σε έναν πίνακα L διάστασης n.
- Πέρα από τον πίνακα χρησιμοποιούμε και έναν ακέραιο δείκτη T με τιμή την θέση του πίνακα που βρίσκεται το τελευταίο στοιχείο της λίστας.

Για παράδειγμα έστω $n = 5$, τότε η λίστα:

κόκκινο

μαύρο

πράσινο

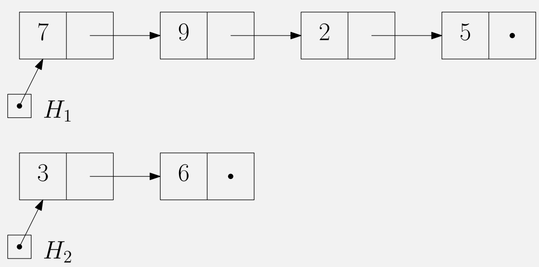
[1] Αν το x περιέχεται σε παραπάνω από έναν κόμβο τότε επιστρέφει την πρώτη εμφάνιση ενώ αν το x δεν εμφανίζεται καθόλου επιστρέφει μήνυμα σφάλματος.

αναπαρίσταται ως εξής: $L =$ [κόκκινο, μαύρο, πράσινο, _ , _] και $T = 3$. Να δώσετε τους αλγόριθμους που υλοποιούν τις ακόλουθες πράξεις:

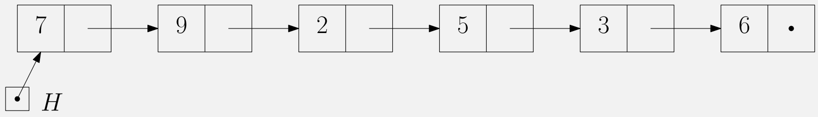
1. Διαπέραση των στοιχείων και εφαρμογή μίας διαδικασίας Process πάνω σε αυτά: Traversal(L, T)
2. Εισαγωγή στοιχείου σε δοσμένη θέση: InsertAtPosition(L, T, i, x), όπου i η θέση της λίστας που θα γίνει η εισαγωγή και x το στοιχείο που θα εισαχθεί. (Θα πρέπει να γίνεται και έλεγχος για το αν η λίστα είναι γεμάτη.)
3. Διαγραφή στοιχείου από δοσμένη θέση: DeletePosition(L, T, i), όπου i η θέση της λίστας από την οποία θα διαγραφεί το στοιχείο. (Θα πρέπει μετά την διαγραφή οι μόνες κενές θέσεις στον πίνακα να βρίσκονται στο τέλος του.)
4. Αναζήτηση στοιχείου και διαγραφή του (εφόσον βρεθεί): Find(L, T, x), όπου x το στοιχείο που αναζητούμε. (Αν δεν βρεθεί το στοιχείο x ο αλγόριθμος θα επιστρέφει το μήνυμα *Το στοιχείο δεν βρέθηκε.*)

2.2. Δώστε αλγόριθμο DeleteOdd(H) που δέχεται ως είσοδο την κεφαλή H μιας απλά συνδεδεμένης λίστα ακεραίων και διαγράφει τα στοιχεία που είναι περιττοί αριθμοί. Μπορείτε (εφόσον κρίνεται ότι σας είναι χρήσιμη) να χρησιμοποιήσετε την συνάρτηση mod που δέχεται ως είσοδο δύο ακεραίους m και n και επιστρέφει το υπόλοιπο της διαίρεσης του m με τον n (για παράδειγμα η τιμή mod$(1871, 9)$ είναι 8).

2.3. Δώστε αλγόριθμο Concatenation(H_1, H_2) που δέχεται σαν είσοδο τους δείκτες κεφαλής H_1 και H_2 δύο απλά συνδεδεμένων λιστών και δημιουργεί καινούρια λίστα που περιέχει τα στοιχεία και των δύο λιστών, σύμφωνα με την σειρά τους μέσα στις λίστες, ξεκινώντας από την λίστα με δείκτη κεφαλής H_1. Για παράδειγμα η κλήση Concatenation(H_1, H_2) όπου:

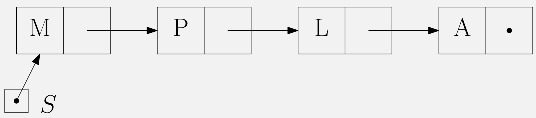


θα δημιουργήσει τη λίστα:

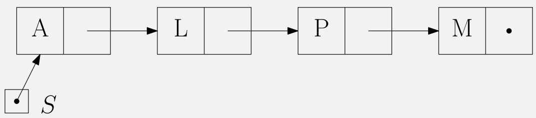


2.4. Δώστε αλγόριθμο Plus(P) που δέχεται ως είσοδο έναν δείκτη που «δείχνει» σε κόμβο μίας (απλά) κυκλικά συνδεδεμένης λίστας ακεραίων αριθμών και προσθέτει i στο i-οστο στοιχείο που θα επισκεφτεί (δηλαδή στο στοιχείο $\textbf{node}(P).item$ προσθέτει 1, στο στοιχείο $\textbf{node}(\textbf{node}(P).next).item$ προσθέτει 2 κ.ο.κ.) και σταματάει όταν βρεθεί στοιχείο με τιμή μεγαλύτερη είτε ίση από 1917. (Ο αλγόριθμος θα πρέπει να ελέγχει, στην ίδια επανάληψη, αν το i-οστό στοιχείο της λίστας είναι μεγαλύτερο από 1917 και αν το άθροισμά του με i ξεπερνάει το 1917.)

2.5. Δώστε αλγόριθμο StringReversal(S) που δέχεται ως είσοδο τον δείκτη κεφαλής S μίας λίστας με χαρακτήρες, ή αλλιώς *αλφαριθμητικό*, και την αντιστρέφει. Παραδείγματος χάρη αν δοθεί το αλφαριθμητικό $MPLA$:



η κλήση StringReversal(S) θα το μετατρέψει στο:



ΚΕΦΑΛΑΙΟ 3

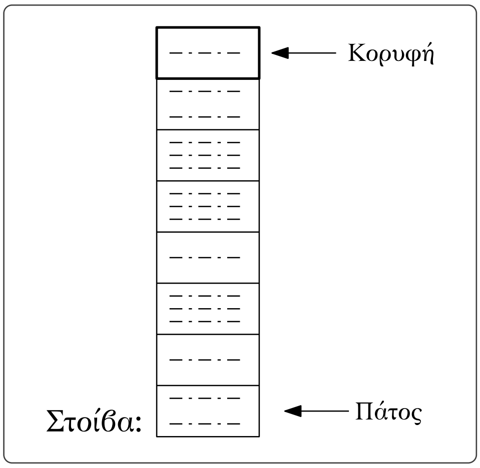
ΣΤΟΙΒΕΣ ΚΑΙ ΟΥΡΕΣ

Σε αυτό το κεφάλαιο θα θέσουμε περιορισμούς στον τρόπο που μπορούμε να εισάγουμε και να διαγράφουμε στοιχεία σε μία λίστα ή/και περιορισμούς ως προς σε ποια στοιχεία της λίστας θα έχουμε πρόσβαση. Ο λόγος που θα το κάνουμε αυτό είναι για να ορίσουμε μερικές «ειδικές» λίστες (*Στοίβες* και *Ουρές*) οι οποίες βρίθουν εφαρμογών. Το κύριο μέρος του κεφαλαίου θα καταναλωθεί για την υλοποίηση αυτών των δομών, θα δούμε όμως και μερικές από τις εφαρμογές τους προκειμένου να τις κατανοήσουμε καλύτερα και να συνειδητοποιήσουμε την αξία τους.

Ένα εύλογο ερώτημα είναι γιατί χρειάζεται να εισάγουμε νέες δομές δεδομένων που τις υλοποιούμε χρησιμοποιώντας τις βασικές δομές που είδαμε στα Κεφάλαια 2 και 3, και γιατί δεν αρκούμαστε μόνο στους πίνακες και στις λίστες. Η απάντηση είναι ότι πρέπει να τις εισάγουμε για «διευκόλυνση» του χρήστη. Μην ξεχνάτε ότι όσα βλέπουμε σε αυτές τις σημειώσεις αποτελούν ένα κομμάτι το οποίο δεν είναι φανερό στον χρήστη. Μελετάμε τον τρόπο που μία στοιχειώδης γλώσσα προγραμματισμού δύναται να υλοποιήσει τις δομές δεδομένων που είναι απαραίτητες για τον σχεδιασμό αλγορίθμων. Ο τρόπος που γίνεται αυτό (στην πλειοψηφία των περιπτώσεων) δεν απασχολεί καθόλου τον χρήστη.

3.1 Στοίβες

Η *στοίβα* είναι ένας ειδικός τύπος λίστας στον οποίο επιτρέπουμε την εισαγωγή και τη διαγραφή (ή την ανάκτηση) στοιχείου μόνο στο ένα άκρο της, την *κορυφή* της στοίβας (το άλλο άκρο θα το αποκαλούμε *πάτο* της στοίβας, δες Σχήμα 3.1.1). Αφού μπορούμε να εισάγουμε και να ανακτήσουμε στοιχείο μόνο από την κορυφή της λίστας οι αλλαγές σε μία στοίβα πάντα αφορούν το στοιχείο που μπήκε τελευταίο. Ακολουθούμε όπως λέμε *LIFO* λογική (Last In First Out). Ένα προσφιλές μας παράδειγμα στοίβας είναι η στοίβα με τα πλυμένα πιάτα σε ένα εστιατόριο ή η θήκη των κερμάτων που χρησιμοποιούν οι επαγγελματίες οδηγοί, οι διανομείς κ.λπ.. Και στα δύο αυτά παραδείγματα μπορούμε να εισάγουμε πιάτο (ή κέρμα) στην κορυφή της στοίβας και αντίστοιχα να πάρουμε πιάτο (ή κέρμα)



Σχήμα 3.1.1: *Τις στοίβες μπορούμε να τις φανταστούμε σαν λίστες οι οποίες ακολουθούν κατακόρυφη δομή (όπως συμβαίνει και με την δομή που περιγράφει η λέξη στοίβα στην πραγματικότητα), με την κορυφή φυσικά προς τα πάνω.*

μόνο από την κορυφή της στοίβας (τουλάχιστον με αυτόν τον τρόπο –στην περίπτωση των πιάτων– ελαχιστοποιούμε την πιθανότητα να κάνουμε κάποια ζημιά).

Μπορούμε να αναπαραστήσουμε μία στοίβα είτε μέσω ενός πίνακα (στατικά) είτε μέσω μιας λίστας (δυναμικά). Θα δούμε και τους δύο τρόπους αναπαράστασης ξεκινώντας από τους πίνακες.

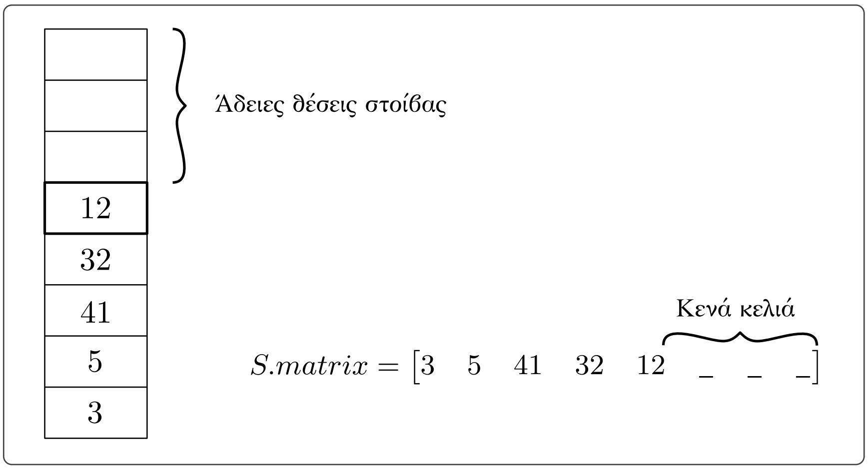
3.1.1 Αναπαράσταση μέσω πινάκων

Η βασική προϋπόθεση για να αναπαραστήσουμε μία στοίβα με έναν πίνακα είναι να γνωρίζουμε εξαρχής το μέγιστο πλήθος στοιχείων που θα χρειαστεί να εισάγουμε σε αυτήν. Όπως θα δούμε, αναπαριστώντας τις στοίβες με αυτόν τον τρόπο προκύπτουν κάποια ουσιαστικά μειονεκτήματα, καθώς πέρα από τον στατικό τρόπο αποθήκευσης που δεν μπορούμε να αποφύγουμε, ενδεχομένως να δεσμεύουμε θέσεις για κελιά που θα παραμείνουν άδεια για μεγάλο χρονικό διάστημα. Το αδιαμφισβήτητο όμως πλεονέκτημα που έχει αυτή η αναπαράσταση (και ίσως το μόνο) είναι η απλότητα.

Ξεκινάμε αρχικοποιώντας έναν «αρκετά μεγάλο» πίνακα. Θα χρειαστεί να κρατήσουμε ακόμα έναν (ακέραιο) δείκτη ο οποίος θα έχει ως τιμή τη θέση του πίνακα που βρίσκεται η κορυφή της στοίβας (δες Σχήμα 3.1.2). Τον δείκτη αυτόν θα τον αποκαλούμε *δείκτη κορυφής*. Θα κάνουμε την παραδοχή ότι τα στοιχεία του πίνακα που βρίσκονται μετά την κορυφή («πιο πάνω» από την κορυφή αν θέλουμε να το δούμε εποπτικά) είναι άδεια [1] και ότι αυτά θα είναι τα μόνα κενά κελιά του πίνακα που αναπαριστά την στοίβα.

Οι τέσσερις πράξεις που υποστηρίζει η δομή δεδομένων στοίβα είναι οι ακόλουθες:

[1] Στην πραγματικότητα ενδεχομένως να μην είναι άδεια, μπορούν να περιέχουν παλιές τιμές. Καθώς όμως αυτές οι τιμές δεν μας απασχολούν πλέον θα τα θεωρούμε άδεια.



Σχήμα 3.1.2: *Αναπαράσταση της στοίβας μέσω του πίνακα S.*

1. *Αρχικοποίηση στοίβας*
2. *Έλεγχος για το αν η στοίβα είναι κενή*
3. *Εισαγωγή στοιχείου (Push)*
4. *Εξαγωγή στοιχείου (Pop)*

Ας τις δούμε αναλυτικά:

1. *Αρχικοποίηση στοίβας:* Αφού μία στοίβα αποτελείται από τον συνδυασμό ενός πίνακα και ενός δείκτη αποτελεί σύνθετο τύπο δεδομένων. Έστω S μία μεταβλητή τύπου στοίβας, θα γράφουμε:

   - $S.matrix$ για να αναφερθούμε στον πίνακα της στοίβας,
   - $S.top$ για να αναφερθούμε στον δείκτη με τιμή τη θέση που βρίσκεται η κορυφή της στοίβας.

   Για παράδειγμα στη στοίβα του Σχήματος 3.1.2 η τιμή του $S.top$ είναι 5 (η πέμπτη θέση του πίνακα αντιστοιχεί στην κορυφή της στοίβας) και η τιμή της κορυφής δίνεται γράφοντας $S.matrix[S.top]$.

   Για να αρχικοποιήσουμε μια στοίβα χωρητικότητας n θα πρέπει να αρχικοποιήσουμε έναν πίνακα με n κελιά ($S.matrix \leftarrow$ **new matrix**(n)) και να δώσουμε στον δείκτη κορυφής την τιμή 0 ($S.top \leftarrow 0$), καθώς η στοίβα είναι κενή και ως εκ τούτου δεν υπάρχει στοιχείο που βρίσκεται στην κορυφή της. Για λόγους συνέπειας με τις προηγούμενες δομές δεδομένων που έχουμε δει, αντί για τις παραπάνω εντολές θα γράφουμε:

   $S \leftarrow$ **new stack**(n)

2. *Έλεγχος για το αν η στοίβα είναι κενή:* Αρκεί να ελέγξουμε αν η τιμή του δείκτη κορυφής $S.top$ ισούται με 0.

IsEmpty(S)

Είσοδος: Στοίβα S
Έξοδος : True αν η στοίβα είναι κενή, False διαφορετικά

if $S.top = 0$ **then**
  return True
else
  return False

Ο αλγόριθμος αυτός έχει σταθερό χρόνο.

3. *Εισαγωγή στοιχείου (Push):* Το μόνο που έχουμε να κάνουμε για αυτήν τη λειτουργία είναι να εισάγουμε το δοσμένο στοιχείο «πάνω» από την κορυφή της στοίβας, ελέγχοντας πρώτα αν η στοίβα είναι γεμάτη.

Push(x, S)

Είσοδος: Στοίβα S και στοιχείο x
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

if $S.top =$ **length**$(S.matrix)$ **then**   *%Η κορυφή είναι η τελευταία θέση του πίνακα* [1]
  return "Γεμάτη στοίβα"
else
  $S.top \leftarrow S.top + 1$
  $S.matrix[S.top] \leftarrow x$

Η πράξη της εισαγωγής χρειάζεται σταθερό χρόνο (εφόσον η **length** όπως είπαμε χρειάζεται σταθερό χρόνο).

4. *Εξαγωγή στοιχείου (Pop):* Κατά την εξαγωγή επιστρέφουμε την κορυφή και έπειτα τη διαγράφουμε. Η διαγραφή αυτή γίνεται «αγνοώντας» πια την ύπαρξη του στοιχείου που βρισκόταν πριν στην κορυφή. Σε αυτή την περίπτωση κάνοντας αυτήν την «τσαπατσουλιά» δεν πληρώνουμε κάτι παραπάνω όσον αφορά τον χώρο που καταναλώνουμε στη μνήμη, καθώς ούτως ή άλλως έχουμε στατική αποθήκευση και οι θέσεις μνήμης που καταναλώνουμε για να αποθηκεύσουμε τον πίνακα δεν αλλάζουν.

[1] Θα μπορούσαμε να προσθέσουμε έναν ξεχωριστό αλγόριθμο για να ελέγχουμε αν η στοίβα είναι γεμάτη. Δεν θα το χρειαστούμε όμως στη συνέχεια γι' αυτό ενσωματώνουμε τον έλεγχο στον Push. Εκτός αυτού, στην αναπαράσταση με λίστες δεν θα υπάρχει ο κίνδυνος να «γεμίσει» η στοίβα.

```
Pop(S)
  Είσοδος: Στοίβα S
  Έξοδος : Το στοιχείο στην κορυφή της στοίβας (αν υπάρχει, αλλιώς
           ενημερώνουμε τον χρήστη με σχετικό μήνυμα)
if IsEmpty(S) then
    return "Άδεια στοίβα"
else
    x ← S.matrix[S.top]
    S.top ← S.top − 1
    return x
```

Ο χρόνος της εξαγωγής είναι επίσης σταθερός.

3.1.2 Αναπαράσταση μέσω συνδεδεμένων λιστών

Υλοποιώντας τις στοίβες χρησιμοποιώντας (απλά) συνδεδεμένες λίστες απαλλασσόμαστε από την απαίτηση να ξέρουμε εξαρχής το μέγεθος της στοίβας που θα χρειαστούμε. Θα θεωρήσουμε ότι η κεφαλή της λίστας θα αποτελεί την κορυφή της στοίβας, οπότε ο δείκτης κεφαλής θα παίξει τον ρόλο του δείκτη κορυφής της στοίβας (δες Σχήμα 3.1.3). Μπορούμε να φανταστούμε ότι η λίστα αυξάνεται σε μέγεθος «προς τα κάτω» κατά την εισαγωγή στοιχείου, καθώς η εισαγωγή θα γίνεται στην κεφαλή της λίστας (με τον αλγόριθμο InseartAtHead).

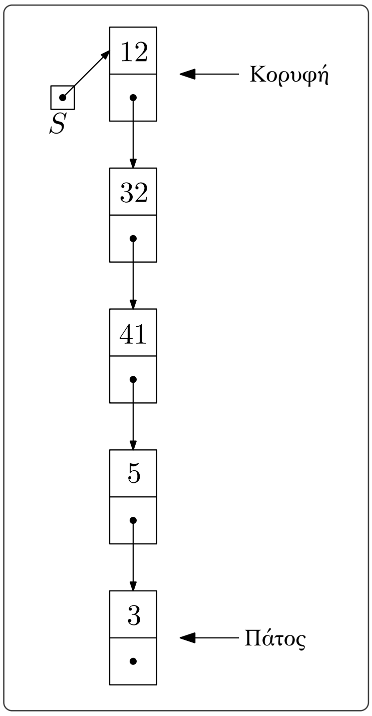
Ας περάσουμε στην υλοποίηση των τριών λειτουργιών της στοίβας:

1. *Αρχικοποίηση στοίβας:* Όπως κάναμε και με τις λίστες αρκεί να ορίσουμε μία μεταβλητή τύπου δείκτη, έστω S, και να της δώσουμε τιμή NIL. Για λόγους συμβατότητας θα γράφουμε:

   ```
   S ← new stack
   ```

2. *Έλεγχος για το αν η στοίβα είναι κενή:* Απλά ελέγχουμε αν η λίστα είναι κενή (δηλαδή αν ο δείκτης S έχει τιμή NIL). Ο IsEmpty για στοίβες είναι εντελώς αντίστοιχος με αυτόν για τις απλά συνδεδεμένες λίστες.

3. *Εισαγωγή στοιχείου (Push):* Αναφέρθηκε και πριν ότι θα κάνουμε εισαγωγή στην κεφαλή της λίστας. Αυτό που θέλει λίγο προσοχή είναι το γεγονός ότι θα μας δίνεται το στοιχείο που θέλουμε να εισάγουμε και όχι ένας δείκτη που δείχνει στη θέση μνήμης που περιέχει τον κόμβο με περιεχόμενο το στοιχείο αυτό (όπως γινόταν στις λίστες). Αυτό γίνεται γιατί όπως είπαμε ο χρήστης δεν θα πρέπει να απασχολείται με τέτοιου είδους λεπτομέρειες (θα τις κρατάμε «κρυφές»). Συνεπώς θα πρέπει πρώτα να δημιουργήσουμε έναν κόμβο που θα περιέχει το δοσμένο στοιχείο, να πάρουμε τη διεύθυνσή του και να τη βάλουμε σε έναν δείκτη P. Μετά από αυτήν την προεργασία μπορούμε να χρησιμοποιήσουμε τον InseartAtHead κατά τα γνωστά.



Σχήμα 3.1.3: *Αναπαράσταση της στοίβας μέσω συνδεδεμένης λίστας.*

Push(x, S)

Είσοδος: Στοίβα S και στοιχείο x
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

```
P ← address(new node(x))
InsertAtHead(P, S)
```

Ο χρόνος που χρειάζεται είναι σταθερός.

4. *Εξαγωγή στοιχείου (Pop):* Η εξαγωγή γίνεται εφαρμόζοντας τον DeleteFirst, αφού πρώτα φροντίσουμε να κρατήσουμε το περιεχόμενο της κεφαλής της λίστας για να το επιστρέψουμε.

Pop(S)

Είσοδος: Στοίβα S
Έξοδος : Το στοιχείο στην κορυφή της στοίβας (αν υπάρχει, αλλιώς ενημερώνουμε τον χρήστη με σχετικό μήνυμα)

```
if IsEmpty(S) then
    return "Άδεια στοίβα"
```

else
  $x \leftarrow$ **node**$(S).item$
  DeleteFirst(S)
  return x

Ο χρόνος εξαγωγής στοιχείου είναι επίσης σταθερός.

3.1.3 Εφαρμογές στοίβας

Θα δούμε δύο βασικές εφαρμογές στις οποίες οι στοίβες φαίνονται πολύ χρήσιμες. Παρατηρήστε ότι σε όσα ακολουθούν δεν θα μας απασχολεί ο τρόπος που έχει υλοποιηθεί η στοίβα που θα χρησιμοποιήσουμε. Θα μας ενδιαφέρει μόνο ότι πληροί τις «προδιαγραφές» μιας στοίβας.

Υπολογισμός αριθμητικών παραστάσεων

Έχουμε συνηθίσει να γράφουμε μία αριθμητική παράσταση χρησιμοποιώντας την *ενδοδεματική μορφή* όπου οι τελεστές των πράξεων βρίσκονται ανάμεσα από τα στοιχεία πάνω στα οποία εφαρμόζονται. Παραδείγματος χάρη γράφουμε:

$$\alpha+\beta, \;\; (\alpha+\beta)\cdot\gamma, \;\; \alpha\cdot\beta+\gamma\cdot\delta$$

Άλλος ένας τρόπος γραφής είναι η *προδεματική ή πολωνική μορφή* όπου οι τελεστές προηγούνται των στοιχείων πάνω στα οποία εφαρμόζονται:

$$+\alpha\beta, \;\; \cdot+\alpha\beta\gamma, \;\; +\cdot\alpha\beta\cdot\gamma\delta$$

Παρατηρήστε ότι σύμφωνα με αυτό τον τρόπο γραφής των παραστάσεων δεν χρειάζεται να χρησιμοποιήσουμε παρενθέσεις καθώς δεν υπάρχει περίπτωση αμφισημίας.

Τέλος, πολλοί μεταγλωτιστές γλωσσών προγραμματισμού χρησιμοποιούν τη *μεταδεματική (ή αντίστροφη πολωνική) μορφή* σύμφωνα με την οποία οι τελεστές γράφονται μετά:

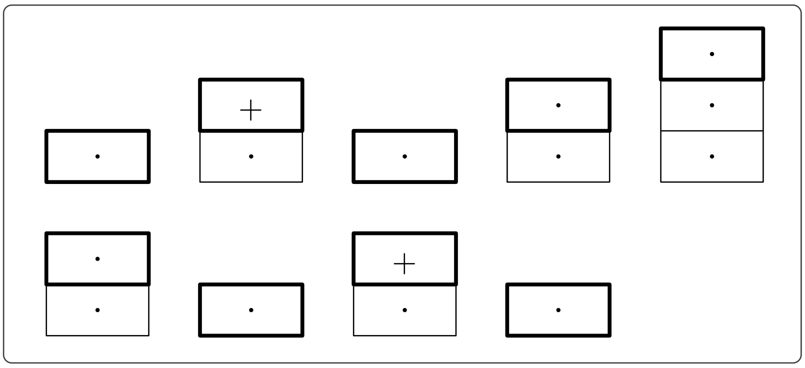
$$\alpha\beta+, \;\; \alpha\beta+\gamma\cdot, \;\; \alpha\beta\cdot\gamma\delta\cdot+$$

Εφόσον ο χρήστης εισάγει στον Η/Υ την παράσταση σε ενδοδεματική μορφή ενώ ο Η/Υ εσωτερικά χρησιμοποιεί τη μεταδεματική μορφή, ο υπολογισμός της παράστασης θα πρέπει να γίνει σε δύο στάδια:

1. Μετατροπή της ενδοδεματικής μορφής της παράστασης σε μεταδεματική μορφή.

2. Υπολογισμός της τιμής της παράστασης.

Και στα δύο αυτά στάδια γίνεται χρήση μίας στοίβας. Ας τα δούμε αναλυτικά.



Σχήμα 3.1.4: *Από τα αριστερά προς τα δεξιά βλέπουμε τις αλλαγές που θα προκύψουν στη στοίβα. Συνολικά θα γίνουν 5 εισαγωγές τελεστών και πέντε εξαγωγές.*

1. *Μετατροπή ενδοδεματικής μορφής σε μεταδεματική:* Η ιδέα του αλγορίθμου που υλοποιεί αυτήν τη μετατροπή είναι να διαβάζουμε την παράσταση (στην ενδοδεματική μορφή) από τα αριστερά προς τα δεξιά, αγνοώντας τις αριστερές παρενθέσεις, όποτε διαβάζουμε έναν αριθμό τον «τυπώνουμε» ενώ όταν διαβάζουμε έναν τελεστή τον εισάγουμε στη στοίβα μας. Όταν συναντήσουμε δεξιά παρένθεση θα εξάγουμε έναν τελεστή από τη στοίβα και θα τον «τυπώσουμε». Βασική προϋπόθεση για να δουλέψει αυτός ο αλγόριθμος είναι η παράσταση σε ενδοδεματική μορφή να περιέχει *όλες* της τις παρενθέσεις. Σε αντίθετη περίπτωση δεν θα μας επιστρέψει σωστό αποτέλεσμα [1]. Στο Σχήμα 3.1.4 βλέπουμε τα διάφορα στιγμιότυπα της στοίβας κατά τη μετατροπή της παράστασης $(5 \cdot (((9+8) \cdot (4 \cdot 6)) + 7))$ στην μεταδεματική της μορφή $5\ 9\ 8 + 4\ 6 \cdot \cdot 7 + \cdot$. Για λόγους απλότητας θα υποθέσουμε ότι έχουμε να αντιμετωπίσουμε μια αριθμητική παράσταση που περιέχει μόνο τους τελεστές $+$ και $\cdot$ και ότι η παράσταση μας δίνεται αποθηκευμένη σε έναν πίνακα [2]. Καθώς δεν έχουμε δηλώσεις τύπων στη γλώσσα μας ο μόνος τρόπος να ελέγξουμε αν το περιεχόμενο ενός κελιού του πίνακα είναι ακέραιος αριθμός είναι να ελέγξουμε την ισότητά του με ένα ακέραιο αριθμό. Αυτό φυσικά δεν θα ήταν εφικτό, έτσι για τους σκοπούς του παραδείγματος θα πρέπει να περιοριστούμε σε ένα πεπερασμένο πλήθος ακέραιων αριθμών. Ας υποθέσουμε λοιπόν ότι η παράστασή μας περιέχει μόνο αριθμούς στο σύνολο $\{0, 1, 2, \ldots, 9\}$.

InfixToPostfix(E)

Είσοδος: Πίνακας E που περιέχει μία αριθμητική παράσταση εκφρασμένη σε ενδοδεματική μορφή

[1] Όσον αφορά τις εξωτερικές παρενθέσεις δεν υπάρχει ουσιαστικό πρόβλημα, καθώς μπορούμε εύκολα να τροποποιήσουμε τον αλγόριθμο InfixToPostfix να κάνει μία επιπλέον εξαγωγή στο τέλος. Το πρόβλημα προκύπτει αν έχουμε παραλείψει παρενθέσεις εσωτερικά στην παράσταση.

[2] Αυτό θα βοηθήσει να αναγνωρίζουμε πότε έχουμε π.χ. διψήφιους αριθμούς και πότε δύο μονοψήφιους.

Έξοδος : Πίνακας E' που περιέχει την παράσταση εκφρασμένη σε μεταδεματική μορφή

$S \leftarrow$ **new stack**
$E' \leftarrow$ **new matrix**(**length**(E)) *%Ο E' θα έχει μικρότερο μέγεθος από τον E αλλά δεν ξέρουμε το ακριβές του μέγεθος εξαρχής* [1]
$j \leftarrow 1$
for $i \leftarrow 1$ **to** n
  if $E[i] = 0$ **or** $E[i] = 1$ **or** $\cdots$ **or** $E[i] = 9$ **then**
    $E'[j] \leftarrow E[i]$
    $j \leftarrow j + 1$
  else if $E[i] = +$ **or** $E[i] = \cdot$ **then**
    Push$(E[i], S)$
  else if $E[i] =)$ **then**
    $E'[j] \leftarrow$ Pop(S)
    $j \leftarrow j + 1$
return E'

Ο χρόνος για το πρώτο στάδιο υπολογισμού της αριθμητικής παράστασης είναι γραμμικός ως προς το πλήθος στοιχείων που περιέχει, καθώς στην ουσία κάνει μία απλή διαπέραση στον πίνακα.

2. *Υπολογισμός τιμής:* Η ιδέα είναι να διαβάσουμε πάλι την παράσταση (σε μεταδεματική μορφή πλέον) και όποτε συναντάμε έναν αριθμό να τον εισάγουμε σε μια στοίβα. Όταν συναντήσουμε έναν τελεστή εξάγουμε *δύο* στοιχεία από τη στοίβα, εφαρμόζουμε τον τελεστή πάνω τους και ξαναεισάγουμε το αποτέλεσμα στη στοίβα. Στο τέλος το μοναδικό στοιχείο που θα περιέχει η στοίβα θα είναι η τιμή της αριθμητικής παράστασης.

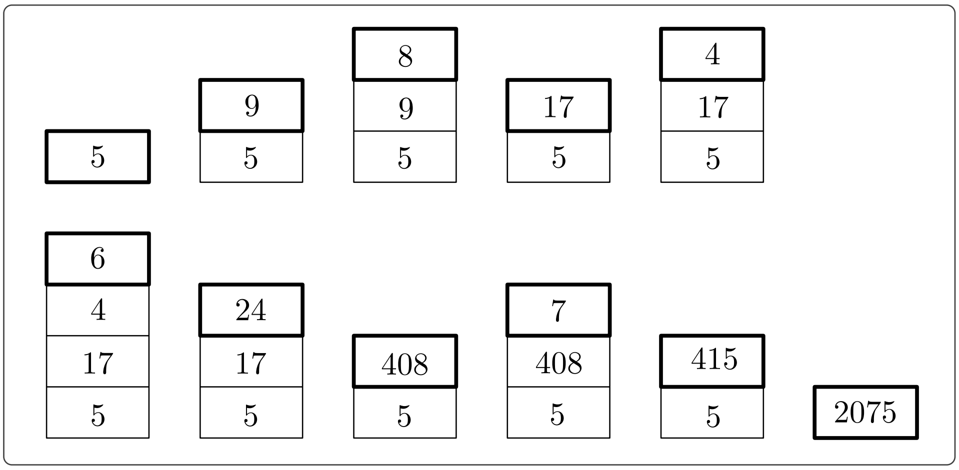
   Στο Σχήμα 3.1.5 βλέπουμε τα διάφορα στιγμιότυπα της στοίβας κατά τον υπολογισμό της τιμής της παράστασης $5\ 9\ 8 + 4\ 6 \cdot \cdot 7 + \cdot$.

   Ας δούμε τον αλγόριθμο που υλοποιεί την παραπάνω ιδέα.

PostfixEvaluate(E)

Είσοδος: Πίνακας E που περιέχει μία αριθμητική παράσταση εκφρασμένη σε μεταδεματική μορφή
Έξοδος : Η τιμή της αριθμητικής παράστασης

[1] Μπορούμε φυσικά να το υπολογίσουμε μετρώντας το πλήθος αριστερών και δεξιών παρενθέσεων και αφαιρώντας το από το συνολικό πλήθος στοιχείων της παράστασης.



Σχήμα 3.1.5: *Η στοίβα κατά τον υπολογισμό της τιμής της παράστασης* $5\ 9\ 8 + 4\ 6 \cdot \cdot 7 + \cdot$.

```
S ← new stack
x ← 0
for i ← 1 to length(E)
    if E[i] = 0 or E[i] = 1 or ··· or E[i] = 9 then
        Push(E[i], S)
    else if E[i] = + then
        x ← Pop(S)+ Pop(S)
        Push(x, S)
    else                                                  %Αν E[i] = ·
        x ← Pop(S)· Pop(S)
        Push(x, S)
return Pop(S)
```

Ο χρόνος και για το δεύτερο στάδιο υπολογισμού της αριθμητικής παράστασης είναι γραμμικός [1].

Κλήσεις υποπρογραμμάτων

Υποθέστε ότι τρέχουμε ένα πρόγραμμα Α το οποίο κατά την εκτέλεσή του καλεί ένα άλλο πρόγραμμα το Β, το οποίο με τη σειρά του καλεί ένα άλλο πρόγραμμα το Γ. Για να ολοκληρωθεί η εκτέλεση του Α χρειάζονται τα δεδομένα που θα παραχθούν από την εκτέλεση του Β και για να ολοκληρωθεί η εκτέλεση του Β χρειάζονται τα δεδομένα που θα παραχθούν από το Γ. Μεγεθύνετε αυτό το παράδειγμα κατά εκατοντάδες εμφωλευμένες κλήσεις και θα βρεθείτε μπροστά σε ένα τεράστιο μπλέξιμο.

[1] Υπό την προϋπόθεση ότι οι πράξεις γίνονται σε σταθερό χρόνο.

Ο Η/Υ μας προκειμένου να καταγράψει τη σειρά με την οποία θα πρέπει να εκτελεστούν τα προγράμματα χρησιμοποιεί μία στοίβα. Κάθε φορά που προκύπτει μία νέα κλήση (η οποία και θα πρέπει να ολοκληρωθεί πριν απ' όλες τις προηγούμενες) ο Η/Υ θα την τοποθετήσει στην κορυφή της στοίβας. Όταν πλέον σταματήσουν να προκύπτουν άλλες κλήσεις και χρειαζόμαστε τα αποτελέσματά τους θα κάνει εξαγωγή την καταγραφή της κλήσης, θα την εκτελέσει και θα τροφοδοτήσει τα δεδομένα στην νέα κορυφή της στοίβας.

Η παραπάνω διαδικασία βρίσκει μεγάλη εφαρμογή όταν έχουμε να κάνουμε με *αναδρομικούς αλγόριθμους*, αλγόριθμους δηλαδή που λύνουν ένα πρόβλημα λύνοντας ένα ή περισσότερα στιγμιότυπα του ίδιου προβλήματος. Ο τόπος που γίνεται αυτό είναι με μια σειρά *αναδρομικών* κλήσεων, μέσα στον κώδικά τους, του ίδιου τους του «εαυτό», με είσοδο τα *υποπροβλήματα* αυτά. Τα υποπροβλήματα θα πρέπει να σχετίζονται μεταξύ τους, δηλαδή η «λύση» του ενός να εξαρτάται από τη λύση του επόμενου, και να έχουν ολοένα και μικρότερο μέγεθος. Έτσι τελικά θα οδηγηθούμε σε μία οριακή *αρχική συνθήκη*, σε ένα πρόβλημα δηλαδή που η λύση του είναι τετριμμένη. Από εκεί και πέρα τα υποπροβλήματα λύνονται διαδοχικά (με αντίστροφή σειρά) μέχρι να πάρουμε τη λύση του αρχικού προβλήματος.

Ας δούμε ένα πολύ γνωστό παράδειγμα αναδρομικού αλγόριθμου.

Παράδειγμα 3.1.1. Ο αλγόριθμος που ακολουθεί υπολογίζει τον n-οστό όρο της ακολουθίας Fibonacci:

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n) = f(n-1) + f(n-2)) \text{ , για } n \geq 2 \end{cases}$$

Fibonacci(n)

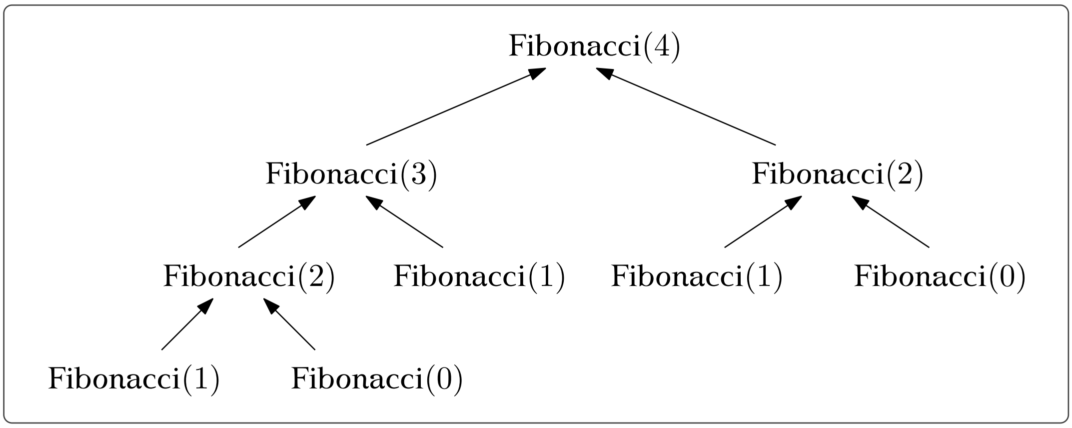
Είσοδος: Φυσικός αριθμός n
Έξοδος : Ο n-οστός όρος της ακολουθίας Fibonacci

```
if n = 0 or n = 1 then
    return 1
else
    return Fibonacci(n − 1)+Fibonacci(n − 2)
```

Το δέντρο των αναδρομικών κλήσεων που θα κάνει ο αλγόριθμος για $n = 4$ φαίνεται στο Σχήμα 3.1.6, ενώ στο Σχήμα 3.1.7 φαίνεται η οργάνωση αυτών των αναδρομικών κλήσεων με μία στοίβα.

Ο παραπάνω αλγόριθμος παρόλο που είναι πολύ «κομψός» έχει ένα πολύ σοβαρό ελάττωμα. Υπολογίζει πολλές φορές την ίδια τιμή (για παράδειγμα με είσοδο το 4 θα υπολογίσει το Fibonacci(1) τρεις φορές). Ως εκ τούτου ο χρόνος του είναι εκθετικός ($O(2^n)$ όπου n ο αριθμός της εισόδου).

Ο αλγόριθμος που ακολουθεί (δεν χρησιμοποιεί αναδρομή και) υπολογίζει τον n-οστό όρο σε γραμμικό χρόνο ($O(n)$), καθώς αποφεύγει τον επαναλαμβανόμενο υπολογισμό ίδιων τιμών (δες και Άσκηση 1.5).



Σχήμα 3.1.6: *Το δέντρο των αναδρομικών κλήσεων του Fibbonacci για $n = 4$. Τα βέλη δείχνουν τη «φορά» που θα ακολουθήσουν τα δεδομένα για να υπολογιστεί η τιμή Fibbonacci*(4).

QuickFibonacci(n)

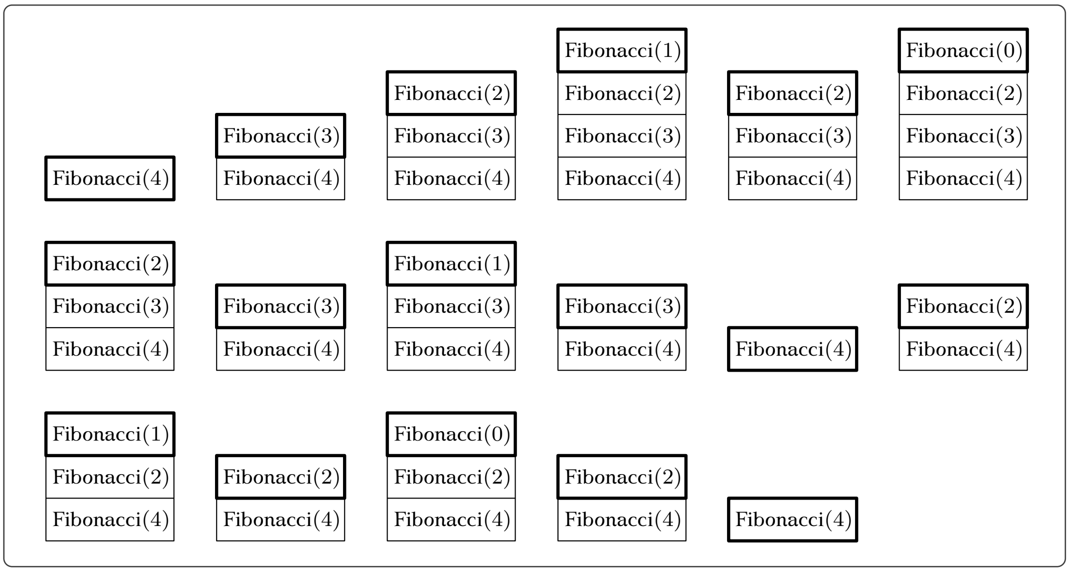
Είσοδος: Φυσικός αριθμός n
Έξοδος : Ο n-οστός όρος της ακολουθίας Fibonacci

```
if n = 0 or n = 1 then
    return 1
else
    pre ← 1
    cur ← 1
    for i ← 2 to n
        new ← prev + cur
        prev ← cur
        cur ← new
return cur
```

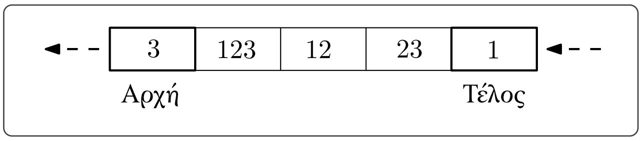
3.2 Ουρές

Μπορούμε να φανταστούμε πολλές καταστάσεις στην καθημερινότητά μας όπου η λογική LIFO δεν μπορεί να εφαρμοστεί. Για παράδειγμα φανταστείτε τι θα συνέβαινε αν την εφάρμοζαν τα ταμεία των εμπορικών καταστημάτων ή τα γραφεία εξυπηρέτησης κάποιας δημόσιας υπηρεσίας. Σε αυτές τις περιπτώσεις χρειαζόμαστε τη λογική *FIFO* (First In First Out) όπου ο πρώτος που μπαίνει στο σύστημα είναι και ο πρώτος που θα βγει από αυτό.

Η δομή δεδομένων που ακολουθεί αυτήν τη λογική ονομάζεται *ουρά*. Σε μια ουρά μπορεί να γίνει εξαγωγή στοιχείου μόνο από την αρχή της, ενώ οι εισαγωγές γίνονται πάντα στο τέλος της (δες Σχήμα 3.2.1). Όπως και στις στοίβες μπορούμε να αναπαραστήσουμε τις ουρές είτε



Σχήμα 3.1.7: *Τα στιγμιότυπα της στοίβας αναδρομικών κλήσεων του Fibbonacci για $n = 4$.*



Σχήμα 3.2.1: *Παράδειγμα ουράς. Εξάγεται το 3 ενώ εισαγωγή γίνεται μετά το 1.*

χρησιμοποιώντας πίνακες είτε συνδεδεμένες λίστες.

3.2.1 Αναπαράσταση μέσω πινάκων

Εκτός από τον πίνακα που θα αποθηκεύει τα στοιχεία της ουράς θα χρειαστούμε και δύο (ακέραιους) δείκτες που θα επισημαίνουν την αρχή και το τέλος της ουράς. Παρατηρήστε ότι καθώς θα γίνονται εξαγωγές από την αρχή της ουράς ο πίνακας θα έχει «κενά» κελιά τόσο στην αρχή του όσο και στο τέλος του (δες Σχήμα 3.2.2). Έστω Q μεταβλητή τύπου ουράς. Θα γράφουμε:

- $Q.matrix$ για να αναφερθούμε στον πίνακα της ουράς,

- $Q.front$ για να αναφερθούμε στον δείκτη με τιμή την αρχή της ουράς,

- $Q.rear$ για να αναφερθούμε στον δείκτη με τιμή το τέλος της ουράς.

Ένα παράδειγμα φαίνεται στο Σχήμα 3.2.2. Οι ουρές υποστηρίζουν τις ακόλουθες λειτουργίες:

		3	123	12	23	1	

Q

$$Q.matrix = [\_ \quad \_ \quad 3 \quad 123 \quad 12 \quad 23 \quad 1 \quad \_]$$

Σχήμα 3.2.2: *Αναπαράσταση της ουράς Q (χωρητικότητας 8) μέσω πίνακα. Η τιμή του δείκτη $Q.front$ είναι 3 ενώ του $Q.rear$ 7.*

1. *Αρχικοποίηση ουράς*
2. *Έλεγχος για το αν η ουρά είναι κενή*
3. *Εισαγωγή στοιχείου (Enqueue)*
4. *Εξαγωγή στοιχείου (Dequeue)*

Η αναπαράσταση των ουρών μέσω πίνακα, πέρα από τα γενικά μειονεκτήματα που έχουν οι αναπαραστάσεις δομών με πίνακα, πάσχουν από ακόμα ένα: το γεγονός ότι ο πίνακας θα έχει κενά κελιά και στην αρχή του. Για να γίνει κατανοητός ο λόγος που αυτό δημιουργεί πρόβλημα θα πρέπει πρώτα να περάσουμε στις λεπτομέρειες των λειτουργιών μίας ουράς.

1. *Αρχικοποίηση ουράς:* Για να αρχικοποιήσουμε μία ουρά με χωρητικότητα n θα χρησιμοποιούμε απλά την εντολή:

   $Q \leftarrow$ **new queue**(n)

   Στην ουσία αρχικοποιούμε έναν πίνακα $Q.matrix$ με n κελιά και δίνουμε στους δύο δείκτες την τιμή 0.

2. *Έλεγχος για το αν η ουρά είναι κενή:* Όταν η ουρά δεν περιέχει κανένα στοιχείο ο δείκτης $Q.front$ (όπως και ο $Q.rear$) θα έχει τιμή 0.

   IsEmpty(Q)

   Είσοδος: Ουρά Q
   Έξοδος : True αν η ουρά είναι κενή, False διαφορετικά

   ```
   if Q.front = 0 then
       return True
   else
       return False
   ```

3. *Εισαγωγή στοιχείου (Enqueue):* Η εισαγωγή όπως είπαμε γίνεται στο τέλος της ουράς. Χρειάζεται μεγάλη προσοχή για το πότε η ουρά είναι πραγματικά γεμάτη, καθώς

μπορούν να υπάρχουν διαθέσιμες θέσεις –κενά κελιά– στην αρχή του πίνακα (που προέκυψαν από εξαγωγές στοιχείων) ενώ να μην υπάρχουν διαθέσιμες θέσεις στο τέλος του πίνακα. Έτσι εσφαλμένα θα πιστεύαμε ότι η ουρά είναι γεμάτη.

Θα πρέπει λοιπόν να βρεθεί ένας τρόπος να εκμεταλλευτούμε και αυτές τις θέσεις. Σε πρώτη φάση θα αφήσουμε στην άκρη αυτό το γεγονός, ευελπιστώντας ότι ο τρόπος που θα κάνουμε τις εξαγωγές κάπως θα το επιλύσει...

Enqueue(x, Q)

Είσοδος: Ουρά Q και στοιχείο x
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

if $Q.rear = $ **length**$(Q.matrix)$ **then**
    return "Γεμάτη ουρά"
else
    $Q.rear \leftarrow Q.rear + 1$
    $Q.matrix[Q.rear] \leftarrow x$
    if $Q.front = 0$ **then**   *% Είναι η πρώτη εισαγωγή στην ουρά*
        $Q.front \leftarrow 1$

Ο χρόνος που χρειάζεται ο παραπάνω αλγόριθμος για να κάνει την εισαγωγή στοιχείου είναι σταθερός.

4. *Εξαγωγή στοιχείου (Dequeue):* Για να διευθετήσουμε το πρόβλημα που δημιουργούν τα κενά κελιά στην αρχή του πίνακα μπορούμε μετά από κάθε εξαγωγή (και διαγραφή) στοιχείου να μετακινούμε όλα τα στοιχεία της ουράς μία θέση αριστερά, έτσι ώστε να καλύπτουμε το κενό κελί (δες Σχήμα 3.2.3).

   Η στρατηγική αυτή είναι ιδιαίτερα χρονοβόρα για πράξη που θα εφαρμόζεται συνεχώς (ο χρόνος που θα χρειαστούμε είναι $O(n)$, όπου n το μέγεθος της ουράς). Στη συνέχεια θα δώσουμε έναν πολύ καλύτερο τρόπο να λύσουμε το πρόβλημα, ας δούμε πρώτα όμως (για εξάσκηση) τον αλγόριθμο που υλοποιεί την εξαγωγή σύμφωνα με όσα είπαμε παραπάνω.

Dequeue(Q)

Είσοδος: Ουρά Q
Έξοδος : Το πρώτο στοιχείο της ουράς (αν υπάρχει, αλλιώς ενημερώνουμε τον χρήστη με σχετικό μήνυμα)

if IsEmpty(Q) **then**
    return "Άδεια ουρά"

3	123	12	23	1			

Q

$$Q.matrix = \begin{bmatrix} 3 & 123 & 12 & 23 & 1 & \_ & \_ & \_ \end{bmatrix}$$

123	12	23	1				

Q

$$Q.matrix = \begin{bmatrix} 123 & 12 & 23 & 1 & \_ & \_ & \_ & \_ \end{bmatrix}$$

Σχήμα 3.2.3: *Κάτω φαίνεται η καινούργια ουρά (και ο πίνακας που την αναπαριστά) μετά την εξαγωγή.*

```
else
    x ← Q.matrix[1]
    if Q.rear = 1 then                                    %Αδειάζουμε την ουρά
        Q.front ← 0
        Q.rear ← 0
    else
        for i ← 1 to Q.rear − 1
            Q.matrix[i] ← Q.matrix[i + 1]
        Q.rear ← Q.rear − 1
    return x
```

Μία καλύτερη προσέγγιση είναι να αναπαραστήσουμε τις ουρές χρησιμοποιώντας *κυκλικούς πίνακες*, πίνακες δηλαδή όπου θεωρούμε πως το τελευταίο κελί τους έχει σαν επόμενο το πρώτο κελί του πίνακα.

Δεν θα χρειαστεί να τροποποιήσουμε τη δομή δεδομένων του μονοδιάστατου πίνακα που είδαμε στο Κεφάλαιο 1, θα κάνουμε την εισαγωγή με έναν πιο έξυπνο τρόπο. Αυτό που θα κάνουμε είναι όποτε χρειάζεται να εισάγουμε στοιχείο σε μια ουρά Q και ισχύει ότι $Q.rear =$ **length**$(Q.matrix)$ (πράγμα που σύμφωνα με τα προηγούμενα θα σήμαινε ότι ο πίνακας είναι γεμάτος) θα το εισάγουμε στο κελί 1 του πίνακα (εφόσον αυτό είναι κενό).

Σε αυτήν την προσέγγιση η ουρά θα είναι γεμάτη όταν η επόμενη θέση από το $Q.rear$ στον πίνακα είναι αυτή του $Q.front$ (δες Σχήμα 3.2.4). Η πράξη της εισαγωγής σύμφωνα με αυτή την προσέγγιση υλοποιείται ως εξής:

	$Q.rear$				$Q.front$		
23	1				3	123	12

Q

$$Q.matrix = \begin{bmatrix} 23 & 1 & \_ & \_ & \_ & 3 & 123 & 12 \end{bmatrix}$$

Σχήμα 3.2.4: *Παράδειγμα αναπαράστασης ουράς με «κυκλικό» πίνακα.*

Enqueue(x, Q)

Είσοδος: Ουρά Q και στοιχείο x
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

if $Q.rear =$ **length**($Q.matrix$) **then**
  $temp \leftarrow 1$ *%Η $temp$ κρατάει την πιθανή θέση για να γίνει η εισαγωγή*
else
  $temp \leftarrow Q.rear + 1$
if $temp = Q.front$ **then**
  return "Γεμάτη ουρά"
else
  $Q.rear \leftarrow temp$
  $Q.matrix[Q.rear] \leftarrow x$
  if $Q.front = 0$ **then** *%Είναι η πρώτη εισαγωγή στην ουρά*
    $Q.front \leftarrow 1$

Ο χρόνος της εισαγωγής είναι πάλι σταθερός.

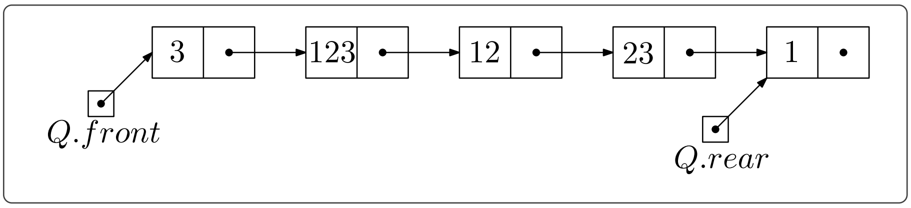
Κατά την εξαγωγή δεν θα εξάγουμε το πρώτο στοιχείο του πίνακα, θα πρέπει να εξάγουμε το στοιχείο που βρίσκεται στη θέση $Q.front$. Μετά θα αυξήσουμε την τιμή αυτού του δείκτη κατά ένα ή θα του δώσουμε τιμή 1 αν η προηγούμενή του τιμή ήταν ίση με τη διάσταση του πίνακα.

Η πράξη της εξαγωγής υλοποιείται ως εξής:

Dequeue(Q)

Είσοδος: Ουρά Q
Έξοδος : Το πρώτο στοιχείο της ουράς (αν υπάρχει, αλλιώς ενημερώνουμε τον χρήστη με σχετικό μήνυμα)

if IsEmpty(Q) **then**
  return "Άδεια ουρά"



Σχήμα 3.2.5: *Παράδειγμα αναπαράστασης ουράς με λίστα.*

else
  $x \leftarrow Q.matrix[Q.front]$
  if $Q.front = Q.rear$ **then** *%Αδειάσαμε την ουρά*
    $Q.front \leftarrow 0$
    $Q.rear \leftarrow 0$
  else if $Q.front =$ **length**$(Q.matrix)$ **then**
    $Q.front \leftarrow 1$
  else
    $Q.front \leftarrow Q.front + 1$
  return x

Με το τέχνασμα που εφαρμόσαμε καταφέραμε και η πράξη της εξαγωγής να έχει σταθερό χρόνο.

3.2.2 Αναπαράσταση μέσω συνδεδεμένων λιστών

Για να αναπαραστήσουμε μία ουρά χρησιμοποιώντας (απλά) συνδεδεμένες λίστες, πέρα από τον δείκτη κεφαλής που θα «δείχνει» στο πρώτο στοιχείο της ουράς (αυτό που θα εξαχθεί), θα χρειαστούμε έναν δείκτη ο οποίος θα «δείχνει» στον τελευταίο κόμβο της λίστας (μετά από τον οποίο θα γίνεται η εισαγωγή στην ουρά) [1]. Θα γράφουμε:

- $Q.front$ (για λόγους συμβατότητας) για να αναφερθούμε στον δείκτη που δείχνει την αρχή της ουράς (η τιμή του θα ισούται με την τιμή του δείκτη κεφαλής της λίστας που την αναπαριστά),
- $Q.rear$ για να αναφερθούμε στον δείκτη που δείχνει το τέλος της ουράς (Σχήμα 3.2.5).

Ας δούμε πως διαμορφώνονται οι λειτουργίες της ουράς:

1. *Αρχικοποίηση ουράς:* Πάλι θα χρησιμοποιήσουμε την εντολή:

   $Q \leftarrow$ **new queue**

[1] Αν δεν χρησιμοποιούσαμε αυτόν τον δείκτη θα έπρεπε κάθε φορά που κάναμε εισαγωγή να βρίσκαμε τον τελευταίο κόμβο της λίστας, πράγμα που θα έκανε την εισαγωγή ασύμφορη (γραμμικός χρόνος).

με την οποίο στην ουσία θα αρχικοποιούμε την τιμή των δύο δεικτών $Q.front$ και $Q.rear$ στην τιμή NIL.

2. *Έλεγχος για το αν η ουρά είναι κενή:* Απλά ελέγχουμε αν ο δείκτης $Q.front$ (ή ο $Q.rear$) έχει τιμή NIL).

IsEmpty(Q)

Είσοδος: Ουρά Q
Έξοδος : True αν η ουρά είναι κενή, False διαφορετικά

if $Q.front =$ NIL **then**
  return True
else
  return False

3. *Εισαγωγή στοιχείου (Enqueue):* Προτού εισάγουμε το στοιχείο θα πρέπει να ελέγξουμε αν είναι η πρώτη εισαγωγή για να αλλάξουμε και τον δείκτη $Q.front$ (πέρα από τον $Q.rear$).

Enqueue(x, Q)

Είσοδος: Ουρά Q και στοιχείο x
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

$P \leftarrow$ **address**(**new node**(x))
if $Q.front =$ NIL **then** *%Πρώτη εισαγωγή στην ουρά*
  $Q.front \leftarrow P$
else *%Η InsertAfter στην ουσία*
  node($Q.rear$)$.next \leftarrow P$
$Q.rear \leftarrow P$

4. *Εξαγωγή στοιχείου (Dequeue):* Αντίστοιχα με την εισαγωγή θα χρειαστεί επιπλέον να ελέγξουμε αν εξάγουμε το τελευταίο στοιχείο της ουράς (για να αλλάξουμε και τον δείκτη $Q.rear$).

Dequeue(Q)

Είσοδος: Ουρά Q
Έξοδος : Το πρώτο στοιχείο της ουράς (αν υπάρχει, αλλιώς ενημερώνουμε τον χρήστη με σχετικό μήνυμα)

```
if IsEmpty(Q) then
    return "Άδεια ουρά"
else
    x ← node(Q.front).item
    if Q.front = Q.rear then                          %Αδειάσαμε την ουρά
        Q.front ← NIL
        Q.rear ← NIL
    else                                              %H DeleteFirst στην ουσία
        Q.front ← node(Q.front).next
    return x
```

Ο χρόνο που χρειάζονται όλες οι πράξεις είναι σταθερός.

3.2.3 Ουρές προτεραιότητας

Υποθέστε, ο μη γένοιτο, ότι βρίσκεστε στα επείγοντα περιστατικά ενός νοσοκομείου και περιμένετε στην ουρά για να εξεταστείτε. Η ουρά αυτή φυσικά ακολουθεί την FIFO λογική με μία πολύ σημαντική εξαίρεση. Αν αφιχθεί κάποιο πολύ σοβαρό περιστατικό παίρνει προτεραιότητα έναντι των ήδη υπαρχόντων περιστατικών, πηγαίνοντας αυτομάτως στην αρχή της ουράς. Τέτοιου είδους ουρές καλούνται *ουρές προτεραιότητας* και ακολουθούν την λογική *HPIFO* (Highest Priority In First Out).

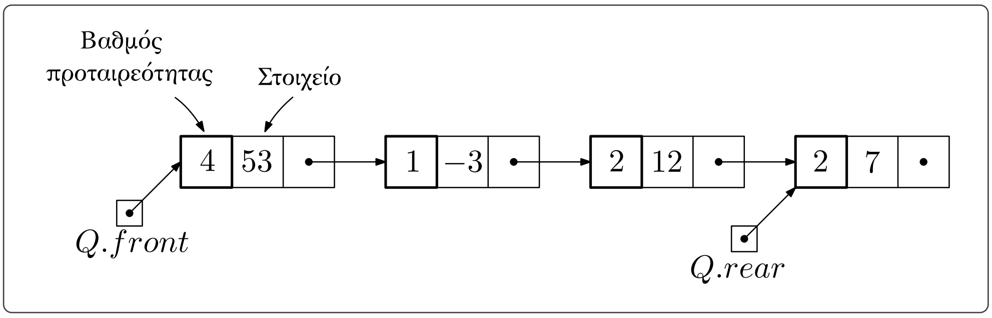
Για να αναπαραστήσουμε μία ουρά προτεραιότητας θα χρησιμοποιήσουμε συνδεδεμένες λίστες σε συνδυασμό με δύο δείκτες που θα δείχνουν τον πρώτο και τον τελευταίο κόμβο. Η επιπλέον οργάνωση που χρειάζεται αυτή η ουρά θα δοθεί μέσω ενός έξτρα πεδίου σε κάθε κόμβο της λίστας, έναν ακέραιο αριθμό που θα παίρνει ως τιμή τον *βαθμό προτεραιότητας* του στοιχείου που περιέχει ο κόμβος.

Έστω x μεταβλητή τύπου κόμβου ουράς προτεραιότητας. Γράφοντας $x.priority$ θα αναφερόμαστε στον βαθμό προτεραιότητας του κόμβου x, ο οποίος –ας συμφωνήσουμε– ότι θα παίρνει ακέραιες τιμές στο διάστημα $[1, +\infty)$ και ότι η *μικρότερη τιμή θα έχει προτεραιότητα έναντι των μεγαλυτέρων τιμών.* Τα στοιχεία με ίδιο βαθμό προτεραιότητας θα εξυπηρετούνται σύμφωνα με την FIFO λογική (δες Σχήμα 3.2.6).

Να σημειώσουμε επίσης ότι εφόσον ο κόμβος πλέον περιέχει δύο πεδία που ορίζει ο «χρήστης» θα πρέπει να τροποποιήσουμε την εντολή δημιουργίας κόμβου ώστε να δέχεται ακόμα ένα όρισμα:

- **new priority node**(y, p): Αρχικοποιούμε μία μεταβλητή τύπου κόμβου με περιεχόμενο y (που δεν συνδέεται με κάποιον άλλο κόμβο) και βαθμό προτεραιότητας p.

Οι πράξεις που υποστηρίζουν οι ουρές προτεραιότητας είναι οι ίδιες με αυτές που υποστηρίζουν οι απλές ουρές.



Σχήμα 3.2.6: *Παράδειγμα αναπαράστασης ουράς προτεραιότητας. Οι κόμβοι της λίστας περιέχουν ένα έξτρα πεδίο που περιέχει τον βαθμό προτεραιότητας. (Εφόσον δεν εισαχθούν άλλα στοιχεία στην ουρά) τα στοιχεία θα εξαχθούν με την ακόλουθη σειρά: -3, 12, 7, 53.*

1. *Αρχικοποίηση ουράς προτεραιότητας:* Δεν παρουσιάζει διαφορές από την αρχικοποίηση μίας απλής ουράς. Υλοποιείται με την εντολή **new priority queue**.

2. *Έλεγχος για το αν η ουρά είναι κενή:* Και εδώ δεν έχουμε κάποια διαφορά από τις απλές ουρές.

3. *Εισαγωγή στοιχείου (Enqueue):* Υλοποιούμε την πράξη της εισαγωγής όπως ακριβώς και στις απλές ουρές, με τον αλγόριθμο Enqueue. Θα πρέπει όμως ο αλγόριθμος αυτός να δέχεται ένα επιπλέον όρισμα, τον βαθμό προτεραιότητας του στοιχείου που θα εισαχθεί.

Συνέπεια της επιλογής μας να κρατήσουμε τον αλγόριθμο εισαγωγής απλό είναι ότι η πράξη της εξαγωγής θα παρουσιάσει σοβαρές διαφορές από αυτήν των απλών ουρών [1].

4. *Εξαγωγή στοιχείου (Dequeue):* Το πρόβλημα που καλούμαστε να αντιμετωπίσουμε είναι το πως θα βρούμε *γρήγορα* τον κόμβο με τη χαμηλότερη προτεραιότητα για να τον εξάγουμε από την ουρά. Ας σκεφτούμε μερικούς τρόπους να το επιτύχουμε αυτό:

- Η πρώτη μας απόπειρα είναι απλά να βρίσκουμε το στοιχείο που πρέπει να εξαχθεί κάνοντας γραμμική αναζήτηση. Καθώς η γραμμική αναζήτηση χρειάζεται χρόνο $O(n)$, όπου n το πλήθος των στοιχείων που έχει η ουρά, ο αλγόριθμος της εξαγωγής θα χρειάζεται γραμμικό χρόνο. Ο χρόνος αυτός, όπως έχουμε τονίσει αρκετές φορές, είναι απαγορευτικός όταν έχουμε να κάνουμε με πράξεις που θα εφαρμόζονται συνεχώς.

- Μία δεύτερη απόπειρα θα ήταν να πάμε πίσω στην πράξη της εισαγωγής και να την τροποποιήσουμε έτσι ώστε η ουρά να διατηρείται πάντα ταξινομημένη σύμφωνα με τον βαθμό προτεραιότητας (από τη μικρότερη τιμή προς τη μεγαλύτερη). Με αυτόν τον τρόπο θα εξάγουμε πάντα το πρώτο στοιχείο της ουράς.

[1] Τουλάχιστον πετυχαίνουμε το FIFO. Θα χρειαστεί όμως να κάνουμε αρκετή δουλειά για να πετύχουμε και το HPIFO.

Δυστυχώς όμως ούτε έτσι βελτιώνονται ποιοτικά οι χρόνοι των πράξεων, καθώς, ναι μεν η εξαγωγή θα γίνεται πιο γρήγορα (σε σταθερό χρόνο), η εισαγωγή όμως θα γίνεται πιο αργά (σε γραμμικό χρόνο), και αυτό γιατί θα πρέπει κάθε φορά να βρίσκουμε την κατάλληλη θέση για να εισάγουμε το καινούργιο στοιχείο αναγκαζόμενοι (στη χειρότερη περίπτωση) να διασχίζουμε συνεχώς όλη τη λίστα.

Ο αλγόριθμος που υλοποιεί την πράξη της εξαγωγής σύμφωνα με αυτά που περιγράψαμε στην πρώτη απόπειρα είναι ο εξής (η δεύτερη απόπειρα αφήνεται ως άσκηση):

Dequeue(Q)

Είσοδος: Ουρά Q
Έξοδος : Το πρώτο στοιχείο της ουράς (αν υπάρχει, αλλιώς ενημερώνουμε τον χρήστη με σχετικό μήνυμα)

```
if IsEmpty(Q) then
    return "Άδεια ουρά"
else if Q.front = Q.rear then                 %Ένα μόνο στοιχείο στην ουρά (την αδειάζουμε)
    x ← node(Q.front).item
    Q.front ← NIL
    Q.rear ← NIL
else
    P ← Q.front                               %Ο κόμβος που εξετάζουμε
    PREV ← NIL                                %Ο προηγούμενος από αυτόν που εξετάζουμε
    MIN ← P                                   %Ο κόμβος με την υψηλότερη προτεραιότητας (απ' όσους έχουμε ελέγξει)
    MINPREV ← PREV                            %Ο προηγούμενος από αυτόν
    while P ≠ NIL
        if node(P).priority < node(MIN).priority then
            MIN ← P
            MINPREV ← PREV
        PREV ← P
        P ← node(P).next
    x ← node(MIN).item
    if MINPREV = NIL then                     %Εξάγουμε το πρώτο στοιχείο στην ουρά
        Q.front ← node(MIN).next
    else if node(MIN).next = NIL then         %Εξάγουμε το τελευταίο στοιχείο στην ουρά
        Q.rear ← MINPREV
        DeleteNext(MINPREV)
    else
        DeleteNext(MINPREV)
```

26 **return** x

Στις πρώτες 6 γραμμές του κώδικα διευθετούμε τις περιπτώσεις όπου έχουμε άδεια ουρά ή ουρά που περιέχει μόνο ένα στοιχείο (συνεπώς δεν χρειάζεται να αναζητήσουμε το στοιχείο με την υψηλότερη προτεραιότητα, δηλαδή τον κόμβο που περιέχει τον μικρότερο βαθμό προτεραιότητας). Έπειτα στις γραμμές 12–20 κάνουμε τη διαπέραση για να βρούμε τον κόμβο με την υψηλότερη προτεραιότητα (θα πρέπει να κρατήσουμε και έναν δείκτη που δείχνει στον προηγούμενο από αυτόν κόμβο για να χρησιμοποιήσουμε τη DeleteNext και να διαγράψουμε τον κόμβο με την υψηλότερη προτεραιότητα από την ουρά). Τέλος στις γραμμές 22–28 αντιμετωπίζουμε τις περιπτώσεις που εξάγουμε το πρώτο ή το τελευταίο στοιχείο της ουράς και ως εκ τούτου θα πρέπει να διορθώσουμε τους δείκτες $Q.front$ και $Q.rear$ αντίστοιχα.

Στη χειρότερη περίπτωση θα χρειάζεται να διασχίζουμε ολόκληρη την ουρά σε κάθε εξαγωγή επομένως ο χρόνος που χρειάζεται ο αλγόριθμος Dequeue θα είναι $O(n)$, όπου n το πλήθος στοιχείων της ουράς.

Φαντάζει απίθανο να αποφύγουμε το γεγονός ότι μία από τις δύο πράξεις (εισαγωγή και εξαγωγή) θα έχει γραμμικό χρόνο, στο επόμενο κεφάλαιο όμως θα δούμε ότι αυτό είναι εφικτό, μπαίνοντας σε έναν καινούργιο «κόσμο» δομών δεδομένων, τον κόσμο των *μη-γραμμικών* δομών. Θα χρησιμοποιήσουμε τη δομή δεδομένων *σωρό*, όπου εκμεταλλευόμενοι τη *μη-γραμμική* οργάνωση που διατηρεί στα στοιχεία, θα μπορέσουμε να υλοποιήσουμε τις πράξεις της εισαγωγής και της διαγραφής σε χρόνο $O(\log n)$ (όπου n το πλήθος των στοιχείων). Παρόλο που οι εισαγωγές θα χρειάζονται περισσότερο χρόνο από $O(1)$ του τρόπου που περιγράψαμε παραπάνω, αν παρατηρήσουμε τον συνολικό χρόνο που θα χρειαστούμε σε μία ακολουθία από n εισαγωγές και n εξαγωγές θα διαπιστώσουμε ότι χρησιμοποιώντας σωρούς πετυχαίνουμε συνολικό χρόνο $O(n \log n)$ αντί για $O(n^2)$ που πετυχαίναμε με την προσέγγιση που είδαμε πριν [1].

Κλείνοντας αυτό το κεφάλαιο (για να συνεχίσουμε τη συζήτησή σχετικά με τις ουρές προτεραιότητας στο επόμενο), να αναφέρουμε ότι μπορούμε να προσθέσουμε και άλλες χρήσιμες λειτουργίες στις ουρές προτεραιότητας, όπως για παράδειγμα την εύρεση της τιμής του χαμηλότερου βαθμού προτεραιότητας από τα στοιχεία της ουράς [2].

Ασκήσεις

3.1. Δώστε αλγόριθμο που λύνει το πρόβλημα που περιγράφεται στην Άσκηση 2.5 με χρονική πολυπλοκότητα $O(n)$, όπου n το πλήθος χαρακτήρων του αλφαριθμητικού.

3.2. Δώστε αλγόριθμο Binary(n) που δέχεται ως είσοδο έναν φυσικό αριθμό n και

[1] Εδώ έχουμε ένα παράδειγμα *αντισταθμιστικής ανάλυσης πολυπλοκότητας*.

[2] Αυτό θα μας ήταν πολύ χρήσιμο αν έπρεπε να εισάγουμε ένα στοιχείο στην ουρά με πάρα πολύ χαμηλή προτεραιότητα, που θα θέλαμε δηλαδή να εξυπηρετηθεί αφού πρώτα εξυπηρετηθούν όλα τα ήδη υπάρχοντα στοιχεία.

επιστρέφει την δυαδική αναπαράστασή του. Για παράδειγμα η κλήση Binary(19) θα επιστρέψει την τιμή 10011. Ο αλγόριθμος αυτός θα χρησιμοποιεί μία στοίβα για να υπολογίσει την δυαδική αναπαράσταση και στο τέλος θα επιστρέφει την δυαδική αναπαράσταση σε έναν πίνακα. (Μπορείτε να χρησιμοποιήσετε την συνάρτηση mod που αναφέρθηκε στην Άσκηση 2.2 καθώς και το γεγονός ότι η δυαδική αναπαράσταση του αριθμού n έχει το πολύ $\log_2 n$ ψηφία.)

3.3. Υποθέστε ότι σας δίνεται μία ουρά Q με στοιχεία ακέραιους αριθμούς που είναι διατεταγμένοι κατά αύξουσα τιμή. Κάντε τις κατάλληλες αλλαγές στον αλγόριθμο εισαγωγής Enqueue(x, Q) έτσι ώστε η ουρά μετά την εισαγωγή του x να παραμένει διατεταγμένη (προφανώς το στοιχείο x δεν θα εισάγεται στο τέλος της ουράς αλλά σε κατάλληλη θέση). Στα στοιχεία με ίση τιμή θα ακολουθείται η FIFO λογική.

3.4. Δώστε αλγόριθμο Check(E) που δέχεται σαν είσοδο μία αριθμητική παράσταση αποθηκευμένη σε έναν πίνακα E, η οποία περιέχει τριών ειδών παρενθέσεις: ( ), [ ] και { }. Σκοπός του αλγορίθμου είναι να ελέγξει αν υπάρχει το ίδιο πλήθος αριστερών και δεξιών παρενθέσεων του κάθε τύπου και αν η σειρά των παρενθέσεων είναι σωστή (δηλαδή αν μία παρένθεση ενός τύπου «κλείνει» με παρένθεση του ίδιου τύπου), επιστρέφοντας το μήνυμα «Ναι» αν ισχύουν τα παραπάνω και «Όχι» σε αντίθετη περίπτωση. Παραδείγματος χάρη για την παράσταση $\{2 \cdot [7 \cdot (2+3)+3]-4\}$ ο αλγόριθμος επιστρέφει «Ναι», ενώ για τις παραστάσεις $\{(1+3) \cdot [9-6]$ και $(2+3] \cdot [1-6)$ θα επιστρέψει «Όχι». Ο αλγόριθμος αυτός θα πρέπει να έχει χρόνο $O(n)$, όπου n το πλήθος συμβόλων της αριθμητικής παράστασης (μέγεθος του πίνακα E δηλαδή).

3.5. Δώστε αλγόριθμο LowestPriority(Q) που δέχεται σαν είσοδο μία ουρά προτεραιότητας Q και επιστρέφει την τιμή του μεγαλύτερου βαθμού προτεραιότητας που έχουν τα στοιχεία της ουράς.

3.6. Θεωρήστε την ακόλουθη αναπαράσταση μίας ουράς προτεραιότητας Q με το πολύ n-στοιχεία:

- Τα στοιχεία της ουράς αποθηκεύονται σε έναν δισδιάστατο πίνακα $2 \times n$, όπου η πρώτη γραμμή του περιέχει τα στοιχεία της ουράς και η δεύτερη τον βαθμό προτεραιότητάς τους. Στον πίνακα αυτόν αναφερόμαστε γράφοντας Q.matrix. (Για παράδειγμα το στοιχείο Q.matrix$[1, i]$ είναι το i-οστό στοιχείο της ουράς και το στοιχείο Q.matrix$[2, i]$ είναι ο βαθμό προτεραιότητάς του.)
- Πέρα από τον πίνακα χρησιμοποιούμε δύο ακέραιους δείκτες Q.front και Q.rear με τιμή την στήλη του πίνακα που βρίσκεται το πρώτο και το τελευταίο στοιχείο της ουράς αντίστοιχα.

Δώστε τους αλγορίθμους που υλοποιούν τις πράξεις εισαγωγής και εξαγωγής σύμφωνα με αυτή την αναπαράσταση.

3.7. Ένας άλλος τρόπος να ακολουθούμε την HPIFO λογική σε μία ουρά προτεραιότητας είναι να κρατάμε ταξινομημένη την ουρά κατά αύξοντα βαθμό προτεραιότητας, όπου τα στοιχεία με τον ίδιο βαθμό προτεραιότητας ακολουθούν μεταξύ τους την σειρά με την οποία εισήχθησαν στην ουρά.
Δώστε τους αλγορίθμους που υλοποιούν τις πράξεις εισαγωγής και εξαγωγής σύμφωνα με την παραπάνω ιδέα (χρησιμοποιώντας απλά συνδεδεμένη λίστα και δύο δείκτες που δείχνουν στην κεφαλή και στο τέλος της λίστας).

ΚΕΦΑΛΑΙΟ 4

ΔΕΝΤΡΑ

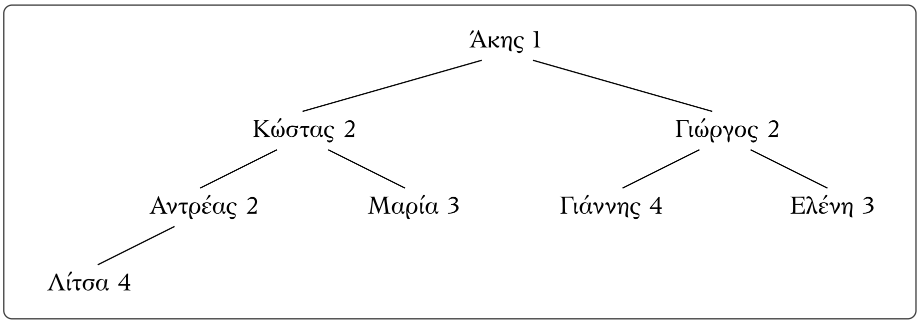
Θα συνεχίσουμε να μελετάμε τις ουρές προτεραιότητας. Ας ξεκινήσουμε με ένα παράδειγμα.

Παράδειγμα 4.0.1. Ας υποθέσουμε ότι έχει διαμορφωθεί η ακόλουθη ουρά προτεραιότητας με τους ασθενείς στα επείγοντα περιστατικά:

Όνομα	**Βαθμός προτεραιότητας**
Γιάννης	4
Κώστας	2
Μαρία	3
Ελένη	3
Γιώργος	2
Λίτσα	4
Αντρέας	2
Άκης	1

Σύμφωνα με την αναπαράσταση που είδαμε στο τέλος του προηγούμενου κεφαλαίου η εισαγωγή γίνεται σε σταθερό χρόνο (εισάγουμε το στοιχείο πάντα στο τέλος της ουράς και χρησιμοποιούμε έναν δείκτη για να μην χρειάζεται να το βρίσκουμε κάθε φορά) ενώ η εξαγωγή σε γραμμικό χρόνο (στην παραπάνω ουρά για να εξαχθεί το στοιχείο με την υψηλότερη προτεραιότητα, ο Άκης δηλαδή, θα γίνουν 8 «βήματα»).

Ας υποθέσουμε ότι είχαμε οργανώσει τους ασθενής όπως φαίνεται στο σχεδιάγραμμα στο Σχήμα 4.0.1, έτσι ώστε κάθε ασθενής που βρίσκεται *από κάτω από κάποιον άλλο ασθενή να έχει μεγαλύτερο βαθμό προτεραιότητας* (χαμηλότερη προτεραιότητα). Ας υποθέσουμε επιπλέον ότι οι πράξεις τις εισαγωγής και της εξαγωγής συντηρούν αυτήν τη *δενδρική διάταξη* των στοιχείων της ουράς και ότι μπορούν να το κάνουν αυτό αρκετά γρήγορα, σε χρόνο



Σχήμα 4.0.1: *Δενδρική διάταξη των στοιχείων της ουράς προτεραιότητας.*

$O(\log n)$ για ουρά με n στοιχεία (δυστυχώς το $O(1)$ δεν είναι εφικτό) [1].

Ας δούμε τον χρόνο που θα χρειαστούμε για την ακολουθία των 8 εισαγωγών και των 8 εξαγωγών συνολικά. Με την αναπαράσταση του προηγούμενου κεφάλαιου χρειάζονται 8 βήματα για τις εισαγωγές και 36 βήματα για τις 8 εξαγωγές, συνολικά 44 βήματα, ενώ με τη δενδρική δομή 14 βήματα για τις εισαγωγές και 14 βήματα για τις 8 εξαγωγές, συνολικά 28 βήματα.

Πιο αναλυτικά σύμφωνα με τον πρώτο τρόπο σε μία ακολουθία n εισαγωγών και εξαγωγών χρειάζονται:

$$\underbrace{1+\cdots+1}_{n-\text{φορές}}+n+n-1+\cdots+1=n+\frac{n(n+1)}{2}=O(n^2)\text{ βήματα}$$

Με τον δεύτερο τρόπο, αν καταφέρουμε να υλοποιήσουμε τις δύο πράξεις σε χρόνο $O(\log n)$, θα χρειάζονται (το πολύ):

$$\underbrace{\log n+\cdots+\log n}_{2n-\text{φορές}}=2n\log n=O(n\log n)\text{ βήματα}$$

Αν δεν σας είναι ξεκάθαρος ο βαθμός της βελτίωσης στον χρόνο που απαιτείται για τις n εισαγωγές και εξαγωγές, ένα πιο ρεαλιστικό παράδειγμα θα σας πείσει. Πάρτε για $n=1024$:

$$\begin{aligned} 1^{\text{ος}}\text{ τρόπος:} &\quad 1024\cdot 1024=1048576\text{ βήματα}\\ 2^{\text{ος}}\text{ τρόπος:} &\quad 1024\cdot 10=10240\text{ βήματα}\end{aligned}$$

Θα αναρωτιέστε πως προκύπτει το $\log n$ στον χρόνο των δύο πράξεων. Θα δείξουμε ότι το ύψος αυτού του δέντρου (η μεγαλύτερη «απόσταση» που απέχει κάποιο στοιχείο από το στοιχείο που βρίσκεται πάνω-πάνω) είναι $O(\log n)$ και ότι οι πράξεις χρειάζονται χρόνο ανάλογο προς αυτό το ύψος. Προκειμένου όμως να το κάνουμε αυτό θα πρέπει να αναφερθούμε πρώτα στα *δέντρα* και να ορίσουμε κάποιες βασικές έννοιες.

[1] Παρατηρήστε ότι το στοιχείο με την υψηλότερη προτεραιότητα θα είναι πάντα το πάνω-πάνω στοιχείο του δέντρου.

4.1 Ορισμοί

Στο προηγούμενο παράδειγμα είδαμε πως μία *μη-γραμμική* δομή δεδομένων μπορεί να μας δώσει τη δυνατότητα να υλοποιήσουμε υπολογιστικά «βαριές» πράξεις πολύ πιο γρήγορα. Η δομή δεδομένων που χρησιμοποιήσαμε καλείται *δέντρο* (για την ακρίβεια χρησιμοποιήσαμε ένα δέντρο ειδικής μορφής, τον λεγόμενο *σωρό*, που αποτελεί μία εξειδίκευση των *δυαδικών δέντρων αναζήτησης*). Τα δέντρα αποτελούν ίσως την πιο απλή μορφή *γραφημάτων*, μία δομή που θα εξετάσουμε στο επόμενο κεφάλαιο.

Ορισμός 4.1.1. Ένα δέντρο $T = (V, E, r)$ αποτελείται από ένα σύνολο *κορυφών* V, ένα σύνολο *ακμών* E και μία κορυφή $r \in V$. Την κορυφή r (που ξεχωρίσαμε) την αποκαλούμε *ρίζα* του δέντρου. Οι ακμές αποτελούν *διατεταγμένα ζεύγη* κορυφών του V.

Έστω ακμή $e = (u, v) \in E$. Θα λέμε ότι η e *ξεκινάει* από την κορυφή u και *καταλήγει* στην κορυφή v (έτσι φαίνεται ότι οι ακμές έχουν κάποια *φορά*, στην προκειμένη από τη u προς τη v).

Αυτό που ξεχωρίζει τα δέντρα από τα υπόλοιπα γραφήματα είναι οι ακόλουθες δύο ιδιότητες:

1. Για κάθε κορυφή $u \in V$ υπάρχει ακμή που είτε ξεκινάει από τη u είτε καταλήγει στη u.

2. Η ρίζα r είναι η μόνη κορυφή του V στην οποία δεν καταλήγει κάποια ακμή.

3. Δεν υπάρχει σε αυτά (*διατεταγμένος* ή και *μη-διατεταγμένος*) *κύκλος* [1].

Τέλος, στα δέντρα θα υπάρχουν κάποιες κορυφές από τις οποίες δεν θα ξεκινάει καμία ακμή [2]. Τις κορυφές αυτές θα τις αποκαλούμε *φύλλα* του δέντρου.

Ας δούμε ένα παράδειγμα για να γίνει ευκολότερα κατανοητός ο παραπάνω ορισμός.

Παράδειγμα 4.1.2. Θεωρήστε το ακόλουθο δέντρο $T = (V, E, r)$:

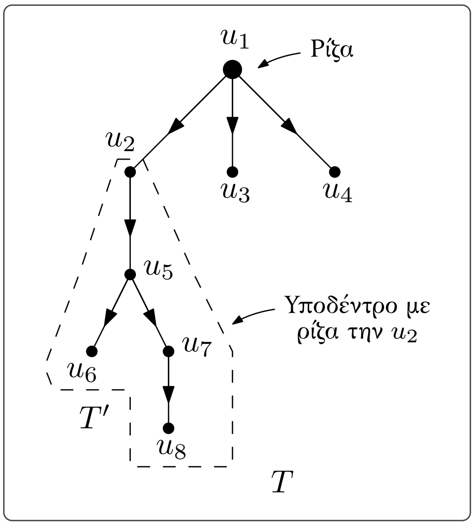
$$\begin{aligned}
&V = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}\\
&E = \{(u_1, u_2), (u_1, u_3), (u_1, u_4), (u_2, u_5), (u_5, u_6), (u_5, u_7), (u_7, u_8)\}\\
&r = u_1
\end{aligned}$$

Παρατηρήστε ότι φύλλα του T αποτελούν οι κορυφές u_3, u_4, u_6 και u_8.

Για να κάνουμε το παραπάνω συνδυαστικό αντικείμενο περισσότερο παραστατικό συνήθως το απεικονίζουμε χρησιμοποιώντας σημεία του επιπέδου για τις κορυφές και βέλη για τις ακμές, που ακολουθούν φυσικά τη φορά της ακμής (δες Σχήμα 4.1.1). Τη ρίζα συνήθως την τοποθετούμε στο πάνω μέρος του δέντρου και τις υπόλοιπες κορυφές από κάτω

[1] Ο ακριβής ορισμός ενός κύκλου είναι ο εξής: Έστω σύνολο κορυφών $V = \{u_1, \ldots, u_n\}$. Το σύνολο ακμών $E = \{(u_1, u_2), (u_2, u_3), \ldots, (u_{n-1}, u_n), (u_n, u_1)\}$ αποτελεί έναν διατεταγμένο κύκλο. Αν περιορίσουμε τις απαιτήσεις μας και ζητήσουμε οι κορυφές να συνδέονται μέσω ακμών με τον παραπάνω τρόπο, αδιαφορώντας όμως για τη φορά των ακμών τότε έχουμε έναν μη-διατεταγμένο κύκλο.

[2] Αυτό προκύπτει καθώς λόγω του 3. στο δέντρο δεν υπάρχουν κύκλοι. Μπορείτε να αποδείξετε τυπικά την ύπαρξη φύλλων σε ένα δέντρο;



Σχήμα 4.1.1: *Απεικόνιση του δέντρου στο Παράδειγμα 4.1.2 (και ενός υποδέντρου του).*

της (αντίθετα δηλαδή από τα δέντρα που βλέπουμε στον φυσικό κόσμο), προσπαθώντας να τοποθετούμε της κορυφές στην ίδια απόσταση από τη ρίζα ανάλογα με το *επίπεδο* που βρίσκονται (την έννοια του επιπέδου θα την ορίσουμε σε λίγο).

Ορισμός 4.1.3. Έστω δέντρο $T = (V, E, r)$ και $u_1, u_2 \in V$. Αν $(u_1, u_2) \in E$ θα λέμε ότι η u_1 είναι ο *γονέας* της u_2 και ότι η u_2 είναι το *παιδί* της u_1. Αν επιπλέον υπάρχει κορυφή $u_3 \in V$ και ακμή $(u_1, u_3) \in E$ θα λέμε ότι οι κορυφές u_2 και u_3 είναι *αδέρφια*. Τέλος, θα χρησιμοποιούμε τις λέξεις *απόγονος* και *πρόγονος* για να αναφερθούμε στις κορυφές του δέντρου, κατ' αναλογία με το νόημα των λέξεων αυτών στην καθομιλουμένη.

Παράδειγμα 4.1.4. Στο δέντρο του Παραδείγματος 4.1.2 οι κορυφές u_2, u_3, u_4 είναι αδέρφια, παιδιά της u_1 που αποτελεί τον γονέα της. Οι κορυφές u_5, u_6, u_7, u_8 αποτελούν απόγονους της u_2 και η u_5 είναι πρόγονος των u_6, u_7 και u_8.

Ορισμός 4.1.5. *Μονοπάτι* από μία κορυφή u σε μία κορυφή v ενός δέντρου είναι μια ακολουθία κορυφών που ξεκινάει από τη u, καταλήγει στη v και κάθε ενδιάμεση κορυφή (όπως και η v) είναι παιδί της προηγούμενη κορυφής στην ακολουθία. Οι κορυφές u και v είναι τα *άκρα* του μονοπατιού. Το *μήκος* του μονοπατιού είναι ο αριθμός των ακμών που αυτό περιέχει.

Παράδειγμα 4.1.6. Στο δέντρο του Παραδείγματος 4.1.2 η ακολουθία $[u_1, u_2, u_5, u_6]$ αποτελεί μονοπάτι με άκρα τις u_1 και u_6. Το μήκος του είναι 3.

Πρόταση 4.1.7. Έστω δέντρο $T = (V, E, r)$ και $u, v \in V$ όπου η v είναι απόγονος της u. Υπάρχει μοναδικό μονοπάτι στο T με άκρα τις u και v.

Απόδειξη. Παρόλο που εύκολα κάποιος καταλαβαίνει ότι αν υπήρχαν δύο διαφορετικά μονοπάτια με άκρα αυτές τις κορυφές τότε θα έπρεπε το T να περιέχει κύκλο (πράγμα που αντιβαίνει στην υπόθεσή μας ότι το T είναι δέντρο), για να αποδείξουμε προτάσεις που αφορούν δέντρα (και γενικότερα γραφήματα) θα πρέπει να είμαστε πιο προσεκτικοί και κυρίως, πιο τυπικοί.

Θα αποδείξουμε την πρόταση αυτή εφαρμόζοντας επαγωγή στο πλήθος κορυφών του δέντρου:

Επαγωγική Βάση: Θεωρούμε ότι το δέντρο T περιέχει δύο κορυφές, έστω u και v και έστω ότι η v είναι απόγονος της u (παιδί της u για την ακρίβεια). Προφανώς υπάρχει μοναδικό μονοπάτι με άκρα αυτές τις κορυφές, το $[u, v]$.

Επαγωγική Υπόδεση: Υποθέτουμε ότι για κάθε δέντρο με $|V| = n$ ισχύει ότι για οποιεσδήποτε δύο κορυφές του που η μία αποτελεί απόγονο της άλλης υπάρχει μοναδικό μονοπάτι που τις «ενώνει».

Επαγωγικό Βήμα: Έστω T δέντρο με $|V| = n + 1$ και έστω δύο κορυφές του $u, v \in V$ όπου η v είναι απόγονος της u. Εύκολα παρατηρούμε ότι κάποιος απόγονος της u θα είναι φύλλο του T (μπορεί και η ίδια η v), έστω η κορυφή w. Αν αφαιρέσουμε τη w από το δέντρο T θα πάρουμε ένα δέντρο T' με n κορυφές, για το οποίο θα ισχύει η επαγωγική υπόθεση, άρα θα υπάρχει μοναδικό μονοπάτι από τη u προς οποιονδήποτε απόγονό της.

Στην περίπτωση που η w είναι διαφορετική από τη v έχουμε τελειώσει, καθώς το μονοπάτι που «ενώνει» τη u με τη v στο T' υπάρχει και στο T και μάλιστα είναι μοναδικό (αφού η προσθήκη της w δεν μπορεί να δημιουργήσει μονοπάτι που δεν καταλήγει στη w).

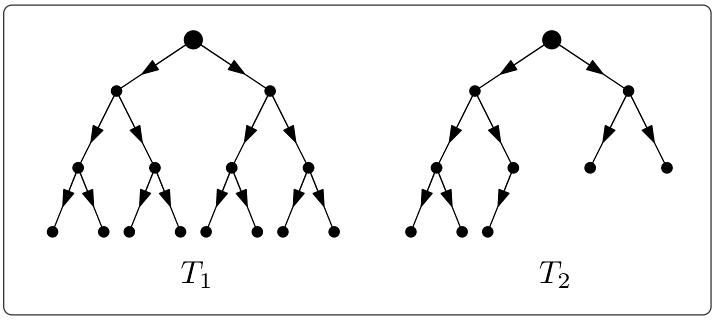
Στην περίπτωση τώρα που οι v και w ταυτίζονται, υποθέστε ότι $(x, v) \in E$ για κάποια κορυφή $x \in V$ του T (μπορεί η x να ταυτίζεται με τη u) και παρατηρήστε ότι η u συνδέεται με μοναδικό μονοπάτι με τη x στο T' (αν $x = u$ έχουμε το τετριμμένο μονοπάτι $[u]$), έστω το $[u, u_1, \ldots, u_k, x]$. Παρατηρήστε ότι η ακολουθία $[u, u_1, \ldots, u_k, x, v]$ αποτελεί μονοπάτι στο T με άκρα u και v και ότι είναι μοναδικό (καθώς το $[u, u_1, \ldots, u_k, x]$ είναι το μοναδικό μονοπάτι στο T με άκρα u και x). □

Ορισμός 4.1.8. Το *ύψος μιας κορυφής* u είναι το μήκος του (μοναδικού) μονοπατιού από τη ρίζα r προς τη u. Το σύνολο των κορυφών με το ίδιο ύψος απαρτίζει ένα *επίπεδο* κορυφών. Το *ύψος του δέντρου* ισούται με το μέγιστο ύψος των κορυφών του.

Παράδειγμα 4.1.9. Στο δέντρο του Παραδείγματος 4.1.2 η κορυφή u_7 έχει ύψος 3, το ύψος του δέντρου είναι 4 και οι κορυφές του χωρίζονται σε επίπεδα ως εξής:

Επίπεδο 0: $\{u_1\}$ (κορυφές ύψους 0)
Επίπεδο 1: $\{u_2, u_3, u_4\}$ (κορυφές ύψους 1)
Επίπεδο 2: $\{u_5\}$ (κορυφές ύψους 2)
Επίπεδο 3: $\{u_6, u_7\}$ (κορυφές ύψους 3)
Επίπεδο 4: $\{u_8\}$ (κορυφές ύψους 4)

Ορισμός 4.1.10. Ο *βαθμός* μιας κορυφής ισούται με το πλήθος των παιδιών της (των ακμών που ξεκινάνε από αυτήν την κορυφή).



Σχήμα 4.2.1: *Παραδείγματα δυαδικών δέντρων. Το T_1 είναι πλήρες ενώ το T_2 όχι. Το T_2 όμως είναι σχεδόν πλήρες.*

Ορισμός 4.1.11. Έστω δέντρο $T = (V, E, r)$. *Υποδέντρο* του T είναι ένα δέντρο που σχηματίζεται αν θεωρήσουμε ως ρίζα κάποια άλλη κορυφή $r' \in V \setminus \{r\}$, ως σύνολο κορυφών V' τις κορυφές που είναι απόγονοι της r' και σύνολο ακμών E' όλες τις ακμές του E που και τα δύο άκρα τους εμφανίζονται στο V'.

Παράδειγμα 4.1.12. Ένα υποδέντρο του δέντρου στο Παράδειγμα 4.1.2 είναι το $T' = (V', E', r')$ με $V' = \{u_2, u_5, u_6, u_7, u_8\}$, $E' = \{(u_2, u_5), (u_5, u_6), (u_5, u_7), (u_7, u_8)\}$ και $r' = u_2$. Το T' φαίνεται στο Σχήμα 4.1.1.

4.2 Δυαδικά δέντρα

Η ειδική κατηγορία δέντρων που θα μας απασχολήσει κατά κύριο λόγο σε αυτό το κεφάλαιο είναι τα *δυαδικά δέντρα*.

Ορισμός 4.2.1. Ένα δέντρο καλείται *δυαδικό* αν κάθε κορυφή του έχει βαθμό ≤ 2 (δες Σχήμα 4.2.1).

Πριν δούμε πως μπορούμε να αναπαραστήσουμε τα δυαδικά δέντρα ας αναπτύξουμε την ορολογία που χρειαζόμαστε για να κάνουμε την περιγραφή πιο απλή.

Εξ ορισμού κάθε κορυφή σε ένα δυαδικό δέντρο μπορεί να έχει το πολύ δύο παιδιά. Τα παιδιά αυτά θα τα διακρίνουμε στο *αριστερό* παιδί και στο *δεξί* παιδί. Στην αρχή η διάκριση αυτή δεν θα φέρει κάποιο βαθύτερο νόημα, θα γίνεται κάπως αυθαίρετα. Όταν θα ασχοληθούμε όμως με συγκεκριμένες δενδρικές δομές δεδομένων θα αναδειχθεί η σημασία της. Στην περίπτωση που μία κορυφή έχει ένα μόνο παιδί ας υποθέσουμε προς στιγμήν ότι είναι αριστερό (στις δομές δεδομένων που θα δούμε παρακάτω θα επιτρέπουμε σε μία κορυφή να έχει μόνο ένα παιδί που θα είναι δεξί). Αφού χαρακτηρίσουμε τα παιδιά της κάθε κορυφής (έστω και αυθαίρετα) θα υπάρχει μοναδικός τρόπος να σχεδιάσουμε το δέντρο στο χαρτί μας. Βασιζόμενοι σε αυτήν την απεικόνιση δίνουμε τους ακόλουθους ορισμούς [1].

[1] Παρατηρήστε ότι διαφορετικοί χαρακτηρισμοί των παιδιών των κορυφών ενός δέντρου δημιουργούν διαφορετικούς τρόπους απεικόνισης του ίδιου δέντρου. Σε όσα ακολουθούν θα θεωρούμε ότι αυτές οι απεικονίσεις αντιστοιχούν σε διαφορετικά δέντρα.

Ορισμός 4.2.2. Ένα δυαδικό δέντρο λέγεται *πλήρες* αν κάθε επίπεδό του έχει τον μέγιστο δυνατό αριθμό κορυφών (δες Σχήμα 4.2.1).

Τα πλήρη δυαδικά δέντρα έχουν μία πολύ σημαντική (για τους δικούς μας σκοπούς) ιδιότητα: Έχουν *μικρό ύψος* σε σχέση με το πλήθος των κορυφών τους. Άλλη μία κατηγορία δέντρων που έχει αυτήν την επιθυμητή ιδιότητα, αλλά πιο «χαλαρή» δομή από τα πλήρη δυαδικά δέντρα, είναι τα λεγόμενα *σχεδόν πλήρη* δυαδικά δέντρα.

Ορισμός 4.2.3. Ένα δυαδικό δέντρο λέγεται *σχεδόν πλήρες* αν όλα τα επίπεδά του *εκτός από το τελευταίο* έχουν τον μέγιστο δυνατό αριθμό κορυφών (δες Σχήμα 4.2.1) [1].

Τέλος, στη συνέχεια θα μας φανεί χρήσιμος ο παρακάτω ορισμός.

Ορισμός 4.2.4. Έστω δυαδικό δέντρο $T = (V, E, r)$ και $x \in V$. Έστω επίσης ότι η $y \in V$ είναι το αριστερό παιδί της x και η $z \in V$ το δεξί. Το υποδέντρο με ρίζα τη y το αποκαλούμε *αριστερό υποδέντρο της* x και το υποδέντρο με ρίζα τη z *δεξί υποδέντρο της* x. Αν η x δεν έχει αριστερό ή δεξί παιδί τότε το αντίστοιχο υποδέντρο θα είναι *κενό.*

4.2.1 Αναπαράσταση δυαδικών δέντρων με συνδεδεμένες λίστες

Τα δυαδικά δέντρα μπορούμε να τα αναπαραστήσουμε και με πίνακες και με λίστες. Ας ξεκινήσουμε με την αναπαράσταση μέσω διπλά συνδεδεμένων λιστών (την αναπαράσταση με πίνακες θα τη δούμε σε επόμενη παράγραφο, όταν θα ασχοληθούμε με τους *σωρούς*). Όπως θα δείτε ευθύς αμέσως χρησιμοποιούμε τη φιλοσοφία των διπλά συνδεδεμένων λιστών μόνο που χαλαρώνουμε την απαίτηση οι κόμβοι να συνδέονται με γραμμικό τρόπο (δες Σχήμα 4.2.3 για μία πρόγευση).

Ένας κόμβος ενός δυαδικού δέντρου θα περιέχει τα ακόλουθα πεδία:

1. Περιεχόμενο.

2. Δείκτης με τιμή τη διεύθυνση μνήμης που έχει αποθηκευτεί ο κόμβος που περιέχει το αριστερό παιδί του κόμβου (δες Σχήμα 4.2.2). Τον δείκτη αυτόν θα τον αποκαλούμε *δείκτη αριστερού παιδιού.*

3. Δείκτης με τιμή τη διεύθυνση μνήμης που έχει αποθηκευτεί ο κόμβος που περιέχει το δεξί παιδί του κόμβου (δες Σχήμα 4.2.2). Τον δείκτη αυτόν θα τον αποκαλούμε *δείκτη δεξιού παιδιού.*

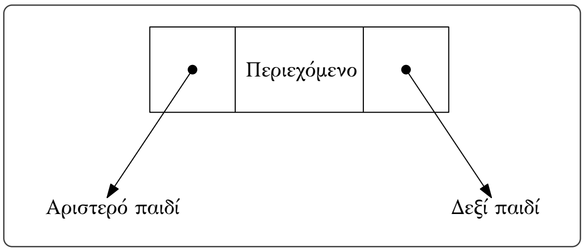
Έστω x μεταβλητή τύπου κόμβου δυαδικού δέντρου, θα γράφουμε:

- $x.item$ για να αναφερθούμε στο περιεχόμενο του κόμβου x,

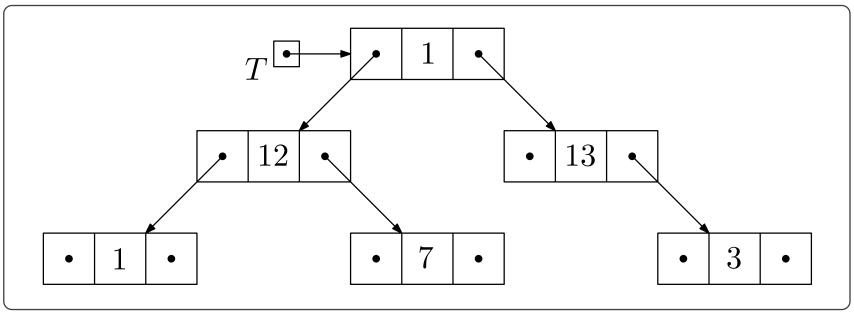
- $x.left$ για να αναφερθούμε στον δείκτη αριστερού παιδιού του κόμβου x και

- $x.right$ για να αναφερθούμε στον δείκτη δεξιού παιδιού του κόμβου x.

[1] Προφανώς ένα πλήρες δυαδικό δέντρο είναι και σχεδόν πλήρες.



Σχήμα 4.2.2: *Παράδειγμα κόμβου δυαδικού δέντρου.*



Σχήμα 4.2.3: *Παράδειγμα δυαδικού δέντρου.*

Για τη δημιουργία ενός κόμβου δυαδικού δέντρου θα χρησιμοποιήσουμε την εντολή **new tree node**. Επίσης θα χρησιμοποιούμε τις εντολές **address** και **node** κατά τα γνωστά. Τέλος, για να ξεχωρίσουμε τη ρίζα του δέντρου θα χρησιμοποιήσουμε τον *δείκτη ρίζας*, ο οποίος ταυτίζεται με τον δείκτη κεφαλής αυτής της ιδιότυπης «λίστας» (δες Σχήμα 4.2.3).

Παρατηρήστε ότι σύμφωνα με την αναπαράσταση αυτή είναι ρητά ορισμένο το αριστερό και δεξί παιδί μίας κορυφής [1].

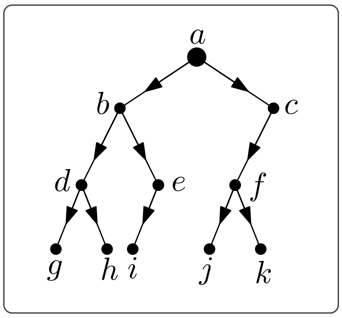
4.2.2 Πράξεις σε δυαδικά δέντρα

Μπορούμε να ορίσουμε πολλές πράξεις πάνω στα δυαδικά δέντρα, όπως για παράδειγμα την εύρεση του γονέα μιας δοσμένης κορυφής, την εύρεση της αδερφής μίας κορυφής, την καταμέτρηση του πλήθους των κορυφών, την εύρεσης του ύψους του δέντρου κ.λπ.. Εμείς θα δούμε μόνο τρεις από τις βασικές πράξεις, την αρχικοποίηση, τον έλεγχο για το αν το δέντρο είναι κενό και τη διαπέραση [2]. Η εισαγωγή και η εξαγωγή θα μας απασχολήσουν σε πιο εξειδικευμένες μορφές δυαδικών δέντρων.

1. *Αρχικοποίηση δυαδικού δέντρου:* Η αρχικοποίηση γίνεται δίνοντας στην ουσία την τιμή NIL στον δείκτη ρίζας. Για αυτόν τον σκοπό θα χρησιμοποιήσουμε την εντολή:

   $T \leftarrow$ **new binary tree**

[1] Αυτός είναι ο λόγος που στους πρότερους ορισμούς υπήρξε αυτή η διάκριση.
[2] Οι πράξεις που αναφέρθηκαν πριν αποτελούν πολύ καλές ασκήσεις!



Σχήμα 4.2.4: *Το δέντρο του Παραδείγματος 4.2.5.*

Ο χρόνος της αρχικοποίησης είναι $O(1)$.

2. *Έλεγχος για το αν το δέντρο είναι κενό:* Ελέγχουμε αν ο δείκτης ρίζας έχει τιμή NIL. Ο έλεγχος αυτός φυσικά γίνεται σε σταθερό χρόνο.

3. *Διαπέραση δυαδικού δέντρου:* Κάποιος θα μπορούσε να σκεφτεί πολλούς τρόπους να επισκεφτεί όλες τις κορυφές ενός δέντρου. Οι τρεις βασικότεροι είναι οι ακόλουθοι:

    - *Προδιατεταγμένη:* Επισκεπτόμαστε πρώτα τη ρίζα του δέντρου, μετά το αριστερό υποδέντρο και έπειτα το δεξί υποδέντρο.
    - *Ενδοδιατεταγμένη:* Επισκεπτόμαστε πρώτα το αριστερό υποδέντρο, μετά τη ρίζα και έπειτα το δεξί υποδέντρο.
    - *Μεταδιατεταγμένη:* Επισκεπτόμαστε πρώτα το αριστερό υποδέντρο, μετά το δεξί υποδέντρο και τέλος τη ρίζα.

Ένα παράδειγμα θα βοηθήσει αρκετά στην κατανόηση των παραπάνω διαδικασιών.

Παράδειγμα 4.2.5. Ας δούμε το δέντρο του Σχήματος 4.2.4. Οι κορυφές του θα επισκεφθούν με την ακόλουθη σειρά:

- *Προδιατεταγμένη:* $a, b, d, g, h, e, i, c, f, j, k$
- *Ενδοδιατεταγμένη:* $g, d, h, b, i, e, a, j, f, k, c$
- *Μεταδιατεταγμένη:* $g, h, d, i, e, b, j, k, f, c, a$

Θα δώσουμε τον αλγόριθμο μόνο για την προδιατεταγμένη διαπέραση του δέντρου. Ο αλγόριθμος αυτός χρησιμοποιεί αναδρομή [1].

[1] Μπορούμε φυσικά να αποφύγουμε την αναδρομή χρησιμοποιώντας μία στοίβα.

PreorderTraversal(T)

Είσοδος: Ο δείκτης ρίζας T ενός δυαδικού δέντρου
Έξοδος : Τίποτα (εσωτερικά ο αλγόριθμος εφαρμόζει την Process στο περιεχόμενο κάθε κορυφής του δέντρου)

if $T \neq$ NIL **then**
  Process(**node**(T).$item$)
  PreorderTraversal(**node**(T).$left$) *%Η πρώτη αναδρομική κλήση γίνεται με είσοδο το υποδέντρο με ρίζα το αριστερό παιδί της ρίζας του T*
  PreorderTraversal(**node**(T).$right$) *%Η δεύτερη με είσοδο το υποδέντρο με ρίζα το δεξί παιδί της ρίζας του T*

Σε ένα δέντρο με n κορυφές θα γίνουν $2n+1$ αναδρομικές κλήσεις (δύο για κάθε κορυφή, συν την αρχική κλήση), σε κάθε αναδρομική κλήση ο χρόνος που καταναλώνει ο αλγόριθμος είναι ο χρόνος της Process, έστω $O(T(n))$, άρα ο συνολικός χρόνος που χρειάζεται ο PreorderTraversal είναι $O(n \cdot T(n))$.

4.3 Δυαδικά δέντρα αναζήτησης

Θα επιβάλουμε επιπλέον δομή στα δυαδικά δέντρα καθώς αυτό μας εξυπηρετεί σε πάρα πολλές εφαρμογές. Σκοπός μας είναι να κρατήσουμε τις κορυφές του δέντρου (το περιεχόμενό τους για να είμαστε ακριβείς) διατεταγμένες, άρα βασική προϋπόθεση για να προχωρήσουμε είναι τα στοιχεία που θα αποθηκεύσουμε στο δέντρο να μπορούν να διαταχθούν (να είναι για παράδειγμα ακέραιοι, χαρακτήρες ή οτιδήποτε άλλο θέλουμε, αρκεί πρώτα να έχουμε συμφωνήσει σε μία διάταξη).

Ορισμός 4.3.1. Έστω δυαδικό δέντρο $T = (V, E, r)$ και $x, y \in V$. Το T είναι *δυαδικό δέντρο αναζήτησης* αν και μόνο αν:

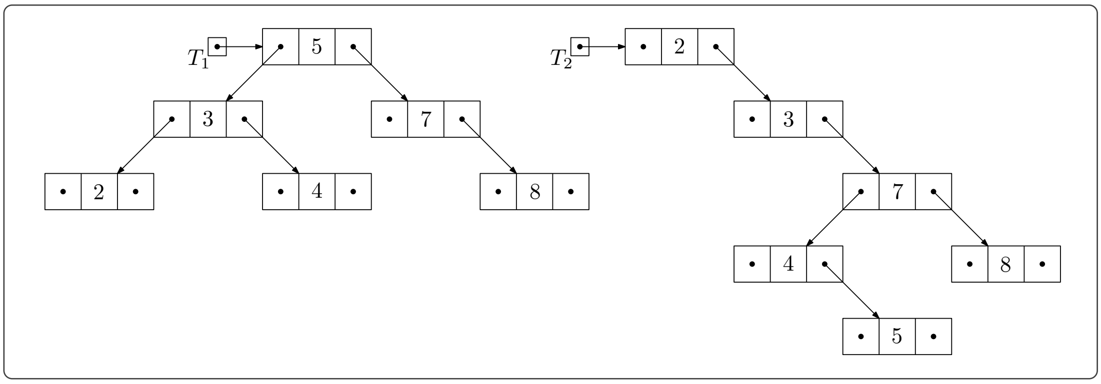
1. Αν η y βρίσκεται στο αριστερό υποδέντρο της x τότε $y.item \leq x.item$ και

2. αν η y βρίσκεται στο δεξί υποδέντρο της x τότε $y.item > x.item$,

όπου $<$ η διάταξη μεταξύ των στοιχείων του δέντρου.

Όπως αποκαλύπτει και το όνομα τα δυαδικά δέντρα αναζήτησης κάνουν πολύ πιο γρήγορη την αναζήτηση στοιχείων. Θα δούμε σε λίγο ότι ο χρόνος για την πράξη της αναζήτησης θα είναι γραμμικός ως προς το ύψος του δέντρου. Επιπλέον, το κόστος που θα πληρώσουμε για να πετύχουμε γρήγορη αναζήτηση θα είναι μικρό καθώς, ναι μεν στην πράξη της εισαγωγής θα χρειαστεί να τοποθετήσουμε το νέο στοιχείο σε κατάλληλη θέση, αλλά για αυτό χρειαζόμαστε επίσης γραμμικό ως προς το ύψος του δέντρου χρόνο.

Το ύψος του δέντρου θα αποτελέσει την σημαντικότερη παράμετρο σε αυτό το κεφάλαιο. Στο Σχήμα 4.3.1 βλέπουμε δύο δυαδικά δέντρα αναζήτησης με ακριβώς τα ίδια στοιχεία. Στο



Σχήμα 4.3.1: *Παραδείγματα δυαδικών δέντρων αναζήτησης.*

T_1 (που έχει ύψος 2) όλες οι πράξεις είναι πιο αποδοτικές από ότι στο T_2 (που έχει ύψος 4). Στη χειρότερη περίπτωση βέβαια το ύψος του δέντρου θα είναι ανάλογο του πλήθους των κορυφών του δέντρου (ίσο με το πλήθος των κορυφών μείον ένα για την ακρίβεια). Συνεπώς στη γενική περίπτωση (καθώς ενδιαφερόμαστε για την ανάλυση πολυπλοκότητας χειρότερης περίπτωσης) δεν κερδίζουμε κάτι. Ας πάρουμε για παράδειγμα την πράξη της αναζήτησης. Αν το δυαδικό δέντρο αναζήτησης έχει ύψος ανάλογο με το πλήθος των κορυφών, τότε η αναζήτηση θα χρειαστεί γραμμικό χρόνο, όσο χρειάζεται και η απλή γραμμική αναζήτηση δηλαδή. Οι δύο ειδικές κατηγορίες δέντρων όμως που είδαμε πριν (πλήρη και σχεδόν πλήρη δυαδικά δέντρα) έχουν ύψος της τάξης του λογάριθμου του πλήθους των κορυφών, συνεπώς για αυτές τις κατηγορίες δυαδικών δέντρων αναζήτησης πετυχαίνουμε «εκθετική» βελτίωση στην πράξη της αναζήτησης. Θα τα δούμε όλα αυτά πολύ διεξοδικά στη συνέχεια.

Αναζήτηση σε δυαδικό δέντρο αναζήτησης

Ο τρόπος που έχουν τοποθετηθεί τα στοιχεία στις κορυφές του δέντρου μας δίνει το δικαίωμα να αναζητήσουμε ένα στοιχείο επισκεπτόμενοι μόνο ένα μικρό κλάσμα των κορυφών του (όταν το δέντρο φυσικά έχει μικρό ύψος). Πριν δούμε τον αλγόριθμο (που υλοποιεί τη δυαδική αναζήτηση στην ουσία, δες τον αλγόριθμο BinarySearch), ας δούμε την κεντρική ιδέα που θα ακολουθήσουμε:

1. Εξετάζουμε αν η τιμή της ρίζας είναι ίση με την τιμή που ψάχνουμε:
   - Αν είναι ίση σταματάμε (έχουμε βρει το στοιχείο που ψάχνουμε).
   - Αν η τιμή που ψάχνουμε είναι *μικρότερη* προχωράμε στο *αριστερό υποδέντρο* («ξεχνώντας» το δεξί υποδέντρο) καθώς αυτό περιέχει τις τιμές που είναι μικρότερες είτε ίσες από το στοιχείο που ψάχνουμε.
   - Αν η τιμή που ψάχνουμε είναι *μεγαλύτερη* προχωράμε στο *δεξί υποδέντρο* (και «ξεχνάμε» το αριστερό) καθώς αυτό περιέχει τις τιμές που είναι μεγαλύτερες από το στοιχείο που ψάχνουμε.

2. Αν κάποτε φτάσουμε σε *κενό υποδέντρο* (εξετάσαμε κάποιο φύλλο δηλαδή) τότε σταματάμε καθώς το στοιχείο που ψάχνουμε δεν υπάρχει στο δέντρο.

Search(T, x)

Είσοδος: Ο δείκτης ρίζας T ενός δυαδικού δέντρου αναζήτησης και ένα στοιχείο x
Έξοδος : Η διεύθυνση στη μνήμη που είναι αποθηκευμένος ο κόμβος που περιέχει το x [1]

if $T = \text{NIL}$ **then** *% Κενό υποδέντρο*
    return "Δεν υπάρχει"
else if **node**$(T).item = x$ **then**
    return T
else if $x <$ **node**$(T).item$ **then**
    Search(**node**$(T).left, x)$
else
    Search(**node**$(T).right, x)$

Παράδειγμα 4.3.2. Ο αλγόριθμος Search με είσοδο το δέντρο T_1 του Σχήματος 4.3.1 και τον ακέραιο 6 θα κάνεις τις ακόλουθες αναδρομικές κλήσεις:

1. Search$(T_1, 6)$
2. Search(**node**$(T1).right, 6)$ (η είσοδος είναι το υποδέντρο με ρίζα την κορυφή που περιέχει το 7)
3. Search(**node**(**node**$(T1).right).left, 6)$ (η είσοδος είναι κενό υποδέντρο)

επιστρέφοντας τελικά το μήνυμα "Δεν υπάρχει". Παρατηρήστε ότι στο T_2 θα χρειάζονταν 5 αναδρομικές κλήσεις (αντί για 3 που χρειάστηκαν στο T_1).

Για να βρούμε τη χρονική πολυπλοκότητα του αλγορίθμου Search αρκεί να παρατηρήσουμε ότι σε κάθε αναδρομική κλήση προχωράμε από μία κορυφή σε ακριβώς ένα από τα παιδιά της, άρα στη χειρότερη περίπτωση θα γίνουν τόσες κλήσεις όσες και το ύψος του δέντρου συν ένα, επιπλέον σε κάθε κλήση γίνονται έλεγχοι που «κοστίζουν» σταθερό χρόνο. Συνεπώς ο χρόνος του αλγορίθμου είναι $O(h)$, όπου h το ύψος του δέντρου.

Αυτό όπως είπαμε και πριν στη γενική περίπτωση δεν μας δίνει πολύ καλό χρόνο καθώς το ύψος του δέντρου μπορεί να έχει μέγεθος ανάλογο με το πλήθος κορυφών του δέντρου, οπότε ο χρόνος του αλγορίθμου θα είναι $O(n)$, όπου n το πλήθος κορυφών. Συνεπώς είναι απαραίτητο να βρούμε τρόπους να κρατήσουμε το ύψος του δέντρου όσο πιο μικρό γίνεται.

Η παρακάτω πρόταση θα δικαιολογήσει γιατί μπήκαμε στον κόπο να ορίσουμε τα σχεδόν πλήρη δυαδικά δέντρα.

[1] Αν το x περιέχεται σε παραπάνω από έναν κόμβο τότε επιστρέφει την πρώτη εμφάνιση (την πιο κοντινή στη ρίζα), ενώ αν το x δεν εμφανίζεται καθόλου επιστρέφει μήνυμα σφάλματος.

Πρόταση 4.3.3. Έστω σχεδόν πλήρες δυαδικό δέντρο $T = (V, E, r)$, έστω h το ύψος του και n το πλήθος των κορυφών του. Ισχύει ότι $h = O(\log n)$ (στην πραγματικότητα $h \leq \log n$).

Απόδειξη. Αφού το ύψος του T είναι h, οι κορυφές του χωρίζονται σε $h + 1$ επίπεδα, από τα οποία όλα εκτός από το τελευταίο έχουν το μέγιστο δυνατό πλήθος κορυφών. Ας δούμε αναλυτικά πόσες κορυφές έχει το κάθε επίπεδο:

Επίπεδο 0:	1 κορυφή (τη ρίζα)
Επίπεδο 1:	2 κορυφές (αφού το δέντρο είναι σχεδόν πλήρες)
Επίπεδο 2:	2^2 κορυφές
	$\vdots$
Επίπεδο k:	2^k κορυφές
	$\vdots$
Επίπεδο $h-1$:	2^{h-1} κορυφές
Επίπεδο h:	τουλάχιστον 1 κορυφή (και το πολύ 2^h κορυφές)

Επομένως ισχύει ότι:

$$n \;\geq\; 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1 = 2^h$$

Λογαριθμούμε και τα δύο μέρη της παραπάνω ανισότητας (χωρίς αυτή να αλλάζει φορά):

$$\begin{aligned} \log n &\geq \log 2^h \\ \log n &\geq h \end{aligned}$$

Οπότε $h = O(\log n)$. □

Επομένως, όταν έχουμε να αντιμετωπίσουμε σχεδόν πλήρη δυαδικά δέντρα αναζήτησης η αναζήτηση μπορεί να γίνει πολύ πιο γρήγορα από την απλή γραμμική αναζήτηση. Αργότερα θα προχωρήσουμε αυτήν την ιδέα ακόμα περισσότερο και θα δούμε ότι το δέντρο δεν χρειάζεται καν να είναι σχεδόν πλήρες για να έχουμε τα επιθυμητά αποτελέσματα [1]. Ο λόγος που θα το κάνουμε αυτό είναι ότι η εισαγωγή κορυφής σε ένα σχεδόν πλήρες δυαδικό δέντρο αναζήτησης θα μας δίνει ένα δέντρο που να μεν είναι δυαδικό δέντρο αναζήτησης, δεν θα είναι όμως σχεδόν πλήρες. Δυστυχώς δεν είναι εφικτό να μετατρέψουμε αυτό το δέντρο σε σχεδόν πλήρες (τουλάχιστον όχι σε χρόνο που θα κάνει την πράξη της εισαγωγής αποτελεσματική). Συνεπώς θα πρέπει να βρούμε κάποια άλλη ιδιότητα των δυαδικών δέντρων αναζήτησης, που θα συντηρείται πιο εύκολα, και θα κρατάει το ύψος τους μικρό (δες Παράγραφο 4.5).

[1] Παρατηρήστε ότι δεν χρειάζεται το ύψος h να φράσσεται άνω από το $\log n$ για να ισχύει ότι $h = O(\log n)$, αρκεί να φράσσεται από ένα σταθερό πολλαπλάσιο του $\log n$.

Εισαγωγή στοιχείου σε δυαδικό δέντρο αναζήτησης

Για να διατηρήσουμε την ιδιότητα των δυαδικών δέντρων αναζήτησης θα πρέπει να εισάγουμε τα νέα στοιχεία σε κατάλληλη θέση. Παρατηρήστε ότι πάντα μπορούμε να τα εισάγουμε ως φύλλα στο δέντρο χωρίς το δέντρο να πάψει να είναι δυαδικό (φτάνει φυσικά να τα προσθέσουμε σε κορυφή με το πολύ ένα παιδί). Οπότε το ζητούμενο είναι να βρεθεί η κατάλληλη κορυφή της οποίας το νέο στοιχεία θα αποτελέσει παιδί. Να τονίσουμε ότι σε αυτήν τη φάση δεν ενδιαφερόμαστε να κρατήσουμε το ύψος του δέντρου χαμηλό, θα το αφήσουμε εντελώς στην «τύχη» του [1].

Παράδειγμα 4.3.4. Ας εισάγουμε το στοιχείο 6 στα δέντρα του Σχήματος 4.3.1. Στο T_1 πρέπει να την εισάγουμε ως αριστερό παιδί της κορυφής που περιέχει το 7, ενώ στο T_2 ως δεξί παιδί της κορυφής που περιέχει το 5.

Η κεντρική ιδέα της πράξης της εισαγωγής είναι η εξής:

1. Επαναλαμβάνουμε τη διαδικασία της αναζήτησης με αναζητούμενο το στοιχείο που πρέπει να εισαχθεί, έστω x.
2. Αν φτάσουμε σε φύλλο ή κατάλληλη κορυφή βαθμού 1 θα εισάγουμε μία νέα κορυφή στο δέντρο με περιεχόμενο το x ως αριστερό ή δεξιό παιδί (ανάλογα με το αν αυτή η κορυφή έχει τιμή «μικρότερη» ή «μεγαλύτερη» από x).
3. Αν συναντήσουμε κορυφή που περιέχει το x κατά την αναζήτηση θα συνεχίσουμε την αναζήτηση πηγαίνοντας προς τα αριστερά.

Insert(x, T)

Είσοδος: Ο δείκτης ρίζας T ενός δυαδικού δέντρου αναζήτησης και ένα στοιχείο x

Έξοδος : Ο δείκτης ρίζας του δυαδικού δέντρου αναζήτησης με την επιπλέον κορυφή

```
P ← address(new tree node(x))
if T = NIL then                                % Το δέντρο είναι κενό
    T ← P
    return T
Y ← T
Z ← NIL
while Y ≠ NIL
    Z ← Y                                      % Θυμόμαστε την προηγούμενη κορυφή
    if x ≤ node(Y).item then
        Y ← node(Y).left
    else
        Y ← node(Y).right
```

[1] Η εισαγωγή των ίδιων στοιχείων με διαφορετική σειρά μπορεί να οδηγήσει σε διαφορετικά δυαδικά δέντρα αναζήτησης, με μεγάλες διαφορές ενδεχομένως και στα ύψη τους.

```
13 if x ≤ node(Z).item then                  %Προσοχή, η κορυφή που δείχνει ο Z
                                                       μπορεί να έχει ένα παιδί:
14    node(Z).left ← P                       %- Το βήμα 10 μας εξασφαλίζει ότι
        node(Z).left = NIL
15 else
16    node(Z).right ← P                      %- Το βήμα 12 μας εξασφαλίζει ότι
        node(Z).right = NIL
17 return T
```

Στην χειρότερη περίπτωση ο Insert θα προσθέσει το νέο φύλλο ως παιδί του πιο απομακρυσμένου φύλλου από τη ρίζα, συνεπώς στο **while** θα γίνουν $O(h)$ επαναλήψεις, όπου h το ύψος του δέντρου. Ως συνέπεια ο χρόνος του Insert είναι $O(h)$.

Διαγραφή στοιχείου από δυαδικό δέντρο αναζήτησης

Η διαγραφή ενός στοιχείου (και της κορυφής που το περιέχει) όπως είναι φυσικό «κόβει» ένα δυαδικό δέντρο αναζήτησης στα δύο. Θα πρέπει να το «ξαναενώσουμε» με μεγάλη προσοχή ώστε να μην πληχθεί η ιδιότητα ότι τα στοιχεία στο αριστερό υποδέντρο είναι μικρότερα (είτε ίσα) από τη ρίζα του υποδέντρου, ενώ αυτά στο δεξί υποδέντρο μεγαλύτερα. Ας υποθέσουμε για αρχή ότι στο δέντρο δεν υπάρχουν κορυφές με ίδιο περιεχόμενο.

Θα χωρίσουμε την πράξη της διαγραφής σε τρεις κατηγορίες ανάλογα με το πόσα παιδιά έχει η προς διαγραφή κορυφή:

1. Αν *δεν έχει παιδιά* (είναι φύλλο) απλά τη διαγράφουμε.

2. Αν έχει *ένα μόνο παιδί* τη διαγράφουμε και στη θέση της βάζουμε το μοναδικό παιδί της (μαζί με ότι βρίσκεται από κάτω από αυτό φυσικά).

3. Αν έχει *δύο παιδιά*:

   (α′) Βρίσκουμε τον απόγονό της που έχει την αμέσως μεγαλύτερη τιμή [1], έστω την κορυφή y.

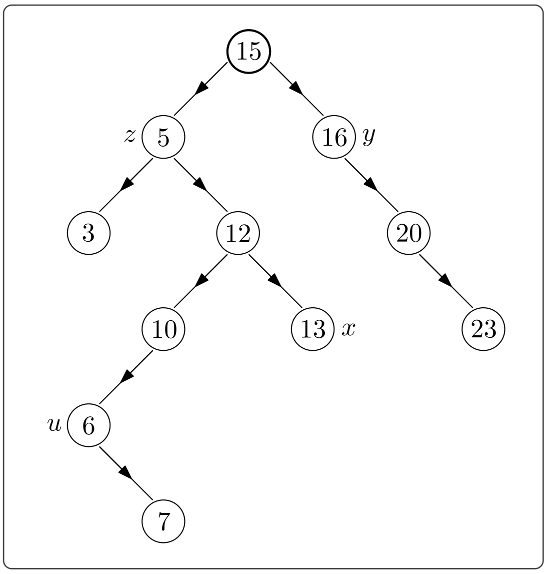
   (β′) Διαγράφουμε την κορυφή και στη θέση της βάζουμε τη y.

   (γ′) Όσον αφορά τη παλιά θέση της y, τη διαγράφουμε με διαγραφή τύπου 1 ή τύπου 2 ανάλογα με το αν έχει ή δεν έχει παιδί.

Οι παραπάνω περιπτώσεις θα γίνουν πιο εύκολα κατανοητές μέσω ενός παραδείγματος.

Παράδειγμα 4.3.5. Θεωρήστε το δέντρο του Σχήματος 4.3.2. Η πρώτη περίπτωση διαγραφής προκύπτει αν διαγράψουμε την κορυφή x που δεν έχει κανένα παιδί. Η διαγραφή της μας δίνει το δέντρο του Σχήματος 4.3.3 (που εξακολουθεί να είναι δυαδικό δέντρο αναζήτησης).

[1] Παρατηρήσετε ότι αυτή η κορυφή δεν θα έχει αριστερό παιδί.



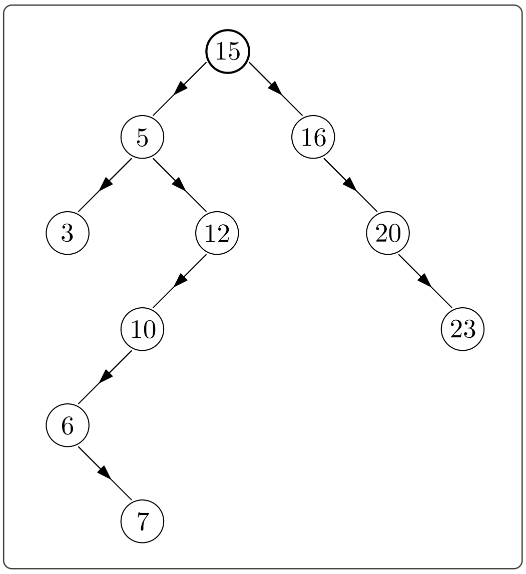
Σχήμα 4.3.2: *Το δέντρο του Παραδείγματος 4.3.5.*

Η δεύτερη περίπτωση προκύπτει κατά τη διαγραφή της y. Σε αυτή την περίπτωση απλά βάζουμε στη θέση της y το μοναδικό παιδί της (και ότι βρίσκεται από κάτω του) και καταλήγουμε στο δέντρο του Σχήματος 4.3.4.

Τέλος, η τρίτη περίπτωση διαγραφής προκύπτει όταν διαγράφουμε τη z. Σε αυτήν την περίπτωση όλο το δεξί υποδέντρο έχει στοιχεία μεγαλύτερα από το στοιχείο της z και η αριστερότερη κορυφή του υποδέντρου, η u, περιέχει τη μικρότερη τιμή στο υποδέντρο. Συνεπώς θα αντικαταστήσουμε την κορυφή z με τη u και έπειτα θα διαγράψουμε τη u με διαγραφή τύπου 2 (στο παράδειγμά μας η u έχει ένα παιδί). Τελικά καταλήγουμε στο δέντρο του Σχήματος 4.3.5.

Παρατηρήστε ότι και με τους τρεις τρόπους που διαχειριζόμαστε τη διαγραφή ενός στοιχείου οδηγούμαστε σε δέντρο που συνεχίζει να είναι δυαδικό δέντρο αναζήτησης. Επίσης, παρατηρήστε ότι στην τρίτη κατηγορία διαγραφής η ζητούμενη κορυφή θα είναι η «αριστερότερη» κορυφή στο δεξί υποδέντρο της κορυφής που διαγράφουμε. Επομένως για να τη βρούμε δεν χρειάζεται να κάνουμε κάποια σύγκριση τιμών, απλά να προσανατολιστούμε σωστά πάνω στο δέντρο. Για να βοηθήσουμε λίγο στην κατανόηση του αλγορίθμου που υλοποιεί τη διαγραφής θα τον χωρίσουμε σε τρεις επιμέρους αλγορίθμους, έναν για κάθε τύπο διαγραφής. Θα ξεκινήσουμε βλέποντας έναν αλγόριθμο που βρίσκει τον γονέα μίας δοσμένης κορυφής. Τον αλγόριθμο αυτόν θα τον χρειαστούμε καθώς κατά τη διαγραφή μίας κορυφής οι αλλαγές στο δέντρο γίνονται στους δείκτες αριστερού ή δεξιού παιδιού του γονέα της κορυφής που διαγράφουμε. Θα πρέπει λοιπόν να έχουμε έναν τρόπο να βρίσκουμε αυτήν την κορυφή [1].

[1] Θα μπορούσαμε βέβαια να προσδέσουμε έναν δείκτη στους κόμβους του δέντρου που θα έχει τιμή τη διεύθυνση στη μνήμη που είναι αποθηκευμένος ο πατέρας της κορυφής. Αυτό όμως θα έκανε πιο πολύπλοκη την αναπαράσταση της δομής.



Σχήμα 4.3.3: *Το δέντρο του Παραδείγματος 4.3.5 μετά τη διαγραφή της κορυφής x.*

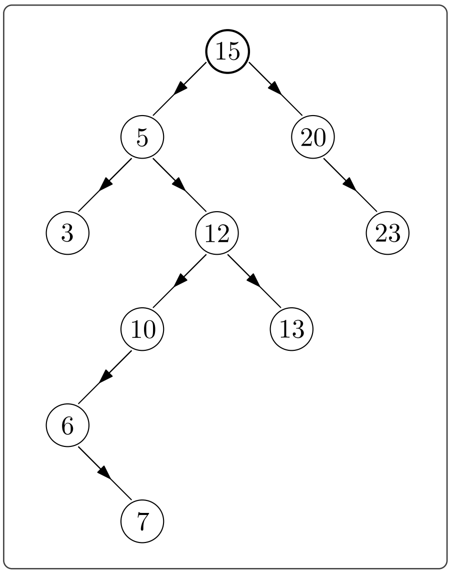
FindParent(T, P)

Είσοδος: Ο δείκτης ρίζας T ενός δυαδικού δέντρου αναζήτησης και δείκτης P που δείχνει την κορυφή της οποίας θέλουμε να βρούμε τον γονέα

Έξοδος : Δείκτης που δείχνει τον γονέα της κορυφής που δείχνει ο P (αν ο P δείχνει τη ρίζα επιστρέφει NIL)

1 **if** $T = \text{NIL}$ **then**
2   **return** "Κενό δέντρο"
3 $X \leftarrow T$
4 $Parent \leftarrow \text{NIL}$
5 **while** $X \neq \text{NIL}$
6   **if** $P = X$ **then**
7   **return** $Parent$
8   **else if node**$(P).item \leq$ **node**$(X).item$ **then**
9         $Parent \leftarrow X$
10         $X \leftarrow$ **node**$(X).left$
11   **else**
12         $Parent \leftarrow X$
13         $X \leftarrow$ **node**$(X).right$

Παρατηρήστε ότι στο βήμα 6 πρέπει να ελέγξουμε τους δύο δείκτες αν ταυτίζονται και *όχι* τα στοιχεία που περιέχουν οι αντίστοιχες κορυφές (όπως κάνουμε στο βήμα 8) γιατί το



Σχήμα 4.3.4: *Το δέντρο του Παραδείγματος 4.3.5 μετά τη διαγραφή της κορυφής y.*

στοιχείο που περιέχει η κορυφή που δείχνει ο P μπορεί να εμφανίζεται πολλές φορές στο δέντρο. Επίσης, ενδεχομένως να έπρεπε να προσθέσουμε την ακόλουθη γραμμή στο τέλος του αλγορίθμου:

return "Ο δείκτης δεν δείχνει σε κορυφή του δέντρου"

για να προφυλαχθούμε από την περίπτωση που ο χρήστης μας έδωσε λάθος δείκτη. Δεν μας απασχολούν όμως τόσο εξεζητημένες λεπτομέρειες καθώς οι «αλληλοεπιδράσεις» με τους χρήστες σε αυτές τις σημειώσεις έχουν κρατηθεί στα απολύτως απαραίτητα [1].

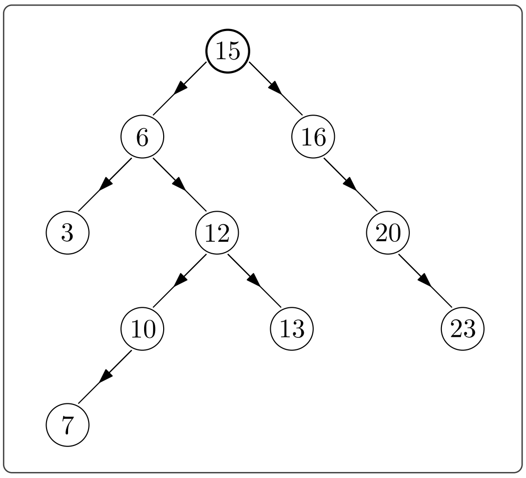
Ο αλγόριθμος FindParent έχει χρόνο $O(h)$, όπου h το ύψος του δέντρου, καθώς στο **While** στη χειρότερη περίπτωση θα χρειαστεί να φτάσουμε μέχρι το πιο απομακρυσμένο φύλλο από τη ρίζα. Ας δούμε τον αλγόριθμο που υλοποιεί τη διαγραφή τύπου 1:

DeletionCase1(T, P)

Είσοδος: Ο δείκτης ρίζας T ενός δυαδικού δέντρου αναζήτησης και δείκτης P που δείχνει την προς διαγραφή κορυφή
Έξοδος : Ο δείκτης ρίζας του δυαδικού δέντρου αναζήτησης μετά τη διαγραφή

$Parent \leftarrow$ FindParent(T, P)
if $Parent =$ NIL **then** *%Διαγράφουμε τη ρίζα (που δεν έχει παιδιά καθώς είμαστε στην πρώτη περίπτωση)*
  $T \leftarrow$ NIL

[1] Στην ουσία ο χρήστης είμαστε εμείς οι ίδιοι και, γνωρίζοντας την εσωτερική λειτουργία των αλγορίθμων, μπορούμε εύκολα να αποφεύγουμε λάθη αυτής της μορφής. Αυτό μας γλιτώνει από τον μπελά να ελέγχουμε όλα τα δυνατά λάθη που μπορεί να προκύψουν.



Σχήμα 4.3.5: *Το δέντρο του Παραδείγματος 4.3.5 μετά τη διαγραφή της κορυφής z.*

---

else if $P =$ **node**$(Parent).left$ **then** *%Αν ο P δείχνει σε αριστερό παιδί*
  node$(Parent).left \leftarrow$ NIL
else *%Αν ο P δείχνει σε δεξί παιδί*
  node$(Parent).right \leftarrow$ NIL
return T

Ο χρόνος του αλγορίθμου είναι $O(h)$ εξαιτίας του FindParent στο πρώτο βήμα.

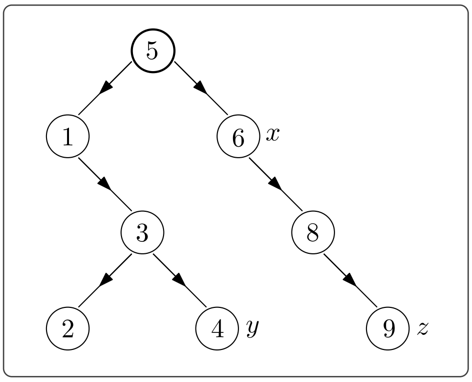
Ο αλγόριθμος που υλοποιεί τη διαγραφή τύπου 2 είναι ο ακόλουθος:

DeletionCase2(T, P)

Είσοδος: Ο δείκτης ρίζας T ενός δυαδικού δέντρου αναζήτησης και δείκτης P που δείχνει την προς διαγραφή κορυφή

Έξοδος : Ο δείκτης ρίζας του δυαδικού δέντρου αναζήτησης μετά τη διαγραφή

$Parent \leftarrow$ FindParent(T, P)
if **node**$(P).left \neq$ NIL **then** *%Το (μοναδικό) παιδί που έχει η κορυφή είναι αριστερό*
  $Child \leftarrow$ **node**$(P).left$
else
  $Child \leftarrow$ **node**$(P).right$
if $Parent =$ NIL **then** *%Διαγράφουμε τη ρίζα (που έχει ένα παιδί καθώς είμαστε στη δεύτερη περίπτωση)*
  $T \leftarrow Child$
else if $P =$ **node**$(Parent).left$ **then** *%Αν ο P δείχνει σε αριστερό παιδί*
  node$(Parent).left \leftarrow Child$



Σχήμα 4.3.6: *Όλες οι κορυφές στο δεξιό υποδέντρο της x έχουν τιμή μεγαλύτερη από την τιμή της x και απ' όλες αυτές η αριστερότερη έχει την μικρότερη τιμή. Όλες οι κορυφές των οποίων η y ανήκει στο δεξί υποδέντρο τους έχουν τιμή μικρότερη από την τιμή της y. Σε όσες ανήκει στο αριστερό υποδέντρο έχουν τιμή μεγαλύτερη, οπότε η πρώτη που θα συναντήσουμε (ανεβαίνοντας προς τη ρίζα) έχει τη ζητούμενη τιμή. Για την κορυφή z δεν ισχύει τίποτα από τα παραπάνω και ως συνέπεια έχει τη μεγαλύτερη τιμή στο δέντρο.*

---

else *%Αν ο P δείχνει σε δεξί παιδί*
  node$(Parent).right \leftarrow Child$
return T

Ο χρόνος και αυτού του αλγορίθμου είναι $O(h)$ (εξαιτίας πάλι του FindParent).

Στον αλγόριθμο που υλοποιεί τη διαγραφή τύπου 3 θα χρησιμοποιήσουμε την *υπορουτίνα* Successor που βρίσκει την κορυφή του δέντρου που έχει την αμέσως μεγαλύτερη τιμή από την τιμή μιας δοσμένης κορυφής. Διακρίνουμε τρεις περιπτώσεις:

1. Είδαμε ότι στην περίπτωση που το δεξί υποδέντρο είναι μη κενό τότε η ζητούμενη κορυφή είναι η αριστερότερη κορυφή σε αυτό το υποδέντρο. Αυτή είναι η περίπτωση που θα μας απασχολήσει στην διαγραφή τύπου 3 (καθώς διαγράφουμε κορυφή με δύο παιδιά).

2. Στην περίπτωση που το δεξί υποδέντρο είναι κενό τότε η ζητούμενη κορυφή είναι ο πρώτος πρόγονος της δοσμένης κορυφής που την περιέχει στο αριστερό υποδέντρο (δες Σχήμα 4.3.6).

3. Αν δεν υπάρχει κορυφή με αυτή την ιδιότητα (ούτε καν η ρίζα του δέντρου) τότε η δοσμένη κορυφή έχει τη μεγαλύτερη τιμή στο δέντρο.

Successor(T, P)

Είσοδος: Ο δείκτης ρίζας T δυαδικού δέντρου αναζήτησης και δείκτης P που δείχνει σε κορυφή του δέντρου

Έξοδος : Ο δείκτης που δείχνει στην κορυφή που περιέχει το αμέσως μεγαλύτερο στοιχείο από το $\mathbf{node}(P).item$

1 **if** $\mathbf{node}(P).right \neq \mathrm{NIL}$ **then** *%Μη κενό δεξί υποδέντρο*
2     $X \leftarrow \mathbf{node}(P).right$
3   **while** $\mathbf{node}(X).left \neq \mathrm{NIL}$ *%Μέχρι να βρούμε κορυφή που δεν έχει αριστερό παιδί*
4         $X \leftarrow \mathbf{node}(X).left$
5   **return** X
6 **else**
7     $Y \leftarrow \mathrm{FindParent}(T, P)$
8     $X \leftarrow P$
9   **while** $Y \neq \mathrm{NIL}$ **and** $X = \mathbf{node}(Y).right$ *%Δεν φτάσαμε στη ρίζα και η X είναι δεξί παιδί*
10         $X \leftarrow Y$
11         $Y \leftarrow \mathrm{FindParent}(T, Y)$
12   **if** $Y = \mathrm{NIL}$ **then** *%Φτάσαμε στη ρίζα*
13   **return** "Δεν υπάρχει μεγαλύτερο στοιχείο"
14   **else**
15   **return** Y

Ο χρόνος του Successor είναι $O(h^2)$ καθώς μέσα στο **while** στη γραμμή 9 καλούμε $O(h)$ φορές τον αλγόριθμο FindParent.

Παρατηρήστε όμως ότι στη διαγραφή τύπου 3 όμως γνωρίζουμε ότι η κορυφή έχει μη κενό δεξί υποδέντρο συνεπώς θα εκτελεστούν μόνο οι γραμμές 1–5 του Successor που έχουν χρόνο $O(h)$.

DeletionCase3(T, P)

Είσοδος: Ο δείκτης ρίζας T ενός δυαδικού δέντρου αναζήτησης και δείκτης P που δείχνει την προς διαγραφή κορυφή

Έξοδος : Ο δείκτης ρίζας του δυαδικού δέντρου αναζήτησης μετά τη διαγραφή

1 $Succ \leftarrow \mathrm{Successor}(T, P)$
2 **if** $\mathbf{node}(Succ).left = \mathrm{NIL}$ **and** $\mathbf{node}(Succ).right = \mathrm{NIL}$ **then** *%Αφαιρούμε τη $Succ$*
3     $T \leftarrow \mathrm{DeletionCase1}(T, Succ)$ *%Η $Succ$ δεν έχει κανένα παιδί*
4 **else**
5     $T \leftarrow \mathrm{DeletionCase2}(T, Succ)$ *%Η $Succ$ έχει 1 παιδί (δεξί)*
6 $Parent \leftarrow \mathrm{FindParent}(T, P)$ *%Ο δείκτης T πλέον δείχνει το τροποποιημένο δέντρο από τις γραμμές 1–5*
7 $\mathbf{node}(Succ).left \leftarrow \mathbf{node}(P).left$ *%Η $Succ$ παίρνει τη θέση της P*
8 $\mathbf{node}(Succ).right \leftarrow \mathbf{node}(P).right$

if $Parent = \text{NIL}$ **then** *%Διαγράφουμε τη ρίζα*
  $T \leftarrow Succ$
else if $P = \textbf{node}(Parent).left$ **then**
  $\textbf{node}(Parent).left \leftarrow Succ$
else
  $\textbf{node}(Parent).right \leftarrow Succ$
return T

Ο χρόνος του αλγορίθμου είναι $O(h)$ λόγω των Successor (στην περίπτωσή μας ο χρόνος του θα είναι $O(h)$), DeletionCase1 ή DeletionCase2 και FindParent που όλοι εκτελούνται από μία φορά.

Ο συνολικός αλγόριθμος διαγραφής είναι ο ακόλουθος:

Deletion(T, P)

Είσοδος: Ο δείκτης ρίζας T ενός δυαδικού δέντρου αναζήτησης και δείκτης P που δείχνει την προς διαγραφή κορυφή
Έξοδος : Ο δείκτης ρίζας του δυαδικού δέντρου αναζήτησης μετά τη διαγραφή

if $\textbf{node}(P).left = \text{NIL}$ **and** $\textbf{node}(P).right = \text{NIL}$ **then** *%Κανένα παιδί*
  return DeletionCase1(T, P)
else if $\textbf{node}(P).left \neq \text{NIL}$ **and** $\textbf{node}(P).right \neq \text{NIL}$ **then** *%Δύο παιδιά*
  return DeletionCase3(T, P)
else
  return DeletionCase2(T, P)

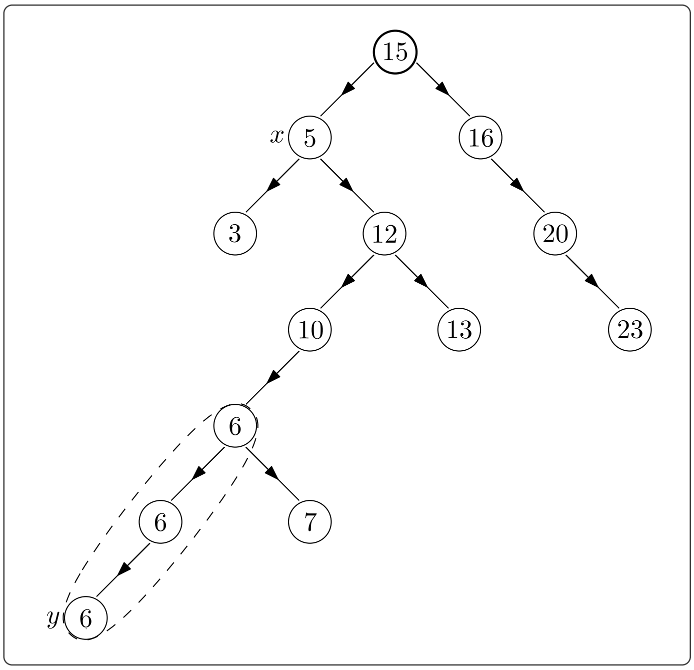
Ας δούμε τώρα πως η ύπαρξη κορυφών με ίδιο περιεχόμενο δημιουργεί πρόβλημα κατά τη διαγραφή τύπου 3 (στα δύο πρώτα είδη διαγραφής δεν υπάρχει θέμα). Έστω ότι διαγράφουμε κορυφή x για την οποία υπάρχουν πολλές κορυφές στο δέντρο με περιεχόμενο την αμέσως μεγαλύτερη τιμή από το $x.item$ (δες Σχήμα 4.3.7). Παρατηρήστε ότι ο Successor θα επιστρέψει την αριστερότερη από αυτές, έστω την y, και έπειτα ο DeletionCase3 θα αντικαταστήσει την x με την y. Οπότε στο δέντρο που θα προκύψει θα υπάρχουν κορυφές στο δεξί υποδέντρο της y (όπου πλέον βρίσκεται στην παλιά θέση της x) με περιεχόμενο $y.item$, πράγμα που αντιβαίνει στη δεύτερη προϋπόθεση του Ορισμού 4.3.1.

Ο πιο απλός τρόπος να διορθωθεί αυτό είναι να επιτρέψουμε στον Ορισμό 4.3.1 το δεξί υποδέντρο να έχει και τιμές ίσες με $x.item$ [1]:

2. αν η y βρίσκεται στο δεξί υποδέντρο της x τότε $y.item \geq x.item$,

Αν όμως επιμείνουμε στον παλιό ορισμό των δυαδικών δέντρων αναζήτησης θα πρέπει να αντιμετωπίσουμε ολόκληρη την «αλυσίδα» [2] των κορυφών που έχουν το ίδιο περιεχόμενο

[1] Παρατηρήστε ότι αυτό δεν δημιουργεί πρόβλημα στον αλγόριθμο αναζήτησης.
[2] Μπορείτε να σκεφτείτε γιατί θα είναι συνδεδεμένες με αυτόν τον τρόπο;



Σχήμα 4.3.7: *Παράδειγμα του προβλήματος με την ύπαρξη κορυφών με ίδιο περιεχόμενο.*

με την y σαν μία «μεγάλη» κορυφή και να αντικαταστήσουμε την x με αυτήν τη μεγάλη κορυφή (δες Σχήμα 4.3.8) [1].

4.4 Σωροί

Στην αρχή του κεφαλαίου, για να αναδείξουμε τη χρησιμότητα των μη-γραμμικών δομών, περιγράψαμε έναν τρόπο να οργανώσουμε τα στοιχεία μίας ουράς προτεραιότητας σε ένα δυαδικό δέντρο έτσι ώστε να μπορούμε να εξάγουμε το στοιχείο με την υψηλότερη προτεραιότητα πολύ γρήγορα («πληρώνοντας» κάτι παραπάνω για την εισαγωγή στοιχείου στην ουρά και γενικότερα για τη συντήρηση του δέντρου). Τα δέντρα αυτά καλούνται *Σωροί*. Ας δώσουμε έναν πιο τυπικό ορισμό.

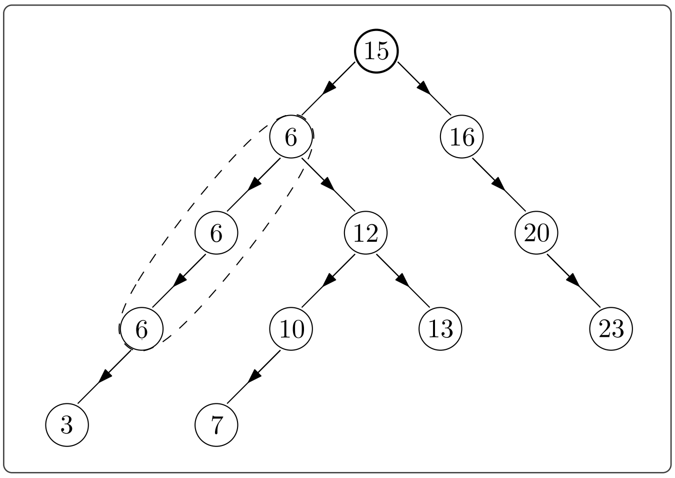
Ορισμός 4.4.1. Έστω $T = (V, E, r)$ δυαδικό δέντρο και $x, y \in V$. Το T είναι *σωρός* αν και μόνο αν:

1. Είναι σχεδόν πλήρες δυαδικό δέντρο και

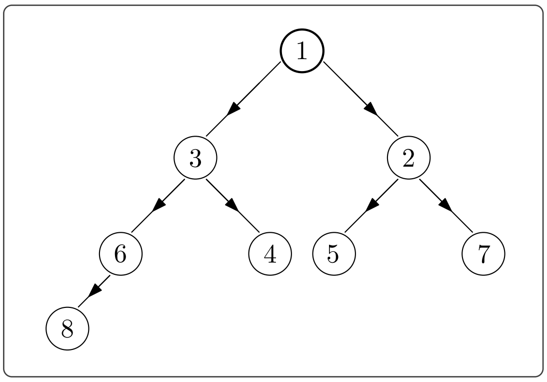
2. αν η y είναι απόγονος της x τότε $x.item \leq y.item$,

όπου $<$ η διάταξη μεταξύ των στοιχείων του δέντρου.

[1] Θα χρειαστεί στην πρώτη περίπτωση του Successor ο αλγόριθμος να μας επιστρέφει δύο δείκτες που θα δείχνουν στα άκρα αυτής της αλυσίδας κορυφών, και φυσικά ο DeletionCase3 να κάνει τις αλλαγές που περιγράψαμε παραπάνω. Οι λεπτομέρειες αφήνονται ως άσκηση.



Σχήμα 4.3.8: *Αντικατάσταση της x με ολόκληρη αλυσίδα κορυφών.*



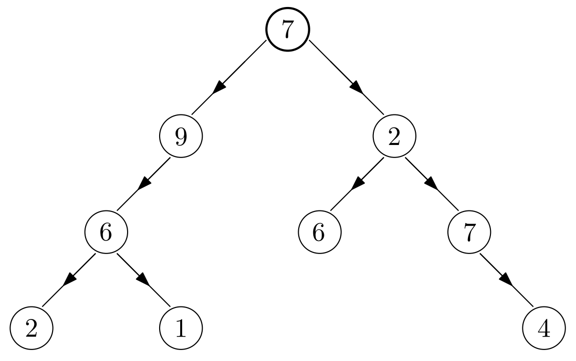
Σχήμα 4.4.1: *Παράδειγμα σωρού ελαχίστου.*

Παρατήρηση 4.4.2. Ο σωρός του Ορισμού 4.4.1 είναι *σωρός ελαχίστου* (Σχήμα 4.4.1). Θα μπορούσαμε αν θέλαμε στη δεύτερη ιδιότητα να είχαμε την αντίστροφη φορά της ανισότητας, ορίζοντας *σωρούς μεγίστου.*

Όπως θα δούμε οι δύο βασικές πράξεις σε έναν σωρό, η εισαγωγή και η εξαγωγή της ρίζας (του μικρότερου στοιχείου δηλαδή), χρειάζονται χρόνο $O(h)$, όπου h το ύψος του δέντρου. Καθώς ο σωρός είναι σχεδόν πλήρες δυαδικό δέντρο το ύψος του είναι λογαριθμικό ως προς το πλήθος των κορυφών του. Συνεπώς ο χρόνος που χρειάζονται οι βασικές πράξεις σε ένα σωρό είναι $O(\log n)$, όπου n το πλήθος στοιχείων που περιέχει.

Για να επιτύχουμε τον παραπάνω χρόνο θα πρέπει να υιοθετήσουμε μία διαφορετική αναπαράσταση των δυαδικών δέντρων από αυτήν που είδαμε στην Παράγραφο 4.2.1. Θα τα αναπαραστήσουμε χρησιμοποιώντας πίνακες.

Βέβαια θα πρέπει να εξασφαλίσουμε και ένα πρωτόκολλο επαύξησης του μεγέθους του πίνακα όποτε χρειαζόμαστε περισσότερο χώρο για την αποθήκευση της δομής.



$$M = [\underbrace{7}_{0}\ \ \underbrace{9\ \ 2}_{1}\ \ \underbrace{6\ \ \_\ \ 6\ \ 7}_{2}\ \ \underbrace{2\ \ 1\ \ \_\ \ \_\ \ \_\ \ \_\ \ \_\ \ 4}_{3}]$$

Επίπεδο:

Σχήμα 4.4.2: *Αναπαράσταση δυαδικού δέντρου με πίνακα.*

4.4.1 Αναπαράσταση δυαδικών δέντρων με πίνακες

Παρατηρήστε ότι για να αναπαραστήσουμε ένα δυαδικό δέντρο ύψους h θα πρέπει να χρησιμοποιήσουμε έναν πίνακα με $2^{h+1} - 1$ κελιά (όσο δηλαδή είναι το μέγιστο πλήθος κορυφών που μπορεί να έχει το δέντρο). Θα αποθηκεύσουμε τα στοιχεία του δέντρου *ανά επίπεδο* ως εξής:

- Η ρίζα (μοναδική κορυφή στο επίπεδο 0) αποθηκεύεται στο πρώτο κελί του πίνακα.
- Αποθηκεύουμε την κάθε κορυφή του επιπέδου, από τα αριστερά προς τα δεξιά, στο πρώτο κενό κελί του πίνακα. Αν από το επίπεδο λείπει κάποια κορυφή αφήνουμε το αντίστοιχο κελί «κενό» τοποθετώντας το σύμβολο _ (δες Σχήμα 4.4.2).

Η αναπαράσταση αυτή θα μας φανεί πολύ χρήσιμη στους σωρούς γιατί μας επιτρέπει να βρίσκουμε το αριστερότερο «διαθέσιμο» φύλλο στο *κάτω-κάτω επίπεδο* [1] πολύ πιο γρήγορα απ' ότι στην αναπαράσταση με συνδεδεμένες λίστες. Πάσχει όμως από το πρόβλημα της στατικής καταχώρησης στη μνήμη. Υπάρχουν φυσικά τρόποι να διορθώσουμε αυτό το πρόβλημα, για παράδειγμα μπορούμε όταν γεμίζει ο πίνακας να τον αντιγράφουμε σε έναν πίνακα με διπλάσιο μέγεθος (δημιουργώντας χώρο για ένα ακόμα επίπεδο) ή μπορούμε να αποθηκεύουμε το κάθε επίπεδο σε ξεχωριστό πίνακα και όποτε προκύπτει η ανάγκη για νέο επίπεδο (αυξάνεται δηλαδή το ύψος του δέντρου) να χρησιμοποιούμε έναν ακόμα πίνακα κατάλληλης διάστασης. Χάριν ευκολίας δεν θα μας απασχολήσει όμως αυτό [2].

Πρόταση 4.4.3. Έστω δυαδικό δέντρο $T = (V, E, r)$, με n κορυφές, και έστω M ο πίνακας που το αναπαριστά. Ισχύουν τα ακόλουθα:

[1] Το επίπεδο που περιέχει τις κορυφές με το μέγιστο ύψος στο δέντρο.
[2] Θα θεωρούμε ότι έχουμε «κανονική» δυναμική καταχώρηση.

1. Ο γονέας του στοιχείου $M[i]$ (αν $i > 1$) είναι το στοιχείο $M[\lfloor \frac{i}{2} \rfloor]$.

2. Το αριστερό παιδί του στοιχείου $M[i]$ (αν το $M[i]$ δεν είναι φύλλο) είναι το $M[2i]$ (αν $M[2i] = \_$ τότε δεν έχει αριστερό παιδί).

3. Το δεξί παιδί του στοιχείου $M[i]$ (αν το $M[i]$ δεν είναι φύλλο) είναι το $M[2i+1]$ (αν $M[2i+1] = \_$ τότε δεν έχει δεξί παιδί).

Απόδειξη. Έστω ότι στη θέση i του πίνακα αποθηκεύεται το περιεχόμενο της κορυφής u, έστω ότι η κορυφή u βρίσκεται στο επίπεδο k και ότι είναι η λ-οστή κορυφή του επιπέδου από τα αριστερά (Σχήμα 4.4.3). Παρατηρήστε ότι:

$$i \;=\; 2^0 + 2^1 + \cdots + 2^{k-1} + \lambda = 2^k - 1 + \lambda \tag{4.1}$$

1. Ας υποθέσουμε ότι η u είναι αριστερό παιδί (σε αυτή την περίπτωση το i είναι άρτιο). Ο γονέας της θα είναι αποθηκευμένος στην ακόλουθη θέση στον M (δες Σχήμα 4.4.3):

$$\begin{aligned} 2^0 + 2^1 + \cdots + 2^{k-2} + \frac{\lambda - 1}{2} + 1 &= 2^{k-1} - 1 + \frac{\lambda - 1}{2} + 1 \\ &= \frac{2^k + \lambda - 1}{2} \overset{(4.1)}{=} \frac{i}{2} \end{aligned}$$

Στην περίπτωση που η u είναι δεξί παιδί (σε αυτή την περίπτωση το i είναι περιττό) ο γονέας της θα είναι αποθηκευμένος στην ακόλουθη θέση:

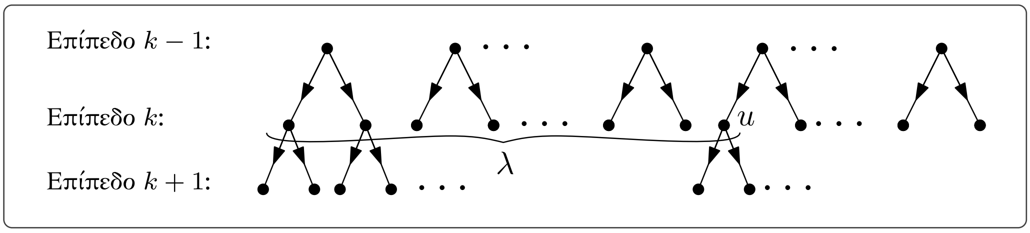
$$\begin{aligned} 2^0 + 2^1 + \cdots + 2^{k-2} + \frac{\lambda}{2} &= 2^{k-1} - 1 + \frac{\lambda}{2} \\ &= \frac{2^k + \lambda - 1}{2} - \frac{1}{2} \overset{(4.1)}{=} \left\lfloor \frac{i}{2} \right\rfloor \end{aligned}$$

2. Το αριστερό παιδί της u βρίσκεται στη θέση:

$$\begin{aligned} i + 2^k - \lambda + 2(\lambda - 1) + 1 &= i + 2^k + \lambda - 1 \\ &\overset{(4.1)}{=} 2i \end{aligned} \tag{4.2}$$

3. Από τη σχέση (4.2) προκύπτει ότι το δεξί παιδί της u βρίσκεται στη θέση $2i + 1$. □

Παρατηρήστε ότι σύμφωνα με την Πρόταση 4.4.3 στην αναπαράσταση με πίνακες μπορούμε να βρίσκουμε τον γονέα μίας κορυφής σε σταθερό χρόνο, σε αντίθεση με την αναπαράσταση με συνδεδεμένες λίστες.



Σχήμα 4.4.3: *Η κορυφή u στην Πρόταση 4.4.3*

4.4.2 Πράξεις σε σωρούς

Για τους σωρούς μας θα χρησιμοποιήσουμε σχεδόν πλήρη δυαδικά δέντρα που επιπλέον ικανοποιούν την ιδιότητα ότι στο τελευταίο επίπεδο (που περιέχει τις κορυφές με το μεγαλύτερο ύψος) οι κορυφές είναι κατανεμημένες «όσο πιο αριστερά γίνεται». Ο λόγος είναι ότι είναι πιο βολικό να έχουμε δέντρα αυτής της μορφής ως στόχο κατά την εισαγωγή στοιχείων στο δέντρο ή τη διαγραφή της ρίζας.

Παρατήρηση 4.4.4. Σε ένα σχεδόν πλήρες δυαδικό δέντρο με αυτήν την ειδική μορφή τα μόνο κενά κελιά στον πίνακα που το αναπαριστά θα βρίσκονται στο τέλος του και θα αντιστοιχούν στις κορυφές που λείπουν από το κάτω-κάτω επίπεδο.

Ας δούμε πως διαμορφώνονται οι βασικές πράξεις των σωρών όταν χρησιμοποιούμε πίνακες για την αναπαράστασή τους:

1. *Αρχικοποίηση σωρού:* Για να αρχικοποιήσουμε έναν σωρό θα ορίσουμε έναν πίνακα με ένα κελί που θα είναι κενό [1]. Αναφερόμαστε στον πίνακα αυτόν γράφοντας $H.matrix$ και χρησιμοποιούμε την εντολή:

   $H \leftarrow$ **new heap** [2]

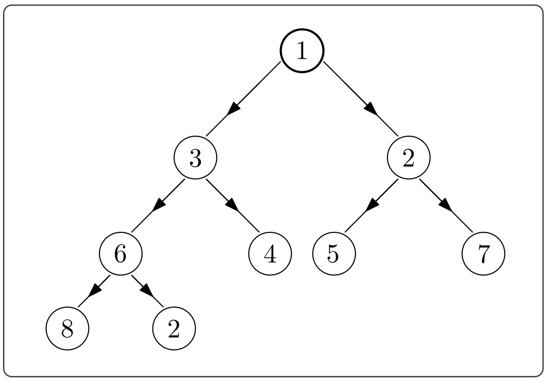
   Ο χρόνος της αρχικοποίησης είναι $O(1)$.

2. *Έλεγχος για το αν ένας σωρός είναι κενός:* Ελέγχουμε αν το πρώτο κελί του πίνακα είναι κενό (περιέχει το σύμβολο _ δηλαδή).

   IsEmpty(H)

   Είσοδος: Σωρός H
   Έξοδος : True αν ο σωρός είναι κενός, False διαφορετικά

[1] Έπειτα θα αυξάνουμε το μέγεθος του πίνακα ανάλογα με το μέγεθος του σωρού (σύμφωνα με όσα είπαμε πριν).
[2] Η H είναι μεταβλητή τύπου σωρού. Θα δούμε στη συνέχεια γιατί πρέπει να ορίσουμε τον σωρό σαν σύνθετο τύπο δεδομένων και όχι σαν έναν απλό πίνακα.



Σχήμα 4.4.4: *Αν εισάγουμε μία νέα κορυφή που περιέχει το στοιχείο 2 στον σωρό του Σχήματος 4.4.1 θα πάρουμε ένα σχεδόν πλήρες δυαδικό δέντρο, το οποίο όμως δεν θα είναι σωρός.*

```
if H.matrix[1] = _ then
    return True
else
    return False
```

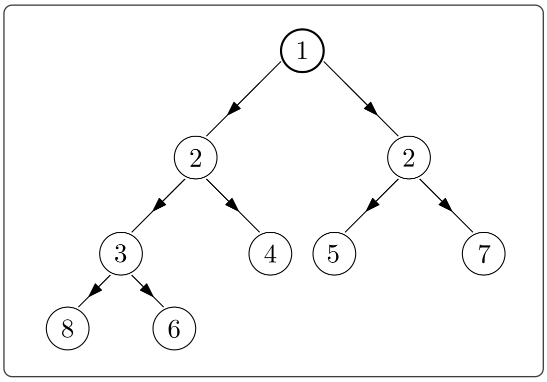
3. *Εισαγωγή στοιχείου:* Η εισαγωγή ενός στοιχείου και κατ' επέκταση μίας καινούριας κορυφής σε ένα σωρό θα πρέπει να γίνει με τέτοιο τρόπο ώστε το δέντρο που θα προκύψει να συνεχίσει να ικανοποιεί τις δύο ιδιότητες του σωρού:

   - Να είναι σχεδόν πλήρες δυαδικό δέντρο (με την ειδική μορφή που αναφέραμε παραπάνω).
   - Κάτω από κάθε κορυφή να βρίσκονται κορυφές με περιεχόμενο που έχει μεγαλύτερη τιμή.

   Θα χωρίσουμε την εισαγωγή σε δύο στάδια, σε κάθε ένα από τα οποία θα φροντίσουμε το νέο δέντρο να ικανοποιεί και μία από τις ιδιότητες.

   1^o *στάδιο:* Τοποθετούμε την προς εισαγωγή κορυφή στο *αριστερότερο διαθέσιμο φύλλο στο κάτω-κάτω επίπεδο* (αν το δέντρο είναι πλήρες την τοποθετούμε κάτω από το αριστερότερο φύλλο). Με αυτόν τον τρόπο ικανοποιούμε την πρώτη ιδιότητα (δες Σχήμα 4.4.4).

   Για να μπορέσουμε να κάνουμε την εισαγωγή σε χρόνο γραμμικό ως προς το ύψος του δέντρου θα πρέπει να βρούμε έναν τρόπο να ανακαλύπτουμε ποιο είναι το αριστερότερο διαθέσιμο φύλο στο κάτω-κάτω επίπεδο γρήγορα. Καθώς το δέντρο είναι σχεδόν πλήρες με την ειδική μορφή, από την Παρατήρηση 4.4.4, η ζητούμενη θέση στον πίνακα θα είναι αυτή που περιέχει το πρώτο κενό. Για να τη βρούμε όμως δεν



Σχήμα 4.4.5: *Παρατηρήστε ότι αντιμεταθέτοντας το περιεχόμενο της κορυφής που εισήγαμε με την κορυφή* 6 *(δες Σχήμα 4.4.4) επανορθώνουμε τη δεύτερη ιδιότητα του σωρού «τοπικά». Επαναλαμβάνοντας αυτό το βήμα, κινούμενοι προς την ρίζα μέχρι να βρούμε κορυφή με μικρότερη είτε ίση τιμή (την κορυφή* 1 *σε αυτό το παράδειγμα), θα έχουμε επανορθώσει τη δεύτερη ιδιότητα και «καθολικά». Έτσι το δέντρο μας θα είναι και πάλι σωρός.*

μπορούμε να αποφύγουμε τη γραμμική αναζήτηση (έστω και στον μισό πίνακα), γεγονός που θα μας στοιχίσει γραμμικό χρόνο ως προς το πλήθος των κορυφών (και όχι το ύψος του σωρού) [1].

Παράδειγμα 4.4.5. Ο πίνακας H που ακολουθεί αναπαριστά το δέντρο του Σχήματος 4.4.1. Παρατηρήστε ότι το αριστερότερο φύλλο στο κάτω-κάτω επίπεδο αντιστοιχεί στη θέση 9, την θέση δηλαδή που βρίσκεται το πρώτο κενό.

$$H.matrix = \begin{bmatrix} 1 & 3 & 2 & 6 & 4 & 5 & 7 & 8 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \end{bmatrix}$$

Για να μπορούμε να ξέρουμε εξαρχής ποιο είναι το πρώτο κενό κελί του πίνακα που αναπαριστά έναν σωρό θα χρειαστεί να προσθέσουμε στον σωρό έναν (ακέραιο) δείκτη που θα έχει ως τιμή το πλήθος στοιχείων που περιέχει ο σωρός [2]. Συνεπώς, κατά την αρχικοποίηση πέρα από τον ορισμό ενός πίνακα (με ένα κενό κελί) θα ορίσουμε και αυτόν τον δείκτη δίνοντάς του την τιμή 0. Αν η H είναι μία μεταβλητή τύπου σωρού θα αναφερόμαστε σε αυτόν τον δείκτη γράφοντας $H.length$ [3].

2^o *στάδιο:* Για να «επιδιορθώσουμε» το δέντρο έτσι ώστε να ικανοποιεί και τη δεύτερη ιδιότητα του σωρού θα πρέπει να «σπρώξουμε» προς τα πάνω (προς τη ρίζα δηλαδή) την κορυφή που εισήγαμε, αντιμεταθέτοντας το περιεχόμενό της με το περιεχόμενο της εκάστοτε κορυφής που ελέγχουμε. Θα σταματήσουμε τις αντιμεταθέσεις όταν βρούμε κορυφή που να περιέχει μικρότερη είτε ίση τιμή (δες Σχήμα 4.4.5).

[1] Παρατηρήστε ότι το ίδιο θα ίσχυε και αν αναπαριστούσαμε τους σωρούς με συνδεδεμένες λίστες.
[2] Πλέον ο σωρός θα αποτελεί μία σύνθετη δομή δεδομένων καθώς δεν θα αποτελείται μόνο από έναν πίνακα.
[3] Παρατηρήστε ότι το πρώτο κενό κελί του πίνακα είναι το κελί $H.length + 1$.

Η διαταραχή αυτή προκλήθηκε γιατί τοποθετήσαμε την κορυφή αυτή σε λάθος θέση. Το πρόβλημα αυτό, δηλαδή όταν μία κορυφή –*ακριβώς μία*- χαλάει τη δεύτερη ιδιότητα του σωρού, θα το αντιμετωπίσουμε πιο σφαιρικά, καθώς θα μας απασχολήσει και κατά τη διαγραφή (της ρίζας). Εκεί θα πάρουμε ένα φύλλο του δέντρου και θα αντικαταστήσουμε τη ρίζα με αυτό, οπότε πάλι θα έχουμε μία κορυφή που χαλάει τον σωρό.

Υπάρχουν δύο τρόποι με τους οποίους μόνο μία κορυφή μπορεί να μην ικανοποιεί τη δεύτερη ιδιότητα:

- Ο πρώτος είναι όταν ο γονέας της έχει τιμή μεγαλύτερη από αυτήν της κορυφής (όπως ενδεχομένως και κάποιοι πρόγονοί της). Σε αυτήν την περίπτωση πρέπει να «σπρώξουμε» την κορυφή προς τα πάνω.
- Ο δεύτερος είναι όταν τουλάχιστον ένα από τα παιδιά της κορυφής έχει τιμή μικρότερη από αυτήν της κορυφής (όπως ενδεχομένως και κάποιοι άλλοι απόγονοί της). Σε αυτήν την περίπτωση θα πρέπει να «σπρώξουμε» την κορυφή προς τα κάτω (προς τα φύλλα), ακολουθώντας την κατεύθυνση του παιδιού που έχει τιμή μικρότερη από της κορυφής μας (αν και τα δύο έχουν μικρότερη τιμή θα κατευθυνθούμε προς αυτό που έχει τη μικρότερη τιμή από τα δύο). Ο σωρός θα έχει επιδιορθωθεί όταν φτάσουμε σε κορυφή που και τα δυο παιδιά της θα έχουν τιμή μεγαλύτερη είτε ίση από της κορυφής που εξετάζουμε (δες Σχήμα 4.4.6).

Παρατηρήστε ότι μπορεί να συμβαίνει ακριβώς ένας από αυτούς τους δύο τρόπους, διότι σε αντίθετη περίπτωση, αν π.χ. υπήρχε και πρόγονος με τιμή μεγαλύτερη και απόγονος με τιμή μικρότερη τελικά θα υπάρχουν δύο κορυφές που «χαλάνε» τον σωρό. Εκτός αυτού, στις δύο περιπτώσεις που μας ενδιαφέρουν είτε η κορυφή δεν θα έχει παιδιά (όπως στην εισαγωγή) είτε δεν θα έχει γονέα (όπως στη διαγραφή).

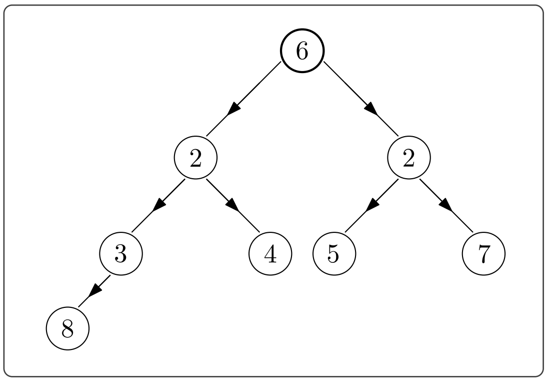
Για κάθε μία περίπτωση θα δώσουμε ξεχωριστό αλγόριθμο επιδιόρθωσης. Ας ξεκινήσουμε από την πρώτη:

HeapifyUp(M, i)

Είσοδος: Πίνακας M που αναπαριστά σχεδόν πλήρες δυαδικό δέντρο για το οποίο όλες οι κορυφές ικανοποιούν τη δεύτερη ιδιότητα, εκτός από την κορυφή στη θέση i

Έξοδος : Τίποτα (εσωτερικά επιδιορθώνεται ο σωρός)

$parent \leftarrow \lfloor \frac{i}{2} \rfloor$
if $parent \geq 1$ **and** $M[i] < M[parent]$ **then** *%Αλλιώς δεν έχουμε τίποτα να κάνουμε*
    $temp \leftarrow M[i]$
    $M[i] \leftarrow M[parent]$
    $M[parent] \leftarrow temp$
    HeapifyUp$(M, parent)$ *%Η κορυφή στη θέση $parent$ ενδεχομένως χαλάει τώρα τον σωρό*



Σχήμα 4.4.6: *Αν αφαιρέσουμε τη ρίζα στον σωρό του Σχήματος 4.4.5, προκειμένου να μην χαλάσουμε τη πρώτη ιδιότητα θα πρέπει να τοποθετήσουμε στη θέση της ρίζας το δεξιότερο φύλλο στο κάτω-κάτω επίπεδο. Αυτό μας αφήνει με ένα σχεδόν πλήρες δυαδικό δέντρο, που δεν είναι όμως σωρός, λόγω του ότι και τα δυο παιδιά της νέας ρίζας έχουν τιμή μικρότερη από αυτή.*

Παρατηρήστε ότι μετά την αντιμετάθεση της κορυφής στη θέση i με τον γονέα της μπορεί και πάλι να μην ικανοποιείται η δεύτερη ιδιότητα του σωρού. Αυτό θα συμβαίνει γιατί ο (νέος) γονέας της κορυφής θα έχει τιμή μεγαλύτερη από αυτή. Συνεπώς θα πρέπει να συνεχίσουμε να την «σπρώχνουμε» προς τα πάνω εφαρμόζοντας τον αλγόριθμο HeapifyUp (Βήμα 6).

Το μέγιστο πλήθος των αναδρομικών κλήσεων που θα κάνει ο HeapifyUp είναι $O(h)$, όπου h το ύψος. Καθώς το ύψος είναι λογαριθμικό ως προς το πλήθος κορυφών n ο χρόνος του HeapifyUp είναι $O(\log n)$.

Ο αλγόριθμος που επιδιορθώνει τον σωρό στη δεύτερη περίπτωση, όπου τουλάχιστον ένα από τα παιδιά της κορυφής στη θέση i έχει τιμή μικρότερη από αυτή, είναι ο εξής:

HeapifyDown(M, i)

Είσοδος: Πίνακας M που αναπαριστά σχεδόν πλήρες δυαδικό δέντρο για το οποίο όλες οι κορυφές ικανοποιούν τη δεύτερη ιδιότητα, εκτός από την κορυφή στη θέση i

Έξοδος : Τίποτα (εσωτερικά επιδιορθώνεται ο σωρός)

$left \leftarrow 2i$
$right \leftarrow 2i + 1$
$smallest \leftarrow i$
if $left \leq M.length$ **and** $M[i] > M[left]$ **then**
  $smallest \leftarrow left$
if $right \leq M.length$ **and** $M[smallest] > M[right]$ **then**
  $smallest \leftarrow right$

8 **if** $smallest \neq i$ **then**
9 $\quad temp \leftarrow M[i]$
10 $\quad M[i] \leftarrow M[smallest]$
11 $\quad M[smallest] \leftarrow temp$
12 $\quad$ HeapifyDown$(M, smallest)$ *%Η κορυφή στη θέση* $smallest$ *ενδεχομένως χαλάει τώρα τον σωρό*

Παρατηρήστε ότι η αντιμετάθεση της κορυφής στη θέση i με το μικρότερο από τα δύο παιδιά της ενδεχομένως να μην είναι αρκετή να επιδιορθώσει τον σωρό. Αυτό όμως δεν θα συμβαίνει λόγω της πρώτης περίπτωσης (ο γονέας της δηλαδή να έχει μεγαλύτερη τιμή), μπορεί όμως και πάλι τουλάχιστον ένα από τα παιδιά της να έχει μικρότερη τιμή από αυτή. Συνεπώς συνεχίζουμε να τη «σπρώχνουμε» προς τα κάτω εφαρμόζοντας τον αλγόριθμο HeapifyDown (Βήμα 12) μέχρι ο σωρός να έχει επιδιορθωθεί καθολικά. Στη χειρότερη περίπτωση θα χρειαστεί να γίνουν $O(h)$ κλήσεις του HeapifyDown στο Βήμα 12, συνεπώς ο χρόνος του είναι $O(h)$ ή αλλιώς $O(\log n)$.

Ο συνολικός αλγόριθμος επιδιόρθωσης σωρού είναι ο ακόλουθος:

Heapify(M, i)

Είσοδος: Πίνακας M που αναπαριστά σχεδόν πλήρες δυαδικό δέντρο για το οποίο όλες οι κορυφές ικανοποιούν τη δεύτερη ιδιότητα, εκτός από την κορυφή στη θέση i
Έξοδος : Τίποτα (εσωτερικά επιδιορθώνεται ο σωρός)

$parent \leftarrow \lfloor \frac{i}{2} \rfloor$
$left \leftarrow 2i$
$right \leftarrow 2i + 1$
if $parent \geq 1$ **and** $M[i] < M[parent]$ **then**
$\quad$ HeapifyUp(M, i)
else if $(left \leq M.length$ **and** $M[i] > M[left])$ **or** $(right \leq M.length$ **and** $M[i] > M[right])$ **then**
$\quad$ HeapifyDown(M, i)

Τονίζουμε το γεγονός ότι ο Heapify στην περίπτωση όπου τελικά η κορυφή που βρίσκεται στη θέση i δεν χαλάει τη δεύτερη ιδιότητα του σωρού δεν θα προκαλέσει καμία αλλαγή στο δέντρο.

Ο χρόνος του αλγορίθμου είναι $O(\log n)$ λόγω τον HeapifyUp και HeapifyDown (που θα εκτελεσθούν το πολύ μία φορά, αλλά όχι και οι δύο).

Έχοντας κάνει την κατάλληλη προεργασία για να αντιμετωπίσουμε και το δεύτερο στάδιο της εισαγωγής μπορούμε να παρουσιάσουμε τον αλγόριθμος που την υλοποιεί:

Insert(x, H)

Είσοδος: Σωρός H και στοιχείο x
Έξοδος : Τίποτα (εσωτερικά γίνεται η εισαγωγή)

1 **if** IsEmpty(H) **then**
2 $\quad H.matrix[1] \leftarrow x$
3 $\quad H.length \leftarrow 1$ *%Εδώ θα χρειαστεί να «αυξήσουμε» το μέγεθος του πίνακα*
4 **else**
$\quad$ *%Πρώτο στάδιο*
5 $\quad H.length \leftarrow H.length + 1$ *%Αν ο πίνακας είναι γεμάτος «αυξάνουμε» το μέγεθός του*
6 $\quad H.matrix[H.length] \leftarrow x$
$\quad$ *%Δεύτερο στάδιο*
7 $\quad$ Heapify$(H.matrix, H.length)$

Ο παραπάνω αλγόριθμος εκτελεί σταθερό πλήθος βασικών πράξεων σε όλα τα βήματά του εκτός από το Βήμα 7 που χρησιμοποιεί σαν υπορουτίνα τον Heapify. Επομένως ο χρόνος του είναι $O(\log n)$.

4. *Εξαγωγή ρίζας:*[1] Κατά την εξαγωγή (και διαγραφή) της ρίζας προκύπτουν αντίστοιχα προβλήματα με την εισαγωγή. Καταρχάς, η αφαίρεση της ρίζας μας αφήνει με δύο ξεχωριστά δέντρα τα οποίο θα πρέπει να «ενώσουμε» σε ένα. Ο τρόπος να το κάνουμε αυτό είναι να πάρουμε το δεξιότερο φύλλο στο κάτω-κάτω επίπεδο του σωρού και να το τοποθετήσουμε στη θέση της ρίζας. Αυτό έχει το πλεονέκτημα ότι το δέντρο που θα πάρουμε θα συνεχίσει να είναι σχεδόν πλήρες και μάλιστα θα έχει την ειδική μορφή που βάλαμε ως στόχο. Δεν θα είναι φυσικά σωρός (πήραμε ένα «μεγάλο στοιχείο» και το βάλαμε στη ρίζα του δέντρου, συνεπώς είναι πολύ πιθανό να μην ικανοποιεί τη δεύτερη ιδιότητα του σωρού).

   Παρατηρήστε ότι μετά από αυτήν την αλλαγή η ρίζα είναι η μόνη κορυφή που (ενδεχομένως) δεν ικανοποιεί τη δεύτερη ιδιότητα του σωρού. Αυτό μπορεί να συμβεί μόνο επειδή η ρίζα περιέχει στοιχείο με τιμή μεγαλύτερη από κάποιο από τα παιδιά της. Σε αυτήν την περίπτωση ο Heapify (πιο συγκεκριμένα ο HeapifyDown) μπορεί να επιδιορθώσει τον σωρό μας. Ας δούμε τον αλγόριθμο:

Deletion(H)

Είσοδος: Σωρός H
Έξοδος : Το στοιχείο στη ρίζα του σωρού (εσωτερικά γίνεται η διαγραφή της ρίζας)

1 **if** IsEmpty(H) **then**
2 $\quad$ **return** “Κενός σωρός”

[1] Παρατηρήστε ότι η ρίζα περιέχει τη μικρότερη τιμή του σωρού γι’ αυτό και την εξάγουμε.

$x \leftarrow H.matrix[1]$
% Πρώτο στάδιο
$H.matrix[1] \leftarrow H.matrix[H.length]$
$H.matrix[H.length] \leftarrow \_$ *% «Σβήνουμε» το δεξιότερο φύλλο στο κάτω-κάτω επίπεδο*
$H.length \leftarrow H.length - 1$ *% Αν «άδειασε» το κάτω-κάτω επίπεδο υποδιπλασιάζουμε το μέγεθος του πίνακα*
% Δεύτερο στάδιο
Heapify$(H.matrix, 1)$
return x

Ο χρόνος που χρειάζεται ο αλγόριθμος είναι $O(\log n)$ (εξαιτίας του Heapify).

4.4.3 Ουρές Προτεραιότητας με Σωρούς

Αν κοιτάξουμε προσεκτικά τις πράξεις της εισαγωγής και της εξαγωγής της ρίζας σε έναν σωρό ελαχίστου θα καταλήξουμε στο συμπέρασμα ότι η ρίζα του σωρού πάντα περιέχει τη μικρότερη τιμή από αυτές που εμφανίζονται στις κορυφές του.

Είναι εμφανές λοιπόν ότι για να αναπαραστήσουμε μία ουρά προτεραιότητας με έναν σωρό (ελαχίστου) το μόνο που θα χρειαστεί να κάνουμε είναι χρησιμοποιήσουμε τον βαθμό προτεραιότητας για την ταξινόμηση των κορυφών. Έτσι οι πράξεις της εισαγωγής και της εξαγωγής της ρίζας, όχι μόνο θα υλοποιήσουν σωστά τις πράξεις των ουρών προτεραιότητας, αλλά επιπλέον θα το κάνουν σε χρόνο γραμμικό ως προς το *ύψος* του δέντρου, που με τη σειρά του είναι της τάξης μεγέθους του λογαρίθμου του πλήθους των στοιχείων-κορυφών (ότι ακριβώς υποσχεθήκαμε στην αρχή του κεφαλαίου).

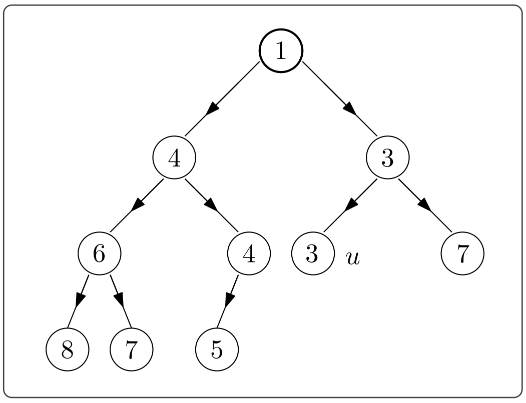
Οι δύο αλγόριθμοι θα λειτουργήσουν σωστά όμως *μόνο* στην περίπτωση όπου έχουμε στοιχεία με διαφορετικούς βαθμούς προτεραιότητας: Θυμηθείτε ότι σε μια ουρά προτεραιότητας στα στοιχεία με τον ίδιο βαθμό ακολουθούμε τη FIFO λογική. Όμως αυτό δεν εξασφαλίζεται από τους αλγορίθμους εισαγωγής και εξαγωγής (ναι μεν το στοιχείο με τον μικρότερο βαθμό προτεραιότητας θα «προωθηθεί» προς τα πάνω κατά την επιδιόρθωση του σωρού, ενδεχομένως όμως τελικά να καταλήξει πιο κοντά στη ρίζα από ένα στοιχείο με τον ίδιο βαθμό προτεραιότητας που ήδη υπήρχε στον σωρό, δες Σχήμα 4.4.7).

Για να λύσουμε αυτό το πρόβλημα θα εφαρμόσουμε το εξής τέχνασμα: Μαζί με τον βαθμό προτεραιότητας θα αποθηκεύουμε και έναν αύξων αριθμό [1] που θα υποδηλώνει τη σειρά που μπήκε το στοιχείο στον σωρό. Όποτε συγκρίνουμε στοιχεία με τον ίδιο βαθμό προτεραιότητας θα συμβουλευόμαστε και αυτόν τον αριθμό για να βρούμε ποιο από τα δύο τελικά θα προωθηθεί προς τη ρίζα [2].

Με αυτόν τον τρόπο όλες οι κορυφές θα ταξινομούνται βάση ενός μοναδικού «αναγνωριστικού» (των συνδυασμό βαθμού προτεραιότητας και αύξοντα αριθμού), συνεπώς οι δύο

[1] Μπορούμε εναλλακτικά να χρησιμοποιήσουμε ένα *timestamp* όπως π.χ. την ώρα που έγινε η εισαγωγή του στοιχείου.

[2] Και σε αυτήν την περίπτωση το μικρότερο είναι αυτό που «κερδίζει».



Σχήμα 4.4.7: *Παρατηρήστε ότι αν προσθέσουμε μία κορυφή με τιμή 3 στο δέντρο τότε αυτή θα καταλήξει πιο κοντά στη ρίζα από την κορυφή u.*

αλγόριθμοι θα φέρουν το επιθυμητό αποτέλεσμα ακόμα και στην περίπτωση όπου έχουμε στοιχεία με ίδιο βαθμό προτεραιότητας.

4.5 Δέντρα AVL [1]

Είδαμε ότι στα δυαδικά δέντρα αναζήτησης (και στους σωρούς) οι βασικές πράξεις χρειάζονται χρόνο $O(h)$, όπου h το ύψος του δέντρου. Στην περίπτωση των σωρών καταφέραμε να διατηρήσουμε ένα σχεδόν πλήρες δυαδικό δέντρο, επιτυγχάνοντας έτσι ύψος μικρότερο είτε ίσο από $\log n$ (n το πλήθος των κορυφών του δέντρου). Στόχος μας είναι να καταφέρουμε κάτι αντίστοιχο και για τα δυαδικά δέντρα αναζήτησης. Για να το επιτύχουμε αυτό δεν είναι αναγκαίο φυσικά το δέντρο μας να είναι σχεδόν πλήρες. Το μόνο που χρειαζόμαστε είναι να διατηρήσουμε το ύψος του $O(\log n)$.

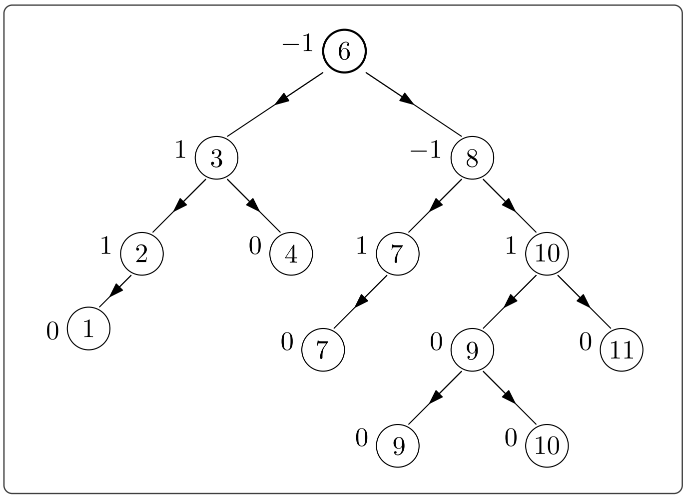
Σύμβαση 4.5.1. Σε όσα ακολουθούν θα χρειαστεί να αλλάξουμε τον ορισμό του ύψους ενός δέντρου (δες Ορισμό 4.1.8), όχι ουσιαστικά όμως. Στην ουσία θα μετράμε κορυφές αντί για ακμές στο μονοπάτι που ενώνει μία κορυφή με τη ρίζα του δέντρου (Ορισμός 4.1.5). Έτσι το δέντρο που αποτελείται από μία μόνο κορυφή (τη ρίζα του) θα έχει ύψος 1 και το κενό δέντρο ύψος 0.

Ορισμός 4.5.2. Έστω $T = (V, E, r)$ δυαδικό δέντρο αναζήτησης. Το T είναι *δέντρο AVL* αν και μόνο αν για κάθε $u \in V$ τα ύψη του αριστερού και του δεξιού υποδέντρου του διαφέρουν κατά απόλυτη τιμή το πολύ κατά 1 (δες Σχήμα 4.5.1).

Η ακόλουθη πρόταση μας εξασφαλίζει ότι ένα δέντρο AVL έχει ύψος λογαριθμικό ως προς το πλήθος των κορυφών του.

Πρόταση 4.5.3. Έστω δέντρο AVL $T = (V, E, r)$ με $|V| = n$. Το ύψος του T είναι $O(\log n)$.

[1] Το όνομα προέρχεται από τα αρχικά των G. Adelson-Velsky και E. Landis που τα εισήγαγαν.



Σχήμα 4.5.1: *Δίπλα σε κάθε κορυφή έχουμε σημειώσει τη διαφορά στο ύψος των δύο υποδέντρων (ύψος αριστερού μείον ύψος δεξιού). Καθώς η διαφορά για κάθε κορυφή είναι* $-1, 0$ *ή* 1 *το δέντρο αυτό είναι AVL.*

Απόδειξη. Αντί να βρούμε ένα άνω φράγμα για το ύψος h του T (που να σχετίζεται με το n), αρκεί αν βρούμε ένα κάτω φράγμα για το n (που να σχετίζεται με το h).

Έστω $f(h)$ το *ελάχιστο* πλήθος κορυφών που μπορεί να έχει ένα δέντρο AVL ύψους h. Παρατηρούμε ότι $f(1) = 1$ και $f(2) = 2$. Για $h \geq 3$ ισχύει ότι:

$$f(h) = f(h-1) + f(h-2) + 1$$

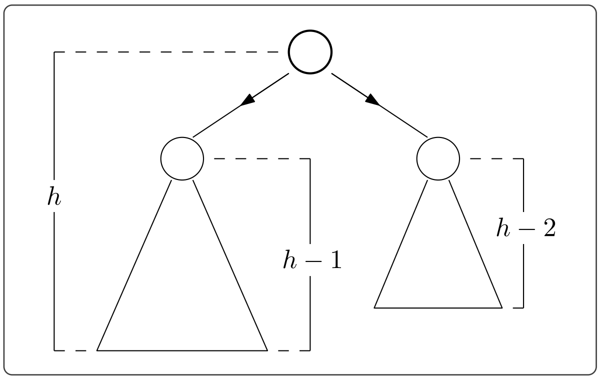
καθώς το δέντρο AVL με τις λιγότερες κορυφές με ύψος h θα απαρτίζεται από το δέντρο AVL ύψους $h-1$ με τις λιγότερες κορυφές και το δέντρο AVL ύψους $h-2$ με τις λιγότερες κορυφές (Σχήμα 4.5.2). Οπότε η τιμή $f(h)$ δίνεται από την ακόλουθη αναδρομική σχέση:

$$\begin{cases} f(h) = f(h-1) + f(h-2) + 1 \\ f(2) = 2 \\ f(1) = 1 \end{cases}$$

Η συνάρτηση f είναι αύξουσα [1] οπότε:

$$\begin{aligned} f(h) &\geq 2 \cdot f(h-2) \\ &\geq 2^2 \cdot f(h-4) \\ &\vdots \\ &\geq 2^i \cdot f(h-2i) \end{aligned}$$

[1] Προφανώς το δέντρο AVL ύψους $h-1$ με τις λιγότερες κορυφές θα έχει το πολύ όσες κορυφές έχει το δέντρο AVL ύψους h με τις λιγότερες κορυφές.



Σχήμα 4.5.2: *Παρατηρήστε ότι στο δέντρο AVL ύψους h με το ελάχιστο πλήθος κορυφών η ρίζα θα πρέπει να έχει διαφορά αριστερού και δεξιού υποδέντρου 1 ή -1.*

όπου το i είναι τέτοιο ώστε $h - 2i = 2$ (καθώς γνωρίζουμε το $f(2)$), δηλαδή $i = \frac{h-2}{2}$. Συνεπώς:

$$\begin{aligned} f(h) &\geq 2^{\frac{h-2}{2}} \cdot f(2) \\ &= 2^{\frac{h-2}{2}} \cdot 2 \\ &= 2^{\frac{h-2}{2}+1} \end{aligned}$$

Λογαριθμούμε (με βάση το 2) και τα δύο μέλη της ανισότητας και παίρνουμε:

$$\begin{aligned} \log f(h) &\geq \log 2^{\frac{h-2}{2}+1} \\ &= \frac{h-2}{2} + 1 \end{aligned}$$

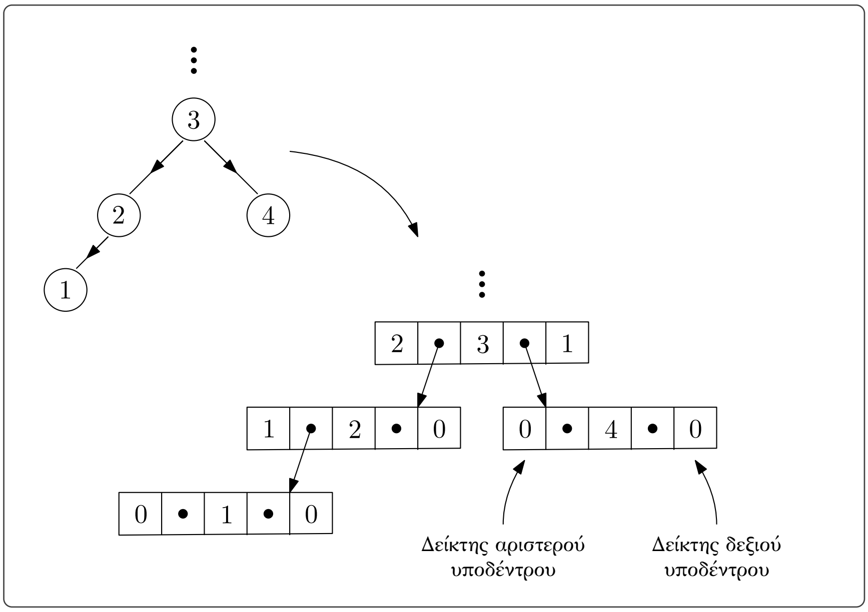
Οπότε $h \leq 2 \log f(h)$, και αφού $n \geq f(h)$ έπεται ότι $h = O(\log n)$. □

Καθώς το ύψος των δέντρων AVL είναι αρκούντως μικρό, το πρόβλημα που θα κληθούμε (ξανά) να λύσουμε είναι πως να διατηρήσουμε την ιδιότητά τους μετά από μία εισαγωγή ή μία διαγραφή. Πρώτα από όλα θα πρέπει να γνωρίζουμε πότε η ιδιότητα αυτή παραβιάζεται για να κάνουμε διορθωτικές κινήσεις πάνω στο δέντρο.

Θα αναπαραστήσουμε τα δέντρα AVL με διπλά συνδεδεμένες λίστες (όπως κάναμε για τα απλά δυαδικά δέντρα αναζήτησης), θα χρειαστεί όμως να προσθέσουμε δύο ακόμα πεδία στους κόμβους τους, ένα για το ύψος του αριστερού υποδέντρου και ένα για το ύψος του δεξιού. Έστω x μεταβλητή τύπου κόμβου δέντρου AVL, θα γράφουμε:

- $x.left\_subtree$ για να αναφερθούμε στο ύψος του αριστερού υποδέντρου (*δείκτης αριστερού υποδέντρου*) και

- $x.right\_subtree$ για να αναφερθούμε στο ύψος του δεξιού υποδέντρου (*δείκτης δεξιού υποδέντρου*).



Σχήμα 4.5.3: *Αναπαράσταση δέντρου AVL.*

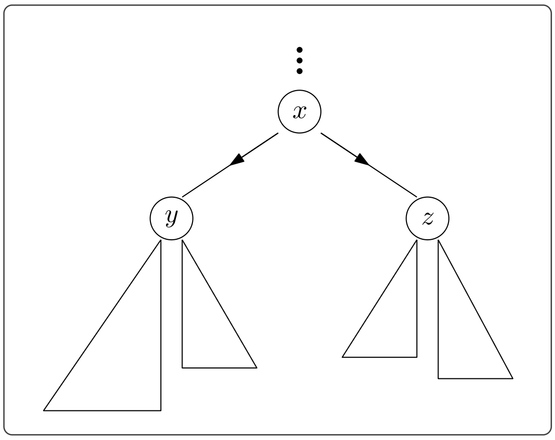
Ένα παράδειγμα φαίνεται στο Σχήμα 4.5.3. Κάθε καινούργια κορυφή που εισάγεται στο δέντρο (καθώς, όπως είδαμε, στα δυαδικά δέντρα αναζήτησης οι καινούργιες κορυφές εισάγονται ως φύλλα) θα έχει τιμή 0 και στους δύο δείκτες υποδέντρων. Φυσικά η εισαγωγή ενδέχεται να αλλάξει τις τιμές για τους πρόγονους αυτής της κορυφής (δηλαδή στις κορυφές που ανήκουν στο μονοπάτι από τη ρίζα προς αυτήν την κορυφή). Είναι σημαντικό ότι οι αλλαγές στα ύψη θα προκύψουν μόνο σε αυτές τις κορυφές (δηλαδή σε $O(\log n)$ κορυφές) και όχι σε όλες τις κορυφές του δέντρου. Ο τρόπος που υπολογίζουμε τα νέα ύψη των υποδέντρων για αυτές τις κορυφές είναι απλός. Έστω x μία κορυφή, y το αριστερό παιδί της και z το δεξί. Το ύψος του αριστερού και δεξιού υποδέντρου της x δίνεται από τους ακόλουθους τύπους (δες Σχήμα 4.5.4):

$$x.left\_subtree = \max\{y.left\_subtree, y.right\_subtree\} + 1$$
$$x.right\_subtree = \max\{z.left\_subtree, z.right\_subtree\} + 1$$

Έχουμε ήδη μιλήσει για τα δύο από τα συνολικά τρία στάδια που χρειαζόμαστε για να κάνουμε εισαγωγή σε ένα δέντρο AVL. Προτού μιλήσουμε και για το τρίτο ας ανακεφαλαιώσουμε:

1^o *στάδιο*: Εισάγουμε την καινούργια κορυφή ως φύλλο του δέντρου, σε κατάλληλη όμως θέση (δες Σελίδα 94).

2^o *στάδιο*: Υπολογίζουμε τα ύψη (και τις διαφορές στα ύψη) για τους προγόνους της κορυφής που μόλις εισήγαμε. Παρατηρήστε ότι ενδέχεται το ύψος σε (ακριβώς) ένα από τα δύο υποδέντρα κάποιας κορυφής να αυξηθεί κατά ένα. Αυτό σημαίνει ότι οι διαφορές στα ύψη των υποδέντρων για τους προγόνους της κορυφής που εισήγαμε μπορούν να ανήκουν



Σχήμα 4.5.4: *Υπολογισμός των υψών των υποδέντρων της x.*

πλέον στο σύνολο $\{-2,-1,0,1,2\}$. Αν υπάρχει κορυφή με διαφορά -2 ή 2 θα χρειαστεί να επιδιορθώσουμε το δέντρο.

3^o *στάδιο*: (Εφόσον χρειάζεται) θα κάνουμε *τοπικούς μετασχηματισμούς* στο δέντρο (τις λεγόμενες *περιστροφές*) έτσι ώστε οι διαφορές να ανήκουν και πάλι στο σύνολο $\{-1,0,1\}$.

Ο χρόνος που χρειαζόμαστε στο πρώτο στάδιο είναι $O(h)$, όπου h το ύψος του δέντρου (χρησιμοποιούμε τον Insert στη Σελίδα 94), που από την Πρόταση 4.5.3 είναι $O(\log n)$. Στο δεύτερο στάδιο θα χρειαστεί να επανυπολογίσουμε τους δείκτες αριστερού και δεξιού υποδέντρου σε το πολύ $O(\log n)$ κορυφές (ξεκινάμε φυσικά από την κορυφή που μόλις εισήγαμε), και ο χρόνος που χρειαζόμαστε για τον επανυπολογισμό είναι σταθερός [1]. Η διαδικασία στην οποία θα καταναλώσουμε πολύ χρόνο είναι η εύρεση του γονέα της κάθε κορυφής: Με τον αλγόριθμος FindParent στη Σελίδα 97 θα χρειαστούμε χρόνο $O(\log n)$ για την κάθε αναζήτηση του γονέα, οπότε συνολικά για το δεύτερο στάδιο θα χρειαστεί χρόνος $O(\log^2 n)$ [2]. Θα πρέπει λοιπόν να μπορούμε να βρίσκουμε τον γονέα της κορυφής σε σταθερό χρόνο. Για να το επιτύχουμε αυτό θα προσθέσουμε ένα ακόμα πεδίο στον κόμβο των δυαδικών δέντρων αναζήτησης. Θα κρατάμε για κάθε κορυφή έναν δείκτη που θα δείχνει προς τον γονέα της (δες Σχήμα 4.5.5). Ο δείκτη γονέα της ρίζας προφανώς θα έχει τιμή NIL. Θα γράφουμε:

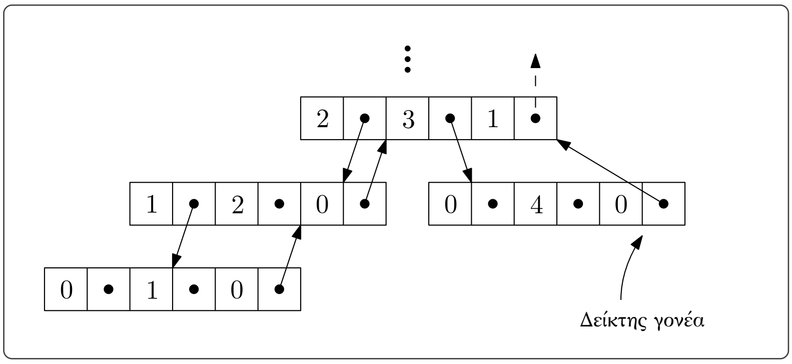
- $x.parent$ για να αναφερθούμε στον δείκτη γονέα του κόμβου x.

Έτσι και για το δεύτερο στάδιο θα χρειαζόμαστε χρόνο $O(\log n)$.

Στο τρίτο στάδιο, όπως θα δούμε, παρόλο που μπορούν να υπάρχουν πολλές κορυφές με διαφορά υψών -2 ή 2, αρκεί να κάνουμε μόνο μία περιστροφή. Η περιστροφή θα γίνει στην τελευταία κορυφή από αυτές στο μονοπάτι από τη ρίζα προς την κορυφή που εισήγαμε,

[1] Η αλήθεια είναι ότι ενδέχεται να μην χρειαστεί να κάνουμε όλους τους επαναυπολογισμούς, καθώς θα σταματήσουμε όταν συναντήσουμε κορυφή με διαφορά υψών -2 ή 2 και θα περάσουμε στο τρίτο στάδιο.

[2] Θα μπορούσαμε βέβαια να καλέσουμε μόνο μία φορά τον FindParent, για την καινούργια κορυφή που εισήγαμε, και να κρατήσουμε δείκτες για όλες τις κορυφές που επισκεφθήκαμε.



Σχήμα 4.5.5: *Οι κόμβοι ενός δέντρου AVL μετά από την προσθήκη και του δείκτη γονέα.*

έστω την κορυφή z. Παρατηρήστε ότι ο λόγος που η διαφορά υψών είναι -2 ή 2 είναι ότι ένα από τα υποδέντρα της z είναι «πιο ψηλό» από το άλλο (έχει ύψος κατά δύο μονάδες μεγαλύτερο). Με την περιστροφή που θα κάνουμε θα «κοντύνουμε» το ψηλό υποδέντρο κατά μία μονάδα και θα «ψηλώσουμε» το κοντό υποδέντρο κατά μία μονάδα. Όπως γίνεται εύκολα αντιληπτό η αλλαγή αυτή θα διορθώσει και τις διαφορές υψών για τις υπόλοιπες κορυφές που έχουν πρόβλημα [1].

Παρατηρήστε ότι για το τρίτο στάδιο χρειαζόμαστε σταθερό χρόνο, καθώς οι περιστροφές είναι τοπικοί μετασχηματισμοί (θα χρειαστεί να αλλάξουμε σταθερό πλήθος δεικτών αριστερού και δεξιού παιδιού) και εμείς θα χρειαστεί να κάνουμε μία ή δύο περιστροφές. Στο στάδιο αυτό μπορούμε να περάσουμε αμέσως μόλις βρούμε στο δεύτερο στάδιο κορυφή με διαφορά υψών -2 ή 2 (οπότε στην ουσία τα δύο στάδια συνδυάζονται).

Τελικά ο συνολικός χρόνος που χρειάζεται η εισαγωγή στοιχείου σε δέντρο AVL είναι $O(\log n)$.

Ας δούμε τις τέσσερις περιπτώσεις περιστροφών που θα χρειαστεί να κάνουμε. Οι λεπτομέρειες καθώς και ο αλγόριθμος εισαγωγής αφήνονται σαν άσκηση. Έστω z ή «κρίσιμη» κορυφή, y το παιδί της με το μεγαλύτερο ύψος υποδέντρου και x το παιδί της y με το μεγαλύτερο ύψος υποδέντρου [2]. Υπάρχουν τέσσερις τρόποι να συνδέονται οι κορυφές z, y και x, οι οποίο φαίνονται στο Σχήμα 4.5.6.

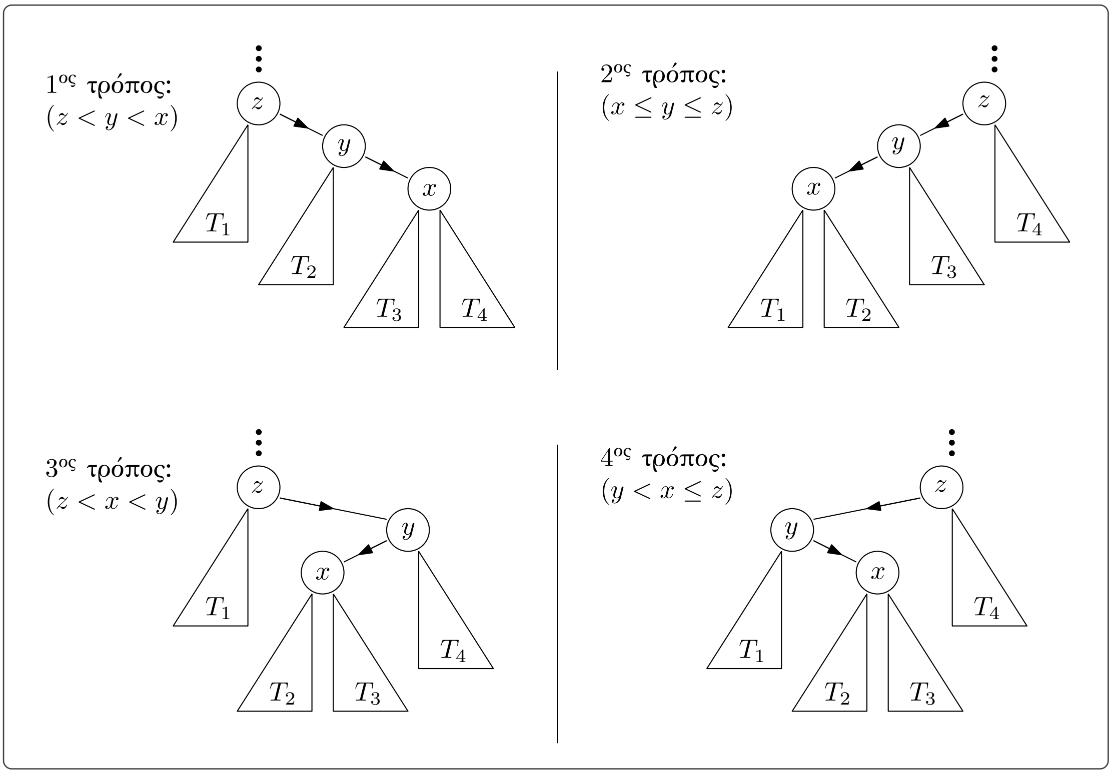
Για κάθε έναν από αυτούς τους τρόπους μπορούμε να ισορροπήσουμε το δέντρο κάνοντας περιστροφές. Στο Σχήμα 4.5.7 φαίνονται οι περιστροφές που πρέπει να γίνουν για τις δύο πρώτες περιπτώσεις.

Στις άλλες δύο περιπτώσεις θα χρειαστεί να κάνουμε δύο περιστροφές (Σχήμα 4.5.8), καθώς αν κάναμε μόνο μία περιστροφή, ναι μεν μπορεί να μειώναμε τη διαφορά υψών της z, αλλά μπορεί παράλληλα να αυξάναμε τη διαφορά υψών της x.

Θα χρειαστεί να δουλέψουμε με αντίστοιχο τρόπο και κατά τη διαγραφή μίας κορυφής από το δέντρο μας:

[1] Ο λόγος είναι ότι θα έχουμε διορθώσει το προβληματικό υποδέντρο μέσω της περιστροφής.

[2] Αφού η z έχει διαφορά υψών -2 ή 2 θα πρέπει το μονοπάτι από την z προς την κορυφή που εισήγαμε, έστω w, να περιέχει τουλάχιστον 2 κορυφές. Μπορεί φυσικά οι x και w να ταυτίζονται.



Σχήμα 4.5.6: *Οι τέσσερις τρόποι σύνδεσης των κορυφών* z, y *και* x*.*

1ο *στάδιο*: Διαγράφουμε την κορυφή (δες Σελίδα 95).

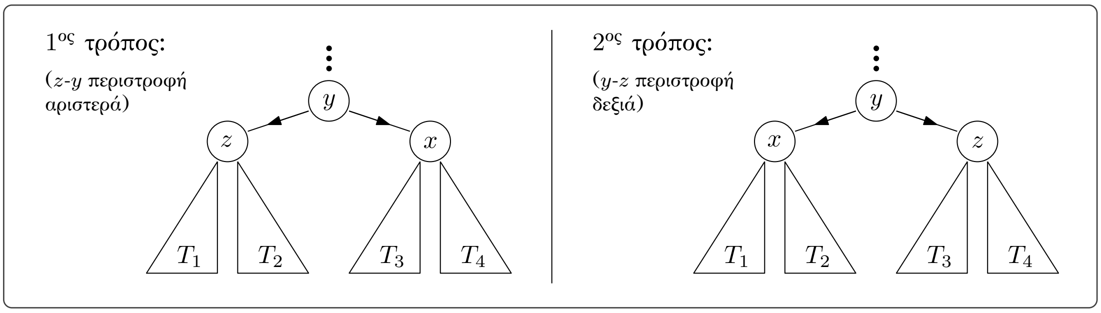
2ο *στάδιο*: Υπολογίζουμε τα ύψη (και τις διαφορές υψών) για τους προγόνους του γονέα της κορυφής που μόλις διαγράψαμε.

3ο *στάδιο*: (Εφόσον χρειάζεται) κάνουμε τις απαραίτητες περιστροφές.

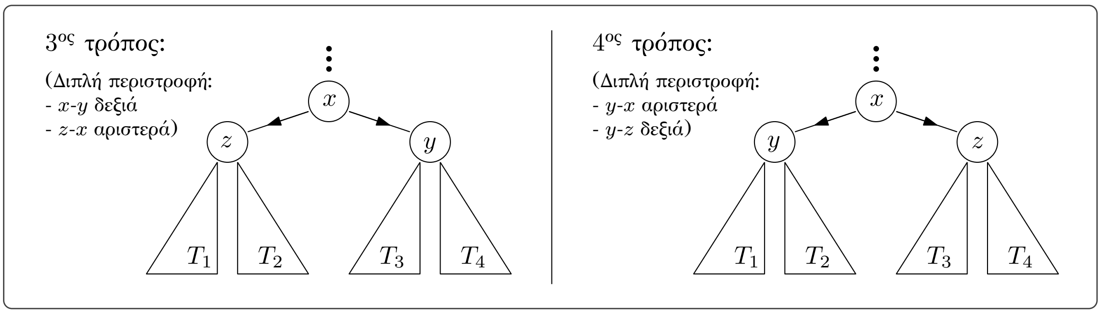
Κατά την εισαγωγή στοιχείου είδαμε ότι για να επαναφέρουμε την ισορροπία στη διαφορά υψών των κορυφών του δέντρου αρκούσε να κάνουμε (το πολύ δύο) περιστροφές για την πρώτη κορυφή που η διαφορά κατά απόλυτη τιμή έγινε 2 (πρώτη στο μονοπάτι από την κορυφή που εισήγαμε προς τη ρίζα). Αυτό συνέβαινε γιατί αν εξισορροπήσουμε τα υποδέντρα αυτής της κορυφής στην ουσία επαναφέρουμε το δέντρο στην προηγούμενη κατάστασή του (που φυσικά ικανοποιούσε την ιδιότητα των δέντρων AVL).

Στη διαγραφή δυστυχώς αυτό δεν ισχύει, καθώς η αφαίρεση της κορυφής από το δέντρο προκαλεί μεγαλύτερη «διαταραχή» της ισορροπίας. Αυτό σημαίνει ότι μπορεί να χρειαστεί να κάνουμε περιστροφές για παραπάνω από μία κορυφές. Ας δούμε ένα παράδειγμα.

Παράδειγμα 4.5.4. Υποθέστε ότι στο δέντρο AVL του Σχήματος 4.5.1 θέλουμε να διαγράψουμε την κορυφή που περιέχει το στοιχείο 4. Η διαγραφή αυτή θα προκαλέσει αλλαγή στη διαφορά υψών του γονέα της (από 1 θα γίνει 2). Κάνοντας περιστροφή προς τα δεξιά οδηγούμαστε στο δέντρο του Σχήματος 4.5.9, στο οποίο, ναι μεν έχουμε ισορροπήσει



Σχήμα 4.5.7: *Οι περιστροφές για τις δύο πρώτες περιπτώσεις.*



Σχήμα 4.5.8: *Οι περιστροφές για τις δύο τελευταίες περιπτώσεις.*

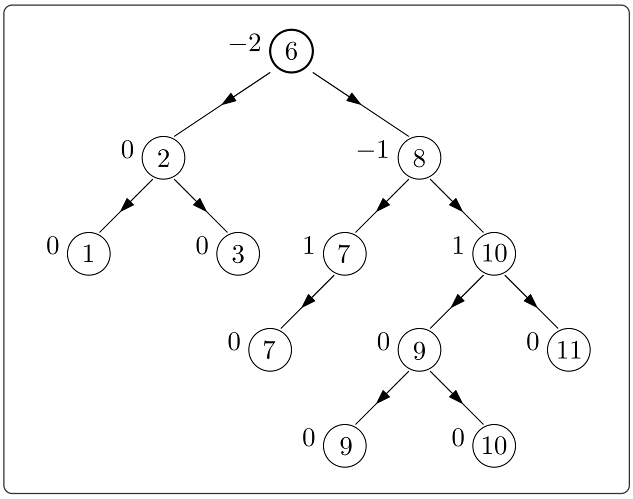
τη διαφορά υψών τοπικά, έχουμε προκαλέσει όμως ανισορροπία για τη διαφορά της ρίζας. Επομένως θα χρειαστεί να κάνουμε μία ακόμα περιστροφή (αριστερή περιστροφή, δες Σχήμα 4.5.10).

Παρατηρήστε ότι οι μόνες κορυφές που ενδέχεται να χρειαστεί να δεχθούν περιστροφή για να ισορροπήσει ξανά το δέντρο είναι οι πρόγονοι του γονέα της κορυφής που διαγράψαμε. Επομένως στη χειρότερη περίπτωση το πλήθος τους είναι όσο και το ύψος του δέντρου. Καθώς το δέντρο είναι AVL, αυτό σημαίνει ότι θα χρειαστούν το πολύ $O(\log n)$ περιστροφές.

Συνολικά ο χρόνος της διαγραφής είναι $O(\log n)$:

- Το πρώτο στάδιο χρειάζεται χρόνο $O(\log n)$ (γραμμικό ως προς το ύψος του δέντρο),
- το δεύτερο είναι αντίστοιχο με την εισαγωγή κορυφής οπότε χρειάζεται χρόνο $O(\log n)$ και τέλος
- στο τρίτο θα χρειαστούν $O(\log n)$ περιστροφές σταθερού χρόνου.

Εν κατακλείδι μπορούμε να έχουμε τις καλές ιδιότητες των δυαδικών δέντρων αναζήτησης μαζί με αποδοτικούς αλγορίθμους για τις βασικές λειτουργίες τους.



Σχήμα 4.5.9: *Το δέντρο του Σχήματος 4.5.1 μετά τη διαγραφή της κορυφής που περιείχε το στοιχείο* 4 *και την περιστροφή προς τα δεξιά.*

4.6 Κοκκινόμαυρα δέντρα

Ένας άλλος τύπος *ισοζυγισμένων* δυαδικών δέντρων αναζήτησης είναι τα λεγόμενα κοκκινόμαυρα δέντρα. Σε αυτά τα δέντρα οι κορυφές είναι χρωματισμένες κόκκινες ή μαύρες, έτσι ώστε να ισχύουν τα ακόλουθα:

1. Η ρίζα έχει μαύρο χρώμα.
2. Το χρώμα των παιδιών κάθε κόκκινης κορυφής έχουν μαύρο χρώμα (για τα παιδιά των μαύρων κορυφών δεν υπάρχει περιορισμός).
3. Για κάθε κορυφή u όλα τα μονοπάτια από τη u προς τα φύλλα του υποδέντρου με ρίζα τη u περιέχουν το ίδιο πλήθος μαύρων κορυφών (δες Σχήμα 4.6.1).

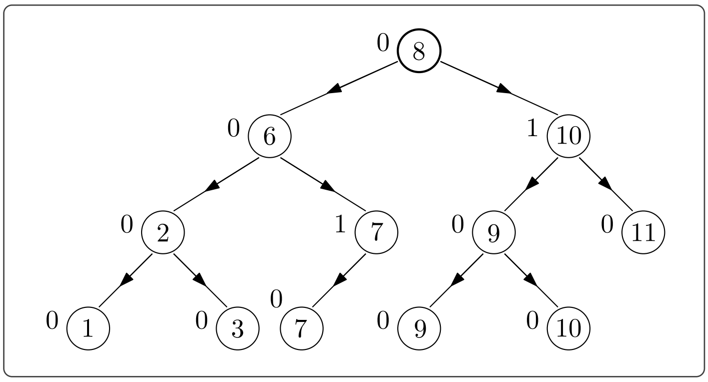
(Μην αμελούμε το γεγονός ότι το δέντρο είναι δυαδικό δέντρο αναζήτησης, συνεπώς πληροί και τις προϋποθέσεις του Ορισμού 4.3.1.)

Για την αναπαράστασή τους χρησιμοποιούμε έναν κόμβο δυαδικού δέντρου με επιπλέον πεδία για το χρώμα και για τον γονέα της κορυφής. Επίσης (για «τεχνικούς» λόγους) θεωρούμε ότι οι δείκτες με τιμή NIL δείχνουν προς κάποιο πλασματικό «εξωτερικό» κόμβο-φύλλο που έχει μαύρο χρώμα.

Ο λόγος που τα δέντρα αυτά έχουν ισοζυγισμένο ύψος φαίνεται στην ακόλουθη πρόταση.

Πρόταση 4.6.1. Έστω $T = (V, E, r)$ κοκκινόμαυρο δέντρο με $|V| = n$. Το ύψος του T είναι το $O(\log(n))$.

Απόδειξη. Παρατηρήστε ότι (λόγω του ότι τα παιδιά των κόκκινων κορυφών είναι απαραίτητα μαύρες κορυφές) σε κάθε μονοπάτι του δέντρου, μεταξύ δύο μαύρων κορυφών μπορεί



Σχήμα 4.5.10: *Το δέντρο του Σχήματος 4.5.9 μετά την περιστροφή προς τα αριστερά. Παρατηρήστε ότι πλέον έχουμε ένα δέντρο AVL.*

να παρεμβάλετε το πολύ μία κόκκινη κορυφή. Συνεπώς το μέγιστο μήκος μονοπατιού, έστω h, από τη ρίζα προς κάποιο φύλλο του δέντρου (το ύψος του δέντρου δηλαδή) θα είναι το πολύ δύο φορές το ελάχιστο μήκος μονοπατιού, έστω h', από τη ρίζα προς κάποιο φύλλο, έστω h' [1].

Αφού ένα πλήρες δυαδικό δέντρο ύψους h έχει ακριβώς $2^{h+1}-1$ κορυφές, έπεται ότι:

$$2^{h'+1}-1 \leq n \leq 2^{h+1}-1$$

(η πρώτη ανισότητα ισχύει γιατί αν το δέντρο περιείχε μόνο μαύρες κορυφές θα ήταν πλήρες δυαδικό δέντρο με ύψος h') ή

$$2^{h'+1} \leq n+1 \leq 2^{h+1}$$

Λογαριθμούμε τα μέρη αυτών των ανισοτήτων και έχουμε ότι:

$$h'+1 \leq \log(n+1) \leq h+1$$

ή

$$h' \leq \log(n+1)-1 \leq h$$

Όμως είδαμε ότι $h \leq 2h'$, δηλαδή $\frac{h}{2} \leq h'$, συνεπώς:

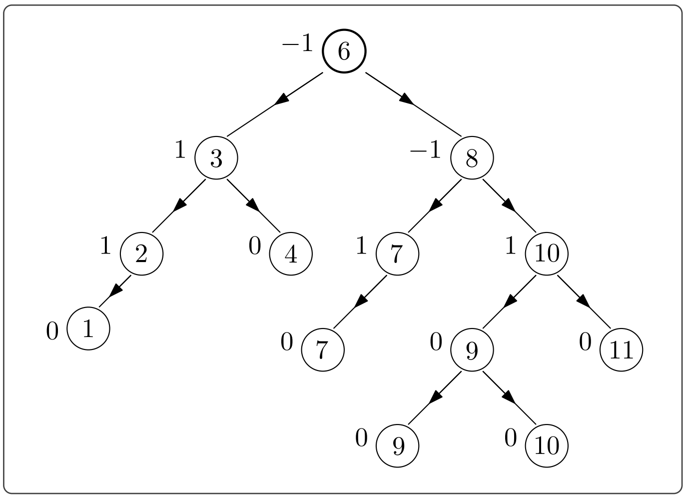
$$\frac{h}{2} \leq \log(n+1)-1$$

Άρα $h \leq 2(\log(n+1)-1)$. □

Θα εισάγουμε τις καινούργιες κορυφές κατά τα γνωστά [2], δίνοντάς τους κόκκινο χρώμα (έτσι ώστε να μη αλλοιωθεί το πλήθος των μαύρων κορυφών στα μονοπάτια του δέντρου

[1] Τα δύο αυτά μονοπάτια έχουν το ίδιο πλήθος μαύρων κορυφών και το μακρύ μπορεί να έχει επιπλέον μία κόκκινη κορυφή για κάθε μαύρη

[2] Ως κατάλληλα φύλλα του δέντρου δηλαδή.



Σχήμα 4.6.1: *Παράδειγμα κοκκινόμαυρου δέντρου.*

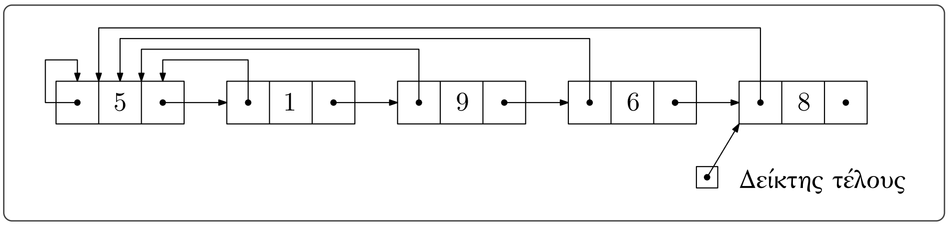
από τη ρίζα προς τα φύλλα). Με αυτόν τον τρόπο όμως μπορεί να υπάρξουν δύο «συνεχόμενες» κόκκινες κορυφές. Το πρόβλημα αυτό λύνεται κάνοντας περιστροφές ή/και αλλαγές χρωμάτων. Κατά τη διαγραφή τα προβλήματα είναι αντίστοιχα (διαδοχικές κόκκινες κορυφές ή ασυμμετρία στο πλήθος μαύρων κορυφών στα μονοπάτια), πάλι όμως μπορούμε να διορθώσουμε το δέντρο με περιστροφές και αλλαγές χρωμάτων [1].

Ο χρόνος που χρειαζόμαστε για τις παραπάνω πράξεις είναι γραμμικός ως προς το ύψος του δέντρου (στη χειρότερη περίπτωση θα χρειαστεί να κάνουμε τόσους αναχρωματισμούς-περιστροφές όσο και το ύψος του δέντρου). Συνεπώς χρειάζεται χρόνος $O(\log n)$ σε δέντρο με n κορυφές.

4.7 Ξένα σύνολα

Μία δομή δεδομένων που βρίσκει πολλές εφαρμογές είναι η δομή των *ξένων συνόλων*. Σε αυτήν τη δομή διατηρούμε μία συλλογή $\mathcal{C} = \{S_1, \ldots, S_k\}$ από *ξένα* μεταξύ τους σύνολα, που απαρτίζονται από στοιχεία ενός συνόλου U (το λεγόμενο και *σύμπαν*). Προκειμένου να αναφερθούμε στο σύνολο $S \in \mathcal{C}$ της συλλογής θα χρησιμοποιήσουμε ένα στοιχείο $s \in S$ ως *αντιπρόσωπο*. Το s δεν είναι απαραίτητο να φέρει κάποια συγκεκριμένη ιδιότητα για να «αντιπροσωπεύει» το εκάστοτε σύνολο (θα μπορούσε φυσικά να ισχύει και κάτι τέτοιο, αλλά δεν χρειάζεται στις περισσότερες των εφαρμογών), το μόνο που μας απασχολεί είναι η επιλογή του να είναι συνεπής. Θέλουμε εν ολίγοις, αν θέσουμε το ερώτημα «Ποιος είναι ο αντιπρόσωπος του S;» σε δύο διαφορετικές χρονικές στιγμές, μεταξύ των οποίων δεν έχει προκληθεί κάποια αλλαγή στο S, η απάντηση που θα λάβουμε να είναι και στις δύο φορές το ίδιο στοιχείο.

[1] Για περισσότερες πληροφορίες θα χρειαστεί να ανατρέξετε στην βιβλιογραφία



Σχήμα 4.7.1: *Αναπαράσταση του συνόλου* $\{5, 1, 9, 6, 8\}$.

Ας σκεφτούμε λίγο τι πράξεις θα θέλαμε να υποστηρίζει αυτή η δομή. Σίγουρα θα χρειαστεί να μπορούμε να βρούμε τον αντιπρόσωπο του συνόλου που ανήκει ένα στοιχείο [1]. Εύλογο είναι λοιπόν όλα τα στοιχεία του κάθε συνόλου να συνδέονται με κάποιον τρόπο με τον αντιπρόσωπό τους. Επίσης θα χρειαστούμε και κάποια πράξη αρχικοποίησης της δομής. Δεν θα μας απασχολήσει το πως ο Η/Υ θα «συντηρεί» τη συλλογή $\mathcal{C}$, θα εξετάσουμε μόνο τον τρόπο που δημιουργούμε ένα καινούριο σύνολο (αδιαφορώντας για το πως αυτό μετά τη δημιουργία του θα προσαρτηθεί στη $\mathcal{C}$). Τέλος θα χρειαστούμε και τις βασικές συνολοθεωρητικές πράξεις. Η τομή φυσικά δεν θα έχει νόημα καθώς τα σύνολά μας είναι ξένα μεταξύ τους, οπότε μένει να υλοποιήσουμε την πράξη της ένωσης δύο συνόλων.

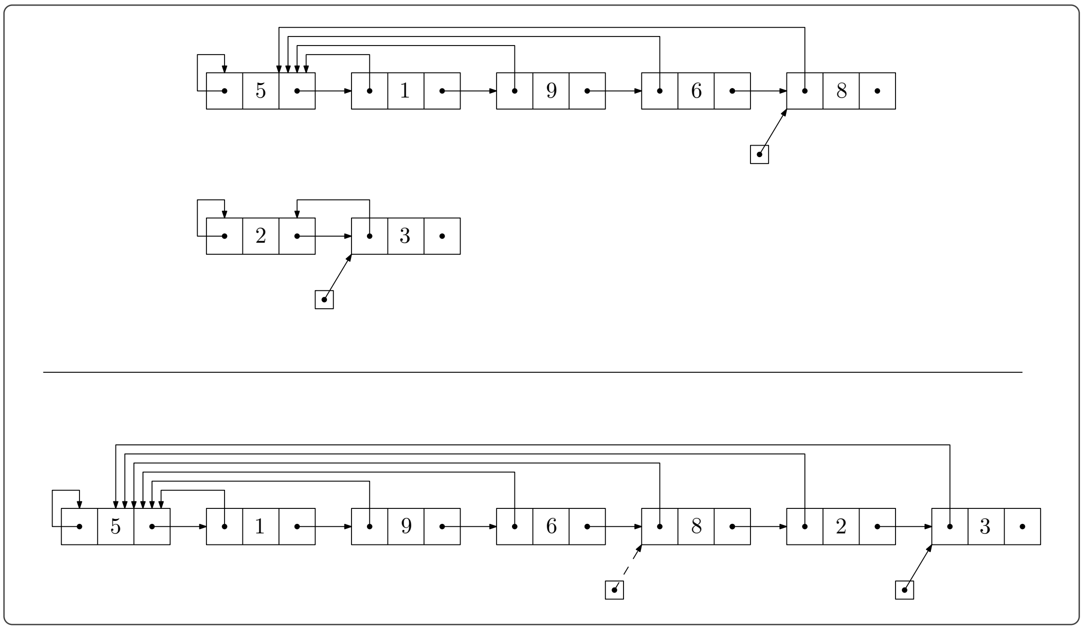
Προκειμένου να κρατήσουμε το σύνολο των υποστηριζόμενων λειτουργιών στο ελάχιστο δυνατό, η πρόσθεση στοιχείο σε ένα σύνολο θα γίνεται μέσω των πράξεων της δημιουργίας συνόλου (μονοσύνολου για την ακρίβεια) και της ένωσης συνόλου. Τέλος, δεν θα ασχοληθούμε με την πράξη της αφαίρεσης στοιχείου από το σύνολο.

Ο λόγος που συζητάμε τα ξένα σύνολα στο κεφάλαιο που μελετάμε τα δέντρα υποδηλώνει φυσικά το γεγονός ότι μπορούμε να τα αναπαραστήσουμε χρησιμοποιώντας και δέντρα (που δεν θα είναι κατ' ανάγκη δυαδικά). Προτού όμως περάσουμε σε αυτήν την αναπαράσταση (η οποία όπως θα δούμε στη συνέχεια –εφόσον γίνει «σωστά»– είναι πολύ αποδοτική) θα δούμε μία ποιο στοιχειώδη αναπαράσταση μέσω συνδεδεμένων λιστών. Τονίζουμε ότι οι αναπαραστάσεις θα αφορούν τα σύνολα αυτά καθεαυτό και όχι τη συλλογή που τα περιέχει.

4.7.1 Αναπαράσταση ξένων συνόλων με συνδεδεμένες λίστες

Θα χρησιμοποιήσουμε διπλά συνδεδεμένες λίστες όπου το πρώτο στοιχείο της λίστας θα αποτελεί τον αντιπρόσωπο του συνόλου, τα υπόλοιπα στοιχεία του συνόλου θα συνδέονται μεταξύ τους μέσω του δείκτη επόμενου, ενώ ο δείκτης προηγούμενου θα δείχνει προς τον αντιπρόσωπο του συνόλου. Για αυτόν τον λόγο θα αποκαλούμε εφεξής τον δείκτη προηγούμενου *δείκτη αντιπροσώπου*. Ο δείκτης αντιπροσώπου του αντιπροσώπου του συνόλου θα δείχνει προς τον εαυτό του. Θα χρησιμοποιήσουμε έναν ακόμα δείκτη που θα δείχνει στο τελευταίο στοιχείο της λίστας, τον *δείκτη τέλους*, ο οποίος θα μας φανεί χρήσιμος κατά την ένωση δύο συνόλων. Στο Σχήμα 4.7.1 μπορείτε να δείτε την αναπαράσταση του συνόλου $\{5, 1, 9, 6, 8\}$.

[1] Στην ουσία, δοσμένου ενός στοιχείου $u \in U$ θέλουμε να μπορούμε να βρούμε σε ποιο από τα $S_1, \ldots, S_k$ ανήκει.



Σχήμα 4.7.2: *Η ένωση του* $\{5, 1, 9, 6, 8\}$ *με το* $\{2, 3\}$.

Ας δούμε (χωρίς να παραθέσουμε τους αλγόριθμους) πως μπορούμε να υλοποιήσουμε τις τρεις λειτουργίες που υποστηρίζουν τα ξένα σύνολα [1].

1. *Δημιουργία συνόλου:* Για να δημιουργήσουμε ένα σύνολο που περιέχει το στοιχείο x αρκεί να δημιουργήσουμε μία λίστα με έναν κόμβο που θα περιέχει το x. Ο δείκτης επομένου του κόμβου θα έχει τιμή NIL, ενώ οι δείκτες αντιπροσώπου και τέλους θα δείχνουν προς αυτόν τον κόμβο.

2. *Εύρεση αντιπροσώπου:* Αν μας δώσουν ένα στοιχείο (τον κόμβο που το περιέχει για την ακρίβεια) βρίσκουμε τον αντιπρόσωπο του συνόλου που ανήκει απλά ακολουθώντας τον δείκτη αντιπροσώπου του. Με αυτόν τον τρόπο μπορούμε άμεσα να ελέγξουμε αν δύο στοιχεία ανήκουν στο ίδιο σύνολο, βρίσκοντας τους αντιπροσώπους τους και συγκρίνοντάς τους. Υποθέστε ότι εκτός από τον δείκτη τέλους είχαμε κρατήσει και τον δείκτη κεφαλής της λίστας. Παρατηρήστε ότι δεν θα μπορούσαμε να τον χρησιμοποιήσουμε για να βρούμε το σύνολο που ανήκει το στοιχείο καθώς δεν ξέρουμε a priori τη λίστα που το περιέχει. Για αυτόν τον λόγο δεν υπάρχει η ανάγκη του δείκτη κεφαλής.

3. *Ένωση συνόλων:* Υποθέστε ότι μας δίνονται δύο στοιχεία x και y, βρίσκουμε τους αντιπροσώπους των συνόλων που ανήκουν, και καθώς ανήκουν σε δύο διαφορετικά

[1] Η αυστηρή υλοποίηση των λειτουργιών αυτών αφήνεται ως άσκηση.

σύνολα, έστω τα S_x και S_y αντίστοιχα, αποφασίζουμε να τα ενώσουμε [1].

Ο πιο απλός τρόπος να υλοποιήσουμε την ένωση είναι να συνδέσουμε τις δύο λίστες σε σειρά. Σε αυτό το σημείο θα χρειαστούμε τον δείκτη τέλους του ενός συνόλου, ας υποθέσουμε του S_x. Αφού βρούμε τον τελευταίο κόμβο του S_x θα αλλάξουμε τον δείκτη επόμενού του έτσι ώστε να δείχνει τον πρώτο κόμβο του S_y, τον αντιπρόσωπο του S_y δηλαδή (θυμηθείτε ότι έχουμε ήδη βρει τους δύο αντιπροσώπους). Με αυτόν τον τρόπο, πέρα από το γεγονός ότι έχουμε συνδέσει τις δύο λίστες, έχουμε προαποφασίσει στην ουσία ότι ο αντιπρόσωπος του $S_x \cup S_y$ θα είναι ο αντιπρόσωπος του S_x. Το μόνο που απομένει να κάνουμε είναι να αλλάξουμε τον δείκτη αντιπροσώπου σε όλους τους κόμβους του S_y ώστε να δείχνουν τον νέο αντιπρόσωπό τους, τον αντιπρόσωπο του S_x [2]. Δυστυχώς αυτό το κομμάτι της ένωσης είναι ιδιαίτερα χρονοβόρο, με χρόνο που μπορεί να γίνει γραμμικός ως προς το $|U|$ [3]. Στο Σχήμα 4.7.2 μπορείτε να δείτε την ένωση του $\{5, 1, 9, 6, 8\}$ με το $\{2, 3\}$.

Για να μελετήσουμε καλύτερα τον χρόνο που χρειαζόμαστε για τις τρεις λειτουργίες των συνόλων θα χρειαστεί να κάνουμε *αντισταθμιστική ανάλυση*. Θα υπολογίσουμε πόσος χρόνος χρειάζεται για μια ακολουθία εκτέλεσης m λειτουργιών η οποίο ξεκινάει με n δημιουργίες συνόλων [4] (προφανώς $m \geq n$). Καθώς οι ενώσεις συνόλων μειώνουν το πλήθος συνόλων της συλλογής (αυξάνοντας το πλήθος στοιχειών των συνόλων που δημιουργούν) η ακολουθία μπορεί να περιέχει το πολύ $n - 1$ ενώσεις. Ο χρόνος που απαιτείται για μία δημιουργία συνόλου ή μία εύρεση αντιπροσώπου είναι σταθερός. Ας υποθέσουμε ότι είναι c_1 και c_2 αντίστοιχα. Η ένωση συνόλων χρειάζεται (στην περίπτωση όπου κρατάμε τον αντιπρόσωπο του συνόλου με τα λιγότερα στοιχεία) χρόνο $c_3 \cdot k$, όπου k το πλήθος στοιχείων του μεγαλύτερου από τα δύο σύνολα που ενώνουμε. Επομένως η χειρότερη δυνατή ακολουθία λειτουργιών (ως προς τον χρόνο που απαιτείται για την ολοκλήρωσής της) περιέχει φυσικά n δημιουργίες συνόλων, έστω για τα στοιχεία $x_1, \ldots, x_n$, $n-1$ ενώσεις (όπου πάντα κρατάμε σαν αντιπρόσωπο αυτόν του συνόλου με τα λιγότερα στοιχεία, στον πίνακα που ακολουθεί φαίνεται η χειρότερη δυνατή ακολουθία ενώσεων) και $m - 2n + 1$ ευρέσεις αντιπροσώπου.

Ας δούμε τον χρόνο που θα χρειαστεί η ακολουθία αναλυτικά [5]:

[1] Φυσικό επόμενο αυτής της ένωσης είναι η αφαίρεση των S_x, S_y από τη $\mathcal{C}$ και η προσθήκη του $S_x \cup S_y$. Ο λόγος που πρέπει να γίνει αυτό είναι ότι η $\mathcal{C}$ περιέχει μόνο ξένα μεταξύ τους σύνολα. Η αφαίρεση των S_x, S_y από τη $\mathcal{C}$ δεν χρειάζεται να γίνεται ρητά. Παρατηρήστε ότι μετά την ένωση όποτε κάνουμε εύρεση αντιπροσώπου για κάποιο στοιχείο του S_x ή του S_y θα πάρουμε τον αντιπρόσωπο του $S_x \cup S_y$. Συνεπώς τα σύνολα S_x, S_y είναι σαν να μην «υφίστανται». Το παράδειγμα αυτό (ενδεχομένως) κάνει πιο εμφανή τον λόγο που δεν ενδιαφερόμαστε για το πως διατηρείται η συλλογή $\mathcal{C}$ στη μνήμη του Η/Υ (το μόνο που θα χρειαστεί να κρατήσουμε είναι μία λίστα με τους αντιπροσώπους των συνόλων).

[2] Εφόσον γίνει και αυτό μπορούμε να «ξεχάσουμε» τον δείκτη τέλους του S_x.

[3] Στη χειρότερη περίπτωση θα χρειαστεί να ενώσουμε ένα μονοσύνολο με όλο το υπόλοιπο U και λόγω αβλεψίας μας θα κρατήσουμε σαν αντιπρόσωπο το μοναδικό στοιχείο του μονοσύνολου.

[4] Μπορούμε να θεωρήσουμε ότι $|U| = n$, δηλαδή ότι το σύμπαν απαρτίζεται από τα στοιχεία αυτών των μονοσυνόλων.

[5] Υποθέτουμε ότι η ακολουθία ξεκινάει με τις n δημιουργίες, έπειτα έχουμε τις $n - 1$ ενώσεις και τέλος τις ευρέσεις αντιπροσώπου. Φυσικά οι ενώσεις και οι ευρέσεις δεν θα γίνουν με αυτήν τη σειρά (ούτως η άλλως η ένωση προϋποθέτει ότι πρώτα έγιναν δύο ευρέσεις αντιπροσώπου). Παρατηρήστε όμως ότι με όποια σειρά και να γίνουν οι ευρέσεις ο χρόνος που θα χρειαστούν θα είναι ο ίδιος.

	Πράξη	**Χρόνος**
1.	Δημιουργία $\{x_1\}$	c_1
2.	Δημιουργία $\{x_2\}$	c_1
$\vdots$	$\vdots$	$\vdots$
$n.$	Δημιουργία $\{x_n\}$	c_1
$n+1.$	$\{x_2\} \cup \{x_1\}$	$c_3 \cdot 1$
$n+2.$	$\{x_3\} \cup \{x_1, x_2\}$	$c_3 \cdot 2$
$\vdots$	$\vdots$	$\vdots$
$2n-1.$	$\{x_n\} \cup \{x_1, \ldots, x_{n-1}\}$	$c_3 \cdot (n-1)$
$2n.$	Εύρεση αντιπροσώπου	c_2
$\vdots$	$\vdots$	$\vdots$
$m.$	Εύρεση αντιπροσώπου	c_2

Επομένως χρειάζεται χρόνος:

$$\begin{aligned}
c_1 \cdot n + c_3 \cdot \sum_{i=1}^{n-1} i + c_2 \cdot (m - 2n + 1) &= c_1 \cdot n + c_3 \cdot \frac{n \cdot (n-1)}{2} + c_2 \cdot (m - 2n + 1) \\
&= O(n) + O(n^2) + O(m) \\
&= O(m + n^2)
\end{aligned}$$

Μπορούμε να βελτιώσουμε αυτόν τον χρόνο εφαρμόζοντας ένα απλό τέχνασμα. Αντί να επιλέγουμε τον αντιπρόσωπο της ένωσης δύο συνόλων στην τύχη θα διαλέγουμε πάντα τον αντιπροσώπου του συνόλου με τα περισσότερα στοιχεία. Κατά αυτόν τον τρόπο θα χρειαστεί να κάνουμε πολύ λιγότερες αλλαγές στους δείκτες αντιπροσώπων.

Για να το πετύχουμε όμως αυτό θα πρέπει να προσθέσουμε στα σύνολα έναν μετρητή ο οποίος θα καταγράφει το πλήθος στοιχειών που έχει το σύνολο. Έτσι στην αναπαράσταση του συνόλου εκτός από τη λίστα θα προσθέσουμε και μία μεταβλητή που θα παίρνει θετικές ακέραιες τιμές, τη λεγόμενη *τάξη* του συνόλου. Σε κάθε ένωση δύο συνόλων S_x, S_y θα βρίσκουμε αυτό με τη μεγαλύτερη τάξη και θα επιλέγουμε τον αντιπρόσωπό του σαν τον αντιπρόσωπο του $S_x \cup S_y$ (αν έχουν την ίδια τάξη τότε επιλέγουμε αυθαίρετα). Η τάξη της ένωσης θα είναι το άθροισμα των τάξεων των S_x, S_y.

Για να υλοποιήσουμε αυτό το εγχείρημα θα προσθέσουμε ακόμα ένα πεδίο στους κόμβους της λίστας, στο οποίο θα αποθηκεύουμε την τάξη του συνόλου. Μετά την ένωση θα ενημερώνουμε την τάξη του αντιπροσώπου του νέου συνόλου. Η τάξη για τα υπόλοιπα στοιχεία, θα έχει φυσικά κάποια τιμή, αλλά αυτή δεν θα αντιπροσωπεύει κάτι [1].

Πρόταση 4.7.1. Μια ακολουθία από m πράξεις, με $n \leq m$ δημιουργίες συνόλων, χρειάζεται συνολικό χρόνο $O(m + n \log n)$.

[1] Ο λόγος που το κάνουμε αυτό (αντί απλά να κρατήσουμε ένα νούμερο παράλληλα με τη λίστα που αναπαριστά το κάθε σύνολο) θα φανεί στην επόμενη παράγραφο.

Απόδειξη. Όπως πριν η ακολουθία ξεκινάει με n δημιουργίες συνόλου, έπειτα έχουμε $n-1$ ενώσεις, που τελικά θα μας οδηγήσουν σε ένα σύνολο με n στοιχεία, έστω το U. Διάσπαρτες ανάμεσά τους έχουμε $m-2n+1$ ευρέσεις αντιπροσώπου.

Για να μετρήσουμε τον συνολικό χρόνο της ακολουθίας το κρίσιμο σημείο είναι να μετρήσουμε το μέγιστο πλήθος αλλαγών στους δείκτες αντιπροσώπων για τα στοιχεία του U. Έστω $x \in U$. Την πρώτη φορά που άλλαξε ο δείκτης του x η ένωση δημιούργησε ένα σύνολο με τουλάχιστον δύο στοιχεία. Τη δεύτερη φορά ένα σύνολο με τουλάχιστον 4 στοιχεία [1], και γενικότερα μετά την i-οστή αλλαγή του δείκτη του x η ένωση θα έχει τάξη 2^i. Οπότε όταν θα πάρουμε το U θα έχουν γίνει το πολύ $\log n$ αλλαγές δείκτη στο x. Άρα ο συνολικός χρόνος για τις $n-1$ ενώσεις (μέχρι να πάρουμε το U) είναι $O(n \log n)$ (n στοιχεία, από $O(\log n)$ αλλαγές το καθένα). Για τις υπόλοιπες πράξεις της ακολουθίας (όπως είδαμε πριν) θα χρειαστούμε χρόνο $O(m+n)$. Συνεπώς ο συνολικός χρόνος της ακολουθίας είναι $O(m+n\log n)$. □

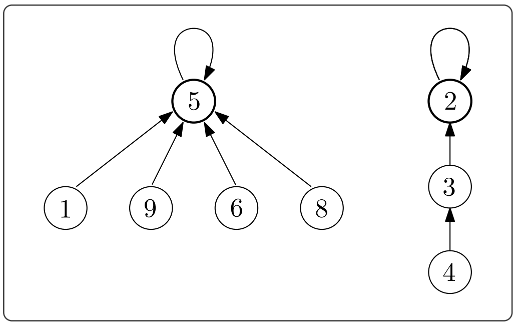
4.7.2 Αναπαράσταση ξένων συνόλων με δέντρα

Μπορούμε να αναπαραστήσουμε τα σύνολά μας χρησιμοποιώντας δέντρα (όχι απαραίτητα δυαδικά), η ρίζα των οποίων θα παίζει τον ρόλο του αντιπροσώπου του συνόλου. Δεν έχουμε συζητήσει πως μπορούμε να αναπαραστήσουμε γενικά δέντρα. Στο επόμενο κεφάλαιο θα δούμε πως αναπαριστούμε *γραφήματα*, καλύπτοντας έτσι και την αναπαράσταση των δέντρων. Στην περίπτωσή μας όμως για να τα αναπαραστήσουμε χρειάζεται μόνο να κρατήσουμε στους κόμβους τους έναν δείκτη που θα δείχνει τον γονέα τους (*δείκτης γονέα*). Ο δείκτης γονέα της ρίζας θα δείχνει στον εαυτό της (Σχήμα 4.7.3). Αυτός ο δείκτης αρκεί για να μπορούμε να βρίσκουμε τον αντιπρόσωπο του συνόλου και κατ' επέκταση σε ποιο σύνολο ανήκει ένα στοιχείο: Αν μας δοθεί μία κορυφή ακολουθούμε τους δείκτες γονέα μέχρι να φτάσουμε στη ρίζα του δέντρου.

Όπως αντιλαμβάνεστε αυτό απαιτεί χρόνο ανάλογο με το ύψος του δέντρου, πράγμα που καθιστά την πράξη εύρεσης αντιπροσώπου πιο αργή από την αναπαράσταση με λίστες. Ιδανικά, για να επιταχύνουμε όσον είναι δυνατό αυτήν την πράξη θα θέλαμε ο δείκτης γονέα να δείχνει απευθείας στη ρίζα του δέντρου (όπως παραδείγματος χάρη στο σύνολο $\{5,1,9,6,8\}$ Σχήμα 4.7.3). Αυτό όμως θα μας ανάγκαζε να αλλάζουμε τους δείκτες γονέα ενός ολόκληρου δέντρου κατά την ένωση δύο συνόλων (όπως κάναμε στην ουσία και με την αναπαράσταση με λίστες). Σκοπός μας είναι να βελτιώσουμε τον συνολικό χρόνο που θα χρειαστεί σε μια ακολουθία από m πράξεις, και όπως θα δούμε, μπορούμε να πετύχουμε περισσότερα κρατώντας την πράξη της εύρεσης αντιπροσώπου πιο «αργή». Στην ουσία αυτό που θα κάνουμε είναι να προσπαθούμε να μειώνουμε συνεχώς το ύψος του δέντρου (με κάθε πράξη εύρεσης αντιπροσώπου) έτσι, αντισταθμιστικά για ολόκληρη την ακολουθία των m πράξεων, οι πράξεις εύρεσης αντιπροσώπου και ένωσης συνόλων θα χρειάζονται χρόνο κοντά στο $O(1)$.

Τέλος, παρατηρήστε ότι πλέον δεν χρειαζόμαστε τον δείκτη ρίζας του δέντρου καθώς υπάρχει άλλος τρόπος να βρίσκουμε τη ρίζα του (ούτως ή άλλως δεν θα μπορούσαμε να τον

[1] Παρατηρήστε ότι αν η ένωση είχε 3 στοιχεία ο δείκτης του x δεν θα είχε αλλάξει εξαρχής καθώς το x θα ανήκε στο σύνολο με τη μεγαλύτερη τάξη.



Σχήμα 4.7.3: *Αναπαράσταση των συνόλων* $\{5, 1, 9, 6, 8\}$ *και* $\{2, 3, 4\}$. *Τα βέλη στις ακμές αντιπροσωπεύουν τον δείκτη γονέα κάθε κορυφής.*

χρησιμοποιήσουμε διότι δεν ξέρουμε a priori σε ποιο σύνολο-δέντρο ανήκει ένα στοιχείο).

Ας δούμε τις τρεις πράξεις των ξένων συνόλων σε αυτήν την αναπαράσταση:

1. *Δημιουργία συνόλου:* Για τη δημιουργία ενός συνόλου αρκεί να κατασκευάσουμε ένα δέντρο με μία μόνο κορυφή, φροντίζοντας ο δείκτης γονέα να δείχνει προς τον εαυτό της. Όπως κάναμε και στην αναπαράσταση με λίστες, για να δημιουργήσουμε σύνολα με περισσότερα στοιχεία θα αποταθούμε στην πράξη της ένωσης.

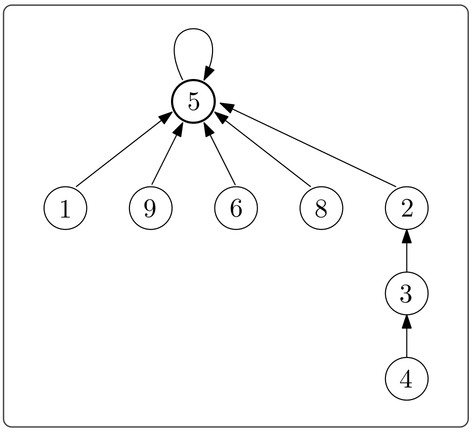
2. *Εύρεση αντιπροσώπου:* Δοσμένου ενός στοιχείου του συνόλου για να βρούμε τον αντιπρόσωπό του θα πρέπει να βρούμε τη ρίζα του δέντρου. Ο τρόπος που το κάνουμε αυτό αναφέρθηκε και πριν: Ακολουθούμε τον δείκτη γονέα κάθε κορυφής μέχρι να βρούμε την κορυφή με δείκτη γονέα που δείχνει στον εαυτό της.

3. *Ένωση συνόλων:* Για να ενώσουμε τα δέντρα που αντιπροσωπεύουν δύο σύνολα θα συνδέσουμε τις δύο ρίζες και θα επιλέξουμε ποια από τις δύο θα αποτελεί τη ρίζα στο καινούργιο δέντρο (Σχήμα 4.7.4).

   Για να το κάνουμε αυτό, δοσμένων δύο στοιχείων x και y των οποίων τα σύνολα, S_x και S_y αντίστοιχα θέλουμε να ενώσουμε, πρώτα βρίσκουμε τους αντιπροσώπους τους και έπειτα αλλάζουμε τον δείκτη γονέα του αντιπροσώπου του ενός, έστω του S_x, έτσι ώστε να δείχνει τον αντιπρόσωπο του S_y. Έτσι ο αντιπρόσωπος του $S_x \cup S_y$ θα είναι ο αντιπρόσωπος του S_y.

Παρατηρήστε ότι χρησιμοποιώντας δέντρα μπορούμε να δημιουργήσουμε σύνολα και να ενώσουμε σύνολα σε σταθερό χρόνο (εφόσον φυσικά έχουμε βρει τους αντιπροσώπους τους). Για την εύρεση του αντιπροσώπου θα χρειαστεί να «ανεβούμε» μέχρι τη ρίζα του δέντρου, που στη χειρότερη περίπτωση σημαίνει ότι ο χρόνος που θα δαπανήσουμε είναι ανάλογος με το ύψος του δέντρου, το οποίο με τη σειρά του μπορεί να είναι ανάλογο του πλήθους στοιχείων που περιέχει το σύνολο (και άρα ανάλογο με το $|U|$). Είναι λοιπόν επιτακτική ανάγκη για μία ακόμα φορά να κρατήσουμε το ύψος των δέντρων χαμηλό.

Το πρώτο βήμα προς αυτόν τον σκοπό είναι φυσικά να επιλέγουμε πάντα ως ρίζα του δέντρου της ένωσης τη ρίζα του δέντρου που έχει το μεγαλύτερο ύψος (αντίστοιχα με αυτό που



Σχήμα 4.7.4: *Η ένωση του* $\{5, 1, 9, 6, 8\}$ *με το* $\{2, 3, 4\}$ *από το Σχήμα 4.7.3. Σε αυτήν την περίπτωση το δέντρο που αναπαριστά το σύνολο* $\{2, 3, 4\}$ *προσαρτήθηκε σε αυτό που αναπαριστά το σύνολο* $\{5, 1, 9, 6, 8\}$. *Θα μπορούσε φυσικά να συμβεί και το αντίστροφο.*

κάναμε στην αναπαράσταση με τις λίστες). Θα θέλαμε λοιπόν να κρατάμε την πληροφορία του ύψους του δέντρου, π.χ. σε έναν ακέραιο δείκτη, έτσι ώστε να μπορούμε να αποφασίσουμε ποιο από τα δύο δέντρα θα προσαρτηθεί στο άλλο. Προκειμένου να ενσωματώσουμε αυτόν τον δείκτη στο δέντρο μας θα προσθέσουμε ένα ακόμα πεδίο στις κορυφές, που θα περιέχει έναν φυσικό αριθμό. Τον αριθμό αυτόν θα τον αποκαλούμε *τάξη* της κορυφής και θα ισούται με το ύψος του υποδέντρου που έχει ρίζα την εκάστοτε κορυφή (Σχήμα 4.7.5).

Παρατηρήστε ότι η τάξη μίας κορυφής αλλάζει μόνο κατά την ένωση δύο συνόλων, μόνο για μία από τις δύο ρίζες των δέντρων (τη ρίζα του δέντρου της ένωσης) και μόνο αν και τα δύο δέντρα έχουν το ίδιο ύψος (Σχήμα 4.7.5). Συνεπώς η διατήρηση αυτών των δεικτών δεν επιβαρύνει ουσιαστικά τον χρόνο των πράξεών μας.

Πρόταση 4.7.2. Έστω T το δέντρο που αναπαριστά ένα σύνολο με n στοιχεία. Η τάξη κάθε κορυφής του T είναι μικρότερη είτε ίση του $\log n$.

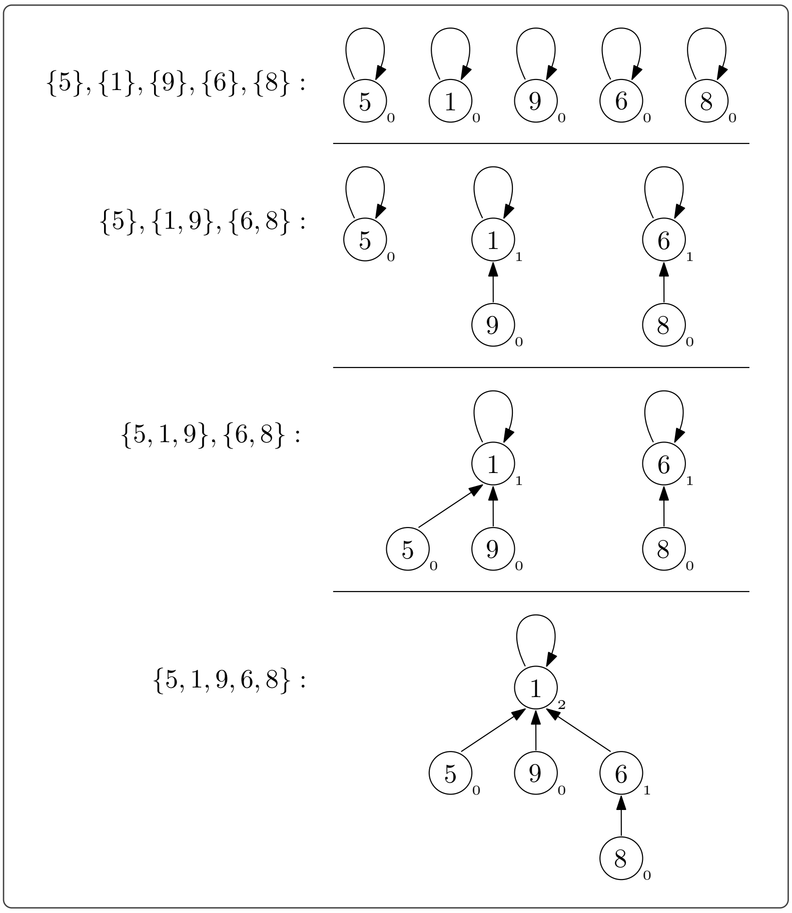
Απόδειξη. Αρκεί να δείξουμε ότι κάθε κορυφή τάξης k έχει τουλάχιστον 2^k κορυφές στο υποδέντρο της. Ας δούμε γιατί: Έστω r η τάξη της ρίζας του T (η μεγαλύτερη τάξη στο T). Σύμφωνα με τον ισχυρισμό μας το T θα έχει τουλάχιστον 2^r κορυφές, οπότε $n \geq 2^r$. Συνεπώς $\log n \geq r$.

Τον ισχυρισμό μας θα τον αποδείξουμε με επαγωγή στο k:

Επαγωγική Βάση: Για $k = 0$ το δέντρο περιέχει μόνο μία κορυφή και ο ισχυρισμός επαληθεύεται.

Επαγωγική Υπόθεση: Υποθέτουμε ότι κάθε κορυφή τάξης k έχει τουλάχιστον 2^k κορυφές στο υποδέντρο της.

Επαγωγικό Βήμα: Έστω u μία κορυφή τάξης $k+1$. (Όπως είδαμε και πριν) η τάξη της u έγινε $k+1$ έπειτα από την ένωση δύο συνόλων, ενός με αντιπρόσωπο τη u, έστω το S_u, και

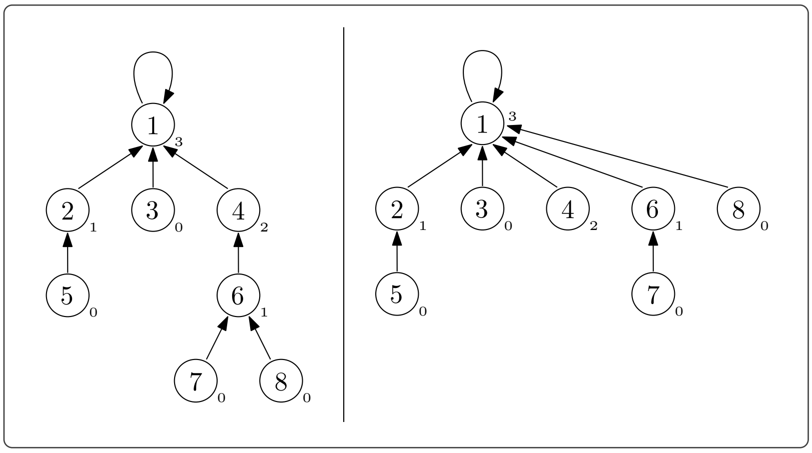


Σχήμα 4.7.5: *Αριστερά: Ένας τρόπος δημιουργίας του συνόλου* $\{5, 1, 9, 6, 8\}$*. Τα νούμερα δίπλα στις κορυφές αντιστοιχούν στην τάξη τους.*

ενός άλλου, έστω S_v, με αντιπρόσωπο την κορυφή v. Παρόλο που δεν ξέρουμε το πλήθος στοιχείων των S_u και S_v ξέρουμε ότι τόσο η u όσο και η v έχουν τάξη k στα δέντρα που αναπαριστούν τα δύο σύνολα (σε αντίθετη περίπτωση η τάξη της u θα παρέμενε k στο δέντρο που αναπαριστά το $S_u \cup S_v$). Από την επαγωγική υπόθεση τα δύο δέντρα έχουν τουλάχιστον 2^k κορυφές οπότε το καινούργιο δέντρο έχει τουλάχιστον 2^{k+1} κορυφές. $\square$

Από την παραπάνω πρόταση προκύπτει ότι μία ακολουθία από m πράξεις, με $n \leq m$ δημιουργίες συνόλων, απαιτεί χρόνο $O(m \log n)$ (δημιουργία και ένωση χρειάζονται σταθερό χρόνο και οι ευρέσεις αντιπροσώπου χρόνο $O(\log n)$).

Μπορούμε να βελτιώσουμε περαιτέρω τον χρόνο εφαρμόζοντας ακόμα ένα απλό τέχνασμα. Κάθε φορά που κάνουμε εύρεση αντιπροσώπου για ένα στοιχείο x θα αλλάζουμε τους δείκτες γονέα των κορυφών που επισκεπτόμαστε έτσι ώστε να δείχνουν στη ρίζα του δέντρου (Σχήμα 4.7.6). Με αυτόν τον τρόπο επιφέρουμε επιπλέον συμπίεση στο ύψος του δέντρου. Φυσικά για να γίνει αυτό εφικτό θα χρειαστεί να κάνουμε δύο διαπεράσεις από την κορυφή



Σχήμα 4.7.6: *Το αποτέλεσμα της πράξης της εύρεσης αντιπροσώπου για το στοιχείο 8. Παρατηρήστε ότι δεν αλλάζουμε τις τάξης των κορυφών, συνεπώς δεν μας δίνουν πλέον ακριβή τιμή για το ύψος του υποδέντρου. Αυτό δεν μας επηρεάζει ουσιαστικά (το ύψος του δέντρου θα είναι πάντα μικρότερο είτε ίσο από την τάξη της ρίζας).*

του στοιχείο x προς τη ρίζα του δέντρου, την πρώτη για να βρούμε τη ρίζα και τη δεύτερη για να αλλάξουμε τις τιμές στους δείκτες γονέα. Αυτό δεν θα επηρεάσει τον χρόνο αρνητικά (ούτε οι επιπλέον αλλαγές στους δείκτες) όπως προκύπτει από την ακόλουθη πρόταση.

Πρόταση 4.7.3. Μια ακολουθία από m πράξεις, με $n \leq m$ δημιουργίες συνόλων, χρειάζεται συνολικό χρόνο $O(m \cdot \alpha(n))$, όπου α μία συνάρτηση με πάρα πολύ αργό ρυθμό αύξησης.

Για να γίνει κατανοητό πόσο αργός είναι ο ρυθμός που αυξάνει η συνάρτηση α παραθέτουμε το γεγονός ότι η τιμή $\alpha(n)$ για τιμές του n μέχρι και 10^{80} είναι μικρότερη από 4. Επομένως για «εύλογες» τιμές του n (για τιμές δηλαδή που μπορούμε να συναντήσουμε στις εφαρμογές) μπορούμε να θεωρήσουμε ότι ο χρόνος που χρειαζόμαστε για μια ακολουθία m εφαρμογών δημιουργίας, ένωσης και εύρεσης αντιπροσώπου είναι γραμμικός ως προς το m.

Η απόδειξη της Πρότασης 4.7.3 περιέχει πολλές τεχνικές λεπτομέρειες και ξεφεύγει των σκοπών αυτών των σημειώσεων.

Στο κεφάλαιο που ακολουθεί θα μιλήσουμε για τον πλέον γενικό τρόπο να οργανώσουμε τα δεδομένα μας, επεκτείνοντας τις ιδέες που αναπτύξαμε σε αυτό το κεφάλαιο.

Ασκήσεις

4.1. Ένα δέντρο με n κορυφές έχει ακριβώς $n-1$ ακμές.

4.2. Δείξτε ότι ένα σχεδόν πλήρες δυαδικό δέντρο με n κορυφές, όπου στο κάτω κάτω επίπεδο οι κορυφές βρίσκονται όσο πιο αριστερά γίνεται, έχει τουλάχιστον $n/2$

φύλλα.

4.3. Έστω δυαδικό δέντρο $T = (V, E, r)$ όπου κάθε κορυφή έχει είτε 0 είτε 2 παιδιά. Έστω $f(T)$ το πλήθος των κορυφών του T με δύο παιδιά και $l(T)$ το πλήθος των φύλλων του. Δείξτε ότι $l(T) = f(T) + 1$.

4.4. Σχεδιάστε αλγόριθμους InOrderTraversal(T) και PostOrderTraversal(T) που υλοποιούν την ενδοδιατεταγμένη και την μεταδιατεταγμένη διαπέραση στο δέντρο με δείκτη ρίζας T.

4.5. Ξαναγράψτε τον αλγόριθμο PreOrderTraversal, που υλοποιεί την προδιατεταγμένη διαπέραση σε ένα δυαδικό δέντρο, έτσι ώστε να μην χρησιμοποιεί αναδρομή.

4.6. Σχεδιάστε αλγόριθμο Check(T) που δέχεται σαν είσοδο ένα δυαδικό δέντρο (με στοιχεία φυσικούς αριθμούς) και ελέγχει αν είναι δυαδικό δέντρο αναζήτησης, επιστρέφοντας κατάλληλα το μήνυμα "ΝΑΙ" ή "ΟΧΙ".

4.7. Διορθώστε των αλγόριθμο DeletionCase3 έτσι ώστε να δουλεύει ακόμα και όταν έχουμε κορυφές με το ίδιο περιεχόμενο.

4.8. Έστω δυαδικό δέντρο αναζήτησης T (με στοιχεία φυσικούς αριθμούς) και x φυσικός αριθμός. Δώστε αλγορίθμους:

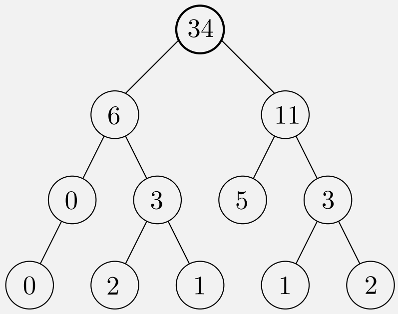
1. Maximum(T): Βρίσκει το μεγαλύτερο στοιχείο στο δέντρο T και επιστρέφει τον δείκτη που δείχνει στον κόμβο που το περιέχει.
2. Minimum(T): Βρίσκει το μικρότερο στοιχείο στο δέντρο T και επιστρέφει τον δείκτη που δείχνει στον κόμβο που το περιέχει.
3. Successor(T, x): Βρίσκει το στοιχείο με την αμέσως μεγαλύτερη τιμή από το x στο δέντρο T (αν υπάρχει) και επιστρέφει τον δείκτη που δείχνει στον κόμβο που το περιέχει.
4. Predecessor(T, x) Βρίσκει το στοιχείο με την αμέσως μικρότερη τιμή από το x στο δέντρο T (αν υπάρχει) και επιστρέφει τον δείκτη που δείχνει στον κόμβο που το περιέχει.

(Μπορείτε να υποθέσετε ότι στο δέντρο T δεν εμφανίζεται κάποιο στοιχείο παραπάνω από μία φορά.)

Τελευταία ενημέρωση 28/1/2025, στις 09:53.

4.9. Υποθέστε ότι κατά την αναπαράσταση των δυαδικών δέντρων αναζήτησης χρησιμοποιούμε κόμβους που έχουν έναν παραπάνω δείκτη ο οποίος «δείχνει» τον γονέα μιας κορυφής (και έχει τιμή NIL για την ρίζα του δέντρου). Έστω X μεταβλητή τύπου κόμβου (αυτής της μορφής). Θα αναφερόμαστε στον δείκτη γονέα μιας κορυφής γράφοντας $x.parent$. Τροποποιήστε τους αλγορίθμους εισαγωγής και διαγραφής ώστε να λαμβάνουν υπόψιν τους αυτήν την αλλαγή.

4.10. Έστω δυαδικό δέντρο $T = (V, E, r)$ με τιμές φυσικούς αριθμούς. Το T ονομάζεται *δέντρο αθροίσματος* αν κάθε κορυφή του (εκτός από τα φύλλα) έχει ως τιμή το άθροισμα όλων των απογονών της. Παραδείγματος χάρη:



Δώστε αλγόριθμο IsSumTree(T) που δέχεται σαν είσοδο τον δείκτη ρίζας ενός δυαδικού δέντρου T και επιστρέφει "ΝΑΙ" αν το T είναι δέντρο αθροίσματος ή "ΟΧΙ" αν δεν είναι.

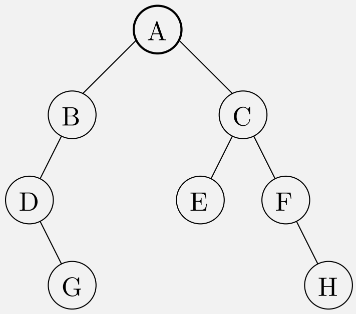
4.11. Θεωρήστε την αναπαράσταση ξένων συνόλων με δέντρα. Έστω x μεταβλητή κόμβου. Γράφοντας $x.item$ αναφερόμαστε στο πεδίο που αποθηκεύει το στοιχείο του x, με $x.parent$ στον δείκτη γονέα του κόμβου και με $x.rank$ στην τάξη του x. Δώστε τους ακόλουθους αλγορίθμους:

1. Find(P) που δέχεται σαν είσοδο δείκτη P που δείχνει προς τον κόμβο που περιέχει το στοιχείο a και επιστρέφει δείκτη προς τον αντιπρόσωπό του συνόλου που ανήκει το a.

2. Union(P, Q) που δέχεται σαν είσοδο δείκτες P, Q που δείχνουν σε δύο κόμβους και ενώνει τα σύνολα που περιέχουν τα στοιχεία των κόμβων (θα πρέπει να γίνεται έλεγχος για το αν τα στοιχεία ανήκουν στο ίδιο σύνολο).

Οι αλγόριθμοί σας θα πρέπει να υλοποιούν και τα δύο «τεχνάσματα» που αναφέρονται στην Παράγραφο 4.7.2.

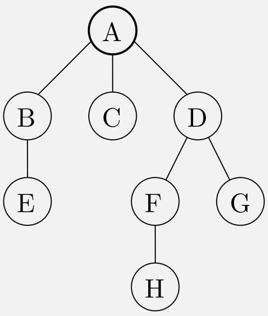
4.12. Δώστε αλγόριθμο BFS1(T) που δέχεται σαν είσοδο τον δείκτη ρίζας T ενός δυαδικού δέντρου και επισκέπτεται κάθε κορυφή του ανά επίπεδο, εφαρμόζοντας σε αυτές μία διαδικασία Process. Για παράδειγμα στο δέντρο:

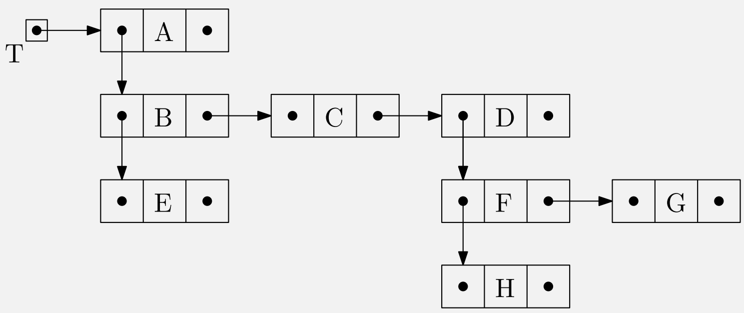


οι κορυφές θα επισκεφθούν με την σειρά: A,B,C,D,E,F,G,H.

4.13. Υποθέστε ότι αναπαριστούμε τα δυαδικά δέντρα των σωρών με διπλά συνδεδεμένες λίστες. Δώστε τους αλγορίθμους που υλοποιούν την εισαγωγή και τη διαγραφή (ρίζας) σύμφωνα με αυτήν την αναπαράσταση. (Αν θέλετε μπορείτε να χρησιμοποιήσετε τον αλγόριθμο BFS1 της Άσκησης 4.12).

4.14. Θεωρήστε την ακόλουθη αναπαράσταση (γενικών) δέντρων. Κάθε κόμβος x αποτελείται από τρία πεδία: το στοιχείο της κορυφής, έναν δείκτη για το αριστερό παιδί της και έναν δείκτη για τον αδερφό της. Στα πεδία αυτά αναφερόμαστε γράφοντας $x.item$, $x.left$ και $x.sibling$ αντίστοιχα. Τέλος χρησιμοποιούμε και έναν δείκτη που δείχνει την ρίζα του δέντρου. Παραδείγματος χάρη:





Δώστε αλγόριθμο BFS2(T) που δέχεται σαν είσοδο τον δείκτη ρίζας T ενός δέντρου και επισκέπτεται κάθε κορυφή του ανά επίπεδο, εφαρμόζοντας σε αυτές μία διαδικασία Process (στο παραπάνω δέντρο οι κορυφές θα επισκεφθούν με την σειρά: A,B,C,D,E,F,G,H.).

ΚΕΦΑΛΑΙΟ 5

ΓΡΑΦΗΜΑΤΑ

Τα *γραφήματα* είναι μια συνδυαστική δομή (γενίκευση των δέντρων που είδαμε στο Κεφάλαιο 4) που παρουσιάζει τεράστιο ενδιαφέρον τόσο από μαθηματική σκοπιά, καθώς αποτελούν έναν αφηρημένο τρόπο περιγραφής πολλών μαθηματικών αντικειμένων, όσο και από την σκοπιά της Θεωρητικής Πληροφορικής εξαιτίας των αμέτρητων εφαρμογών τους. Σκοπός του κεφαλαίου αυτού είναι να παρουσιαστούν ως έννοια, χωρίς να υπεισέλθουμε στις μαθηματικές ιδιότητες που τα διέπουν, και να συζητηθούν (εν συντομία) οι αναπαραστάσεις τους και κάποιοι βασικοί αλγόριθμοι. Οι αλγοριθμικές εφαρμογές τους αποτελούν συνήθως ύλη κάποιου μαθήματος με αμιγές αντικείμενο τους *Αλγόριθμους*. Αντίστοιχα οι μαθηματικές ιδιότητές τους μελετώνται στο μάθημα της Θεωρίας Γραφημάτων.

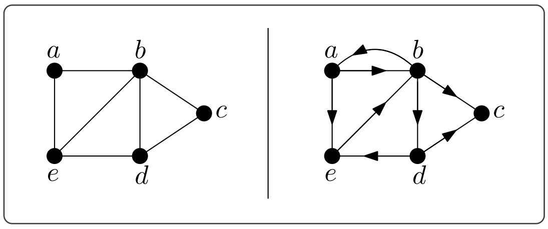
Ο τυπικός ορισμός τους είναι ο ακόλουθος.

Ορισμός 5.0.1. Ένα *γράφημα* $G = (V, E)$ αποτελείται από ένα σύνολο κορυφών V και ένα σύνολο ακμών E, όπου ακμές μπορούν να είναι:

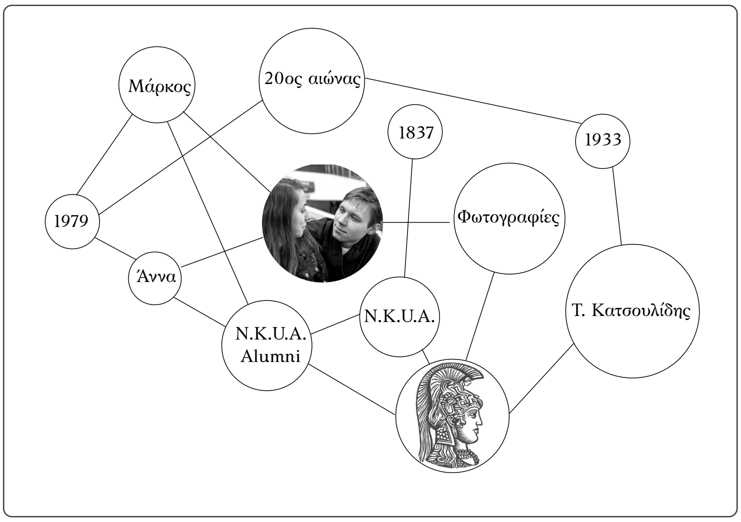
- δισύνολα στοιχείων του V (σε αυτήν την περίπτωση έχουμε ένα *απλό* γράφημα) ή
- διατεταγμένα ζεύγη στοιχείων του V (*κατευθυνόμενο* γράφημα) [1].

Στο Σχήμα 5.0.1 (αριστερά) βλέπουμε το (απλό) γράφημα $G = (V, E)$, με σύνολο κορυφών $V = \{a, b, c, d, e\}$ και ακμών $E = \{\{a, b\}, \{a, e\}, \{b, e\}, \{b, c\}, \{b, d\}, \{c, d\}, \{d, e\}\}$. Χρησιμοποιώντας διαγράμματα σαν και αυτό του Σχήματος 5.0.1 μπορούμε να έχουμε μία πιο εποπτική εικόνα για το γράφημα. Καθώς όμως αυτός ο τρόπος σχεδίασης ενός γραφήματος (σημεία του επιπέδου για τις κορυφές και γραμμές –που ενώνουν κορυφές– για τις

[1] Υπάρχουν και άλλες γενικεύσεις αυτού του ορισμού, όπως για παράδειγμα τα *πολυγραφήματα* όπου επιτρέπεται να έχουμε πολλαπλές ακμές μεταξύ δύο κορυφών. Στα πολυγραφήματα μπορούμε επίσης να επιτρέψουμε και μονοσύνολα κορυφών ως ακμές (οι λεγόμενες «λούπες»). Στα *υπεργραφήματα* οι ακμές μπορούν να περιέχουν και παραπάνω από δύο στοιχεία. Τέλος υπάρχουν και *άπειρα γραφήματα* (όπου το V είναι ένα άπειρο σύνολο) και *τυχαία γραφήματα*.



Σχήμα 5.0.1: *Παραδείγματα γραφημάτων. Αριστερά βλέπουμε ένα απλό γράφημα και δεξιά ένα κατευθυνόμενο.*



Σχήμα 5.0.2: *Παράδειγμα γραφήματος που αναπαριστά πληροφορία.*

ακμές [1]) είναι εντελώς αυθαίρετος, όποτε χρειάζεται να αποδείξουμε κάποια ιδιότητα για το γράφημα θα πρέπει να επιστρέψουμε στον συνολοθεωρητικό ορισμό.

Σε πολλές εφαρμογές (όπως για παράδειγμα στις πλατφόρμες κοινωνικής δικτύωσης σαν το facebook, το instagram κ.λπ.) τα περιεχόμενα αποθηκεύονται ως κορυφές και έπειτα προσθέτονται ακμές για να επισημάνουν τον συσχετισμό περιεχομένου. Για παράδειγμα στο Σχήμα 5.0.2 μπορούμε να δούμε την οργάνωση ποικίλης πληροφορίας σε ένα γράφημα [2].

Σε άλλες εφαρμογές προσθέτουμε *βάρη* στις ακμές έτσι ώστε να αντιπροσωπεύουν το κόστος μετάβασης από τη μία κορυφή στην άλλη. Τέτοιου είδους γραφήματα αναφέρονται στη βιβλιογραφία ως *βεβαρημένα* γραφήματα. Κλασικό παράδειγμα είναι ένας οδικός χάρτης μεταξύ πόλεων, όπου οι πόλεις αποτελούν τις κορυφές του γραφήματος, οι ακμές αντιστοιχούν στις οδικές αρτηρίες που συνδέουν τις πόλεις και τα βάρη στο συνολικό μήκος μίας διαδρομής μεταξύ δύο πόλεων (ή τον μέσο χρόνο που χρειάζεται για να ολοκληρωθεί

[1] Όταν οι ακμές έχουν «κατευθύνσεις» τις δηλώνουμε με ένα βέλος πάνω στις γραμμές.

[2] Για να ξεχωρίσουμε τι τύπου πληροφορία περιέχει η κάθε κορυφή θα χρειαστεί βέβαια να τους προσθέσουμε *ετικέτες* (*tags*).

μία διαδρομή κ.α.).

Προτού δούμε τρόπους να αναπαραστήσουμε τα γραφήματα θα χρειαστεί να δώσουμε κάποιους ορισμούς.

Ορισμός 5.0.2. Έστω (απλό) γράφημα $G = (V, E)$ και κορυφή $u \in V$. Ο *βαθμός* της u ισούται με το πλήθος των ακμών του E στις οποίες εμφανίζεται (ή αλλιώς τις ακμές των οποίων αποτελεί *άκρο*). Τον βαθμό της u τον συμβολίζουμε ως $\deg_G(u)$ (ή και $\deg(u)$ αν είναι ξεκάθαρο ότι αναφερόμαστε στο γράφημα G).

Παραδείγματος χάρη στο απλό γράφημα του Σχήματος 5.0.1 έχουμε ότι $\deg(b) = 4$.

Ορισμός 5.0.3. Έστω κατευθυνόμενο γράφημα $G = (V, E)$ και κορυφή $u \in V$. Ο *εσώβαθμος* της u ισούται με το πλήθος των ακμών στις οποίες εμφανίζεται στη δεύτερη συντεταγμένη (τις ακμές που «καταλήγουν» στη u δηλαδή) και ο *εξώβαθμος* με το πλήθος των ακμών στις οποίες εμφανίζεται στην πρώτη συντεταγμένη (τις ακμές που «ξεκινάνε» από τη u). Για τον εσώβαθμο και τον εξώβαθμο χρησιμοποιούμε τον συμβολισμό $\text{indeg}_G(u)$ και $\text{outdeg}_G(u)$ αντίστοιχα (ή και $\text{indeg}(u)$, $\text{outdeg}(u)$ αν είναι ξεκάθαρο ότι αναφερόμαστε στο γράφημα G).

Παραδείγματος χάρη, στο κατευθυνόμενο γράφημα (στα δεξιά) του Σχήματος 5.0.1 ισχύει ότι $\text{indeg}(b) = 2$ και $\text{outdeg}(b) = 3$.

5.1 Αναπαράσταση γραφημάτων

Μια καλή αφετηρία για την επιλογή τρόπου αναπαράστασης μίας δομής δεδομένων είναι να σκεφτούμε τις λειτουργίες που θα θέλαμε να υποστηρίζει. Δοσμένου ενός γραφήματος $G = (V, E)$ θα μας ενδιέφεραν τα ακόλουθα [1]:

- Έλεγχος για το αν μία ακμή e εμφανίζεται στο γράφημα (αν $e \in E$ δηλαδή). Ο έλεγχος αυτός μπορεί να τεθεί και ως εξής: Εξετάζουμε αν δύο κορυφές ενώνονται με ακμή (είναι *γείτονες*).
- Εύρεση των άκρων μίας ακμής.
- Εύρεση του βαθμού μιας κορυφής.
- Προσθήκη/αφαίρεση κορυφής.
- Προσθήκη/αφαίρεση ακμής.
- Διαπέραση των κορυφών του γραφήματος.
- Διαπέραση των ακμών του γραφήματος.

Αν το γραφημά μας είναι κατευθυνόμενο τότε θα θέλαμε να υποστηρίζει επιπλέον λειτουργίες, όπως παραδείγματος χάρη:

[1] Η λίστα των λειτουργιών όπως αντιλαμβάνεστε είναι ανεξάντλητη. Εδώ αναφέρουμε μόνο τις βασικές.

- Εύρεση εσώβαθμου/εξώβαθμου κορυφής.
- Δημιουργία λίστας με τις εισερχόμενες/εξερχόμενες ακμές σε μία κορυφή, καθώς και των άκρων τους.

Τέλος, αν το γράφημα μας είναι *βεβαρημένο* θα θέλαμε να μπορούμε να ανανεώνουμε τις τιμές στα βάρη των ακμών, να ταξινομούμε τις ακμές σύμφωνα με το βάρος τους κ.λπ..

Δεν θα παρουσιάσουμε τους αλγορίθμους που υλοποιούν τις παραπάνω λειτουργίες, θα συζητήσουμε όμως λίγο τον χρόνο που χρειάζονται ανάλογα με την εκάστοτε αναπαράσταση. Είναι επίσης απαραίτητο να αναφερθούμε στον χώρο που καταναλώνει η κάθε αναπαράσταση, καθώς στις περισσότερες εφαρμογές τα γραφήματα που διαπραγματευόμαστε περιέχουν έναν τεράστιο αριθμό κορυφών ή/και ακμών, καθιστώντας τον χώρο που δαπανάται για την αποθήκευση τους μία εξίσου σημαντική παράμετρο (πέρα από τον χρόνο των αλγορίθμων).

5.1.1 Αναπαράσταση με πίνακες

Ας θεωρήσουμε ότι οι κορυφές ενός γραφήματος $G = (V, E)$ είναι αριθμημένες σύμφωνα με μία αυθαίρετη διάταξη. Μπορούμε να αναπαραστήσουμε το G με έναν $n \times n$ πίνακα A, όπου $n = |V|$, ως εξής:

$$A[i,j] = \begin{cases} 1 & , \text{Αν } \{i,j\} \in E \quad (\text{ή } (i,j) \in E \text{ αν το } G \text{ είναι κατευθυνόμενο}) \\ 0 & , \text{Αλλιώς} \end{cases}$$

Ο πίνακας A ονομάζεται *πίνακας γειτνίασης*. Για παράδειγμα οι πίνακες γειτνίασης των γραφημάτων του Σχήματος 5.0.1, όπου αντιστοιχούμε τους αριθμούς $1, 2, 3, 4, 5$ στις κορυφές με τον προφανή τρόπο (το 1 στην a, το 2 στην b κ.λπ.) είναι:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Παρατηρήστε ότι σε ένα απλό γράφημα ο πίνακας γειτνίασης είναι πάντα συμμετρικός ως προς την κύρια διαγώνιο και ότι στα γραφήματά που δεν περιέχουν «λούπες» οι τιμές της κύριας διαγώνιου θα είναι πάντα 0. Αν έχουμε βεβαρημένο γράφημα μπορούμε αντί για 1 στον πίνακα γειτνίασης να προσθέσουμε το βάρος της εκάστοτε ακμής [1].

Ο πίνακας γειτνίασης έχει $|V|^2$ στοιχεία συνεπώς ο χώρος που χρειαζόμαστε για να τον αποθηκεύσουμε είναι $O(|V|^2)$. Αν το γράφημα είναι *αραιό*, περιέχει δηλαδή λίγες ακμές, θα

[1] Αν υπάρχουν ακμές με βάρος 0 θα πρέπει να χρησιμοποιήσουμε κάποιο άλλο σύμβολο για να επισημάνουμε τη μη-ύπαρξη ακμής.

μπορούσαμε με τεχνάσματα όπως αυτό του Παραδείγματος 1.2.3 να μειώσουμε τον χώρο που χρειαζόμαστε (ο πίνακας θα περιέχει ελάχιστες τιμές διαφορετικές από 0). Μπορούμε βέβαια πολύ πιο απλά να χρησιμοποιήσουμε μία πιο «έξυπνη» αναπαράσταση για το γράφημα (δες Παράγραφο 5.1.2).

Ένας άλλος τρόπος να αναπαραστήσουμε ένα γράφημα είναι με τον λεγόμενο *πίνακα προσπτώσεως*. Ας υποθέσουμε ότι πέρα από τις κορυφές έχουμε απαριθμήσει και τις ακμές ενός γραφήματος $G = (V, E)$. Ο πίνακας προσπτώσεως B έχει διάσταση $|V| \times |E|$ και τα στοιχεία του έχουν τιμές 0 και 1 ως εξής:

$$B[i, j] = \begin{cases} 1 & \text{, Αν η κορυφή } i \text{ είναι άκρο της ακμής } j \\ 0 & \text{, Αλλιώς} \end{cases}$$

Αν έχουμε κατευθυνόμενο γράφημα μπορούμε να χρησιμοποιήσουμε διαφορετικά σύμβολα για να επισημάνουμε το γεγονός ότι μία ακμή «ξεκινάει» από κάποια κορυφή ή «καταλήγει» σε κάποια κορυφή (π.χ. 1 και -1, θα μπορούσαμε βέβαια απλά να χωρίσουμε την πληροφορία αυτή σε δύο διαφορετικούς πίνακες προσπτώσεως). Αν το γράφημα είναι βεβαρημένο μπορούμε πάλι να αντικαταστήσουμε τα 1 στον πίνακα με τα βάρη των ακμών.

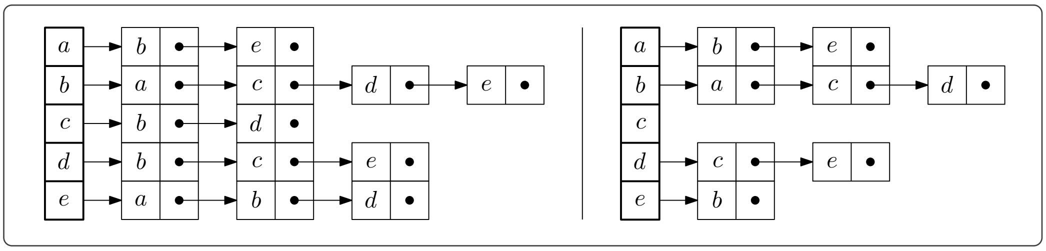
Ο χώρος που χρειαζόμαστε για να αποθηκεύσουμε έναν πίνακα προσπτώσεως είναι $O(|V| \cdot |E|)$. Καθώς το μέγιστο πλήθος ακμών που μπορεί να έχει ένα γράφημα είναι $O(|V|^2)$ (για του λόγου το ακριβές, $\binom{|V|}{2}$ ακμές ένα απλό γράφημα και $2\binom{|V|}{2}$ ένα κατευθυνόμενο) ένας πίνακας προσπτώσεως μπορεί να έχει μέγεθος της τάξης του $O(|V|^3)$. Σε (πολύ) αραιά γραφήματα όμως, όπου $|E| \leq |V|$, το μέγεθός του είναι μικρότερο από $O(|V|^2)$ και επιπλέον έχει το πλεονέκτημα ότι είναι «ακμοκεντρικός», πράγμα απαραίτητος σε μερικές εφαρμογές [1].

Χρόνος βασικών λειτουργιών

Ας εξετάσουμε πρώτα την περίπτωση του πίνακα γειτνίασης. Οι πράξεις της προσθήκης και της αφαίρεσης μίας κορυφής θα μας οδηγήσουν σε έναν καινούριο πίνακα. Θα πρέπει λοιπόν να αντιγράψουμε τον παλιό πίνακα στον νέο με την επικαιροποιημένη διάσταση. Ο χρόνος που θα δαπανήσουμε για αυτό είναι $O(|V|^2)$. Η προσθήκη και η διαγραφή μίας ακμής όμως χρειάζεται σταθερό χρόνο καθώς το μόνο που απαιτεί είναι η αλλαγή της τιμής δύο κελιών του πίνακα στα απλά γραφήματα και ενός στα κατευθυνόμενα (από 0 σε 1 ή το ανάποδο). Ο έλεγχος για το αν δύο κορυφές i, j είναι γειτονικές χρειάζεται χρόνο $O(1)$ καθώς πάλι αρκεί να διαβάσουμε το περιεχόμενο του κελιού $A[i, j]$. Τέλος ο υπολογισμός του βαθμού μίας κορυφής i χρειάζεται χρόνο $O(|V|)$ (μετράμε το πλήθος των μη-μηδενικών στοιχείων στη γραμμή i ή στη στήλη i όταν το γράφημα είναι απλό, στα κατευθυνόμενα γραφήματα το πλήθος των μη-μηδενικών στοιχείων της γραμμή i μας δίνει τον εξώβαθμο και της στήλης i τον εσώβαθμο).

Σε ένα πίνακα προσπτώσεως ο χρόνος για την προσθήκη/αφαίρεση κορυφής ή ακμής είναι $O(|V| \cdot |E|)$ καθώς σε κάθε περίπτωση πρέπει να αντιγράψουμε τον πίνακα σε έναν νέο. Ο έλεγχος για το αν δύο κορυφές είναι γειτονικές χρειάζεται χρόνο $O(|E|)$ (εξετάζουμε

[1] Επίσης είναι ο μόνος τρόπος να αναπαραστήσουμε υπεργραφήματα.



Σχήμα 5.1.1: *Αναπαράσταση των γραφημάτων του Σχήματος 5.0.1 με λίστες γειτνίασης. Στο κατευθυνόμενο γράφημα η λίστα της κορυφής u περιέχει τις κορυφές στις οποίες καταλήγει ακμή που ξεκινάει από τη u. Παρατηρήστε ότι στο απλό γράφημα η κάθε ακμή έχει καταγραφεί δύο φορές, μία για κάθε άκρο της.*

αν οι γραμμές που αντιστοιχούν στις δύο κορυφές έχουν 1 στην ίδια στήλη). Ο ίδιος χρόνος χρειάζεται και για να υπολογιστεί ο βαθμός μίας κορυφής (μετράμε τα μη-μηδενικά στοιχεία της αντίστοιχης γραμμής του πίνακα).

5.1.2 Αναπαράσταση με λίστες γειτνίασης

Ας κάνουμε πρώτα μία σημαντική παρατήρηση. Στην αναπαράσταση με πίνακες οι κορυφές δεν αποθηκεύονταν κάπου ρητά. Αντιθέτως υπονοούνταν έμμεσα από τη συνολική αναπαράσταση του γραφήματος. Αυτό μας στερεί τη δυνατότητα να αποθηκεύσουμε τυχόν επιπλέον πληροφορίες που συνοδεύουν τις κορυφές (αποθηκεύουμε μόνο τη «δομή» του γραφήματος) [1].

Στην αναπαράσταση με *λίστες γειτνίασης* ενός γραφήματος $G = (V, E)$ θα διατηρήσουμε μία συλλογή με τις $|V|$ κορυφές του G είτε σε ένα πίνακα είτε σε μία λίστα ή ακόμα και σε ένα σύνολο [2]. Έτσι θα μπορούμε να αποθηκεύσουμε και την επιπλέον πληροφορία που περιέχει μία κορυφή. Το ίδιο θα κάνουμε και με τις ακμές του γραφήματος. Επιπλέον για κάθε κορυφή $u \in V$ θα διατηρούμε μία λίστα με τους γείτονες της (ή με τι ακμές που την περιέχουν ως άκρο). Η σειρά με την οποία εμφανίζονται οι γείτονες της u στη λίστα μπορεί να είναι αυθαίρετη. Στο Σχήμα 5.1.1 φαίνονται οι λίστες γειτνίασης για τα γραφήματα του Σχήματος 5.0.1.

Ο χώρος που θα χρειαστεί να καταναλώσουμε σε αυτήν την αναπαράσταση είναι της τάξης του $O(\deg(u))$ για κάθε κορυφή $u \in V$, καθώς τόσους κόμβους θα περιέχει η λίστα της κάθε κορυφής, συν φυσικά $O(|V| + |E|)$ επιπλέον χώρο για να αποθηκεύσουμε τις κορυφές και τις ακμές αυτές καθεαυτές. Σύμφωνα με τις προτάσεις που ακολουθούν, στα απλά γραφήματα

$$\sum_{u \in V} \deg(u) = 2 \cdot |E|$$

[1] Θα πρέπει να αποθηκευτούν παραδείγματος χάρη σε κάποιον άλλον πίνακα.

[2] Μπορούμε ακόμα για κάθε κορυφή να κρατήσουμε έναν δείκτη ο οποίος θα δείχνει τις θέσεις μνήμης που έχει αποθηκευτεί η επιπλέον πληροφορία που ακολουθεί την κορυφή.

ενώ στα κατευθυνόμενα

$$\sum_{u \in V} \mathrm{indeg}(u) = \sum_{u \in V} \mathrm{outdeg}(u) = |E|$$

οπότε ο συνολικό χώρος που χρειαζόμαστε για να αποθηκεύσουμε τις λίστες είναι $O(|E|)$, και τελικά θα καταναλώσουμε συνολικά χώρο $O(|V| + |E|)$. Όπως αντιλαμβάνεστε σε ένα αραιό γράφημα αυτός ο τρόπος αναπαράστασης είναι ο πλέον αποδοτικός (όσον αφορά τον χώρο) [1].

Πρόταση 5.1.1. Έστω απλό γράφημα $G = (V, E)$. Ισχύει ότι:

$$\sum_{u \in V} \mathrm{deg}(u) = 2 \cdot |E|$$

Απόδειξη. Κάθε ακμή έχει διπλομετρηθεί στο άθροισμα, μία φορά για κάθε άκρο της. □

Πρόταση 5.1.2. Έστω κατευθυνόμενο γράφημα $G = (V, E)$. Ισχύει ότι:

$$\sum_{u \in V} \mathrm{indeg}(u) = \sum_{u \in V} \mathrm{outdeg}(u) = |E|$$

Απόδειξη. Μία κατευθυνόμενη ακμή μετριέται από μία φορά στο κάθε άθροισμα, καθώς συνεισφέρει μία μονάδα στον εξώβαθμο της κορυφής από την οποία ξεκινάει και μία μονάδα στον εσώβαθμο της κορυφής που καταλήγει. □

Χρόνος βασικών λειτουργιών

Στην αναπαράσταση με λίστες γειτνίασης η προσθήκη κορυφής και ακμής χρειάζεται σταθερό χρόνο (αντιστοιχούν στη δημιουργία μίας κενής λίστας και την προσθήκη κόμβου σε δύο λίστες αντίστοιχα). Η αφαίρεση κορυφής χρειάζεται χρόνο $O(|E|)$ (θα πρέπει να αφαιρεθεί από όλες τις ακμές που την περιείχαν σαν άκρο) ενώ η διαγραφή ακμής χρόνο $O(|V|)$ (θα πρέπει να γίνουν αλλαγές στις λίστες των δύο άκρων της και κάθε λίστα μπορεί να περιέχει $O(|V|)$ κορυφές).

Ο έλεγχος για το αν δύο κορυφές είναι γειτονικές (είτε σε ένα απλό είτε σε ένα κατευθυνόμενο γράφημα) χρειάζεται χρόνο $O(|V|)$, καθώς θα χρειαστεί να ψάξουμε τις λίστες των δύο κορυφών. Τέλος, ο χρόνος υπολογισμού του βαθμού μίας κορυφής u είναι $O(\mathrm{deg}(u))$ σε ένα απλό γράφημα (μετράμε τα στοιχεία που έχει η λίστα της), ο υπολογισμός του εξώβαθμου (για τον ίδιο λόγο) $O(\mathrm{outdeg}(u))$ σε ένα κατευθυνόμενο γράφημα, ενώ του εσώβαθμου

[1] Φανταστείτε το γράφημα του παγκοσμίου ιστού, όπου έχουμε μία κορυφή για κάθε σελίδα και κατευθυνόμενες ακμές από μία σελίδα προς μία άλλη αν στην πρώτη σελίδα υπάρχει υπερσύνδεσμος προς τη δεύτερη. Αν σκεφτούμε ότι αυτό το γράφημα θα έχει περίπου 1,8 δισεκατομμύρια κορυφές (τον δωδέκατο μήνα του έτους 2020) και κάθε σελίδα κατά μέσω όρο έχει ελάχιστους υπερσυνδέσμους –ας πούμε 10– τότε οι ακμές του γραφήματος θα είναι περίπου 18 δισεκατομμύρια. Με λίστες γειτνίασης θα χρειαζόμασταν –αν υποθέσουμε ότι για να αποθηκεύσουμε μία ακμή ή μία κορυφή χρειάζεται ένα Byte– περίπου 17GB για τις ακμές και 1,7 GB για τις κορυφές, οπότε συνολικά κοντά στα 20GB. Με ένα πίνακα γειτνίασης όμως θα χρειαζόμασταν περίπου 2878 Petabytes (δηλαδή περίπου 3017485142 GB) και με ένα πίνακα προσπτώσεως περίπου 28777 Petabytes.

$O(|E|)$, καθώς θα πρέπει να μετρήσουμε τις εμφανίσεις της u σε όλες τις λίστες (που περιέχουν πλήθος κόμβων ίσο με το άθροισμα όλων των εξώβαθμων, δηλαδή ίσο με $|E|$ σύμφωνα με την Πρόταση 5.1.2).

Ανακεφαλαιώνοντας παραθέτουμε έναν πίνακα που περιέχει τα βασικά χαρακτηριστικά για τις αναπαραστάσεις (απλών) γραφημάτων που είδαμε:

	Πίνακας Γειτνίασης	**Πίνακας Προσπτώσεως**	**Λίστες Γειτνίασης**
Χώρος:	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$	$O(\|V\| + \|E\|)$
Προσθήκη κορυφής:	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$	$O(1)$
Προσθήκη ακμής:	$O(1)$	$O(\|V\| \cdot \|E\|)$	$O(1)$
Αφαίρεση κορυφής:	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$	$O(\|E\|)$
Αφαίρεση ακμής:	$O(1)$	$O(\|V\| \cdot \|E\|)$	$O(\|V\|)$
Έλεγχος γειτνίασης:	$O(1)$	$O(\|E\|)$	$O(\|V\|)$
Εύρεση βαθμού:	$O(\|V\|)$	$O(\|E\|)$	$O(\deg(u))$

5.2 Αναζήτηση γραφημάτων

Το πρόβλημα που θα μας απασχολήσει σε αυτήν την παράγραφο είναι η διαπέραση των κορυφών ενός γραφήματος. Είθισται στη βιβλιογραφία οι αλγόριθμοι που υλοποιούν αυτήν τη διαπέραση να αποκαλούνται *αλγόριθμοι αναζήτησης* γραφήματος. Εμείς θα δούμε τους δύο πιο γνωστούς αλγορίθμους: την *αναζήτηση κατά βάθος* και την *αναζήτηση κατά πλάτος*. Η διαφορά τους έγκειται στην σειρά με την οποία «επισκέπτονται» τις κορυφές του γραφήματος.

Στην αναζήτηση γραφημάτων μας βολεύει να χρησιμοποιούμε την αναπαράσταση με λίστες γειτνίασης. Ο λόγος είναι ότι στους αλγορίθμους αναζήτησης χρειάζεται να βρίσκουμε τις κορυφές με τις οποίες συνδέεται (μέσω ακμής) κάποια δοσμένη κορυφή. Παρατηρήστε ότι οι λίστες γειτνίασης μας παρέχουν άμεσα αυτήν την πληροφορία.

Σε όσα ακολουθούν θα επικεντρωθούμε στα μη-κατευθυνόμενα γραφήματα. Οι πλειονότητα όμως όσων θα δούμε μεταφέρεται άμεσα και σε κατευθυνόμενα γραφήματα.

5.2.1 Αναζήτηση κατά βάθος (DFS)

Μας δίνουν ένα γράφημα $G = (V, E)$ και μία κορυφή u (ή πιο απλά επιλέγουμε εμείς μία κορυφή για να ξεκινήσουμε). Σκοπός μας είναι να επισκεφθούμε όλες τις κορυφές που είναι «προσβάσιμες» από τη u. Η ιδέα του αλγορίθμου είναι η ακόλουθη:

> *Ξεκινάμε από τη u και επισκεπτόμαστε κάθε φορά μία γειτονική κορυφή που δεν έχουμε ήδη επισκεφθεί. Όταν τελικά φτάσουμε σε μία κορυφή της οποίας έχουμε ήδη επισκεφθεί όλες τις γειτονικές κορυφές, γυρνάμε πίσω στην αμέσως προηγούμενη που επισκεφθήκαμε και ελέγχουμε αν μας έχει «ξεφύγει» κάποια [1]. Επαναλαμβάνουμε αυτήν την οπισθοδρόμηση και επανεξέταση έως ότου επισκεφθούμε όλες τις κορυφές του γραφήματος.*

Είναι εμφανές ότι η στρατηγική που ακολουθούμε είναι να προχωράμε όλο και πιο «βαθιά» στο γράφημα, όσο αυτό είναι δυνατό (κινούμενοι φυσικά από κορυφή σε γειτονική κορυφή). Όταν πια φτάσουμε σε ένα σημείο όπου δεν μπορούμε να πάμε πιο βαθιά (γιατί έχουμε ήδη επισκεφθεί όλες τις γειτονικές κορυφές) γυρνάμε προς τα πίσω και εξετάζουμε αν μπορούμε να ξεκινήσουμε κάποια καινούργια «εφόρμηση».

Για να υλοποιήσουμε την παραπάνω ιδέα θα χρειαστούμε μία «κιμωλία» με την οποία θα σημειώνουμε τις κορυφές που έχουμε επισκεφθεί και ένα «νήμα» το οποίο θα χρησιμοποιήσουμε για να γυρίζουμε πίσω στις κορυφές που επισκεφθήκαμε:

- Για «κιμωλία» θα χρησιμοποιήσουμε έναν βοηθητικό πίνακα με τόσες θέσεις όσες και οι κορυφές του γραφήματος. Στην αρχή κάθε θέση του πίνακα θα έχει τιμή False και κατά την εκτέλεση του αλγορίθμου όποτε επισκεπτόμαστε μία κορυφή θα αλλάζουμε την τιμή στην αντίστοιχη θέση σε True.

- Για «νήμα» θα χρησιμοποιήσουμε μία στοίβα στην οποία θα εισάγουμε μία-μία τις κορυφές που επισκεπτόμαστε. Παρατηρήστε ότι κατά την οπισθοδρόμηση θα χρειαστεί να ξαναπεράσουμε από τις κορυφές ακολουθώντας τη λογική LIFO.

Προτού δούμε τον αλγόριθμο θα χρειαστεί να δώσουμε κάποιους ορισμούς και να κάνουμε κάποιες παρατηρήσεις.

Ορισμός 5.2.1. Έστω γράφημα $G = (V, E)$. *Μονοπάτι* στο G με άκρα τις κορυφές $x, y \in V$ είναι μία ακολουθία διακεκριμένων κορυφών $u_1, \ldots, u_n$ του V, με $u_1 = x$ και $u_2 = y$, τέτοια ώστε για κάθε $i = 1, \ldots, n-1$ να ισχύει ότι $\{u_i, u_{i+1}\} \in E$. Το *μήκος* του μονοπατιού ισούται με $n - 1$ (ισούται δηλαδή με το πλήθος των ακμών που περιέχει).

Ορισμός 5.2.2. Ένα γράφημα $G = (V, E)$ καλείται *συνεκτικό* αν και μόνο αν για κάθε $x, y \in V$ υπάρχει μονοπάτι με άκρα τις κορυφές x, y.

Παρατηρήστε ότι αν το γράφημα δεν είναι συνεκτικό η διαδικασία που περιγράψαμε παραπάνω θα επισκεφθεί μόνο τις κορυφές που βρίσκονται στο συνεκτικό κομμάτι του γραφήματος (*συνεκτική συνιστώσα*) που περιέχει την κορυφή αφετηρίας u. Για να επισκεφθούμε όλες τις κορυφές του γραφήματος θα πρέπει να επανεκκινήσουμε αυτήν τη διαδικασία έχοντας ως αφετηρία μία καινούργια κορυφή που δεν έχουμε ακόμα επισκεφθεί (και άρα θα

[1] Δηλαδή ξαναελέγχουμε αν για αυτήν την κορυφή υπάρχει γειτονική κορυφή την οποία δεν έχουμε ήδη επισκεφθεί

βρίσκεται σε διαφορετική συνιστώσα από τη u). Ανάλογα με το πλήθος των συνεκτικών συνιστωσών του γραφήματος ενδέχεται να επανεκκινήσουμε τη διαδικασία αρκετές φορές.

Θα ξεκινήσουμε με έναν βοηθητικό αλγόριθμο που επισκέπτεται όλες τις κορυφές της συνεκτικής συνιστώσας. Αν u κορυφή του γραφήματος με $Adj[u]$ θα συμβολίζουμε τη λίστα με τους γείτονές της. Στον πίνακα που θα χρησιμοποιήσουμε σαν «κιμωλία» θα θεωρούμε ότι η θέση i αντιστοιχεί στην κορυφή i. Χάριν απλότητας όμως θα συγχέουμε την αρίθμηση των κελιών με τις κορυφές, όπως επίσης θα εισάγουμε κατευθείαν τις κορυφές στη στοίβα αντί να εισάγουμε π.χ. τον δείκτη κεφαλής που δείχνει τη λίστα γειτνίασης της εκάστοτε κορυφής. Ο αλγόριθμος θα είναι δηλαδή περισσότερο περιγραφικός παρά τυπικός.

DFS-Explore(G, M, u)

Είσοδος: Γράφημα $G = (V, E)$, πίνακας M με $|V|$ κελιά που περιέχουν τιμές True/False και κορυφή $u \in V$

Έξοδος : Ο πίνακας M όπου έχει αλλάξει η τιμή σε True για όλες τις κορυφές που είναι προσβάσιμες από τη u

1 $S \leftarrow$ **new stack**
2 Push(u, S)
3 **while not** IsEmpty(S)
4     $v \leftarrow$ Pop(S)
5   **if** $M[v] =$ False **then**
6         $M[v] \leftarrow$ True
7   **for** $w \in Adj[v]$
8   Push(w, S)
9 **return** M

Θα αναρωτιέστε με ποια σειρά εξετάζονται οι κορυφές στο βήμα 7. Η σειρά που ακολουθείτε είναι η σειρά των κορυφών στη λίστα γειτνίασης της v. Παραδείγματος χάρη στο γράφημα του Σχήματος 5.0.1 αν ξεκινήσουμε από την κορυφή a θα επισκεφθούμε τις κορυφές με τη σειρά a, e, d, c, b λόγω του ότι το γράφημα έχει αναπαρασταθεί με τις λίστες γειτνίασης του Σχήματος 5.1.1. Θα αποδείξουμε ότι η σειρά αυτή δεν επηρεάζει ούτε την ορθότητα αλλά ούτε και τη χρονική πολυπλοκότητα του αλγορίθμου.

Πρόταση 5.2.3. Έστω συνεκτικό γράφημα $G = (V, E)$. Για οποιαδήποτε κορυφή $u \in V$ όλες οι τιμές του πίνακα που θα επιστρέψει η εκτέλεση DFS-Explore(G, M, u) (όπου M πίνακας με $|V|$ κελιά που περιέχουν τιμή False) θα είναι True.

Απόδειξη. Έστω (προς άτοπο) κορυφή $z \in V$ με $M[z] =$ False μετά την εκτέλεση του DFS-Explore(G, M, u). Θεωρούμε τυχαίο μονοπάτι από τη u στη z και θεωρούμε ότι η κορυφή $y \in V$ είναι η πρώτη κορυφή σε αυτό το μονοπάτι με $M[y] =$ False [1] και x η αμέσως προηγούμενή της στο μονοπάτι. Αφού $M[x] =$ True η x κάποια στιγμή μπήκε στη στοίβα

[1] Η y δεν είναι κατ' ανάγκη η z. Επιπλέον, δεν μπορεί να είναι η u καθώς στο βήμα 2 η u μπαίνει στη στοίβα και στην πρώτη επανάληψη του while θα βγει από αυτή, άρα τελικά $M[u] =$ True.

[1]. Παρατηρήστε ότι $y \in Adj[x]$, οπότε όταν θα βγει η κορυφή x από τη στοίβα στο βήμα 8 θα μπει στη στοίβα και η y. Αυτό σημαίνει ότι και η y με τη σειρά της θα βγει κάποτε από τη στοίβα (ο αλγόριθμος τερματίζει όταν η στοίβα έχει αδειάσει) και στο βήμα 6 το $M[y]$ θα πάρει τιμή True, πράγμα που είναι άτοπο. $\square$

Από την παραπάνω πρόταση έπεται ότι αν τρέξουμε τον DFS-Explore με αφετηρία μία κορυφή από κάθε συνεκτική συνιστώσα του γραφήματος τότε τελικά θα επισκεφθούμε όλες του τις κορυφές. Ο αλγόριθμος που υλοποιεί την αναζήτηση κατά βάθος είναι ο ακόλουθος.

DFS(G)

Είσοδος: Γράφημα $G = (V, E)$
Έξοδος : Τίποτα (επισκεπτόμαστε όλες τις κορυφές του γραφήματος)

1 $M \leftarrow$ **new matrix**$(|V|)$
2 **for** $u \in V$
3     $M[u] \leftarrow$ False
4 **for** $u \in V$
5   **if** $M[u] =$ False **then**
6         $M \leftarrow$ DFS-Explore(G, M, u)

Για τον χρόνο του αλγορίθμου DFS παρατηρήστε ότι, όσον αφορά τον συνολικό χρόνο για το for στη γραμμή 2 η απάντηση είναι ξεκάθαρη: Σε κάθε επανάληψη γίνεται ένα σταθερό πλήθος πράξεων που χρειάζονται σταθερό χρόνο (άρα $O(|V|)$ συνολικά). Στο δεύτερο for (γραμμή 4) βλέπουμε ότι ο DFS-Explore θα κληθεί το πολύ μία φορά για κάθε κορυφή του γραφήματος (για την ακρίβεια ακριβώς τόσες φορές όσες και οι συνεκτικές συνιστώσες). Στον DFS-Explore κάθε κορυφή θα την επισκεφθούμε ακριβώς μία φορά (δηλαδή η αντίστοιχη τιμή στον M θα γίνει True) και τη φορά που θα την επισκεφθούμε θα κάνουμε τόσες εισαγωγές στην στοίβα όσες και οι κορυφές στη λίστα γειτνίασής της. Συνεπώς θα γίνουν:

$$\sum_{u \in V} |Adj[u]| = |E|$$

εισαγωγές (με $|Adj[u]|$ συμβολίζουμε το πλήθος στοιχείων στη λίστα γειτνίασης της u), και ως συνέπεια θα γίνουν άλλες τόσες εξαγωγές και λοιπές πράξεις σταθερού χρόνου. Οπότε ο χρόνος που χρειαζόμαστε για το δεύτερο for είναι $O(|E|)$. Τελικά, ο συνολικός χρόνος του DFS είναι $O(|V| + |E|)$.

Υπάρχει ένας απλός τρόπος να βελτιώσουμε λίγο τον αλγόριθμο DFS-Explore. Θα παρατηρήσατε ότι ο αλγόριθμος ενδέχεται να εισάγει στη στοίβα πολλές φορές την ίδια κορυφή. Μπορούμε να κάνουμε οικονομία στις εισαγωγές (και κατ' επέκταση στις εξαγωγές) αν χρησιμοποιήσουμε στον πίνακα M και μια ενδιάμεση τιμή εκτός από τις True και False, ας πούμε τη Found. Την τιμή αυτή θα τη χρησιμοποιούμε για τις κορυφές που έχουν εισαχθεί στη στοίβα αλλά δεν έχουν ακόμα επεξεργασθεί (αυτές θα έχουν τιμή True). Τέλος,

[1] Για να πάρει το $M[x]$ τιμή True θα πρέπει πρώτα η x να βγει από τη στοίβα, συνεπώς θα πρέπει σε κάποια πρότερη χρονική στιγμή να είχε μπει στη στοίβα

στο βήμα 8 θα προσθέτουμε στη στοίβα μόνο τις κορυφές με τιμή False και έπειτα φυσικά θα αλλάζουμε την τιμή τους σε Found.

Παρατηρήστε επίσης ότι πλέον δεν θα υπάρχουν στη στοίβα κορυφές με τιμή True. Ο τροποποιημένος αλγόριθμος είναι ο ακόλουθος:

DFS-Explore(G, M, u)

Είσοδος: Γράφημα $G = (V, E)$, πίνακας M με $|V|$ κελιά που περιέχουν τιμές True/False και κορυφή $u \in V$

Έξοδος : Ο πίνακας M όπου έχει αλλάξει η τιμή σε True για όλες τις κορυφές που είναι προσβάσιμες από τη u

1 $S \leftarrow$**new stack**
2 Push(u, S)
3 **while not** IsEmpty(S)
4     $v \leftarrow$ Pop(S)
5     $M[v] \leftarrow$ True
6   **for** $w \in Adj[v]$
7   **if** $M[w] =$ False **then**
8   Push(w, S)
9             $M[w] \leftarrow$ Found
10 **return** M

Ο χρόνος του DFS με αυτήν την παραλλαγή του DFS-Explore δεν βελτιώνεται ουσιαστικά (παραμένει δηλαδή $O(|V| + |E|)$).

5.2.2 Αναζήτηση κατά πλάτος (BFS)

Η ιδέα της αναζήτησης κατά πλάτος είναι η εξής:

> *Ξεκινώντας από μία κορυφή επισκεπτόμαστε όλες τις γειτονικές της κορυφές (που δεν έχουμε ακόμα επισκεφθεί) και έπειτα επαναλαμβάνουμε για τις καινούργιες κορυφές που επισκεφθήκαμε.*

Προσπαθούμε συνεπώς να μην εφορμάμε βαθιά στο γράφημα (όπως κάναμε στην αναζήτηση κατά βάθος) αλλά να επισκεπτόμαστε όσες περισσότερες κορυφές μπορούμε από την κορυφή που βρισκόμαστε την κάθε στιγμή. Η αναζήτηση συνεπώς αναπτύσσεται «κατά πλάτος». Παρατηρήστε επίσης ότι με αυτόν τον τρόπο θα επισκεπτόμαστε τις κορυφές της κάθε συνεκτικής συνιστώσας σε κύματα, από αυτές που είναι πιο «κοντά» στην αφετηρία μας προς αυτές που είναι πιο «μακριά». Για να κάνουμε πιο τυπικές τις έννοιες «κοντά» και «μακριά» θα χρειαστεί να ορίσουμε μία έννοια απόστασης μεταξύ κορυφών.

Ορισμός 5.2.4. Έστω γράφημα $G = (V, E)$ και κορυφές $u, v \in V$. Η *απόσταση* των u, v συμβολισμός $\deg_G(u, v)$, ισούται με το μήκος του συντομότερου μονοπατιού με άκρα αυτές τις δύο κορυφές [1].

Στην αναζήτηση κατά πλάτος επισκεπτόμαστε πρώτα τις κορυφές που απέχουν απόσταση 1 από την αφετηρία, μετά αυτές που απέχουν απόσταση 2 και ούτω καθεξής. Πάλι θα χρειαστούμε μία «κιμωλία» για να σημειώνουμε τις κορυφές που έχουμε επισκεφθεί, αλλά πλέον δεν χρειαζόμαστε το «νήμα» που θα χρησιμοποιούσαμε για την οπισθοδρόμηση [2]. Αντ' αυτού θα χρειαστεί να βρούμε έναν τρόπο να χρίζουμε αφετηρία τις κορυφές σύμφωνα με τη σειρά με την οποία τις επισκεφθήκαμε (η πρώτη που επισκεφθήκαμε θα είναι και η πρώτη που θα γίνει νέα αφετηρία για την αναζήτηση). Είναι εμφανές ότι η δομή δεδομένων που θα πρέπει να χρησιμοποιήσουμε είναι η ουρά.

Θα ξεκινήσουμε πάλι με έναν βοηθητικό αλγόριθμο που επισκέπτεται τις κορυφές της συνεκτικής συνιστώσας, και θα χρησιμοποιήσουμε την ενδιάμεση τιμή Found έτσι ώστε κάθε κορυφή να μπαίνει στην ουρά ακριβώς μία φορά:

BFS-Explore(G, M, u)

Είσοδος: Γράφημα $G = (V, E)$, πίνακας M με $|V|$ κελιά που περιέχουν τιμές True/False και κορυφή $u \in V$

Έξοδος : Ο πίνακας M όπου έχει αλλάξει η τιμή σε True για όλες τις κορυφές που είναι προσβάσιμες από τη u

```
1  Q ← new queue
2  Enqueue(u, Q)
3  while not IsEmpty(Q)
4      v ← Dequeue(Q)
5      M[v] ← True
6      for w ∈ Adj[v]
7          if M[w] = False then
8              Enqueue(w, Q)
9              M[w] ← Found
10 return M
```

Πάλι η σειρά που εξετάζονται οι κορυφές στο βήμα 6 είναι η σειρά που εμφανίζονται στη λίστα γειτνίασης της v: Στο γράφημα του Σχήματος 5.0.1 αν ξεκινήσουμε από την κορυφή a θα επισκεφθούμε τις κορυφές με τη σειρά a, b, e, c, d.

Η ορθότητα του αλγορίθμου αποδεικνύεται στην ακόλουθη πρόταση.

Πρόταση 5.2.5. *Έστω συνεκτικό γράφημα $G = (V, E)$. Για οποιαδήποτε κορυφή $u \in V$ όλες οι τιμές του πίνακα που θα επιστρέψει η εκτέλεση* BFS-Explore(G, M, u) *(όπου M πίνακας με $|V|$ κελιά που περιέχουν τιμή* False*) θα είναι* True.

Απόδειξη. Αρκεί να δείξουμε ότι για κάθε κορυφή v που απέχει απόσταση d $(< +\infty)$ από τη u ισχύει ότι $M[v] =$ True. Αυτό θα το δείξουμε με επαγωγή στο d:

[1] Αν δεν υπάρχει τέτοιο μονοπάτι (αν δηλαδή οι κορυφές ανήκουν σε διαφορετικές συνεκτικές συνιστώσες) η απόσταση είναι $+\infty$.

[2] Δεν θα γυρνάμε προς τα πίσω, θα «πηδάμε» από κορυφή σε κορυφή.

Επαγωγική Βάση: Έστω $v \in V$ που απέχει απόσταση 1 από τη u (δηλαδή συνδέεται με ακμή με τη u). Στην πρώτη εφαρμογή του βήματος 4 θα βγει από την ουρά η u και έπειτα (στο βήμα 6) θα μπει η v (μαζί με όλες τις υπόλοιπες που συνδέονται με ακμή με τη u). Όταν τελικά βγει η v από την ουρά το $M[v]$ θα γίνει True.

Επαγωγική Υπόθεση: Έστω ότι για κάθε $v \in V$ που απέχει απόσταση d από τη u ισχύει ότι $M[v] = \text{True}$.

Επαγωγικό Βήμα: Έστω $v \in V$ που απέχει απόσταση $d + 1$ από τη u. Θεωρούμε τα συντομότερα μονοπάτια από τη u στη v και την αμέσως προηγούμενες από τη v κορυφές σε αυτά τα μονοπάτια. Αυτές οι κορυφές απέχουν απόσταση d από τη u άρα (από την επαγωγική υπόθεση) η τιμή του M στις αντίστοιχες θέσεις θα είναι True. Ας υποθέσουμε ότι x είναι η πρώτη από αυτές που μπήκε στην ουρά και κατά συνέπεια η πρώτη που έπειτα βγήκε από την ουρά. Όταν η x βγήκε από την ουρά αφού οι x και v συνδέονται με ακμή και $M[v] = \text{False}$ [1] η v θα μπήκε με τη σειρά της στην ουρά. Συνεπώς όταν αργότερα βγήκε η v από την ουρά το $M[v]$ θα έγινε True. □

Προφανώς θα χρειαστεί να τρέξουμε τον BFS-Explore για μία κορυφή κάθε συνεκτικής συνιστώσας του γραφήματος:

BFS(G)

Είσοδος: Γράφημα $G = (V, E)$
Έξοδος : Τίποτα (επισκεπτόμαστε όλες τις κορυφές του γραφήματος)

$M \leftarrow$ **new matrix**$(|V|)$
for $u \in V$
  $M[u] \leftarrow$ False
for $u \in V$
  if $M[u] = \text{False}$ **then**
    $M \leftarrow$ BFS-Explore(G, M, u)

Ο χρόνος του BFS είναι $O(|V| + |E|)$ καθώς και πάλι στο δεύτερο for θα χρειαστεί να ελέγξουμε για κάθε κορυφή ακριβώς μία φορά τη λίστα γειτνίασής της.

5.3 Ελάχιστο επικαλύπτον δέντρο

Υπάρχουν πολλές εφαρμογές στις οποίες μας ενδιαφέρει να κρατήσουμε κάποιους εικονικούς «σταθμούς» συνδεδεμένους ούτως ώστε να μπορούμε να «μεταφέρουμε πληροφορία» από οποιονδήποτε σταθμό σε οποιονδήποτε άλλο σταθμό (ενδεχομένως κάνοντας πρώτα κάποιες ενδιάμεσες στάσεις). Στις περισσότερες από αυτές τις εφαρμογές οι συνδέσεις που θα χρησιμοποιήσουμε κοστίζουν και θα πρέπει να ελαχιστοποιήσουμε το συνολικό κόστος. Αυτό σημαίνει ότι θα χρειαστεί να κρατήσουμε τις λιγότερες δυνατές συνδέσεις.

[1] Η τιμή του δεν θα μπορούσε να είναι True γιατί τότε δεν θα είχαμε κάτι να αποδείξουμε, αλλά ούτε και Found καθώς η x είναι η πρώτη κορυφή που βγαίνει από την ουρά και γειτονεύει με τη v σε συντομότερο μονοπάτι.

Συμβαίνει συχνά κάποιες συνδέσεις μεταξύ σταθμών να μην είναι υλοποιήσιμες, όπως επίσης συμβαίνει το κόστος των συνδέσεων να μην είναι ομοιόμορφο, συνεπώς κάποιες συνδέσεις θα κοστίζουν περισσότερο από κάποιες άλλες.

Ας εκφράσουμε το παραπάνω πρόβλημα χρησιμοποιώντας γραφήματα. Έστω γράφημα $G = (V, E)$ όπου οι κορυφές του αντιστοιχούν στους σταθμούς και οι ακμές του στις επιτρεπτές συνδέσεις μεταξύ σταθμών [1]. Ας υποθέσουμε επίσης ότι μας δίνεται και μια συνάρτηση $w : E \to \mathbb{N} \setminus \{0\}$ που ορίζει πόσο κοστίζει η κάθε σύνδεση (την τιμή $w(e)$ για μία ακμή $e \in E$ την αποκαλούμε *βάρος* της e). Το ζητούμενο φυσικά είναι να βρεθεί ένα σύνολο ακμών $E' \subseteq E$ τέτοιο ώστε για οποιεσδήποτε κορυφές $u, v \in V$ να υπάρχει μονοπάτι που να τις συνδέει και χρησιμοποιεί μόνο ακμές του E' και επιπλέον ελαχιστοποιεί την ποσότητα $W = \sum_{e \in E'} w(e)$ (το *συνολικό βάρος*).

Ας γενικεύσουμε τον ορισμό των δέντρων που είδαμε στο προηγούμενο κεφάλαιο.

Ορισμός 5.3.1. Ένα γράφημα $G = (V, E)$ καλείται δέντρο αν είναι συνεκτικό και δεν περιέχει κύκλο [2].

Συμβολισμός 5.3.2. Έστω γράφημα $G = (V, E)$ και $E' \subseteq E$. Με $G[E']$ συμβολίζουμε το γράφημα που *ενάγουν* οι ακμές του E' (και οι κορυφές που εμφανίζονται στο E').

Για ένα σύνολο ακμών E' που αποτελεί λύση του προβλήματός μας είναι προφανές ότι το $G[E']$ είναι συνεκτικό γράφημα. Αυτό το γράφημα θα είναι δέντρο όπως φαίνεται από την ακόλουθη πρόταση.

Πρόταση 5.3.3. Έστω συνεκτικό γράφημα $G = (V, E)$ και $e \in E$ ακμή που ανήκει σε κύκλο. Η αφαίρεση της e δεν πλήττει την συνεκτικότητά του G.

Απόδειξη. Έστω G' το γράφημα μετά την αφαίρεση της e. Έστω (προς άτοπο) ότι το G' δεν είναι συνεκτικό, τότε υπάρχουν κορυφές u, v που δεν συνδέονται με μονοπάτι. Στο G οι u, v συνδέονταν με μονοπάτι το οποίο «κατέστρεψε» η αφαίρεση της e. Παρατηρήστε ότι αντί για την e μπορούμε να χρησιμοποιήσουμε όλες τις άλλες ακμές του κύκλου που ανήκει η e και να κατασκευάσουμε στο G' μονοπάτι που ενώνει τις u, v. Άτοπο. □

Παρατηρήστε ότι αν μία ακμή $e \in E'$ σχημάτιζε κύκλο τότε θα μπορούσαμε να την αφαιρέσουμε διατηρώντας το γράφημα $G[E']$ συνεκτικό, μειώνοντας το συνολικό άθροισμα βαρών (αφού τα βάρη είναι θετικοί αριθμοί). Καθώς το E' ελαχιστοποιεί το συνολικό βάρος έπεται ότι στο $G[E']$ δεν μπορεί να υπάρχει κύκλος και ως συνεκτικό και άκυκλο γράφημα το $G[E']$ αποτελεί δέντρο.

Ορισμός 5.3.4. Έστω συνεκτικό γράφημα $G = (V, E)$. Ένα δέντρο $T = (V', E')$ με $V' = V$ και $E' \subseteq E$ καλείται *επικαλύπτον δέντρο.*

[1] Το πρόβλημα φυσικά έχει νόημα όταν το G είναι συνεκτικό.

[2] Η διαφορά με τα δέντρα που μελετήσαμε στο προηγούμενο κεφάλαιο είναι ότι οι ακμές δεν έχουν κατευθύνσεις και ότι δεν έχουμε ξεχωρίσει μία κορυφή που θα παίξει τον ρόλο της ρίζας του δέντρου. Εύκολα μπορούμε να επαληθεύσουμε ότι για αυτά τα δέντρα ισχύουν αντίστοιχες ιδιότητες με τα δέντρα του Κεφαλαίου 4, όπως π.χ. ότι έχουν τουλάχιστον δύο φύλλα (αν έχουν τουλάχιστον δύο κορυφές) ή ότι οποιεσδήποτε δύο κορυφές συνδέονται με μοναδικό μονοπάτι.

Συνεπώς το ζητούμενο είναι να βρεθεί επικαλύπτον δέντρο που ελαχιστοποιεί το συνολικό βάρος ακμών. Η ακόλουθη πρόταση μας καθορίζει το πλήθος ακμών που θα έχει αυτό το δέντρο.

Πρόταση 5.3.5. Ένα δέντρο $T = (V, E)$ με n κορυφές έχει ακριβώς $n - 1$ ακμές.

Απόδειξη. Θα το δείξουμε με επαγωγή στο n.

Επαγωγική Βάση: Όταν το δέντρο έχει ακριβώς μία κορυφή δεν θα έχει καμία ακμή.

Επαγωγική Υπόθεση: Υποθέτουμε ότι κάθε δέντρο με n κορυφές έχει ακριβώς $n - 1$ ακμές.

Επαγωγικό Βήμα: Έστω δέντρο T με $n + 1$ κορυφές. Αφαιρούμε φύλλο του T και παίρνουμε δέντρο T' με n κορυφές και (από την επαγωγική υπόθεση) $n - 1$ ακμές. Αυτό σημαίνει ότι το T έχει $(n - 1) + 1 = n$ ακμές. □

Πρόταση 5.3.6. Ένα συνεκτικό γράφημα $G = (V, E)$ με n κορυφές και ακριβώς $n - 1$ ακμές είναι δέντρο.

Απόδειξη. Έστω (προς άτοπο) ότι το G δεν είναι δέντρο, δηλαδή ότι περιέχει κύκλους. Από την Πρόταση 5.3.3 μπορούμε να αφαιρέσουμε για κάθε κύκλο μία ακμή και το γράφημα τελικά να παραμείνει συνεκτικό. Εφόσον το γράφημα θα είναι συνεκτικό και δεν θα περιέχει κύκλους θα είναι δέντρο με n κορυφές και (από την Πρόταση 5.3.5) $n-1$ ακμές. Αυτό μπορεί να συμβαίνει μόνο αν τελικά δεν αφαιρέσαμε καμία ακμή, δηλαδή όταν το G εξ αρχής ήταν δέντρο. Άτοπο. □

Ισχύει ότι οποιεσδήποτε $|V| - 1$ ακμές και να διαλέξουμε αν δεν σχηματίζουν κύκλο τότε θα σχηματίζουν ένα επικαλύπτον δέντρο. Για να το αποδείξουμε αυτό χρειαζόμαστε την ακόλουθη πρόταση.

Πρόταση 5.3.7. Κάθε γράφημα $G = (V, E)$ με $|E| \geq |V| - 1$ που δεν περιέχει κύκλο είναι συνεκτικό [1].

Απόδειξη. Έστω (προς άτοπο) ότι το G δεν είναι συνεκτικό, επομένως θα περιέχει $k \geq 2$ συνεκτικές συνιστώσες που (ως άκυκλα γραφήματα) θα είναι δέντρα. Αν το πλήθος των κορυφών της κάθε συνιστώσας είναι $n_1, n_2, \ldots, n_k$ (με $n_1 + n_2 + \cdots + n_k = |V|$) από την Πρόταση 5.3.5 έπεται ότι θα έχουν ακριβώς $n_1 - 1, n_2 - 1, \ldots, n_k - 1$ ακμές αντίστοιχα. Αυτό σημαίνει ότι το πλήθος ακμών του G θα είναι συνολικά:

$$|E| = \sum_{i=1}^{k}(n_i - 1) = \sum_{i=1}^{k} n_i - k = |V| - k < |V| - 1$$

Άτοπο. □

[1] Παρατηρήστε ότι αν επιλέξουμε $|V| - 1$ ακμές που δεν σχηματίζουν κύκλο, από την Πρόταση 5.3.6 έπεται ότι το γράφημα που θα ενάγουν θα είναι δέντρο και μάλιστα επικαλύπτον.

Έχουμε λοιπόν καταλήξει σε έναν υποτυπώδη αλγόριθμο εύρεσης ενός επικαλύπτον δέντρου:

> *Με είσοδο ένα γράφημα $G = (V, E)$ διαλέγουμε $|V| - 1$ ακμές που δεν σχηματίζουν κύκλο.*

Εμείς φυσικά θέλουμε το επικαλύπτον δέντρο με το ελάχιστο συνολικό κόστος. Συνεπώς θα πρέπει κάθε φορά να διαλέγουμε την «ελαφρύτερη» ακμή που δεν σχηματίζει κύκλο. Θα δείξουμε ότι αυτές οι «άπληστες» επιλογές θα μας οδηγήσουν τελικά σε ένα ελάχιστο επικαλύπτον δέντρο.

Η πρώτη μας επιλογή θα είναι η ελαφρύτερη ακμή από όλες. Η επιλογή μας είναι σωστή σύμφωνα με την ακόλουθη Παρατήρηση.

Παρατήρηση 5.3.8. Έστω γράφημα $G = (V, E)$ και συνάρτηση βαρών $w : E \to \mathbb{N} \setminus \{0\}$. Αν $e \in E$ είναι η ακμή με το μικρότερο βάρος τότε υπάρχει ελάχιστο επικαλύπτον δέντρο που περιέχει την e.

Πράγματι έστω $T = (V, E')$ ελάχιστο επικαλύπτον δέντρο του G που δεν περιέχει την e. Αν προσθέσουμε την e στο T θα δημιουργηθεί κύκλος [1]. Αν αφαιρέσουμε μία οποιαδήποτε ακμή αυτού του κύκλου από το T και προσθέσουμε την e θα πάρουμε επικαλύπτον δέντρο με συνολικό βάρος το πολύ το βάρος του T.

Αυτή η ιδέα θα μας οδηγήσει στην *ιδιότητα της τομής* που θα αιτιολογήσει την ορθότητα των υπόλοιπων άπληστων επιλογών μας.

Ορισμός 5.3.9. Έστω γράφημα $G = (V, E)$. Κάθε διαμέριση $(S, V \setminus S)$ των κορυφών του G καλείται *τομή* του γραφήματος. Οι ακμές με ένα άκρο στο S και ένα στο $V \setminus S$ καλούνται *ακμές της τομής*. Τέλος, έστω $E' \subseteq E$, μία τομή *σέβεται* το E' αν καμία από τις ακμές του E' δεν είναι ακμή της τομής.

Θεώρημα 5.3.10 (Ιδιότητα της τομής)**.** Έστω συνεκτικό γράφημα $G = (V, E)$ και συνάρτηση βαρών $w : E \to \mathbb{N} \setminus \{0\}$. Αν $A \subseteq E$ σύνολο ακμών που ανήκουν σε ένα ελάχιστο επικαλύπτον δέντρο, $(S, V \setminus S)$ τομή που σέβεται το A και e η ακμή της τομής με το μικρότερο βάρος, τότε το σύνολο $A \cup \{e\}$ ανήκει σε ένα ελάχιστο επικαλύπτον δέντρο.

Απόδειξη. Έστω $T = (V, E')$ ελάχιστο επικαλύπτον δέντρο που περιέχει τις ακμές του A. Διακρίνουμε δύο περιπτώσεις:

- Η e είναι ακμή του T, τότε δεν έχουμε τίποτα να αποδείξουμε.
- Η e δεν είναι ακμή του T. Παρατηρήστε ότι το γράφημα $G[E' \cup \{e\}]$ δεν είναι δέντρο και πως η e δημιουργεί κύκλο. Επιπλέον, αφού το T είναι επικαλύπτον δέντρο πρέπει να υπάρχει ακμή $e \in E'$ που είναι ακμή της τομής $(S, V \setminus S)$ και μάλιστα συμμετέχει στον ίδιο κύκλο με την e. Αφού η τομή $(S, V \setminus S)$ σέβεται το A έπεται ότι $e' \notin A$,

[1] Για τα άκρα της e θα υπάρχουν δύο διακεκριμένα μονοπάτια, τα μονοπάτι που προϋπήρχε στο T και αυτό που αποτελείται μόνο από την ακμή e. Αυτά τα δύο μονοπάτια σχηματίζουν κύκλο.

οπότε το σύνολο $(E' \setminus \{e'\}) \cup \{e\}$ περιέχει τις ακμές του $A \cup \{e\}$. Θα δείξουμε ότι το $T' = G[(E' \setminus \{e'\}) \cup \{e\}]$ είναι ελάχιστο επικαλύπτον δέντρο.

Πρώτον το T' είναι επικαλύπτον δέντρο καθώς είναι συνεκτικό (η αφαίρεση της e' δεν πλήττει την συνεκτικότητα αφού είναι ακμή κύκλου), δεν περιέχει κύκλους και έχει $|V| - 1$ ακμές (ακριβώς όσες και το T).

Τέλος, αν $W(T)$ είναι το συνολικό βάρος του T και $W(T')$ του T', ισχύει ότι:

$$W(T') = W(T) - w(e') + w(e)$$

Όμως από την υπόθεση έχουμε ότι $w(e') \leq w(e)$, άρα $W(T') \leq W(T)$. Αυτό σημαίνει ότι και το T' έχει το ελάχιστο δυνατό συνολικό βάρος. □

Η ιδιότητα της τομής μας εξασφαλίζει ότι αν επιλέγουμε πάντα την ακμή με το ελάχιστο βάρος που δεν σχηματίζει κύκλο (συνεπώς αποτελεί την ελάχιστη ακμή μίας τομής $(S, V \setminus S)$ που σέβεται τις ακμές που έχουμε επιλέξει έως τώρα) θα οδηγηθούμε σε ένα ελάχιστο επικαλύπτον δέντρο. Το μόνο που απομένει είναι να βρούμε τρόπο να ελέγξουμε πότε μία ακμή δημιουργεί κύκλο.

Παρατηρήστε ότι αν $E' \subseteq E$ τέτοιο ώστε το $G[E']$ δεν περιέχει κύκλο τότε μία ακμή e δεν μπορεί να δημιουργεί κύκλο στο γράφημα $G[E' \cup \{e\}]$ αν κάποιο από τα άκρα της δεν ανήκει στο E'. Επομένως μία ασφαλής επιλογή θα ήταν μία ακμή με (τουλάχιστον ένα) άκρο που δεν είναι άκρο ακμής που έχουμε επιλέξει ήδη. Μια άλλη ασφαλής επιλογή θα ήταν μία ακμή που τα άκρα της ανήκουν σε διαφορετικές συνεκτικές συνιστώσες του $G[E']$. Ανάλογα με ποια από αυτές τις επιλογές θα υιοθετήσουμε μπορούμε να σχεδιάσουμε δύο διαφορετικούς αλγόριθμους, η ορθότητα των οποίων έπεται από την ιδιότητα της τομής.

5.3.1 Ο αλγόριθμος του Kruskal

Στον αλγόριθμο που θα δούμε θα διατηρούμε κάποιες συλλογές ακμών που δεν θα σχηματίζουν κύκλο –κάποια δέντρα δηλαδή– και θα προσθέτουμε μία ακμή (την ελαφρύτερη) μόνο αν τα άκρα της ανήκουν σε διαφορετικά δέντρα της συλλογής. Έτσι θα ενώνουμε συνεχώς τα δέντρα αυτής της συλλογής (ή θα δημιουργούμε καινούργια δέντρα που θα αποτελούνται από μία μόνο ακμή και στη συνέχεια θα ενωθούν με τα υπόλοιπα) μέχρι να φτάσουμε σε ένα επικαλύπτον δέντρο. Εφόσον επιλέγουμε πάντα την ακμή με το μικρότερο βάρος που δεν σχηματίζει κύκλο, άρα τα άκρα της θα ανήκουν σε διαφορετικές συνεκτικές συνιστώσες, (από την ιδιότητα της τομής) το δέντρο στο οποίο θα καταλήξουμε θα έχει το ελάχιστο δυνατό συνολικό βάρος.

Για να υλοποιήσουμε αυτήν την ιδέα θα χρησιμοποιήσουμε ένα σύνολο για κάθε δέντρο που θα περιέχει της κορυφές του δέντρου, θα επιλέγουμε την ελαφρύτερη ακμή που τα άκρα της ανήκουν σε διαφορετικά σύνολα και έπειτα θα τα ενώνουμε. Θα χρησιμοποιήσουμε τον ακόλουθο συμβολισμό:

- Makeset(u): Δημιουργία του μονοσυνόλου $\{u\}$
- Find(u): Εύρεση του συνόλου που ανήκει το στοιχείο u (εύρεση αντιπροσώπου δηλαδή)

- Union(u, v): Ένωση των συνόλων που ανήκουν τα στοιχεία u, v

Τέλος, θα θεωρήσουμε ότι αντί για τα βάρη των ακμών μας δίνεται μία ταξινομημένη ουρά με στοιχεία τις ακμές κατά αύξον βάρος [1].

Ο αλγόριθμος που ακολουθεί ανακαλύφθηκε από τον Joseph Kruskal το 1956.

Kruskal(G, Q)

Είσοδος: Γράφημα $G = (V, E)$ και ταξινομημένη ουρά Q με τις ακμές του E κατά αύξον βάρος

Έξοδος : Ουρά Q' με τις ακμές του ελάχιστου επικαλύπτον δέντρου

```
1 Q' ← new queue
2 for u ∈ V
3     Makeset(u)
4 while not IsEmpty(Q)
5     {u, v} ← Dequeue(Q)
6     if Find(u) ≠ Find(v) then
7         Enqueue({u, v}, Q')
8         Union(u, v)
9 return Q'
```

Παρατηρήστε ότι, σύμφωνα με όσα είπαμε παραπάνω, θα μπορούσαμε να σταματήσουμε όταν έχουμε επιλέξει $|V| - 1$ ακμές [2].

Ας δούμε τον χρόνο που χρειάζεται αλγόριθμος:

- Το βήμα 1 χρειάζεται χρόνο $O(1)$.
- Στα βήματα 2-3 θα γίνουν $|V|$ δημιουργίες μονοσυνόλων.
- Στο while θα γίνουν (στη χειρότερη περίπτωση) $|E|$ έλεγχοι κενότητας, εξαγωγές και εισαγωγές σε ουρά, $|E|$ ενώσεις συνόλων και $O(|E|)$ ευρέσεις αντιπροσώπων.
- Τέλος, το βήμα 9 χρειάζεται χρόνο $O(1)$.

Συγκεντρωτικά έχουμε $|E|$ πράξεις ουράς, με συνολικό χρόνο $O(|E|)$, και μια ακολουθία από $O(|E|)$ πράξεις συνόλων με $|V| \leq |E| - 1$ δημιουργίες συνόλων (καθώς το γράφημα είναι συνεκτικό). Αν υποθέσουμε ότι έχουμε χρησιμοποιήσει την αναπαράσταση ξένων συνόλων με δέντρα, τότε από την Πρόταση 4.7.3 η ακολουθία χρειάζεται χρόνο $O(|E| \cdot \alpha(|V|))$ (όπου α μία συνάρτηση που αυξάνει πάρα πολύ αργά [3]). Επομένως, συνολικά ο χρόνος που χρειάζεται ο Kruskal είναι $O(|E| \cdot \alpha(|V|))$.

[1] Παρατηρήστε ότι στην πραγματικότητα δεν μας χρειάζεται να γνωρίζουμε το ακριβές βάρος της κάθε ακμής. Το μόνο που μας χρειάζεται είναι να τις ελέγχουμε από την ελαφρύτερη προς τη βαρύτερη.

[2] Στον Kruskal ελέγχουμε και τις υπόλοιπες ακμές της ουράς οι οποίες προφανώς δημιουργούν κύκλο καθώς οι ακμές στην Q' σχηματίζουν επικαλύπτον δέντρο (άρα όλες οι κορυφές του γραφήματος ανήκουν στο ίδιο σύνολο).

[3] Θα μπορούσαμε να υποθέσουμε με ασφάλεια ότι $\alpha(n) = O(\log n)$.

Δεν έχουμε αναφερθεί καθόλου στον χρόνο που θα χρειαστούμε να δημιουργήσουμε την ουρά Q της εισόδου, να ταξινομήσουμε δηλαδή στην ουσία τα στοιχεία του E. Όπως θα δούμε στο κεφάλαιο που ακολουθεί, αυτό μπορεί αν γίνει σε χρόνο $O(|E|\log|E|)$. Θυμηθείτε όμως ότι $|E| = O(|V|^2)$, οπότε $\log|E| = O(\log|V|)$. Τελικά μπορούμε να θεωρήσουμε ότι ο παραπάνω αλγόριθμος εύρεσης ενός ελάχιστου επικαλύπτον δέντρου χρειάζεται χρόνο $O(|E|\log|V|)$ ή $O(|V|^2\log|V|)$.

5.3.2 Ο αλγόριθμος του Prim

Μια παραλλαγή του αλγορίθμου του Kruskal είναι να φροντίζουμε οι ακμές που θα έχουμε επιλέξει την κάθε δεδομένη χρονική στιγμή να σχηματίζουν ένα μοναδικό δέντρο. Για να το πετύχουμε αυτό θα επιλέγουμε πάντα την ελαφρύτερη ακμή που θα «επεκτείνει» αυτό το δέντρο. Έτσι η ιδιότητα της τομής θα μας εξασφαλίζει ότι οι «άπληστες» επιλογές μας θα οδηγήσουν τελικά σε ένα ελάχιστο επικαλύπτον δέντρο. Το ζητούμενο είναι πως θα γνωρίζουμε ποια ακμή θα πρέπει να προσθέσουμε στο δέντρο.

Αντί να επικεντρωνόμαστε στην εύρεση αυτής της ακμής μπορούμε να ψάξουμε να βρούμε κατάλληλη κορυφή για να προσθέσουμε στο δέντρο (το άλλο άκρο μίας ακμής που επεκτείνει το δέντρο δηλαδή). Ας υποθέσουμε ότι έχουμε επιλέξει τις κορυφές του $V' \subseteq V$. Για κάθε κορυφή $u \in V \setminus V'$ θεωρούμε την ποσότητα:

$$cost(u) = \min\{w(\{u,v\}) \mid \{u,v\} \in E \text{ και } v \in V'\}$$

Η κορυφή που θα πρέπει να εισάγουμε είναι αυτή με την ελάχιστη τιμή $cost$, καθώς τότε η ακμή $\{u,v\}$ θα είναι η ελαφρύτερη της τομής $(V', V \setminus V')$.

Το πρόβλημα που θα πρέπει να αντιμετωπίσουμε για να υλοποιήσουμε αυτήν την ιδέα είναι ότι οι τιμές $cost$ αλλάζουν ανάλογα με το ποια κορυφή θα προσθέσουμε στο V'. Επομένως δεν μπορούμε να υποθέσουμε ότι μας δίνεται π.χ. μία ταξινομημένη ουρά (όπως στον Kruskal) με τις κορυφές κατά αύξουσα τιμή $cost$. Θα πρέπει να αναθεωρούμε αυτές τις τιμές κατά τη διάρκεια του αλγορίθμου.

Θα χρησιμοποιήσουμε μια ουρά προτεραιότητας με βαθμούς προτεραιότητας τις τιμές $cost$. Οι αρχικές τιμές για τις κορυφές θα είναι NIL (ή κάποια άλλη «παράλογη» τιμή). Αυτό σημαίνει ότι στην ουσία δεν υπάρχει ακμή που συνδέει κάποια κορυφή με τον δέντρο (πράγμα αληθές καθώς το δέντρο δεν περιέχει καμία κορυφή). Έπειτα, θα προσθέτουμε στο δέντρο την κορυφή που θα εξάγουμε από την ουρά προτεραιότητας, δηλαδή αυτή με το ελάχιστο $cost$. Μετά την προσθήκη της κορυφής όμως θα πρέπει να ενημερώνουμε το $cost$ των γειτονικών της κορυφών έξω από το δέντρο, μειώνοντας το όποτε κριθεί απαραίτητο (θεωρούμε ότι η τιμή NIL είναι μεγαλύτερη από οποιονδήποτε βαθμό προτεραιότητας). Ο λόγος που πρέπει να γίνει αυτό είναι γιατί με την καινούργια κορυφή στο δέντρο οι ακμές της τομής έχουν αλλάξει (έχουν αφαιρεθεί κάποιες και έχουν προστεθεί κάποιες άλλες).

Στα προηγούμενα κεφάλαια δεν είδαμε πως μπορούμε να ενημερώσουμε τον βαθμό προτεραιότητας ενός στοιχείου μίας ουράς προτεραιότητας, αλλά εύκολα κάποιος μπορεί να σχεδιάσει έναν αλγόριθμο που υλοποιεί αυτήν την πράξη [1]. Έστω Update(u,p,Q) ο αλγό-

[1] Να μια πολύ καλή άσκηση!

ριθμος, όπου u το στοιχείο, p ο νέος βαθμός προτεραιότητας και Q η ουρά. Αν χρησιμοποιήσουμε σωρούς για την αναπαράσταση των ουρών προτεραιότητας ο Update μπορεί να υλοποιηθεί με χρόνο $O(\log n)$, όπου n το πλήθος στοιχείων της ουράς. Για τις υπόλοιπες πράξεις θα χρησιμοποιήσουμε τις εξής εντολές:

- **new priority queue**: Δημιουργία ουράς προτεραιότητας.
- Enqueue(u, p, Q): Εισαγωγή του στοιχείου u στην Q με βαθμό p.
- Dequeue(Q): Εξαγωγή στοιχείου από την Q.

Στην αναπαράσταση με σωρό η δημιουργία χρειάζεται σταθερό χρόνο ενώ οι εισαγωγή και η εξαγωγή στοιχείου χρόνο $O(\log n)$, όπου n το πλήθος στοιχείων.

Είμαστε έτοιμοι να δούμε τον Αλγόριθμο του Prim [1]. Η έξοδος του αλγορίθμου θα είναι πάλι το «σύνολο» ακμών ενός ελάχιστου επικαλύπτον δέντρου, θα είναι εκφρασμένο βέβαια με διαφορετικό τρόπο από όσα έχουμε δει έως τώρα. Θα δίνεται με τη μορφή πίνακα όπου κάθε «κελί» θα αντιστοιχεί στο ένα άκρο της ακμής και το περιεχόμενό του στο άλλο άκρο.

Prim(G, W)

Είσοδος: Γράφημα $G = (V, E)$ και πίνακας δισδιάστατος W με τα βάρη των ακμών
Έξοδος : Πίνακας $Prev$ με τις ακμές ενός ελάχιστου επικαλύπτον δέντρου (αν $Prev[u] = v$ τότε η ακμή $\{u, v\}$ ανήκει στο δέντρο)

$Q' \leftarrow$ **new queue**
$Cost \leftarrow$ **new matrix**$(|V|)$
$Prev \leftarrow$ **new matrix**$(|V|)$
for $u \in V$
    $Cost[u] \leftarrow$ NIL
    $Prev[u] \leftarrow$ NIL
choose $u_0 \in V$ *% Επιλέγουμε μία κορυφή για να ξεκινήσουμε*
$Cost[u_0] \leftarrow 0$
$Q \leftarrow$ **new priority queue**
for $u \in V$
    Enqueue$(u, Cost[u], Q)$
while not IsEmpty(Q)
    $x \leftarrow$ Dequeue(Q)
    for $y \in Adj[x]$
        if $y \in Q$ **and** $Cost[y] > W[x, y]$ **then**
            $Cost[y] \leftarrow W[x, y]$
            Update$(y, Cost[y], Q)$
            $Prev[y] \leftarrow x$
return $Prev$

[1] Ο αλγόριθμος αυτός έχει «ανακαλυφθεί» τρεις φορές: το 1930 από τον Vojtěch Jarník, το 1957 από τον Robert C. Prim και το 1959 από τον Edsger W. Dijkstra. Στη βιβλιογραφία συνήθως φέρει το όνομα του Prim.

Η ορθότητα του αλγορίθμου πηγάζει από την ιδιότητα της τομής. Το μόνο που θέλει αιτιολόγηση είναι ότι η αυθαίρετη επιλογή της κορυφής u_0 στο βήμα 7, και κατ' επέκταση η επιλογή της ελαφρύτερης ακμής με άκρο την u_0, θα μας οδηγήσει σε ελάχιστο επικαλύπτον δέντρο.

Παρατήρηση 5.3.11. Έστω γράφημα $G = (V, E)$ και συνάρτηση βαρών $w : E \to \mathbb{N} \setminus \{0\}$. Έστω $u \in V$ και $e \in E$ η ακμή με το μικρότερο βάρος που έχει σαν άκρο την u. Υπάρχει ελάχιστον επικαλύπτον δέντρο που περιέχει την e.

Πράγματι αν προσθέσουμε την e σε ένα ελάχιστο επικαλύπτον δέντρο T θα δημιουργηθεί κύκλος. Αυτός ο κύκλος θα περιέχει ακόμα μία ακμή με άκρο την u_0, έστω την e'. Αν αφαιρέσουμε την e' και προσθέσουμε την e στο T θα πάρουμε επικαλύπτον δέντρο με μικρότερο (είτε ίσο) συνολικό βάρος.

Για τον χρόνο του αλγορίθμου παρατηρούμε ότι τα βήματα 1-11 χρειάζονται χρόνο της τάξης του $O(|V| \log |V|)$. Μέσα στο while θα εκτελεσθούν $|V|$ εξαγωγές και $|E|$ ενημερώσεις βαθμού προτεραιότητας (όσα δηλαδή και τα συνολικά στοιχεία στις λίστες γειτνίασης) [1]. Οπότε ο Prim χρειάζεται χρόνο $O(|V| \log |V| + |E| \log |V|)$ άρα (εφόσον το γράφημα έχει υποτεθεί συνεκτικό) $O(|E| \log |V|)$.

Ασκήσεις

5.1. Δώστε αλγόριθμους που υλοποιούν τις βασικές λειτουργίες των γραφημάτων, για τους τρεις τρόπους αναπαράστασης που είδαμε (δες τον πίνακα στη Σελίδα 146).

5.2. Δώστε αλγόριθμο που δέχεται σαν είσοδο ένα (μη-κατευθυνόμενο) γράφημα $G = (V, E)$ και μία ακμή $e \in E$, και ελέγχει αν το G περιέχει κύκλο που περνάει από την e. Ο χρόνος του αλγορίθμου θα πρέπει να είναι $O(|V| + |E|)$.

5.3. Στην επαγγελματική πάλη οι παλαιστές συνήθως χωρίζονται σε δύο κατηγορίες, στους «καλούς» και στους «κακούς». Χάριν θεάματος καλλιεργείται αντιπαλότητα μεταξύ δύο παλαιστών, η οποία αναπόφευκτα θα οδηγήσει σε μάχη. Οι μάχες με την υψηλότερη τηλεθέαση είναι οι μάχες μεταξύ ενός καλού και ενός κακού παλαιστή. Το πρόβλημα που καλούμαστε να λύσουμε είναι το εξής: Μας δίνεται ένα σύνολο από n παλαιστές και μία λίστα με m ζευγάρια παλαιστών οι οποίοι είναι αντίπαλοι μεταξύ τους (κάθε παλαιστής είναι αντίπαλος με τουλάχιστον έναν παλαιστή). Σκοπός μας είναι να ελέγξουμε αν μπορούμε να χωρίσουμε τους παλαιστές σε καλούς και κακούς έτσι ώστε να μην υπάρχει αντιπαλότητα μεταξύ δύο καλών ή δύο κακών παλαιστών.

1. Ορίστε τυπικά το πρόβλημα αυτό χρησιμοποιώντας ένα γράφημα.
2. Χρησιμοποιήστε τον αλγόριθμο BFS-Eplore για να λύσετε το πρόβλημα σε χρόνο $O(n + m)$.

[1] Όσον αφορά τον έλεγχο για το αν $y \in Q$ μπορούμε να υποθέσουμε ότι γίνεται σε σταθερό χρόνο αποθηκεύοντας επιπλέον πληροφορία στις κορυφές.

5.4. Δείξτε ότι το ελάχιστο επικαλύπτον δέντρο είναι πάντα μοναδικό αν τα βάρη των ακμών είναι διακεκριμένα.

5.5. Ένα (μη-κατευθυνόμενο) γράφημα $G = (V, E)$ καλείται *διμερές* αν υπάρχει διαμέριση V_1, V_2 του V έτσι ώστε κάθε ακμή $e \in E$ να έχει το ένα άκρο στο V_1 και το άλλο στο V_2.

1. Δείξτε ότι ένα γράφημα $G = (V, E)$, με $|E| \geq 1$, είναι διμερές αν και μόνο αν μπορούμε να αναθέσουμε στις κορυφές του V δύο χρώματα έτσι ώστε τα άκρα κάθε ακμής του E να έχουν διαφορετικό χρώμα.

2. Δώστε αλγόριθμο IsBipartite(G) που αποφασίζει αν ένα γράφημα G είναι διμερές ή όχι. Ο αλγόριθμος θα πρέπει να έχει χρόνο $O(|V| + |E|)$.

ΚΕΦΑΛΑΙΟ 6

ΑΛΓΟΡΙΘΜΟΙ ΤΑΞΙΝΟΜΗΣΗΣ

Το τελευταίο που θα μελετήσουμε σε αυτές τις σημειώσεις είναι οι αλγόριθμοι ταξινόμησης. Το πρόβλημα της ταξινόμησης μπορεί να περιγραφεί ως εξής:

> *Μας δίνεται μία ακολουθία από στοιχεία και κάποια σχέση διάταξης που τα διέπει. Το ζητούμενο είναι να επιστραφεί μία ακολουθία με τα στοιχεία ταξινομημένα ως προς αυτήν τη σχέση (π.χ. κατά αύξουσα σειρά).*

Η αξία αυτών των αλγορίθμων δεν χρειάζεται να αναλυθεί περαιτέρω (ήδη σιωπηρά τους έχουμε χρησιμοποιήσει). Σε πάρα πολλές εφαρμογές το πρώτο στάδιο επίλυσης του προβλήματος είναι η ταξινόμηση των δεδομένων. Είναι απολύτως αναγκαίο το στάδιο αυτό να διεκπεραιώνεται γρήγορα, καθώς σε αντίθετη περίπτωση υπάρχει ο κίνδυνος ο χρόνος που θα χρειαστούμε για να ταξινομήσουμε τα δεδομένα να «καπελώνει» τον χρόνο που θα χρειαστούμε για να ολοκληρώσουμε τις υπόλοιπες διεργασίες. Πάρτε για παράδειγμα τον αλγόριθμο του Kruskal. Φανταστείτε για κάποιο γράφημα $G = (V, E)$ να χρειαζόμασταν για την ταξινόμηση των ακμών, κατά αύξοντα αριθμό βάρους, χρόνο $O(|E|^2)$!

Θα ξεκινήσουμε βλέποντας κάποιους απλούς αλγορίθμους που βασίζονται πάνω στη σύγκριση τιμών και στην τοποθέτηση τους σε κατάλληλη θέση.΄Επειτα θα περάσουμε σε πιο σύνθετους αλγόριθμους. Σε όσα ακολουθούν για λόγους ευκολίας θα υποθέσουμε τα εξής:

- Τα στοιχεία μας είναι φυσικοί αριθμοί και δεν έχουμε επαναλήψεις στοιχείων.
- Τα στοιχεία είναι αποθηκευμένα σε έναν μονοδιάστατο πίνακα.
- Στόχος μας είναι να ταξινομήσουμε τα στοιχεία σε αύξουσα σειρά.

6.1 Ταξινόμηση σε τετραγωνικό χρόνο

Θα ξεκινήσουμε βλέποντας απλούς αλγορίθμους ταξινόμησης που υλοποιούν βασικές ιδέες όπως η *παρεμβολή* στοιχείων στη σωστή θέση, η *αντιμετάθεση* στοιχείων και η επαναληπτική *επιλογή* του μικρότερου στοιχείου. Οι αλγόριθμοι αυτοί είναι πολύ εύκολοι στην κατανόηση, δυστυχώς όμως δεν είναι οι βέλτιστοι δυνατοί όσον αφορά τον χρόνο που χρειάζονται: Ο χρόνος τους είναι τετραγωνικός ως προς το μέγεθος του πίνακα. Στην επόμενη παράγραφο θα δούμε ότι μπορούμε να καταφέρουμε ακόμα καλύτερους χρόνους [1].

6.1.1 Ταξινόμηση παρεμβολής

Φανταστείτε ότι έχουμε καταφέρει να ταξινομήσουμε τα πρώτα i στοιχεία του A. Για να επεκτείνουμε τη λίστα των ταξινομημένων στοιχείων αποφασίζουμε να εντάξουμε και το στοιχείο $A[i+1]$. Σκοπός μας φυσικά είναι να επεκτείνουμε αυτήν τη λίστα ώστε να καταλαμβάνει ολόκληρο τον A. Για να ταξινομήσουμε και το $A[i+1]$ το μόνο που έχουμε να κάνουμε είναι να το τοποθετήσουμε στη σωστή σειρά ανάμεσα στα πρώτα i στοιχεία. Ο πιο απλός τρόπος να το πετύχουμε αυτό είναι κάνοντας γραμμική αναζήτηση μεταξύ των στοιχείων $A[1], \ldots, A[i]$ μέχρι να βρούμε στοιχείο $A[k]$ τέτοιο ώστε $A[i+1] < A[k]$. Τότε η θέση του $A[i+1]$ είναι μεταξύ των στοιχείων $A[k-1]$ και $A[k]$ [2], για να παρεμβάλουμε όμως το $A[i+1]$ σε αυτήν τη θέση θα πρέπει πρώτα να «μετακινήσουμε» όλα τα στοιχεία από το $A[k]$ ως και το $A[i]$ μία θέση δεξιά.

Στην εισαγωγή των σημειώσεων είδαμε πως μπορούμε να κάνουμε γραμμική αναζήτηση (σελ. 13). Θα τροποποιήσουμε τον αλγόριθμο LinearSearch έτσι ώστε να επεξεργάζεται ένα αρχικό τμήμα του πίνακα και τελικά να επιστρέφει τη θέση που βρίσκεται το *αμέσως μεγαλύτερο* στοιχείο από το στοιχείο εισόδου.

LinearSearch(A, i, x)

Είσοδος: Πίνακας A, θέση i του πίνακα [3] και αριθμός x
Έξοδος : Η θέση του πρώτου στοιχείου που είναι μεγαλύτερο από x στον υποπίνακα $A[1, \ldots, i]$ (αν όλα τα στοιχεία έχουν τιμή μικρότερη από x τότε επιστρέφει $i+1$)

```
k ← 1
while k ≤ i and A[k] < x
    k ← k + 1
return k
```

Ο αλγόριθμος που υλοποιεί την *ταξινόμηση παρεμβολής*, όπως αποκαλείται συχνά στη βιβλιογραφία, είναι ο ακόλουθος:

InsertionSort(A)

Είσοδος: Πίνακας A
Έξοδος : Τίποτα (εσωτερικά τα στοιχεία του πίνακα A έχουν ταξινομηθεί)

[1] Έχουμε ήδη δει στο Παράδειγμα 1.1.1 έναν αλγόριθμο που ταξινομεί τον πίνακα σε γραμμικό χρόνο (υπό προϋποθέσεις βέβαια).
[2] Φυσικά αν $k = 1$ τότε το $A[i+1]$ θα πρέπει να τοποθετηθεί στην αρχή του πίνακα.
[3] Υποθέτουμε ότι $i \leq \textbf{length}(A)$.

```
1 for i ← 2 to length(A)
2     k ← LinearSearch(A, i − 1, A[i])
3     tmp ← A[i]
4     for j ← 0 to i − k − 1                 %Αν k > i − 1 τότε δεν μπαίνουμε στο for
5         A[i − j] ← A[i − j − 1]
6     A[k] ← tmp
```

Ας δούμε πόσο χρόνο χρειάζεται ο αλγόριθμος αυτός για να ταξινομήσει τον πίνακα. Ας υποθέσουμε ότι ο πίνακας A έχει n στοιχεία. Για κάθε $i \in \{2, \ldots, n\}$ θα γίνουν ακριβώς $i-1$ «βήματα» σταθερού χρόνου (k βήματα μέσα στον LinearSearch και $i-k-1$ στο δεύτερο for). Συνεπώς ο χρόνος του αλγορίθμου είναι:

$$\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2)$$

6.1.2 Δυαδική ταξινόμηση παρεμβολής

Μια προφανής βελτίωση στο χρόνο του InsertionSort θα ήταν να βρίσκαμε πιο γρήγορα τη θέση που θα πρέπει να πάρει το στοιχείου $A[i]$ στον υποπίνακα $A[1, \ldots, i-1]$. Θα μπορούσαμε αντί για γραμμική αναζήτηση να κάνουμε δυαδική αναζήτηση, εκμεταλλευόμενοι το γεγονός ότι ο υποπίνακας είναι ταξινομημένος. Θα χρειαστεί να τροποποιήσουμε με αντίστοιχο τρόπο τον αλγόριθμο BinarySearch που είδαμε στην εισαγωγή (σελ. 17):

BinarySearch(A, i, x)

Είσοδος: Ταξινομημένος πίνακας A, θέση i του πίνακα και αριθμός x
Έξοδος : Η θέση του πρώτου στοιχείου που είναι μεγαλύτερο από x στον υποπίνακα $A[1, \ldots, i]$ (αν όλα τα στοιχεία έχουν τιμή μικρότερη από x τότε επιστρέφει $i+1$)

```
l ← 1, r ← i
k ← ⌊(l+r)/2⌋
while l ≤ r
    if A[k] < x then
        l ← k + 1
    else
        r ← k − 1
    k ← ⌊(l+r)/2⌋
return k + 1
```

Αν αντικαταστήσουμε στη γραμμή 2 του InsertionSort τον LinearSearch με τον παραπάνω αλγόριθμο θα έχουμε σημαντική βελτίωση στον *μέσο* χρόνο του αλγορίθμου, καθώς αντί για $O(i)$ βήματα για την εύρεση του k θα χρειαστεί να κάνουμε μόνο $O(\log i)$. Όσον αφορά όμως τη *χειρότερη δυνατή είσοδο* για αυτήν την εκδοχή του InsertionSort, όταν δηλαδή τα στοιχεία του A είναι σε αντίστροφη σειρά, τότε πάλι θα χρειαστούμε χρόνο $O(n^2)$: Για κάθε $i = 2, \ldots n-1$ το k που θα επιστρέφει ο BinarySearch θα είναι ίσο με 1 [1]. Οπότε θα γίνουν $1 + 2 + \cdots + (n-1)$ μετατοπίσεις στοιχείων προς τα δεξιά.

6.1.3 Ταξινόμηση φυσαλίδας

Θα ταξινομήσουμε τον πίνακα κάνοντας διαδοχικές διαπεράσεις συγκρίνοντας τα γειτονικά στοιχεία και αντιμεταθέτοντάς τα όποτε χρειάζεται. Αυτό θα έχει ως αποτέλεσμα στην πρώτη διαπέραση το μεγαλύτερο στοιχείο να τοποθετηθεί στην τελευταία θέση του πίνακα, στη δεύτερη διαπέραση το δεύτερο μεγαλύτερο στοιχείο στην προτελευταία θέση του πίνακα και κ.ο.κ. [2].

BubbleSort(A)

Είσοδος: Πίνακας A
Έξοδος : Τίποτα (εσωτερικά τα στοιχεία του πίνακα A έχουν ταξινομηθεί)

```
for i ← 1 to length(A) − 1
    for j ← 1 to length(A) − 1
        if A[j] > A[j + 1] then
            tmp ← A[j]
            A[j] ← A[j + 1]
            A[j + 1] ← tmp
```

Είναι εμφανές ότι ο χρόνος του BubbleSort είναι $O(n^2)$, όπου n το πλήθος στοιχείων του πίνακα. Μπορούμε να βελτιώσουμε τον χρόνο του αλγορίθμου με δύο τρόπους:

- Δεν χρειάζεται σε κάθε διαπέραση να πηγαίνουμε μέχρι το τέλος του πίνακα, καθώς γνωρίζουμε π.χ. ότι μετά την πρώτη διαπέραση στην τελευταία θέση του πίνακα θα είναι το μεγαλύτερο στοιχείο του πίνακα (και προφανώς σύγκριση του προηγούμενου στοιχείου με αυτό το στοιχείο δεν έχει νόημα).
- Αν σε κάποια διαπέραση δεν χρειαστεί να αντιμεταθέσουμε κάποια στοιχεία τότε ο πίνακας έχει ήδη ταξινομηθεί.

Η βελτίωση στον χρόνο όμως δεν είναι ουσιαστική καθώς και πάλι θα χρειαστεί να κάνουμε $O(n^2)$ συγκρίσεις (στη χειρότερη περίπτωση [3]).

6.1.4 Ταξινόμηση επιλογής

Θα μπορούσαμε κάνοντας μία διαπέραση να επιλέξουμε το μικρότερο στοιχείο του πίνακα και έπειτα να το τοποθετήσουμε στην πρώτη θέση του. Στην επόμενη διαπέραση (που

[1] Παρατηρήστε ότι για αυτήν την είσοδο η γραμμική αναζήτηση του k είναι πολύ πιο αποδοτική.

[2] Το όνομα του αλγορίθμου πηγάζει από το γεγονός ότι τα στοιχεία από το μεγαλύτερο προς μικρότερο θα «αναδύονται» προς το τέλος του πίνακα.

[3] Αν ο πίνακας είναι ταξινομημένος αντίστροφα θα γίνουν $(n-1) + (n-2) + \cdots + 1$ αντιμεταθέσεις στοιχείων.

θα ξεκινήσει από τη δεύτερη θέση του πίνακα) θα βρούμε το δεύτερο μικρότερο στοιχείο και θα το βάλουμε στη δεύτερη θέση. Συνεχίζοντας με τον ίδιο τρόπο θα έχουμε ταξινομήσει τον πίνακα:

SelectionSort(A)

Είσοδος: Πίνακας A
Έξοδος : Τίποτα (εσωτερικά τα στοιχεία του πίνακα A έχουν ταξινομηθεί)

```
for i ← 1 to length(A)
    min ← i
    for j ← i + 1 to length(A)
        if A[j] < A[min] then
            min ← j
    tmp ← A[i]
    A[i] ← A[min]
    A[min] ← tmp
```

Αν $min = i$ προφανώς δεν χρειάζεται να αντιμεταθέσουμε τα $A[min]$ και $A[i]$. Ο χρόνος του αλγορίθμου είναι $O(n^2)$ καθώς το πλήθος συγκρίσεων που θα χρειαστεί είναι:

$$(n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = O(n^2)$$

Μπορεί να αποδειχθεί ότι το ελάχιστο πλήθος συγκρίσεων που θα χρειαστούμε για να ταξινομήσουμε έναν πίνακα με n στοιχεία είναι $O(n \log n)$. Στη συνέχεια θα δούμε έναν αλγόριθμο που χρειάζεται ακριβώς τόσες συγκρίσεις.

6.2 Ταξινόμηση σωρού

Οι αλγόριθμοι που είδαμε μέχρι τώρα δεν είναι λιγότερο αποτελεσματικοί μόνο επειδή (ενδεχομένως) να κάνουν παραπάνω συγκρίσεις από τις απαραίτητες. Σε μερικές περιπτώσεις αλλάζουν θέση σε ένα στοιχείο πάρα πολλές φορές (π.χ. στον SelectionSort με είσοδο τον πίνακα $A = [n, 1, 2, 3, \dots, n-1]$ το στοιχείο n θα μεταφερθεί $n-1$ φορές!). Ιδανικά θα θέλαμε να κάνουμε τις απαραίτητες συγκρίσεις και έπειτα να τοποθετήσουμε τα στοιχεία *απευθείας* στη σωστή τους θέση. Ένας απλός τρόπος να το επιτύχουμε αυτό είναι να χρησιμοποιήσουμε κάποια βοηθητική δομή δεδομένων για να αποθηκεύουμε τα στοιχεία στην κατάλληλη θέση, όσο εμείς διεκπεραιώνουμε τις συγκρίσεις, και έπειτα να τα αντιγράψουμε πίσω στον πίνακα. Θα μπορούσαμε επίσης να επιλέξουμε μία δομή που θα μας εξυπηρετήσει και στις δύο εργασίες που έχουμε αναλάβει.

Θυμηθείτε ότι σε έναν σωρό ελαχίστου το μικρότερο στοιχείο βρίσκεται πάντα στη ρίζα του σωρού. Θα μπορούσαμε λοιπόν να εισάγουμε τα στοιχεία του πίνακα σε έναν σωρό και έπειτα να κάνουμε μια σειρά εξαγωγών, τοποθετώντας το πρώτο στοιχείο που θα εξάγουμε στην πρώτη θέση του πίνακα, το δεύτερο στη δεύτερη κ.ο.κ..

HeapSort(A)

Είσοδος: Πίνακας A
Έξοδος : Τίποτα (εσωτερικά τα στοιχεία του πίνακα A έχουν ταξινομηθεί)

$H \leftarrow$ **new heap**
for $i \leftarrow 1$ **to length**(A)
  Insert($A[i], H$)
for $i \leftarrow 1$ **to length**(A)
  $A[i] \leftarrow$ Deletion(H)

Σε κάθε φάση του αλγορίθμου ο σωρός H θα περιέχει το πολύ n στοιχεία, πράγμα που σημαίνει ότι το ύψος του θα είναι $O(\log n)$. Ο HeapSort θα χρειαστεί να κάνει n εισαγωγές στον σωρό και n εξαγωγές από αυτόν, επομένως ο συνολικός του χρόνος θα είναι $O(n \log n)$.

Παρόλο που καταφέραμε να ταξινομήσουμε τον πίνακα πιο αποδοτικά (όσον αφορά τον χρόνο), σπαταλήσαμε όμως περισσότερο χώρο απ' ότι χρειαζόταν (τουλάχιστον τον διπλάσιο απ' τους αλγόριθμους που είδαμε μέχρι τώρα). Αυτό σε μερικές εφαρμογές μπορεί να είναι απαγορευτικό. Εδώ εντοπίζεται ένα πολύ καλό παράδειγμα της χρησιμοποίησης σε έναν αλγόριθμο χώρου ως αντάλλαγμα για χρόνο.

6.3 Ταξινόμηση με συγχώνευση

Στον επόμενο αλγόριθμο που θα δούμε θα εφαρμόσουμε μία πολύ ισχυρή τεχνική σχεδίασης αλγορίθμων το *διαίρει και κυρίευε:* Θα «σπάμε» τον πίνακα αναδρομικά στη μέση, δημιουργώντας δύο υποπίνακες, έπειτα τέσσερις κ.ο.κ. μέχρι να φτάσουμε σε πίνακες μεγέθους 1 (όπου προφανώς θα είναι ταξινομημένοι). Έπειτα θα συγχωνεύουμε κατάλληλα τους (ταξινομημένους) υποπίνακες ώστε το αποτέλεσμα της συγχώνευσης να είναι ταξινομημένος πίνακας, δημιουργώντας στην αρχή πίνακες μεγέθους 2, μετά 4 κ.ο.κ., και καταλήγοντας τελικά στον αρχικό μας πίνακα.

Θα αφήσουμε στην άκρη προσωρινά τον αλγόριθμο που υλοποιεί τη συγχώνευση (ο Merge στον αλγόριθμο που ακολουθεί) και θα δούμε τον συνολικό αλγόριθμο.

MergeSort(A, l, r)

Είσοδος: Πίνακας A, και θέσεις l, r του πίνακα με $l \leq r$
Έξοδος : Ο πίνακας A με τα στοιχεία στις θέσεις από l μέχρι και r ταξινομημένα

if $l < r$ **then**
  $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
  return Merge(MergeSort(A, l, m), MergeSort($A, m+1, r$))
else
  return A

Προφανώς για να ταξινομήσουμε ολόκληρο τον A θα τον τρέξουμε με l ίσο με ένα και r ίσο με το μέγεθος του πίνακα.

Ας σκεφτούμε πως μπορούμε να συγχωνεύσουμε δύο ταξινομημένους πίνακες A, B σε έναν ταξινομημένο πίνακα C. Το πρώτο στοιχείο του C θα είναι το μικρότερο από τα στοιχεία $A[1], B[1]$. Ας υποθέσουμε ότι $A[1] < B[1]$, τότε το δεύτερο στοιχείο του C θα είναι το μικρότερο από τα στοιχεία $A[2], B[1]$. Επαναλαμβάνοντας αυτήν την ιδέα θα καταλήξουμε σε έναν ταξινομημένο πίνακα που περιέχει τα στοιχεία και των δύο πινάκων.

```
Merge(A, B)

Είσοδος: Ταξινομημένοι πίνακες A, B
Έξοδος : Ταξινομημένος πίνακας C με στοιχεία τα στοιχεία των A, B

C ← new matrix(length(A)+ length(B))
k ← 1, i ← 1, j ← 1
while i ≤ length(A) and j ≤ length(B)
    if A[i] < B[j] then
        C[k] ← A[i]
        i ← i + 1
    else
        C[k] ← B[j]
        j ← j + 1
    k ← k + 1
while i ≤ length(A)                    % Αν έχουν περισσέψει στοιχεία στον A
    C[k] ← A[i]
    i ← i + 1
    k ← k + 1
while j ≤ length(B)                    % Αν έχουν περισσέψει στοιχεία στον B
    C[k] ← B[j]
    j ← j + 1
    k ← k + 1
return C
```

Εύκολα παρατηρούμε ότι ο χρόνος του Merge είναι $O(k + l)$ όπου k, l τα μεγέθη των πινάκων A, B αντίστοιχα. Ας υποθέσουμε ότι η συνάρτηση χρονικής πολυπλοκότητας του MergeSort είναι $T(n)$. Η συνάρτηση αυτή ικανοποιεί την εξής αναδρομική σχέση (ας υποθέσουμε χωρίς βλάβη της γενικότητας ότι το n είναι δύναμη του δύο):

$$\begin{cases} T(n) = 2 \cdot T(\frac{n}{2}) + O(n) \\ T(1) = O(1) \end{cases}$$

καθώς σε κάθε αναδρομική κλήση για έναν πίνακα μεγέθους n καλούμε δύο φορές τον MergeSort με είσοδο πίνακες μεγέθους $\frac{n}{2}$ και μία φορά τον Merge για πίνακες με αθροιστικό μέγεθος n.

Παρατηρήστε ότι μετά από k αναδρομικές κλήσεις θα έχουμε 2^k πίνακες μεγέθους $\frac{n}{2^k}$ να συγχωνεύσουμε, και για κάθε συγχώνευση θα χρειαστούμε χρόνο $O(\frac{n}{2^k})$. Για να φτάσουμε σε πίνακα μεγέθους 1 θα χρειαστεί να γίνουν $\log n$ αναδρομικές κλήσεις. Οπότε συνολικά ο χρόνος που χρειαζόμαστε για τα $\log n$ επίπεδα είναι:

$$T(n) = \sum_{k=0}^{\log n - 1} 2^k \cdot O\Big(\frac{n}{2^k}\Big) = O(n) \cdot \sum_{k=0}^{\log n - 1} 1 = O(n \log n)$$

Συνεπώς και ο MergeSort έχει υποτετραγωνικό χρόνο, όμως πάλι σπαταλάμε επιπλέον χώρο για τον προσωρινό πίνακα C μέσα στον Merge.

Θα δούμε ότι στην πραγματικότητα δεν είναι απαραίτητος ο επιπλέον χώρος για να τοποθετήσουμε τα στοιχεία στη σωστή θέση. Το μόνο που χρειάζεται να κάνουμε είναι να τα τοποθετούμε σε μια «σχεδόν σωστή» θέση που σε κάθε επανάληψη θα τη βελτιώνουμε. Αν σταθούμε «τυχεροί» αυτό θα το επιτύχουμε σε υποτετραγωνικό χρόνο.

6.4 «Γρήγορη» ταξινόμηση

Ο πιο ευρέως χρησιμοποιούμενος αλγόριθμος ταξινόμησης είναι ο QuickSort. Ο χρόνος του στη χειρότερη περίπτωση είναι τετραγωνικός, όμως στις περισσότερες περιπτώσεις ο χρόνος του είναι αντίστοιχος με τον MergeSort. Το μεγάλο πλεονέκτημά του είναι ότι κάνει την ταξινόμηση «επί τόπου», πράγμα που τον καθιστά τρομερά χρήσιμο σε περιπτώσεις όπου ο χώρος είναι εξίσου σημαντικός. Και αυτός ο αλγόριθμος βασίζεται στην ιδέα του διαίρει και κυρίευε.

Φανταστείτε ότι γνωρίζουμε τη θέση p όπου (περίπου) τα μισά στοιχεία του πίνακα έχουν τιμή μικρότερη είτε ίση με την τιμή $A[p]$ και τα υπόλοιπα μεγαλύτερη από $A[p]$. Θα μεταφέρουμε τα μεγαλύτερα στοιχεία δεξιά από τη θέση p και τα μικρότερα αριστερά, και θα επαναλάβουμε για τους υποπίνακες $A[1, \ldots, p]$ και $A[p+1, \ldots, n]$ (όπου ο A έχει n στοιχεία).

Αν μπορούμε σε κάθε αναδρομική κλήση να βρούμε τη θέση p που «χωρίζει» τον πίνακα ακριβώς στη μέση ο χρόνος του αλγορίθμου περιγράφεται από την αναδρομική σχέση:

$$\begin{cases} T(n) = 2 \cdot T(\frac{n}{2}) + O(n) \\ T(1) = O(1) \end{cases}$$

καθώς «σπάμε» τον πίνακα σε δύο υποπίνακες με το μισό μέγεθος και ο χρόνος για την αντιμετάθεση των στοιχείων είναι συνολικά ανάλογος με το μέγεθος των δύο πινάκων. Βέβαια για να βρίσκουμε συνεχώς τη θέση p θα πρέπει να έχουμε ταξινομήσει (έστω και πρόχειρα) τον πίνακα, δηλαδή να έχουμε εξ' αρχής λυμένο το πρόβλημά μας!

Η επιλογή του p είναι ο πιο σημαντικός παράγοντας όσον αφορά τον χρόνο του αλγορίθμου: Αν για κακή μας τύχη επιλέγουμε συνεχώς p τέτοιο ώστε το $A[p]$ να είναι το μικρότερο στοιχείο του πίνακα (ή το μεγαλύτερο) τότε τα καινούργια υποπροβλήματα θα είναι ένας πίνακας με μέγεθος $n-1$ και ένας με μέγεθος 1. Συνεπώς ο χρόνος του αλγορίθμου θα δίνεται από τη σχέση:

$$\begin{cases} T(n) = T(n-1) + T(1) + O(n) = T(n-1) + O(n) \\ T(1) = O(1) \end{cases}$$

και θα είναι $O(n^2)$. Αποδεικνύεται ότι αν επιλέγουμε το p τυχαία τότε «περιμένουμε» ότι ο αλγόριθμος θα χρειαστεί χρόνο $O(n \log n)$ [1]. Ας δούμε τον αλγόριθμο:

QuickSort(A, l, r)

Είσοδος: Πίνακας A, και θέσεις l, r του πίνακα με $l \leq r$
Έξοδος : Τίποτα (εσωτερικά τα στοιχεία του πίνακα A έχουν ταξινομηθεί)

1 **if** $l < r$ **then**
2     $i \leftarrow$Partition(A, l, r)
3   QuickSort$(A, l, i-1)$
4   QuickSort$(A, i+1, r)$

Το σύνολο της δουλειάς γίνεται και πάλι στην υπορουτίνα Partition η οποία ταξινομεί τα στοιχεία ως προς ένα τυχαίο στοιχείο (το στοιχείο στη θέση p):

Partition(A, l, r)

Είσοδος: Πίνακας A, και θέσεις l, r του πίνακα με $l \leq r$
Έξοδος : Θέση i τέτοια ώστε τα στοιχεία με τιμή μεγαλύτερη από $A[i]$ έχουν μεταφερθεί δεξιά της και τα υπόλοιπα αριστερά

1 $p \leftarrow$ **random**(l, r)
    %Βάζουμε το στοιχεία $A[p]$ στο τέλος για να μην χάσουμε τη θέση του
2 $tmp \leftarrow A[p]$
3 $A[p] \leftarrow A[r]$
4 $A[r] \leftarrow tmp$
5 $i \leftarrow l - 1$
6 **for** $j \leftarrow l$ **to** $r - 1$
7   **if** $A[j] < A[r]$ **then**
8         $i \leftarrow i + 1$
9         $tmp \leftarrow A[j]$
10         $A[j] \leftarrow A[i]$
11         $A[i] \leftarrow tmp$
    %Μένει να τοποθετήσουμε και το στοιχεία $A[r]$ (πρώην $A[p]$) στη σωστή θέση
12 $i \leftarrow i + 1$
13 $tmp \leftarrow A[r]$
14 $A[r] \leftarrow A[i]$
15 $A[i] \leftarrow tmp$
16 **return** i

Στο βήμα 1 του αλγορίθμου επιλέγουμε τυχαία μία τιμή ανάμεσα στις $l, l+1, \ldots, r$ για να κάνουμε την πρόχειρη ταξινόμηση των στοιχείων ως προς το στοιχείο $A[p]$. Θα μπορούσαμε πολύ απλά να διαλέξουμε το πρώτο στοιχείο του πίνακα (το στοιχείο $A[l]$ δηλαδή). Αυτή η επιλογή δεν θα αλλοίωνε την αναμενόμενη χρονική πολυπλοκότητα του αλγόριθμου. Αυτό

[1] Δηλαδή η χρονική πολυπλοκότητα αναμενόμενης περίπτωσης του αλγορίθμου είναι $O(n \log n)$.

όμως θα είχε τραγικά αποτελέσματα όταν ο πίνακας είναι ήδη ταξινομημένος (θα χρειαζόμασταν, όπως αναφέραμε πριν, χρόνο $O(n^2)$ για πίνακα με n στοιχεία). Συνεπώς μια πιο ασφαλής επιλογή είναι να το επιλέγουμε τυχαία (σύμφωνα με την ομοιόμορφη κατανομή) ή ακόμα να επιλέξουμε την *ενδιάμεση τιμή* μεταξύ του πρώτου, τελευταίου και μεσαίου στοιχείου του πίνακα [1].

Ασκήσεις

6.1. Δείξτε ότι ένας αλγόριθμος ταξινόμησης που βασίζεται στη σύγκριση στοιχείων δεν μπορεί να κάνει λιγότερες από $c \cdot n \log n$ συγκρίσεις (στη χειρότερη περίπτωση) για κάποια θετική σταθερά c, όπου n το πλήθος των προς ταξινόμηση στοιχείων.

6.2. Έστω πίνακας A που περιέχει n ακέραιους αριθμούς. Ένα στοιχείο του A καλείται *κυρίαρχο* αν και μόνο αν εμφανίζεται τουλάχιστον $n/2+1$ φορές στον πίνακα. Δώστε αλγόριθμο IsDominate(A, n, x) που δέχεται ως είσοδο έναν πίνακα A με n ακέραιους αριθμούς και ένα στοιχείο του x, και ελέγχει αν το x είναι κυρίαρχο επιστρέφοντας κατάλληλα ΝΑΙ ή ΟΧΙ. Ο χρόνος του αλγορίθμου θα πρέπει να είναι $O(n \log n)$.

6.3. Σας δίνεται ταξινομημένος πίνακας A που περιέχει n διακεκριμένους ακέραιους αριθμούς. Δώστε αλγόριθμο Check(A, n) που ελέγχει αν υπάρχει $i \in \{1, 2, \cdots, n\}$ τέτοιο ώστε $A[i] = i$ επιστρέφοντας κατάλληλα ΝΑΙ ή ΟΧΙ. Ο χρόνος του αλγορίθμου θα πρέπει να είναι $O(\log n)$.

6.4. Σας δίνονται ταξινομημένοι πίνακες A, B που περιέχουν m και n διακεκριμένους ακέραιους αριθμούς αντίστοιχα, όπου $m, n \geq 1$. Δώστε αλγόριθμο Smallest(A, m, B, n, k) που βρίσκει τον k-οστό μικρότερο αριθμό και των δύο πινάκων (υποθέστε ότι $k \leq \min\{i, j\}$). Ο χρόνος του αλγορίθμου θα πρέπει να είναι $O(\log k)$. (Για παράδειγμα αν $A = [-3, 0, 4, 7, 11, 24, 56, 85]$ και $B = [-4, -1, 9, 18, 34, 41, 62]$ τότε ο Smallest$(A, 8, B, 7, 7)$ επιστρέφει 9).

[1] Με αυτόν τον τρόπο δεν θα έχουμε τη χειρότερη δυνατή επίδοση αν ο πίνακας είναι ταξινομημένος (κατά αύξουσα ή φθίνουσα σειρά).

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Γεώργιος Φρ. Γεωργακόπουλος: *Δομές Δεδομένων Έννοιες, Τεχνικές και Αλγόριθμοι*, Πανεπιστημιακές Εκδόσεις Κρήτης

[2] Παναγιώτης Δ. Μποζάνης: *Δομές Δεδομένων*, Εκδόσεις Τζιόλα

[3] Μανώλης Λουκάκης: *Δομές Δεδομένων, Αλγόριθμοι*, Εκδόσεις Σοφία

[4] Νικόλαος Μισυρλής: *Δομές δεδομένων με C*, Εκδόσεις Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών

[5] Ιωάννης Χατζηλυγερούδης: *Δομές Δεδομένων*, Ελληνικό Ανοικτό Πανεπιστήμιο

[6] Αθανάσιος Κ. Τσακαλίδης: *Δομές Δεδομένων*, Πανεπιστήμιο Πατρών

[7] Michael T. Goodrich, Roberto Tamassia: *Δομές Δεδομένων & Αλγόριθμοι σε JAVA*, Εκδόσεις Δίαυλος

[8] Sahnii Sartaj: *Δομές δεδομένων, αλγόριθμοι και εφαρμογές C++*, Εκδόσεις Τζιόλα

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Εισαγωγή στους Αλγορίθμους*, Πανεπιστημιακές Εκδόσεις Κρήτης

ΛΙΣΤΑ ΑΛΓΟΡΙΘΜΩΝ