ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Μαριος Συντιχακης

ΑΘΗΝΑ 2005

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή στη θεωρία αλγορίθμων

1.1 Αλγόριθμοι

Κάθε φορά που γράφουμε ένα πρόγραμμα, προδιαγράφουμε με τρόπο αυστηρό, ένα σύνολο υπολογιστικών βημάτων καθώς επίσης και την ακολουθία εκτέλεσής τους για την λύση ενός καλά ορισμένου προβλήματος. Κάθε φορά που γράφουμε ένα πρόγραμμα, ταυτόχρονα καθορίζουμε μια "συνταγή" για την επίλυση ενός υπολογιστικού προβλήματος. Η συνταγή αυτή ονομάζεται *αλγόριθμος*.

Ορισμός 1-1. Ένας αλγόριθμος είναι μια πεπερασμένη, αυστηρά καθορισμένη, ακολουθία υπολογιστικών βημάτων για την ολοκλήρωση ενός υπολογισμού.

Παραδοσιακά, γίνεται μια σαφής διάκριση μεταξύ του αλγορίθμου και της υλοποίησής του σε μια γλώσσα προγραμματισμού. Με βάση τη διάκριση αυτή, ο αλγόριθμος είναι η ιδέα για την επίλυση του υπολογιστικού προβλήματος, ενώ το πρόγραμμα, η υλοποίηση, η εκτέλεση, της ιδέας. Το πλεονέκτημα που προσφέρει η διάκριση του αλγορίθμου από το πρόγραμμα που τον υλοποιεί, είναι η αφαίρεση των λεπτομερειών υλοποίησης κατά την μελέτη της αποτελεσματικότητας διαφορετικών αλγορίθμων για την επίλυση του ιδίου προβλήματος. Έτσι, είναι απαραίτητη μια *ψευδογλώσσα προγραμματισμού*, αρκετά αυστηρή ώστε να μπορεί να εκφράσει με σαφήνεια τα βήματα του αλγορίθμου, αλλά και αρκετά αφηρημένη ώστε να

μας απαλλάσσει από λεπτομέρειες υλοποίησης κατά τη μελέτη της απόδοσής του.

Εδώ δεν θα ακολουθήσουμε αυτή την προσέγγιση. Καʹ αρχήν, διότι στην πραγματικότητα δεν υπάρχει ουσιαστική διαφορά μεταξύ αλγορίθμων και προγραμμάτων· κάθε πρόγραμμα είναι ένας αλγόριθμος. Κατά δεύτερο λόγο, διότι δεν είναι ανάγκη να μάθουμε άλλη μια γλώσσα προγραμματισμού και μάλιστα αφηρημένη. Ένας ακόμη σημαντικότερος λόγος για τον οποίο δεν ακολουθούμε αυτή την προσέγγιση, είναι ότι όχι μόνο δεν χρειάζεται να αφαιρούμε λεπτομέρειες υλοποίησης, αλλά πρέπει να τις λαμβάνουμε σοβαρά υπόψη μας κατά την ανάλυση της αποτελεσματικότητας ενός αλγορίθμου. Τέλος, ο σημαντικότερος λόγος είναι ότι, όπως θα δούμε με λεπτομέρεια στα κεφάλαια που ακολουθούν, οι αλγόριθμοι είναι άρρηκτα συνδεδεμένοι με τα δεδομένα τα οποία χειρίζονται κατά την εκτέλεσή τους. Και το ενδιαφέρον μας, εστιάζεται κυρίως στα δεδομένα αυτά, και ειδικότερα, στους μηχανισμούς οργάνωσής τους με τρόπο τέτοιο ώστε να αυξάνεται η αποτελεσματικότητα των προγραμμάτων μας.

Πέρα από μια ακολουθία ενεργειών, οι αλγόριθμοι έχουν μερικά επιπλέον χαρακτηριστικά τα οποία τους ξεχωρίζουν από τις συνταγές μαγειρικής:

i) Εισόδους,
ii) εξόδους,
iii) πεπερασμένο μέγεθος,
iv) αυστηρότητα, και τέλος,
v) περατότητα.

Κώδικας 1.1: Άθροισμα όρων ακολουθίας

```
1 public static final long sum(int[] a) {
2    long sum = 0;
3    int i = 0;
4    while (i < a.length) {
5       sum = sum + a[i++];
6    }
7    return sum;
8 }
```

Δεδομένα εισόδου και εξόδου. Ένας αλγόριθμος έχει μηδέν ή περισσότερες εισόδους και τουλάχιστον μία έξοδο. Τόσο οι είσοδοι όσο

και οι έξοδοι έχουν πεπερασμένο μέγεθος. *Είσοδος* του αλγορίθμου είναι κάθε δεδομένο το οποίο απαιτείται για την ολοκλήρωση ενός υπολογισμού. *Έξοδος* του αλγορίθμου είναι κάθε αποτέλεσμα υπολογισμού που παράγει ο αλγόριθμος. Ο αλγόριθμος για το άθροισμα των όρων μιας ακολουθίας του Κώδικα 1.1, έχει μία είσοδο, την παράμετρο a, και μια έξοδο, την τιμή της τοπικής μεταβλητής sum. Και οι δύο είναι πεπερασμένου μεγέθους.

Πεπερασμένο μέγεθος. Ένας αλγόριθμος πρέπει να έχει πεπερασμένο αριθμό βημάτων και επομένως να εκτελεί ένα πεπερασμένο αριθμό στοιχειωδών λειτουργιών όπως πρόσθεση, σύγκριση, ανάθεση ολίσθηση. Ο αλγόριθμος του Κώδικα 1.1 θα τερματίσει μόλις η τοπική μεταβλητή i πάρει την τιμή a.length. Αν N είναι το πλήθος των θέσεων της συστοιχίας, και δεδομένου ότι η αρχική τιμή της i είναι 0 και αυξάνεται κατά 1 κάθε φορά που εκτελείται το σώμα του βρόχου while, ο αλγόριθμος θα εκτελέσει συνολικά $4N + 3$ στοιχειώδεις λειτουργίες προκειμένου να ολοκληρώσει τον υπολογισμό: N συγκρίσεις, N μοναδιαίες αυξήσεις, N προσθέσεις, και $N + 3$ αναθέσεις.

Αυστηρότητα. Κάθε βήμα ενός αλγορίθμου πρέπει να είναι σαφές, αυστηρά καθορισμένο και αναμφισβήτητο. Οι πράξεις που εκτελούνται πρέπει να είναι καλώς ορισμένες σε κάθε περίπτωση. Ο Κώδικας 1.1 για παράδειγμα, είναι αρκετά σαφής ώστε ο μεταγλωττιστής της Java να τον μετατρέψει σε εντολές της JVM.

Περατότητα. Κάθε αλγόριθμος πρέπει να ολοκληρώνει τον υπολογισμό του σε πεπερασμένο και μετρήσιμο χρονικό διάστημα. Αξίζει να σημειώσουμε εδώ πως το πεπερασμένο μέγεθος ενός αλγορίθμου δεν εγγυάται κατά ανάγκη ότι ο αλγόριθμος ολοκληρώνει τον υπολογισμό για κάθε πιθανό συνδυασμό των δεδομένων εισόδου.

Στις επόμενες ενότητες θα επιχειρήσουμε μια εισαγωγή στο εξαιρετικά σημαντικό, και ταυτόχρονα συναρπαστικό, αντικείμενο της ανάλυσης αλγορίθμων. Ο σκοπός μας δεν είναι η εξαντλητική παρουσίαση του αντικειμένου, αλλά η παρουσίαση των βασικών αρχών που απαιτούνται για την κατανόηση της αποτελεσματικότητας των

```
00: lconst_0
01: lstore_1
02: iconst_0
03: istore_3
04: goto  17
07: lload_1
08: aload_0
09: iload_3
10: iinc  3, 1
13: iaload
14: i2l
15: ladd
16: lstore_1
17: iload_3
18: aload_0
19: arraylength
20: if_icmplt 7
23: lload_1
24: lreturn
```

Σχήμα 1-1: Οι εντολές JVM του Κώδικα 1.1.

μηχανισμών οργάνωσης δεδομένων που παρουσιάζονται στα επόμενα κεφάλαια.

Ασκήσεις

1-1 Υλοποιήστε ένα αλγόριθμο που εντοπίζει τη μέγιστη τιμή ενός πίνακα ακεραίων.

▷ **1-2** Ένας πίνακας περιέχει $N - 1$ διακριτούς ακεραίους στο διάστημα $[1, N]$. Υλοποιείστε ένα αλγόριθμο για τον εντοπισμό του μοναδικού αριθμού που δεν περιέχεται στον πίνακα.

1-3 Υλοποιήστε ένα αλγόριθμο που υπολογίζει τη διασπορά των τιμών ενός πίνακα πραγματικών αριθμών.

● **1-4** Ένας πίνακας με N γραμμές και N στήλες περιέχει τιμές 0 και 1. Σε κάθε γραμμή του πίνακα, όλα τα 0 προηγούνται του πρώτου 1. Κάθε γραμμή i, $0 \leq i < N - 1$ έχει τουλάχιστον όσα 0 έχει η γραμμή $i + 1$. Υλοποιήστε ένα αλγόριθμο σε Java, που υπολογίζει το πλήθος των 1 που περιέχονται στον πίνακα.

1-5 Σε τι μοιάζουν και σε τι διαφέρουν οι συνταγές μαγειρικής και οι αλγόριθμοι;

1.2 Μαθηματικές έννοιες

Σύνολα και διαστήματα. Ένα πεπερασμένο σύνολο συνήθως συμβολίζεται με την παράθεση των στοιχείων του εντός αγκυλών π.χ., $\{1, 2, 3\}$. Με παρόμοιο συμβολισμό μπορούμε να παριστάνουμε και απειροσύνολα χρησιμοποιώντας αποσιωπητικά. Για παράδειγμα το σύνολο των ακεραίων αριθμών είναι το σύνολο $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Το σύνολο των φυσικών αριθμών είναι το $\mathbb{N} = \{1, 2, 3, \ldots\}$. Με $\mathbb{R}$ θα συμβολίζουμε το σύνολο των πραγματικών αριθμών. Το σύνολο των θετικών πραγματικών αριθμών θα συμβολίζουμε με $\mathbb{R}^+$, ενώ το σύνολο των μη αρνητικών πραγματικών είναι το

$$\mathbb{R}^* = \mathbb{R}^+ \cup \{0\}.$$

Για διατεταγμένα σύνολα, μπορούμε να χρησιμοποιούμε το συμβολισμό διαστήματος προκειμένου να υποδείξουμε ένα συνεχές υποσύνολό τους. Έτσι, αν μιλάμε για φυσικούς αριθμούς, το διάστημα

$(1, 8]$ συμβολίζει το σύνολο $\{2, 3, \ldots, 8\}$. Μια παρένθεση υποδεικνύει ένα διάστημα ανοικτό από την αντίστοιχη πλευρά, ενώ μια αγκύλη, διάστημα κλειστό από την αντίστοιχη πλευρά. Για διαστήματα φυσικών αριθμών ισχύει η σχέση

$$[a, b] = (a - 1, b + 1) = (a - 1, b] = [a, b + 1)\,. \tag{1.1}$$

Ακέραιες συναρτήσεις. Αν x είναι ένας πραγματικός αριθμός τότε θα συμβολίζουμε με $\lfloor x \rfloor$ το *δάπεδό* του, δηλαδή το μεγαλύτερο ακέραιο που είναι μικρότερος του x. Ανάλογα ορίζουμε την *οροφή* του x, $\lceil x \rceil$ ως το μικρότερο ακέραιο που είναι μεγαλύτερος του x. Προφανώς ισχύει ότι

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1. \tag{1.2}$$

Επιπλέον, οι παρακάτω σχέσεις προκύπτουν εύκολα από τους προηγούμενους ορισμούς.

$$\lfloor x \rfloor = \lceil x \rceil\,,\ x \in \mathbb{Z} \tag{1.3}$$

$$\lceil x \rceil = \lfloor x \rfloor + 1,\ x \in \mathbb{R} \tag{1.4}$$

$$\lfloor -x \rfloor = -\lceil x \rceil \tag{1.5}$$

Αν x και y είναι πραγματικοί αριθμοί, ορίζουμε τον τελεστή mod ως εξής:

$$x \bmod y = \begin{cases} x - y\lfloor x/y \rfloor & \text{αν } y \neq 0 \\ x & \text{αν } y = 0 \end{cases}$$

Η ποσότητα $x \bmod y$ κείται στο διάστημα $[0, y)$ όταν ο y είναι θετικός, και στο διάστημα $(y, 0]$ όταν ο y είναι αρνητικός. Επιπλέον η ποσότητα $x - x \bmod y$ είναι ακέραιο πολλαπλάσιο του y. Όταν περιοριζόμαστε στο σύνολο των ακεραίων, η ποσότητα $x \bmod y$ είναι το υπόλοιπο της διαίρεσης x/y.

Αθροίσματα και γινόμενα. Το άθροισμα $x_1 + x_2 + \cdots + x_N$ συμβολίζεται ως

$$\sum_{i=1}^{N} x_i,$$

ενώ το γινόμενο $x_1 x_2 \cdots x_N$ με

$$\prod_{i=1}^{N} x_i.$$

Στους συμβολισμούς αυτούς το i ονομάζεται *μεταβλητή ελέγχου του αθροίσματος*. Ορισμένες φορές, και ειδικά όταν πρόκειται για αθροίσματα απείρων όρων, αντί να καθορίζουμε τα όρια της μεταβλητής ελέγχου του αθροίσματος, καθορίζουμε μια συνθήκη την οποία πρέπει να πληροί η μεταβλητή ελέγχου προκειμένου ο αντίστοιχος όρος να συμμετάσχει στο άθροισμα. Έτσι για παράδειγμα, μπορούμε να συμβολίσουμε το άθροισμα απείρων όρων $x_1 + x_2 + \ldots$ ως

$$\sum_{i \geq 0} x_i.$$

Ανάλογες συμβάσεις ισχύουν και για τα γινόμενα. Οι επόμενες σχέσεις αποτελούν βασικές ιδιότητες των αθροισμάτων.

$$\left(\sum_{P(i)} x_i \right) \left(\sum_{Q(j)} x_j \right) = \sum_{P(i)} \sum_{Q(j)} x_i x_j \tag{1.6}$$

$$\sum_{P(i)} x_i = \sum_{P(j)} x_j = \sum_{P(Q(i))} x_{Q(i)} \tag{1.7}$$

$$\sum_{P(i)} \sum_{Q(j)} x_{ij} = \sum_{Q(j)} \sum_{P(i)} x_{ij} \tag{1.8}$$

Οι σχέσεις (1.6), (1.7) και (1.8) είναι γνωστές ως *επιμεριστικός νόμος*, *αλλαγή μεταβλητής ελέγχου*, και *αλλαγή σειράς*, αντίστοιχα.

Ορισμένα αθροίσματα παρουσιάζουν ιδιαίτερο ενδιαφέρον στη μελέτη της συμπεριφοράς αλγορίθμων, και για το λόγο αυτό τα αναφέρουμε εδώ. Το άθροισμα των N πρώτων φυσικών αριθμών

$$\sum_{i=1}^{N} i = \frac{N\,(N+1)}{2}. \tag{1.9}$$

Το άθροισμα τετραγώνων των N πρώτων φυσικών αριθμών

$$\sum_{i=1}^{N} i^2 = \frac{2N^3 + 3N^2 + N}{6}. \tag{1.10}$$

Το άθροισμα των N πρώτων δυνάμεων του 2. Πρόκειται για μια ειδική περίπτωση της (1.13) για $a = 2$.

$$\sum_{i=0}^{N-1} 2^i = 2^N - 1 \tag{1.11}$$

Το άθροισμα των N πρώτων όρων αριθμητικής προόδου με αρχικό όρο a_0 και βήμα ω

$$\sum_{i=0}^{N-1} (a_0 + i\omega) = a_0 N + \frac{\omega}{2} N(N-1) = \frac{a_0 + a_{N-1}}{2} N. \tag{1.12}$$

Το άθροισμα των N πρώτων δυνάμεων του πραγματικού αριθμού a

$$\sum_{i=0}^{N-1} a^i = \frac{a^N - 1}{a - 1}. \tag{1.13}$$

Και τέλος, το αρμονικό άθροισμα

$$H_N = \sum_{i=1}^{N} \frac{1}{i} \approx 1 + \ln N. \tag{1.14}$$

Λογάριθμοι. Ονομάζουμε λογάριθμο με βάση b του $x \in \mathbb{R}^+$ και συμβολίζουμε με $\log_b x$, τον αριθμό y για τον οποίο ισχύει $b^y = x$. Ο λογάριθμος ενός θετικού πραγματικού αριθμού είναι η δύναμη στην οποία πρέπει να υψωθεί η βάση του λογαρίθμου ώστε το αποτέλεσμα να ισούται με x. Για ορισμένους λογαρίθμους που απαντώνται πολύ συχνά, χρησιμοποιούμε συμβολισμούς χωρίς βάση. Έτσι ο λογάριθμος με βάση $e \approx 2.7182818284$, ή αλλιώς *φυσικός λογάριθμος*, του x, συμβολίζεται με

$$\ln x \equiv \log_e x,$$

ο λογάριθμος με βάση 10, γνωστός και ως *δεκαδικός* ή *κοινός λογάριθμος* του x συμβολίζεται με

$$\log x \equiv \log_{10} x,$$

και τέλος ο *δυαδικός λογάριθμος*, ο λογάριθμος με βάση 2 του x, πολύ σημαντικός στην ανάλυση της πολυπλοκότητας αλγορίθμων, συμβολίζεται με

$$\lg x \equiv \log_2 x.$$

Οι παρακάτω σχέσεις αποτελούν βασικές ιδιότητες των λογαρίθμων.

$$\log_b x^a = a \log_b x \tag{1.15}$$

$$\log_b b^a = a \tag{1.16}$$

$$\log_b x = \frac{\log_a x}{\log_a b} \tag{1.17}$$

$$\log_b \prod_{P(i)} x_i = \sum_{P(i)} \log_b x_i \tag{1.18}$$

$$(\ln x)' = \frac{1}{x} \tag{1.19}$$

Προσέγγιση αθροισμάτων. Το άθροισμα των τιμών μιας συνάρτησης στο διάστημα $[a, b] \subseteq \mathbb{N}$ μπορεί να προσεγγιστεί με τεχνικές ολοκλήρωσης. Μια προσέγγιση για την ποσότητα $\sum_{i=a}^{b} f(i)$ δίνεται από τη σχέση (1.20) αν η συνάρτηση f είναι αύξουσα, ή από τη σχέση (1.21) αν η συνάρτηση f είναι φθίνουσα.

$$\int_{a-1}^{b} f(x)\,dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a}^{b+1} f(x)\,dx \tag{1.20}$$

$$\int_{a}^{b+1} f(x)\,dx \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)\,dx \tag{1.21}$$

Ο κανόνας L' Hôspital. Αν $\lim_{N\to\infty} f(N) = \lim_{N\to\infty} g(N) = \infty$ και οι συναρτήσεις f και g είναι παραγωγίσιμες, τότε ισχύει:

$$\lim_{N\to\infty} \frac{f(N)}{g(N)} = \lim_{N\to\infty} \frac{f'(N)}{g'(N)} \tag{1.22}$$

Πιθανότητες. Ας υποθέσουμε ότι κατά την εκτέλεση ενός τυχαίου πειράματος, όπως το στρίψιμο ενός νομίσματος ή η επιλογή ενός αντικειμένου από κληρωτίδα, υπάρχει ένα πεπερασμένο σύνολο ενδεχομένων εκβάσεων του πειράματος, $E = \{e_0, e_1, \ldots, e_{N-1}\}$. Το σύνολο E των ενδεχομένων εκβάσεων, ονομάζεται *δειγματικός χώρος*. Αν σε κάθε $e_i \in E$, $0 \leq i \leq N-1$ αντιστοιχίσουμε ένα πραγματικό αριθμό $p(e_i)$ έτσι ώστε να ισχύουν οι σχέσεις (1.23) και (1.24), θα λέμε ότι $p(e_i)$ είναι η *πιθανότητα* εμφάνισης του ενδεχομένου e_i.

$$0 \leq p(e_i) \leq 1, \ \forall i \in [0, N-1] \tag{1.23}$$

$$\sum_{i=0}^{N-1} p(e_i) = 1 \tag{1.24}$$

Η συνάρτηση $p : E \to [0,1]$ ονομάζεται *συνάρτηση πιθανότητας* και με βάση αυτή μπορούμε να υπολογίσουμε όχι μόνο την πιθανότητα ενός μεμονωμένου ενδεχομένου αλλά και υποσυνόλων του δειγματικού χώρου χρησιμοποιώντας τη σχέση (1.25).

$$p(A) = \sum_{e \in A} p(e), \ A \subseteq E \tag{1.25}$$

Ασκήσεις

1-6 Απαριθμήστε τα στοιχεία του συνόλου $\left(0, 3\lfloor \frac{\pi}{3} \rfloor\right]$.

1-7 Απλουστεύστε την παράσταση $\lceil \lfloor x \rfloor \rceil$, $x \in \mathbb{R}$.

1-8 Υπολογίστε το άθροισμα $2^{21} + 2^{22} + \cdots + 2^{32}$.

1-9 Αποδείξτε ότι αν ο d διαιρεί τους φυσικούς αριθμούς m και $N \bmod m$, τότε διαιρεί και τον N.

1-10 Υπολογίστε τις ποσότητες $\log_a 1$, $\lg 256$, $\ln e^{\pi}$, $\log(1000)$, $\log(-100)$.

1-11 Αποδείξτε ότι $\log x^N = N \log x$, $N \in \mathbb{N}$, $x \in \mathbb{R}^+$.

1-12 Υπάρχει κάποια απλή σχέση μεταξύ $\ln 2$ και $\lg e$; Γενικότερα μεταξύ $\log_a b$ και $\log_b a$;

1-13 Υπολογίστε ένα διάστημα για την τιμή του αρμονικού αθροίσματος H_N χρησιμοποιώντας την (1.20).

1-14 Επιλέγουμε στην τύχη ένα μεταξύ τεσσάρων νομισμάτων και το στρίβουμε. Ποιά είναι πιθανότητα μετά την εκτέλεση αυτού του τυχαίου πειράματος, τρία από τα νομίσματα να δείχνουν γράμματα, αν αρχικά τα νομίσματα έδειχναν

i) κεφάλι, γράμματα, γράμματα, γράμματα,
ii) κεφάλι, γράμματα, κεφάλι, γράμματα, και,
iii) γράμματα, κεφάλι, κεφάλι, κεφάλι.

1-15 Αποδείξτε την σχέση (1.12).

1-16 Ένας πίνακας $M(i,j)$ με N γραμμές και N στήλες περιέχει τιμές 0 και 1. Ισχύει ότι $m_{i,j} = 1$ για κάθε $j \in [i, N)$, $0 \leq i < N$. Πόσα 0 έχει ο πίνακας;

1.3 Αναδρομικοί ορισμοί και μαθηματική επαγωγή

Ένας *αναδρομικός ορισμός* είναι ένας ορισμός ο οποίος χρησιμοποιεί την ίδια την οριζόμενη έννοια. Αν και φαίνεται παράδοξο εκ πρώτης όψεως, μια έννοια μπορεί να οριστεί με βάση τον εαυτό της. Το παραγοντικό ενός φυσικού αριθμού για παράδειγμα, ορίζεται ως το γινόμενο του αριθμού αυτού με το παραγοντικό του προηγούμενου φυσικού αριθμού. Ή πιο αυστηρά

$$N! = \begin{cases} 1 & \text{αν } N = 0 \\ N(N-1)! & \text{αν } N > 0 \end{cases}. \tag{1.26}$$

Για να επιδείξουμε τη χρησιμότητα και την ορθότητα του αναδρομικού ορισμού της σχέσης (1.26) ας υπολογίσουμε το $4!$.

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times 3 \times 2! \\ &= 4 \times 3 \times 2 \times 1! \\ &= 4 \times 3 \times 2 \times 1 \times 1 \\ &= 24 \end{aligned}$$

Ένα άλλο πολύ γνωστό παράδειγμα αναδρομικού ορισμού είναι η ακολουθία Fibonacci, η οποία ορίζεται από τις σχέσεις

$$F_N = \begin{cases} 0 & \text{αν } N = 0 \\ 1 & \text{αν } N = 1 \\ F_{N-1} + F_{N-2} & \text{αν } N \geq 2 \end{cases} \tag{1.27}$$

Ο πρώτος και ο δεύτερος όρος της ακολουθίας Fibonacci είναι 0 και 1 αντίστοιχα, ενώ κάθε όρος εκτός των δύο πρώτων, προκύπτει ως το άθροισμα των δύο προηγούμενων όρων της ακολουθίας.

Οι αναδρομικοί ορισμοί παρέχουν ένα δηλωτικό τρόπο για τον υπολογισμό μιας ποσότητας. Κάθε φορά που θέλουμε να υπολογίσουμε μια ποσότητα χρησιμοποιώντας ένα αναδρομικό ορισμό, είμαστε αναγκασμένοι να υπολογίσουμε (χρησιμοποιώντας αναδρομικά τον ίδιο ορισμό) ενδιάμεσες μικρότερες ποσότητες ώσπου να φτάσουμε σε μια περίπτωση του προβλήματος για την οποία το αποτέλεσμα είναι γνωστό, και έτσι, δεν απαιτείται εκ νέου εφαρμογή του ορισμού. Μια τέτοια συνθήκη ονομάζεται *βάση του αναδρομικού ορισμού*. Στη συνέχεια εφαρμόζοντας διάφορες πράξεις στα συσσωρευμένα ενδιάμεσα αποτελέσματα, προκύπτει το αποτέλεσμα του υπολογισμού. Στην περίπτωση της σχέσης (1.26), προκειμένου να υπολογίσουμε το παραγοντικό του αριθμού N, πρέπει να υπολογίσουμε τα παραγοντικά όλων των μικρότερων φυσικών αριθμών εκτός του 1, του οποίου το παραγοντικό είναι γνωστό, και στην συνέχεια να υπολογίσουμε το γινόμενό τους. Παρόμοια, για να υπολογίσουμε τον όρο F_N της ακολουθίας Fibonacci, πρέπει πρώτα να υπολογίσουμε τους όρους $F_{N-1}, F_{N-2}, \ldots, F_2$, και στην συνέχεια να υπολογίσουμε το άθροισμά τους και να το προσθέσουμε στο άθροισμα των δύο πρώτων όρων της ακολουθίας. Ένας αναδρομικός ορισμός μπορεί να έχει μία και μοναδική βάση, όπως για παράδειγμα ο ορισμός (1.26), ή περισσότερες από μία βάσεις όπως ο ορισμός (1.27).

Η αναδρομή έχει πολλές ομοιότητες με την επαγωγή η οποία είναι μια θεμελιώδης τεχνική για την απόδειξη θεωρημάτων. Θα περιγράψουμε την εφαρμογή της τεχνικής αυτής για την απόδειξη προτάσεων της μορφής

$$P(N),\ N \in \mathbb{N}$$

όπως για παράδειγμα $\frac{N^2+1}{2} > 12,\ N \geq 5$.

i). Αρχικά αποδεικνύουμε το $P(1)$, την ισχύ δηλαδή της πρότασης $P(N)$ για $N = 1$. Η απόδειξη αυτή αποτελεί τη βάση της επαγωγής ή αλλιώς την επαγωγική βάση.

ii). Στη συνέχεια θεωρούμε ότι ισχύει το θεώρημα $P(k)$ για κάθε $k \leq N$. Η υπόθεση αυτή ονομάζεται *επαγωγική υπόθεση*.

iii). Τελευταίο, και συνήθως δυσκολότερο, στάδιο της απόδειξης αποτελεί η απόδειξη του θεωρήματος $P(k+1)$ δεδομένου ότι ισχύει το θεώρημα $P(k)$ για κάθε $k \leq N$. Το στάδιο αυτό ονομάζεται επαγωγικό βήμα .

Εφαρμόζοντας τα παραπάνω ας αποδείξουμε ότι

$$N! = \prod_{i=1}^{N} i,\ N \in \mathbb{N}. \tag{1.28}$$

Απόδειξη. Αρχικά θα πρέπει να αποδείξουμε ότι η (1.28) ισχύει για $N = 1$, ή αλλιώς ότι

$$1! = \prod_{i=1}^{1} i.$$

Αυτό προκύπτει αμέσως από τη βάση του αναδρομικού ορισμού (1.26). Θεωρούμε τώρα ότι η (1.28) ισχύει για κάθε $k \leq N$. Δηλαδή ότι

$$k! = \prod_{i=1}^{k} i,\ k \leq N. \tag{1.29}$$

Για να ολοκληρώσουμε την απόδειξη, θα πρέπει να αποδείξουμε ότι αν η (1.29) ισχύει, τότε ισχύει και η (1.28) για $N = k+1$, δηλαδή

$$(k+1)! = \prod_{i=1}^{k+1} i,\ k \in \mathbb{N}.$$

Εφαρμόζοντας το αναδρομικό σκέλος της σχέσης (1.26) προκύπτει ότι

$$(k+1)! = (k+1)\,k! = (k+1)\prod_{i=1}^{k} i = \prod_{i=1}^{k+1} i$$

όπως έπρεπε να αποδείξουμε. □

Ο μέθοδος που χρησιμοποιήσαμε την αρχή αυτής της ενότητας για τον υπολογισμό του $4!$, η συνεχής δηλαδή εφαρμογή του αναδρομικού ορισμού ξεκινώντας από ένα αριθμό $k > 1$ ως ότου φτάσουμε στη βάση του ορισμού, είναι γνωστός ως τηλεσκοπικό ανάπτυγμα. Αυτή η απλή μέθοδος σε συνδυασμό με την επαγωγή είναι εξαιρετικά χρήσιμη στην ανάλυση της αποτελεσματικότητας

αλγορίθμων όπως θα δούμε παρακάτω. Η μέθοδος αυτή θα γίνει περισσότερο κατανοητή στην επίλυση μιας πιο γενικής αναδρομικής σχέσης όπως η σχέση $C_N = C_{N-1} + N$, $C_1 = 1$.

$$\begin{aligned} C_N &= C_{N-1} + N \\ &= C_{N-2} + (N-1) + N \\ &= C_{N-3} + (N-2) + (N-1) + N \\ &= \vdots \\ &= 1 + 2 + \cdots + N - 1 + N \\ &= \frac{N(N+1)}{2}. \end{aligned}$$

Το αποτέλεσμα αυτό, μπορεί να επαληθευτεί με μια επαγωγική απόδειξη.

Ασκήσεις

1-17 Αποδείξτε τη σχέση (1.10) χρησιμοποιώντας επαγωγή.

▷ **1-18** Ένας φυσικός αριθμός m λέγεται *πρώτος* αν είναι μεγαλύτερος του 1 και δεν διαιρείται από κανένα αριθμό στο διάστημα $(1, m)$. Αποδείξτε με ένα επαγωγικό επιχείρημα πως κάθε φυσικός αριθμός μεγαλύτερος του 1, μπορεί να γραφεί ως γινόμενο πρώτων αριθμών.

● **1-19** Δώστε ένα αναδρομικό ορισμό για την παράσταση $\log(N!)$.

1-20 Δώστε ένα αναδρομικό ορισμό για το άθροισμα

$$\sum_{i=1}^{N} i = 1 + 2 + \cdots + N.$$

● **1-21** Χρησιμοποιώντας επαγωγή, αποδείξτε ότι $F_N \leq \phi^{N-1}$, $\phi = \left(1 + \sqrt{5}\right)/2$. *Υπόδειξη*: Αποδείξτε πρώτα ότι $\phi^2 = \phi + 1$.

1-22 Χρησιμοποιώντας επαγωγή, αποδείξτε ότι απαιτούνται F_{N+1} εφαρμογές του ορισμού (1.27) για τον υπολογισμό του όρου F_N της ακολουθίας Fibonacci.

▷ **1-23** Υπολογίστε μια κλειστή μορφή για την αναδρομική σχέση $C_N = C_{N-1} + \ln N$ όπου $C_1 = 0$.

● **1-24** Υπολογίστε μια κλειστή μορφή για την αναδρομική σχέση $C_N = C_{N/2} + N$, $C_1 = 0$. *Υπόδειξη*: Θεωρείστε ότι $N = 2^k$, $k \geq 1$.

1.4 Ανάλυση αλγορίθμων

Ο όρος ανάλυση αλγορίθμων περιγράφει τη διαδικασία της μελέτης της συμπεριφοράς ενός αλγορίθμου σε σχέση με προκαθορισμένα κριτήρια. Υπάρχουν πολλά κριτήρια τα οποία καθοδηγούν την ανάλυση ενός αλγορίθμου, ανάλογα με το πεδίο εφαρμογής. Είναι η *ορθότητα* αυτό που θα μας απασχολήσει κατά κύριο λόγο σε ένα πρόγραμμα το οποίο εκτελείται μια φορά την εβδομάδα για να μεταφέρει μερικά δεδομένα μεταξύ δύο συσκευών αποθήκευσης. Σε μια εφαρμογή λήψης παραγγελιών μέσω Διαδικτύου η οποία πρέπει να εξυπηρετεί χιλιάδες (κατά κανόνα απαιτητικούς) πελάτες θα ενδιαφερθούμε πέραν της ορθότητας και για τον αριθμό των στοιχειωδών λειτουργιών που απαιτούνται για την καταχώρηση μιας παραγγελίας. Αυτό θα μας δώσει μια εκτίμηση του αριθμού των παραγγελιών που μπορούμε να λαμβάνουμε ανά δευτερόλεπτο ή του αριθμού των πελατών που μπορούμε να εξυπηρετούμε ανά ώρα. Τα πιο σημαντικά κριτήρια ανάλυσης αλγορίθμων είναι

i) η ορθότητα,
ii) η απαιτούμενη ποσότητα μνήμης, και τέλος,
iii) ο χρόνος εκτέλεσης.

Πολύ σημαντικές ιδιότητες ενός αλγορίθμου είναι εξάλλου η *απλότητα*, η *κομψότητα*, καθώς επίσης η *ορθή δόμησή* του. Οι μελέτη των ιδιοτήτων αυτών ωστόσο, εμπίπτει περισσότερο στο πεδίο της τεχνολογίας λογισμικού παρά σε αυτό της ανάλυσης αλγορίθμων.

1.4.1 Ορθότητα

Προκειμένου να μελετήσουμε την ορθότητα ενός αλγορίθμου, πρέπει κατ' αρχήν να έχουμε μια σαφή διατύπωση για το τι πρέπει να κάνει ο αλγόριθμος αυτός. Πρέπει δηλαδή να έχουμε ένα ξεκάθαρο ορισμό για το τι είναι *ορθό* στη συγκεκριμένη περίπτωση. Η πιο συνηθισμένη πρακτική είναι να περιγράψουμε συνθήκες που πρέπει να πληρούν τα δεδομένα εισόδου καθώς επίσης και τα αναμενόμενα αποτελέσματα και στη συνέχεια να αποδείξουμε ότι σε κάθε περίπτωση στην οποία τα δεδομένα εισόδου πληρούν τις αναγκαίες συνθήκες, ο αλγόριθμος τερματίζει παράγοντας τα αναμενόμενα αποτελέσματα.

Για απλούς αλγορίθμους μπορούμε κατά κανόνα, χρησιμο-

ποιώντας διάφορες μαθηματικές τεχνικές, να διατυπώσουμε προτάσεις που αποδεικνύουν την ορθή συμπεριφορά τους. Μπορούμε να απομονώσουμε για παράδειγμα τις τιμές των μεταβλητών ελέγχου των βρόχων και χρησιμοποιώντας επαγωγή να δείξουμε ότι έχουν τις αναμενόμενες τιμές στο τέλος της επεξεργασίας.

Στη γενική περίπτωση η χρήση τυπικών μεθόδων μπορεί να είναι αρκετά δύσκολη ακόμη και για προγράμματα μερικών δεκάδων γραμμών κώδικα. Είναι εύκολα αντιληπτό πως για μια μικρή εφαρμογή, μεγέθους 10.000 γραμμών κώδικα, η χρήση τυπικών μεθόδων είναι πρακτικά ανεφάρμοστη. Από την άλλη πλευρά, ακόμη κι όταν έχουμε σχετικά μικρά προγράμματα να ελέγξουμε, τα ερωτήματα τα οποία μπορούμε να απαντήσουμε είναι περιορισμένα.

Τα παραπάνω δεν σημαίνουν σε καμία περίπτωση βέβαια ότι πρέπει να ξεχάσουμε τη χρήση τυπικών μεθόδων για την ανάλυση ορθότητας των αλγορίθμων. Αντίθετα, όπως γίνεται συνήθως όταν δεν μπορούμε να λύσουμε ένα μεγάλο πρόβλημα, προσπαθούμε να το χωρίσουμε σε μικρότερα προβλήματα τα οποία μπορούμε να αντιμετωπίσουμε με μεγαλύτερη άνεση. Έτσι πρέπει πάντα να φροντίζουμε ώστε όλα τα εξαρτήματα που απαρτίζουν ένα σύστημα λογισμικού να έχουν φτιαχτεί μικρά, σαφή, ξεκάθαρα, και, πάνω από όλα, ανεξάρτητα ώστε να μπορούμε να εφαρμόσουμε τυπικές μεθόδους για την αξιολόγηση της ορθότητάς τους.

Στη σύγχρονη τεχνολογία λογισμικού, όπου αρκετά συχνά οι εφαρμογές ξεπερνούν σε μέγεθος το ένα εκατομμύριο γραμμές κώδικα, πέρα από τις απλές τεχνικές που θα περιγράψουμε εδώ, υπάρχουν δεκάδες άτυποι αλλά παραγωγικότατοι τρόποι να ενισχύσουμε την εμπιστοσύνη μας στην ορθότητα των αλγορίθμων μας. Εδώ θα αρκεστούμε να παρουσιάσουμε μερικές βασικές τεχνικές ανάλυσης ορθότητας χρησιμοποιώντας ως παράδειγμα τον Κώδικα 1.2.

Έστω μια ακολουθία ακεραίων $a = a_0, a_1, \ldots, a_{N-1}$. Έστω επίσης ότι θέλουμε να εντοπίσουμε το μικρότερο $i \in [0, N)$ έτσι ώστε $a_i = x$, όπου x είναι ένας δοσμένος ακέραιος, που ονομζουμε ζητούμενο στοιχείο ή ζητούμενη τιμή. Ο Κώδικας 1.2 είναι ένας αλγόριθμος για την επίλυση του προβλήματος, γνωστός ως *ακολουθιακή αναζήτηση*. Ο αλγόριθμος αυτός, συγκρίνει διαδοχικά τα στοιχεία της ακολουθίας με το ζητούμενο στοιχείο μέχρι είτε να διαπιστώσει ισότητα, είτε να εξαντλήσει την ακολουθία. Θα αποδείξουμε ότι ο Κώδικας 1.2 είναι ορθός.

Απόδειξη. Αν υπάρχει στοιχείο με τιμή ίση της ζητούμενης, τότε η συνθήκη `a[i] == x` στη γραμμή 4 θα γίνει αληθής για κάποια τιμή i της τοπικής μεταβλητής `i` στο διάστημα $[0, N)$, $N = \texttt{a.length} \in \mathbb{N}$ και ο αλγόριθμος θα τερματίσει επιστέφοντας την τιμή αυτή. Αν ωστόσο κανένα στοιχείο δεν είναι ίσο με `x`, η τοπική μεταβλητή `i` θα πάρει διαδοχικά όλες τις τιμές στο διάστημα $[0, N]$ γεγονός που θα οδηγήσει το βρόχο `while` να τερματίσει λόγω της συνθήκης `i < a.length`. Έτσι, ο αλγόριθμος θα τερματίσει επιστρέφοντας την τιμή -1. Συμπερασματικά, ο Κώδικας 1.2 επιστρέφει τη θέση του στοιχείου της συστοιχίας `a` που ισούται με `x`, ή περιστρέφει την τιμή -1 αν τέτοιο στοιχείο δεν υπάρχει. □

Κώδικας 1.2: Ακολουθιακή αναζήτηση.

```
1 public static final int indexOf(int x, int[] a) {
2    int i = 0;
3    while (i < a.length) {
4       if (a[i] == x) return i;
5       i = i + 1;
6    }
7    return -1;
8 }
```

1.4.2 Απαιτούμενη ποσότητα μνήμης

Πολύ συχνά, για να λύσουμε ένα υπολογιστικό πρόβλημα χρειαζόμαστε πέρα από τα δεδομένα εισόδου, μνήμη για αποθήκευση ενδιάμεσων αποτελεσμάτων, υπολογισμών, ή μετασχηματισμό των δεδομένων εισόδου. Ο Κώδικας 1.2 για παράδειγμα, πέρα από τη μνήμη που απαιτείται για την αποθήκευση των στοιχείων της συστοιχίας `a` χρησιμοποιεί τη βοηθητική μεταβλητή `i`. Η ποσότητα μνήμης που απαιτείται για να αποθηκευτούν τα δεδομένα που χρησιμοποιεί ένας αλγόριθμος συμπεριλαμβανομένων και των δεδομένων εισόδου, είναι ένα σημαντικό κριτήριο αποτελεσματικότητας. Ο Κώδικας 1.2 για παράδειγμα, όταν εκτελείται σε μια εικονική μηχανή Java, απαιτεί $4(N+2)$ ψηφιοσυλλαβές μνήμης για την εκτέλεσή του, όπου N είναι το πλήθος των στοιχείων της συστοιχίας `a`.

Πριν από μερικά χρόνια οι απαιτήσεις μνήμης είχαν πολύ σημαντικό ρόλο στην αξιολόγηση της αποτελεσματικότητας ενός αλγορίθμου. Με την πάροδο των ετών και τη συνεχή μείωση της τιμής των

αποκομμάτων μνήμης, το κριτήριο αυτό τείνει να πάρει δευτερεύουσα σημασία, καθώς παρά την συνεχή αύξηση της υπολογιστικής ισχύος, ο χρόνος εκτέλεσης ενός προγράμματος εξακολουθεί να έχει τον κεντρικό ρόλο στην αξιολόγηση της αποτελεσματικότητάς του.

Αυτό δεν σημαίνει βέβαια πως πρέπει να αδιαφορούμε για τις απαιτήσεις μνήμης ενός προγράμματος. Η ποσότητα των ψηφιοσυλλαβών στην κεντρική μνήμη ενός υπολογιστή, αν και έχει αυξηθεί εξαιρετικά, παραμένει πεπερασμένη. Δεν πρέπει λοιπόν να τη σπαταλάμε χωρίς λόγο.

1.4.3 Χρόνος εκτέλεσης

Γενικά περιμένουμε από ένα αλγόριθμο να εκτελεί το μικρότερο δυνατό αριθμό λειτουργιών για την ολοκλήρωση του υπολογισμού. Όσο λιγότερες εντολές εκτελεί ένας αλγόριθμος τόσο πιο γρήγορα εκτελείται και κατά συνέπεια, αν η ταχύτητα είναι το κριτήριο αποτελεσματικότητας, τόσο πιο αποτελεσματικός είναι. Ας υποθέσουμε πως έχουμε διαθέσιμους δύο διαφορετικούς αλγορίθμους A και B για την επίλυση του ίδιου προβλήματος. Και οι δύο είναι αποδεδειγμένα ορθοί και οι απαιτήσεις τους σε μνήμη είναι περίπου οι ίδιες. Ποιο θα πρέπει να προτιμήσουμε; Προφανώς αυτόν που εκτελείται σε λιγότερο χρόνο. Πως ωστόσο μπορούμε να μετρήσουμε πόσο χρόνο απαιτεί ένας αλγόριθμος για την ολοκλήρωσή του, όταν είναι γνωστό το μέγεθος των δεδομένων εισόδου;

Εμπειρική μελέτη

Ο πιο απλός, αν και κάπως παρεξηγημένος, τρόπος είναι να εκτελέσουμε πειράματα. Μπορούμε να επιλέξουμε ένα σύνολο δεδομένων εισόδου, να εκτελέσουμε τα δύο προγράμματα για το συγκεκριμένο σύνολο δεδομένων εισόδου, και να μετρήσουμε το χρόνο που χρειάστηκε καθένα για να ολοκληρώσει τη δουλειά. Υπάρχουν αρκετά αντεπιχειρήματα σε αυτή την προσέγγιση.

Αν τη στιγμή που έτρεχε ο αλγόριθμος A, το λειτουργικό σύστημα ξεκίνησε μια προγραμματισμένη εργασία δημιουργίας αντιγράφων ασφαλείας, τότε σίγουρα η μέτρηση δεν είναι αντικειμενική. Κατά κανόνα, οι αλγόριθμοι είναι ευαίσθητοι στα δεδομένα εισόδου. Για μια συγκεκριμένη κατηγορία δεδομένων μπορεί να είναι

πολύ πιο γρήγοροι από ότι για μια άλλη. Πως εξασφαλίζουμε ότι τα δεδομένα που επιλέξαμε για ένα πείραμα είναι αφενός ρεαλιστικά και αφετέρου αντικειμενικά; Ένας αλγόριθμος μπορεί να είναι εξαιρετικά γρήγορος όταν το μέγεθος των δεδομένων εισόδου είναι μικρό, αλλά η ταχύτητά του να μειώνεται δραματικά όταν το μέγεθος των δεδομένων εισόδου μεγαλώνει. Συχνά επίσης συμβαίνει και το αντίστροφο αν και κάτι τέτοιο φαίνεται παράλογο εκ πρώτης όψεως.

Τα παραπάνω δεν θα πρέπει να μας οδηγήσουν στο συμπέρασμα ότι η πειραματική μελέτη της ταχύτητας αλγορίθμων δεν είναι αξιόπιστη. Αντίθετα σε πολλές περιπτώσεις είναι εξαιρετικά σημαντική. Για το λόγο αυτό προτείνουμε μερικούς τρόπους για τον περιορισμό της επίδρασης των παραπάνω παραγόντων.

i). Πρέπει να εκτελούμε πειράματα σε συνεχώς αυξανόμενα σε μέγεθος δεδομένα εισόδου.

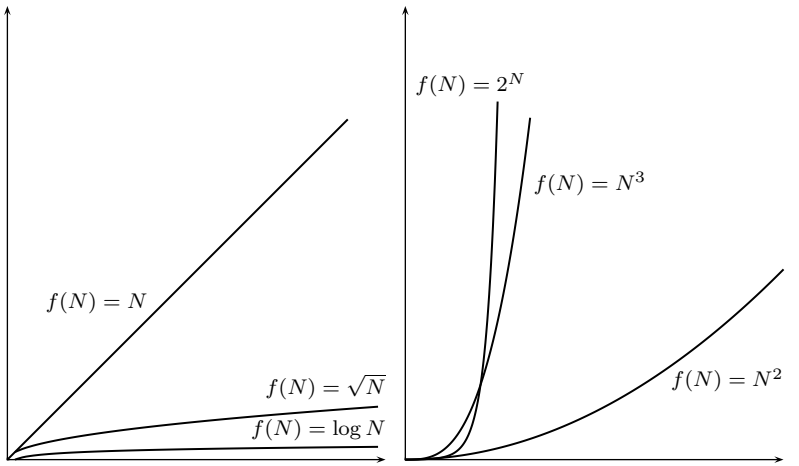
ii). Για κάθε συγκεκριμένο μέγεθος των δεδομένων εισόδου πρέπει να δημιουργούμε ένα σύνολο από διαφορετικά, τυχαία δεδομένα.

iii). Πρέπει να εκτελούμε κάθε πρόγραμμα περισσότερες από μία φορές για κάθε σύνολο δεδομένων εισόδου και να υπολογίζουμε το μέσο χρόνο που χρειάστηκε για να ολοκληρώσει την επεξεργασία.

iv). Πρέπει πάντα να φροντίζουμε πως το σύνολο των δεδομένων εισόδου είναι αντικειμενικό, πως δεν ευνοεί κανένα από τα προγράμματα που συγκρίνουμε.

Ο σχεδιασμός ενός αξιόπιστου πειράματος απαιτεί αρκετή προσπάθεια και ορθή μεθοδολογική προσέγγιση. Σε μερικές περιπτώσεις ωστόσο, η διαφορά ταχύτητας μεταξύ δύο αλγορίθμων είναι τόσο μεγάλη ώστε η εκτέλεση πειραμάτων είναι περιττή. Αν γνωρίζουμε για παράδειγμα ότι σε δεδομένα εισόδου μεγέθους N, ο αλγόριθμος A εκτελεί κατά μέσο όρο $10 \ln N$ λειτουργίες ενώ ο αλγόριθμος B εκτελεί $\left(N^2 - N\right)/2$ τότε μια γρήγορη ματιά στο Σχήμα 1-2 αρκεί για να μας πείσει πως ο πρώτος είναι ταχύτερος εφόσον το N είναι αρκετά μεγάλο (βλέπε Άσκηση 1-27).

Το Σχήμα 1-2 παρουσιάζει μερικές συναρτήσεις του N που εμφανίζονται πολύ συχνά στην ανάλυση αλγορίθμων. Κάθε μία από τις απεικονιζόμενες συναρτήσεις ορίζει μια *κλάση αποτελεσματικότητας αλγορίθμων*. Συνήθως λέμε πως ένα πρόγραμμα εκτελείται σε *λογαριθμικό* χρόνο σε σχέση με το πλήθος των δεδομένων εισόδου,



Σχήμα 1-2: Ρυθμός αύξησης συναρτήσεων.
Το σχήμα παρουσιάζει μερικές συναρτήσεις που εμφανίζονται πολύ συχνά στην ανάλυση αλγορίθμων. Στο αριστερό μέρος του σχήματος παρουσιάζονται οι συναρτήσεις που αυξάνουν με σχετικά αργό ρυθμό καθώς αυξάνει το μέγεθος του προβλήματος. Αντίθετα, στο δεξί μέρος του σχήματος, παρουσιάζονται συναρτήσεις που αυξάνουν με ταχύτατο ρυθμό. Για τους σκοπούς της παρουσίασης, η κλίμακα στον κάθετο άξονα είναι 1 στο αριστερό γράφημα, και 0.01 στο δεξί.

όταν το πλήθος των λειτουργιών που εκτελεί δίνεται από μια συνάρτηση της μορφής $f(N) = c \log N$. Αντίστοιχα χαρακτηρίζουμε το χρόνο εκτέλεσης ενός προγράμματος ως *υπογραμμικό*, *γραμμικό*, *τετραγωνικό*, *κυβικό* ή *εκθετικό* σε σχέση πάντα με το μέγεθος των δεδομένων εισόδου.

Θεωρητική μελέτη

Το ερώτημα ωστόσο είναι πως μπορούμε να εφαρμόσουμε αναλυτικές μεθόδους για να προσδιορίσουμε τη συνάρτηση που δίνει τον αριθμό των στοιχειωδών λειτουργιών ενός αλγορίθμου, ως συνάρτηση του μεγέθους των δεδομένων εισόδου. Για το σκοπό αυτό θα παρουσιάσουμε μερικά παραδείγματα ανάλυσης του χρόνου εκτέλεσης ορισμένων πολύ γνωστών αλγορίθμων.

Ακολουθιακή αναζήτηση. Η υλοποίηση του αλγορίθμου έχει ήδη παρουσιαστεί στον Κώδικα 1.2 στη σελίδα 16, όπου η παράμετρος a παριστάνει την ακολουθία a και η παράμετρος x το ζητούμενο στοιχείο x. Θα προσπαθήσουμε να ποσοτικοποιήσουμε τον χρόνο εκτέλεσης της υλοποίησης αυτής, αναλύοντας τον αριθμό των συγκρίσεων που εκτελεί η μέθοδος `index0f` στη γραμμή 4. Αν $a_0 = x$, η μέθοδος `index0f` εκτελεί

$$B(N) = 1$$

συγκρίσεις. Αυτή είναι και η *καλύτερη περίπτωση* (best case), η περίπτωση δηλαδή στην οποία η μέθοδος εκτελεί το μικρότερο δυνατό αριθμό συγκρίσεων και κατά συνέπεια έχει το μικρότερο χρόνο εκτέλεσης.

Όταν $a_{N-1} = x$ ή όταν $a_i \neq x$ για κάθε $i \in [0, N)$, η μέθοδος απαιτεί N συγκρίσεις. Κατά συνέπεια στη *χειρότερη περίπτωση* (worst case) η μέθοδος εκτελεί

$$W(N) = N$$

συγκρίσεις. Η ανάλυσή μας θα ήταν σαφέστατα χρησιμότερη αν μπορούσαμε να εκφράσουμε τον *αναμενόμενο αριθμό συγκρίσεων* διατυπώνοντας έτσι μια γενικότερη (και ακριβέστερη) εκτίμηση για τον χρόνο εκτέλεσης της μεθόδου στη *μέση περίπτωση* (average case).

Ας υποθέσουμε πως η πιθανότητα να υπάρχει κάποιο $i \in [0, N)$ έτσι ώστε $a_i = x$, είναι p. Κατά συνέπεια η πιθανότητα να μην υπάρχει τέτοιο στοιχείο είναι $q = 1 - p$. Θα θεωρήσουμε πως αν το x είναι στοιχείο της ακολουθίας, η πιθανότητα να βρίσκεται στη θέση i είναι $p_i = \frac{1}{N}$. Με διαφορετική διατύπωση, εφόσον το x αποτελεί στοιχείο της ακολουθίας, η πιθανότητα να βρίσκεται σε οποιαδήποτε θέση, είναι η ίδια. Αν είμαστε σίγουροι πως το x είναι στοιχείο της ακολουθίας, απαιτείται 1 σύγκριση για τον εντοπισμό του αν βρίσκεται στη θέση 0, ενώ απαιτούνται 2 συγκρίσεις αν βρίσκεται στη θέση 1, και γενικά, απαιτούνται $i + 1$ συγκρίσεις αν βρίσκεται στη θέση i. Συνεπώς ο αναμενόμενος αριθμός συγκρίσεων είναι

$$\begin{aligned} A_p(N) &= p_1 + 2p_2 + \cdots + Np_N \\ &= \frac{1}{N} + \frac{2}{N} + \cdots + \frac{N}{N} \\ &= \frac{1}{N} \sum_{k=1}^{N} k \\ &= \frac{1}{N} \frac{N+1}{2} N \\ &= \frac{N+1}{2}. \end{aligned} \tag{1.30}$$

Αν το x δεν είναι στοιχείο της ακολουθίας, απαιτούνται

$$A_q(N) = N \tag{1.31}$$

συγκρίσεις, όσες και στην χειρότερη περίπτωση. Συνδυάζοντας τις (1.30) και (1.31) με τις πιθανότητες εμφάνισης των αντίστοιχων ενδεχομένων, έχουμε τελικά τον αναμενόμενο αριθμό συγκρίσεων

$$A(N) = pA_p(N) + qA_q(N) = p\frac{N+1}{2} + qN. \qquad (1.32)$$

Είναι ενδιαφέρον να ξαναδούμε τον τρόπο με τον οποίο καταλήξαμε σε αυτό το αποτέλεσμα. Αρχικά παρατηρώντας τον Κώδικα 1.2 εντοπίσαμε τις προϋποθέσεις που πρέπει να ικανοποιούν τα δεδομένα εισόδου ώστε να έχουμε τον ελάχιστο χρόνο εκτέλεσης $B(N)$. Με ανάλογο τρόπο, τρόπο εντοπίσαμε τον μέγιστο χρόνο εκτέλεσης $W(N)$. Έχοντας πια μια βαθύτερη κατανόηση για τη λειτουργία του αλγορίθμου αλλά και για τα δεδομένα εισόδου και τον τρόπο με τον οποίο αυτά επηρεάζουν τη συμπεριφορά του αλγορίθμου, καταλήξαμε σε ένα απλό μοντέλο το οποίο λαμβάνει υπόψη του κάθε ενδεχόμενη διάταξη των δεδομένων εισόδου. Εφαρμόζοντας τέλος μερικές σχέσεις και ιδιότητες από την Ενότητα 1.2, υπολογίσαμε τον αναμενόμενο αριθμό συγκρίσεων που εκτελεί ο αλγόριθμος.

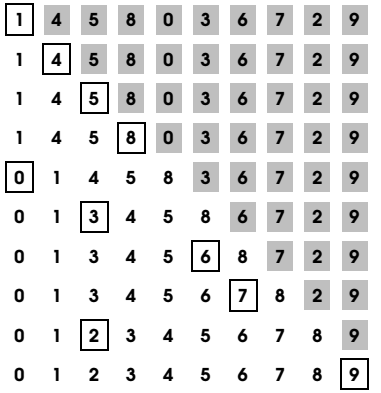
Ταξινόμηση με εισαγωγή. Το πρόβλημα της ταξινόμησης μιας ακολουθίας $N \geq 2$ στοιχείων $a_0, a_1, \ldots, a_{N-1}$ σε αύξουσα (φθίνουσα) διάταξη, συνίσταται στην παραγωγή μιας ακολουθίας $a_{p(0)}, a_{p(1)}, \ldots, a_{p(N-1)}$, έτσι ώστε $a_{p(i)} \leq a_{p(i+1)}$ (ή αντίστοιχα $a_{p(i)} \geq a_{p(i+1)}$ εφόσον πρόκειται για φθίνουσα διάταξη) για κάθε $0 \leq p(i) \leq N-2$.

Για το πρόβλημα της ταξινόμησης έχουν σχεδιαστεί διάφοροι αλγόριθμοι με διάφορα χαρακτηριστικά επίδοσης. Εδώ θα αναλύσουμε το χρόνο εκτέλεσης του αλγορίθμου *ταξινόμησης με εισαγωγή* (insertion sort). Η υλοποίηση του αλγορίθμου δίνεται στον Κώδικα 1.3.

Ο αλγόριθμος insertion sort ξεκινά χωρίζοντας την ακολουθία προς ταξινόμηση σε δύο ακολουθίες: $L = a_0$ και $R = a_1, a_2, \ldots, a_{N-1}$. Το επόμενο βήμα είναι η μεταφορά του όρου a_1 από την ακολουθία R στην ακολουθία L έτσι ώστε η τελευταία να είναι ταξινομημένη. Το βήμα αυτό λέγεται εισαγωγή του όρου a_1 στην L και για το λόγο αυτό ο συγκεκριμένος αλγόριθμος ταξινόμησης ονομάζεται ταξινόμηση με εισαγωγή. Μετά την εισαγωγή του a_1, η

S O R T I N G E X A M P L E
A E E G I L M N O P R S T X
X T S R P O N M L I G E E A

Σχήμα 1-3: Ταξινόμηση ακολουθίας σε αύξουσα και φθίνουσα διάταξη.
Το σχήμα παρουσιάζει το αποτέλεσμα της ταξινόμησης της ακολουθίας **S O R T I N G E X A M P L E** σε αύξουσα και φθίνουσα διάταξη.



Σχήμα 1-4: Ο αλγόριθμος insertion sort. Το σχήμα παρουσιάζει τον τρόπο με τον οποίο λειτουργεί ο αλγόριθμος insertion sort για την ταξινόμηση της ακολουθίας ακεραίων **1 4 5 8 0 3 6 7 2 9**. Η επάνω διαγώνιος είναι η ακολουθία R ενώ η κάτω διαγώνιος είναι η ταξινομημένη ακολουθία L.

L έχει 2 στοιχεία ενώ η R, $N-2$. Το βήμα της εισαγωγής επαναλαμβάνεται για τους όρους $a_2, \ldots, a_{N-1}$ μέχρι η R να εκφυλιστεί σε κενή ακολουθία. Τότε η L θα είναι ταξινομημένη.

Κώδικας 1.3: Ταξινόμηση με εισαγωγή.

```
1  public void sort(int[] a) {
2     for (int i = 1; i < a.length; ++i) {
3        int x = a[i];
4        int j = i - 1;
5        while (j >= 0 && a[j] > x) {
6           a[j + 1] = a[j];
7           j = j - 1;
8        }
9        a[j + 1] = x;
10    }
11 }
```

Θα αναλύσουμε την αποτελεσματικότητα του αλγορίθμου insertion sort υπολογίζοντας το πλήθος των συγκρίσεων που εκτελεί (βρόχος `while` στην γραμμή 9). Για να απλουστεύσουμε την ανάλυσή μας, θα θεωρήσουμε ότι τα στοιχεία της συστοιχίας είναι διακριτά, καμία τιμή δηλαδή, δεν εμφανίζεται περισσότερο από μία φορά. Αν τα στοιχεία είναι αρχικά σε φθίνουσα διάταξη, η συνθήκη `a[j] > x` θα είναι αληθής κάθε φορά που θα εκτελείται ο βρόχος `while`. Αυτό σημαίνει ότι για κάθε τιμή i της μεταβλητής `i` στο διάστημα $[1, N-1]$, $N =$ `a.length`, ο βρόχος `while` θα εκτελείται i φορές (για τις τιμές της μεταβλητής `j` από 0 μέχρι και $i-1$). Κατά συνέπεια ο αριθμός των συγκρίσεων στην περίπτωση αυτή θα είναι

$$W(N) = \sum_{k=1}^{N-1} k = \frac{1+(N-1)}{2}(N-1) = \frac{N(N-1)}{2}. \qquad (1.33)$$

Προφανώς πρόκειται για την χειρότερη περίπτωση. Με ανάλογο τρόπο, εύκολα συμπεραίνουμε ότι στην καλύτερη περίπτωση, όταν δηλαδή τα στοιχεία της συστοιχίας είναι ήδη σε αύξουσα διάταξη, ο βρόχος `while` θα εκτελείται μία και μόνο φορά για κάθε τιμή της μεταβλητής `i`. Κατά συνέπεια ο Κώδικας 1.3 σε αυτή την περίπτωση θα εκτελέσει $N-1$ συγκρίσεις, και επομένως

$$B(N) = N - 1. \qquad (1.34)$$

Δεν μπορούμε φυσικά να περιμένουμε οι δύο πολύ ειδικές αυτές περιπτώσεις να εμφανίζονται συχνά στην πραγματικότητα. Σημαντικό πρακτικό ενδιαφέρον αντίθετα έχουν οι περιπτώσεις κατά τις οποίες τα στοιχεία της ακολουθίας είναι διατεταγμένα με τυχαίο τρόπο.

Παρατηρούμε ότι ο αριθμός των συγκρίσεων στον βρόχο `while` εξαρτάται από την θέση στην οποία θα πρέπει να μεταφερθεί στο στοιχείο `a[i]` σε κάθε εκτέλεση του βρόχου `for`. Για κάθε τιμή i της μεταβλητής `i`, υπάρχουν $i+1$ θέσεις στις οποίες ενδέχεται να μεταφερθεί τελικά το στοιχείο `a[i]`· οι θέσεις από 0 ως $i-1$. Θα θεωρήσουμε ότι κάθε θέση έχει την ίδια πιθανότητα $p = \frac{1}{i+1}$ να αποτελεί τη θέση εισαγωγής. Το πλήθος των συγκρίσεων που πρέπει να εκτελέσει ο αλγόριθμος προκειμένου να εντοπίσει κάθε θέση εισαγωγής δίνεται στον παρακάτω πίνακα.

Θέση Εισαγωγής	0	1	2	3	$\ldots$	$i-2$	$i-1$	i
Αριθμός συγκρίσεων	i	i	$i-1$	$i-2$	$\ldots$	3	2	1

Παρατηρούμε ότι για την εισαγωγή στη θέση 0 απαιτούνται i συγκρίσεις, όσες δηλαδή απαιτούνται και για την εισαγωγή στη θέση 1. Αυτή η μικρή ασυμμετρία παρατηρείται διότι αν το `a[i]` πρέπει να εισαχθεί στη θέση 0, ο βρόχος `while` θα τερματίσει λόγω της συνθήκης `j >= 0`, χωρίς (ευτυχώς) να προλάβει να επιχειρήσει μία επιπλέον σύγκριση.

Με βάση όλα αυτά μπορούμε να υπολογίσουμε τον αναμενόμενο αριθμό συγκρίσεων C_i που απαιτούνται για τον εντοπισμό της θέσης εισαγωγής σε μια ακολουθία μήκους i.

$$\begin{aligned} C_i &= \frac{1}{i+1} + \frac{2}{i+1} + \cdots + \frac{i}{i+1} + \frac{i}{i+1} \\ &= \frac{1}{i+1} \sum_{j=1}^{i} j + \frac{i}{i+1} \\ &= \frac{1}{i+1} \frac{i+1}{2} i + \frac{i}{i+1} = \frac{i}{2} + \frac{i}{i+1} \end{aligned}$$

Ο αναμενόμενος αριθμός συγκρίσεων $A(N)$ για την ταξινόμηση της

συστοιχίας δίνεται από το άθροισμα των C_i για $1 \leq i \leq N-1$.

$$\begin{aligned} A(N) &= \sum_{i=1}^{N-1} C_i \\ &= \sum_{i=1}^{N-1} \left(\frac{i}{2} + \frac{i}{i+1} \right) \\ &= \frac{1}{2} \sum_{i=1}^{N-1} i + \sum_{i=1}^{N-1} \frac{i}{i+1} \\ &= \frac{1}{2} \frac{N(N-1)}{2} + \sum_{i=1}^{N-1} \left(1 - \frac{1}{i+1} \right) \\ &= \frac{N^2}{4} + \frac{3N}{4} - 1 - \sum_{i=1}^{N-1} \frac{1}{i+1} \end{aligned}$$

Από την (1.14) γνωρίζουμε ότι

$$\sum_{i=1}^{N-1} \frac{1}{i+1} \approx \ln N$$

και κατά συνέπεια, ο αναμενόμενος αριθμός συγκρίσεων στοιχείων που απαιτεί ο Κώδικας 1.3 για την ταξινόμηση μιας συστοιχίας N στοιχείων, είναι

$$A(N) \approx \frac{N^2}{4} + \frac{3N}{4} - \ln N - 1. \tag{1.35}$$

Ασκήσεις

▷ **1-25** Τροποποιείστε τον Κώδικα 1.2 ώστε να εκτελείται ταχύτερα. *Υπόδειξη*: μπορούμε να εξαλείψουμε τον έλεγχο `i < a.length` στη γραμμή 3.

1-26 Ο αναμενόμενος αριθμός λειτουργιών που εκτελούν δύο αλγόριθμοι A και B είναι αντίστοιχα $N^2 \ln N$ και N^3. Ποιος από τους δύο αναμένεται ταχύτερος;

1-27 Για την επίλυση ενός συγκεκριμένου υπολογιστικού προβλήματος, έχουμε στη διάθεση μας δύο αλγορίθμους A και B οι οποίοι εκτελούν $144N^2 \ln N$ και $\left(N^3 - 3N^2 + 3N\right)/4$ λειτουργίες αντίστοιχα,

αν το μέγεθος του προβλήματος είναι N. Ποιον από τους δύο αλγορίθμους πρέπει να επιλέξουμε;

▷ **1-28** Θέλουμε να αναζητούμε στοιχεία σε συστοιχίες ταξινομημένες σε αύξουσα διάταξη. Γράψτε ένα αλγόριθμο ακολουθιακής αναζήτησης ο οποίος έχει μικρότερο αναμενόμενο αριθμό συγκρίσεων από αυτόν του Κώδικα 1.2.

1-29 Ο αλγόριθμος ταξινόμησης με επιλογή (selection sort), λειτουργεί με τον ακόλουθο τρόπο για την ταξινόμηση μιας ακολουθίας ακεραίων $a_0, \ldots, a_{N-1}$: υπολογίζει το $\max(a_0, \ldots, a_{N-1})$ και το ανταλλάσσει με το a_{N-1}. Στη συνέχεια υπολογίζει το $\max(a_0, \ldots, a_{N-2})$ και το ανταλλάσσει με το a_{N-2}, και συνεχίζει με αυτό τον τρόπο μέχρι η ακολουθία να είναι ταξινομημένη (βλέπε τον αλγόριθμο 7-3 στη σελίδα 198). Υλοποιήστε τον αλγόριθμο σε Java, και στη συνέχεια υπολογίστε το πλήθος συγκρίσεων που απαιτεί για την ταξινόμηση μιας συστοιχίας N στοιχείων.

1-30 Ο αλγόριθμος ταξινόμησης φυσαλίδας (bubble sort) βασίζεται σε μια πολύ απλή ιδέα. Εκτελεί συνεχή περάσματα από τα αριστερά προς τα δεξιά και ανταλλάσσει κάθε ζεύγος στοιχείων που βρίσκεται σε λάθος διάταξη. Ο αλγόριθμος τερματίζει όταν μετά από ένα πέρασμα διαπιστώσει πως δεν έχει γίνει καμία ανταλλαγή. Υλοποιήστε τον αλγόριθμο σε Java, και στη συνέχεια υπολογίστε το πλήθος συγκρίσεων που απαιτεί για την ταξινόμηση μιας συστοιχίας N στοιχείων.

1-31 Μια παραλλαγή του αλγορίθμου ταξινόμησης bubble sort, γνωστή και ως shake sort, εκτελεί ένα πέρασμα από τα αριστερά προς τα δεξιά ακολουθούμενο από ένα πέρασμα από τα δεξιά προς τα αριστερά. Υλοποιείστε τον αλγόριθμο ταξινόμησης shake sort.

1.4.4 Ασυμπτωτική ανάλυση

Στην προηγούμενη ενότητα υπολογίσαμε τον αριθμό των συγκρίσεων που εκτελεί στη μέση περίπτωση ο αλγόριθμος insertion sort. Παρατηρούμε ωστόσο, ότι εκτός από συγκρίσεις ακεραίων, ο αλγόριθμος αυτός εκτελεί και άλλες λειτουργίες, τον αριθμό των οποίων δεν υπολογίσαμε. Με άλλα λόγια το μέτρο μας δεν είναι και τόσο ακριβές καθώς δεν λαμβάνει υπόψη του τη συνολική συμπεριφορά του αλγορίθμου. Ο λόγος για τον οποίο υιοθετούμε αυτή τη στάση είναι κυρίως πρακτικός. Δεν είναι πάντα εύκολο να προβαίνουμε σε συνολική ανάλυση ακόμη και για απλούς αλγόριθμους. Για το λόγο

αυτό επιλέγουμε να υπολογίσουμε τον αριθμό των λειτουργιών εκείνων που αποτελούν το βασικό υπολογιστικό όπλο ενός αλγορίθμου.

Από την άλλη πλευρά, ακόμη και αν αφιερώναμε κάθε φορά τον απαιτούμενο χρόνο, και πάλι δεν θα ήμασταν σε θέση να έχουμε μια απόλυτα ακριβή εκτίμηση του χρόνου εκτέλεσης του αλγορίθμου insertion sort. Κι αυτό διότι δεν είμαστε σε θέση να γνωρίζουμε πόσο χρόνο απαιτεί η εκτέλεση μιας σύγκρισης ή μιας ανάθεσης σε κάθε υπολογιστικό σύστημα. Επίσης δεν μπορούμε να ξέρουμε τη θέση στην οποία θα βρίσκεται η μεταβλητή `i`, κατά τη στιγμή της εκτέλεσης των αντίστοιχων εντολών μηχανής. Αν βρίσκεται σε ένα καταχωρητή, τότε ή αύξηση θα απαιτεί σημαντικά λιγότερο χρόνο από αυτόν που θα απαιτούσε αν η `i` βρισκόταν στην κεντρική μνήμη. Μπορούμε βέβαια να θεωρήσουμε ότι αν μια στοιχειώδης λειτουργία απαιτεί χρόνο t_A στο υπολογιστικό σύστημα A, και t_B στο υπολογιστικό σύστημα B, τότε

$$t_B = ct_A \text{ και, αντίστοιχα, } t_A = \frac{t_B}{c}$$

όπου η σταθερά c είναι μια εκτίμηση της σχετικής ταχύτητας των δύο υπολογιστικών συστημάτων.

Με αυτή την υπόθεση, μπορούμε να απεμπλακούμε από τις κατασκευαστικές λεπτομέρειες και ταυτόχρονα να έχουμε μια αξιόπιστη προσέγγιση για την ταχύτητα εκτέλεσης των αλγορίθμων μας. Βεβαίως, ακόμη και έτσι, παραμένει ένα πρόβλημα να λύσουμε. Είμαστε πάντα σε θέση να γνωρίζουμε με ακρίβεια τις εντολές που εκτελεί ένας αλγόριθμος; Η προφανής απάντηση είναι, ναι, μπορούμε. Μια πιο προσεκτική ματιά θα δείξει ωστόσο πως τα πράγματα δεν είναι ακριβώς έτσι, εκτός κι αν αποφασίσουμε να γράφουμε τα προγράμματά μας σε γλώσσα μηχανής. Μεταξύ του αλγορίθμου μας και της εκτέλεσης των αντίστοιχων εντολών από ένα υπολογιστή μεσολαβούν μεταγλωττιστές, βελτιστοποιητές και άλλα προγράμματα που μετατρέπουν τον πηγαίο κώδικα σε κώδικα μηχανής. Είμαστε λοιπόν σε θέση να γνωρίζουμε ακριβώς τον τρόπο με τον οποίο ο μεταγλωττιστής ενός συγκεκριμένου κατασκευαστή π.χ., της Sun Microsystems, Inc., θα υλοποιήσει τους βρόχους `for` και `while` στον Κώδικα 1.3; Προφανώς όχι.

Η παραπάνω ιδέα ωστόσο μπορεί να μας φανεί χρήσιμη προκειμένου να μην είμαστε αναγκασμένοι να προχωρούμε κάθε φορά σε

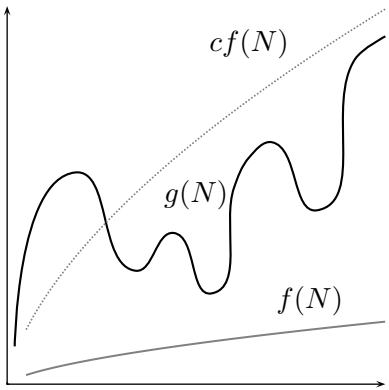
εξαντλητική ανάλυση όλων των λειτουργιών ενός αλγορίθμου. Μπορούμε εύλογα να ισχυριστούμε ότι αν $A(N)$ είναι ο αναμενόμενος αριθμός των βασικών λειτουργιών, που θα εκτελέσει ένα πρόγραμμα, τότε μπορούμε να πούμε ότι $cA(N)$ είναι μια καλή εκτίμηση του συνολικού αριθμού εντολών, εφόσον το N είναι αρκετά μεγάλο. Στην περίπτωση του insertion sort θεωρήσαμε πως η σύγκριση αποτελεί τη βασική λειτουργία, και δεδομένου πως απαιτούνται $\frac{N^2}{4} + \frac{3N}{4} - \ln N - 1$ συγκρίσεις, μπορούμε να θεωρήσουμε πως ο συνολικός αριθμός εντολών είναι ένα μικρό πολλαπλάσιο της ποσότητας αυτής.

Επιπλέον όταν το μέγεθος του προβλήματος είναι μεγάλο, όπως για παράδειγμα η ταξινόμηση μιας συστοιχίας 10.000.000 στοιχείων, ο όρος $\frac{N^2}{4}$ θα πρέπει να τραβήξει την προσοχή μας πολύ περισσότερο από τον όρο $\frac{3N}{4}$ (βλέπε 1.40 στη σελίδα 1.40). Η (1.35) μας λέει πως ο αλγόριθμος insertion sort είναι ένας τετραγωνικός αλγόριθμος. Ο αριθμός των στοιχειωδών λειτουργιών που εκτελεί κατά την ταξινόμηση μιας συστοιχίας N ακεραίων είναι ανάλογος του N^2 και κατά συνέπεια το ίδιο και ο χρόνος εκτέλεσής του. Παρά το γεγονός πως δεν γνωρίζουμε ακριβώς τον απαιτούμενο χρόνο εκτέλεσης του insertion sort, έχουμε μια καλή εκτίμηση στη βάση της οποίας μπορούμε να συγκρίνουμε την ταχύτητά του με αυτή άλλων αλγορίθμων ταξινόμησης.

Ασυμπτωτικές προσεγγίσεις. Καθώς είναι εξαιρετικά χρήσιμο να έχουμε ένα αρκετά αυστηρό τρόπο για την διατύπωση των παραπάνω διαισθητικών συμπερασμάτων, χρειαζόμαστε ένα φορμαλισμό, ένα τυπικό συμβολισμό, για να συγκρίνουμε συναρτήσεις ταχύτητας, ο οποίος ωστόσο δεν θα επηρεάζεται από λεπτομέρειες υλοποίησης, τουλάχιστον, όχι για μεγάλα προβλήματα.

Ορισμός 1-2. Αν $f,g : \mathbb{N} \to \mathbb{R}^+$, θα λέμε ότι η συνάρτηση g ανήκει στο σύνολο $O(f)$ και θα γράφουμε $g(N) = O\left(f(N)\right)$, αν υπάρχουν σταθερές $c \in \mathbb{R}^+$ και $n_0 \in \mathbb{N}$, τέτοιες ώστε $g(N) \leq cf(N)$ για κάθε $N \geq n_0$.

Πρέπει να επισημάνουμε την μάλλον ασυνήθιστη χρήση της ισότητας στη σχέση $g(N) = O\left(f(N)\right)$ η οποία κανονικά έπρεπε να είναι $g \in O\left(f\right)$. Πρόκειται καθαρά για θέμα συμβολισμού, το οποίο δεν



Σχήμα 1-5: Ασυμπτωτική προσέγγιση συνάρτησης.
Η συνάρτηση $g(N)$ είναι ο χρόνος εκτέλεσης ενός αλγορίθμου για ένα πρόβλημα μεγέθους N. Τη συνάρτηση αυτή προσπαθούμε να προσεγγίσουμε μέσω της συνάρτησης $f(N)$, χρησιμοποιώντας την προσέγγιση $O\left(f(N)\right)$.

επηρεάζει το πνεύμα του ορισμού. Το σύνολο $O\left(f\right)$ είναι το σύνολο των συναρτήσεων g οι οποίες φράσσονται από ένα σταθερό πολλαπλάσιο της f. Ο συμβολισμός O ονομάζεται μεγάλο όμικρον (big oh). Ένας τρόπος για να δείξουμε τη σχέση $g(N) = O\left(f(N)\right)$ είναι να αποδείξουμε ότι

$$\lim_{N\to\infty} \frac{g(N)}{f(N)} = a \in \mathbb{R}^*. \tag{1.36}$$

Αν δηλαδή το όριο του λόγου της f προς g είναι μη αρνητικός πραγματικός αριθμός, τότε $g(N) = O\left(f(N)\right)$. Για παράδειγμα για τις συναρτήσεις $f(N) = 3N^2$ και $g(N) = 2n\ln N$ ισχύει $g(N) = O\left(f\left(N\right)\right)$, αλλά όχι το αντίστροφο, καθώς

$$\lim_{N\to\infty} \frac{g(N)}{f(N)} = \lim_{N\to\infty} \frac{2N\ln N}{3N^2} = \lim_{N\to\infty} \frac{2\ln N}{3N} = \lim_{N\to\infty} \frac{\frac{2}{N}}{3} = 0.$$

ενώ

$$\lim_{N\to\infty} \frac{f(N)}{g(N)} = \infty.$$

Ορισμός 1-3. Αν $f, g : \mathbb{N} \to \mathbb{R}^+$, θα λέμε ότι η συνάρτηση g ανήκει στο σύνολο $\Omega(f)$ και θα γράφουμε $g(N) = \Omega(f(N))$, αν υπάρχουν σταθερές $c \in \mathbb{R}^+$ και $n_0 \in \mathbb{N}$, τέτοιες ώστε $g(N) \geq cf(N)$ για κάθε $N \geq n_0$.

Το $\Omega(f)$ είναι το σύνολο των συναρτήσεων g οι οποίες αυξάνουν τουλάχιστον όσο γρήγορα αυξάνεται η f. Ο συμβολισμός $\Omega(f)$ ονομάζεται μεγάλο ωμέγα (big omega) και ένας εναλλακτικός τρόπος για να αποδείξουμε ότι $g(N) = \Omega(f(N))$ είναι να δείξουμε ότι

$$\lim_{N\to\infty} \frac{g(N)}{f(N)} \neq 0. \tag{1.37}$$

Αν δηλαδή το όριο του λόγου της g προς f δεν είναι 0, τότε $g(N) = \Omega\left(f(N)\right)$. Για παράδειγμα, επειδή

$$\lim_{N\to\infty} \frac{N^3}{N^2 - 2n + 1} = \infty$$

συμπεραίνουμε πως

$$N^3 = \Omega\left(N^2 - 2n + 1\right).$$

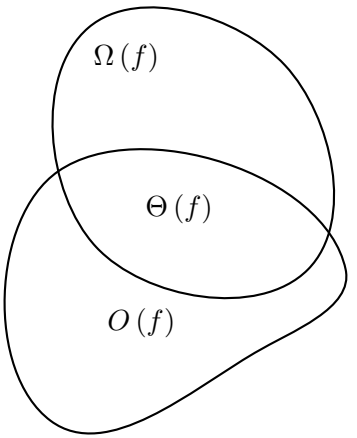
Από τους ορισμούς 1-2 και 1-3 προκύπτει εύκολα ότι

$$g(N) = O(f(N)) \Leftrightarrow f(N) = \Omega(g(N)) \tag{1.38}$$

Ορισμός 1-4. Αν $f, g : \mathbb{N} \to \mathbb{R}^+$, θα λέμε ότι η συνάρτηση g ανήκει στο σύνολο $\Theta(f)$ και θα γράφουμε $g(N) = \Theta(f(N))$, αν υπάρχουν σταθερές c_1, $c_2 \in \mathbb{R}^+$ και $n_0 \in \mathbb{N}$, τέτοιες ώστε $c_1 f(N) \leq g(N) \leq c_2 f(N)$ για κάθε $N \geq n_0$.

Ο παραπάνω ορισμός υπονοεί ότι το $\Theta(f)$ είναι το σύνολο των συναρτήσεων g οι οποίες αυξάνουν το ίδιο γρήγορα με την f. Ο συμβολισμός αυτός ονομάζεται μεγάλο θήτα (big theta). Από τους ορισμούς 1-2 και 1-3 προκύπτει εύκολα ότι $\Theta(f) = O(f) \cap \Omega(f)$. Ένας πιο σύντομος τρόπος για να δείξουμε ότι $g(N) = \Theta(f(N))$ είναι να αποδείξουμε ότι

$$\lim_{N\to\infty} \frac{g(N)}{f(N)} = c \in \mathbb{R}^+. \tag{1.39}$$



Σχήμα 1-6: Τα σύνολα O, Ω και Θ. Το σύνολο $O(f)$ περιλαμβάνει τις συναρτήσεις οι οποίες αυξάνουν με ρυθμό όχι μεγαλύτερο από αυτόν της συνάρτησης f. Το σύνολο $\Omega(f)$ περιλαμβάνει τις συναρτήσεις οι οποίες αυξάνουν με ρυθμό μεγαλύτερο από αυτόν της συνάρτησης f. Το σύνολο $\Theta(f)$ είναι το σύνολο των συναρτήσεων οι οποίες αυξάνουν το ίδιο γρήγορα με την f.

Ιδιότητες των ασυμπτωτικών προσεγγίσεων. Οι συμβολισμοί O, Θ και Ω είναι ανακλαστικοί και μεταβατικοί. Ο συμβολισμός Θ είναι συμμετρικός σε αντίθεση με τους συμβολισμούς O και Ω οι οποίοι είναι αντισυμμετρικοί. Τέλος πολύ χρήσιμη στην ασυμπτωτική ανάλυση αποδεικνύεται η σχέση

$$O(g(N) + f(N)) = O(\max\{g(N), f(N)\}). \tag{1.40}$$

Οι Ορισμοί 1-2, 1-3 και 1-4, σε συνδυασμό με τις παραπάνω ιδιότητες, μας παρέχουν την ευκαιρία να συγκρίνουμε το χρόνο εκτέλεσης αλγορίθμων χωρίς να χρειάζεται να καταφεύγουμε σε εξαντλητικές λεπτομέρειες υλοποίησης ενώ ταυτόχρονα διατηρούμε ένα απόλυτα ικανοποιητικό βαθμό αυστηρότητας. Έτσι μπορούμε να πούμε πως ο αναμενόμενος χρόνος εκτέλεσης του Κώδικα 1.2 είναι $O(N)$, ενώ για τον Κώδικα 1.3 ο χρόνος αυτός είναι $O(N^2)$.

Ασκήσεις

1-32 Διατυπώστε με αυστηρό τρόπο, την ανακλαστικότητα, τη συμμετρία, αντισυμμετρία και τη μεταβατικότητα, των ασυμπτωτικών προσεγγίσεων O, Ω, Θ.

1-33 Δώστε το χρόνο εκτέλεσης των αλγορίθμων ακολουθιακής αναζήτησης και ταξινόμησης με εισαγωγή, στην καλύτερη και χειρότερη περίπτωση, χρησιμοποιώντας τις ασυμπτωτικές προσεγγίσεις O και Θ.

1-34 Υπολογίστε μια εκτίμηση για το μέγιστο μέγεθος N, ενός προβλήματος που μπορούμε να επιλύσουμε σε ένα υπολογιστή που εκτελεί 10^9 εντολές το δευτερόλεπτο, έχοντας στη διάθεσή μας τρεις αλγορίθμους που εκτελούν αντίστοιχα $\Theta\left(2^N\right)$, $\Theta\left(N^3\right)$, $\Theta\left(\ln N\right)$ εντολές για τη λύση προβλήματος.

1-35 Αποδείξτε τις παρακάτω σχέσεις.

(i) $f\left(N\right) = O\left(f\left(N\right)\right)$.
(ii) $cO\left(f\left(N\right)\right) = O\left(f\left(N\right)\right)$.
(iii) $O\left(cf\left(N\right)\right) = O\left(f\left(N\right)\right)$.
(iv) $O\left(f\left(N\right)\right)O\left(g\left(N\right)\right) = O\left(f\left(N\right)g\left(N\right)\right)$.

1-36 Αποδείξτε ότι $F_N = \Theta\left(\phi^N\right)$.

1.5 Αναδρομικοί αλγόριθμοι

Οι αναδρομικοί ορισμοί παρέχουν ένα δηλωτικό τρόπο περιγραφής ενός υπολογισμού. Οι αλγόριθμοι από την άλλη πλευρά, αποτελούν κι αυτοί με τη σειρά τους οδηγίες για την εκτέλεση ενός υπολογισμού. Σε σύγκριση με τους αναδρομικούς ορισμούς, οι αλγόριθμοι είναι κατά κανόνα περισσότερο διαδικαστικοί καθώς καθορίζουν εντολές που πρέπει να εκτελεστούν με μια συγκεκριμένη σειρά. Μπορούμε ωστόσο να χρησιμοποιήσουμε την τεχνική της αναδρομής για να σχεδιάσουμε αναδρομικούς αλγορίθμους.

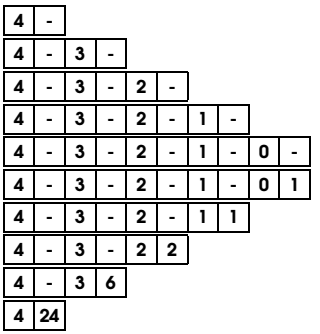
Ένας αναδρομικός αλγόριθμος είναι ένας αλγόριθμος ο οποίος επιλύει ένα πρόβλημα χρησιμοποιώντας τον εαυτό του προκειμένου να επιλύσει ένα ή και περισσότερα μικρότερα μέρη του ίδιου ακριβώς προβλήματος κάθε φορά, μέχρις ότου να επιτευχθεί μια συνθήκη για την οποία το αποτέλεσμα του υπολογισμού είναι γνωστό χωρίς αναδρομική κλήση.

Η υλοποίηση αναδρομικών αλγορίθμων επιτυγχάνεται με αναδρομικές μεθόδους. Πριν αναλύσουμε τη μηχανική των αναδρομικών αλγορίθμων ας δούμε ένα παράδειγμα αναδρομικής μεθόδου για τον υπολογισμό του παραγοντικού ενός φυσικού αριθμού.

Κώδικας 1.4: Αναδρομικός υπολογισμός του $N!$.

```
public static final long factorial(int n) {
   if (n == 0) return 1;
   return n * factorial(n - 1);
}
```

Η αναδρομική βάση του αλγορίθμου, είναι η τιμή 0 για την τυπική παράμετρο `n`. Στην περίπτωση αυτή η λύση του προβλήματος είναι γνωστή και δεν απαιτείται αναδρομική κλήση. Αν η τιμή της παραμέτρου είναι μεγαλύτερη του 0, τότε η μέθοδος `factorial` θα καλέσει τον εαυτό της (αναδρομικό βήμα) με όρισμά `n-1`. Αυτή η διαδικασία θα συνεχιστεί μέχρι να γίνει μια αναδρομική κλήση με όρισμα 0. Τότε θα ενεργοποιηθεί η βάση της αναδρομής και η μέθοδος θα επιστρέψει την τιμή 1. Αυτό θα έχει ως αποτέλεσμα την επιστροφή όλων των αναδρομικών κλήσεων και τον πολλαπλασιασμό των αποτελεσμάτων μέχρι να φτάσουμε στην αρχική κλήση, η οποία θα επιστρέψει το αποτέλεσμα του υπολογισμού όπως φαίνεται στο Σχήμα 1-7.



Σχήμα 1-7: Δυναμική της στοίβας κατά τον υπολογισμό του $4!$ από την αναδρομική μέθοδο `factorial`.

Παρατηρώντας τον Κώδικα 1.4, και με βάση τα όσα έχουμε πει για την αναδρομή, μπορούμε να διατυπώσουμε τις βασικές αρχές για το σχεδιασμό ορθών αναδρομικών μεθόδων.

i) Κάθε αναδρομική μέθοδος πρέπει να έχει τουλάχιστον μία αναδρομική βάση. Πρέπει δηλαδή να λύνει τουλάχιστον μία περίπτωση του προβλήματος χωρίς να εκτελέσει αναδρομική κλήση.

ii) Κάθε αναδρομική κλήση της μεθόδου πρέπει να μειώνει το μέγεθος του προβλήματος.

iii) Τα μερικά αποτελέσματα των αναδρομικών κλήσεων πρέπει να συνδυάζονται για τον υπολογισμό του τελικού αποτελέσματος.

Ο αλγόριθμος του Ευκλείδη. Ένας από τους παλαιότερους γνωστούς αλγόριθμους είναι ο αλγόριθμος του Ευκλείδη, ο οποίος υπολογίζει το μέγιστο κοινό διαιρέτη δύο φυσικών αριθμών n και m. Ο αλγόριθμος του Ευκλείδη βασίζεται στην παρατήρηση ότι αν ο φυσικός d διαιρεί τους φυσικούς m και $n \bmod m$, τότε διαιρεί και το N (βλέπε Άσκηση 1-9) και επιπλέον αν ο m δεν διαιρεί το n τότε ο μέγιστος κοινός διαιρέτης των n και m δεν μπορεί να είναι μεγαλύτερος του $n \bmod m$. Η αναδρομική λύση του αλγορίθμου δίνεται στον Κώδικα 1.5.

Κώδικας 1.5: Ο αλγόριθμος του Ευκλείδη

```
public static final int gcd(int n, int m) {
   return m == 0 ? n : gcd(m, n % m);
}
```

Υπολογισμός του μεγίστου όρου ακολουθίας. Μπορούμε να υπολογίσουμε το μέγιστο μιας πεπερασμένης ακολουθίας πραγματικών αριθμών $a_0, a_1, \ldots, a_{N-1}$ χρησιμοποιώντας ένα αναδρομικό αλγόριθμο. Κατ' αρχήν θα μας βοηθούσε ένας αναδρομικός ορισμός για το μέγιστο όρο μιας ακολουθίας, δεδομένου ότι μπορούμε εύκολα να υπολογίσουμε το μέγιστο δύο αριθμών.

Ο μέγιστος μιας ακολουθίας $a_0, a_1, \ldots, a_{N-1}$ είναι ο μέγιστος των a_0 και του μεγίστου της ακολουθίας $a_1, a_2, \ldots, a_{N-1}$.

$$\max(a_0, \ldots, a_{N-1}) = \max(a_0, \max(a_1, \ldots, a_{N-1}))$$

Η τεχνική πρέπει να έχει αρχίσει να γίνεται οικεία. Η βάση της αναδρομής είναι η περίπτωση που η ακολουθία έχει μήκος $N = 1$ οπότε ο μέγιστος είναι ο πρώτος και μοναδικός όρος της ακολουθίας. Το αναδρομικό βήμα είναι ο υπολογισμός του μεγίστου της ακολουθίας $a_1, a_2, \ldots, a_{N-1}$. Η υλοποίηση του αλγόριθμου δίνεται στον Κώδικα 1.6.

Κώδικας 1.6: Αναδρομικός υπολογισμός του μέγιστου στοιχείου μιας συστοιχίας.

```
public static final int max(int[] a, int i) {
   if (i == a.length - 1) return a[i];
   int x = max(a, i+1);
   return a[i] > x ? a[i] : x;
}
```

Είναι προφανές ότι θα μπορούσαμε πολύ εύκολα να λύσουμε το πρόβλημα χωρίς χρήση αναδρομής όπως στον Κώδικα 1.7. Στην πραγματικότητα για κάθε αναδρομικό αλγόριθμο, μπορούμε εύκολα ή δύσκολα να σχεδιάσουμε ένα επαναληπτικό αλγόριθμο που λύνει το ίδιο πρόβλημα. Γιατί λοιπόν να χρησιμοποιούμε αναδρομή; Το βασικό πλεονέκτημα ενός αναδρομικού αλγορίθμου είναι η απλότητα και η σαφήνεια με την οποία περιγράφει τη λύση ενός προβλήματος αφαιρώντας σχεδόν όλες τις διαδικαστικές λεπτομέρειες. Σε πολλά προβλήματα, όπως θα δούμε παρακάτω, αυτή η απλότητα είναι το πιο σημαντικό εργαλείο για την κατανόηση της

φύσης του προβλήματος και την ανάλυση του αντίστοιχου αλγορίθμου.

Κώδικας 1.7: Επαναληπτικός υπολογισμός του μέγιστου στοιχείου μιας συστοιχίας.

```
public static final int max(int[] a) {
   if (a.length == 0) throw new IllegalArgumentException();
   int max = a[0];
   for (int i = 1; i < a.length; ++i) {
      if (a[i] > max) max = a[i];
   }
   return max;
}
```

Δυαδική αναζήτηση. Όταν τα στοιχεία μιας ακολουθίας $a_l, \ldots, a_r$ είναι ταξινομημένα σε αύξουσα διάταξη, μπορούμε να βρούμε τη θέση ενός στοιχείου x με τον εξής αναδρομικό αλγόριθμο: Συγκρίνουμε το στοιχείο x με το στοιχείο a_m, $m = \lfloor (l + r)/2 \rfloor$. Αν $a_m = x$ τότε η θέση του στοιχείου x είναι m. Διαφορετικά αν $a_m < x$ αναζητούμε με τον ίδιο τρόπο το x στην ακολουθία $a_{m+1}, \ldots, a_r$, ενώ αν $a_m > x$ αναζητούμε το x στην ακολουθία $a_l, \ldots, a_{m-1}$.

Μια αναδρομική υλοποίηση του αλγορίθμου αυτού που είναι γνωστός ως *δυαδική αναζήτηση*, δίνεται στον Κώδικα 1.8, ενώ ένα παράδειγμα της εκτέλεσης του αλγορίθμου δίνεται στο Σχήμα 1-9.

Κώδικας 1.8: Αναδρομική δυαδική αναζήτηση.

```
1 public static final int bsearch(int[] a, int x, int l, int r) {
2    if (l > r) return -1;
3    int m = (l + r) / 2;
4    int cmp = a[m] - x;
5    if (cmp == 0) return m;
6    if (cmp < 0) return bsearch(a, x, m+1, r);
7    return bsearch(a, x, l, m-1);
8 }
```

Ο αλγόριθμος δυαδικής αναζήτησης, μειώνει το μέγεθος του προβλήματος περίπου κατά το ήμισυ, σε κάθε αναδρομική κλήση. Υπάρχουν δύο περιπτώσεις στις οποίες δεν γίνεται αναδρομική κλήση. Όταν `a[m] == x` (γραμμή 10) και όταν `l > r` (γραμμή 3). Ο αλγόριθμος φαίνεται ελαφρά διαφορετικός από τους προηγούμενους καθώς έχει δύο αναδρομικές κλήσεις ενώ οι προηγούμενοι αναδρομικοί αλγόριθμοι που είδαμε είχαν μία, ωστόσο στην ουσία έχει

1000	1000	1000	1000	1000
1020	1020	1020	1020	1020
1050	1050	1050	1050	1050
1060	1060	1060	1060	1060
1070	1070	1070	1070	1070
1080	1080	1080	1080	1080
1098	1098	1098	1098	1098
1120	1120	1120	1120	1120
1144	1144	1144	1144	1144
1150	1150	1150	1150	1150
1189	1189	1189	1189	1189
1202	1202	1202	1202	1202
1204	1204	1204	1204	1204
1220	1220	1220	1220	1220
1230	1230	1230	1230	1230
1240	1240	1240	1240	1240
1259	1259	1259	1259	1259
1260	1260	1260	1260	1260
1270	1270	1270	1270	1270
1285	1285	**1285**	1285	1285
1414	1414	1414	**1414**	1414
1618	1618	1618	1618	**1618**
1624	1624	1624	1624	1624
1652	**1652**	1652	1652	1652
1660	1630	1630	1630	1630
1660	1660	1660	1660	1660
1700	1700	1700	1700	1700
1772	1772	1772	1772	1772
1781	1781	1781	1781	1781
1881	1881	1881	1881	1881
2302	2302	2302	2302	2302
3141	3141	3141	3141	3141

Σχήμα 1-9: Ο αλγόριθμος δυαδικής αναζήτησης.
Το σχήμα παρουσιάζει τη λειτουργία του αλγορίθμου κατά την αναζήτηση της τιμής **1618**. Η γκρι περιοχή καθορίζει την ακολουθία στην οποία περιορίζεται η αναζήτηση σε κάθε αναδρομική εκτέλεση του αλγορίθμου.

ακριβώς τον ίδιο σκοπό· να μειώσει το μέγεθος του προβλήματος κατά 1 σε κάθε βήμα. Βεβαίως εδώ η μείωση του μεγέθους είναι πολύ πιο δραστική, και για το λόγο αυτό ο αλγόριθμος είναι εξαιρετικά αποτελεσματικός. Αυτή η τεχνική είναι γνωστή ως διαίρει και βασίλευε (divide and conquer) και έχει εφαρμοστεί σε πολλά πεδία, πολύ πριν βρει εφαρμογή στη σχεδίαση και ανάλυση αλγορίθμων.

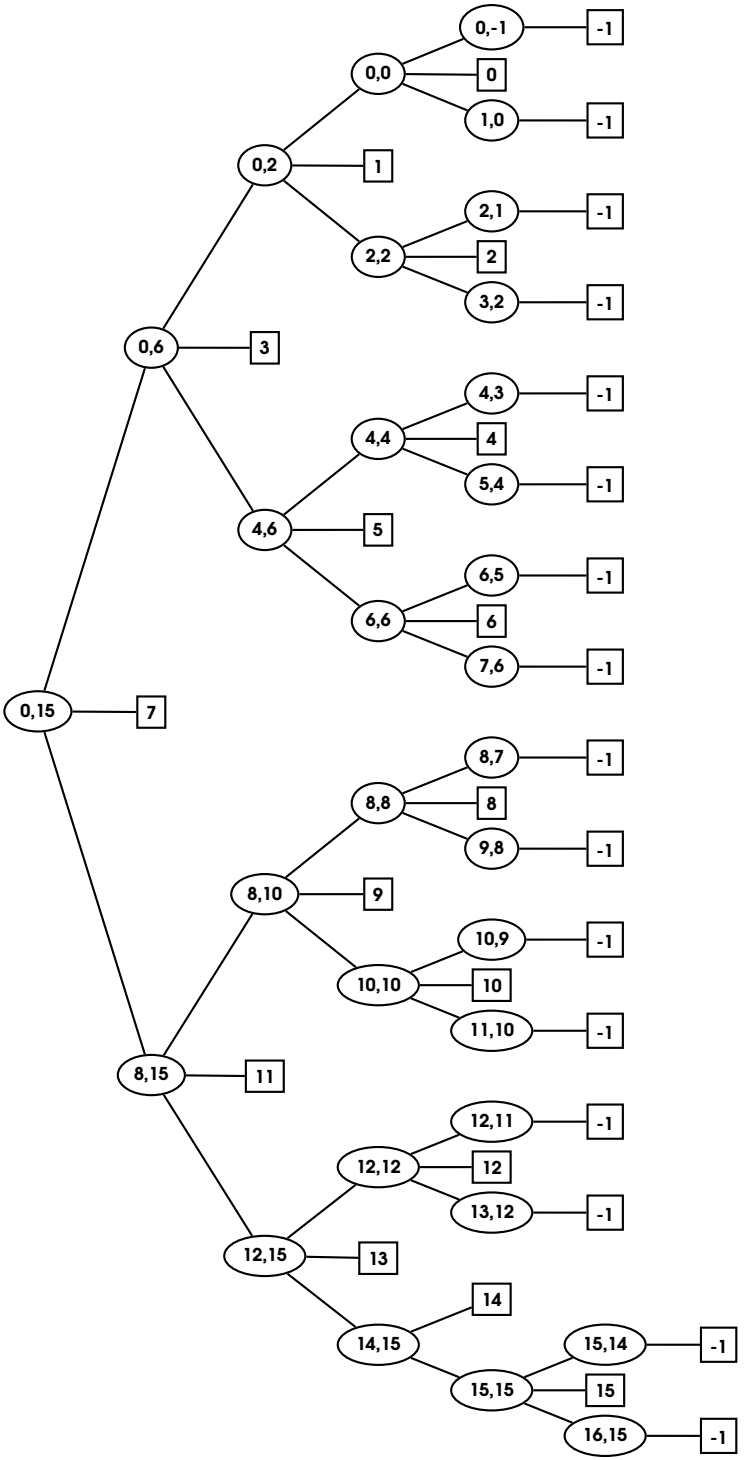
Θεώρημα 1-1. *Ο αλγόριθμος δυαδικής αναζήτησης εκτελεί το πολύ* $\lg N + 1$ *συγκρίσεις αυτικειμένων.*

Απόδειξη. α υπολογίσουμε τον αριθμό των συγκρίσεων στην χειρότερη περίπτωση, όταν δηλαδή η ζητούμενη τιμή δεν υπάρχει στην συστοιχία. Για να απλουστεύσουμε την ανάλυσή μας, θα θεωρήσουμε ότι το πλήθος των στοιχείων N είναι δύναμη του 2 δηλαδή $N = 2^k$, $k \geq 0$. Ο αριθμός των συγκρίσεων που εκτελεί ο αλγόριθμος δίνεται από την αναδρομική σχέση $C_{2^k} = C_{2^{k-1}} + 1$, $C_0 = 1$. Επιλύοντας αυτή την αναδρομική σχέση με την τηλεσκοπική μέθοδο, προκύπτει ότι

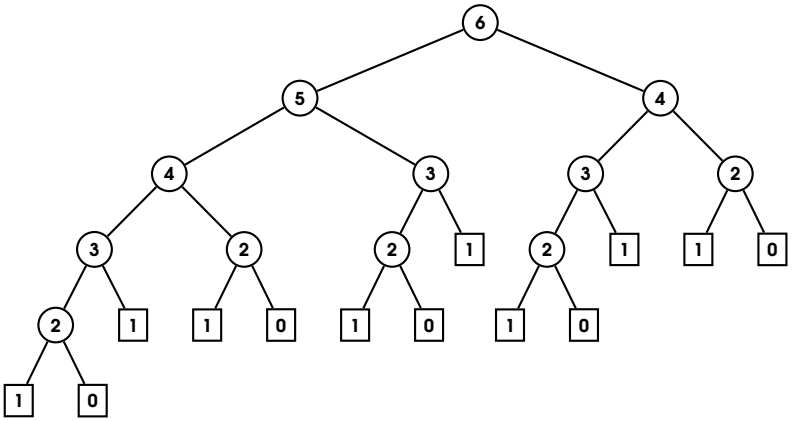
$$\begin{aligned} C_{2^k} &= C_{2^{k-1}} + 1 \\ &= C_{2^{k-2}} + 1 + 1 \\ &= C_{2^{k-3}} + 1 + 1 + 1 \\ &= \vdots \\ &= C_0 + 1 + \cdots + 1 \\ &= k + 1 \end{aligned}$$

και καθώς $k = \lg N$, τελικά το πλήθος συγκρίσεων που εκτελεί ο αλγόριθμος για μια ανεπιτυχή αναζήτηση είναι $W(N) = \lg N + 1$. □

Το αποτέλεσμα αυτό είναι εξαιρετικά σημαντικό. Ο χρόνος που απαιτείται για τον εντοπισμό μιας τιμής σε ένα ταξινομημένο πίνακα είναι, στη χειρότερη περίπτωση, ανάλογος του δυαδικού λογαρίθμου του μεγέθους του πίνακα. Απαιτούνται δηλαδή 32 συγκρίσεις το πολύ, για να εντοπίσουμε μια τιμή, μεταξύ δυόμισι δισεκατομμυρίων τιμών! Ο χρόνος αυτός είναι πρακτικά σταθερός, αν αναλογιστούμε πως, αυτή τη στιγμή, στις περισσότερες γλώσσες προγραμματισμού μια συστοιχία δεν μπορεί να έχει περισσότερο από 2^{31} θέσεις.



Σχήμα 1-8: Η λειτουργία του αλγορίθμου δυαδικής αναζήτησης.
Το σχήμα παρουσιάζει όλες τις πιθανές αποφάσεις του αλγορίθμου δυαδικής αναζήτησης κατά την εκτέλεσή του για μια συστοιχία 16 στοιχείων.
Τα οβάλ παριστάνουν αναδρομικές κλήσεις της μεθόδου `bsearch` του Κώδικα 1.8. Οι αντίστοιχοι αριθμοί είναι οι τιμές των τυπικών παραμέτρων `l` και `r`.
Οι ακμές παριστάνουν τις αποφάσεις που λαμβάνει ο αλγόριθμος δυαδικής αναζήτησης για το υποσύνολο της συστοιχίας στο οποίο θα περιοριστεί η αναζήτηση κατά την επόμενη αναδρομική κλήση.
Τα τετράγωνα παριστάνουν κλήσεις της μεθόδου που τερματίζουν την εκτέλεση της μεθόδου επιστρέφοντας την αντίστοιχη τιμή.



Σχήμα 1-10: Ιχνηλάτηση των αναδρομικών κλήσεων για τον υπολογισμό του F_6.
Οι κύκλοι παριστάνουν αναδρομικές κλήσεις για τον υπολογισμό του αντίστοιχου όρου της ακολουθίας Fibonacci Τα τετράγωνα παριστάνουν κλήσεις της μεθόδου που επιστρέφουν χωρίς να εκτελέσουν αναδρομικές κλήσεις. Το σχήμα επιδεικνύει την αναποτελεσματικότητα της αναδρομής όταν τα υποπροβλήματα επικαλύπτονται. Ο όρος F_2 για παράδειγμα, υπολογίζεται 5 φορές.

Δυναμικός προγραμματισμός. Το βασικό χαρακτηριστικό των αλγορίθμων που χρησιμοποιούν την τεχνική διαίρει και βασίλευε, είναι η διαμέριση του προβλήματος, ο χωρισμός δηλαδή του προβλήματος σε μικρότερα, ανεξάρτητα υποπροβλήματα όμοια με το αρχικό πρόβλημα. Αν κατά τη σχεδίαση ενός αναδρομικού αλγορίθμου δεν φροντίσουμε για την διαμέριση του προβλήματος, το αποτέλεσμα θα είναι να λύσουμε αρκετά υποπροβλήματα περισσότερες από μία φορές. Ένα κλασικό παράδειγμα είναι ο υπολογισμός του N-στού όρου της ακολουθίας Fibonacci.

Κώδικας 1.9: Αναδρομικός υπολογισμός του F_N

```
public static final int fibonacci(int n) {
   switch (n) {
      case 0: return 0;
      case 1: return 1;
   }
   return fibonacci(n-1) + fibonacci(n-2)
}
```

Για τον υπολογισμό του F_6 με την αναδρομική μέθοδο `fibonacci` του Κώδικα 1.9, οι όροι F_3 και F_2 θα υπολογιστούν τόσο κατά τον υπολογισμό του F_5 όσο και κατά τον υπολογισμό του F_4 όπως φαίνεται στο Σχήμα 1-10. Συνολικά απαιτούνται F_{N+1} κλήσεις της μεθόδου για τον υπολογισμό του F_N (βλέπε Άσκηση 1-22, σελίδα 13) και σχεδόν όλες υπολογίζουν αποτελέσματα τα οποία είναι ήδη γνωστά από προηγούμενους υπολογισμούς. Επειδή δε, $F_N = \Theta\left(\phi^N\right)$ —βλέπε Άσκηση 1-36, σελίδα 30— είναι σαφές πως

δεν θα ζήσουμε για να μάθουμε τον F_{100} αν βασιστούμε στον αλγόριθμο αυτό.

Μια πιο αποτελεσματική προσέγγιση παρουσιάζεται στον Κώδικα 1.10. Αυτή η εκδοχή της μεθόδου `fibonacci` χρησιμοποιεί ένα πίνακα προϋπολογισμένων όρων της ακολουθίας τον οποίο συμβουλεύεται πριν από την εκτέλεση κάθε αναδρομικής κλήσης. Αν η ζητούμενη τιμή έχει υπολογιστεί ήδη, η αναδρομική κλήση αποφεύγεται. Αν αντίθετα η ζητούμενη τιμή δεν είναι γνωστή, υπολογίζεται με μια αναδρομική κλήση και αποθηκεύεται στον πίνακα προϋπολογισμένων τιμών.

Κώδικας 1.10: Υπολογισμός του $N!$ με δυναμικό προγραμματισμό.

```
private static final int[] F = new int[46];
public static final int fibonacci(int n) {
   if (F[n] != 0) return F[n];
   if (n == 0) return 0;
   return F[n] = fibonacci(n-1) + fibonacci(n-2);
}
```

Βέβαια τα πράγματα μπορούν να γίνουν ακόμη απλούστερα από την παρατήρηση πως για τον υπολογισμό του F_N δεν χρειάζεται να γνωρίζουμε όλους τους όρους F_k, $k < N$ αλλά μόνο τους F_{N-1} και F_{N-2} και αντί να υπολογίζουμε "από πάνω προς τα κάτω", μπορούμε να υπολογίζουμε "από κάτω προς τα πάνω", καταλήγοντας στην παρακάτω επαναληπτική λύση η οποία θα ήταν και η ταχύτερη.

```
public static int fibonacci(int n) {
   int f0 = 0, f1 = 1, f = 0;
   for (int i = 2; i <= n; ++i) {
      f = f0 + f1;
      f0 = f1;
      f1 = f;
   }
   return f;
}
```

Ο Κώδικας 1.10 ωστόσο επιδεικνύει μια τεχνική επίλυσης σύνθετων προβλημάτων η οποία είναι εξαιρετικά σημαντική όταν το αρχικό πρόβλημα υποδιαιρείται σε υποπροβλήματα τα οποία δεν είναι ανεξάρτητα. Οι τεχνική αυτή ονομάζεται *δυναμικός προγραμματισμός*. Αν και την παρουσιάζουμε σε μια ενότητα που αφορά αναδρομικούς αλγορίθμους, η τεχνική έχει εφαρμογή τόσο σε αναδρομικούς όσο και σε επαναληπτικούς αλγορίθμους.

Η αποτελεσματικότητα της αναδρομής. Τα παραδείγματα αναδρομικών αλγορίθμων που εξετάσαμε στις προηγούμενες παραγράφους αυτής της ενότητας, καταδεικνύουν την ισχύ της αναδρομής τόσο στην σχεδίαση όσο και στην ανάλυση της αποτελεσματικότητας αλγορίθμων. Είναι ωστόσο εξαιρετικά δύσκολο να βρει κανείς άλλα πλεονεκτήματα στους αναδρομικούς αλγορίθμους πέρα από την απλότητα στην διατύπωση της λύσης. Μπορεί αντιθέτως να βρει αρκετά μειονεκτήματα.

i) Οι αναδρομικοί αλγόριθμοι χρησιμοποιούν κατά κανόνα κλήσεις μεθόδων ενώ οι επαναληπτικοί αλγόριθμοι απλούστερες λειτουργίες όπως σύγκριση, ανάθεση και αύξηση. Κατά συνέπεια οι επαναληπτικοί αλγόριθμοι τείνουν να εκτελούνται σαφώς πιο γρήγορα από τους αντίστοιχους αναδρομικούς.

ii) Οι αναδρομικοί αλγόριθμοι στην συντριπτική τους πλειοψηφία, τείνουν να χρησιμοποιούν περισσότερη μνήμη από ότι οι αντίστοιχοι επαναληπτικοί. Αυτό οφείλεται στο γεγονός ότι για την υλοποίηση κάθε κλήσης απαιτείται μνήμη για την αποθήκευση των ορισμάτων, των τοπικών μεταβλητών, της διεύθυνσης επιστροφής κλπ. Ακόμη και αν απαιτείται προσομοίωση της αναδρομής με στοίβα (βλέπε 5.3.2), οι επαναληπτικοί αλγόριθμοι εξακολουθούν να υπερτερούν.

iii) Σε μερικές περιπτώσεις, αναδρομικοί αλγόριθμοι όπως αυτός του Κώδικα 1.9, εκτελούν πολύ περισσότερες λειτουργίες από τους αντίστοιχους επαναληπτικούς.

Συμπερασματικά μπορούμε να πούμε ότι οι αναδρομικοί αλγόριθμοι είναι χρήσιμοι στα αρχικά στάδια επίλυσης ενός προβλήματος όταν μια επαναληπτική λύση δεν είναι προφανής. Επιπλέον η αναδρομή είναι χρήσιμη στην ανάλυση αλγορίθμων καθώς η συμπεριφορά του αλγορίθμου μπορεί στις περισσότερες περιπτώσεις να συμπυκνωθεί σε μια αναδρομική σχέση. Όταν όμως μπορούμε να σχεδιάσουμε ένα κομψό επαναληπτικό αλγόριθμο για τη λύση ενός προβλήματος, τότε πρέπει να προτιμούμε την επανάληψη από την αναδρομή.

Ασκήσεις

1-37 Υλοποιήστε ένα αναδρομικό και ένα επαναληπτικό αλγόριθμο για την αναστροφή των στοιχείων μιας συμβολοσειράς.

1-38 Σχολιάστε την παρακάτω αναδρομική μέθοδο.

```
public static final int weird(int n) {
   if (n == 1) return 1;
   if (n % 5 == 0) return weird(n / 5);
   return weird(3 * n / 2);
}
```

1-39 Υλοποιήστε μια αναδρομική μέθοδο για τον υπολογισμό του μεγίστου όρου μιας ακολουθίας $a_0, a_1, \ldots, a_{N-1}$, με βάση τον ορισμό

$$\max(a_0, \ldots, a_{N-1}) = \max(\max(L), \max(R)),$$

όπου $L = a_0, \ldots, a_{\lfloor N/2 \rfloor - 1}$, $R = a_{\lfloor N/2 \rfloor}, \ldots, a_{N-1}$.

1-40 Υλοποιήστε τον αλγόριθμο δυαδικής αναζήτησης, χωρίς αναδρομή.

1-41 Υπολογίστε το χρόνο εκτέλεσης του Κώδικα 1.10.

1-42 Γράψτε ένα αναδρομικό αλγόριθμο για τον υπολογισμό του $\log(N!)$.

1-43 Υλοποιήστε μια εφαρμογή σε Java, η οποία συγκρίνει πειραματικά το χρόνο εκτέλεσης των μεθόδων αναζήτησης που παρουσιάστηκαν στο κεφάλαιο αυτό.

1-44 Υπολογίστε την ποσότητα μνήμης που απαιτείται για τον υπολογισμό του όρου F_N της ακολουθίας Fibonacci με καθένα από τους αλγορίθμους που παρουσιάστηκαν σε αυτή την ενότητα.

ΚΕΦΑΛΑΙΟ 2

Αφηρημένοι τύποι δεδομένων

2.1 Εισαγωγή

Υπάρχουν δύο θεμελιωδώς διαφορετικές θεωρήσεις της έννοιας του *τύπου δεδομένων*. Με βάση την πρώτη, ένας τύπος δεδομένων είναι ένα σύνολο τιμών που καθορίζεται από το *μορφότυπο* και το *μήκος* του. Το μορφότυπο ενός τύπου καθορίζει τη σύμβαση που χρησιμοποιείται για την αναπαράσταση των τιμών του στη μνήμη του υπολογιστή. Γνωρίζοντας τη σύμβαση αυτή, καθώς και το πλήθος των ψηφιοσυλλαβών που απαιτούνται για την αποθήκευση μιας τιμής, μπορούμε να ορίσουμε το σύνολο των τιμών του αντίστοιχου τύπου δεδομένων. Στη γλώσσα Java, ο τύπος `int`, χρησιμοποιεί παράσταση συμπληρώματος ως προς δύο και έχει μήκος τεσσάρων ψηφιοσυλλαβών· είναι δηλαδή το σύνολο των τιμών $\left[-2^{31}, 2^{31}-1\right]$. Αυτός είναι ο κλασικός τρόπος θεώρησης ενός τύπου δεδομένων ως συνόλου τιμών. Οι πρωτογενείς τύποι των περισσότερων γλωσσών προγραμματισμού προσεγγίζονταιμε βάση αυτή τη θεώρηση.

Εκτός όμως από τους πρωτογενείς τύπους δεδομένων, η Java, όπως οι περισσότερες γλώσσες προγραμματισμού άλλωστε, παρέχει στους προγραμματιστές τη δυνατότητα ορισμού τύπων μέσω του μηχανισμού κλάσεων και διεπαφών. Οι τύποι αυτοί ονομάζονται *αφηρημένοι τύποι δεδομένων*.

Η κλασική θεώρηση της έννοιας του τύπου δεδομένων αποτυγχάνει στην περίπτωση των αφηρημένων τύπων δεδομένων. Αν και είναι λογικό να ορίσουμε ως τιμές ενός αφηρημένου τύπου δεδο-

boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double

Πίνακας 2-1: Περιβλήματα πρωτογενών τύπων.

μένων τα αντικείμενά του, δεν έχουμε κανένα τρόπο να γνωρίζουμε τα δομικά χαρακτηριστικά των αντικειμένων αυτών. Κατά συνέπεια χρειαζόμαστε μια νέα θεώρηση για την έννοια του τύπου δεδομένων η οποία δεν θα βασίζεται σε λεπτομέρειες υλοποίησης.

Ορισμός 2-1. Ένας αφηρημένος τύπους δεδομένων είναι το σύνολο των αντικειμένων που υποστηρίζουν ένα αυστηρά καθορισμένο, κατά κανόνα μη-κενό, σύνολο λειτουργιών.

Η θεώρηση της έννοιας του τύπου δεδομένων με βάση τον Ορισμό 2-1 δεν βασίζεται στην απαρίθμηση των τιμών ενός τύπου δεδομένων, αλλά στον αυστηρό καθορισμό των λειτουργιών που μπορούν να εκτελέσουν τα αντικείμενά του. Οι λειτουργίες ενός αφηρημένου τύπου δεδομένων, καθορίζουν τη συμπεριφορά των αντικειμένων του. Αξίζει να τονίσουμε εδώ, πως η θεώρηση αυτή μπορεί να χρησιμοποιηθεί και για τους πρωτογενείς τύπους δεδομένων. Για παράδειγμα, ο πρωτογενής τύπος `int` στη Java είναι το σύνολο των αντικειμένων (μεταβλητών) για τα οποία ορίζονται οι τελεστές +, −, κτλ.

Από εδώ και στο εξής θα χρησιμοποιούμε τον όρο *αφηρημένος τύπος δεδομένων*, ή απλά *τύπος*, για να αναφερθούμε στο σύνολο των αντικειμένων που επιδεικνύουν μια συγκεκριμένη συμπεριφορά, ενώ όταν χρησιμοποιούμε τον όρο *τύπος δεδομένων* θα αναφερόμαστε σε ένα σύνολο τιμών, και κατά βάση σε πρωτογενείς τύπους δεδομένων.

Η γλώσσα Java, παρέχει οκτώ πρωτογενείς τύπους δεδομένων. Για κάθε ένα από αυτούς υπάρχει ένας αντίστοιχος αφηρημένος τύπος με ανάλογη λειτουργικότητα. Οι τύποι αυτοί (Πίνακας 2-1) έχει καθιερωθεί να ονομάζονται *περιβλήματα* (wrappers) καθώς, από δομική άποψη τουλάχιστον, απλώς "περιβάλλουν" μια μεταβλητή ενός πρωτογενούς τύπου δεδομένων.

Μια συζήτηση για αφηρημένους τύπους δεδομένων είναι αναπόφευκτα αφηρημένη. Ένα παράδειγμα αφηρημένου τύπου δεδομένων θα διευκολύνει την περιγραφή. Ας υποθέσουμε πως ξεκινάμε την ανάπτυξη ενός μικρού πακέτου γραφικών. Ένα από τα πρώτα πράγματα που χρειαζόμαστε είναι ο ορισμός αντικειμένων που παριστάνουν σημεία του καμβά στον οποίο σχεδιάζονται γραφικά αντικείμενα όπως ευθύγραμμα τμήματα, κύκλοι κτλ. Είναι λογικό

να θεωρήσουμε πως το σύστημα συντεταγμένων χρησιμοποιεί μη-αρνητικούς ακεραίους. Ένα σημείο είναι ένα ζεύγος συντεταγμένων, οπότε είναι λογικό να ξεκινήσουμε κάπως έτσι:

```
public class Point {
   public int x, y;
}
```

Τα πεδία x και y είναι οι συντεταγμένες στον οριζόντιο και κάθετο άξονα αντίστοιχα.

Η κλάση `Point` είναι μεν αφηρημένος τύπος δεδομένων με βάση τον Ορισμό 2-1, αλλά δεν είναι ούτε ιδιαίτερα χρήσιμος, ούτε ορθά σχεδιασμένος. Η δομή του εκτίθεται δημόσια, και δεν καθορίζει δημόσιες μεθόδους. Το βασικό χαρακτηριστικό ενός αφηρημένου τύπου δεδομένων είναι η απόκρυψη της δομής των αντικειμένων του. Ένας τέτοιος σχεδιασμός αφήνει μία και μόνη επιλογή για την επικοινωνία με αντικείμενα αυτού του τύπου: τη χρήση των λειτουργιών που ο τύπος παρέχει. Με τον τρόπο αυτό είναι δυνατή η διαφοροποίηση της υλοποίησης των λειτουργιών, χωρίς να απαιτείται καμία τροποποίηση στις μεθόδους άλλων αφηρημένων τύπων δεδομένων που λειτουργούν ως χρήστες των παρεχομένων υπηρεσιών. Η απόκρυψη των λεπτομερειών υλοποίησης, είναι το αποφασιστικό βήμα προς τη σχεδίαση αυτόνομων, επαναχρησιμοποιήσιμων και ευέλικτων *εξαρτημάτων λογισμικού*.

Μέθοδοι πρόσβασης. Το πρώτο βήμα για τη βελτίωση της σχεδίασης του τύπου `Point` είναι να αποκρύψουμε όσο περισσότερο μπορούμε τα δομικά του χαρακτηριστικά και να καθορίσουμε *μεθόδους πρόσβασης* σε αυτά. Οι μέθοδοι πρόσβασης είναι ένας απλός μηχανισμός μέσω του οποίου ένας αφηρημένος τύπος δεδομένων επιτρέπει την ανάγνωση ή και την τροποποίηση των πεδίων των αντικειμένων του με τρόπο ελεγχόμενο. Διακρίνουμε τις μεθόδους πρόσβασης σε μεθόδους ανάγνωσης και μεθόδους εγγραφής.

Ορισμός 2-2. Αν `field` είναι το όνομα ενός πεδίου τύπου `T` μιας κλάσης `C`, η μέθοδος της `C` με επικεφαλίδα `T getField()` ονομάζεται *μέθοδος ανάγνωσης* του πεδίου `field`, ενώ η μέθοδος με επικεφαλίδα `void setField(T)` ονομάζεται *μέθοδος εγγραφής* του πεδίου `field`.

Θα εξελίξουμε τον τύπο `Point` έτσι ώστε να υποστηρίζει μεθόδους πρόσβασης για τα πεδία x και y τα οποία θα μετατρέψουμε σε ι-

διωτικά. Έτσι η πρόσβαση στα πεδία των αντικειμένων τύπου `Point` θα γίνεται μέσω των μεθόδων ανάγνωσης `getX()` και `getY()` και των μεθόδων εγγραφής `setX(int)` και `setY(int)`.

```
public class Point {
   private int x, y;
   public int getX() {
      return x;
   }
   public void setX(int x) {
      this.x = x < 0 ? 0 : x;
   }
   public int getY() {
      return y;
   }
   public void setY(int y) {
      this.y = y < 0 ? 0 : y;
   }
}
```

Πρέπει να ομολογήσουμε πως στην συγκεκριμένη περίπτωση δεν είναι δυνατή η πλήρης απόκρυψη των δομικών χαρακτηριστικών του τύπου `Point` καθώς είναι προφανές πως ένα αντικείμενο αυτού του τύπου αποτελείται από δύο ακέραιες συντεταγμένες. Το γεγονός όμως ότι ο χειρισμός συντεταγμένων των αντικειμένων τύπου `Point` γίνεται πλέον μόνο μέσω των μεθόδων πρόσβασης που παρέχονται, είναι ήδη ένα σημαντικό κέρδος. Οι μέθοδοι εγγραφής για παράδειγμα, εξασφαλίζουν ότι οι συντεταγμένες κάθε αντικειμένου τύπου `Point` είναι πάντα μη-αρνητικές. Αυτός ο έλεγχος, αν και φαντάζει απλοϊκός, μπορεί να αποτρέψει την εμφάνιση μεγάλου αριθμού λαθών.

Κατασκευαστές και μέθοδοι κατασκευής. Για να διευκολύνουμε τους χρήστες του τύπου `Point` στην κατασκευή αντικειμένων, θα ορίσουμε τρεις κατασκευαστές. Ο πρώτος είναι ο εξ' ορισμού κατασκευαστής (default constructor) ο οποίος κατασκευάζει το σημείο $(0, 0)$.

```
public Point() {
   super();
   x = y = 0;
}
```

Ορισμός 2-3. Ο χωρίς παραμέτρους κατασκευαστής του αφηρημένου τύπου T, ονομάζεται *εξ΄ ορισμού κατασκευαστής* και έχει την επικεφαλίδα `T()`.

Ο δεύτερος κατασκευαστής του τύπου `Point` έχει δύο τυπικές παραμέτρους, που καθορίζουν τις συντεταγμένες του αντικειμένου που θα κατασκευαστεί. Πολλές φορές τέτοιου είδους κατασκευαστές ονομαζονται και παραμετρικοί.

```
public Point(int x, int y) {
   super();
   this.x = x;
   this.y = y;
}
```

Ο τελευταίος κατασκευαστής είναι ένας κατασκευαστής αντιγραφής (copy constructor) ο οποίος κατασκευάζει ένα αντικείμενο τύπου `Point`, αντιγράφοντας τις συντεταγμένες ενός άλλου αντικειμένου τύπου `Point`.

Ορισμός 2-4. Ο *κατασκευαστής αντιγραφής* του αφηρημένου τύπου T έχει την επικεφαλίδα `T(T)`.

```
public Point(Point p) {
   this(p.x, p.y);
}
```

Σε πολλές εφαρμογές γραφικών, είναι χρήσιμη η κατασκευή τυχαίων σημείων. Θα μπορούσαμε να τροποποιήσουμε τον εξ' ορισμού κατασκευαστή ώστε να θέτει τυχαίες μη-αρνητικές τιμές στα πεδία του αντικειμένου. Κάτι τέτοιο ωστόσο, αν και δεν είναι κατ' ανάγκη κακό, δεν θα ήταν μια καλή επιλογή. Ας φανταστούμε για λίγο τον τύπο `int` ως αφηρημένο τύπο δεδομένων. Είναι γνωστό πως η τιμή ενός τέτοιου αντικειμένου αμέσως μετά την κατασκευή του είναι 0. Αν ο τύπος αυτός κατασκευαζόταν ώστε να έχει τυχαία ή ακαθόριστη τιμή, η ζωή των προγραμματιστών που τον χρησιμοποιούν θα ήταν μάλλον δυσκολότερη. Για τον ίδιους λόγους, θα κρατήσουμε τον εξ' ορισμού κατασκευαστή του τύπου `Point` ως έχει και θα ορίσουμε μια *μέθοδο κατασκευής* (factory method) η οποία δημιουργεί τυχαία σημεία. Οι μέθοδοι κατασκευής είναι γνωστές και ως *εικονικοί κατασκευαστές* (virtual constructors) και στη Java, έχουν κατά σύμβαση ονόματα όπως `newInstance` ή `getInstance`, χωρίς φυσικά αυτό να είναι υποχρεωτικό.

```
public static Point newRandomInstance() {
   int x = (int) (Math.random() * Short.MAX_VALUE);
   int y = (int) (Math.random() * Short.MAX_VALUE);
   return new Point(x,y);
}
```

Οι μέθοδοι κατασκευής είναι ένας εναλλακτικό μηχανισμός κατασκευής αντικειμένων που παρέχει μεγαλύτερη ευελιξία από τους κατασκευαστές.

- Ένας κατασκευαστής του τύπου T επιστρέφει υποχρεωτικά ένα αντικείμενο τύπου T. Αντίθετα, μια μέθοδος κατασκευής του τύπου T μπορεί να επιστρέψει ένα αντικείμενο οποιουδήποτε τύπου S, αρκεί αυτός να είναι απόγονος του T.
- Μια μέθοδος κατασκευής μπορεί να ακυρωθεί ή να δηλωθεί ως τελική, κάτι που δεν μπορεί να γίνει για κανένα κατασκευαστή.
- Επιπλέον, μια μέθοδος κατασκευής του τύπου T δεν είναι ανάγκη να οριστεί από τον τύπο T, αλλά από οποιονδήποτε τύπο.
- Τέλος, μια μέθοδος κατασκευής μπορεί να είναι στατική ή όχι. Στην περίπτωση βέβαια που μια μέθοδος κατασκευής του τύπου T ορίζεται στον ίδιο τον τύπο T, όπως η μέθοδος newRandomInstance, πρέπει να είναι στατική για προφανείς λόγους.

Κάθε φορά που σχεδιάζουμε ένα αφηρημένο τύπο δεδομένων πρέπει να αφιερώσουμε λίγο χρόνο προκειμένου να εκτιμήσουμε την αναγκαιότητα υλοποίησης μεθόδων κατασκευής.

΄Αλλες μέθοδοι. Θα υλοποιήσουμε δύο ακόμη μεθόδους για τον τύπο Point. Η μέθοδος move(int,int), μετακινεί ένα αντικείμενο τύπου Point μετατοπίζοντας τις τρέχουσες συντεταγμένες του κατά dx και dy αντίστοιχα, φροντίζοντας πάντα οι νέες συντεταγμένες να μην έχουν αρνητικές τιμές, αν και οι τιμές των παραμέτρων dx και dy μπορεί να είναι αρνητικές. Αξίζει να παρατηρήσουμε πως η μέθοδος αν και έχει πρόσβαση στις ιδιωτικές μεταβλητές του αντικειμένου, χρησιμοποιεί τις μεθόδους εγγραφής. Επιπλέον επιστρέφει το αντικείμενο έτσι ώστε να έχουμε τη δυνατότητα αλυσίδωσης κλήσεων όπως για παράδειγμα p.move(3,-2).distance(q). Η μέθοδος distance(Point) υπολογίζει την απόσταση μεταξύ δύο σημείων του

επιπέδου.

```
public Point move(int dx, int dy) {
   setX(x + dx);
   setY(y + dy);
   return this;
}
public int distance(Point p) {
   int dx = x - p.x;
   int dy = y - p.y;
   return (int) (Math.sqrt(dx*dx + dy*dy) + 0.5);
}
```

Σταθερά και μεταβλητά αντικείμενα. Η κατάσταση των αντικειμένων ενός αφηρημένου τύπου, όπως αντικατοπτρίζεται από τις τιμές των πεδίων του, μπορεί να είναι *σταθερή* (immutable) ή μεταβλητή (mutable). Η δυνατότητα μεταβολής της κατάστασης των αντικειμένων ενός τύπου καθορίζεται από τη διεπαφή του, και ισχύει για όλα τα αντικείμενά του. Δεν μπορεί δηλαδή κάποια από τα αντικείμενα ενός τύπου να είναι μεταβλητά και κάποια άλλα όχι.

Ορισμός 2-5. Τα αντικείμενα ενός αφηρημένου τύπου ονομάζονται *σταθερά αντικείμενα* όταν η κατάστασή τους δεν μπορεί να τροποποιηθεί μετά την κατασκευή τους. Αν αυτό δεν ισχύει, τα αντικείμενα ονομάζονται *μεταβλητά αντικείμενα*.

Τα αντικείμενα του τύπου `Point` είναι μεταβλητά καθώς η κατάστασή τους μπορεί να τροποποιηθεί μετά την κατασκευή τόσο από τις μεθόδους εγγραφής, όσο και από τη μέθοδο `move`. Χαρακτηριστικά παραδείγματα τύπων σταθερών αντικειμένων είναι τα περιβλήματα πρωτογενών τύπων (βλέπε Πίνακα 2-1 στη σελίδα 42) καθώς επίσης και ο τύπος `String`. Η μεταβλητότητα των αντικειμένων ενός τύπου έχει ιδιαίτερη σημασία τόσο στη γλώσσα Java, όσο και στην υλοποίηση αφηρημένων δομών δεδομένων. Οι επιπλοκές που μπορεί να προκληθούν θα συζητηθούν παρακάτω.

Θεμελιώδεις μέθοδοι. Εκτός από τις μεθόδους του τύπου `Point` που έχουμε ορίσει, υπάρχουν μερικές λειτουργίες οι οποίες είναι τόσο θεμελιώδεις ώστε απαιτούνται σχεδόν για κάθε αφηρημένο τύπο δεδομένων. Πρόκειται για τις λειτουργίες εκείνες που είναι διαθέσιμες, υπό μορφή τελεστών, για τους πρωτογενείς τύπους δεδομένων.

Η γλώσσα προγραμματισμού C++ για παράδειγμα, παρέχει σε κάθε αφηρημένο τύπο δεδομένων τη δυνατότητα να υπερφορτώσει τη λειτουργία τελεστών όπως =, ==, !=, κτλ. ορίζοντας έτσι την σημασιολογία τους όταν τα ορίσματά τους είναι αντικείμενα του τύπου αυτού. Αν και η Java δεν προσφέρει αυτή την δυνατότητα στους προγραμματιστές, οι σχεδιαστές της έχουν προβλέψει τη συγκεκριμένη ανάγκη και έχουν καθορίσει εναλλακτικούς τρόπους για την επίτευξη ανάλογης λειτουργικότητας. Στις επόμενες ενότητες θα παρουσιάσουμε τις σχεδιαστικές αυτές επιλογές, και θα περιγράψουμε τις τεχνικές που μπορούν να εφαρμοστούν όταν ένας αφηρημένος τύπος δεδομένων πρέπει να παρέχει λειτουργίες ανάλογες με αυτές των τελεστών =, ==, !=, <, <=, >, >=.

Ασκήσεις

2-1 Δώστε τις επικεφαλίδες του εξ' ορισμού κατασκευαστή, του κατασκευαστή αντιγραφής, καθώς και των μεθόδων πρόσβασης για τα πεδία `name` και `serialNumber` της κλάσης `SparePart`.

2-2 Πόσες μεθόδους κατασκευής μπορείτε να εντοπίσετε στον τύπο `Integer`;

2-3 Υλοποιήστε τον αφηρημένο τύπο `ImmutablePoint` του οποίου τα αντικείμενα παριστάνουν σημεία των οποίων οι συντεταγμένες δεν μπορούν να τροποποιηθούν.

2-4 Υλοποιήστε τον αφηρημένο τύπο `LineSegment` που παριστάνει ένα ευθύγραμμο τμήμα.

2-5 Υλοποιήστε τον αφηρημένο τύπο δεδομένων `Complex` που παριστάνει μιγαδικούς αριθμούς. Προσπαθήστε να καθορίσετε μεθόδους που υλοποιούν τη σημασιολογία των τελεστών == και != για αντικείμενα τύπου `Complex`.

2.2 Σχέσεις ισοδυναμίας

Η ισότητα μεταξύ των στοιχείων ενός συνόλου S, όπως τη γνωρίζουμε από τα μαθηματικά, είναι μια σχέση ισοδυναμίας· ένα σύνολο $R \subseteq S \times S$ που κατέχει τις εξής ιδιότητες.

i). Ανακλαστικότητα: $(a, a) \in R$ για κάθε $a \in S$.

ii). Συμμετρία: Αν $(a, b) \in R$ τότε $(b, a) \in R$.

iii). Μεταβατικότητα: Αν $(a, b), (b, c) \in R$ τότε $(a, c) \in R$.

Στη γλώσσα προγραμματισμού Java, μια σχέση ισοδυναμίας μεταξύ αντικειμένων ορίζεται μέσω της μεθόδου `equals` της κλάσης `Object`, η οποία θα λέγαμε πως αποτελεί τον τελεστή ισότητας αντικειμένων.

```
public boolean equals(Object)
```

Ορισμός 2-6. Έστω `x` και `y`, μεταβλητές αναφοράς τύπου `Object`. Θα λέμε πως τα αντικείμενα στα οποία αναφέρονται οι μεταβλητές αυτές είναι *ίσα*, τότε και μόνο τότε αν `x.equals(y)`.

Η υλοποίησή της μεθόδου `equals` στην κλάση `Object` πληροί τις ιδιότητες μιας σχέσης ισοδυναμίας.

i). Ανακλαστικότητα: Για κάθε αντικείμενο `x` τύπου `Object`, η κλήση `x.equals(x)` επιστρέφει την τιμή `true`.

ii). Συμμετρία: Για κάθε ζεύγος αντικειμένων `x`, `y`, η κλήση `x.equals(y)` επιστρέφει την τιμή `true` τότε και μόνο τότε αν η κλήση `y.equals(x)` επιστρέφει την τιμή `true`.

iii). Μεταβατικότητα: Αν `x`, `y` και `z` είναι αντικείμενα τύπου `Object`, και οι κλήσεις `x.equals(y)` και `y.equals(z)` επιστρέφουν την τιμή `true`, το ίδιο θα κάνει και η κλήση `x.equals(z)`.

iv). Συνέπεια: Για κάθε ζεύγος αντικειμένων `x`, `y`, διαδοχικές κλήσεις της μεθόδου `x.equals(y)` επιστρέφουν την ίδια τιμή.

Η υλοποίηση της μεθόδου από κάθε αφηρημένο τύπο πρέπει να ικανοποιεί τις ιδιότητες αυτές, όπως και αυτή της `Object`:

```
public boolean equals(Object object) {
   return this == object;
}
```

Για την κλάση `Object`, δύο αντικείμενα είναι ίσα τότε και μόνο τότε αν συμπίπτουν. Για μια απόγονο `T` της `Object` ωστόσο, αυτός ο ορισμός της ισότητας μπορεί να μην είναι ορθός. Τότε η `T` έχει την ευθύνη να ακυρώσει τη μέθοδο ώστε να ορίσει ορθά τη σημασιολογία της ισότητας αντικειμένων τύπου `T`. Στον αφηρημένο τύπο δεδομένων `Point` για παράδειγμα, η ισότητα δύο αντικειμένων πρέπει να βασίζεται στην ισότητα των συντεταγμένων τους. Δύο διαφορετικά σημεία που έχουν τις ίδιες συντεταγμένες, είναι ίσα. Η υλοποίηση της μεθόδου `equals` για την κλάση `Point`, δίνεται στο παρακάτω απόσπασμα κώδικα.

```
1 public boolean equals(Object o) {
2    if (this == o) return true;
3    if (o == null) return false;
4    if (o.getClass() != Point.class) return false;
5    Point p = (Point) o;
6    return this.x == p.x && this.y == p.y;
7 }
```

Είναι εύκολο να αποδείξει κανείς πως η παραπάνω υλοποίηση έχει τις επιθυμητές ιδιότητες, τηρεί δηλαδή το πρωτόκολλο που θέτει η κλάση `Object`. Αξίζει να σημειώσουμε ωστόσο, πως με βάση το πρωτόκολλο της `Object`, είναι απολύτως νόμιμο να συγκρίνουμε αντικείμενα διαφορετικού τύπου, αν και κάτι τέτοιο δεν είναι ιδιαίτερα συνηθισμένο. Ορισμένες φορές ωστόσο μπορεί να χρειαστεί να συγκρίνουμε αντικείμενα ενός τύπου T, με αντικείμενα κάποιου απογόνου του, έστω S. Σε τέτοιες περιπτώσεις πρέπει να φροντίσουμε ώστε να εξασφαλίζεται η ανακλαστικότητα της υλοποίησης (όπως έχουμε κάνει για τον τύπο `Point`). Σε πολλές περιπτώσεις, υλοποιήσεις της `equals` κάνουν χρήση του τελεστή `instanceof`. Για παράδειγμα, θα μπορούσαμε να είχαμε υλοποιήσει τη μέθοδο `equals` όπως στο παρακάτω απόσπασμα.

```
1 public boolean equals(Object o) {
2    if (this == o) return true;
3    if (!(o instanceof Point) return false;
4    Point p = (Point) o;
5    return this.x == p.x && this.y == p.y;
6 }
```

Πρόκειται για μια προσπάθεια συντόμευσης του κώδικα· αν η παράμετρος `o` είναι `null` τότε η τιμή της παράστασης `o instanceof T` είναι πάντα `false`, και κατά συνέπεια ο έλεγχος `if (o == null)` μπορεί να παραλειφθεί. Η υλοποίηση αυτή λειτουργεί ορθά στις περισσότερες, αλλά όχι σε όλες τις περιπτώσεις καθώς ο τελεστής `instanceof`, σε αντίθεση με τον τελεστή ισότητας, δεν είναι συμμετρικός. Αυτό υπό την παρουσία ιεραρχιών κληρονομικότητας μπορεί να δημιουργήσει προβλήματα στη συμμετρικότητα της υλοποίησης.

Ασκήσεις

●**2-6** Αποδείξτε πως η μέθοδος `equals` του αφηρημένου τύπου `Point`, είναι μια σχέση ισοδυναμίας.

●**2-7** Υλοποιήστε τη μέθοδο `static boolean equals(int[] a, int[] b)` για τη σύγκριση δύο συστοιχιών.

▷**2-8** Επαναλάβετε την προηγούμενη άσκηση, αυτή τη φορά για συστοιχίες αντικειμένων.

●**2-9** Δώστε ένα παράδειγμα όπου η υλοποίηση της μεθόδου `equals` με τη χρήση του τελεστή `instanceof` μπορεί να παραβιάσει την ιδιότητα της συμμετρίας.

2-10 Υλοποιήστε τη μέθοδο `equals` για την κλάση `LineSegment` της Άσκησης 2-4 στη σελίδα 48.

2.3 Αντιγραφή αντικειμένων

Ο τελεστής ανάθεσης αντιγράφει την τιμή μιας μεταβλητής ενός πρωτογενή τύπου δεδομένων σε μια άλλη μεταβλητή. Αν `v` και `w` είναι μεταβλητές του πρωτογενή τύπου δεδομένων `P`, τότε αμέσως μετά την εκτέλεση της εντολής `v = w` θα ισχύει `v == w`. Είναι πολύ χρήσιμο να έχουμε ανάλογους μηχανισμούς και για αφηρημένους τύπους δεδομένων.

Ορισμός 2-7. Έστω `p` και `q`, μεταβλητές αναφοράς τύπου `Object`. Θα λέμε πως το αντικείμενο στο οποίο αναφέρεται η `p` είναι *αντίγραφο* ή *κλώνος* του αντικειμένου στο οποίο αναφέρεται η `q`, τότε και μόνο τότε, αν `p != q` και `p.equals(q)`.

Καθώς η αντιγραφή, ή αλλιώς κλωνοποίηση, των αντικειμένων ενός τύπου απαιτεί γνώση της δομικής υπόστασης του τύπου αυτού, η ευθύνη για την περιγραφή της διαδικασίας κλωνοποίησης ενός αντικειμένου πρέπει να καθορίζεται από τον ίδιο τον τύπο. Στη Java, η μέθοδος κλωνοποίησης ορίζεται στην κλάση `Object` και έχει την εξής επικεφαλίδα.

```
protected Object clone() throws CloneNotSupportedException
```

Κανένας αφηρημένος τύπος δεδομένων δεν είναι κατ' αρχήν κλωνοποιήσιμος καθώς η μέθοδος ορίζεται από την κλάση `Object` με

προστατευμένη πρόσβαση. Έτσι ένα αντικείμενο δεν μπορεί να καλέσει τη μέθοδο αυτή σε αντικείμενο άλλου τύπου αν ο τελευταίος δεν έχει ακυρώσει τη μέθοδο ώστε να αυξήσει την ορατότητά της. Πριν περιγράψουμε τις υποχρεώσεις που αναλαμβάνει ένας προγραμματιστής ακυρώνοντας τη μέθοδο clone της κλάσης Object, ας εξετάσουμε τον τρόπο με τον οποίο αυτή λειτουργεί, υποθέτοντας πως καλείται σε ένα αντικείμενο ο τύπου T και πως κανένας πρόγονος του T δεν έχει ακυρώσει τη μέθοδο clone() του τύπου Object. Αν η κλάση T δεν υλοποιεί τη διεπαφή Cloneable, η μέθοδος θα σημαίνει μια εξαίρεση τύπου CloneNotSupportedException. Διαφορετικά, η μέθοδος θα κατασκευάσει ένα αντικείμενο c τύπου T και θα αντιγράψει τις τιμές των πεδίων του αντικειμένου o στα αντίστοιχα πεδία του αντικειμένου c. Τέλος, η μέθοδος θα επιστρέψει το αντικείμενο c, που είναι κλώνος του o.

Θα πρέπει να τονίσουμε πως η κλωνοποίηση έχει άμεση σχέση με την κληρονομικότητα. Για να μπορεί ένας τύπος T να κλωνοποιεί αντικείμενα, πρέπει όλοι οι προγονοί του να υποστηρίζουν κλωνοποίηση, καθώς κάποια από τα πεδία του τύπου αυτού μπορεί να έχουν κληρονομηθεί και κατά συνέπεια η κλωνοποίησή τους είναι ευθύνη του αντίστοιχου προγόνου. Θα πρέπει να αναφέρουμε εδώ πως οι συστοιχίες οποιουδήποτε τύπου είναι κλωνοποιήσιμες.

Η διαδικασία της κλωνοποίησης πρέπει πάντα να ξεκινάει από τη μέθοδο Object.clone(). Έτσι, κάθε τύπος που υποστηρίζει την κλωνοποίηση των αντικειμένων του, πρέπει να υλοποιεί τη διεπαφή Cloneable και να ακυρώνει τη μέθοδο clone() που έχει κληρονομήσει. Η νέα υλοποίηση πρέπει αναγκαστικά να καλέσει την κληρονομηθείσα υλοποίηση super.clone(). Τηρώντας αυτό το πρωτόκολλο αυστηρά, κάποτε θα κληθεί η μέθοδος Object.clone() από την οποία θα ξεκινήσει η διαδικασία της κλωνοποίησης.

Η διεπαφή Cloneable δεν ορίζει μεθόδους. Ο λόγος ύπαρξής της είναι για να υποδείξει ο τύπος που την υλοποιεί, στην μέθοδο Object.clone() ότι επιτρέπεται η κλωνοποίηση των αντικειμένων του. Όταν ένας τύπος πρέπει να απαγορεύσει την κλωνοποίηση της δομής των αντικειμένων του, τότε πρέπει να υλοποιήσει τη διεπαφή Cloneable και να ακυρώσει τη μέθοδο clone() ως εξής:

```
public Object clone() throws CloneNotSupportedException {
   throw new CloneNotSupportedException ();
}
```

Αντίθετα όταν ένας τύπος υποστηρίζει κλωνοποίηση, πρέπει να υλοποιήσει τη διεπαφή `Cloneable` και να ακυρώσει τη μέθοδο `clone()` ως εξής:

```
public Object clone() throws CloneNotSupportedException {
   return super.clone();
}
```

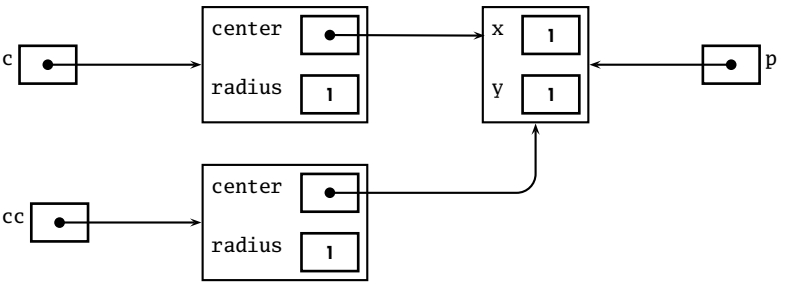
Το αποτέλεσμα ονομάζεται *ρηχή αντιγραφή* (shallow copy). Για να κατανοήσουμε καλύτερα τι αυτό σημαίνει, θα μελετήσουμε τον τύπο `Circle`.

```
public class Circle implements Cloneable {
   private Point center;
   private int radius;
   public Circle() {
      center = new Point();
      radius = 1;
   }
   public Circle(Point p, int r) {
      center = p;
      radius = r;
   }
   public Point getCenter() {
      return center;
   }
   public int getRadius() {
      return radius;
   }
   protected Object clone() throws CloneNotSupportedException {
      return super.clone();
   }
}
```

Ο τύπος `Circle` υποστηρίζει κλωνοποίηση με βάση το πρωτόκολλο που περιγράψαμε παραπάνω. Είναι ενδιαφέρον να δούμε τι θα συμβεί αν εκτελέσουμε το παρακάτω πρόγραμμα.

```
   public static void main(String[] args) throws Exception {
      Point p = new Point(1,1);
      Circle c = new Circle(p, 1);
      Circle cc = (Circle) c.clone();
      p.setX(5);
      System.out.println(cc.getCenter().getX());
   }
```

Το πρόγραμμα αυτό θα τυπώσει 5 αντί για 1 όπως ενδεχομένως θα περιμέναμε. Και αυτό οφείλεται στο γεγονός πως η μέθοδος



Σχήμα 2-1: Ρηχή κλωνοποίηση.
Η μέθοδος clone() της κλάσης Object κατασκευάζει ένα αντικείμενο τύπου Circle και αντιγράφει τις τιμές των πεδίων του αντικειμένου c στα αντίστοιχα πεδία του κλώνου.

Object.clone() εκτελεί ρηχή και όχι *βαθιά αντιγραφή*. Το Σχήμα 2-1 δίνει την εξήγηση για το τι έχει συμβεί.

Η βαθιά αντιγραφή σε αντίθεση με την ρηχή, όχι μόνο αντιγράφει τις τιμές των πεδίων πρωτογενών τύπων, αλλά επιπλέον κλωνοποιεί τα αντικείμενα στα οποία αναφέρονται τα πεδία τύπου αναφοράς. Έτσι αν θέλαμε η μέθοδος clone() της κλάσης Circle να εκτελεί βαθιά αντιγραφή, θα έπρεπε να την είχαμε υλοποιήσει ως εξής.

```
protected Object clone() throws CloneNotSupportedException {
   Circle clone = (Circle) super.clone();
   clone.center = (Point) center.clone();
   return clone;
}
```

Φυσικά, στην περίπτωση αυτή, και η κλάση Point θα πρέπει να τροποποιηθεί ώστε να υποστηρίζει κλωνοποίηση (βλέπε Άσκηση 2-17).

Το ερώτημα που τίθεται, είναι πότε ένας τύπος πρέπει να υλοποιεί ρηχή και πότε βαθιά αντιγραφή. Η τελική απόφαση ανήκει στο σχεδιαστή του τύπου, αν και μπορεί να διατυπωθεί ένας κανόνας που καλύπτει την πλειοψηφία των περιπτώσεων.

Τα πεδία ενός τύπου μπορεί να είναι είτε πρωτογενούς τύπου, είτε αναφορές σε σταθερά αντικείμενα, είτε τέλος, αναφορές σε μεταβλητά αντικείμενα.

i). Για τα πεδία πρωτογενών τύπων η αντιγραφή είναι πάντοτε βαθιά. Τόσο το πρωτότυπο όσο και οι κλώνοι του, έχουν διαφορετικά, και κατά συνέπεια ανεξάρτητα μεταξύ τους, αντίγραφα των πεδίων αυτών.

ii). Ένα πεδίο που αναφέρεται σε πρέπει κατά κανόνα να αντιγράφεται ρηχά. Από τη στιγμή που η κατάσταση του αντικειμένου

στο οποίο αναφέρεται το πεδίο, δεν μπορεί να μεταβληθεί, είναι πάντα ασφαλές ένας ή περισσότεροι κλώνοι να διατηρούν αναφορές στο αντικείμενο αυτό. Εκτός από ασφαλής, η ρηχή αντιγραφή σταθερών αντικειμένων επιταχύνει την εκτέλεση.

iii). Ένα πεδίο που αναφέρεται σε μεταβλητό αντικείμενο, πρέπει κατά κανόνα να αντιγράφεται βαθιά.

Σε κάθε περίπτωση, η τελική απόφαση ανήκει στον σχεδιαστή του αφηρημένου τύπου καθώς μπορεί να εξαρτάται από άλλες ιδιαιτερότητες που ισχύουν κατά περίπτωση.

Κλωνοποίηση και κατασκευαστές αντιγραφής. Ο προσεκτικός αναγνώστης θα έχει ενδεχομένως προσέξει τη σημαντική ομοιότητα μεταξύ της μεθόδου `clone()` και των κατασκευαστών αντιγραφής που μπορεί να παρέχει ένας αφηρημένος τύπος δεδομένων.

Το ερώτημα που ανακύπτει αφορά τη διαφορά των δύο μηχανισμών αντιγραφής της δομής αντικειμένων. Οι θεωρητικοί του του προγραμματισμού με Java έχουν διατυπώσει διάφορες απόψεις για το θέμα αυτό. Υπάρχουν αρκετοί, όπως για παράδειγμα ο συγγραφέας Douglas Dunn, που πιστεύουν πως οι κατασκευαστές αντιγραφής δεν είναι παρά ένα μικρόβιο που εισχώρησε στον κόσμο της Java, από την C++ και κατά συνέπεια πρέπει να απαγορευτεί. Άλλοι θεωρητικοί, μεταξύ αυτών και ο Joshua Bloch, ένας από τους σχεδιαστές του API της γλώσσας, παραδέχονται πως το πρωτόκολλο που επιβάλλει η κλάση `Object` για την υλοποίηση της `clone()` είναι μάλλον εξωτικό και, κατά συνέπεια, δεν ταιριάζει σε μια αγνή αντικειμενοστρεφή γλώσσα όπως ή Java, αν και ο ίδιος αναγνωρίζει φυσικά το βασικό πλεονέκτημα της `clone()`.

Σε αντίθεση με ένα κατασκευαστή αντιγραφής, η μέθοδος `clone()` είναι πολυμορφική και μπορεί να κληθεί, χωρίς να είναι γνωστός ο τύπος του αντικειμένου. Από την άλλη πλευρά η χρήση της `clone()` παρουσιάζει το μειονέκτημα ότι οποιοσδήποτε τύπος μπορεί, ακυρώνοντάς τη κατάλληλα, να απαγορεύσει την κλωνοποίηση των αντικειμένων του με ότι συνέπειες έχει αυτό για τους απογόνους του.

Οι περισσότεροι τείνουν να συμφωνήσουν με τον Bloch, και συνιστούν προσοχή στη χρήση της `clone()` ακόμη και από έμπειρους προγραμματιστές. Οι δύο μηχανισμοί είναι, σε επίπεδο πρόθεσης

τουλάχιστον, ισοδύναμοι. Επιπλέον, είναι σε ευρεία χρήση. Ένας ορθά σχεδιασμένος αφηρημένος τύπος δεδομένων πρέπει να υποστηρίζει και τους δύο μηχανισμούς εφόσον οι κατάσταση των αντικειμένων του είναι μεταβλητή. Στην αντίθετη περίπτωση, ούτε ο κατασκευαστής αντιγραφής ούτε η μέθοδος `clone()` έχουν σημαντικό λόγο ύπαρξης. Μια προσεκτική ματιά στη σχεδίαση των περισσοτέρων κλάσεων της Java, με εξαίρεση την κλάση `String`, αρκεί για να επιβεβαιώσει την πρακτική αυτή.

Ασκήσεις

2-11 Ολοκληρώστε την υλοποίηση της κλάσης `Circle`, προσθέτοντας δύο ακόμη μεθόδους. Η μέθοδος `boolean contains(Point p)` ελέγχει αν το σημείο `p` ανήκει στον κύκλο, ενώ η μέθοδος `boolean intersects(Circle c)` ελέγχει αν ο κύκλος τέμνεται με τον κύκλο `c`.

2-12 Τροποποιείστε την κλάση `Point` ώστε να υποστηρίζει κλωνοποίηση.

2-13 Σχολιάστε τη σχεδίαση των τύπων του Πίνακα 2-1 στη σελίδα 42, οι οποίοι δεν παρέχουν κατασκευαστές αντιγραφής και επιπλέον δεν υλοποιούν τη διεπαφή `Cloneable`.

▷**2-14** Επαναλάβετε την Άσκηση 2-5 με βάση τα όσα περιγράφονται στις Ενότητες 2.2 και 2.3.

2.4 Σχέσεις διάταξης

Πολλοί αφηρημένοι τύποι δεδομένων είναι *διατεταγμένοι*, μπορούμε δηλαδή να παραθέσουμε τα αντικείμενά τους το ένα μετά το άλλο με μια καθορισμένη σειρά. Πιο τυπικά, λέμε πως ένα σύνολο είναι διατεταγμένο όταν μπορούμε να ορίσουμε μια σχέση διάταξης στα στοιχεία του. Μια διμελής σχέση R είναι μια σχέση διάταξης επί του συνόλου S, όταν κατέχει τις εξής ιδιότητες.

i). Ανακλαστικότητα: $(a, a) \in R$ για κάθε $a \in S$.
ii). Αντισυμμετρία: Αν $(a, b) \in R$ και $(b, a) \in R$ τότε $a = b$.
iii). Μεταβατικότητα: Αν $(a, b) \in R$ και $(b, c) \in R$ τότε $(a, c) \in R$.
iv). Συγκρισιμότητα: Για κάθε $a, b \in S$ ή $(a, b) \in R$ ή $(b, a) \in R$.

Όταν $(a, b) \in R$, λέμε πως το στοιχείο a προηγείται του b στη διάταξη που ορίζει η σχέση R. Πολλές φορές ωστόσο, λέμε άτυπα ότι το a

είναι μικρότερο του b, ή αντίστροφα, ότι το b είναι μεγαλύτερο του a. Όταν δεν μπορούμε να καθορίσουμε τη σχετική σειρά δύο στοιχείων, λέμε πως αυτά είναι ίσα.

Ορισμός 2-8. Μια κλάση που υλοποιεί την διεπαφή `java.lang.Comparable` είναι ένας διατεταγμένος αφηρημένος τύπος δεδομένων.

Η διεπαφή `Comparable` ορίζει μία και μοναδική μέθοδο η οποία πρέπει να υλοποιείται έτσι ώστε να ορίζει μια σχέση διάταξης.

```
public interface Comparable {
   int compareTo(Object o);
}
```

Συγκεκριμένα, η μέθοδος πρέπει να επιστρέφει αρνητική ή θετική τιμή, αν το αντικείμενο μέσω του οποίου καλείται, προηγείται ή έπεται αντίστοιχα του ορίσματος της στη διάταξη που ορίζει η `compareTo`. Αν η διάταξη δεν μπορεί να καθοριστεί για ένα ζεύγος αντικειμένων, η μέθοδος πρέπει να επιστρέφει μηδέν. Πιο τυπικά, μια υλοποίηση της μεθόδου `compareTo` πρέπει να πληροί τις εξής απαιτήσεις.

i). Για κάθε ζεύγος αντικειμένων `x` και `y`, το πρόσημο της τιμής που θα επιστρέψει η κλήση `x.compareTo(y)` πρέπει να είναι αντίθετο του προσήμου που θα επιστρέψει η κλήση `y.compareTo(x)`.

ii). Για κάθε τριάδα αντικειμένων `x`, `y` και `z`, για τα οποία οι κλήσεις `x.compareTo(y)` και `y.compareTo(z)` επιστρέφουν θετική τιμή, η κλήση `x.compareTo(z)` πρέπει να επιστρέφει θετική τιμή.

iii). Για κάθε τριάδα αντικειμένων `x`, `y` και `z`, και εφόσον η κλήση `x.compareTo(y)` επιστρέφει την τιμή `0`, το πρόσημο της τιμής που θα επιστρέψει η κλήση `x.compareTo(z)` πρέπει να είναι ίσο με το πρόσημο της τιμής που θα επιστρέψει η κλήση `y.compareTo(z)`.

Είναι προφανές πως υπάρχει μια ισχυρή σχέση μεταξύ της μεθόδου `equals` της κλάσης `Object`και της μεθόδου `compareTo` της διεπαφής `Comparable`. Και οι δύο αυτές μέθοδοι μπορούν να χρησιμοποιηθούν για να συγκρίνουμε δύο αντικείμενα για ισότητα καθώς αν δεν μπορούμε να διατάξουμε δύο αντικείμενα μεταξύ τους, αυτά πρέπει να θεωρηθούν ίσα. Ωστόσο δεν απαιτείται η `compareTo`

να είναι συνεπής με την equals. Δεν είναι δηλαδή απαραίτητο να ισχύει (x.compareTo(y) == 0) == x.equals(y). Είναι όμως κοινός τόπος, πως η υλοποίηση της compareTo από ένα αφηρημένο τύπο δεδομένων, πρέπει να είναι συνεπής ως προς την υλοποίηση της μεθόδου equals από τον ίδιο τύπο.

Στο δισδιάστατο χώρο —για να έρθουμε στον αφηρημένο τύπο Point— δεν ορίζεται σχέση διάταξης. Αν θα θέλαμε όμως να διατάξουμε ολικά τα αντικείμενα του τύπου αυτού, μπορούμε να υλοποιήσουμε τη διεπαφή Comparable ως εξής.

```
public int compareTo(Object o) {
   Point p = (Point) o;
   return x != p.x ? x - p.x : y - p.y;
}
```

Εύκολα μπορεί να διαπιστώσει κανείς πως η υλοποίηση αυτή ικανοποιεί τις απαιτήσεις που θέτει η διεπαφή Comparable. Ένα σημείο που απαιτεί προσοχή είναι πως αν το όρισμα της μεθόδου δεν είναι τύπου Point, η μέθοδος θα σημάνει μια εξαίρεση τύπου ClassCastException κάτι που είναι απολύτως λογικό και εκφράζει τη διαίσθησή μας σχετικά με την διάταξη αντικειμένων που ανήκουν σε ξένα μεταξύ τους σύνολα.

Συγκριτές. Για μερικούς αφηρημένους τύπους δεδομένων μπορούμε να ορίσουμε περισσότερες από μία σχέσεις διάταξης. Για την κλάση String για παράδειγμα, δύο συμβολοσειρές είναι ίσες όταν έχουν ένα προς ένα ίσους χαρακτήρες. Αν δύο συμβολοσειρές διαφέρουν σε κάποια θέση, τότε προηγείται στη διάταξη η συμβολοσειρά με τον μικρότερο χαρακτήρα στη συγκεκριμένη θέση. Σε μερικές εφαρμογές ωστόσο απαιτείται να μην διακρίνουμε μεταξύ πεζών και κεφαλαίων χαρακτήρων. Πως μπορούμε ωστόσο να αλλάξουμε την σημασιολογία των μεθόδων σύγκρισης και διάταξης για μια κλάση;

Ορισμός 2-9. *Συγκριτής* ονομάζεται κάθε αντικείμενο το οποίο μπορεί να ορίσει μια σχέση διάταξης στα αντικείμενα ενός αφηρημένου τύπου δεδομένων, κατά τις απαιτήσεις της διεπαφής Comparable.

Το παρακάτω απόσπασμα κώδικα, καθορίζει την διεπαφή που θα χρησιμοποιούμε στα επόμενα για να ορίζουμε συγκριτές. Κάθε κλάση που υλοποιεί τη διεπαφή Comparator είναι ένας συγκριτής.

```
package aueb.util.api;

public interface Comparator {
   int compare(Object a, Object b);
}
```

Το πλεονέκτημα της χρήσης συγκριτών είναι ότι έχουμε τη δυνατότητα να ορίσουμε περισσότερες από μία διατάξεις. Ο κανόνας είναι ότι αν υπάρχει μια φυσική έννοια διάταξης για τα αντικείμενα ενός τύπου, την ευθύνη ορισμού της διάταξης φέρει ο ίδιος ο τύπος. Αν ωστόσο για οποιοδήποτε λόγο η φυσική διάταξη δεν εξυπηρετεί τους σκοπούς μιας εφαρμογής, τότε μπορούμε να ορίσουμε διαφορετικές διατάξεις χρησιμοποιώντας συγκριτές.

Ασκήσεις

●**2-15** Αποδείξτε πως ο αφηρημένος τύπος δεδομένων `Point` είναι διατεταγμένος.

▷**2-16** Υλοποιήστε ένα συγκριτή αντικειμένων τύπου `char[]`.

▷**2-17** Ανακεφαλαιώνοντας τα όσα ειπώθηκαν στις Ενότητες 2.1–2.4, δώστε μια ολοκληρωμένη υλοποίηση της κλάσης `Point`.

2-18 Υλοποιήστε ένα αφηρημένο τύπο για την παράσταση ρητών αριθμών.

2-19 Μελετήστε τη σχεδίαση και την υλοποίηση των τύπων του Πίνακα 2-1 στη σελίδα 42 και συγκρίνετε με τους τύπους `Point` και `Complex`.

2.5 Δομές δεδομένων

Μέχρι στιγμής έχουμε δει αφηρημένους τύπους δεδομένων που παριστάνουν κατά κανόνα τιμές. Τα αντικείμενα των τύπων `Point` και `Complex` είναι μεν σύνθετα αντικείμενα, καθώς αποτελούνται από δύο ή περισσότερα πεδία, παριστάνουν ωστόσο μια και μοναδική οντότητα, ένα συγκεκριμένο δεδομένο είτε αυτό είναι σημείο, μιγαδικός ή ρητός αριθμός. Αν και η ορθή σχεδίαση τέτοιου είδους δεδομένων παρουσιάζει αρκετές μικρές προκλήσεις όπως είδαμε στις προηγούμενες ενότητες, το ενδιαφέρον μας εστιάζεται σε μια διαφορετική κατηγορία αφηρημένων τύπων δεδομένων που παραδοσιακά ονομάζονται *δομές δεδομένων*.

Μια δομή δεδομένων είναι ένα δυναμικό σύνολο αντικειμένων. Σε αντίθεση με ένα σημείο του επιπέδου ή γενικότερα ένα διάνυσμα, το οποίο αποτελείται από ένα αριθμό συνιστωσών, το πλήθος των οποίων είναι γνωστό κατά την κατασκευή του και παραμένει εφεξής αμετάβλητο, μια δομή δεδομένων είναι μια ομαδοποίηση αντικειμένων που παρουσιάζει σημαντική δυναμικότητα. Ένα αντικείμενο μπορεί να είναι *μέλος*, ή αλλιώς *στοιχείο*, μιας δομής δεδομένων για κάποιο χρονικό διάστημα της εκτέλεσης ενός προγράμματος, αλλά στη συνέχεια να αφαιρεθεί από τη δομή και ενδεχομένως να εισαχθεί σε μια άλλη δομή. Ένα αντικείμενο μπορεί επιπλέον να είναι μέλος περισσότερων από μιας δομών δεδομένων κατά την ίδια χρονική στιγμή.

Για την υλοποίηση τέτοιων αφηρημένων τύπων δεδομένων απαιτούνται εξαιρετικά ευέλικτοι μηχανισμοί οργάνωσης. Επιπλέον, όπως συμβαίνει σε κάθε εφαρμογή λογισμικού, η λειτουργία αυτών των τύπων πρέπει να είναι αποτελεσματική κατά περίπτωση, και ειδικά όταν το πλήθος των αντικειμένων είναι πολύ μεγάλο. Οι μορφές οργάνωσης που έχουν αναπτυχθεί ή επινοηθεί τα τελευταία πενήντα χρόνια προκειμένου να χρησιμοποιηθούν στην υλοποίηση αποτελεσματικών δομών δεδομένων είναι πραγματικά εκπληκτική. Πολλές από τις οργανώσεις αυτές είναι γενικές, και έχουν κατά συνέπεια ευρύτατο πεδίο εφαρμογών, ενώ άλλες είναι πιο ειδικές και η χρήση τους περιορίζεται σε συγκεκριμένες μόνο εφαρμογές.

Παρά την ύπαρξη αυτής της ποικιλότητας ωστόσο, τα λειτουργικά χαρακτηριστικά της συντριπτικής πλειοψηφίας όλων αυτών των μορφών οργάνωσης δεδομένων μπορούν να συμπυκνωθούν σε τρεις θεμελιώδεις λειτουργίες. Ανεξάρτητα από τον τρόπο με τον οποίο να δυναμικό σύνολο αντικειμένων οργανώνεται σε δομή, πρέπει να μπορούμε να *προσθέτουμε* και να *αφαιρούμε* αντικείμενα. Πρέπει ακόμη να μπορούμε να διαπιστώσουμε αν ένα δοσμένο αντικείμενο ανήκει ή όχι σε μια δομή. Αυτές οι λειτουργίες ονομάζονται αντίστοιχα *εισαγωγή*, *διαγραφή* και *αναζήτηση*, και είναι το αντικείμενο που θα μας απασχολήσει στα επόμενα κεφάλαια.

Αν και χρησιμοποιήσαμε τον όρο *σύνολο* για τον ορισμό της έννοιας της δομής δεδομένων, δεν έχουν όλες οι δομές τα χαρακτηριστικά του συνόλου όπως το γνωρίζουμε από τα μαθηματικά. Πολλές φορές ωστόσο είναι πρόσφορη η αντιστοίχιση των τριών βασικών λειτουργιών μιας δομής δεδομένων με τις αντίστοιχες πράξεις

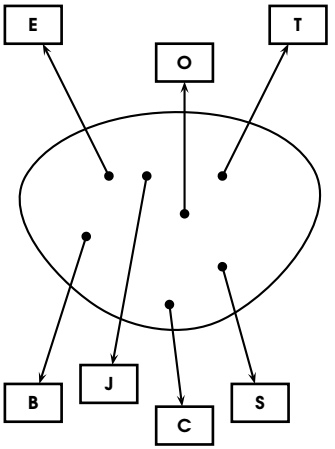
επί συνόλων. Αν S είναι μια δομή δεδομένων τότε η εισαγωγή ενός αντικειμένου o σε αυτή, δημιουργεί τη δομή $S \cup \{o\}$. Αντίστοιχα η διαγραφή ενός αντικειμένου o από τη δομή S είναι ανάλογη της πράξης $S - \{o\}$. Τέλος η αναζήτηση είναι η πράξη $o \in S$.

Εναλλακτικά θα μπορούσε κανείς να θεωρήσει πως η αναζήτηση μπορεί να αντικατασταθεί από την *απαρίθμηση*. Αν μπορούμε να απαριθμήσουμε τα αντικείμενα τα οποία ανήκουν σε μια δομή, τότε με ένα αλγόριθμο όπως αυτόν της ακολουθιακής αναζήτησης, θα μπορούσαμε να διαπιστώσουμε αν ένα δοσμένο αντικείμενο ανήκει στη δομή ή όχι. Αυτή η προσέγγιση είναι μεν ορθή, αλλά ενδεχομένως περιοριστική. Εφόσον η αναζήτηση ενός αντικειμένου σε μια δομή μπορεί να γίνει με τρόπο τέτοιο ώστε να μειώνεται ο χρόνος αναζήτησης, η απαρίθμηση αν και σημαντική λειτουργία, πρέπει να παρακαμφθεί.

Το πιο συναρπαστικό στην υλοποίηση μιας οργάνωσης δεδομένων είναι η επίτευξη υψηλής αποτελεσματικότητας ειδικά σε ότι αφορά το χρόνο εκτέλεσης των βασικών λειτουργιών. Αυτή ποικίλει μεταξύ $O(1)$ και $O(N)$ ή και $O(N \lg N)$ σε μερικές περιπτώσεις. Σε κάθε περίπτωση ωστόσο, το πρωτόκολλο επικοινωνίας μπορεί να είναι κοινό. Ας μην ξεχνάμε, μια δομή δεδομένων δεν παύει να είναι ένας αφηρημένος τύπος δεδομένων. Κατά συνέπεια πρέπει να έχουμε ένα ομοιόμορφο τρόπο επικοινωνίας με τέτοιου είδους αντικείμενα. Για το λόγο αυτό θα περιγράψουμε με περισσότερη λεπτομέρεια πια, τέσσερις κατηγορίες αφηρημένων δομών δεδομένων πριν ξεκινήσουμε την προσπάθεια υλοποίησής τους.

2.6 Συλλογές

Η γενικότερη μορφή ομαδοποίησης δεδομένων είναι η *συλλογή* (collection). Σε ορισμένες περιπτώσεις ο όρος συλλογή θεωρείται συνώνυμο του όρου δομή δεδομένων. Επειδή ωστόσο υπάρχουν περιπτώσεις δομών δεδομένων που δεν παρέχουν κάποιες από τις βασικές λειτουργίες όπως για παράδειγμα η αναζήτηση, θα θεωρήσουμε τη συλλογή ως ειδική περίπτωση δομής δεδομένων. Πολλές φορές μια συλλογή αναφέρεται και ως *πίνακας συμβόλων* (symbol table). Και αυτός ο όρος ωστόσο έχει κάπως ειδική σημασιολογία, ειδικά στο πεδίο των γλωσσών προγραμματισμού, και έτσι θα τον



Σχήμα 2-2: Μια αφηρημένη αναπαράσταση συλλογής αντικειμένων.
Τα αντικείμενα τα οποία περιλαμβάνει η δομή, δεν αποτελούν φυσικό τμήμα της. Η δομή αποτελείται από το μηχανισμό οργάνωσης ο οποίος χρησιμοποιείται για τη συγκρότησή της.

αποφύγουμε.

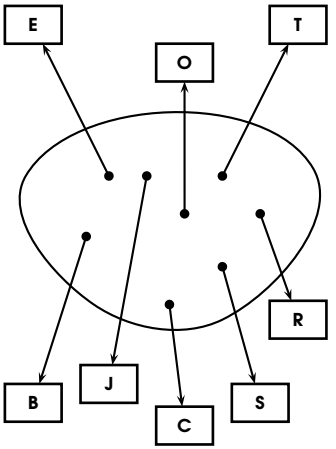
Μια συλλογή παρέχει τις βασικές λειτουργίες της εισαγωγής, διαγραφής, αναζήτησης και απαρίθμησης. Επιπλέον, μια συλλογή παρέχει μερικές χρήσιμες λειτουργίες όπως για παράδειγμα το πλήθος των αντικειμένων που περιλαμβάνει μια δεδομένη χρονική στιγμή. Επειδή οι συλλογές όπως και πολλές άλλες αφηρημένες δομές δεδομένων έχουν πολλά από τα χαρακτηριστικά ενός συνόλου, παρέχουν λειτουργίες με σημασιολογία ανάλογη αυτής των πράξεων ένωσης, τομής και διαφοράς συνόλων.

Οι μέθοδοι που ορίζει η διεπαφή `Collection`, η οποία παρουσιάζεται στον Κώδικα 2.1, καθορίζουν τις λειτουργίες που πρέπει να παρέχει κάθε υλοποίηση συλλογής. Ο μηχανισμός οργάνωσης που θα επιλεχθεί για την υλοποίηση δεν ενδιαφέρει ιδιαίτερα τους χρήστες της διεπαφής, εκτός ίσως από την αποτελεσματικότητα που προσφέρει στις μεθόδους που ορίζονται από τη διεπαφή. Με την υλοποίηση αυτής της διεπαφής θα ασχοληθούμε στα επόμενα κεφάλαια. Εδώ απλώς θα περιγράψουμε τις λειτουργίες που αυτός ο αφηρημένος τύπος δεδομένων προσφέρει.

Κώδικας 2.1: Η διεπαφή `Collection`.

```
1  package aueb.util.api;
2  /**
3   * A collection is a group of objects known as its elements. It provides
4   * three basic operations: insertion, removal, and search.
5   * Any class that implements this interface does not necessarily
6   * guarantee the running time of the methods of this interface. Some
7   * implementations are fast, while others are not.
8   */
9  public interface Collection {
10     /**
11      * Adds an object in this collection.
12      * @param object The object to add.
13      * @return true iff the object was actually added.
14      */
15     boolean add(Object object);
16     /**
17      * Removes an object from this collection.
18      * @param object The object to remove.
19      * @return true iff the object was actually added.
20      */
21     boolean remove(Object object);
22     /**
```

```
23      * Searches for an object in this collection.
24      * @param object The object to search for.
25      * @return true iff there is an object o in this collection such
26      * that element.equals(o).
27      */
28     boolean contains(Object object);
29     /**
30      * The size of this collection.
31      * @return The number of objects in this collection.
32      */
33     int size();
34     /**
35      * Decides whether this collection is empty.
36      * @return true iff there is no element in this collection.
37      */
38     boolean isEmpty();
39     /**
40      * Removes any objects stored in this collection.
41      */
42     void clear();
43     /**
44      * Decides whether this collection is a superset of given collection.
45      * @param c Another collection.
46      * @return true if this instance contains every element of c.
47      */
48     boolean containsAll(Collection c);
49     /**
50      * Adds every element of c to this collection (union).
51      * @param c Another collection.
52      */
53     void addAll(Collection c);
54     /**
55      * Removes every element of c contained in this collection (difference).
56      * @param c Another collection.
57      */
58     void removeAll(Collection c);
59     /**
60      * Removes every element that is not an element of c (intersection).
61      * @param c Another collection.
62      */
63     void retainAll(Collection c);
64     /**
65      * Creates an enumerator on this collection.
66      * @return An enumerator on this collection.
67      */
68     Enumerator enumerator();
```



Σχήμα 2-3: Εισαγωγή αντικειμένου σε μια συλλογή.
Το σχήμα απεικονίζει με αφηρημένο τρόπο την εισαγωγή της συμβολοσειράς **R** στη συλλογή του Σχήματος 2-2.

```
69 | }
```

Αναζήτηση. Η μέθοδος `contains(Object)` της διεπαφής `Collection` υλοποιεί τη λειτουργία της αναζήτησης αντικειμένου στη συλλογή. Μας επιτρέπει να γνωρίζουμε αν ένα αντικείμενο ανήκει στη συλλογή σε μια δεδομένη χρονική στιγμή. Για να άρουμε την ασάφεια σχετικά με την έννοια της λέξης *ανήκει*, δίνουμε τον παρακάτω ορισμό.

Ορισμός 2-10. Θα λέμε ότι ένα αντικείμενο `o`, ανήκει στη συλλογή `c`, αν υπάρχει αντικείμενο `e` στην `c`, τέτοιο ώστε `e.equals(o)`.

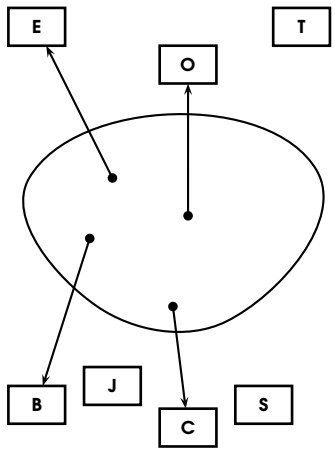
Το όρισμα `object` της μεθόδου `contains` είναι μια αναφορά στο ζητούμενο αντικείμενο και το αποτέλεσμά της είναι `true` ή `false` αν αντίστοιχα το αντικείμενο αυτό ανήκει στη συλλογή ή όχι. Αν `c` είναι η συλλογή του Σχήματος 2-2 για παράδειγμα, η κλήση `c.contains("J")` θα επιστρέψει `true`, ενώ η κλήση `c.contains("M")` θα επιστρέψει `false`.

Εισαγωγή. Η μέθοδος `add(Object)`, εισάγει αντικείμενα στη συλλογή. Το όρισμα `object`, είναι μια αναφορά στο αντικείμενο που πρέπει να εισαχθεί στη συλλογή. Η μέθοδος επιστρέφει `true` ή `false` ανάλογα με το αν το αντικείμενο εισήχθηκε τελικά στη συλλογή ή όχι. Αυτό φαίνεται εκ πρώτης όψεως παράξενο ή και ενδεχομένως περιττό, αλλά είναι απαραίτητο. Μια συλλογή μπορεί να επιτρέπει ή να απαγορεύει *διπλότυπα στοιχεία* (duplicates). Στην πρώτη περίπτωση, περισσότεροι από ένας κλώνοι ενός αντικειμένου μπορούν να συμμετέχουν στη συλλογή. Στην αντίθετη περίπτωση, η εισαγωγή ενός κλώνου δεν θα επιτραπεί. Για παράδειγμα, μια συλλογή που παριστάνει μια κληρωτίδα, δεν θα πρέπει να δέχεται δύο φορές τον ίδιο αριθμό λαχνού. Σε μια τέτοια περίπτωση η μέθοδος `add` πρέπει να τερματίσει επιστρέφοντας την τιμή `false`.

Διαγραφή. Η μέθοδος `remove` αφαιρεί ένα αντικείμενο από τη συλλογή. Το όρισμα `object` είναι μια αναφορά στο αντικείμενο που πρέπει να διαγραφεί. Αν βρεθεί στη συλλογή αντικείμενο `e` τέτοιο ώστε `e.equals(object)`, τότε το αντικείμενο αυτό θα αφαιρεθεί από

τη συλλογή και η μέθοδος θα επιστρέψει την τιμή `true`. Διαφορετικά η μέθοδος θα επιστρέψει την τιμή `false`.

Απαρίθμηση στοιχείων. Έχουμε ήδη αναφέρει πως η απαρίθμηση των στοιχείων μιας δομής δεδομένων είναι θεμελιώδης λειτουργία. Στην πραγματικότητα, όπως θα δούμε αμέσως μετά, όλες οι υπόλοιπες λειτουργίες της διεπαφής `Collection` μπορούν να υλοποιηθούν μέσω των λειτουργιών εισαγωγής, διαγραφής και απαρίθμησης. Το πρόβλημα ωστόσο είναι πως προκειμένου να απαριθμήσουμε τα στοιχεία μιας συλλογής πρέπει να γνωρίζουμε τη δομή της, κάτι το οποίο είτε θέλουμε να αποφύγουμε, είτε δεν μπορούμε να επιτύχουμε. Μπορούμε βεβαίως να παρακάμψουμε αυτή τη δυσκολία, αν κάθε υλοποίηση συλλογής παρέχει ένα μηχανισμό για τη διάσχισή της. Αυτός ακριβώς είναι ο σκοπός της μεθόδου `enumerator()` η οποία είναι μια μέθοδος κατασκευής του τύπου `Enumerator`.



Σχήμα 2-4: Διαγραφή αντικειμένου από μια συλλογή.
Το σχήμα απεικονίζει τη συλλογή του Σχήματος 2-2 έπειτα από τη διαγραφή των αντικειμένων **J** **S** και **T**.

Κώδικας 2.2: Η διεπαφή `Enumerator`.

```
1  package aueb.util.api;
2  /**
3   * An enumerator is an object that knows how to traverse a collection
4   * in order to enumerate its elements.
5   */
6  public interface Enumerator {
7      /**
8       * Tests whether there are more elements to enumerate.
9       * @return <code>true</code> iff there are more elements to enumerate.
10      */
11     boolean hasNext();
12     /**
13      * Advances to the next element in the collection being enumerated.
14      * @throws IllegalStateException When called while hasNext() is false.
15      * @return The next element in the collection being enumerated.
16      */
17     Object next();
18 }
```

Κάθε αντικείμενο του τύπου `Enumerator` είναι ένας *απαριθμητής*· ένα αντικείμενο που γνωρίζει τις δομικές λεπτομέρειες μιας συλλογής και κατά συνέπεια μπορεί να τη "διασχίσει" και να απαριθμήσει τα αντικείμενα τα οποία βρίσκονται αποθηκευμένα σε αυτή. Για να διασχίσουμε μια συλλογή, πρέπει πρώτα να καλέσουμε τη μέθοδο `enumerator()` ώστε να αποκτήσουμε πρόσβαση σε ένα απαριθμητή.

$x \in A$	`a.contains(x)`
$A \leftarrow A \cup \{x\}$	`a.add(x)`
$A \leftarrow A - \{x\}$	`a.remove(x)`
$A \supseteq B$	`a.containsAll(b)`
$A \leftarrow A \cup B$	`a.addAll(b)`
$A \leftarrow A - B$	`a.removeAll(b)`
$A \leftarrow A \cap B$	`a.retainAll(b)`
$\|A\|$	`a.size()`
$\|A\| = 0$	`a.isEmpty()`
$A \leftarrow \emptyset$	`a.clear()`

Σχήμα 2-5: Πράξεις συνόλων και μέθοδοι της διεπαφής `Collection`.
Ο πίνακας δείχνει την αντιστοίχιση των βασικών πράξεων της θεωρίας συνόλων και των μεθόδων της διεπαφής `Collection`. Οι μεταβλητές `a` και `b` είναι αναφορές τύπου `Collection`, ενώ η μεταβλητή `x` είναι τύπου `Object`.

Στη συνέχεια με αλλεπάλληλες κλήσεις στη μέθοδο `next()` μπορούμε να εντοπίσουμε τα αντικείμενα της συλλογής, ένα κάθε φορά.

Η μέθοδος `hasNext()` επιστρέφει `true` ή `false` ανάλογα με το αν υπάρχουν στοιχεία τα οποία δεν έχουν ακόμη εντοπιστεί από τον απαριθμητή. Η μέθοδος `next()` επιστρέφει το "επόμενο" αντικείμενο της συλλογής. Δεν είναι ωστόσο πάντα ασφαλές να καλέσουμε τη μέθοδο αυτή. Αν η πιο πρόσφατη κλήση της `hasNext` έχει επιστρέψει `false`, δεν πρέπει να επιχειρήσουμε να καλέσουμε τη μέθοδο `next`. Κάτι τέτοιο θα έχει σαν αποτέλεσμα τη σήμανση μιας εξαίρεσης τύπου `IllegalStateException`. Μια υλοποίηση της διεπαφής `Enumerator` πρέπει να φροντίζει ώστε αν η μέθοδος `hasNext` επιστρέψει `true`, η αμέσως επόμενη κλήση της `next` να μην αποτύχει.

Βοηθητικές λειτουργίες. Οι μέθοδοι που περιγράψαμε μέχρι στιγμής υποστηρίζουν τις βασικές λειτουργίες μιας συλλογής: αναζήτηση, εισαγωγή, διαγραφή και απαρίθμηση των στοιχείων της. Οι μέθοδοι `size`, `isEmpty` και `clear` υλοποιούν λιγότερο σημαντικές, αλλά πολύ χρήσιμες λειτουργίες. Η μέθοδος `size` επιστρέφει το πλήθος των στοιχείων της συλλογής. Η μέθοδος `isEmpty`, επιστρέφει `true` αν και μόνο αν η μέθοδος `size` επιστρέφει `0`. Τέλος η μέθοδος `clear`, διαγράφει όλα τα στοιχεία της συλλογής.

Πράξεις συνόλων. Η διεπαφή `Collection` παρέχει τέσσερις ακόμη μεθόδους που υλοποιούν πράξεις μεταξύ συνόλων. Αυτές είναι οι μέθοδοι `containsAll`, `addAll`, `removeAll` και `retainAll`. Οι αντίστοιχες πράξεις συνόλων δίνονται στον Πίνακα 2-5. Για παράδειγμα η πράξη $A \cup B$ αντιστοιχεί στην κλήση `a.addAll(b)`. Το αντικείμενο μέσω του οποίου εκτελείται η μέθοδος, δηλαδή η μεταβλητή `a` στο παράδειγμά μας, παίζει το ρόλο του αριστερού ορίσματος του τελεστή $\cup$, το όρισμα της της μεθόδου, η μεταβλητή `b`, είναι το δεξί όρισμα του τελεστή. Όλες οι μέθοδοι της διεπαφής `Collection` μπορούν να υλοποιηθούν με τη χρήση ενός απαριθμητή και των μεθόδων `add` και `remove` όπως δείχνει η αφηρημένη κλάση `AbstractCollection` που παρουσιάζεται στον Κώδικα 2.3.

Κώδικας 2.3: Η κλάση `AbstractCollection`.

```
1 package aueb.util.api;
2 /**
```

```
3  * Partial implementation of {@link aueb.util.api.Collection}. Most
4  * method implementations are expected to be extremely slow.
5  * Concrete subclasses should override those implementations in order
6  * to improve effectiveness.
7  */
8  public abstract class AbstractCollection implements Collection {
9     public boolean contains(Object element) {
10       Enumerator e = enumerator();
11       while (e.hasNext()) {
12          if (e.next().equals(element)) return true;
13       }
14       return false;
15    }
16    public boolean isEmpty() {
17       return size() == 0;
18    }
19    public void clear() {
20       Enumerator e = enumerator();
21       while (e.hasNext()) remove(e.next());
22    }
23    public boolean containsAll(Collection c) {
24       Enumerator e = c.enumerator();
25       while (e.hasNext()) {
26          if (!contains(e.next())) return false;
27       }
28       return true;
29    }
30    public void addAll(Collection c) {
31       Enumerator e = c.enumerator();
32       while (e.hasNext()) add(e.next());
33    }
34    public void removeAll(Collection c) {
35       Enumerator e = c.enumerator();
36       while (e.hasNext()) remove(e.next());
37    }
38    public void retainAll(Collection c) {
39       Enumerator e = enumerator();
40       while (e.hasNext()) {
41          Object element = e.next();
42          if (!c.contains(element)) remove(element);
43       }
44    }
45    public boolean equals(Object object) {
46       if (this == object) return true;
47       if (!(object instanceof Collection)) return false;
48       Collection c = (Collection) object;
```

```
49        return size() == c.size() && containsAll(c);
50     }
51     public int hashCode() {
52        int hash = 17;
53        Enumerator e = enumerator();
54        while (e.hasNext()) hash = 17 * hash + e.next().hashCode();
55        return hash;
56     }
57  }
```

Ασκήσεις

2-20 Ποιά είναι τα αντικείμενα μιας συλλογή που επιτρέπει διπλότυπα έπειτα από την εκτέλεση των πράξεων **+1 -9 +5 + 6 +1 -7 +8 -1 -1 + 3** όπου το σύμβολο **+** υποδεικνύει εισαγωγή ενώ το σύμβολο - διαγραφή;

2-21 Αν η μεταβλητή c είναι αναφορά σε αντικείμενο τύπου `Collection` που δεν επιτρέπει διπλότυπα και περιέχει τα αντικείμενα **1 2 3 5 6** ενώ η μεταβλητή s αναφέρεται στην συλλογή **0 1 2 3 4 5 6 7 8 9**, ποιο είναι το αποτέλεσμα των κλήσεων `c.addAll(s)` και `s.retainAll(c)`;

2-22 Αν οι συλλογές a και b περιλαμβάνουν αντίστοιχα τα αντικείμενα **A B K C J L** και **K D E J L F M**, προσδιορίστε το αποτέλεσμα των παρακάτω κλήσεων.

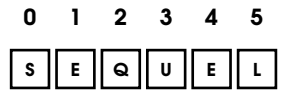
i). `a.containsAll(b)`
ii). `a.addAll(b)`
iii). `b.retainAll(b)`
iv). `a.removeAll(b)`
v). `b.removeAll(a)`
vi). `b.containsAll(a)`

2-22 Προσθέστε στην κλάση `AbstractCollection` τη μέθοδο `void symdif(Collection c)` η οποία υπολογίζει τη συμμετρική διαφορά δύο συνόλων. Η κλήση `a.symdiff(b)` δηλαδή αντιστοιχεί στην πράξη $A \leftarrow (A - B) \cup (B - A)$.

2.7 Ακολουθίες

Ορισμένες οργανώσεις δεδομένων, όπως για παράδειγμα οι μεταθέσεις αριθμών ή άλλων αντικειμένων, δεν μπορούν να παρασταθούν από μια συλλογή καθώς τα στοιχεία μιας συλλογής δεν έχουν συγκεκριμένη θέση. Για τον χειρισμό τέτοιου είδους δεδομένων θα πρέπει

να ορίσουμε μια δομή τα στοιχεία της οποίας είναι τοπολογικά διατεταγμένα με βάση τη θέση στην οποία βρίσκονται αποθηκευμένα.



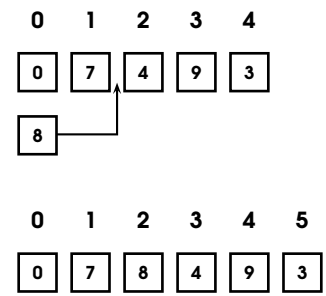
Σχήμα 2-6: Μια αφηρημένη αναπαράσταση ακολουθίας.
Οι θέσεις της ακολουθίας αριθμούνται από τα αριστερά προς τα δεξιά, με την πρώτη θέση να έχει αριθμό 0.

Ορισμός 2-11. Μια *ακολουθία* N στοιχείων είναι μια διατεταγμένη συλλογή. Σε κάθε στοιχείο της συλλογής αντιστοιχεί ένας μη–αρνητικός ακέραιος ο οποίος καθορίζει τη *θέση* του στοιχείου στην ακολουθία. Το πρώτο στοιχείο της ακολουθίας είναι το αντικείμενο στη θέση 0, ενώ το τελευταίο στοιχείο είναι το αντικείμενο στη θέση $N - 1$.

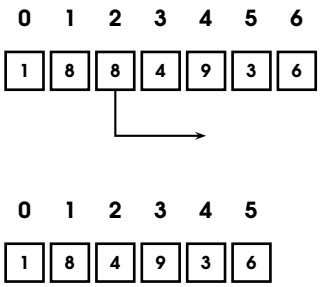
Κάθε στοιχείο μιας ακολουθίας είναι τοποθετημένο σε μια αριθμημένη θέση. Κατά σύμβαση θεωρούμε πως τα στοιχεία της ακολουθίας είναι παρατεταγμένα σε μια οριζόντια γραμμή. Για το λόγο αυτό, πολλές φορές, οι ακολουθίες χαρακτηρίζονται και ως *γραμμικές δομές δεδομένων*. Κατά σύμβαση επίσης, θεωρούμε πως το πρώτο στοιχείο της ακολουθίας, το αντικείμενο στη θέση 0, είναι το αριστερότερο αντικείμενο, ενώ το τελευταίο στοιχείο βρίσκεται στο δεξιό άκρο της ακολουθίας. Το Σχήμα 2-6 απεικονίζει μια ακολουθία με βάση αυτές τις συμβάσεις.

Επειδή μια ακολουθία είναι μια ειδική συλλογή, όλες οι λειτουργίες που παρέχει μια συλλογή πρέπει να παρέχονται και από μία ακολουθία. Επιπρόσθετα, μια ακολουθία πρέπει να υποστηρίζει λειτουργίες για χειρισμό αντικειμένων με βάση τη θέση τους στην ακολουθία. Τέτοιες λειτουργίες είναι για παράδειγμα η εισαγωγή ενός αντικειμένου σε μια δοσμένη θέση της ακολουθίας, καθώς επίσης η διαγραφή και η αντικατάσταση του στοιχείου που βρίσκεται σε μια δοσμένη θέση. Ο εντοπισμός αντικειμένου με βάση τη θέση του, καθώς και ο εντοπισμός της θέσης ενός αντικειμένου είναι ιδιαίτερα χρήσιμες λειτουργίες. Ο Κώδικας 2.4 δίνει τη συνολική εικόνα των λειτουργιών που παρέχει μια ακολουθία.

Η μέθοδος `add(int, Object)` προσθέτει το αντικείμενο `object` στη θέση `position`, της ακολουθίας. Η τιμή του ορίσματος `position` πρέπει να είναι μεταξύ των τιμών 0 και `size()` συμπεριλαμβανομένων των τιμών αυτών. Σε διαφορετική περίπτωση, η μέθοδος σημαίνει μια εξαίρεση τύπου `IndexOutOfBoundsException`. Η μέθοδος `remove(int)`, διαγράφει το αντικείμενο που βρίσκεται στη θέση `position` και στη συνέχεια επιστρέφει μια αναφορά σε αυτό. Η τιμή του ορίσματος `position` πρέπει να είναι μεταξύ των



Σχήμα 2-7: Εισαγωγή αντικειμένου σε ακολουθία.



Σχήμα 2-8: Διαγραφή αντικειμένου από ακολουθία.

τιμών 0 και `size()-1` συμπεριλαμβανομένων των τιμών αυτών. Σε διαφορετική περίπτωση, η μέθοδος σημαίνει την εξαίρεση τύπου `IndexOutOfBoundsException`. Η μέθοδος `indexOf`, επιστρέφει τη θέση της ακολουθίας στην οποία βρίσκεται αποθηκευμένο ένα αντικείμενο ίσο με το αντικείμενο `object`. Αν δεν υπάρχει τέτοιο αντικείμενο στην ακολουθία, η μέθοδος επιστρέφει την τιμή -1. Αν ωστόσο υπάρχουν περισσότερα από ένα τέτοια στοιχεία, η επιστρεφόμενη τιμή εξαρτάται από την υλοποίηση. Κατά κανόνα επιστρέφεται ο αριθμός της αριστερότερης θέσης της ακολουθίας στην οποία εντοπίστηκε ισότητα. Η μέθοδος `get`, επιστρέφει το στοιχείο στη θέση `position`, ενώ η `set`, αλλάζει το στοιχείο που βρίσκεται στη συγκεκριμένη θέση επιστρέφοντας το αντικείμενο το οποίο προηγουμένως βρισκόταν εκεί. Και στις δύο περιπτώσεις, η τιμή του ορίσματος `position` πρέπει να είναι μεταξύ των τιμών 0 και `size()-1` συμπεριλαμβανομένων των τιμών αυτών. Σε διαφορετική περίπτωση, η μέθοδος σημαίνει μια εξαίρεση τύπου `IndexOutOfBoundsException`.

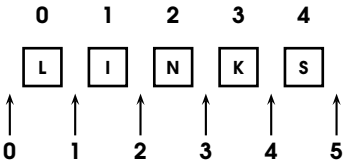
Κώδικας 2.4: Η διεπαφή Sequence.

```
1  package aueb.util.api;
2  /**
3   * A sequence is an ordered collection of objects.
4   * All methods except {@link #indexOf(Object)} throw an instance
5   * of {@link java.lang.IndexOutOfBoundsException} when passed
6   * an index of an invalid position.
7   */
8  public interface Sequence extends Collection{
9      /**
10      * Inserts an object at a given position of this sequence.
11      * @param position The position.
12      * @param object The object to insert.
13      */
14     void add(int position, Object object);
15     /**
16      * Removes the object stored at a given position of this sequence.
17      * @param position The position.
18      * @return true iff the object was actually added.
19      */
20     Object remove(int position);
21     /**
22      * Locates the position a given object is stored at.
23      * @param object The object to search for.
24      * @return The position of object or -1 if object was not found.
25      */
```

```
26    int indexOf(Object object);
27    /**
28     * Retrieves the object stored at a given position.
29     * @param position The position.
30     * @return The object stored at the specified position.
31     */
32    Object get(int position);
33    /**
34     * Replaces the object stored at a given position.
35     * @param position The position.
36     * @param object The object to store.
37     * @return The replaced object.
38     */
39    Object set(int position, Object object);
40    /**
41     * Creates an iterator on this sequence.
42     * @return An iterator positioned before the first object.
43     */
44    Iterator iterator();
45    /**
46     * Creates an iterator on this sequence.
47     * @return An iterator positioned before the i-th object.
48     */
49    Iterator iterator(int position);
50 }
```

Δρομείς. Η διάσχιση ακολουθιών διαφοροποιείται σημαντικά σε σχέση με τη διάσχιση συλλογών. Κατ' αρχήν η διάσχιση μιας ακολουθίας μπορεί να γίνει με κατεύθυνση από τα αριστερά προς τα δεξιά ή αντίστροφα. Καθώς διασχίζουμε την ακολουθία, θέλουμε επιπλέον να γνωρίζουμε τη θέση στην οποία έχει φτάσει η διάσχιση. Τέλος θέλουμε να εισάγουμε, να διαγράφουμε και να αντικαθιστούμε αντικείμενα κατά τη διάρκεια της διάσχισης. Για να ικανοποιήσουμε τις απαιτήσεις αυτές θα επεκτείνουμε το μηχανισμό του απαριθμητή εισάγοντας μια κατασκευή που ονομάζεται *δρομέας* και είναι καταλληλότερη για τη διάσχιση ακολουθιών καθώς, εκτός από τις λειτουργίες ενός απαριθμητή, υποστηρίζει τις προαναφερθείσες λειτουργίες.

Ένας δρομέας ακολουθίας με N στοιχεία έχει $N + 1$ πιθανές θέσεις. Όταν ο δρομέας είναι στη θέση k, $k = 1, \ldots, N - 2$, βρίσκεται *μεταξύ των αντικειμένων* που είναι αποθηκευμένα στις θέσεις $k - 1$ και k της ακολουθίας. Συμβατικά ορίζουμε πως η θέση 0 ενός



Σχήμα 2-9: Ένας δρομέας ακολουθίας με N στοιχεία έχει $N+1$ πιθανές θέσεις.
Όταν ο δρομέας είναι στη θέση k, η επόμενη κλήση της μεθόδου `nextIndex` θα επιστρέψει την τιμή k ενώ η επόμενη κλήση της μεθόδου `previousIndex` θα επιστρέψει την τιμή $k-1$.

δρομέα βρίσκεται πριν από το πρώτο αντικείμενο, ενώ η θέση $N-1$ βρίσκεται αμέσως μετά το τελευταίο αντικείμενο. Η αντιστοιχία μεταξύ των θέσεων ενός δρομέα και των αντικειμένων της ακολουθίας που διασχίζει, παρουσιάζεται στο Σχήμα 2-9.

Η διεπαφή που πρέπει να υποστηρίζει κάθε δρομέας δίνεται στον Κώδικα 2.5. Οι κανόνες που διέπουν τη λειτουργία ενός δρομέα `i` μιας ακολουθίας `s` με N στοιχεία είναι οι εξής:

i). Ο δρομέας που επιστρέφει η κλήση `s.iterator()` είναι τοποθετημένος στη θέση 0, ενώ ο δρομέας που επιστρέφει η κλήση `s.iterator(k)`, είναι τοποθετημένος στη θέση k.

ii). Αν ο δρομέας είναι στη θέση 0, οι κλήσεις `i.hasPrevious()` και `i.previousIndex()` θα επιστρέψουν αντίστοιχα `false` και -1, ενώ η κλήση `i.previous()` θα σημάνει μια εξαίρεση τύπου `IllegalStateException`. Αντίθετα, οι κλήσεις `i.hasNext()` και `i.nextIndex()` θα επιστρέψουν `true` και 0 αντίστοιχα, ενώ η κλήση `i.next()` θα επιστρέψει το αντικείμενο στη θέση 0 της ακολουθίας.

iii). Αν ο δρομέας είναι στη θέση N, οι κλήσεις `i.hasNext()` και `i.nextIndex()` θα επιστρέψουν αντίστοιχα `false` και $N+1$, ενώ η κλήση `i.next()` θα σημάνει μια εξαίρεση τύπου `IllegalStateException`. Αντίθετα, οι κλήσεις `i.hasPrevious()` και `i.previousIndex()` θα επιστρέψουν `true` και $N-1$ αντίστοιχα, ενώ η κλήση `i.next()` θα επιστρέψει το αντικείμενο στη θέση $N-1$ της ακολουθίας.

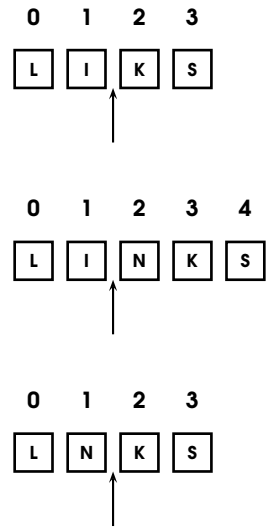
iv). Αν ο δρομέας είναι στη θέση $k, 0 < k < N$, οι κλήσεις `i.hasPrevious()` και `i.hasNext()` θα επιστρέψουν `true`· οι κλήσεις `i.previousIndex()` και `i.nextIndex()` θα επιστρέψουν αντίστοιχα $k-1$ και k, ενώ οι κλήσεις `i.previous()` και `i.next()` θα επιστρέψουν αντίστοιχα τα αντικείμενα στις θέσεις $k-1$ και k της ακολουθίας.

Η μέθοδος `add(Object)`, εισάγει ένα αντικείμενο στην ακολουθία τοποθετώντας το αμέσως πριν από το αντικείμενο το οποίο θα είχε επιστρέψει η `next()` αν είχε κληθεί, και αμέσως μετά τη θέση του στοιχείου που θα είχε επιστρέψει η `previous()` αν είχε κληθεί. Η μέθοδος `remove()` αφαιρεί από την ακολουθία το αντικείμενο που είχε επιστρέψει η τελευταία κλήση της `next()` ή της `previous()`. Κατά αναλογία, η μέθοδος `set(Object)`, αντικαθιστά το αντικείμενο της ακολουθίας που είχε επιστρέψει η τελευταία κλήση της `next()`

ή της previous(Object). Αν η remove() ή η set(Object), κληθεί χωρίς προηγουμένως να έχει κληθεί με επιτυχία μία εκ των μεθόδων next() και previous(), τότε σημαίνεται μια εξαίρεση τύπου IllegalStateException.

Κώδικας 2.5: Η διεπαφή Iterator.

```
1  package aueb.util.api;
2  /**
3   * An iterator is any object able of traversing a sequence. During
4   * traversal, an iterator is positioned either on the left of the
5   * first element, or between two given elements, or on the right of
6   * the last element.
7   */
8  public interface Iterator extends Enumerator {
9      /**
10      * Determines the position of the element on the right.
11      * @return The position number.
12      */
13     int nextIndex();
14     /**
15      * Determines whether there are any elements on the left.
16      * @return True iff there is at least one element on the left.
17      */
18     boolean hasPrevious();
19     /**
20      * Retrieves the object on the left.
21      * @return The object on the left.
22      */
23     Object previous();
24     /**
25      * Determines the position of the element on the left.
26      * @return The position number.
27      */
28     int previousIndex();
29     /**
30      * Inserts an object in the location the iterator is positioned at.
31      * @param object The object to insert.
32      */
33     void add(Object object);
34     /**
35      * Removes the last element accessed by next() or previous().
36      */
37     void remove();
38     /**
39      * Replaces the last object accessed by next() or previous().
```



Σχήμα 2-10: Εισαγωγή και διαγραφή αντικειμένων κατά τη διάσχιση ακολουθίας.
Η μέθοδος add(Object), τοποθετεί το νέο αντικείμενο ανάμεσα στα αντικείμενα που βρίσκονται αριστερά και δεξιά της θέσης του δρομέα. Η μέθοδος remove() αφαιρεί από την ακολουθία το αντικείμενο που είχε επιστρέψει η τελευταία κλήση της next() ή της previous().
Το σχήμα απεικονίζει την κατάσταση ενός δρομέα με τον οποίο διασχίζεται μια ακολουθία από αριστερά προς τα δεξιά. Επάνω, ο δρομέας βρίσκεται στη θέση 2 έχοντας μόλις προσπελάσει το αντικείμενο **I**. Στη μέση, απεικονίζεται το αποτέλεσμα της εισαγωγής ενός νέου αντικειμένου· η ακολουθία μεγαλώνει, ενώ η θέση του δρομέα παραμένει αμετάβλητη. Στο κάτω μέρος του σχήματος φαίνεται η κατάσταση της ακολουθίας και του δρομέα μετά την διαγραφή του τελευταίου αντικειμένου που έχει προσπελαστεί.

```
40      * @param object The replacement object.
41      */
42     void set(Object object);
43 }
```

Ασκήσεις

2-23 Υποθέστε πως η μεταβλητή s αναφέρεται σε ένα αντικείμενο τύπου Sequence το οποίο περιλαμβάνει τα αντικείμενα **A B S T R A C T I O N**. Σχεδιάστε την ακολουθία που προκύπτει από την εκτέλεση του παρακάτω αποσπάσματος κώδικα.

```
Iterator i = s.iterator();
i.next(); i.next(); i.next();
i.replace("T");
i.previous(); i.previous();
i.delete();
```

2-24 Αν η μεταβλητή c είναι αναφορά σε αντικείμενο τύπου Collection που δεν επιτρέπει διπλότυπα και περιέχει τα αντικείμενα **1 2 3 5 6** ενώ η μεταβλητή s αναφέρεται στην ακολουθία **1 1 0 1 1 0 1 1 0**, ποιο είναι το αποτέλεσμα των κλήσεων `c.addAll(s)` και `s.retainAll(c)`;

2-25 Εξηγήστε πως μπορεί να χρησιμοποιηθεί μια ακολουθία για την υλοποίηση των διεπαφών Stack και Queue.

2-26 Δώστε μια αφηρημένη υλοποίηση της διεπαφής Sequence υλοποιώντας όλες τις μεθόδους εκτός από τις `iterator()` και `iterator(int)`.

2-27 Υλοποιείστε τη μέθοδο `addAll(int position, Collection c)` για την κλάση AbstractSequence της προηγούμενης άσκησης. Η μέθοδος πρέπει να εισάγει όλα τα αντικείμενα της συλλογής c στην ακολουθία, ξεκινώντας από της θέση position. Τι πρέπει να κάνει η μέθοδος όταν το πλήθος των αντικειμένων στην ακολουθία είναι μικρότερο από position;

2-28 Υλοποιήστε τη διεπαφή Stack χρησιμοποιώντας μια ακολουθία.

2-29 Υλοποιήστε τη διεπαφή Queue χρησιμοποιώντας μια ακολουθία.

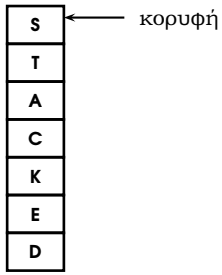
2-30 Υλοποιήστε τη μέθοδο `void shuffle()` η οποία μεταθέτει κατά τυχαίο τρόπο τα αντικείμενα μιας ακολουθίας.

2-31 Υλοποιήστε τη μέθοδο `void reverse()` η οποία αντιστρέφει τη διάταξη των αντικειμένων μιας ακολουθίας.

2-32 Υλοποιήστε τη μέθοδο `void sort()` η οποία ταξινομεί μια ακολουθία χρησιμοποιώντας τον αλγόριθμο insertion sort.

2.8 Στοίβες και ουρές αναμονής

Εκμεταλλευόμενοι τη γενικότητα των διεπαφών `Collection` και `Sequence` μπορούμε να αναπτύξουμε μερικές πολύ χρήσιμες πολιτικές διαχείρισης δεδομένων. Οι στοίβες και οι ουρές αναμονής είναι δύο κατηγορίες δομών δεδομένων που θα μπορούσαν να χαρακτηριστούν ως ειδικές περιπτώσεις ακολουθίας. Είναι ωστόσο τόσο διαδεδομένη η χρήση καθώς και οι εφαρμογές τους ώστε θα τις μελετήσουμε ξεχωριστά.



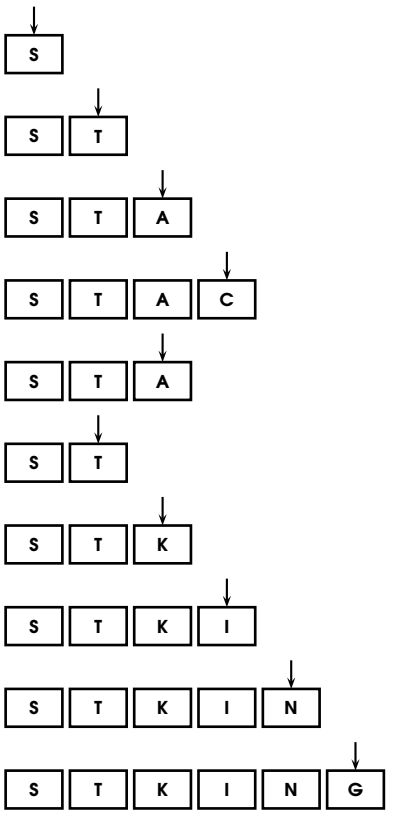
Σχήμα 2-11: Αναπαράσταση στοίβας. Τα αντικείμενα έχουν εισαχθεί με τη σειρά **D E K C A T S**. Στην κορυφή της στοίβας βρίσκεται πάντα το αντικείμενο το οποίο εισήχθηκε πιο πρόσφατα.

Ορισμός 2-12. Μια *στοίβα* είναι μια δομή δεδομένων η οποία έχει ένα νοητό άκρο που ονομάζεται *κορυφή* και από το οποίο γίνονται όλες οι εισαγωγές και διαγραφές αντικειμένων. Η εισαγωγή ενός αντικειμένου σε μια στοίβα ονομάζεται συχνά *ώθηση*, ενώ η διαγραφή αντικειμένου ονομάζεται κατά κανόνα *εξαγωγή*.

Όταν ένα αντικείμενο εισάγεται σε μια στοίβα, τοποθετείται στην κορυφή της όπως παρουσιάζεται στο Σχήμα 2-11. Αντίστοιχα, από μια στοίβα δεν μπορούμε να εξάγουμε οποιοδήποτε στοιχείο επιθυμούμε· το μόνο στοιχείο που μπορεί να διαγραφεί είναι αυτό που βρίσκεται στην κορυφή της στοίβας. Με βάση τους κανόνες αυτούς είναι προφανές πως στην κορυφή μιας στοίβας θα βρίσκεται πάντα το αντικείμενο που εισήχθηκε πιο πρόσφατα. Κάθε φορά που εξάγουμε από μια στοίβα, διαγράφουμε το αντικείμενο που εισήχθηκε τελευταίο. Εξαιτίας αυτής της συμπεριφοράς, οι στοίβες ονομάζονται *δομές Last In, First Out* ή για συντομία δομές LIFO.

Μια στοίβα μπορεί να έχει περιορισμένη ή απεριόριστη μέγιστη *χωρητικότητα* αντικειμένων. Στην πρώτη περίπτωση, είναι παράνομο να προσπαθήσουμε να εισάγουμε ένα αντικείμενο όταν το *μέγεθος* της στοίβας, το πλήθος δηλαδή των στοιχείων που βρίσκονται αποθηκευμένα σε αυτήν, ισούται με τη χωρητικότητά της. Αντίστοιχα είναι παράνομο να προσπαθήσουμε να εξάγουμε ένα αντικείμενο, όταν η στοίβα είναι κενή.

Ο τρόπος με τον οποίο θα αντιμετωπιστούν αυτές τις παράνομες πράξεις, εναπόκειται στην υλοποίηση. Ο πιο συνηθισμένος τρόπος είναι η σήμανση μιας εξαίρεσης. Επειδή ένας προγραμματιστής μπορεί πάντα να γνωρίζει αν μια στοίβα είναι γεμάτη ή όχι, θα θεωρήσουμε ως προγραμματιστικό σφάλμα την εισαγωγή σε μια γεμάτη στοίβα καθώς και την εξαγωγή αντικειμένου από μια



Σχήμα 2-12: Εισαγωγή και εξαγωγή αντικειμένων από μια στοίβα.
Το σχήμα παρουσιάζει μια στοίβα (της οποίας η κορυφή είναι το δεξιότερο άκρο της) κατά την εκτέλεση των λειτουργιών **S T A C - - K I N G** όπου ένα γράμμα υποδεικνύει την εισαγωγή του αντίστοιχου αντικειμένου, ενώ η παύλα, διαγραφή.

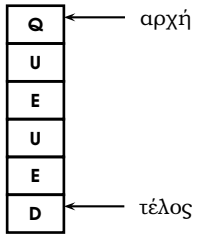
κενή στοίβα. Κατά συνέπεια, οι υλοποιήσεις της διεπαφής Stack που παρουσιάζεται στον Κώδικα 2.6, θα πρέπει σε αυτές τις περιπτώσεις να σημαίνουν μια εξαίρεση τύπου που είναι απόγονος της java.lang.RuntimeException. Μια καλή επιλογή είναι η κλάση java.lang.IllegalStateException.

Κώδικας 2.6: Η διεπαφή Stack.

```
1  package aueb.util.api;
2  /**
3   * Stack interface. An instance of this type is a data structure
4   * with one end called its top. Object insertions and deletions
5   * are performed at the stack's top. Thus, a stack is a last in,
6   * first out data structure.
7   */
8  public interface Stack {
9      /**
10      * Pushes, (i.e., inserts) an object in the stack.
11      * @param object The object to insert.
12      */
13     void push(Object object);
14     /**
15      * References the object at the top of this stack.
16      * @return A reference to the object at the top of this stack.
17      * @throws IllegalStateException if this stack is empty.
18      */
19     Object top();
20     /**
21      * Pops out, (i.e., removes and returns) the object at the top.
22      * @return The object at the top of this stack.
23      * @throws IllegalStateException if this stack is empty.
24      */
25     Object pop();
26     /**
27      * Returns the number of objects currenly in this stack.
28      * @return A non-negative number; the size of this stack.
29      */
30     int size();
31     /**
32      * Tests if this stack is empty.
33      * @return true iff this stack has no objects.
34      */
35     boolean isEmpty();
36     /**
37      * Tests if this stack is full.
38      * @return true iff this stack has reached its capacity.
```

```
39      */
40     boolean isFull();
41 }
```

Αν τροποποιήσουμε ελαφρά τον τρόπο με τον οποίο λειτουργεί και οργανώνεται μια στοίβα, εισάγοντας ένα ακόμη άκρο στην δομή και απαιτώντας η εισαγωγή αντικειμένων να γίνεται από το ένα άκρο ενώ η εξαγωγή από το άλλο, προκύπτει μια πολύ οικεία οργάνωση που ονομάζεται ουρά αναμονής και παρουσιάζεται στο Σχήμα 2-13. Ο κανόνας με βάση τον οποίο λειτουργεί μια ουρά αναμονής, είναι γνωστός ως first in, first out, ή FIFO για συντομία, καθώς το αντικείμενο με το μεγαλύτερο χρόνο παραμονής στην ουρά, βρίσκεται πάντα στην αρχή της, ενώ το αντικείμενο με το μικρότερο χρόνο παραμονής βρίσκεται πάντα στο τέλος της ουράς.

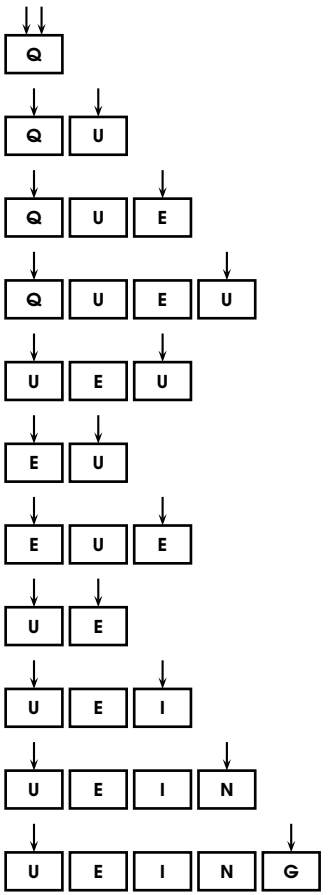


Σχήμα 2-13: Αναπαράσταση ουράς αναμονής.
Τα αντικείμενα έχουν εισαχθεί με τη σειρά **Q U E U E D**. Στην αρχή της ουράς βρίσκεται πάντα το αντικείμενο με το μεγαλύτερο χρόνο αναμονής ενώ στο τέλος το αντικείμενο με τον μικρότερο χρόνο αναμονής.

Ορισμός 2-13. Μια *ουρά αναμονής* είναι μια αφηρημένη δομή δεδομένων με δύο νοητά άκρα που ονομάζονται *αρχή* και *τέλος*. Η εισαγωγή ενός αντικειμένου σε μια ουρά αναμονής γίνεται πάντα στο τέλος της ουράς ενώ η διαγραφή ή αλλιώς *εξαγωγή* ενός αντικειμένου γίνεται πάντα από την αρχή.

Τόσο η οργάνωση όσο και ο τρόπος λειτουργίας της ουράς αναμονής μας είναι πολύ οικεία, καθώς έτσι οργανώνονται και λειτουργούν τα περισσότερα σημεία εξυπηρέτησης όπως ταμεία τραπεζών, εκδοτήρια εισιτηρίων, γραφεία δημοσίων οργανισμών κλπ. Εκτός από την προσομοίωση της λειτουργίας τέτοιων σημείων εξυπηρέτησης, οι ουρές αναμονής έχουν εφαρμογές στη λύση διαφόρων προβλημάτων στην επιστήμη υπολογιστών. Το Σχήμα 2-14 παρουσιάζει ένα παράδειγμα λειτουργίας μιας ουράς αναμονής.

Η διεπαφή μιας ουράς αναμονής παρουσιάζεται στον Κώδικα 2.7. Οι μέθοδοι `put(Object)` και `get()` είναι οι λειτουργίες εισαγωγής και διαγραφής αντίστοιχα. Οι μέθοδοι `head()` και `tail()` είναι οι μέθοδοι πρόσβασης στην αρχή και το τέλος μιας ουράς αναμονής. Τέλος οι μέθοδοι `size()`, `isEmpty()`, και, `isFull()` μας επιτρέπουν να γνωρίζουμε το πλήθος των αντικειμένων που βρίσκονται στην ουρά, καθώς επίσης και αν η ουρά είναι κενή ή γεμάτη. Παρατηρούμε πως όταν η μέθοδος `isFull()` επιστρέφει `true`, είναι παράνομο να επιχειρήσουμε να εισάγουμε αντικείμενο. Παρόμοια, όταν η μέθοδος `isEmpty()` επιστρέφει `true` δεν επιτρέπεται να καλέσουμε τις μεθόδους `get()`, `head()`, και, `tail()`. Η συμπεριφορά



Σχήμα 2-14: Εισαγωγή και εξαγωγή αντικειμένων από μια ουρά αναμονής. Το σχήμα παρουσιάζει μια ουρά αναμονής κατά την εκτέλεση των λειτουργιών **Q U E U - - E - I N G** όπου ένα γράμμα υποδεικνύει εισαγωγή αντικειμένου, ενώ μια παύλα, διαγραφή. Η αρχή και το τέλος της ουράς αναμονής είναι στην αριστερή και δεξιά πλευρά της αντίστοιχα όπως υποδεικνύουν τα βέλη.

μιας υλοποίησης σε αυτές τις περιπτώσεις είναι πανομοιότυπη με αυτή που καθορίζει η διεπαφή Stack.

Κώδικας 2.7: Η διεπαφή Queue.

```
1  package aueb.util.api;
2  /**
3   * Queue interface. An instance of this type is a data structure
4   * with two ends called its head and tail. Object insertions are
5   * performed at the queue's tail, while object deletions are
6   * performed at the queue's head. Thus, a queue is a first in,
7   * first out data structure.
8   */
9  public interface Queue {
10     /**
11      * Inserts an object, at the tail of this queue.
12      * @param object The object to insert.
13      */
14     void put(Object object);
15     /**
16      * References the object at the tail of this queue.
17      * @return A reference to the object at the tail of this queue.
18      * @throws IllegalStateException if this queue is empty.
19      */
20     Object tail();
21     /**
22      * Removes the object at the head of this queue.
23      * @return A reference to the object at the head of this queue.
24      * @throws IllegalStateException if this queue is empty.
25      */
26     Object get();
27     /**
28      * References the object at the head of this queue.
29      * @return A reference to the object at the head of this queue.
30      * @throws IllegalStateException if this queue is empty.
31      */
32     Object head();
33     /**
34      * Returns the number of objects currenly in this queue.
35      * @return A non-negative number; the size of this queue.
36      */
37     int size();
38     /**
39      * Tests if this queue is empty.
40      * @return true iff this queue has no objects.
41      */
42     boolean isEmpty();
```

```
43      /**
44       * Tests if this queue is full.
45       * @return true iff this queue has reached its capacity.
46       */
47      boolean isFull();
48  }
```

Ασκήσεις

2-33 Σχεδιάστε μια στοίβα έπειτα από την εκτέλεση των λειτουργιών **COU-RT-REA-CTION---** όπου ένα γράμμα υποδεικνύει εισαγωγή του αντίστοιχου αντικειμένου, ενώ μια παύλα υποδεικνύει διαγραφή.

2-34 Επαναλάβετε την προηγούμενη άσκηση, χρησιμοποιώντας μια ουρά.

2-35 Γράψτε μια μέθοδο που εκτελεί τις λειτουργίες της Άσκησης 2-33 σε ένα αντικείμενο τύπου `Stack` που είναι παράμετρος της μεθόδου.

2-36 Επαναλάβετε την προηγούμενη άσκηση, χρησιμοποιώντας μια ουρά.

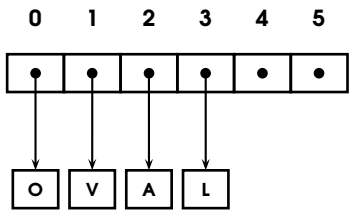
ΚΕΦΑΛΑΙΟ 3

Συστοιχίες

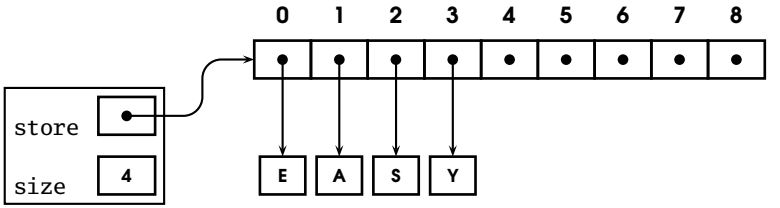
Μια συστοιχία είναι ίσως η απλούστερη μορφή οργάνωσης δεδομένων που γνωρίζουμε η οποία έχει παρόμοια χαρακτηριστικά με μια συλλογή αντικειμένων. Οι συστοιχίες της γλώσσας Java βέβαια, όπως και των περισσοτέρων γλωσσών προγραμματισμού, δεν παρέχουν βασικές λειτουργίες μιας συλλογής όπως η αναζήτηση για παράδειγμα. Μπορούμε ωστόσο να τις χρησιμοποιήσουμε ως τη βάση για την υλοποίηση συλλογών, ακολουθιών και άλλων αφηρημένων δομών δεδομένων. Για μερικές εφαρμογές, οι υλοποιήσεις που βασίζονται σε συστοιχίες έχουν εξαιρετική απόδοση. Σε άλλες περιπτώσεις όπως θα δούμε στα επόμενα κεφάλαια, πολύ αποτελεσματικότερες υλοποιήσεις είναι πιθανές. Ανεξάρτητα από την αποτελεσματικότητά τους πάντως, οι συστοιχίες είναι ένας καθιερωμένος και δημοφιλής μηχανισμός οργάνωσης δεδομένων και, κατά συνέπεια, η μελέτη τους είναι επιβεβλημένη. Επιπλέον, είναι ιδανικό εργαλείο για την εισαγωγή βασικών εννοιών και την κατανόηση των αντίστοιχων υλοποιήσεων. Στο κεφάλαιο αυτό περιγράφονται οι βασικές αρχές για τη χρήση συστοιχιών καθώς επίσης υλοποιήσεις των διεπαφών `Collection`, `Sequence`, `Stack` και `Queue`.

3.1 Συλλογές αντικειμένων

Μια συστοιχία είναι μια παράταξη μεταβλητών του ίδιου τύπου σε συνεχόμενες θέσεις της κεντρικής μνήμης. Κάθε μία από τις μεταβλητές αυτές έχει μια θέση στη συστοιχία μέσω της οποίας μπορούμε



Σχήμα 3-1: Αποθήκευση αντικειμένων σε συστοιχία.



Σχήμα 3-2: Εισαγωγή αντικειμένου σε συλλογή που υλοποιείται με συστοιχία. Το σχήμα απεικονίζει μια συλλογή, έπειτα από την διαδοχική εισαγωγή των αντικειμένων **E A S Y**.

να αναφερθούμε στην αντίστοιχη μεταβλητή. Ας δούμε πως μπορούμε να ορίσουμε τις βασικές λειτουργίες της διεπαφής `Collection` χρησιμοποιώντας μια συστοιχία αναφορών τύπου `Object`.

Κάθε θέση της συστοιχίας αποθηκεύει μια αναφορά σε ένα αντικείμενο της συλλογής αντικειμένων όπως φαίνεται στο Σχήμα 3-1. Όταν κατασκευάζεται μια συστοιχία τύπου `Object[]` όλα τα στοιχεία της έχουν την τιμή `null`. Θεωρούμε τότε πως η αντίστοιχη συλλογή αντικειμένων είναι κενή.

Εισαγωγή. Λέμε πως ένα αντικείμενο ο εισάγεται στη συλλογή, όταν αποθηκεύσουμε μια αναφορά στο ο σε κάποια θέση της συστοιχίας. Η θέση της συστοιχίας στην οποία θα αποθηκευτεί η αναφορά δεν μας ενδιαφέρει για την υλοποίηση της διεπαφής `Collection`. Πρόκειται για μια λεπτομέρεια υλοποίησης και κατά συνέπεια είμαστε ελεύθεροι να την επιλέξουμε ώστε να διευκολύνουμε την υλοποίηση ή να επιτύχουμε την μεγαλύτερη δυνατή αποτελεσματικότητα. Θα επιλέξουμε ως θέση εισαγωγής τη αριστερότερη θέση της συστοιχίας η οποία έχει τιμή `null`. Όταν η συλλογή είναι κενή, η θέση αυτή είναι η θέση 0. Γενικά αν η συλλογή έχει k αντικείμενα, η θέση εισαγωγής είναι η θέση k.

Καθώς το πλήθος των αποθηκευμένων αντικειμένων ισούται με το πλήθος των θέσεων της συστοιχίας που έχουν τιμή διαφορετική από `null`, θα εξυπηρετούσε την αποτελεσματικότητα της υλοποίησης να διατηρούμε μια μεταβλητή `size` με αρχική τιμή 0, η οποία αυξάνεται κατά ένα με κάθε εισαγωγή αντικειμένου και μειώνεται κάθε φορά που διαγράφεται ένα αντικείμενο. Μια συλλογή που υλοποιείται με συστοιχία θα έχει κατά συνέπεια δύο πεδία.

i). Το πεδίο `store` είναι μια αναφορά στη συστοιχία τύπου `Object[]` στην οποία αποθηκεύονται οι αναφορές στα αντικείμενα της συλλογής.

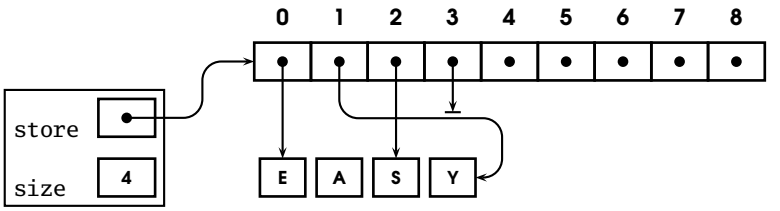
ii). Το πεδίο `size` τύπου `int` μετρά το πλήθος των κατειλημμένων θέσεων της συστοιχίας `store` και ταυτόχρονα καθορίζει τη θέση της συστοιχίας στην οποία θα γίνει η εισαγωγή του επόμενου αντικειμένου.

Η οργάνωση αυτή παρουσιάζεται στο Σχήμα 3-2.

Αναζήτηση και απαρίθμηση. Η απάντηση στην ερώτηση αν ένα αντικείμενο είναι μέλος της συλλογής είναι απλή. Δεν έχουμε παρά να ξεκινήσουμε από την θέση 0 της συστοιχίας και να συγκρίνουμε χρησιμοποιώντας τη μέθοδο `equals` το αντικείμενο στην τρέχουσα θέση της συστοιχίας με το ζητούμενο αντικείμενο. Αν εξαντλήσουμε όλες τις θέσεις της συστοιχίας χωρίς επιτυχία, τότε η απάντηση είναι αρνητική. Αν αντίθετα διαπιστώσουμε ισότητα αντικειμένων σε κάποια θέση, τότε η απάντηση είναι θετική.

Με τον ίδιο ακριβώς τρόπο μπορούμε να απαριθμήσουμε τα αντικείμενα που περιέχει η συλλογή. Ο απαριθμητής των στοιχείων της συλλογής χρειάζεται να διατηρεί μια μεταβλητή που προσδιορίζει τη θέση της συστοιχίας `store` στην οποία βρίσκεται το αντικείμενο που πρέπει να επιστρέψει η επόμενη κλήση της μεθόδου `next`. Η μεταβλητή αυτή πρέπει να αρχικοποιείται με την τιμή 0 κατά την κατασκευή του απαριθμητή και να αυξάνεται σε κάθε κλήση της `next`. Τέλος, καθώς ο απαριθμητής χρειάζεται πρόσβαση στα πεδία `store` και `size` της συλλογής την οποία διασχίζει, θα πρέπει να υλοποιείται από μια εσωτερική κλάση (βλέπε Κώδικα 3.1 στη σελίδα 87).

Διαγραφή. Η διαγραφή απαιτεί δύο βήματα. Το πρώτο είναι ο εντοπισμός της θέσης της συστοιχίας στην οποία βρίσκεται το αντικείμενο που πρέπει να διαγραφεί. Αφού έχουμε εντοπίσει τη θέση αυτή, έστω i, μπορούμε να διαγράψουμε το αντίστοιχο αντικείμενο από τη συλλογή αντικαθιστώντας την αναφορά στη θέση i με την αναφορά στη θέση $N-1$, όπου N είναι το πλήθος των αντικειμένων στη συλλογή. Στη συνέχεια πρέπει να θέσουμε την αναφορά στη θέση $N-1$ ίση με `null` καθώς η θέση είναι πλέον διαθέσιμη για την αποθήκευση κάποιου άλλου αντικειμένου. Η διαδικασία της διαγραφής παρουσιάζεται με ένα παράδειγμα στο Σχήμα 3-3.



Σχήμα 3-3: Διαγραφή αντικειμένου από συλλογή που υλοποιείται με συστοιχία. Το σχήμα παρουσιάζει τη συλλογή του Σχήματος 3-2 έπειτα από τη διαγραφή του αντικειμένου **A**. Η διαγραφή επιτυγχάνεται με την αντικατάσταση της αναφοράς του διαγραφόμενου αντικειμένου με την αναφορά `store[size-1]`.

Επέκταση της συστοιχίας. Το βασικό μειονέκτημα που παρουσιάζει η χρήση συστοιχίας για την υλοποίηση συλλογής, είναι πως το πλήθος των αντικειμένων που μπορούν να αποθηκευτούν περιορίζεται από το μήκος της συστοιχίας. Στις περισσότερες εφαρμογές ωστόσο, δεν μπορούμε να γνωρίζουμε προκαταβολικά το μέγιστο πλήθος αντικειμένων που θα χρειαστεί να αποθηκεύσει μια συλλογή. Ακόμη κι αν η πληροφορία αυτή είναι διαθέσιμη, το μέσο πλήθος αντικειμένων της συλλογής μπορεί να είναι σημαντικά μικρότερο από το μέγιστο, με αποτέλεσμα την άσκοπη δέσμευση μνήμης. Το ιδανικό θα ήταν να μπορούσαμε να αυξήσουμε το μήκος της συστοιχίας μόλις διαπιστώναμε πως το μέγεθος της συλλογής ισούται με το μήκος της συστοιχίας.

Σε ορισμένες γλώσσες προγραμματισμού, όπως η C για παράδειγμα, κάτι τέτοιο είναι εφικτό. Στη Java, αντίθετα, δεν υπάρχει αυτή η δυνατότητα και έτσι είμαστε αναγκασμένοι να την προσομοιώσουμε. Συγκεκριμένα όταν κατά την εισαγωγή αντικειμένου διαπιστώσουμε πως η συστοιχία `store` είναι "γεμάτη", κατασκευάζουμε μια νέα συστοιχία `tmp` με μήκος διπλάσιο από αυτό της `store`, αντιγράφουμε τις αναφορές της `store` στην `tmp`, και τέλος θέτουμε `store = tmp`.

Η τεχνική αυτή ονομάζεται *διπλασιασμός* (doubling) και χρησιμοποιείται σε πολλές υλοποιήσεις συλλογών δυναμικού μεγέθους που βασίζονται σε συστοιχίες, όπως για παράδειγμα η κλάση `java.lang.StringBuffer` της Java. Την τεχνική αυτή θα χρησιμοποιήσουμε και στο Κεφάλαιο 8 στην υλοποίηση πινάκων κατακερματισμού. Ο λόγος που έχουμε επιλέξει να διπλασιάζουμε το μέγεθος αντί για παράδειγμα να το αυξάνουμε κατά ένα σταθερό μέγεθος κάθε φορά, είναι πως έτσι εξασφαλίζεται (βλέπε Άσκηση 3-10) ότι ο χρόνος για την εισαγωγή N αντικειμένων στη συλλογή είναι

$O(N)$. Επιμερίζοντας το χρόνο αυτό σε κάθε εισαγωγή προκύπτει ότι ο χρόνος για την εισαγωγή ενός αντικειμένου είναι σταθερός.

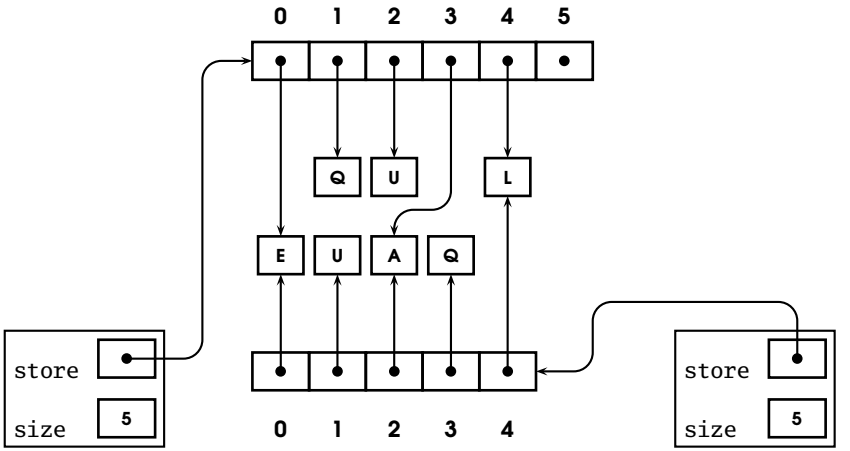
Ισότητα και αντιγραφή. Όπως έχουμε δει (βλέπε Κώδικας 2.3 στη σελίδα 2.3), δύο συλλογές είναι ίσες όταν έχουν το ίδιο πλήθος αντικειμένων και επιπλέον, κάθε αντικείμενο που ανήκει στην μία ανήκει και στην άλλη. Το Σχήμα 3-4 παρουσιάζει ένα παράδειγμα δύο αντικειμένων τύπου `ArrayCollection` που είναι ίσα.

Η δημιουργία ενός κλώνου μιας συλλογής A, είναι η κατασκευή μιας συλλογής που είναι ίση με την A. Με βάση την οργάνωση που έχουμε περιγράψει σε αυτή την ενότητα, για την δημιουργία ενός αντιγράφου αρκεί να κατασκευάσουμε μια συστοιχία μήκους όσο και το πλήθος των αντικειμένων στην A και να αποθηκεύσουμε σε κάθε θέση της μια αναφορά σε ένα αντικείμενο της A. Το αποτέλεσμα απεικονίζεται στο Σχήμα 3-5.

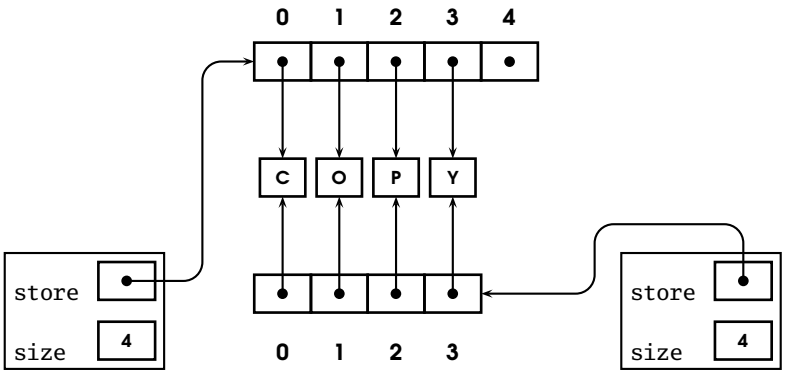
Αξίζει να τονίσουμε εδώ πως η σημασιολογία τόσο της ισότητας όσο και της αντιγραφής είναι ανεξάρτητη του μηχανισμού που χρησιμοποιείται για την υλοποίηση μιας συλλογής. Με άλλα λόγια, οι δύο αυτές πράξεις είναι ανεξάρτητες από λεπτομέρειες υλοποίησης. Δεν θα πρέπει επομένως να μας παραπλανούν τα Σχήματα 3-4 και 3-5· σε καθένα από τα οποία είναι προφανές πως οι συλλογές που απεικονίζονται χρησιμοποιούν τον ίδιο μηχανισμό οργάνωσης. Αυτό συμβαίνει διότι δεν έχουμε ακόμη περιγράψει καμιά άλλη υλοποίηση της διεπαφής `Collection`.

3.1.1 Υλοποίηση της κλάσης `ArrayCollection`

Η υλοποίηση της οργάνωσης που παρουσιάσαμε δίνεται στον Κώδικας 3.1. Η κλάση `ArrayCollection` παρέχει τρεις κατασκευαστές. Ο εξ' ορισμού κατασκευαστής δημιουργεί μια συλλογή αρχικής χωρητικότητας `ArrayCollection.DEFAULT_CAPACITY` αντικειμένων. Ο δεύτερος κατασκευαστής επιτρέπει τη δημιουργία μιας συλλογής με συγκεκριμένη χωρητικότητα, η οποία ωστόσο πρέπει να είναι τουλάχιστον 1. Η κλάση παρέχει επίσης και ένα κατασκευαστή αντιγραφής ο οποίος χρησιμοποιεί απαρίθμηση. Αξίζει να προσέξουμε πως ο κατασκευαστής αντιγραφής δέχεται ένα όρισμα τύπου `Collection` αντί για ένα όρισμα `ArrayCollection` όπως θα περίμενε κανείς με βάση τον αυστηρό ορισμό που δώσαμε στο



Σχήμα 3-4: Ισότητα αντικειμένων τύπου `ArrayCollection`.



Σχήμα 3-5: Αντιγραφή αντικειμένων τύπου `ArrayCollection`.
Το αντικείμενο στη δεξιά πλευρά του σχήματος είναι αντίγραφο του αντικειμένου της αριστερής πλευράς. Παρατηρούμε πως τα αντικείμενα της συλλογής δεν αντιγράφονται. Το αντίγραφο διατηρεί αντίγραφα των αναφορών που διατηρεί και το πρωτότυπο.

Κεφάλαιο 2. Αυτή η επιλογή ωστόσο, παρέχει μεγαλύτερη ευελιξία, καθώς μας επιτρέπει να κατασκευάσουμε ένα αντικείμενο τύπου ArrayCollection απαριθμώντας τα στοιχεία ενός αντικειμένου τύπου Collection ανεξάρτητα από το μηχανισμό που χρησιμοποιεί για την υλοποίηση της διεπαφής. Πρέπει να σημειώσουμε ακόμη, πως η κλάση ArrayCollection ακυρώνει τη μέθοδο clear της διεπαφής AbstractCollection προκειμένου να βελτιώσει την αποτελεσματικότητά της. Το ίδιο θα έπρεπε βεβαίως να γίνει και για άλλες μεθόδους όπως η addAll για παράδειγμα, αλλά αυτό αφήνεται ως άσκηση.

Κώδικας 3.1: Η κλάση ArrayCollection.

```
1  package aueb.util.imp;
2
3  import aueb.util.api.AbstractCollection;
4  import aueb.util.api.Collection;
5  import aueb.util.api.Enumerator;
6  /**
7   * Array-based implementation of {@link aueb.util.api.Collection}.
8   */
9  public class ArrayCollection extends AbstractCollection {
10     /**
11      * Enumerates the elements of the enclosing collection.
12      */
13     private class ArrayEnumerator implements Enumerator {
14        /**
15         * The position of the element that a call to next() will return.
16         */
17        int position;
18        /**
19         * Creates a new instance and sets position to 0.
20         */
21        public ArrayEnumerator() {
22           position = 0;
23        }
24        /**
25         * If position is lower than size, return true.
26         */
27        public boolean hasNext() {
28           return position < size;
29        }
30        /**
31         * Return the next element, and update position.
32         */
```

```
33        public Object next() {
34           if (position == size) throw new IllegalStateException();
35           return store[position++];
36        }
37     }
38     /**
39      * Default initial capacity of the array.
40      */
41     private static final int INITIAL_CAPACITY = 8;
42     /**
43      * The array that stores references to collection elements.
44      */
45     protected Object[] store;
46     /**
47      * The number of elements currently in this collection.
48      */
49     protected int size;
50     /**
51      * Default constructor. Uses default capacity and creates
52      * an empty collection.
53      */
54     public ArrayCollection() {
55        this(INITIAL_CAPACITY);
56     }
57     /**
58      * Creates an empty collection that has a specified initial capacity.
59      * @param capacity The initial capacity of this collection.
60      */
61     public ArrayCollection(int capacity) {
62        super();
63        if (capacity < 1) {
64           throw new IllegalArgumentException();
65        }
66        this.store = new Object[capacity];
67        this.size = 0;
68     }
69     /**
70      * Copy constructor.
71      * @param collection The collection to copy.
72      */
73     public ArrayCollection(Collection collection) {
74        this(collection.size());
75        Enumerator e = collection.enumerator();
76        while (e.hasNext()) store[size++] = e.next();
77     }
78     /**
```

```
79      * Inserts an object in this collection (allows duplicates).
80      * Expands the capacity if necessary.
81      * @param The object to insert.
82      */
83     public boolean add(Object object) {
84        if (object == null) throw new IllegalArgumentException();
85        if (size == store.length) expand();
86        store[size++] = object;
87        return false;
88     }
89     /**
90      * Removes an object from this collection. The first element
91      * that equals object is always removed.
92      * @param The object to remove.
93      */
94     public boolean remove(Object object) {
95        int index = indexOf(object);
96        if (index == -1) {
97           return false;
98        }
99        store[index] = store[size-1];
100       store[--size] = null;
101       return true;
102    }
103    public int size() {
104       return size;
105    }
106    public boolean contains(Object object) {
107       return indexOf(object) != -1;
108    }
109    /**
110     * Removes all element references.
111     */
112    public void clear() {
113       while (size > 0) store[size--] = null;
114    }
115    /**
116     * Locates an object in this collection.
117     * @param object The object to locate.
118     * @return The object positon or -1 if object was not found.
119     */
120    public int indexOf(Object object) {
121       for (int i = 0; i < size; ++i) {
122          if (store[i].equals(object)) return i;
123       }
124       return -1;
```

```
125     }
126     public Enumerator enumerator() {
127        return new ArrayEnumerator();
128     }
129     /**
130      * Expands store by doubling its length.
131      */
132     protected void expand() {
133        int capacity = 2 * store.length;
134        Object[] temp = new Object[capacity];
135        System.arraycopy(store, 0, temp, 0, size);
136        store = temp;
137     }
138  }
```

3.1.2 Αποτελεσματικότητα

Ο Πίνακας 3-1 δίνει το χρόνο εκτέλεσης των βασικών μεθόδων της κλάσης `ArrayCollection` όταν η συλλογή περιέχει N στοιχεία.

Η μέθοδος `add` στη χειρότερη περίπτωση, όταν δηλαδή επιχειρήσουμε να εισάγουμε ένα νέο αντικείμενο σε συλλογή με k στοιχεία και χωρητικότητα k, απαιτεί τη δημιουργία μιας συστοιχίας $2k$ στοιχείων και την αντιγραφή των αναφορών της τρέχουσας συστοιχίας. Αν και δεν είναι ιδιαίτερα εύκολο να υπολογίσουμε τον αναμενόμενο χρόνο για την προσθήκη ενός αντικειμένου στη συλλογή, μπορούμε να υπολογίσουμε τον επιμερισμένο χρόνο (amortized time) για την εισαγωγή ενός αντικειμένου στη συλλογή. Η ιδέα είναι να υπολογίσουμε το χρόνο που απαιτεί η εισαγωγή N αντικειμένων σε μια κενή συλλογή ξεκινώντας από αρχική χωρητικότητα 1, και στη συνέχεια να επιμερίσουμε αυτό το χρόνο σε κάθε εισαγωγή ξεχωριστά. Μπορούμε να αποδείξουμε (βλέπε Άσκηση 3-10) πως ο χρόνος που απαιτείται είναι $O(N)$, και κατά συνέπεια ο επιμερισμένος χρόνος εισαγωγής ενός αντικειμένου είναι $O(N)/N = O(1)$.

Η μέθοδος `contains` χρησιμοποιεί ακολουθιακή αναζήτηση σε μη ταξινομημένη συστοιχία. Με βάση τα όσα έχουμε αναφέρει στο Κεφάλαιο 1 ο αναμενόμενος χρόνος εκτέλεσης είναι $O(N)$. Η μέθοδος `remove` εκτελεί μια αναζήτηση για τον εντοπισμό του αντικειμένου και δύο αναθέσεις για την διαγραφή του. Επομένως ο αναμενόμενος χρόνος εκτέλεσής της είναι $O(N)$. Οι μέθοδος `clear` απαιτεί χρόνο $O(N)$, ενώ τέλος, οι μέθοδοι `size` και `isEmpty` απαι-

τούν σταθερό χρόνο.

Μέθοδος	$W(N)$	$A(N)$
`add`	$O(N)$	$O(1)$
`remove`	$O(N)$	$O(N)$
`contains`	$O(N)$	$O(N)$
`size`	$O(1)$	$O(1)$
`isEmpty`	$O(1)$	$O(1)$
`clear`	$O(N)$	$O(N)$

Πίνακας 3-1: Αποτελεσματικότητα των μεθόδων της κλάσης `ArrayCollection`.

Ασκήσεις

3-1 Μπορούμε να χρησιμοποιήσουμε την κλάση `ArrayCollection` για την παράσταση ενός συνόλου αντικειμένων;

3-2 Σχεδιάστε την κατάσταση ενός αντικειμένου τύπου `ArrayCollection` το οποίο αρχικά δεν περιέχει κανένα αντικείμενο, έπειτα από την εκτέλεση των πράξεων **+1 -9 +5 + 6 +1 -7 +8 -1 -5 +3 +2 +4 -8 +11 -3** όπου το σύμβολο **+** υποδεικνύει εισαγωγή ενώ το σύμβολο **-** διαγραφή.

3-3 Τι επιστρέφει η παρακάτω μέθοδος;

```
boolean method(ArrayCollection c) {
   String alpha = "A";
   c.add(alpha);
   ArrayCollection b = c.clone();
   b.remove(alpha);
   return c.contains(alpha);
}
```

3-4 Τροποποιήστε την κλάση `ArrayCollection` έτσι ώστε να μπορεί να έχει αρχική χωρητικότητα ίση με 0.

3-5 Υπολογίστε τον αναμενόμενο χρόνο εκτέλεσης της κλήσης `a.equals(b)` όταν `a` και `b` είναι αναφορές σε αντικείμενα τύπου `ArrayCollection`.

3-6 Ακυρώστε στην κλάση `ArrayCollection` τη μέθοδο `addAll` που κληρονομεί από την κλάση `AbstractCollection` παρέχοντας μια ελαφρώς ταχύτερη υλοποίηση.

3-7 Υλοποιήστε τη μέθοδο `clone` για την κλάση `ArrayCollection` και υπολογίστε το χρόνο εκτέλεσής της.

3-8 Ακυρώστε στην κλάση `ArrayCollection` τη μέθοδο `retainAll` της κλάσης `AbstractCollection`.

3-9 Ένα σύνολο φυσικών $S \subseteq [0, N)$ μπορεί να οργανωθεί ως μια ακολουθία N δυαδικών ψηφίων $b_0, \ldots, b_{N-1}$. Αν το δυαδικό ψηφίο b_i είναι 1 τότε θεωρούμε πως $i \in S$, διαφορετικά $i \notin S$. Για την υλοποίηση της ακολουθίας δυαδικών ψηφίων μπορούμε να χρησιμοποιήσουμε ένα πίνακα $\lceil N/8 \rceil$ ψηφιοσυλλαβών.

Αυτή η οργάνωση απαιτεί ελάχιστο χώρο, ενώ επιτρέπει ταχύτατη υλοποίηση των μεθόδων εισαγωγής, διαγραφής και αναζήτησης, καθώς και της ένωσης, τομής και διαφοράς συνόλων. Υλοποιείστε

τις μεθόδους της αφηρημένης κλάσης IntegerSet χρησιμοποιώντας τον παραπάνω μηχανισμό.

```
public abstract class IntegerSet {
   public abstract boolean add(int value);
   public abstract boolean remove(int value);
   public abstract boolean contains(int value);
   public abstract int size();
   public abstract boolean isEmpty();
   public abstract void clear();
   public abstract IntegerSet addAll(IntegerSet set);
   public abstract IntegerSet retainAll(IntegerSet set);
   public abstract IntegerSet removeAll(IntegerSet set);
}
```

3-10 Αποδείξτε πως η μέθοδος add(Object) της κλάσης ArrayCollection, απαιτεί χρόνο $O(N)$ για την προσθήκη N αντικειμένων σε κενή συστοιχία με αρχική χωρητικότητα 1.

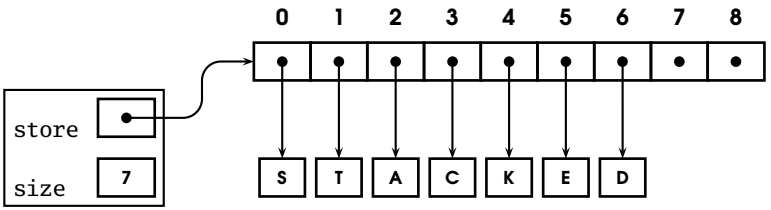
3-11 Τροποποιήστε την κλάση ArrayCollection έτσι ώστε να μην επιτρέπει διπλότυπα. Υπολογίστε τον χρόνο εκτέλεσης της μεθόδου add.

3-12 Τροποποιήστε την κλάση ArrayCollection έτσι ώστε η μέθοδος add να διατηρεί την συστοιχία ταξινομημένη. Φροντίστε να τροποποιήσετε κατάλληλα τον κατασκευαστή αντιγραφής καθώς επίσης και τη μέθοδο contains έτσι ώστε να απαιτεί το πολύ $\lg N + 1$ συγκρίσεις για τον εντοπισμό ενός αντικειμένου.

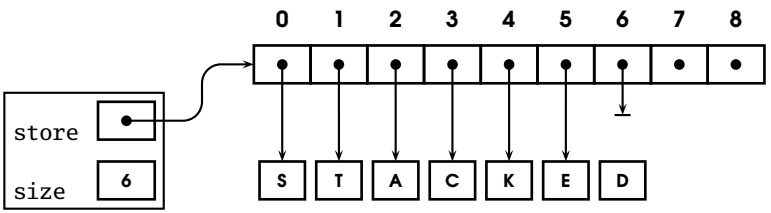
3.2 Στοίβες

Χρησιμοποιώντας την ίδια ακριβώς οργάνωση με αυτή της κλάσης ArrayCollection, αλλά προσαρμόζοντας τους κανόνες εισαγωγής και διαγραφής στοιχείων, μπορούμε να υλοποιήσουμε μια στοίβα. Χρησιμοποιούμε μια συστοιχία τύπου Object[] για την αποθήκευση αναφορών στα αντικείμενα της στοίβας και ένα πεδίο size το οποίο προσδιορίζει ταυτόχρονα το μέγεθος της στοίβας αλλά και τη θέση της συστοιχίας στην οποία βρίσκεται η κορυφή της στοίβας. Η εισαγωγή και διαγραφή γίνεται από τη θέση που καθορίζει το πεδίο αυτό. Η οργάνωση παρουσιάζεται στα Σχήματα 3-6 και 3-7.

Η υλοποίηση που παρουσιάζουμε στον Κώδικα 3.2 έχει μια καθορισμένη μέγιστη χωρητικότητα και κατά συνέπεια δεν παρέχει τη δυνατότητα αυτόματης επέκτασης όπως η κλάση ArrayCollection.



Σχήμα 3-6: Υλοποίηση στοίβας με συστοιχία.
Το σχήμα απεικονίζει μια στοίβα που υλοποιείται με χρήση συστοιχίας. Όταν ένα αντικείμενο ωθείται στη στοίβα, αποθηκεύεται μια αναφορά στο αντικείμενο στη θέση `store[size]` και το πεδίο `size` αυξάνεται κατά 1.



Σχήμα 3-7: Εξαγωγή στοιχείου από στοίβα που υλοποιείται με συστοιχία.
Το σχήμα απεικονίζει τη στοίβα του Σχήματος 3-6 έπειτα από την εξαγωγή του στοιχείου στην κορυφή. Όταν εξάγεται το αντικείμενο στην κορυφή της στοίβας, η αναφορά στη θέση `store[size-1]` γίνεται `null` και το πεδίο `size` μειώνεται κατά 1.

Η μέγιστη χωρητικότητα μιας στοίβας είναι παράμετρος του κατασκευαστή της κλάσης `ArrayStack` και η τιμή του αντίστοιχου ορίσματος πρέπει να είναι μεταξύ 1 και `Integer.MAX_VALUE`. Σε αντίθετη περίπτωση, ο κατασκευαστής σημαίνει μια εξαίρεση τύπου `java.lang.IllegalArgumentException`. Η υλοποίηση των μεθόδων `push` και `pop` είναι εξαιρετικά απλή. Η πρώτη ωθεί ένα αντικείμενο στη στοίβα εφόσον υπάρχει χώρος, διαφορετικά σημαίνει μια εξαίρεση τύπου `IllegalStateException`. Αντίστοιχα η μέθοδος `pop` επιστρέφει το αντικείμενο στην κορυφή της στοίβας εφόσον η στοίβα δεν είναι κενή, διαφορετικά σημαίνει μια εξαίρεση τύπου `IllegalStateException`.

Όλες οι μέθοδοι της κλάσης `ArrayStack`, που συνοψίζονται στον Πίνακα 3-2 εκτελούνται σε σταθερό χρόνο εκτός από τη μέθοδο `clone` η οποία απαιτεί χρόνο ανάλογο του πλήθους των αντικειμένων που βρίσκονται στη στοίβα.

`push`	$O(1)$
`pop`	$O(1)$
`top`	$O(1)$
`size`	$O(1)$
`isEmpty`	$O(1)$
`isFull`	$O(1)$
`clone`	$O(N)$

Πίνακας 3-2: Αποτελεσματικότητα των μεθόδων της κλάσης `ArrayStack`.

Κώδικας 3.2: Υλοποίηση στοίβας με χρήση συστοιχίας

```
1 package aueb.util.imp;
2
3 import aueb.util.api.Stack;
4
5 /**
6  * Array-based stack implementation.
```

```
7  */
8  public final class ArrayStack implements Stack, Cloneable {
9     /**
10     * Default stack capacity.
11     */
12    public static final int INITIAL_CAPACITY = 8;
13    /**
14     * The array that stores references to stack elements.
15     */
16    private Object[] store;
17    /**
18     * The index of the top element and the size of this stack.
19     */
20    private int size;
21    /**
22     * Default constructor. Uses default capacity and creates
23     * an empty stack.
24     */
25    public ArrayStack() {
26       this(INITIAL_CAPACITY);
27    }
28    /**
29     * Creates an empty stack that has a specified capacity.
30     * @param capacity The capacity of this stack.
31     */
32    public ArrayStack(int capacity) {
33       super();
34       if (capacity < 1) {
35          throw new IllegalArgumentException();
36       }
37       store = new Object[capacity];
38       size = -1;
39    }
40    public void push(Object element) {
41       if (element == null) throw new IllegalArgumentException();
42       if (size == store.length -1) throw new IllegalStateException();
43       store[++size] = element;
44    }
45    public Object top() {
46       if (size == -1) throw new IllegalStateException();
47       return store[size];
48    }
49    public Object pop() {
50       if (size == -1) throw new IllegalStateException();
51       Object element = store[size];
52       store[size--] = null;
```

```
53        return element;
54     }
55     public int size() {
56        return size + 1;
57     }
58     public boolean isEmpty() {
59        return size == -1;
60     }
61     public boolean isFull() {
62        return size == store.length - 1;
63     }
64     protected Object clone() {
65        ArrayStack clone;
66        try {
67           clone = (ArrayStack) super.clone();
68        }
69        catch (CloneNotSupportedException e) {
70           throw new InternalError();
71        }
72        clone.store = (Object[]) store.clone();
73        return clone;
74     }
75
76  }
```

Οι στοίβες είναι χαρακτηριστικό παράδειγμα απλής και αποτελεσματικής οργάνωσης με πολλές εφαρμογές στο σχεδιασμό αλγορίθμων. Ας μην ξεχνάμε πως η ίδια η εκτέλεση ενός προγράμματος βασίζεται σε μια στοίβα πλαισίων ενεργοποίησης.

Εκτέλεση μεθόδων. Όταν ένα πρόγραμμα ξεκινά να εκτελείται, το περιβάλλον εκτέλεσης, η εικονική μηχανή στην περίπτωση της Java, φροντίζει για την δημιουργία μιας στοίβας στην οποία εισάγονται αντικείμενα που ονομάζονται *πλαίσια ενεργοποίησης*. Κάθε φορά που μια μέθοδος εκτελεί μια κλήση, δημιουργείται ένα πλαίσιο ενεργοποίησης το οποίο περιλαμβάνει τα ορίσματα της καλούμενης μεθόδου, τις τοπικές της μεταβλητές, την επιστρεφόμενη τιμή της, καθώς επίσης και τη διεύθυνση επιστροφής, την εντολή δηλαδή που θα εκτελεστεί αμέσως μετά την ολοκλήρωση της εκτέλεσης της καλούμενης μεθόδου. Όταν μια μέθοδος ολοκληρώνει την εκτέλεσή της, το πλαίσιο ενεργοποίησής της εξάγεται από τη στοίβα, και η εκτέλεση του προγράμματος συνεχίζει από την εντολή που καθορίζει η διεύθυνση επιστροφής. Φυσικά, το πρώτο πλαίσιο ενεργοποίησης

```
(    (
1    (
+    (
{    ( {
(    ( { (
m    ( { (
a    ( { (
x    ( { (
{    ( { ( {
a    ( { ( {
,    ( { ( {
{    ( { ( { {
b    ( { ( { {
}    ( { ( {
}    ( { (
)    ( {
}    (
/    (
2    (
)
```

Σχήμα 3-8: Έλεγχος ορθότητας παρενθετικών παραστάσεων.
Το σχήμα απεικονίζει την λειτουργία του αλγορίθμου κατά τον έλεγχο της παρενθετικής παράστασης `(1+{(max{a,{b}})}/2)`. Στην αριστερή πλευρά είναι ο τρέχον χαρακτήρας της παράστασης, ενώ στη δεξιά τα περιεχόμενα της στοίβας.

που εισάγεται στη στοίβα είναι αυτό της μεθόδου `main`.

Η στοίβα εκτέλεσης, όπως και η υλοποίηση `ArrayStack`, έχει προκαθορισμένη μέγιστη χωρητικότητα η οποία δεν μπορεί να αλλάξει έπειτα από τη δημιουργία της. Στην περίπτωση που επιχειρηθεί μια κλήση, ενώ η στοίβα εκτέλεσης έχει εξαντλήσει τη διαθέσιμη χωρητικότητα πλαισίων, τότε το περιβάλλον εκτέλεσης σημαίνει μια εξαίρεση τύπου `StackOverflowError` και το πρόγραμμα τερματίζει. Κατά κανόνα κάτι τέτοιο μπορεί να συμβεί μόνο σε προγράμματα που χρησιμοποιούν αναδρομή. Η ποσότητα μνήμης που είναι διαθέσιμη για τη στοίβα εκτέλεσης, μπορεί να καθοριστεί από την επιλογή `-Xss` του προγράμματος `java`.

Έλεγχος παρενθετικών παραστάσεων. Μια *παρενθετική παράσταση* είναι μια παράσταση που περιέχει παρενθετικούς χαρακτήρες όπως $($, και $\{$. Η παράσταση

$$(a+1)\left(1+(\max\{a,[b]\})^2\right) \tag{3.1}$$

είναι μια παρενθετική παράσταση που στο σύστημα LaTeX, το οποίο χρησιμοποιήθηκε για τη στοιχειοθεσία του κειμένου αυτού, παράγεται από την παρακάτω, επίσης παρενθετική, παράσταση.

```
\begin{equation}
(a+1) \left(1+{\left(\max\left\{a,[b]\right\}\right)}^2\right)
\end{equation}
```

Μια παρενθετική παράσταση είναι ορθή όσον αφορά το φώλιασμα ή ταίριασμα των παρενθετικών χαρακτήρων, όταν για κάθε αριστερό παρενθετικό χαρακτήρα, υπάρχει ένας αντίστοιχος δεξιός παρενθετικός χαρακτήρας. Με βάση τον ορισμό αυτό λοιπόν, η παράσταση $[0,n)$, δεν είναι ορθή. Μπορούμε να ελέγξουμε την ορθότητα μιας παρενθετικής παράστασης με τη χρήση στοίβας εξετάζοντας διαδοχικά τους χαρακτήρες της παράστασης ξεκινώντας από τα αριστερά, και προχωρώντας προς τα δεξιά.

1). Αν ο τρέχον χαρακτήρας δεν είναι παρενθετικός, τον αγνοούμε και συνεχίζουμε με τον επόμενο χαρακτήρα.

2). Αν ο τρέχον χαρακτήρας είναι αριστερός παρενθετικός χαρακτήρας, αν είναι δηλαδή ένας από τους χαρακτήρες `'('`, `'{'`, ή `'['` τον εισάγουμε στη στοίβα.

3). Αν ο τρέχον χαρακτήρας είναι δεξιός παρενθετικός χαρακτήρας, δηλαδή ')', '}', ή ']' και η στοίβα είναι κενή, η παρενθετική έκφραση είναι λανθασμένη. Διαφορετικά, εξάγουμε από τη στοίβα τον παρενθετικό χαρακτήρα που βρίσκεται στην κορυφή της. Αυτός πρέπει να είναι ο αριστερός παρενθετικός χαρακτήρας που αντιστοιχεί στον τρέχοντα χαρακτήρα· διαφορετικά η παράσταση δεν είναι ορθή.

Η παρενθετική έκφραση είναι ορθή, μόνο αν μετά από την ολοκλήρωση της διαδικασίας , η στοίβα είναι κενή. Η υλοποίηση δίνεται στον Κώδικα 3.3.

Κώδικας 3.3: Έλεγχος παρενθετικών παραστάσεων.

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4
5  import aueb.util.api.Stack;
6  import aueb.util.imp.ArrayStack;
7
8  public class BracketMatcher {
9     public boolean isValid(String s) {
10       int length = s.length();
11       Stack stack = new ArrayStack(length);
12       for (int i = 0; i < length; ++i) {
13          char nextChar = s.charAt(i);
14          switch (nextChar) {
15             case '(':
16             case '[':
17             case '{':
18                stack.push(new Character(nextChar));
19                continue;
20             case ')':
21                nextChar = '(';
22                break;
23             case ']':
24                nextChar = '[';
25                break;
26             case '}':
27                nextChar = '{';
28                break;
29             default:
30                continue;
31          }
32          char c = ((Character)stack.pop()).charValue();
```

```
33          if (c != nextChar) {
34             return false;
35          }
36       }
37       return stack.isEmpty();
38    }
39    public static void main(String[] args) throws IOException {
40       BufferedReader in = new BufferedReader(
41          new InputStreamReader(System.in)
42       );
43       BracketMatcher matcher = new BracketMatcher();
44       do {
45          System.out.println();
46          System.out.print("> ");
47          String expression = in.readLine();
48          if (expression.length() == 0) {
49             continue;
50          }
51          if ("quit".equals(expression)) {
52             break;
53          }
54          boolean result = matcher.isValid(expression);
55          System.out.print("Expression '");
56          System.out.print(expression);
57          System.out.print("' is ");
58          System.out.println(result ? "valid." : "not valid.");
59       } while (true);
60    }
61 }
```

Υπολογισμός επιθεματικών παραστάσεων. Αρκετές γλώσσες, όπως για παράδειγμα η γλώσσα PostScript με την οποία έχει δημιουργηθεί αυτό το κείμενο, χρησιμοποιούν επιθεματική μορφή αριθμητικών παραστάσεων. Σε μια *επιθεματική παράσταση* οι τελεστές δεν βρίσκονται μεταξύ των ορισμάτων τους, όπως έχουμε συνηθίσει στις ενδοθεματικές παραστάσεις, αλλά έπονται αυτών. Η επιθεματική παράσταση

```
1 1 x - / x + 1 2 / *
```

για παράδειγμα, αντιστοιχεί στην ενδοθεματική παράσταση

$$\frac{1}{2}\left(\frac{1}{1-x}+x\right). \tag{3.2}$$

Για να μετατρέψουμε μια επιθεματική παράσταση σε ενδοθεματική, αντικαθιστούμε κάθε δύο συνεχόμενα ορίσματα που ακολουθούνται από ένα τελεστή, στην αντίστοιχη ενδοθεματική μορφή την οποία περικλείουμε σε παρενθέσεις. Στη συνέχεια μετασχηματίζουμε την παράσταση που προκύπτει με τον ίδιο τρόπο, και συνεχίζουμε έτσι για να επεξεργαστούμε όλους τους τελεστές της παράστασης. Η παραπάνω επιθεματική παράσταση θα μετασχηματιστεί σε ενδοθεματική ως εξής:

```
1 1 x - / x + 1 2 / *
1 (1 - x) / x + (1 / 2) *
(1 / (1 - x)) x + (1 / 2) *
((1 / (1 - x)) + x) (1 / 2) *
((1 / (1 - x)) + x) * (1 / 2)
```

Σε αντίθεση με την ενδοθεματική μορφή, δεν χρειάζονται παρενθέσεις για τον υπολογισμό μιας επιθεματικής παράστασης και αυτό διότι η σειρά με την οποία θα εκτελεστούν οι πράξεις για τον υπολογισμό της τιμής της παράστασης είναι σαφώς καθορισμένη. Μπορούμε μάλιστα χρησιμοποιώντας μια στοίβα να εκτελέσουμε τις πράξεις, και να υπολογίσουμε την τιμή της παράστασης. Αρχικά πρέπει να αναλύσουμε την παράσταση σε λεκτικές μονάδες από τα αριστερά προς στα δεξιά.

Αλγόριθμος 3-1 (`CalculatePostfix`). Υπολογισμός επιθεματικής παράστασης.

1). Κατασκευάζουμε μια (κενή) στοίβα `s`.

2). Αν η ανάλυση της επιθεματικής παράστασης έχει ολοκληρωθεί και η `s` έχει ένα και μόνο στοιχείο, αυτό είναι και το αποτέλεσμα· ο αλγόριθμος τερματίζει με επιτυχία. Αν η `s` έχει μηδέν ή περισσότερα του ενός στοιχεία, ο αλγόριθμος τερματίζει· η παράσταση είναι λανθασμένη. Διαφορετικά συνεχίζουμε από το βήμα 3.

3). Έστω `t` η επόμενη λεκτική μονάδα της παράστασης. Αν η `t` είναι όρισμα, τότε την εισάγουμε στην `s` και συνεχίζουμε από το βήμα 2. Αν η `t` είναι τελεστής συνεχίζουμε από το βήμα 4.

4). Αν η στοίβα έχει λιγότερα από δύο στοιχεία, ο αλγόριθμος τερματίζει· η παράσταση είναι λανθασμένη. Διαφορετικά, εξάγουμε το όρισμα `b` και στη συνέχεια το όρισμα `a` από την `s`, εκτελούμε την πράξη που καθορίζει ο τελεστής `t` (π.χ., `a + b`) στα ορίσματα

1	**1**	
1	**1**	**1**
5	**1**	**1 5**
-	**1**	**-4**
/	**-0.25**	
5	**-0.25**	**5**
+	**4.75**	
1	**4.75**	**1**
2	**4.75**	**1 2**
/	**4.75**	**0.5**
*****	**2.375**	

Σχήμα 3-9: Υπολογισμός επιθεματικής παράστασης.
Το σχήμα παρουσιάζει τον υπολογισμό της επιθεματικής παράστασης **1 1 5 - / 5 + 1 2 / ***. Στην αριστερή πλευρά είναι ο τρέχον χαρακτήρας της παράστασης, ενώ στη δεξιά, τα περιεχόμενα της στοίβας.

a και b, αποθηκεύουμε το αποτέλεσμα στην s και συνεχίζουμε από το βήμα 2.

Ένα παράδειγμα υπολογισμού επιθεματικής παράστασης με τον παραπάνω αλγόριθμο δίνεται στο Σχήμα 3-9.

Για να αναλύσουμε μια παράσταση σε λεκτικές μονάδες, θα χρησιμοποιήσουμε την κλάση `StreamTokenizer` από το πακέτο `java.io`. Πρόκειται για ένα απλό λεκτικό αναλυτή, μια κλάση που παρέχει μεθόδους για την ομαδοποίηση ενός ρεύματος χαρακτήρων (character stream) σε λεκτικές μονάδες (tokenization). Κατασκευάζουμε ένα αντικείμενο τύπου `StreamTokenizer`, περνώντας όρισμα στον κατασκευαστή ένα αντικείμενο τύπου `Reader`, το οποίο θα χρησιμοποιεί ο λεκτικός αναλυτής για να θα διαβάζει χαρακτήρες. Αν θέλουμε να κάνουμε λεκτική ανάλυση σε ένα αντικείμενο τύπου `String`, τότε πρέπει να χρησιμοποιήσουμε την κλάση `StringReader`. Πρόκειται για ένα προσαρμογέα (adapter) που επιτρέπει σε ένα αντικείμενο τύπου `String` να εμφανίζεται ως αντικείμενο τύπου `Reader`. Έτσι, αν `expression` είναι αναφορά σε ένα αντικείμενο τύπου `String` το οποίο περιέχει την έκφραση την οποία θέλουμε να αναλύσουμε, κατασκευάζουμε ένα λεκτικό αναλυτή ως εξής:

```
StreamTokenizer tokenizer = new StreamTokenizer(
   new StringReader(expression)
);
```

Στην συνέχεια μπορούμε να διαβάσουμε μία-μία τις λεκτικές μονάδες με αλλεπάλληλες κλήσεις στη μέθοδο `nextToken()` της κλάσης `StreamTokenizer`, με τρόπο ανάλογο αυτού της διεπαφής `Enumerator`. Η μέθοδος `nextToken` δεν επιστρέφει μια λεκτική μονάδα, αλλά μια τιμή τύπου `int` που καθορίζει τον τύπο της λεκτικής μονάδας την οποία η μέθοδος συναρμολόγησε.

1). Αν η `nextToken()` επιστρέψει την τιμή `StreamTokenizer.TT_NUMBER`, η λεκτική μονάδα είναι ένας αριθμός· η τιμή του δίνεται από το δημόσιο πεδίο `nval`.
2). Αντίστοιχα, η τιμή `StreamTokenizer.TT_WORD`, υποδεικνύει πως η λεκτική μονάδα είναι μια συμβολοσειρά, που έχει αποθηκευτεί στο πεδίο `sval`.
3). Η τιμή `StreamTokenizer.TT_EOL`, υποδεικνύει το χαρακτήρα

αλλαγής γραμμής.

4). Αν η nextToken() επιστρέψει ένα θετικό ακέραιο, η λεκτική μονάδα είναι ο αντίστοιχος χαρακτήρας.

5). Τέλος, η τιμή StreamTokenizer.TT_EOF, υποδεικνύει το τέλος της λεκτικής ανάλυσης.

Το παρακάτω απόσπασμα κώδικα, επιδεικνύει η χρήση της κλάσης StreamTokenizer.

```
StreamTokenizer tokenizer = new StreamTokenizer(new StringReader(in));
   tokenizer.ordinaryChar('/');
   int token = 0;
   do {
      token = tokenizer.nextToken();
      switch (token) {
         case StreamTokenizer.TT_NUMBER:
            System.out.print("Number : ");
            System.out.println(tokenizer.nval);
            break;
         case StreamTokenizer.TT_WORD:
            System.out.print("Word : ");
            System.out.println(tokenizer.sval);
            break;
         case StreamTokenizer.TT_EOL:
            System.out.println("End of line");
            break;
         case StreamTokenizer.TT_EOF:
            break;
         default:
            System.out.print("Char : ");
            System.out.println((char)token);
      }
   } while (token != StreamTokenizer.TT_EOF);
```

Με τα παραπάνω στη διάθεσή μας, μπορούμε να κατασκευάσουμε ένα απλό υπολογιστή επιθεματικών παραστάσεων. Η υλοποίηση δίνεται στον Κώδικα 3.4.

Κώδικας 3.4: Υπολογισμός επιθεματικών παραστάσεων.

```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.io.StreamTokenizer;
5 import java.io.StringReader;
6 import java.text.ParseException;
7
```

```
8  import aueb.util.api.Stack;
9  import aueb.util.imp.ArrayStack;
10
11 public class PostfixCalculator {
12
13    private Stack stack;
14
15    public double valueOf(String e) throws IOException, ParseException {
16       stack = new ArrayStack(e.length());
17       StreamTokenizer tokenizer = new StreamTokenizer(
18          new StringReader(e)
19       );
20       tokenizer.ordinaryChar('/');
21       int token = 0;
22       do {
23          token = tokenizer.nextToken();
24          switch (token) {
25             case StreamTokenizer.TT_NUMBER:
26                stack.push(new Double(tokenizer.nval));
27                break;
28             case '+':
29                add();
30                break;
31             case '-':
32                subtract();
33                break;
34             case '*':
35                multiply();
36                break;
37             case '/':
38                divide();
39                break;
40             case StreamTokenizer.TT_EOF:
41                break;
42             default:
43                throw new ParseException(e, tokenizer.lineno());
44          }
45       } while (token != StreamTokenizer.TT_EOF);
46       return ((Double)stack.pop()).doubleValue();
47    }
48
49    private void add() {
50       double arg2 = ((Double)stack.pop()).doubleValue();
51       double arg1 = ((Double)stack.pop()).doubleValue();
52       stack.push(new Double(arg1 + arg2));
53    }
```

```
54
55    private void subtract() {
56       double arg2 = ((Double)stack.pop()).doubleValue();
57       double arg1 = ((Double)stack.pop()).doubleValue();
58       stack.push(new Double(arg1 - arg2));
59    }
60
61    private void multiply() {
62       double arg2 = ((Double)stack.pop()).doubleValue();
63       double arg1 = ((Double)stack.pop()).doubleValue();
64       stack.push(new Double(arg1 * arg2));
65    }
66
67    private void divide() {
68       double arg2 = ((Double)stack.pop()).doubleValue();
69       double arg1 = ((Double)stack.pop()).doubleValue();
70       stack.push(new Double(arg1 / arg2));
71    }
72
73    public static void main(String[] args) throws IOException {
74       BufferedReader in = new BufferedReader(
75          new InputStreamReader(System.in)
76       );
77       PostfixCalculator calc = new PostfixCalculator();
78       do {
79          System.out.println();
80          System.out.print("> ");
81          String expression = in.readLine();
82          if (expression.length() == 0) {
83             continue;
84          }
85          if ("quit".equals(expression)) {
86             break;
87          }
88          double result = 0.0;
89          try {
90             result = calc.valueOf(expression);
91             System.out.print(result);
92          }
93          catch (ParseException e) {
94             System.out.println("Invalid postfix expression");
95          }
96       } while (true);
97    }
98 }
```

a	a		
*	a	*	
(	a	*	(
1	a 1	*	(
-	a 1	*	(-
b	a 1 b	*	(-
)	a 1 b -	*	
+	a 1 b - *	+	
log	a 1 b - *	+	log
(	a 1 b - *	+	log (
1	a 1 b - * 1	+	log (
-	a 1 b - * 1	+	log (-
a	a 1 b - * 1 a	+	log (-
)	a 1 b - * 1 a -	+	log
	a 1 b - * 1 a - log +		

Σχήμα 3-10: Μετατροπή ενδοθεματικής παράστασης σε επιθεματική μορφή. Το σχήμα παρουσιάζει την διαδικασία μετατροπής της ενδοθεματικής παράστασης **a * (1-b) + log(1-a)**. Στην αριστερή στήλη είναι ο τρέχον χαρακτήρας της παράστασης, στη μεσαία στήλη τα τρέχοντα περιεχόμενα της επιθεματικής παράστασης, ενώ τέλος, στη δεξιά στήλη τα περιεχόμενα της στοίβας.

Κατασκευή επιθεματικών παραστάσεων. Έχουμε αναλύσει τον τρόπο με τον οποίο εκτελείται ο υπολογισμός μιας επιθεματικής παράστασης. Έχουμε επίσης περιγράψει ένα αλγόριθμο για την μετατροπή μιας επιθεματικής παράστασης σε ενδοθεματική. Δεν έχουμε αναλύσει ωστόσο πως μπορεί κανείς να κάνει το αντίστροφο, να μετατρέψει δηλαδή μια ενδοθεματική παράσταση σε επιθεματική. Και καθώς οι περισσότεροι άνθρωποι έχουν πολύ μεγαλύτερη άνεση στην ανάγνωση και τον υπολογισμό ενδοθεματικών παραστάσεων, ένας τέτοιος αλγόριθμος θα ήταν πολύτιμος.

Η διαδικασία της μετατροπής μιας παράστασης από ενδοθεματική σε επιθεματική μορφή, είναι πιο σύνθετη σε σχέση με τη διαδικασία του υπολογισμού της τιμής μιας επιθεματικής παράστασης. Αυτό οφείλεται στην *προτεραιότητα των τελεστών* μιας ενδοθεματικής παράστασης. Ενώ για ένα άνθρωπο είναι αυτονόητο πως στον υπολογισμό της ενδοθεματικής παράστασης (3.2) στη σελίδα 98, η αφαίρεση $1-x$, θα πρέπει να προηγηθεί της διαίρεσης, για ένα πρόγραμμα κάτι τέτοιο δεν είναι καθόλου προφανές. Στις ενδοθεματικές παραστάσεις, κάθε τελεστής έχει μια προτεραιότητα, ένα ακέραιο που καθορίζει τη σειρά με την οποία θα εφαρμοστούν οι τελεστές της παράστασης: οι τελεστές με υψηλότερη προτεραιότητα εφαρμόζονται πριν από τους τελεστές χαμηλότερης προτεραιότητας. Καθώς η εξοικείωσή μας με τις ενδοθεματικές παραστάσεις είναι πολύ μεγάλη, ο υπολογισμός τους, και κατά συνέπεια η σειρά εφαρμογής των τελεστών στα ορίσματά τους, είναι μια σχεδόν αυτοματοποιημένη διαδικασία την οποία εκτελούμε τηρώντας μεν την προτεραιότητα των τελεστών, χωρίς πάντα να έχουμε πλήρη επίγνωση του λόγου για τον οποίο λειτουργούμε κατά αυτό τον τρόπο.

Πιθανότατα δεν προκαλεί έκπληξη το γεγονός ότι το εργαλείο που χρειαζόμαστε είναι, και σε αυτή την περίπτωση, μια στοίβα. Οι κανόνες μετατροπής μιας παράστασης από ενδοθεματική σε επιθεματική μορφή, δίνονται παρακάτω. Ο αλγόριθμος διαβάζει μια ενδοθεματική παράσταση, ως συνήθως, από τα αριστερά προς τα δεξιά παράγοντας μια επιθεματική παράσταση. Η βασική ιδέα είναι να εισάγουμε σε μια στοίβα τους τελεστές καθώς τους συναντάμε, ενώ αντιγράφουμε τα ορίσματα στην επιθεματική παράσταση. Για να εξασφαλίσουμε την ορθότητα της σειράς υπολογισμών ωστόσο, πρέπει πριν εισάγουμε ένα τελεστή στη στοίβα, να εξάγουμε όλους τους τελεστές υψηλότερης προτεραιότητας και να τους αντιγράψου-

με στην επιθεματική παράσταση. Το Σχήμα 3-10 παρουσιάζει ένα παράδειγμα της διαδικασίας αυτής.

1). Αν η τρέχουσα λεκτική μονάδα είναι όρισμα, την αντιγράφουμε στην επιθεματική παράσταση.
2). Αν η τρέχουσα λεκτική μονάδα είναι αριστερή παρένθεση, την εισάγουμε στη στοίβα.
3). Αν η τρέχουσα λεκτική μονάδα είναι δεξιά παρένθεση, εξάγουμε από τη στοίβα και αντιγράφουμε στην επιθεματική παράσταση, όλους τους τελεστές μέχρι να εξάγουμε μια αριστερή παρένθεση την οποία και αγνοούμε.
4). Αν η τρέχουσα λεκτική μονάδα είναι ο τελεστής op, τότε εξάγουμε από τη στοίβα και αντιγράφουμε στην επιθεματική παράσταση, όλους τους τελεστές μέχρι να συναντήσουμε στην κορυφή της στοίβας αριστερή παρένθεση ή τελεστή με προτεραιότητα μικρότερη της προτεραιότητας του τελεστή op ή μέχρι να αδειάσει η στοίβα. Στη συνέχεια εισάγουμε τον τελεστή op στη στοίβα.
5). Όταν η ενδοθεματική παράσταση εξαντληθεί, εξάγουμε από τη στοίβα και αντιγράφουμε στην επιθεματική παράσταση, όλους τους τελεστές.

Ασκήσεις

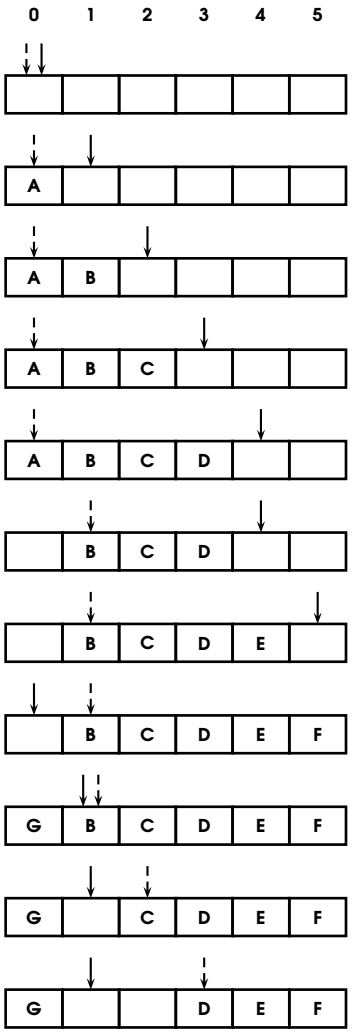
3-13 Σχεδιάστε την κατάσταση ενός αντικειμένου τύπου `ArrayStack` χωρητικότητας 10 το οποίο αρχικά δεν περιέχει κανένα αντικείμενο, έπειτα από την εκτέλεση των λειτουργιών **C O U - R T - R E A - C T I O N - - -** όπου ένα γράμμα υποδεικνύει εισαγωγή του αντίστοιχου αντικειμένου, ενώ μια παύλα υποδεικνύει διαγραφή.

3-14 Χρησιμοποιείστε μια στοίβα για να αντιστρέψετε μια συμβολοσειρά.

3-15 Χρησιμοποιείστε μια στοίβα για να αντιστρέψετε τη διάταξη ενός αντικειμένου τύπου `Sequence`.

3-16 Μετατρέψτε τις επιθεματικές παραστάσεις **1 1 x - / x + 0.5 *** και **1 x / 1 1 x - / ln *** σε ενδοθεματικές.

3-17 Υπολογίστε την τιμή της επιθεματικής παράστασης **1 5 - 4 * 3 - ln** χρησιμοποιώντας τον αλγόριθμο που παρουσιάστηκε στην ενότητα αυτή.



Σχήμα 3-11: Λειτουργία ουράς αναμονής που υλοποιείται με συστοιχία. Το διακεκομμένο και το συμπαγές βέλος δείχνουν αντίστοιχα την αρχή και το τέλος της ουράς. Η αρχή της ουράς μετακινείται κάθε φορά που εξάγεται ένα αντικείμενο. Τόσο η αρχή όσο και το τέλος της ουράς μετακινούνται με κυκλικό τρόπο.

3-18 Μετατρέψτε την ενδοθεματική παράσταση

$$\frac{1}{2}\left(\alpha + \frac{1}{1-\alpha}\right)$$

σε επιθεματική, χρησιμοποιώντας τον αλγόριθμο που παρουσιάστηκε στην ενότητα αυτή.

3-19 Βελτιώστε την κλάση `PostfixCalculator` ώστε να υποστηρίζει τον εκθετικό τελεστή `^`.

3-20 Βελτιώστε την κλάση `PostfixCalculator` ώστε να υποστηρίζει μονομελείς τελεστές.

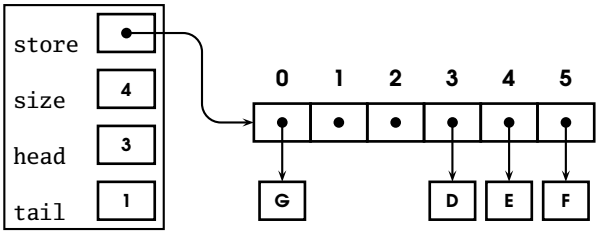
3-21 Η γλώσσα PostScript υποστηρίζει τους τελεστές `add` (πρόσθεση), `sub` (αφαίρεση), `mul` (πολλαπλασιασμός), `div` (διαίρεση), `exp` (ύψωση σε δύναμη), `ln` (φυσικός λογάριθμος), `sin` (ημίτονο), `cos` (συνημίτονο) και`tan` (εφαπτομένη) μεταξύ άλλων. Υλοποιήστε μια κλάση `PostScriptCalculator` στα πρότυπα της κλάσης `PostfixCalculator`, η οποία υπολογίζει επιθεματικές παραστάσεις της γλώσσας PostScript.

3-22 Υλοποιήστε μια μέθοδο της κλάσης `PostfixCalculator` η οποία μετατρέπει μια ενδοθεματική παράσταση σε επιθεματική.

3.3 Ουρές αναμονής

Κάθε σημείο εξυπηρέτησης το οποίο λειτουργεί με τον κανόνα first in, first out διαθέτει ένα χώρο στον οποίο οι εξυπηρετούμενοι αναμένουν προκειμένου να εξυπηρετηθούν. Ο χώρος αναμονής έχει πεπερασμένη χωρητικότητα. Η αρχή της ουράς βρίσκεται ακριβώς μπροστά από το σημείο εξυπηρέτησης, ενώ το τέλος της ουράς είναι το σημείο στο οποίο βρίσκεται ο τελευταίος πελάτης. Όταν ένας νέος πελάτης καταφθάνει στην ουρά αναμονής, πλησιάζει στην αρχή της ουράς όσο περισσότερο γίνεται. Όταν η εξυπηρέτηση του πελάτη που βρίσκεται στην αρχή της ουράς ολοκληρωθεί, οι πελάτες που έπονται μετακινούνται κατά μία θέση πλησιέστερα στην ουρά.

Αν το λογισμικό αντίστοιχο του χώρου αναμονής είναι μια συστοιχία, τότε η υλοποίηση του πρωτοκόλλου της ουράς είναι προφανής. Η αρχή της ουράς είναι η θέση 0 της συστοιχίας, ενώ το τέλος της ουράς είναι η επόμενη της τελευταίας κατειλημμένης θέσης. Κάθε φορά που εξάγεται ένα αντικείμενο από την ουρά, όλα τα



Σχήμα 3-12: Υλοποίηση ουράς αναμονής με συστοιχία.
Απεικονίζεται η κατάσταση της ουράς έπειτα από την εκτέλεση των λειτουργιών **A B C D - E F G - -** όπου ένα γράμμα συμβολίζει εισαγωγή του αντίστοιχου αντικειμένου, ενώ μια παύλα εξαγωγή.

αντικείμενα πρέπει να μετακινηθούν μια θέση "προς τα αριστερά". Αυτή η μετακίνηση ωστόσο, συνεπάγεται N αντιγραφές αναφορών, όπου N είναι το πλήθος των αντικειμένων στην ουρά. Μπορούμε να βελτιώσουμε την αποτελεσματικότητα της εξαγωγής αν θεωρήσουμε πως αντί να μετακινούνται τα αντικείμενα προς την αρχή της ουράς, μετακινείται η αρχή της ουράς προς το επόμενο αντικείμενο όπως επιδεικνύεται στο Σχήμα 3-11. Καθώς τόσο η αρχή όσο και το τέλος της ουράς αναμονής μετακινούνται με κυκλικό τρόπο, ο μηχανισμός αυτός είναι γνωστός και ως *κυκλική ουρά αναμονής*.

Για την υλοποίηση αυτού του μηχανισμού, χρειάζονται δύο πεδία `head` και `tail` τύπου `int`. Το δεύτερο πεδίο είναι το τέλος της ουράς, η θέση δηλαδή της συστοιχίας στην οποία θα γίνει η επόμενη εισαγωγή, ενώ το πρώτο είναι η αρχή της ουράς, η θέση δηλαδή της συστοιχίας από την οποία θα γίνει η επόμενη εξαγωγή. Επιπλέον, προκειμένου να ελέγχουμε εύκολα το πλήθος των στοιχείων της ουράς και να αποφασίζουμε πότε μπορεί να γίνει εισαγωγή ή εξαγωγή και πότε όχι, χρειαζόμαστε ένα επιπλέον πεδίο `size`. Τέλος, απαιτείται μια συστοιχία για την αποθήκευση των αναφορών στα αντικείμενα που βρίσκονται στην ουρά. Η οργάνωση της ουράς στο τέλος του Σχήματος 3-11 παρουσιάζεται στο Σχήμα 3-12.

Όταν μια ουρά είναι κενή, τότε τα πεδία `size`, `head` και `tail` έχουν την τιμή 0. Όταν εισάγεται ένα αντικείμενο, η τιμή των πεδίων `tail` και `size` αυξάνεται κατά 1. Αντίστοιχα όταν εξάγεται το αντικείμενο που βρίσκεται στην αρχή της ουράς αναμονής, η τιμή του πεδίου `head` αυξάνεται κατά 1, ενώ η τιμή του πεδίου `size` μειώνεται κατά 1. Τα πεδία `head` και `tail` αυξάνουν κυκλικά με βάση τη χωρητικότητα της ουράς αναμονής.

Η κλάση `ArrayQueue` η οποία παρουσιάζεται στον Κώδικα 3.5, είναι η υλοποίηση μιας ουράς αναμονής με βάση την οργάνωση

put	$O(1)$
get	$O(1)$
head	$O(1)$
tail	$O(1)$
isEmpty	$O(1)$
isFull	$O(1)$
clone	$O(N)$

Πίνακας 3-3: Αποτελεσματικότητα των μεθόδων της κλάσης ArrayQueue.

που μόλις περιγράψαμε. Η κλάση είναι σχεδόν πανομοιότυπη της ArrayStack, λειτουργεί ωστόσο με εντελώς διαφορετική λογική. Όλες σχεδόν οι μέθοδοι απαιτούν σταθερό χρόνο εκτέλεσης όπως φαίνεται στον Πίνακα 3-3.

Κώδικας 3.5: Η κλάση ArrayQueue.

```
1  package aueb.util.imp;
2
3  import aueb.util.api.Queue;
4  /**
5   * Array-based queue implementation.
6   */
7  public final class ArrayQueue implements Queue {
8     /**
9      * Default queue capacity.
10     */
11    private static final int INITIAL_CAPACITY = 8;
12    /**
13     * The array that stores references to queued elements.
14     */
15    private Object[] store;
16    /**
17     * The number of elements queued.
18     */
19    private int size;
20    /**
21     * Index of the head element of this queue.
22     */
23    private int head;
24    /**
25     * Index of the tail element of this queue.
26     */
27    private int tail;
28    /**
29     * Default constructor. Uses default capacity and creates
30     * an empty queue.
31     */
32    public ArrayQueue() {
33       this(INITIAL_CAPACITY);
34    }
35    /**
36     * Creates an empty queue that has a specified capacity.
37     * @param capacity The capacity of this stack.
38     */
39    public ArrayQueue(int capacity) {
```

```
40        if (capacity < 0) throw new IllegalArgumentException();
41        store = new Object[capacity];
42        size = head = tail = 0;
43     }
44     public void put(Object element) {
45        if (size == store.length) throw new IllegalStateException();
46        if (element == null) throw new IllegalArgumentException();
47        store[tail] = element;
48        tail = ++tail % store.length;
49        ++size;
50     }
51     public Object get() {
52        if (size == 0) throw new IllegalStateException();
53        Object element = store[head];
54        store[head] = null;
55        head = ++head % store.length;
56        --size;
57        return element;
58     }
59     public Object head() {
60        if (size == 0) throw new IllegalStateException();
61        return store[head];
62     }
63     public Object tail() {
64        if (size == 0) throw new IllegalStateException();
65        return store[tail];
66     }
67     public int size() {
68        return size;
69     }
70     public boolean isEmpty() {
71        return size == 0;
72     }
73     public boolean isFull() {
74        return size == store.length;
75     }
76  }
```

Ασκήσεις

3-23 Σχεδιάστε την κατάσταση ενός αντικειμένου τύπου ArrayQueue χωρητικότητας 10 το οποίο αρχικά δεν περιέχει κανένα αντικείμενο, έπειτα από την εκτέλεση των λειτουργιών **C O U - R T - R E A - C T I O N - - -** όπου ένα γράμμα υποδεικνύει εισαγωγή του αντίστοιχου

αντικειμένου, ενώ μια παύλα υποδεικνύει εξαγωγή από την ουρά.

3-24 Σχεδιάστε την κατάσταση ενός αντικειμένου τύπου ArrayQueue χωρητικότητας 5 το οποίο αρχικά δεν περιέχει κανένα αντικείμενο, έπειτα από την εκτέλεση των λειτουργιών **E A S - Y - Q U E - - - S T - - - I O - N - - -** όπου ένα γράμμα υποδεικνύει εισαγωγή του αντίστοιχου αντικειμένου, ενώ μια παύλα υποδεικνύει εξαγωγή από την ουρά.

3-25 Υλοποιήστε τη μέθοδο clone για την κλάση ArrayQueue.

3-26 Προσθέστε δύο μεθόδους στην κλάση ArrayQueue οι οποία παρέχουν τη δυνατότητα να αντικαταστήσουμε το αντικείμενο στην αρχή και το τέλος της ουράς αντίστοιχα.

3-27 Δώστε μια υλοποίηση της διεπαφής Queue η οποία δεν επιτρέπει διπλότυτα. Υπολογίστε τον απαιτούμενο χρόνο εκτέλεσης της μεθόδου put.

3-28 Δώστε μια υλοποίηση της διεπαφής Queue η οποία παρέχει τη δυνατότητα αντικατάστασης ενός αντικειμένου που βρίσκεται στην ουρά αναμονής.

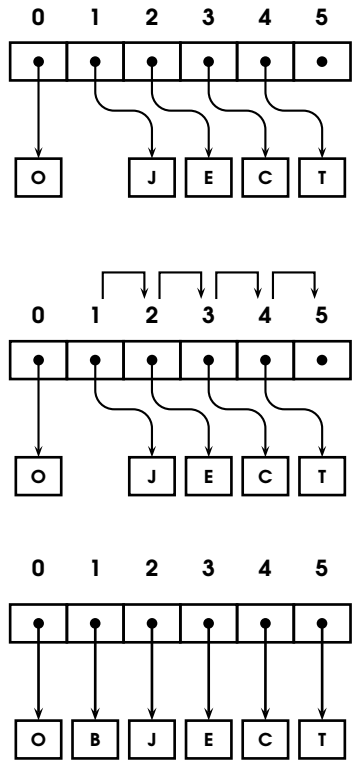
3.4 Ακολουθίες

Μια ακολουθία N αντικειμένων είναι μια συλλογή στην οποία κάθε αντικείμενο βρίσκεται αποθηκευμένο σε μια συγκεκριμένη θέση. Καθώς κάτι ανάλογο συμβαίνει και με τις συστοιχίες, μπορούμε να επεκτείνουμε την κλάση ArrayCollection έτσι ώστε να υποστηρίξουμε τις λειτουργίες της διεπαφής Sequence. Μπορούμε μάλιστα, να κληρονομήσουμε τις περισσότερες μεθόδους από την κλάση ArrayCollection. Υπάρχουν δύο σημεία στα οποία η υλοποίηση μιας ακολουθίας με χρήση συστοιχιών διαφοροποιείται από την αντίστοιχη υλοποίηση μιας συλλογής.

Η πρώτη διαφοροποίηση έχει να κάνει με τις μεθόδους οι οποίες λειτουργούν με βάση τη θέση ενός αντικειμένου στην ακολουθία. Όταν θέλουμε να εισάγουμε ένα αντικείμενο στη θέση i, επεκτείνουμε τη συστοιχία αν χρειάζεται χρησιμοποιώντας τη μέθοδο expand, μετακινούμε τα στοιχεία που βρίσκονται στις θέσεις i ως και $N-1$ της συστοιχίας store κατά μία θέση δεξιά (right shift), και τέλος αποθηκεύουμε το νέο αντικείμενο στη θέση i. Η διαδικασία αυτή παρουσιάζεται στο Σχήμα 3-13. Αντίστοιχα, για να διαγράψουμε το αντικείμενο που βρίσκεται στη θέση i της ακολουθίας, μετακινούμε

τις θέσεις $i+1$ ως $N-1$ της συστοιχίας `store`, κατά μία θέση αριστερά (left shift). Οι λειτουργίες αυτές μπορούν να υλοποιηθούν εύκολα με κατάλληλες κλήσεις στη στατική μέθοδο `arraycopy` της κλάσης `java.lang.System`.

Η δεύτερη διαφοροποίηση προκύπτει από την απαίτηση για υποστήριξη δρομέων. Ο δρομέας της κλάσης `ArraySequence` δεν είναι ωστόσο παρά μια φυσική επέκταση του απαριθμητή της κλάσης `ArrayCollection`. Η κατάσταση ενός δρομέα περιλαμβάνει, εκτός από την επόμενη θέση της ακολουθίας η οποία θα προσπελαστεί, την τελευταία θέση η οποία προσπελάστηκε. Οι μέθοδοι `next`, `previous`, `hasNext`, `hasPrevious`, `nextIndex` και `previousIndex` υλοποιούνται όπως ακριβώς περιγράψαμε στην Ενότητα 2.7. Τέλος, οι μέθοδοι ενός δρομέα για την τροποποίηση της ακολουθίας κατά τη διάσχισή της, υλοποιούνται μέσω κατάλληλων κλήσεων στις αντίστοιχες μεθόδους του περικλείοντος αντικειμένου τύπου `ArraySequence`. Η ολοκληρωμένη υλοποίηση, παρουσιάζεται στον Κώδικα 3.6.



Σχήμα 3-13: Εισαγωγή αντικειμένου σε ακολουθία που υλοποιείται με συστοιχία.
Για την εισαγωγή ενός αντικειμένου στη θέση i απαιτείται πρώτα η μετακίνηση των αναφορών στις θέσεις i ως $N-1$ κατά μία θέση προς τα δεξιά.

Κώδικας 3.6: Η κλάση `ArraySequence`.

```
1  package aueb.util.imp;
2
3  import aueb.util.api.Iterator;
4  import aueb.util.api.Sequence;
5  /**
6   * Array-based implementation of a sequence of objects.
7   */
8  public class ArraySequence extends ArrayCollection implements Sequence {
9      /**
10      * Iterator implementation.
11      */
12     public final class ArrayIterator implements Iterator {
13         /**
14          * The position of this iterator. A call to next() will
15          * return the object store[position], while a call to previous
16          * will return the object store[position-1].
17          */
18         private int position;
19         /**
20          * The previous position of this iterator. A value of -1
21          * indicates that there is no previous position.
22          */
23         private int lastPosition;
```

```
24        /**
25         * Creates an iterator on position 0.
26         */
27        public ArrayIterator() {
28           this(0);
29        }
30        /**
31         * Creates an iterator on a specified position.
32         * @param i The position of the iterator.
33         */
34        public ArrayIterator(int i) {
35           if (i < 0 || i > size) throw new IndexOutOfBoundsException();
36           position = i;
37           lastPosition = -1;
38        }
39        public Object next() {
40           if (position == size) throw new IllegalStateException();
41           return store[lastPosition = position++];
42        }
43        public Object previous() {
44           if (position == 0) throw new IllegalStateException();
45           return store[lastPosition = --position];
46        }
47        public int nextIndex() {
48           return position;
49        }
50        public int previousIndex() {
51           return position - 1;
52        }
53        public boolean hasNext() {
54           return position != size;
55        }
56        public boolean hasPrevious() {
57           return position != 0;
58        }
59        public void add(Object object) {
60           ArraySequence.this.add(position, object);
61           ++position;
62        }
63        public void remove() {
64           if (lastPosition == -1) throw new IllegalStateException();
65           ArraySequence.this.remove(lastPosition);
66           if (lastPosition < position) --position;
67           lastPosition = -1;
68        }
69        public void set(Object object) {
```

```
70          if (lastPosition == -1) throw new IllegalStateException();
71          ArraySequence.this.set(lastPosition, object);
72       }
73    }
74    /**
75     * Creates an empty sequence with the default initial capacity.
76     */
77    public ArraySequence() {
78       super();
79    }
80    /**
81     * Creates an empty sequence with a given initial capacity.
82     * @param capacity The initial capacity of this sequence.
83     */
84    public ArraySequence(int capacity) {
85       super(capacity);
86    }
87    /**
88     * Copy constructor.
89     * @param sequence The sequence to copy.
90     */
91    public ArraySequence(Sequence sequence) {
92       this(sequence.size());
93       Iterator i = sequence.iterator();
94       while (i.hasNext()) add(i.next());
95    }
96    public void add(int i, Object object) {
97       if (object == null) throw new IllegalArgumentException();
98       if (i < 0 || i > size) throw new IndexOutOfBoundsException();
99       if (size == store.length) expand();
100      System.arraycopy(store, i, store, i+1, size-i);
101      store[i] = object;
102      ++size;
103   }
104   /**
105    * Overrides {@link ArrayCollection#remove(Object)}.
106    */
107   public boolean remove(Object object) {
108      int i = indexOf(object);
109      if (i == -1) return false;
110      remove(i);
111      return true;
112   }
113   public Object remove(int i) {
114      if (i < 0 || i >= size) throw new IndexOutOfBoundsException();
115      Object object = store[i];
```

	$W(N)$	$A(N)$
`add(Object)`	$O(N)$	$O(1)$*
`add(int,Object)`	$O(N)$	$O(N)$
`remove(Object)`	$O(N)$	$O(N)$
`remove(int)`	$O(N)$	$O(N)$
`contains`	$O(N)$	$O(N)$
`indexOf`	$O(N)$	$O(N)$
`get`	$O(1)$	$O(1)$
`set`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`isEmpty`	$O(1)$	$O(1)$
`iterator()`	$O(1)$	$O(1)$
`iterator(int)`	$O(1)$	$O(1)$

Πίνακας 3-4: Αποτελεσματικότητα των μεθόδων της κλάσης `ArraySequence` **όταν το μέγεθος της ακολουθίας είναι** N.
Ο αστερίσκος υποδηλώνει πως ο χρόνος εκτέλεσης της μεθόδου είναι επιμερισμένος σε N διαδοχικές κλήσεις.

```
116         System.arraycopy(store, i+1, store, i, size-i-1);
117         store[--size] = null;
118         return object;
119      }
120      public Object get(int i) {
121         if (i < 0 || i >= size) throw new IndexOutOfBoundsException();
122         return store[i];
123      }
124      public Object set(int i, Object object) {
125         if (i < 0 || i >= size) throw new IndexOutOfBoundsException();
126         Object tmp = store[i];
127         store[i] = object;
128         return tmp;
129      }
130      public Iterator iterator() {
131         return new ArrayIterator();
132      }
133      public Iterator iterator(int position) {
134         return new ArrayIterator(position);
135      }
136   }
```

Για τη μέθοδο `add(int, Object)` η ιδανική περίπτωση είναι η εισαγωγή στην τελευταία θέση της ακολουθίας εφόσον η διαθέσιμη χωρητικότητα δεν έχει εξαντληθεί. Η χειρότερη περίπτωση είναι εισαγωγή στην αρχή της ακολουθίας, πράγμα που απαιτεί μετακίνηση όλων των στοιχείων της συστοιχίας κατά μία θέση προς τα δεξιά. Ο αναμενόμενος χρόνος είναι $O(N)$. Αντίστοιχα είναι τα συμπεράσματα και για τις μεθόδους `remove(Object)` και `remove(int)`. Οι μέθοδοι πρόσβασης αντίθετα, καθώς και οι μέθοδοι τοποθέτησης του δρομέα, απαιτούν σταθερό χρόνο.

Ασκήσεις

3-29 Υλοποιήστε τη μέθοδο `void concat(Sequence s)` η οποία εισάγει τα αντικείμενα της ακολουθίας `s` στο τέλος μιας ακολουθίας.

3-30 Δώστε μια αποτελεσματική υλοποίηση της μεθόδου `equals` για την κλάση `ArraySequence`.

3-31 Υλοποιήστε τη μέθοδο `void shuffle()` η οποία μεταθέτει κατά τυχαίο τρόπο τα αντικείμενα μιας ακολουθίας.

3-32 Υλοποιήστε ένα κατασκευαστή της κλάσης `ArraySequence` ο οποίος

δέχεται ως παράμετρο ένα αντικείμενο τύπου `Object[]` και κατασκευάζει την αντίστοιχη ακολουθία.

3-33 Υλοποιήστε τη μέθοδο `void reverse()` για την κλάση `ArraySequence` η οποία αντιστρέφει τη διάταξη των αντικειμένων μιας ακολουθίας.

3-34 Υλοποιήστε τη μέθοδο `void sort()` για την κλάση `ArraySequence` η οποία ταξινομεί μια ακολουθία χρησιμοποιώντας τον αλγόριθμο quick sort.

ΚΕΦΑΛΑΙΟ 4

Συνδεδεμένες λίστες

Παρά το γεγονός ότι έχουμε καταφέρει, χρησιμοποιώντας συστοιχίες, να υλοποιήσουμε ορισμένες σημαντικές, και σε μερικές περιπτώσεις, αποτελεσματικές δομές δεδομένων, δύο σημαντικοί περιορισμοί παραμένουν. Δεν έχουμε στη διάθεσή μας ένα φυσικό τρόπο για να μεταβάλουμε το μέγεθος μιας συστοιχίας μετά τη δημιουργία της. Στην υλοποίηση της κλάσης `ArrayCollection` χρησιμοποιήσαμε την τεχνική του διπλασιασμού για να ξεπεράσουμε τον περιορισμό αυτό. Παρά το γεγονός πως ο χρόνος εισαγωγής είναι σταθερός αν επιμερίσουμε το χρόνο που απαιτείται για το διπλασιασμό της συστοιχίας σε κάθε μία κλήση της μεθόδου `add(Object)`, ο χρόνος που απαιτείται για την εισαγωγή αντικειμένου δεν παύει να είναι ανάλογος του μεγέθους της συστοιχίας στη χειρότερη περίπτωση. Επιπλέον, δεν μπορούμε να εισάγουμε ή να διαγράψουμε αντικείμενα σε μια συστοιχία, χωρίς να χρειαστεί να καταφύγουμε στη μετακίνηση αναφορών. Στο κεφάλαιο αυτό θα εξετάσουμε ένα τρόπο οργάνωσης αντικειμένων σε μια δομή που δεν πάσχει από αυτά τα μειονεκτήματα. Η οργάνωση αυτή ονομάζεται *συνδεδεμένη λίστα* ή *αλυσίδα*.

4.1 Μονή σύνδεση

Τα προβλήματα που αντιμετωπίζουμε στην οργάνωση αντικειμένων με χρήση συστοιχιών οφείλονται στο γεγονός ότι τα στοιχεία μιας συστοιχίας αποθηκεύονται αναγκαστικά σε συνεχόμενες θέσεις μνή-

μης. Με τον τρόπο αυτό έχουμε ένα φυσικό μηχανισμό για να συγκρατούμε αντικείμενα σε μια δομή. Επιπλέον, η αποθήκευση αντικειμένων σε συνεχόμενες θέσεις μνήμης αποτελεί ένα πλεονέκτημα όταν θέλουμε να εντοπίσουμε το αντικείμενο που βρίσκεται σε μια συγκεκριμένη θέση της συστοιχίας, αλλά αποτελεί μειονέκτημα όταν θέλουμε να παρεμβάλουμε ένα αντικείμενο σε μια συγκεκριμένη θέση.

Αν άρουμε τη δέσμευση για την αποθήκευση των στοιχείων σε συνεχόμενες θέσεις μνήμης, τότε η παρεμβολή ενός αντικειμένου γίνεται πολύ εύκολη και αποτελεσματική όπως θα δούμε στις επόμενες παραγράφους. Σε μια τέτοια περίπτωση ωστόσο, θα πρέπει να επινοήσουμε ένα νέο μηχανισμό με τον οποίο θα συγκρατούμε τα αντικείμενα σε μια δομή.

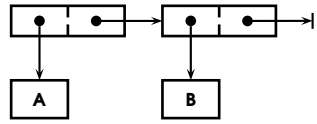
Ορισμός 4-1. Μια *λίστα μονής σύνδεσης* είναι ένα σύνολο *κόμβων* που αποθηκεύουν αντικείμενα. Κάθε κόμβος είναι ένα αντικείμενο με δύο πεδία: μια αναφορά στο αντικείμενο το οποίο ο κόμβος αποθηκεύει, και μια αναφορά στον επόμενο κόμβο της λίστας.

Το Σχήμα 4-1 παρουσιάζει μια λίστα με δύο κόμβους που αποθηκεύουν τα αντικείμενα **A** και **B**. Κάθε κόμβος παριστάνεται με ένα "κουτί" χωρισμένο σε δύο διαμερίσματα, ένα για κάθε πεδίο του κόμβου. Το αριστερό διαμέρισμα είναι η αναφορά στο αντικείμενο που ο κόμβος αποθηκεύει, ενώ το δεξί διαμέρισμα είναι η αναφορά στον επόμενο κόμβο της λίστας. Η αναφορά αυτή ονομάζεται *σύνδεσμος* ή *σύνδεση*.

Στη συνδεδεμένη οργάνωση αντικειμένων, οι κόμβοι παίζουν το ρόλο των θέσεων μιας συστοιχίας. Εκεί αποθηκεύονται τα αντικείμενα τα οποία ανήκουν στη δομή. Επειδή ωστόσο ένας κόμβος μπορεί να βρίσκεται αποθηκευμένος σε οποιαδήποτε θέση της μνήμης, κάθε κόμβος διατηρεί ένα σύνδεσμο, ένα *δείκτη*, προς τον επόμενο κόμβο της λίστας. Οι σύνδεσμοι αποτελούν το μηχανισμό με τον οποίο συγκρατούμε τους κόμβους της λίστας σε μια δομή.

Σε αντίθεση με μια συστοιχία όπου απαιτούνται $4N$ ψηφιοσυλλαβές για την αποθήκευση N αντικειμένων, μια συνδεδεμένη λίστα μονής σύνδεσης απαιτεί $8N$ ψηφιοσυλλαβές. Αυτό είναι ένα μέρος του κόστους που πρέπει να πληρώσουμε για την ευελιξία που προσφέρει η συνδεδεμένη οργάνωση. Πριν προχωρήσουμε στη χρήση

συνδεδεμένων λιστών για την υλοποίηση αφηρημένων δομών δεδομένων θα πρέπει να εξοικειωθούμε κάπως με το χειρισμό κόμβων. Οι υπόλοιπες παράγραφοι της ενότητας αυτής παρουσιάζουν τις βασικές αρχές χειρισμού λιστών μονής σύνδεσης.



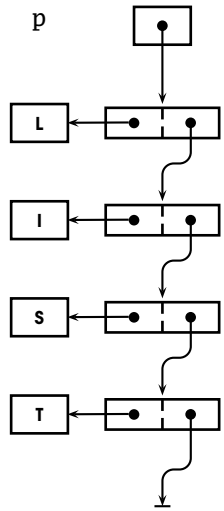
Σχήμα 4-1: Λίστα μονής σύνδεσης. Κάθε κόμβος είναι ένα αντικείμενο που αποτελείται από μια αναφορά στο αντικείμενο το οποίο ο κόμβος αυτός αποθηκεύει, και μια αναφορά στον επόμενο κόμβο της λίστας.

Κατασκευή. Ο ορισμός της κλάσης `SingleLinkNode` προκύπτει με φυσικό τρόπο από τον Ορισμό 4-1. Είναι ίσως ασυνήθιστο, προς το παρόν τουλάχιστον, να συναντάμε κλάσεις αντικειμένων με πεδία που είναι αναφορές σε αντικείμενα της ίδιας κλάσης. Αυτού τους είδους οι δομές ονομάζονται συχνά *αυτοαναφορικές* και οι συνδεδεμένες λίστες αποτελούν τυπικό παράδειγμά τους.

```
class SingleLinkNode {
   public Object element;
   public SingleLinkNode next;
   public SingleLinkNode(Object element, SingleLinkNode next) {
      this.element = element;
      this.next = next;
   }
   public void clear() {
      element = null;
      next = null;
   }
}
```

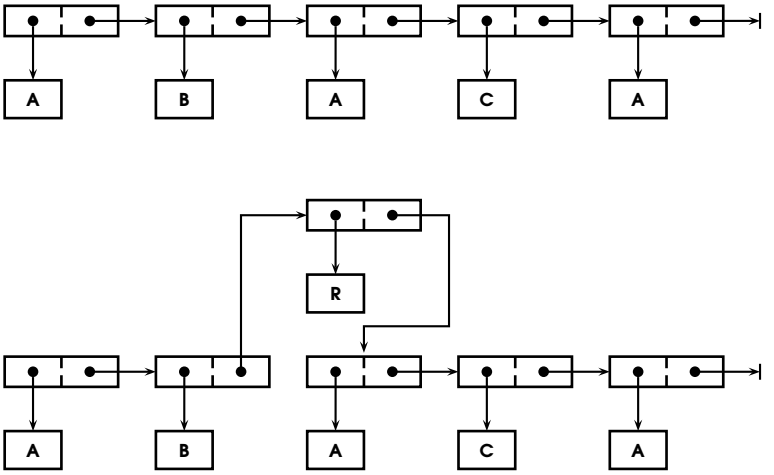
Έχοντας στη διάθεσή μας την κλάση `SingleLinkNode`, μπορούμε να κατασκευάσουμε μερικούς κόμβους και τους συνδέσουμε μεταξύ τους σε μια λίστα. Το παρακάτω απόσπασμα κώδικα για παράδειγμα, κατασκευάζει τη συνδεδεμένη λίστα του Σχήματος 4-2.

```
static SingleLinkNode createList() {
   SingleLinkNode p = new SingleLinkNode(new String("L"),
      new SingleLinkNode(new String("I"),
         new SingleLinkNode(new String("S"),
            new SingleLinkNode(new String("T"), null)
         )
      )
   );
   return p;
}
```



Σχήμα 4-2: Κατασκευή λίστας μονής σύνδεσης.

Η λίστα οριοθετείται από τον πρώτο και τον τελευταίο κόμβο της. Μια σύμβαση για τον καθορισμό του τέλους μιας συνδεδεμένης λίστας είναι να θεωρούμε ως τελευταίο εκείνο τον κόμβο του οποίου το πεδίο



Σχήμα 4-3: Εισαγωγή κόμβου σε συνδεδεμένη λίστα.
Στο κάτω τμήμα του σχήματος παρουσιάζεται η συνδεδεμένη λίστα που προκύπτει έπειτα από την παρεμβολή ενός νέου κόμβου μεταξύ του δεύτερου και του τρίτου κόμβου της συνδεδεμένης λίστας του επάνω τμήματος του σχήματος. Η λειτουργία εκτελείται σε σταθερό χρόνο, καθώς απαιτούνται μονάχα δύο αναθέσεις.

next έχει την τιμή null. Η σύμβαση αυτή ονομάζεται *τερματισμός με* null.

Διάσχιση. Η διάσχιση μιας αλυσίδας κόμβων είναι η πιο βασική μορφή επεξεργασίας συνδεδεμένων λιστών. Για να διασχίσουμε μια συνδεδεμένη λίστα, ξεκινάμε από μια αναφορά που μας οδηγεί στον πρώτο κόμβο της, και ακολουθούμε διαδοχικά τους συνδέσμους μεταξύ των κόμβων, όπως στο παρακάτω απόσπασμα κώδικα που τυπώνει τα αντικείμενα μιας αλυσίδας κόμβων ξεκινώντας από τον κόμβο h.

```
static void traverseList(SingleLinkNode h) {
   while (h != null) {
      System.out.println(h.element.toString());
      h = h.next;
   }
}
```

Το βασικό λειτουργικό μειονέκτημα μιας συνδεδεμένης λίστας, ανεξάρτητα από τον τρόπο σύνδεσης των κόμβων της, είναι ότι δεν μπορούμε να προσπελάσουμε ένα κόμβο χωρίς να έχουμε πρώτα προσπελάσει όλους τους κόμβους της λίστας που προηγούνται του κόμβου αυτού. Έτσι, ενώ σε μια συστοιχία ο χρόνος που απαιτείται για την προσπέλαση του αντικειμένου στη θέση i είναι $O(1)$, σε μια συνδεδεμένη λίστα, ο απαιτούμενος χρόνος είναι $O(i)$.

Εισαγωγή κόμβου. Η εισαγωγή ενός κόμβου n έπειτα από ένα κόμβο p, απαιτεί δύο αναθέσεις όπως παρουσιάζεται στο Σχήμα 4-3. Θέτουμε το πεδίο n.next ίσο με p.next και το πεδίο p.next ίσο με n.

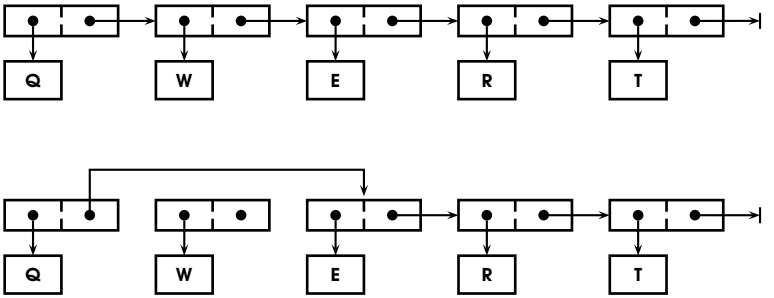
```
static void insertAfter(SingleLinkNode p, SingleLinkNode n) {
   n.next = p.next;
   p.next = n;
}
```

Η μέθοδος insertAfter μας επιτρέπει να εισάγουμε ένα κόμβο σε οποιαδήποτε θέση μιας λίστας μονής σύνδεσης εκτός από την πρώτη θέση. Σε μια τέτοια περίπτωση η υλοποίηση της εισαγωγής είναι ελαφρώς διαφορετική όπως φαίνεται στο παρακάτω απόσπασμα κώδικα. Αν p είναι ο κόμβος που εισάγεται, και h ο πρώτος κόμβος της λίστας, αρκεί να θέσουμε p.next = h. Ο πρώτος κόμβος της λίστας είναι πλέον ο p.

```
static SingleLinkNode push(SingleLinkNode p, SingleLinkNode h) {
   p.next = h;
   return p;
}
```

Διαγραφή κόμβου. Η διαγραφή ενός κόμβου από μια συνδεδεμένη λίστα είναι εξίσου απλή με την εισαγωγή. Αν p είναι ο κόμβος που πρέπει να διαγράψουμε και q ο προηγούμενος κόμβος του p, τότε αρκεί να συνδέσουμε τον q με τον κόμβο p.next και να θέτουμε p.next = null όπως παρουσιάζεται στο Σχήμα 4-4. Στην πράξη ωστόσο, η διαγραφή ενός κόμβου είναι δυσκολότερη καθώς πρέπει να εντοπίσουμε τον κόμβο q ο οποίος προηγείται του κόμβου p στη λίστα. Επιπλέον πρέπει να λάβουμε υπόψη μας την περίπτωση ο p να είναι ο πρώτος κόμβος της λίστας. Το παρακάτω απόσπασμα κώδικα παρουσιάζει την πλήρη εικόνα· η τυπική παράμετρος h είναι ο πρώτος κόμβος της λίστας, ενώ η p είναι ο κόμβος που πρέπει να διαγραφεί.

```
static SingleLinkNode removeNode(SingleLinkNode h, SingleLinkNode p) {
   SingleLinkNode q = null, n = h;
   // Locate a node q such that q.next == p
   while (n != null && n != p) {
      q = n;
      n = n.next;
```



Σχήμα 4-4: Διαγραφή κόμβου από συνδεδεμένη λίστα.
Στο κάτω τμήμα του σχήματος παρουσιάζεται η συνδεδεμένη λίστα που προκύπτει έπειτα από την διαγραφή του δεύτερου κόμβου από τη λίστα του επάνω τμήματος του σχήματος. Η διαγραφή εκτελείται σε σταθερό χρόνο, εφόσον γνωρίζουμε τον κόμβο που πρέπει να διαγραφεί και τον προηγούμενό του.

```
    }
    // p was not found; nothing to remove
    if (n == null) return h;
    // Keep an eye on the node next to p and release p
    n = p.next;
    p.clear();
    // p is the head node; return a new head node
    if (q == null) return n;
    // Link q with the former p.next and return
    q.next = n;
    return h;
}
```

Ασκήσεις

4-1 Σχεδιάστε τη λίστα μονής σύνδεσης που αποθηκεύει τα αντικείμενα **EASYYOU**.

4-2 Στη συνδεδεμένη λίστα της προηγούμενης άσκησης, παρεμβάλετε ένα κόμβο με το αντικείμενο **2** πριν από τον πρώτο κόμβο, και ένα κόμβο με το αντικείμενο **4** μετά τον τέταρτο κόμβο.

4-3 Τα αντικείμενα μιας λίστας μονής σύνδεσης είναι οι συντελεστές a_i του πολυωνύμου

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

Γράψτε μια μέθοδο που δέχεται ως παραμέτρους τον πρώτο κόμβο της λίστας (ο οποίος αποθηκεύει το συντελεστή a_0) και την τιμή της μεταβλητής x, και υπολογίζει την τιμή του πολυωνύμου.

4-4 Υλοποιήστε μια μέθοδο που δέχεται ως παραμέτρους τον πρώτο κόμβο h μιας λίστας μονής σύνδεσης και ένα αντικείμενο o και διαγράφει από τη λίστα τον πρώτο κόμβο που περιέχει μια αναφορά στο αντικείμενο o.

4-5 Υλοποιήστε μια μέθοδο που αντιστρέφει τη σειρά των κόμβων μιας συνδεδεμένης λίστας. Η παράμετρος της μεθόδου είναι μια αναφορά στον πρώτο κόμβο της λίστας.

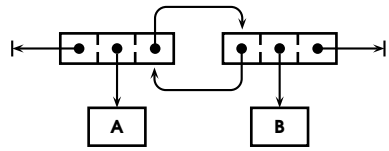
4-6 Υλοποιήστε μια μέθοδο που συνενώνει δύο λίστες μονής σύνδεσης. Η παράμετροι της μεθόδου είναι οι αναφορές στους πρώτους κόμβους των δύο λιστών.

4-7 Υλοποιήστε τη διεπαφή `Collection` χρησιμοποιώντας λίστα μονής σύνδεσης.

4.2 Διπλή σύνδεση

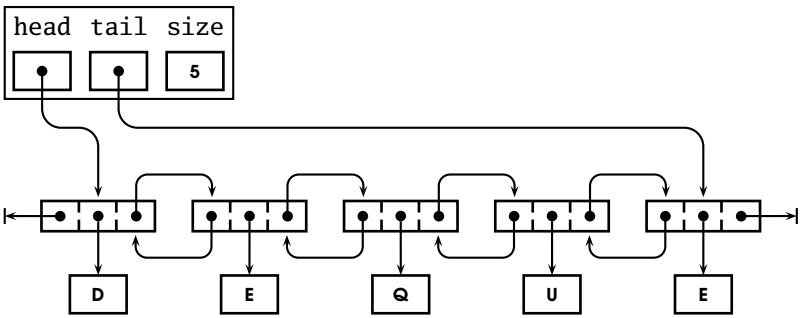
Στις περισσότερες εφαρμογές συνδεδεμένων λιστών δεν μας αρκεί να γνωρίζουμε τον επόμενο κόμβο κάθε κόμβου, αλλά και τον προηγούμενό του. Στις περιπτώσεις αυτές, μπορούμε να συμπεριλάβουμε σε κάθε κόμβο ένα επιπλέον πεδίο, `previous`, το οποίο αναφέρεται στον προηγούμενο κόμβο κάθε κόμβου. Οι συνδεδεμένες λίστες που αποτελούνται από τέτοιου είδους κόμβους, ονομάζονται *λίστες διπλής σύνδεσης*. Κατά αναλογία με την σύμβαση τερματισμού που χρησιμοποιήσαμε για λίστες μονής σύνδεσης, το πεδίο `previous` του πρώτου κόμβου μιας λίστας διπλής σύνδεσης πρέπει να είναι `null` όπως και το πεδίο `next` του τελευταίου κόμβου.

Ορισμός 4-2. Μια *λίστα διπλής σύνδεσης* είναι ένα σύνολο κόμβων που αποθηκεύουν αντικείμενα. Κάθε κόμβος είναι ένα αντικείμενο με τρία πεδία: Μια αναφορά στο αντικείμενο το οποίο ο κόμβος αποθηκεύει, μια αναφορά στον επόμενο κόμβο της λίστας και μια αναφορά στον προηγούμενο κόμβο της λίστας.



Σχήμα 4-5: Συνδεδεμένη λίστα διπλής σύνδεσης.
Κάθε κόμβος είναι ένα αντικείμενο που αποτελείται από μια αναφορά στο αντικείμενο το οποίο ο κόμβος αυτός αποθηκεύει, μία αναφορά στον προηγούμενο κόμβο, και τέλος, μια αναφορά στον επόμενο κόμβο της λίστας.

Οι λίστες διπλής σύνδεσης προσφέρουν μεγαλύτερη ευελιξία από τις λίστες μονής σύνδεσης. Τα πλεονεκτήματα αυτά δεν έρχονται φυσικά χωρίς κόστος. Οι λίστες διπλής σύνδεσης απαιτούν περισσότερη μνήμη για την αποθήκευση των συνδέσεων από ότι για την αποθήκευση των δεδομένων· μια λίστα N κόμβων απαιτεί $12N$ ψηφιοσυλλαβές για την αποθήκευσή της. Επιπλέον ο χειρισμός λιστών



Σχήμα 4-6: Μια διπλοουρά.
Το σχήμα παρουσιάζει την οργάνωση μιας διπλοουράς που υλοποιείται με λίστα διπλής σύνδεσης.

διπλής σύνδεσης είναι πιο περίπλοκος από το χειρισμό λιστών μονής σύνδεσης και απαιτεί αυξημένη προσοχή από την πλευρά του προγραμματιστή. Παρά τα μειονεκτήματά τους οι λίστες διπλής σύνδεσης έχουν πολλές εφαρμογές και στο κεφάλαιο αυτό θα παρουσιάσουμε τις πιο σημαντικές.

Οι βασικές λειτουργίες διάσχισης, εισαγωγής, διαγραφής και αναζήτησης που παρουσιάσαμε στην προηγούμενη ενότητα έχουν ελαφρώς διαφορετική υλοποίηση όταν η λίστα είναι διπλά συνδεδεμένη καθώς υπάρχουν δύο συνδέσεις ανά κόμβο αντί μιας. Θα παρουσιάσουμε τις λειτουργίες αυτές υλοποιώντας μια *διπλοουρά*.

Μια διπλοουρά είναι μια δομή δεδομένων η οποία συνδυάζει τις λειτουργίες ουράς αναμονής και στοίβας. Όπως και μια ουρά αναμονής, μια διπλοουρά έχει δύο άκρα τα οποία ονομάζονται αρχή και τέλος. Οι εισαγωγές και διαγραφές αντικειμένων μπορούν να γίνουν σε καθένα από τα δύο άκρα[1]. Η υλοποίησή που παρουσιάζεται εδώ χρησιμοποιεί μια λίστα διπλής σύνδεσης. Οι κόμβοι της λίστας είναι αντικείμενα της κλάσης `ListNode` και περιλαμβάνουν πεδία για την αποθήκευση τριών αναφορών: μία για το αντικείμενο και δύο για τους γειτονικούς κόμβους.

Κώδικας 4.1: Η κλάση `ListNode`.

```
1 package aueb.util.imp;
2 /**
3  * A list is a sequence of connected nodes that store objects. A list
```

[1] Ο όρος *διπλοουρά* αποτελεί απόδοση στα ελληνικά του όρου deque (ντέκ) η οποία προέρχεται από τη σύντμηση των λέξεων double-ended queue. Ο όρος δεν έχει ερμηνευτεί κυριολεκτικά —κάθε ουρά έχει δύο άκρα όπως έχουμε δει στο Κεφάλαιο 2— αλλά με βάση τη δυνατότητα της δομής να εκτελεί εισαγωγές και διαγραφές και στα δύο άκρα της.

```
4  * node consists of three references; one points to the stored object,
5  * one that points to the next node in the list, and one that points
6  * to the previous node in the list.
7  */
8  class ListNode {
9     /**
10     * Reference to stored object.
11     */
12    public Object element;
13    /**
14     * Reference to previous node.
15     */
16    public ListNode previous;
17    /**
18     * Reference to next node.
19     */
20    public ListNode next;
21    /**
22     * Creates a new node that stores an object.
23     * @param object The object to store.
24     */
25    public ListNode(Object object) {
26       element = object;
27    }
28    /**
29     * Clears all fields of this node.
30     */
31    public void clear() {
32       element = null;
33       previous = next = null;
34    }
35 }
```

Κάθε αντικείμενο τύπου Deque έχει τρία πεδία. Το πεδίο size μετρά το πλήθος των αντικειμένων που βρίσκονται στην δομή· αυξάνεται κατά 1 σε κάθε εισαγωγή και μειώνεται με κάθε εξαγωγή. Τα πεδία head και tail είναι αναφορές στον πρώτο και τελευταίο κόμβο της διπλοουράς αντίστοιχα. Όταν μια διπλοουρά είναι κενή, τα δύο αυτά πεδία έχουν την τιμή null. Όταν η διπλοουρά έχει ένα μοναδικό αντικείμενο, τα δύο αυτά πεδία αναφέρονται στον μοναδικό κόμβο της διπλοουράς. Σε κάθε άλλη περίπτωση, τα πεδία έχουν διαφορετική τιμή. Η οργάνωση μιας διπλοουράς 5 στοιχείων που υλοποιείται με λίστα διπλής σύνδεσης, παρουσιάζεται στο Σχήμα 4-6, ενώ η υλοποίηση της κλάσης Deque δίνεται στον Κώδικα 4.2.

Κώδικας 4.2: Η κλάση Deque.

```
1  package aueb.util.imp;
2  /**
3   * A deque is a data structure with two ends called its head
4   * and tail. A deque supports object insertions and removals
5   * from its both ends.
6   * This is an implementation of a deque that relies on doubly linked lists.
7   */
8  public class Deque {
9     /**
10     * The head of the deque.
11     */
12    private ListNode head;
13    /**
14     * The tail of the deque.
15     */
16    private ListNode tail;
17    /**
18     * The number of objects in the deque.
19     */
20    private int size;
21    /**
22     * Default constructor. Creates an emptpy deque.
23     */
24    public Deque() {
25       super();
26       head = tail = null;
27       size = 0;
28    }
29    /**
30     * Inserts an object before the head of this instance.
31     * @param element The object to insert.
32     */
33    public void pushHead(Object element) {
34       if (element == null) throw new IllegalArgumentException();
35       // Create a new node to store element.
36       ListNode p = new ListNode(element);
37       // head is not null; link p before head.
38       if (head != null) {
39          p.next = head;
40          head.previous = p;
41       }
42       // head == tail == null; update tail
43       else {
44          tail = p;
45       }
```

```
46        // Update head
47        head = p;
48        ++size;
49     }
50     /**
51      * Inserts an object at the tail of this instance.
52      * @param element The object to insert.
53      */
54     public void pushTail(Object element) {
55        if (element == null) throw new IllegalArgumentException();
56        // Create a new node to store element.
57        ListNode p = new ListNode(element);
58        // tail is not null; link p after tail.
59        if (tail != null) {
60           p.previous = tail;
61           tail.next = p;
62        }
63        // tail == head == null; update head
64        else {
65           head = p;
66        }
67        // Update tail
68        tail = p;
69        ++size;
70     }
71     /**
72      * Removes the element at the head of the queue.
73      * @return The element at the head of the queue.
74      * @throws IllegalStateException if the queue is empty.
75      */
76     public Object popHead() {
77        // Ensure not empty.
78        if (size == 0) throw new IllegalStateException();
79        // The node pointed to by head is being removed; keep an eye on it.
80        ListNode p = head;
81        Object object = p.element;
82        // Set head to point to the node next to p.
83        head = p.next;
84        if (head != null) {
85           head.previous = null;
86        }
87        // head became null; so must tail.
88        else {
89           tail = null;
90        }
91        // Clear p's fields, update size and return.
```

```
92          p.clear();
93          --size;
94          return object;
95       }
96       /**
97        * Removes the element at the tail of the queue.
98        * @return The element at the tail of the queue.
99        * @throws IllegalStateException if the queue is empty.
100       */
101      public Object popTail() {
102         // Ensure not empty.
103         if (size == 0) throw new IllegalStateException();
104         // The node pointed to by tail is being removed; keep an eye on it.
105         ListNode p = tail;
106         Object object = p.element;
107         // Set tail to point to the node before p.
108         tail = p.previous;
109         if (tail != null) {
110            tail.next = null;
111         }
112         // tail became null; so must head.
113         else {
114            head = null;
115         }
116         // Clear p's fields, update size and return.
117         p.clear();
118         --size;
119         return object;
120      }
121      /**
122       * Returns tthe element at the head of the queue.
123       * @return The element at the head of the queue.
124       * @throws IllegalStateException if the queue is empty.
125       */
126      public Object head() {
127         // Ensure not empty.
128         if (size == 0) throw new IllegalStateException ();
129         // Return a reference to the object stored at head.
130         return head.element;
131      }
132
133      /**
134       * Returns the element at the tail of the queue.
135       * @return The element at the tail of the queue.
136       * @throws IllegalStateException if the queue is empty.
137       */
```

```
138     public Object tail() {
139        // Ensure not empty.
140        if (size == 0) throw new IllegalStateException();
141        // Return a reference to the object stored at tail.
142        return tail.element;
143     }
144     /**
145      * Returns the size of this instance.
146      * @return The number of objects stored in this instance.
147      */
148     public int size() {
149        return size;
150     }
151     /**
152      * Tests if this instance is empty.
153      * @return true if this instance is empty.
154      */
155     public boolean isEmpty() {
156        return size == 0;
157     }
158 }
```

Όλες οι μέθοδοι της κλάσης Deque εκτελούνται σε σταθερό χρόνο, όπως και οι αντίστοιχες μέθοδοι των κλάσεων ArrayStack και ArrayQueue. Η κλάση Deque απαιτεί περισσότερη μνήμη για την αποθήκευση των αντικειμένων, λόγω των δύο συνδέσμων που διατηρεί κάθε κόμβος, αλλά σε αντιστάθμισμα αυτού του μειονεκτήματος δεν απαιτείται να είναι γνωστή εκ των προτέρων η μέγιστη χωρητικότητα της δομής. Αυτή η ιδιότητα της κλάσης Deque θα αποδειχθεί πολύ χρήσιμη στην υλοποίηση επαναληπτικών αλγορίθμων διάσχισης δυαδικών δέντρων στο Κεφάλαιο 5.

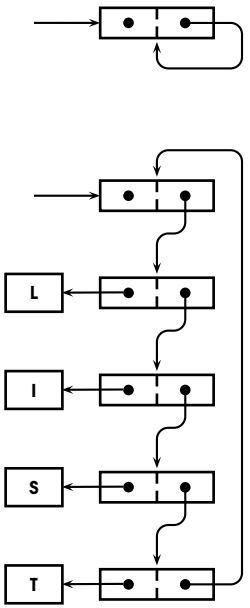
head()	$O(1)$
tail()	$O(1)$
pushHead(Object)	$O(1)$
pushTail(Object)	$O(1)$
popHead()	$O(1)$
popTail()	$O(1)$
size()	$O(1)$

Πίνακας 4-1: Αποτελεσματικότητα της κλάσης Deque.
Ο πίνακας παρουσιάζει το χρόνο εκτέλεσης των μεθόδων της κλάσης Deque· όλες οι μέθοδοι εκτελούνται σε σταθερό χρόνο ανεξάρτητα από το πλήθος των αποθηκευμένων αντικειμένων.

Ασκήσεις

4-8 Σχεδιάστε τη λίστα διπλής σύνδεσης που αποθηκεύει τα αντικείμενα **2 E A S Y 4 Y O U**.

4-9 Σχεδιάστε την δομή διπλοουράς που προκύπτει από την εκτέλεση των λειτουργιών **+E +A S+ $ +Y Q+ * +U $ E+ +S $ $ T+ * I+ +O $ * * N+ $**. Τα γράμματα συμβολίζουν αντικείμενα που εισάγονται στην αρχή της διπλοουράς, αν έπονται του συμβόλου **+** ή στο τέλος της αν προηγούνται του συμβόλου **+**. Το σύμβολο ***** υποδεικνύει εξαγωγή από την αρχή, ενώ το σύμβολο **$** εξαγωγή από το τέλος.



Σχήμα 4-7: Κυκλική λίστα μονής σύνδεσης.
Μια κενή κυκλική λίστα μονής σύνδεσης (επάνω) περιλαμβάνει μόνο τον κόμβο-κεφαλή. Το πεδίο next της κεφαλής αναφέρεται στην ίδια την κεφαλή.
Όταν μια κυκλική λίστα μονής σύνδεσης έχει ένα ή περισσότερους κόμβους (κάτω), ο πρώτος κόμβος είναι εκείνος στον οποίο αναφέρεται το πεδίο next της κεφαλής, ενώ ο τελευταίος κόμβος είναι εκείνος ο κόμβος του οποίου το πεδίο next αναφέρεται στην κεφαλή.

4-10 Υλοποιήστε τις διεπαφές `Stack` και `Queue` χρησιμοποιώντας την κλάση `Deque`.

4-11 Υλοποιήστε τη διεπαφή `Collection` χρησιμοποιώντας λίστα διπλής σύνδεσης.

4-12 Υλοποιήστε μια μέθοδο που αντιστρέφει τη σειρά των κόμβων μιας λίστας διπλής σύνδεσης. Η παράμετρος της μεθόδου είναι μια αναφορά στον πρώτο κόμβο της λίστας.

4.3 Κυκλική σύνδεση

Η σύμβαση ότι μια λίστα τερματίζεται από ένα σύνδεσμο με τιμή `null`, είναι αρκετά συνηθισμένη σε πολλές εφαρμογές, όπως για παράδειγμα στους πίνακες κατακερματισμού με αλυσίδωση, ένας μηχανισμός οργάνωσης δεδομένων που παρουσιάζεται στο Κεφάλαιο 8. Μια άλλη, εξίσου αν όχι περισσότερο δημοφιλής, προσέγγιση είναι η εισαγωγή ενός ειδικού κόμβου ο οποίος ονομάζεται *κεφαλή της λίστας*. Η κεφαλή της λίστας είναι ένας κόμβος ο οποίος δεν χρησιμοποιείται για την αποθήκευση δεδομένων, αλλά για να σηματοδοτεί την αρχή και ταυτόχρονα το τέλος της λίστας.

Σε μια λίστα μονής σύνδεσης, η κεφαλή της λίστας προηγείται του πρώτου κόμβου της λίστας, ενώ ο τελευταίος κόμβος της λίστας προηγείται της κεφαλής της λίστας. Όταν η λίστα είναι κενή, τότε κατά σύμβαση το πεδίο `next` της κεφαλής αναφέρεται στην ίδια την κεφαλή. Έτσι, αν `head` είναι η κεφαλή μιας κυκλικής λίστας μονής σύνδεσης:

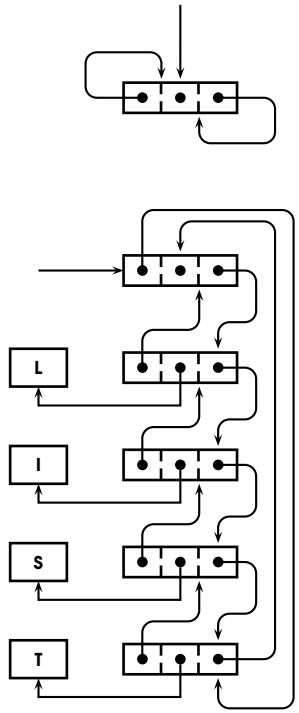
- Ο πρώτος κόμβος της λίστας είναι ο κόμβος `head.next`.
- Ο τελευταίος κόμβος της λίστας είναι εκείνος ο κόμβος `q` για τον οποίο `q.next == head`.
- Όταν η λίστα είναι κενή, τότε `head.next == head`.

Αυτή η τεχνική σύνδεσης παρουσιάζεται στο Σχήμα 4-7. Η συνδεσμολογία μπορεί, με ανάλογο τρόπο, να επεκταθεί και σε λίστες διπλής σύνδεσης όπως φαίνεται στο Σχήμα 4-8. Έτσι αν `head` είναι η κεφαλή μιας κυκλικής λίστας διπλής σύνδεσης:

- Ο πρώτος, έστω `p`, κόμβος της λίστας είναι ο κόμβος `head.next` και `p.previous == head`.
- Ο τελευταίος, έστω `q`, κόμβος της λίστας είναι ο κόμβος `head.previous` και `q.next == head`.

- Όταν η λίστα είναι κενή, τότε `head.next == head` και `head.previous == head`.

Το βασικό πλεονέκτημα που προσφέρει η κυκλική σύνδεση των κόμβων μιας λίστας, είναι ότι τα πεδία `next` και `previous` των κόμβων της λίστας (συμπεριλαμβανομένης και της κεφαλής) δεν είναι ποτέ `null`, ακόμη και αν η λίστα είναι κενή. Έτσι οι περισσότερες λειτουργίες απλοποιούνται σημαντικά. Στην ουσία η κυκλική σύνδεση ομογενοποιεί τη συνδεσμολογία —κάθε κόμβος έχει προηγούμενο και επόμενο κόμβο— με αποτέλεσμα να μην είναι ανάγκη να χειριζόμαστε με διαφορετικό τρόπο τις εισαγωγές στην αρχή ή στο τέλος της λίστας (όπως στην κλάση `Deque` που παρουσιάσαμε στη σελίδα 125). Ο μηχανισμός είναι ιδανικός για την υλοποίηση ακολουθιών. Μια τέτοια υλοποίηση είναι η κλάση `ListSequence` η οποία δίνεται στον Κώδικα 4.3. Η αποτελεσματικότητα των μεθόδων δίνεται στον Πίνακα 4-2.



Σχήμα 4-8: Κυκλική λίστα μονής σύνδεσης.
Μια κενή κυκλική λίστα μονής σύνδεσης (επάνω) περιλαμβάνει μόνο τον κόμβο-κεφαλή. Το πεδίο `next` της κεφαλής αναφέρεται στην ίδια την κεφαλή.
Όταν μια κυκλική λίστα μονής σύνδεσης έχει ένα ή περισσότερους κόμβους (κάτω), ο πρώτος κόμβος είναι εκείνος στον οποίο αναφέρεται το πεδίο `next` της κεφαλής, ενώ ο τελευταίος κόμβος είναι εκείνος ο κόμβος του οποίου το πεδίο `next` αναφέρεται στην κεφαλή.

Κώδικας 4.3: Η κλάση `ListSequence`.

```
1  package aueb.util.imp;
2
3  import aueb.util.api.AbstractCollection;
4  import aueb.util.api.Enumerator;
5  import aueb.util.api.Iterator;
6  import aueb.util.api.Sequence;
7
8  /**
9   * Implementation of sequence using double-link circular list.
10  */
11 class ListSequence extends AbstractCollection implements Sequence{
12     /**
13      * Double-link circular list iterator.
14      */
15     private class ListIterator implements Iterator {
16         /**
17          * The position of this iterator. A call to nextIndex()
18          * will return position, while a call to previousIndex()
19          * will return position-1.
20          */
21         private int position;
22         /**
23          * As with position, a call to next will return node,
24          * while a call to previous will return node.previous.
25          */
26         private ListNode node;
```

```
27          /**
28           * The last node accessed by the iterator.
29           */
30          private ListNode lastNode;
31          /**
32           * Creates an iterator on the enclosing list positioned
33           * at a given position.
34           * @param index
35           */
36          public ListIterator(int i) {
37             if (i < 0 || i > size) {
38                throw new IndexOutOfBoundsException();
39             }
40             node = head.next;
41             position = 0;
42             for(; position < i; ++position) node = node.next;
43          }
44          public boolean hasNext() {
45             return position != size;
46          }
47          public boolean hasPrevious() {
48             return position != 0;
49          }
50          public int nextIndex() {
51             return position;
52          }
53          public int previousIndex() {
54             return position - 1;
55          }
56          public Object next() {
57             if (position == size) throw new IllegalStateException();
58             lastNode = node;
59             node = node.next;
60             ++position;
61             return lastNode.element;
62          }
63          public Object previous() {
64             if (position == 0) throw new IllegalStateException();
65             lastNode = node.previous;
66             node = node.previous;
67             --position;
68             return lastNode.element;
69          }
70          public void add(Object object) {
71             link(node.previous, object);
72             ++position;
```

```
73        }
74        public void remove() {
75           // Check position
76           if (lastNode == null) throw new IllegalStateException();
77           // The object being removed is on the right
78           if (lastNode == node) {
79              node = node.next;
80           }
81           // The object being removed is on the left; update position
82           else {
83              --position;
84           }
85           // Remove the node
86           ListSequence.this.disconnect(lastNode);
87           lastNode = null;
88        }
89        public void set(Object object) {
90           if (lastNode == null) throw new IllegalStateException();
91           lastNode.element = object;
92        }
93     }
94     /**
95      * The head node.
96      */
97     protected ListNode head;
98     /**
99      * The number of objects in the sequence.
100     */
101    protected int size;
102    /**
103     * Default constructor. Creates an empty list.
104     */
105    public ListSequence() {
106       head = new ListNode(null);
107       head.next = head.previous = head;
108       size = 0;
109    }
110    public int size() {
111       return size;
112    }
113    /**
114     * Overrides the slow implementation inherited.
115     */
116    public void clear() {
117       head.next = head.previous = head;
118       size = 0;
```

```
119     }
120     public boolean add(Object object) {
121        add(size, object);
122        return true;
123     }
124     public void add(int i, Object object) {
125        // Disallow null objects from being inserted
126        if (object == null) throw new IllegalArgumentException();
127        // Ensure index is valid
128        if (i < 0 || i > size) {
129           throw new IndexOutOfBoundsException();
130        }
131        // Locate the index-th node
132        ListNode p = null;
133        if (i == 0) {
134           p = head;
135        }
136        else if (i == size) {
137           p = head.previous;
138        }
139        else {
140           p = getNode(i).previous;
141        }
142        // Link a new node right after p
143        link(p, object);
144     }
145     public Object remove(int i) {
146        if (i < 0 || i >= size) throw new IndexOutOfBoundsException();
147        ListNode n = getNode(i);
148        Object element = n.element;
149        disconnect(n);
150        return element;
151     }
152     public boolean remove(Object object) {
153        ListNode n = getNode(object);
154        if (n == null) return false;
155        disconnect(n);
156        return true;
157     }
158     public Object get(int i) {
159        if (i < 0 || i >= size) throw new IndexOutOfBoundsException();
160        return getNode(i).element;
161     }
162     public Object set(int i, Object object) {
163        if (i < 0 || i > size) throw new IndexOutOfBoundsException();
164        ListNode p = getNode(i);
```

```
165         Object temp = p.element;
166         p.element = object;
167         return temp;
168      }
169      public int indexOf(Object object) {
170         int i = 0;
171         for (ListNode p = head.next; p != head; p = p.next, ++i) {
172            if (p.element.equals(object)) return i;
173         }
174         return -1;
175      }
176      public boolean contains(Object object) {
177         return indexOf(object) != -1;
178      }
179      public Iterator iterator() {
180         return new ListIterator(0);
181      }
182      public Enumerator enumerator() {
183         return iterator();
184      }
185      public Iterator iterator(int i) {
186         return new ListIterator(i);
187      }
188      /**
189       * Locates a node based on its position.
190       * @param i The position of the node.
191       * @return The i-th node of the list.
192       */
193      protected ListNode getNode(int i) {
194         ListNode p = head;
195         for (int j = 0; j <= i; ++j) p = p.next;
196         return p;
197      }
198      /**
199       * Locates the leftmost node that contains an object.
200       * @param object The object to locate.
201       * @return The eftmost node that contains object.
202       */
203      protected ListNode getNode(Object object) {
204         for (ListNode p = head.next; p != head; p = p.next) {
205            if (p.element.equals(object)) return p;
206         }
207         return null;
208      }
209      /**
210       * Inserts a node right after of a given node.
```

	$W(N)$	$A(N)$
`add(Object)`	$O(N)$	$O(N)$
`add(int,Object)`	$O(N)$	$O(N)$
`remove(Object)`	$O(N)$	$O(N)$
`remove(int)`	$O(N)$	$O(N)$
`contains()`	$O(N)$	$O(N)$
`indexOf()`	$O(N)$	$O(N)$
`get(int)`	$O(N)$	$O(N)$
`set(int,Object)`	$O(1)$	$O(1)$
`size()`	$O(1)$	$O(1)$
`isEmpty()`	$O(1)$	$O(1)$
`clear()`	$O(1)$	$O(1)$

Πίνακας 4-2: Αποτελεσματικότητα των μεθόδων της κλάσης `ListSequence`.

```
211      * @param node The node.
212      * @param object The object to store in the new node.
213      * @return The node inserted.
214      */
215     protected ListNode link(ListNode node, Object object) {
216        ListNode newNode = new ListNode(object);
217        newNode.previous = node;
218        newNode.next = node.next;
219        node.next.previous = newNode;
220        node.next = newNode;
221        ++size;
222        return newNode;
223     }
224     /**
225      * Disconnects a node from the list.
226      * @param node The node to disconnect.
227      */
228     protected void disconnect(ListNode node) {
229        node.previous.next = node.next;
230        node.next.previous = node.previous;
231        node.clear();
232        --size;
233     }
234  }
```

Ασκήσεις

4-13 Υλοποιείστε χρησιμοποιώντας τον αλγόριθμο ταξινόμησης με εισαγωγή, τη μέθοδο `void sort()` της κλάσης `ListSequence` η οποία θα ταξινομεί τα στοιχεία της ακολουθίας σε αύξουσα διάταξη.

4-14 Επαναλάβετε την προηγούμενη άσκηση χρησιμοποιώντας τον αλγόριθμο bubble sort.

4-15 Μπορούμε να βελτιώσουμε ελαφρώς την αποτελεσματικότητα της μεθόδου `getNode` της κλάσης `ListSequence` αν αντί να ξεκινάμε τη διάσχιση από το αριστερό άκρο διασχίζοντας τη λίστα μέχρι να εντοπίσουμε την επιθυμητή θέση, ξεκινάμε από το άκρο που είναι πιο κοντά στην επιθυμητή θέση. Συγκεκριμένα θα ξεκινάμε από τον πρώτο κόμβο αν $i < N/2$ όπου i είναι η θέση του κόμβου που θέλουμε να προσπελάσουμε και N είναι το πλήθος των κόμβων στη λίστα, ενώ διαφορετικά θα ξεκινάμε από τον τελευταίο κόμβο και θα

διασχίζουμε προς τα πίσω. Ακυρώστε τη μέθοδο `getNode(int)` στον Κώδικας 4.3, ώστε να λειτουργεί με αυτή τη λογική.

4-16 Υλοποιήστε τη μέθοδο `void reverse()` της κλάσης `ListSequence` η οποία αντιστρέφει την ακολουθία.

4-17 Υλοποιήστε τη μέθοδο `void shuffle()` της κλάσης `ListSequence` η οποία μεταθέτει τυχαία τα στοιχεία της ακολουθίας.

4-18 Υλοποιήστε τη μέθοδο `void addAll(Sequence)` της κλάσης `ListSequence`.

4-19 Δώστε μια υλοποίηση της μεθόδου `addAll(Collection c)` για την κλάση `ListSequence` η οποία βελτιώνει την αποτελεσματικότητα της υλοποίησης που παρέχει η κλάση `AbstractCollection`.

ΚΕΦΑΛΑΙΟ 5

Δέντρα

5.1 Εισαγωγή

Ένα *δέντρο* είναι μια ιεραρχική οργάνωση αυθαίρετων αντικειμένων που ονομάζονται *κόμβοι*. Οι ιεραρχικές συσχετίσεις των αντικειμένων αυτών καθορίζονται από τις μεταξύ τους συνδέσεις οι οποίες ονομάζονται *ακμές*. Τέτοιες ιεραρχικές οργανώσεις μας είναι πολύ οικείες: οι γενεαλογίες, τα οργανογράμματα επιχειρήσεων, τα συστήματα αρχείων κτλ., παριστάνονται με δέντρα, τα οποία με τη σειρά τους αποτελούν ειδίκευση των δομών που ονομάζονται *γράφοι* ή *δίκτυα*.

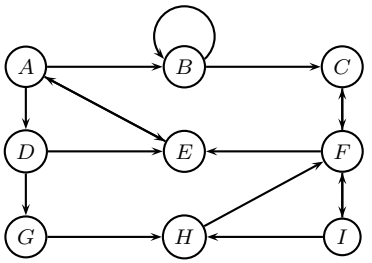
Ορισμός 5-1. Ένας γράφος $G = (V, E)$ αποτελείται από ένα σύνολο κόμβων $V = \{v_1, v_2, \ldots v_N\}$ και ένα σύνολο ακμών $E \subseteq V \times V$. Οι κόμβοι του γράφου χρησιμοποιούνται για την αποθήκευση πληροφοριών όπως ονόματα πόλεων, ανθρώπων κτλ., και οι ακμές του για να συνδέουν ζεύγη κόμβων μεταξύ τους. Λέμε ότι ο κόμβος v συνδέεται άμεσα με τον κόμβο w αν υπάρχει μια $e = (v, w)$ στο σύνολο ακμών του γράφου.

Το Σχήμα 5-1, παριστάνει ένα γράφο με σύνολο κόμβων

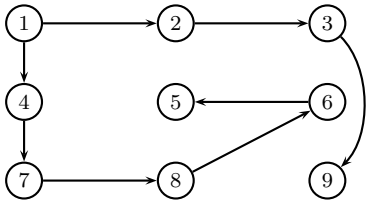
$$V = \{A, B, C, D, E, F, G, H, I\}$$

και σύνολο ακμών

$$E = \{(A, B), (B, C), (C, I), (A, D), (D, G), (F, E), (G, H), (H, F)\}.$$



Σχήμα 5-1: Γραφική αναπαράσταση δικτύου.
Οι κόμβοι παριστάνονται με κύκλους ενώ οι ακμές με βέλη.



Σχήμα 5-2: Γραφική παράσταση δέντρου.
Ένα δέντρο είναι ένας γράφος στον οποίο κάθε ζεύγος κόμβων συνδέεται από ένα ακριβώς μονοπάτι.

Αν υποθέσουμε πως ο γράφος αυτός παριστάνει το οδικό δίκτυο που συνδέει ένα σύνολο πόλεων, ξεκινώντας από ένα κόμβο (πόλη) και ακολουθώντας τις ακμές (οδούς) θα επισκεφτούμε ένα σύνολο πόλεων. Ξεκινώντας για παράδειγμα από την πόλη A μπορούμε να φτάσουμε στις πόλεις D, G, H, F, E. Μια τέτοια διαδρομή ονομάζεται μονοπάτι.

Ορισμός 5-2. Ένα *μονοπάτι* είναι μια ακολουθία κόμβων $v_1, v_2, \ldots, v_k$ στην οποία κάθε κόμβος συνδέεται με τον επόμενό του· ισχύει δηλαδή ότι $(v_i, v_{i+1}) \in E$, $1 \leq i \leq k-1$.

Ορισμός 5-3. Ένας γράφος $T = (V, E)$ είναι δέντρο αν υπάρχει *ακριβώς ένα* μονοπάτι μεταξύ κάθε ζεύγους κόμβων $(v, w) \in V$.

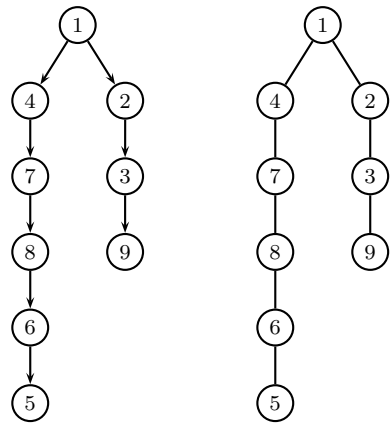
Ένα δέντρο με ρίζα (rooted tree) είναι ένα δέντρο στο οποίο υπάρχει ένας κόμβος στον οποίο δεν οδηγεί κανένα μονοπατι. Στα επόμενα όταν θα λέμε δέντρο θα εννοούμε δέντρο με ρίζα. Το δέντρο του Σχήματος 5-2 για παράδειγμα, έχει ρίζα τον κόμβο 1.

Αν σε ένα δέντρο υπάρχει μια ακμή (v, w), θα λέμε πως κόμβος v είναι *πατέρας* του w και πως ο w είναι *παιδί* του v. Στο δέντρο του Σχήματος 5-2 για παράδειγμα, ο κόμβος 7 είναι παιδί του κόμβου 4. Ένας κόμβος μπορεί να έχει μηδέν ή περισσότερα παιδιά και ακριβώς ένα πατέρα, εκτός φυσικά και πρόκειται για την ρίζα, η οποία δεν επιτρέπεται να έχει πατέρα. Ένας κόμβος χωρίς παιδιά, ονομάζεται *φύλλο*. Οι κόμβοι v οι οποίοι προηγούνται ενός κόμβου w στο μονοπάτι από τη ρίζα ως τον w, ονομάζονται *πρόγονοι* του κόμβου w. Παρόμοια, όλοι οι κόμβοι w που βρίσκονται σε κάποιο μονοπάτι που ξεκινά από τον κόμβο v ονομάζονται *απόγονοι* του κόμβου v. Αν σχεδιάσουμε το δέντρο του Σχήματος 5-2 με λίγο διαφορετικό τρόπο, όπως για παράδειγμα στο Σχήμα 5-3, τότε οι παραπάνω σχέσεις μεταξύ κόμβων είναι πολύ πιο ευδιάκριτες, χωρίς να χρειάζεται να χρησιμοποιούμε βέλη για δηλώσουμε την κατεύθυνση από τον πατέρα προς τα παιδιά. Η ρίζα του δέντρου βρίσκεται πάντα στην κορυφή, ενώ τα παιδιά ενός κόμβου βρίσκονται ακριβώς κάτω από αυτόν.

Κάθε κόμβος μαζί με τους απογόνους του, σχηματίζουν ένα δέντρο το οποίο θα ονομάζουμε *υπόδεντρο*.

Ένα *διατεταγμένο δέντρο* (ordered tree) είναι ένα δέντρο στο οποίο η διάταξη των παιδιών κάθε κόμβου καθορίζεται ρητά. Αν

επιπλέον απαιτήσουμε κάθε κόμβος να έχει ακριβώς M παιδιά, τότε έχουμε ένα M-δικό δέντρο. Φυσικά, για πρακτικούς λόγους επιτρέπουμε οι κόμβοι ενός M-δικού δέντρου να έχουν λιγότερα από M παιδιά. Έτσι, εισάγουμε τεχνητούς κόμβους που ονομάζονται *εξωτερικοί κόμβοι*, έτσι ώστε να ικανοποιείται η συνθήκη ορισμού. Με βάση αυτή τη σύμβαση, όλοι οι κόμβοι που δεν είναι εξωτερικοί ονομάζονται *εσωτερικοί κόμβοι* και πλέον τα φύλλα είναι εκείνοι οι εσωτερικοί κόμβοι που όλα τα παιδιά τους είναι εξωτερικοί κόμβοι. Η πιο απλή αλλά και πιο δημοφιλής περίπτωση M-δικών δέντρων είναι τα δυαδικά δέντρα.



Σχήμα 5-3: Το δέντρο του Σχήματος 5-2, σχεδιασμένο ώστε να είναι προφανής η σχέση πατέρα-παιδιού.
Στα αριστερά η κατεύθυνση των ακμών καθορίζεται ρητά, ενώ στα δεξιά υπονοείται κατεύθυνση από πάνω προς τα κάτω. Κάθε κόμβος w εκτός της ρίζας έχει ακριβώς ένα κόμβο v πάνω απ' αυτόν έτσι ώστε $(v, w) \in E$.

Ορισμός 5-4. Ένα δυαδικό δέντρο είναι είτε ένας εξωτερικός κόμβος, είτε ένας εσωτερικός κόμβος με δύο παιδιά που είναι δυαδικά δέντρα και ονομάζονται αριστερό και δεξί δέντρο.

Ο Ορισμός 5-4 είναι αναδρομικός. Το απλούστερο δυαδικό δέντρο είναι το κενό δυαδικό δέντρο, που έχει ρίζα ένα εξωτερικό κόμβο. Κάθε κόμβος που έχει δύο παιδιά που είναι, κενά ή όχι, δυαδικά δέντρα, αποτελεί ένα δυαδικό δέντρο.

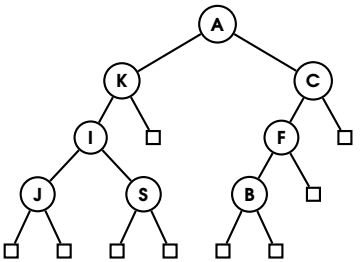
Θεώρημα 5-1. *Ένα δυαδικό δέντρο με N εσωτερικούς κόμβους έχει $N + 1$ εξωτερικούς κόμβους.*

Απόδειξη. Θα αποδείξουμε την ιδιότητα αυτή με επαγωγή στο πλήθος των κόμβων ενός δυαδικού δέντρου.

Ένα δυαδικό δέντρο ενός εσωτερικού κόμβου έχει δύο εξωτερικούς κόμβους (το αριστερό και το δεξί υπόδεντρο). Επομένως το Θεώρημα 5-1 ισχύει για $N = 1$.

Ας υποθέσουμε πως ένα δυαδικό δέντρο με $k \geq 1$ εσωτερικούς κόμβους έχει $k + 1$ εξωτερικούς κόμβους. Χρησιμοποιώντας την επαγωγική υπόθεση θα υπολογίσουμε το πλήθος των εξωτερικών κόμβων ενός δυαδικού δέντρου με $k + 1$ εσωτερικούς κόμβους.

Δεδομένου ότι η ρίζα είναι εσωτερικός κόμβος, ένα δυαδικό δέντρο με $k + 1$ εσωτερικούς κόμβους θα έχει m εσωτερικούς κόμβους στο αριστερό του υπόδεντρο, και $k - m$ στο δεξί, για κάποιο $m \in [0, k]$. Με βάση την επαγωγική υπόθεση, το αριστερό υπόδεντρο έχει $m+1$ εξωτερικούς κόμβους, και το δεξί $k-m+1$. Συνολικά υπάρχουν λοιπόν $k + 2$ εξωτερικοί κόμβοι. □



Σχήμα 5-4: Δυαδικό δέντρο.
Οι εσωτερικοί κόμβοι συμβολίζονται με κύκλους ενώ οι εξωτερικοί κόμβοι συμβολίζονται με τετράγωνα χωρίς περιεχόμενο. Ένας κόμβος που τα παιδιά του είναι εξωτερικοί κόμβοι είναι φύλλο.

Θεώρημα 5-2. *Ένα δυαδικό δέντρο με N εσωτερικούς κόμβους έχει $2N$ ακμές. Οι $N-1$ απ' αυτές οδηγούν σε εσωτερικούς κόμβους, και $N+1$ οδηγούν σε εξωτερικούς κόμβους.*

Απόδειξη. Η απόδειξη του θεωρήματος αυτού απορρέει φυσικά από τον Ορισμό 5-4 και από το Θεώρημα 5-1. Έχουμε N εσωτερικούς κόμβους και συνεπώς θα πρέπει να υπάρχουν ακμές προς όλους τους κόμβους εκτός από τη ρίζα. Επομένως ακριβώς $N-1$ ακμές του δέντρου οδηγούν σε εσωτερικούς κόμβους. Επειδή το δέντρο έχει $N+1$ εξωτερικούς κόμβους, θα πρέπει να υπάρχουν ακριβώς τόσες ακμές που οδηγούν σε αυτούς. Συνολικά το δέντρο έχει $2N$ ακμές. □

Ορισμός 5-5. Το *επίπεδο* ενός κόμβου δέντρου, είναι το επίπεδο του πατέρα του συν ένα. Το επίπεδο της ρίζας ενός δέντρου είναι μηδέν. Το *ύψος* ενός δέντρου είναι το μέγιστο από τα επίπεδα των κόμβων του δέντρου.

Ορισμός 5-6. Το *μήκος μονοπατιού* ενός δέντρου είναι το άθροισμα των επιπέδων των κόμβων του. Το *μήκος εσωτερικού μονοπατιού* ενός δέντρου είναι το άθροισμα των επιπέδων των εσωτερικών κόμβων του. Το *μήκος εξωτερικού μονοπατιού* ενός δέντρου είναι το άθροισμα των επιπέδων των εξωτερικών κόμβων του.

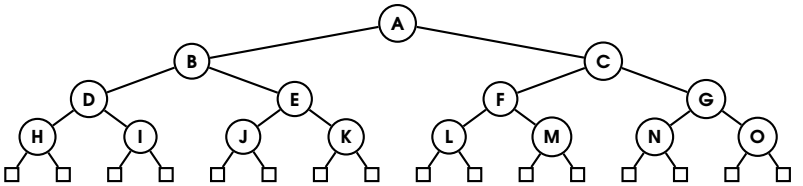
Θεώρημα 5-3. *Το μήκος εξωτερικού μονοπατιού ενός δυαδικού δέντρου με N εσωτερικούς κόμβους είναι κατά $2N$ μεγαλύτερο του μήκους εσωτερικού μονοπατιού.*

Απόδειξη. Έστω k το πλήθος των εσωτερικών κόμβων ενός δυαδικού δέντρου. Αν $k = 1$, το μήκος εσωτερικού μονοπατιού είναι $I = 0$. Επιπλέον το δέντρο έχει δύο εξωτερικούς κόμβους στο επίπεδο 1, και κατά συνέπεια το μήκος εξωτερικού μονοπατιού $E = 2$. Επομένως ισχύει $E = I + 2k$ για $k = 1$.

Ας υποθέσουμε τώρα ότι σε κάθε δέντρο με N εσωτερικούς κόμβους ισχύει ότι

$$E = I + 2k, \quad 1 \leq k \leq N. \tag{5.1}$$

Μπορούμε να μετατρέψουμε κάθε δέντρο T με N εσωτερικούς κόμβους σε ένα δέντρο με $N+1$ εσωτερικούς κόμβους, αντικαθιστώντας ένα τυχαίο εξωτερικό κόμβο με ένα φύλλο. Έστω T' το δέντρο που



Σχήμα 5-5: Ένα πλήρες δυαδικό δέντρο ύψους 4.
Σε κάθε επίπεδο i του δέντρου υπάρχουν 2^i εσωτερικοί κόμβοι. Όλοι οι εξωτερικοί κόμβοι βρίσκονται στο τελευταίο επίπεδο του δέντρου.

προκύπτει, το οποίο θα έχει μήκος εσωτερικού μονοπατιού I' και μήκος εξωτερικού μονοπατιού E'. Θα ισχύει βεβαίως

$$E' = E - l + 2\,(l+1) = E + l + 2 \tag{5.2}$$

και

$$I' = I + l \tag{5.3}$$

όπου l είναι το επίπεδο του εξωτερικού κόμβου που αντικαταστάθηκε. Με βάση τις σχέσεις (5.1) και (5.3), η σχέση 5.2 γίνεται

$$E' = 2k + \left(I' - l\right) + l + 2 = 2\,(k+1) + I'$$

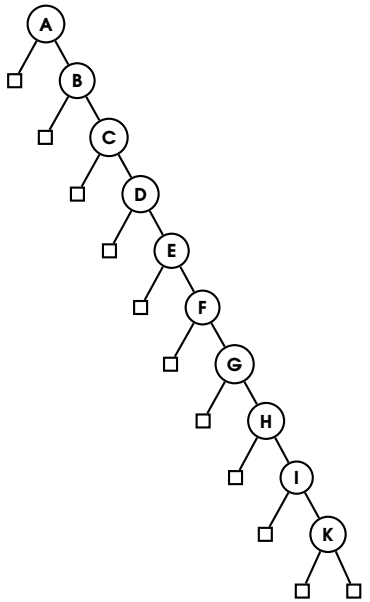
ολοκληρώνοντας την απόδειξη. □

Θεώρημα 5-4. *Το ύψος ενός δυαδικού δέντρου με N εσωτερικούς κόμβους είναι μεταξύ των τιμών $\lfloor \lg N \rfloor + 1$ και N συμπεριλαμβανομένων αυτών.*

Απόδειξη. Έστω ένα δυαδικό δέντρο με ύψος h και πλήθος εσωτερικών κόμβων N. Προφανώς $N \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$ και κατά συνέπεια

$$\lg\,(N+1) \leq h$$

Εξάλλου ο μόνος τρόπος να κατασκευάσουμε ένα δυαδικό δέντρο με το μέγιστο δυνατό ύψος είναι να έχουμε ακριβώς ένα εσωτερικό κόμβο σε κάθε επίπεδο. Αν το δέντρο έχει N εσωτερικούς κόμβους, τότε $h = N$. Εν γένει λοιπόν, $h \leq N$. □



Σχήμα 5-6: Ένα δυαδικό δέντρο ύψους 10.
Σε κάθε επίπεδο του δέντρου εκτός από το πρώτο και το τελευταίο, υπάρχει μόνο ένας εσωτερικός και ένας εξωτερικός κόμβος.

Ασκήσεις

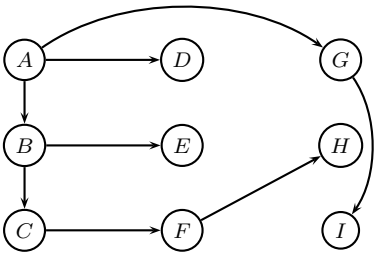
5-1 Σχεδιάστε το δέντρο $T = (V, E)$ με σύνολο κόμβων

$$V = \{k, n, i, g, h, t, s\}$$

και σύνολο ακμών

$$V = \{(k, n), (k, i), (n, g), (i, h), (i, t)\}\,.$$

5-2 Σχεδιάστε το παρακάτω δέντρο όπως στο Σχήμα 5-3. Ποιά είναι τα σύνολα V και E;



5-3 Υπολογίστε το μέγιστο πλήθος ακμών ενός γράφου με $|V|$ κόμβους.

5-4 Ένας γράφος S λέγεται *υπογράφος* του γράφου G, αν το σύνολο κόμβων του και το σύνολο ακμών του είναι υποσύνολα των αντίστοιχων συνόλων του G. Ένας *συνεκτικός γράφος* είναι ένας γράφος που περιλαμβάνει ένα τουλάχιστον μονοπάτι μεταξύ κάθε ζεύγους κόμβων. Σχεδιάστε τρεις τουλάχιστον συνεκτικούς υπογράφους του γράφου του Σχήματος 5-1 στη σελίδα 140.

5-5 Αποδείξτε πως ένας συνεκτικός γράφος $G = (V, E)$ με $|V| - 1$ ακμές είναι δέντρο.

5-6 Ένα δυαδικό δέντρο με ύψος h, έχει σε κάθε επίπεδο εκτός από το τελευταίο, το μέγιστο αριθμό εσωτερικών κόμβων. Δώστε τα όρια στα οποία κινείται το πλήθος N των εσωτερικών κόμβων του δέντρου.

5-7 Υπολογίστε το μήκος εσωτερικού και εξωτερικού μονοπατιού του δέντρου του Σχήματος 5-4 στη σελίδα 142.

5-8 Αποδείξτε πως το μήκος εσωτερικού μονοπατιού ενός δυαδικού δέντρου με N εσωτερικούς κόμβους είναι μεταξύ των τιμών $N \lg\left(\frac{N}{4}\right)$ και $N\frac{(N-1)}{2}$.

5.2 Υλοποίηση δυαδικών δέντρων

Για να υλοποιήσουμε ένα δυαδικό δέντρο πρέπει να μπορούμε να παραστήσουμε κόμβους και ακμές. Κάθε εσωτερικός κόμβος μπορεί να παρασταθεί από ένα αντικείμενο το οποίο έχει τρεις αναφορές: μία για το αποθηκευμένο αντικείμενο, την οποία ονομάζουμε `element`, και μία για κάθε κόμβο-παιδί τις οποίες ονομάζουμε `left` και `right`. Για τους εξωτερικούς κόμβους υπάρχουν δύο προσεγγίσεις. Με βάση την πρώτη, θεωρούμε πως αν η μεταβλητή `left` ή `right` ενός κόμβου είναι `null`, το αντίστοιχο παιδί είναι εξωτερικός κόμβος. Μια άλλη προσέγγιση είναι η χρήση ενός ειδικού αντικειμένου που παριστάνει τους εξωτερικούς κόμβους. Αν και στην πράξη χρησιμοποιείται η πρώτη προσέγγιση, εμείς θα παρουσιάσουμε και τις δύο.

Κώδικας 5.1: Η κλάση `BinaryTree`

```
1  package aueb.util.imp;
2
3  public class BinaryTree {
4
5     public static class Node {
6
7        protected static final Node EXTERNAL = new Node();
8
9        protected Object element;
10       protected Node left, right, parent;
11
12       private Node() {
13          super();
14       }
15
16       protected Node(Object element) {
17          this(element, EXTERNAL, EXTERNAL, EXTERNAL);
18       }
19
20       protected Node(Object element, Node parent, Node left, Node right) {
21          super();
22          if (parent == null || left == null || right == null) {
23             throw new IllegalArgumentException();
24          }
25          this.element = element;
26          this.left = left;
27          this.right = right;
```

```
28        }
29
30        public Object getElement() {
31           illegalOnExternalNode();
32           return element;
33        }
34
35        public void setElement(Object element) {
36           illegalOnExternalNode();
37           this.element = element;
38        }
39
40        public Node getParent() {
41           illegalOnExternalNode();
42           return parent;
43        }
44
45        public Node getLeftChild() {
46           illegalOnExternalNode();
47           return left;
48        }
49
50        public Node getRightChild() {
51           illegalOnExternalNode();
52           return right;
53        }
54
55        public boolean isExternal() {
56           return this == EXTERNAL;
57        }
58
59        public boolean isInternal () {
60           return this != EXTERNAL;
61        }
62
63        public boolean isLeaf() {
64           return this.left.isExternal() && this.right.isExternal();
65        }
66
67        private void illegalOnExternalNode() {
68           if (this == EXTERNAL) {
69              throw new IllegalStateException();
70           }
71        }
72
73     }
```

Η κλάση BinaryTree.Node του Κώδικα 5.1 υλοποιεί ένα κόμβο δυαδικού δέντρου σε Java. Η μεταβλητή element παριστάνει την πληροφορία που αποθηκεύει κάθε κόμβος. Κάθε κόμβος έχει επίσης δύο μεταβλητές left και right οι οποίες είναι οι ακμές που τον συνδέουν με τα δύο παιδιά του. Το πεδίο parent είναι μια αναφορά στον πατέρα ενός κόμβου.

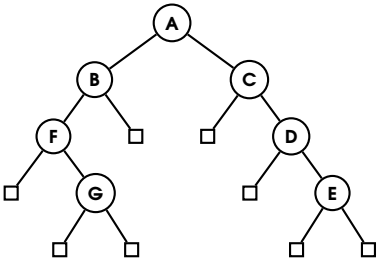
Η κλάση BinaryTree.Node, ορίζει ένα ειδικό κόμβο EXTERNAL (προσέξτε πως για την κατασκευή του χρησιμοποιείται ο ιδιωτικός κατασκευαστής της κλάσης) ο οποίος παριστάνει ένα εξωτερικό κόμβο. Έτσι αν το αριστερό (δεξί) παιδί ενός εσωτερικού κόμβου είναι εξωτερικός κόμβος, η μεταβλητή left (right) θα περιέχει τη διεύθυνση του αντικειμένου EXTERNAL.

Η κλάση παρέχει τρεις κατασκευαστές. Ο πρώτος είναι ιδιωτικός και χρησιμοποιείται μόνο και μόνο για την κατασκευή ενός και μοναδικού αντικειμένου, του κόμβου BinaryTree.Node.EXTERNAL που παριστάνει τους εξωτερικούς κόμβους. Οι υπόλοιποι κατασκευαστές είναι προστατευμένοι και κατά συνέπεια ορατοί εντός του πακέτου αλλά και από υποκλάσεις ανεξαρτήτως πακέτου.

Οι μέθοδοι getElement, getParent, getLeftChild και getRightChild, επιστρέφουν το περιεχόμενο αντικείμενο, τον πατέρα, το αριστερό και το δεξί παιδί ενός κόμβου, αντίστοιχα. Οι μέθοδοι αυτές δεν μπορούν να εκτελεστούν από τον εξωτερικό κόμβο. Για το λόγο αυτό, χρησιμοποιούμε την ιδιωτική μέθοδο illegalOnExternalNode η οποία σημαίνει μια εξαίρεση IllegalStateException όταν εκτελεστεί από τον εξωτερικό κόμβο BinaryTree.Node.EXTERNAL.

Για να μπορούμε να διακρίνουμε αν ένας κόμβος είναι φύλλο, εσωτερικός ή εξωτερικός κόμβος, η κλάση παρέχει τις μεθόδους isLeaf, isInternal και isExternal αντίστοιχα.

Για να σχηματίσουμε δυαδικά δέντρα, δεν έχουμε παρά να κατασκευάσουμε μερικά αντικείμενα της κλάσης BinaryTree.Node και να τα συνδέσουμε κατάλληλα. Η κλάση BinaryTree που παρουσιάζεται στον Κώδικα 5.2 παρέχει όλες τις απαιτούμενες μεθόδους για το χειρισμό δυαδικών δέντρων. Μπορούμε να πάρουμε μια ιδέα για τον τρόπο χρήσης της, μελετώντας το παρακάτω απόσπασμα κώδικα, το οποίο κατασκευάζει ένα δυαδικό δέντρο που αντιστοιχεί σε αυτό του Σχήματος 5-7.



Σχήμα 5-7: Κατασκευάζοντας ένα δυαδικό δέντρο.

```
BinaryTree tree = new BinaryTree();

BinaryTree.Node a = BinaryTree.makeLeafNode("A");
BinaryTree.Node b = BinaryTree.makeLeafNode("B");
BinaryTree.Node c = BinaryTree.makeLeafNode("C");
BinaryTree.Node d = BinaryTree.makeLeafNode("D");
BinaryTree.Node e = BinaryTree.makeLeafNode("E");
BinaryTree.Node f = BinaryTree.makeLeafNode("F");
BinaryTree.Node g = BinaryTree.makeLeafNode("G");

tree.setRootNode(a);
tree.setLeftChild(a, b);
tree.setRightChild(a, c);
tree.setLeftChild(b, f);
tree.setRightChild(c, d);
tree.setRightChild(d, e);
tree.setRightChild(f, g);
```

Κώδικας 5.2: Η κλάση BinaryTree (συνέχεια)

```
74
75      protected Node root;
76
77      public BinaryTree() {
78         super();
79         root = Node.EXTERNAL;
80      }
81
82      public BinaryTree(Node node) {
83         super();
84         root = node;
85         root.parent = Node.EXTERNAL;
86      }
87
88      public static Node makeLeafNode(Object element) {
89         return new Node(element);
90      }
91
92      public Node setRootNode(Node root) {
93         if (this.root != Node.EXTERNAL) {
94            throw new IllegalStateException();
95         }
96         checkNode(root);
97         return this.root = root;
98      }
99
100     public Node getRoot() {
```

```
101         return root;
102     }
103
104     public void setElement(Node node, Object element) {
105         checkNode(node);
106         checkElement(element);
107         node.element = element;
108     }
109
110     public void setLeftChild(Node parent, Node child) {
111         if (parent.left != Node.EXTERNAL) {
112             throw new IllegalStateException();
113         }
114         child.parent = parent;
115         parent.left = child;
116     }
117
118     public void setRightChild(Node parent, Node child) {
119         if (parent.right != Node.EXTERNAL) {
120             throw new IllegalStateException();
121         }
122         child.parent = parent;
123         parent.right = child;
124     }
125
126     public BinaryTree getLeftSubtree(Node node) {
127         checkNode(node);
128         return new BinaryTree(node.left);
129     }
130
131     public BinaryTree getLeftSubtree() {
132         return new BinaryTree(root.left);
133     }
134
135     public BinaryTree getRightSubtree(Node node) {
136         checkNode(node);
137         return new BinaryTree(node.right);
138     }
139
140     public BinaryTree getRightSubtree() {
141         return new BinaryTree(root.right);
142     }
143
144     public Node getSibling(Node node) {
145         checkNode(node);
146         return node == node.parent.left
```

```
147             ? node.parent.left
148             : node.parent.right;
149     }
150
151     public boolean isRoot(Node node) {
152         return root == node;
153     }
154
155     public int level(Node node) {
156         return isRoot(node) ? 0 : 1 + level(node.parent);
157     }
158
159     public int height() {
160         return height(root);
161     }
162
163     private int height(Node node) {
164         return node.isLeaf()
165             ? 1
166             : 1 + Math.max(height(node.left), height(node.right));
167     }
168
169     private void checkNode(Node node) {
170         if (node == null) {
171             throw new IllegalArgumentException();
172         }
173     }
174
175     private void checkElement(Object element) {
176         if (element == null) {
177             throw new IllegalArgumentException();
178         }
179     }
180
181 }
```

Η μόνη μεταβλητή που απαιτείται είναι μια αναφορά στη ρίζα του δέντρου. Ο εξ' ορισμού κατασκευαστής δημιουργεί ένα κενό δυαδικό δέντρο, δηλαδή ένα εξωτερικό κόμβο. Χρησιμοποιώντας τη στατική μέθοδο κατασκευής makeLeafNode μπορούμε να κατασκευάσουμε φύλλα.

Η μέθοδος setRootNode δέχεται ένα όρισμα τύπου BinaryTree.Node το οποίο γίνεται ρίζα του δέντρου. Κάτι τέτοιο μπορεί να γίνει ωστόσο μόνο εφόσον το δέντρο είναι κενό. Διαφορετικά η μέθοδος σημαίνει μια εξαίρεση τύπου

`IllegalStateException`.

Ασκήσεις

5-9 Γράψτε ένα απόσπασμα κώδικα το οποίο κατασκευάζει το δυαδικό δέντρο του Σχήματος 5-4.

5-10 Γράψτε ένα απόσπασμα κώδικα το οποίο κατασκευάζει το δυαδικό δέντρο του Σχήματος 5-6.

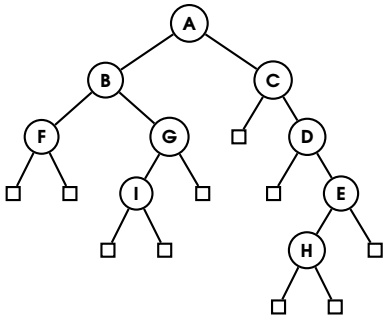
5-11 Υλοποιήστε την κλάση `OrderedTree`, τα αντικείμενα της οποίας είναι διατεταγμένα δέντρα. Κάθε κόμβος του δέντρου μπορεί να έχει μηδέν ή περισσότερα παιδιά.

5.3 Διάσχιση δυαδικών δέντρων

Η διάσχιση ενός δυαδικού δέντρου είναι η επεξεργασία όλων των κόμβων του με ένα συστηματικό τρόπο. Πρόκειται για την πιο θεμελιώδη διαδικασία επεξεργασίας ενός δυαδικού δέντρου, και αποτελεί σημαντικό τμήμα πολλών αλγορίθμων.

Σε αντίθεση με μια ακολουθία, όπου ο τρόπος διάσχισης προκύπτει διαισθητικά (για παράδειγμα ξεκινώντας από τον πρώτο κόμβο προς τον τελευταίο), ένα δυαδικό δέντρο δεν φαίνεται να έχει ένα προφανή τρόπο διάσχισης. Είναι λογικό να ξεκινήσουμε τη διάσχιση από τη ρίζα του δέντρου. Στη συνέχεια ωστόσο, έχουμε αρκετές επιλογές.

- Μπορούμε να επισκεφτούμε τη ρίζα, στη συνέχεια να προχωρήσουμε στο αριστερό υπόδεντρο, και αφού ολοκληρώσουμε τη διάσχισή του, να επιστρέψουμε στη ρίζα και να διασχίσουμε το δεξί υπόδεντρο με τον ίδιο τρόπο.
- Μπορούμε να διασχίσουμε το αριστερό υπόδεντρο, στη συνέχεια να επισκεφτούμε τη ρίζα, και τέλος να διασχίσουμε με τον ίδιο τρόπο το δεξί υπόδεντρο.
- Μπορούμε να διασχίσουμε το αριστερό και έπειτα το δεξί υπόδεντρο, και τέλος να επιστρέψουμε στη ρίζα προκειμένου να την επισκεφτούμε.
- Μπορούμε να επισκεφτούμε το αριστερό και έπειτα το δεξί παιδί της ρίζας και να συνεχίσουμε με τον ίδιο τρόπο με το



ΠΔΤ	ΕΔΤ	ΜΔΤ	ΚΕΠ
A	F	F	A
B	B	G	B
F	I	I	C
G	G	B	F
I	A	A	G
C	C	H	D
D	D	E	I
E	H	D	E
H	E	C	H

Σχήμα 5-8: Διάσχιση δυαδικών δέντρων.
Ο πίνακας παρουσιάζει τη σειρά με την οποία ανακαλύπτονται οι κόμβοι του δέντρου από κάθε μέθοδο διάσχισης.

επόμενο επίπεδο του δέντρου ώσπου να φτάσουμε στο τελευταίο.

Οι τρεις πρώτες από τις παραπάνω μεθόδους διάσχισης είναι σχεδόν πανομοιότυπες και διαφέρουν μονάχα κατά την χρονική στιγμή που επιλέγουμε να επισκεφτούμε τη ρίζα σε σχέση με τα παιδιά της.

Η πρώτη περίπτωση, στην οποία επισκεπτόμαστε τη ρίζα προτού διασχίσουμε τα υπόδεντρά της, ονομάζεται *προδιατεταγμένη διάσχιση* preorder traversal. Η δεύτερη, στην οποία επισκεπτόμαστε τη ρίζα μεταξύ της διάσχισης του αριστερού και του δεξιού υποδέντρου, ονομάζεται *ενδοδιατεταγμένη διάσχιση* (inorder traversal). Η τρίτη περίπτωση, στην οποία επισκεπτόμαστε τη ρίζα αφού προηγουμένως έχουμε επισκεφτεί και τα δύο υπόδεντρα, ονομάζεται *μεταδιατεταγμένη διάσχιση* (postorder traversal). Η τέταρτη μέθοδος διάσχισης ονομάζεται *διάσχιση κατά επίπεδα* level-order traversal.

Η διαφορά μεταξύ των τριών πρώτων μεθόδων και της τέταρτης είναι ότι οι πρώτες προχωρούν *διάσχιση κατά βάθος*, ενώ η τελευταία προχωρεί *κατά πλάτος* όπως φαίνεται και στο Σχήμα 5-8.

5.3.1 Αναδρομική διάσχιση κατά βάθος

Μια παρατηρητική ματιά στον τρόπο που περιγράψαμε τους τρόπους διάσχισης κατά βάθος, μας πείθει πως έχουμε μπροστά μας τρεις αναδρομικούς αλγορίθμους. Πράγματι μπορούμε εύκολα να υλοποιήσουμε τις τρεις αυτές μεθόδους διάσχισης χρησιμοποιώντας αναδρομή. Επειδή ωστόσο έχουμε τρεις διαφορετικές επιλογές, θα χρησιμοποιήσουμε μια αφηρημένη κλάση `TreeTraversal`, η οποία συγκεντρώνει τα κοινά χαρακτηριστικά όλων των μεθόδων διάσχισης. Η κλάση αυτή παρουσιάζεται στον Κώδικα 5.3.

Κώδικας 5.3: Η αφηρημένη κλάση `TreeTraversal`.

```
1  package aueb.util.imp.traversals;
2
3  import aueb.util.imp.BinaryTree;
4
5  public abstract class TreeTraversal {
6
7     public static interface NodeVisitor {
8        public void visit(BinaryTree.Node node);
9     }
10
```

```
11     protected BinaryTree tree;
12     protected NodeVisitor visitor;
13
14     public TreeTraversal(BinaryTree tree, NodeVisitor visitor) {
15        super();
16        if (tree == null || visitor == null) {
17           throw new IllegalArgumentException();
18        }
19        this.tree = tree;
20        this.visitor = visitor;
21     }
22
23     public abstract void start();
24  }
```

Η κλάση TreeTraversal έχει δύο μεταβλητές: Η πρώτη είναι μια αναφορά στο δυαδικό δέντρο προς διάσχιση, και η δεύτερη μια αναφορά σε ένα αντικείμενο που υλοποιεί τη διεπαφή TreeTraversal.NodeVisitor. Και οι δύο μεταβλητές αρχικοποιούνται κατά την κατασκευή ενός αντικειμένου και δεν μπορούν να αλλάξουν από εκεί και έπειτα.

Η διεπαφή TreeTraversal.NodeVisitor ορίζει μια και μοναδική μέθοδο visit. Η ύπαρξή της εξυπηρετεί *το διαχωρισμό του αλγορίθμου διάσχισης από τον αλγόριθμο επεξεργασίας των κόμβων του δέντρου*. Κάθε συγκεκριμένη απόγονος της κλάσης TreeTraversal πρέπει να καλεί τη μέθοδο visit στο αντικείμενο visitor, κάθε φορά που ανακαλύπτει ένα κόμβο κατά τη διάσχιση, ώστε να το "ειδοποιήσει" για την ανακάλυψη του κόμβου. Το είδος της επεξεργασίας που θα ακολουθήσει, αποτελεί ευθύνη της υλοποίησης του αντικειμένου visitor με το οποίο αρχικοποιήθηκε το αντικείμενο διάσχισης.

Κώδικας 5.4: Η κλάση RecursivePreorder

```
1  package aueb.util.imp.traversals;
2
3  import aueb.util.imp.BinaryTree;
4
5  public class RecursivePreorder extends TreeTraversal {
6
7     public RecursivePreorder(BinaryTree tree, NodeVisitor visitor) {
8        super(tree, visitor);
9     }
10
```

```
11     public void start() {
12        preorder(tree);
13     }
14
15     public void preorder(BinaryTree tree) {
16        BinaryTree.Node root = tree.getRoot();
17        if (root.isExternal()) {
18           return;
19        }
20        visitor.visit(root);
21        preorder(tree.getLeftSubtree());
22        preorder(tree.getRightSubtree());
23     }
24
25  }
```

Η κλάση RecursivePreorder που παρουσιάζεται στον Κώδικα 5.4 είναι μια απόγονος της κλάσης TreeTraversal που υλοποιεί τον αναδρομικό αλγόριθμο προδιατεταγμένης διάσχισης. Ο κατασκευαστής της κλάσης, απλώς καλεί τον κατασκευαστή της TreeTraversal· η μέθοδος preorder είναι η υλοποίηση του αλγορίθμου διάσχισης. Αν το δέντρο tree έχει ως ρίζα ένα εξωτερικό κόμβο, τότε δεν υπάρχουν κόμβοι για επεξεργασία (γραμμές 17-19). Διαφορετικά, πρέπει να γίνει επεξεργασία της ρίζας του δέντρου. Αυτό το αναλαμβάνει το αντικείμενο visitor. Είναι ωστόσο ευθύνη της μεθόδου διάσχισης να ειδοποιήσει το αντικείμενο αυτό καλώντας τη μέθοδο visit με όρισμα τον κόμβο που ανακαλύφθηκε. Μόλις η επεξεργασία ολοκληρωθεί (γραμμή 20) η διάσχιση θα πρέπει να προχωρήσει με αναδρομικό τρόπο στη διάσχιση του αριστερού υποδέντρου (γραμμή 21) και τέλος στη διάσχιση του δεξιού υποδέντρου (γραμμή 22). Για να ξεκινήσουμε τη διάσχιση δεν έχουμε παρά να καλέσουμε τη μέθοδο start σε ένα αντικείμενο τύπου RecursivePreorder το οποίο έχουμε κατασκευάσει προηγουμένως. Σε απλές περιπτώσεις όπως όταν για παράδειγμα θέλουμε απλώς να τυπώσουμε τα περιεχόμενα των κόμβων του δέντρου μπορούμε να συντομεύσουμε τη διαδικασία χρησιμοποιώντας μια ανώνυμη εσωτερική κλάση.

```
new RecursivePreorder(
   tree,
   new TreeTraversal.NodeVisitor() {
      public void visit(BinaryTree.Node n) {
         System.out.println(n);
```

```
      }
    }
).start();
```

Μπορούμε πολύ εύκολα να υλοποιήσουμε ανάλογες κλάσεις οι οποίες εκτελούν αναδρομική ενδοδιατεταγμένη και μεταδιατεταγμένη διάσχιση δυαδικών δέντρων. Η μόνη διαφορά θα είναι το όνομα της μεθόδου που υλοποιεί τη διάσχιση (π.χ., `inorder` αν πρόκειται για ενδοδιατεταγμένη διάσχιση ή `postorder` αν πρόκειται για μεταδιατεταγμένη) καθώς επίσης η θέση της γραμμής 20 του Κώδικα 5.4 σε σχέση με τις γραμμές 21 και 22. Η υλοποίηση των κλάσεων αυτών αφήνεται ως άσκηση.

Μπορούμε βέβαια, να συνδυάσουμε και τις τρεις μεθόδους αναδρομικής διάσχισης σε μία γενική μέθοδο. Παρατηρούμε ότι κατά την διάσχιση κατά βάθος, συναντάμε κάθε κόμβο τρεις φορές:

i). όταν ξεκινάμε την επεξεργασία του δέντρου που έχει ρίζα τον κόμβο αυτό,

ii). αφού έχουμε επεξεργαστεί το αριστερό του υπόδεντρο, και

iii). αφού έχουμε επεξεργαστεί το δεξί του υπόδεντρο.

Στην πρώτη περίπτωση λέμε ότι *η διάσχιση βρίσκεται αριστερά από του κόμβο*, στη δεύτερη ότι *βρίσκεται κάτω από τον κόμβο* και στην τρίτη ότι *βρίσκεται δεξιά απ' τον κόμβο*. Αυτή η μέθοδος διάσχισης ονομάζεται περίπατος Euler.

Μπορούμε λοιπόν να σχεδιάσουμε μια κλάση για τη διάσχιση δυαδικών δέντρων η οποία ανάλογα με τη θέση στην οποία βρίσκεται σε σχέση με ένα κόμβο, να καλεί μια μέθοδο επεξεργασίας όπως τη μέθοδο *ισιτ* της διεπαφής `TreeTraversal.NodeVisitor`. Βεβαίως τώρα πρέπει να έχουμε τρεις μεθόδους επεξεργασίας, μία για κάθε φορά που απαντάται ένας κόμβος κατά τη διάσχιση. Θα ονομάσουμε τις μεθόδους αυτές `left`, `below`, και `right`.

Η κλάση `EulerTour` που παρουσιάζεται στον Κώδικα 5.5 δεν είναι παρά η συγχώνευση του κώδικα της κλάσης `RecursivePreorder` με αυτόν των κλάσεων `RecursiveInorder` και `RecursivePostorder` της Άσκησης 5-14. Ωστόσο η διεπαφή του αντικειμένου επεξεργασίας πρέπει να αλλάξει, καθώς θα υπάρχουν πλέον τρεις αντί μιας μεθόδου. Χρειαζόμαστε μια διεπαφή ανάλογη της `TreeTraversal.NodeVisitor` την οποία θα ονομάσουμε `EulerTour.NodeVisitor` και θα περιλαμβάνει τις τρεις μεθόδους επεξεργασίας `left`, `down` και `right`.

Κώδικας 5.5: Η κλάση EulerTour

```
1  package aueb.util.imp.traversals;
2
3  import aueb.util.imp.BinaryTree;
4
5  /**
6   * Euler tour for binary trees.
7   * @author <a href="mailto:msintichakis@yahoo.com">Marios Sintichakis</a>
8   */
9  public class EulerTour {
10
11    public static interface NodeVisitor {
12
13        /**
14         * Before the traversal of a node's left subtree.
15         * @param node The node.
16         */
17        public void left(BinaryTree.Node node);
18
19        /**
20         * Right after having traversed a node's left subtree.
21         * @param node The node.
22         */
23        public void below(BinaryTree.Node node);
24
25        /**
26         * Right after having traversed a node's right subtree.
27         * @param node The node.
28         */
29        public void right(BinaryTree.Node node);
30
31    }
32
33    private BinaryTree tree;
34    private NodeVisitor visitor;
35
36    public EulerTour(BinaryTree tree, NodeVisitor visitor) {
37        super();
38        if (tree == null || visitor == null) {
39            throw new IllegalArgumentException();
40        }
41        this.tree = tree;
42        this.visitor = visitor;
43    }
44
45    public void start() {
```

```
46          traverse(tree.getRoot());
47      }
48
49      /**
50       * Template method that recursively processes all nodes in a binary tree.
51       * The actual processing is delegated to the visitor.
52       * @param node The root of the tree to traverse.
53       */
54      private void traverse(BinaryTree.Node node) {
55          if (node.isExternal()) {
56              return;
57          }
58          visitor.left(node);
59          traverse(node.getLeftChild());
60          visitor.below(node);
61          traverse(node.getRightChild());
62          visitor.right(node);
63      }
64
65  }
```

Με βάση τη σχεδίαση αυτή, μια ενδοδιατεταγμένη διάσχιση του δέντρου `tree` η οποία απλώς τυπώνει την πληροφορία που είναι αποθηκευμένη σε κάθε κόμβο, υλοποιείται ως εξής:

```
new EulerTour(
   tree,
   new EulerTour.NodeVisitor() {
      public void left(BinaryTree.Node n) {
         //Nothing
      }
      public void below(BinaryTree.Node n) {
         System.out.println(n);
      }
      public void right(BinaryTree.Node n) {
         //Nothing
      }
   }
).start();
```

5.3.2 Επαναληπτική διάσχιση κατά βάθος.

Όπως ξέρουμε, κάθε αναδρομικός αλγόριθμος συνεπάγεται μια επιβάρυνση στο χρόνο εκτέλεσης του προγράμματος λόγω του σημαντικού αριθμού κλήσεων μεθόδων. Γνωρίζουμε ωστόσο ότι μπορούμε,

εύκολα ή δύσκολα, για κάθε αναδρομικό αλγόριθμο να κατασκευάσουμε ένα ισοδύναμο επαναληπτικό.

Για τους αλγόριθμους κατά βάθος διάσχισης δυαδικών δέντρων, μπορούμε να αντικαταστήσουμε τις αναδρομικές κλήσεις με ένα βρόχο. Σε κάθε επανάληψη του βρόχου θα επεξεργαζόμαστε τη ρίζα ενός υποδέντρου. Βεβαίως χρειαζόμαστε να αποθηκεύουμε τις ρίζες των υποδέντρων που συναντάμε, μέχρι να φτάσει η στιγμή της επεξεργασίας τους. Στην αναδρομική υλοποίηση αυτό επιτυγχάνεται από την ίδια τη στοίβα που δημιουργείται από τις αναδρομικές κλήσεις. Είναι λογικό λοιπόν, στην επαναληπτική υλοποίηση, να χρησιμοποιήσουμε μια στοίβα για να επιτύχουμε το ίδιο αποτέλεσμα.

Αρχικά η στοίβα θα έχει ένα μόνο στοιχείο, τη ρίζα του δέντρου. Σε κάθε επανάληψη του βρόχου διάσχισης εξάγουμε το στοιχείο που βρίσκεται στην κορυφή της στοίβας· αν αυτό είναι το περιεχόμενο ενός κόμβου, το επεξεργαζόμαστε. Αν αντίθετα είναι κόμβος του δέντρου, τότε ωθούμε στη στοίβα το αριστερό και το δεξί παιδί της ρίζας του υποδέντρου αυτού, καθώς επίσης και το στοιχείο που βρίσκεται αποθηκευμένο στη ρίζα. Η σειρά με την οποία πρέπει να γίνουν αυτές οι πράξεις, εξαρτάται από τον τρόπο διάσχισης που θέλουμε να υλοποιήσουμε και παρουσιάζεται στον Πίνακα 5-1. Η παραπάνω διαδικασία συνεχίζεται επαναληπτικά ώσπου καταλήξουμε με μια κενή στοίβα, οπότε και ολοκληρώνεται η διάσχιση του δέντρου.

ΠΔΤ	ΕΔΤ	ΜΔΤ
PUSH R	PUSH R	PUSH E
PUSH L	PUSH E	PUSH R
PUSH E	PUSH L	PUSH L

Πίνακας 5-1: Χειρισμός της στοίβας κατά την επαναληπτική διάσχιση δυαδικών δέντρων.
Η εντολή **PUSH R** (**PUSH L**) υποδηλώνει ώθηση στη στοίβα του κόμβου-ρίζας του αριστερού (δεξιού) υποδέντρου του τρέχοντος υποδέντρου ενώ η εντολή **PUSH E** υποδηλώνει την ώθηση στη στοίβα του αντικειμένου που βρίσκεται αποθηκευμένο στη ρίζα του τρέχοντος υποδέντρου.

Παρατηρούμε ότι το δεξί υπόδεντρο ωθείται στη στοίβα πάντα πριν από το αριστερό έτσι ώστε να εξαχθεί από τη στοίβα αφού θα έχει ήδη εξαχθεί το δεξί. Μια τελευταία λεπτομέρεια που δεν έχει διευκρινιστεί είναι η χρονική στιγμή που πρέπει να επεξεργαστούμε ένα κόμβο: όταν συναντήσουμε στη στοίβα ένα αντικείμενο που δεν είναι δέντρο. Η υλοποίηση της ενδοδιατεταγμένης διάσχισης δίνεται στον Κώδικα 5.6.

Κώδικας 5.6: Επαναληπτική ενδοδιατεταγμένη διάσχιση δυαδικού δέντρου.

```
1  package aueb.util.imp.traversals;
2
3  import aueb.util.imp.BinaryTree;
4  import aueb.util.imp.Deque;
5
6  public class IterativeInorder extends TreeTraversal {
7
```

```
8     public IterativeInorder(BinaryTree tree, NodeVisitor visitor) {
9        super(tree, visitor);
10    }
11
12    public void start() {
13       Deque s = new Deque();
14       BinaryTree t;
15       BinaryTree.Node node = null;
16
17       s.pushHead(tree);
18       while (!s.isEmpty()) {
19          Object o = s.popHead();
20          if (o instanceof BinaryTree) {
21             t = (BinaryTree)o;
22             node = t.getRoot();
23             if (node.isExternal()) {
24                continue;
25             }
26             s.pushHead(t.getRightSubtree());
27             s.pushHead(node);
28             s.pushHead(t.getLeftSubtree());
29             continue;
30          }
31          node = (BinaryTree.Node)o;
32          visitor.visit(node);
33       }
34    }
35
36 }
```

5.3.3 Επαναληπτική διάσχιση κατά πλάτος.

Η μέθοδος διάσχισης κατά πλάτος επισκέπτεται τους κόμβους ενός δυαδικού δέντρου από το επίπεδο 0, και από αριστερά προς τα δεξιά κατά επίπεδο. Ας αναλύσουμε λίγο περισσότερο τα μηχανικά αυτής της διαδικασίας.

Αν το δέντρο το οποίο διασχίζουμε έχει ένα μοναδικό κόμβο, τη ρίζα, τότε επισκεπτόμαστε τον κόμβο αυτό και ολοκληρώνουμε τη διάσχιση. Αν ωστόσο η ρίζα έχει δύο παιδιά, τότε πρέπει να επισκεφτούμε πρώτα το αριστερό, έπειτα το δεξί και στη συνέχεια να προχωρήσουμε στο επόμενο επίπεδο, και να επαναλάβουμε τα ίδια βήματα. Με άλλα λόγια καθώς διασχίζουμε ένα επίπεδο από τα αριστερά προς τα δεξιά, πρέπει να σημειώνουμε με κάποιο τρόπο

τους κόμβους του επόμενου επιπέδου (με την ίδια σειρά, από τα αριστερά προς τα δεξιά) ώστε να συνεχίσουμε τη διάσχιση του επιπέδου αυτού. Η πιο κατάλληλη δομή δεδομένων για μια τέτοιου είδους επεξεργασία είναι μια ουρά αναμονής στην οποία εισάγουμε τα παιδιά ενός κόμβου, καθώς τον επεξεργαζόμαστε. Η υλοποίηση του αλγορίθμου διάσχισης κατά επίπεδα δίνεται στον Κώδικα 5.7.

Κώδικας 5.7: Επαναληπτική διάσχιση δυαδικού δέντρου κατά πλάτος.

```
1  package aueb.util.imp.traversals;
2
3  import aueb.util.imp.BinaryTree;
4  import aueb.util.imp.Deque;
5
6  public class BreadthFirst extends TreeTraversal {
7
8     public BreadthFirst(BinaryTree tree, NodeVisitor visitor) {
9        super(tree, visitor);
10    }
11
12    public void start() {
13       Deque s = new Deque();
14       BinaryTree.Node node = tree.getRoot();
15       if (node.isExternal()) {
16          return;
17       }
18       s.pushTail(node);
19       while (!s.isEmpty()) {
20          node = (BinaryTree.Node) s.popHead();
21          if (node.isExternal()) {
22             continue;
23          }
24          visitor.visit(node);
25          s.pushTail(node.getLeftChild());
26          s.pushTail(node.getRightChild());
27       }
28    }
29
30 }
```

5.3.4 Δυαδικά δέντρα ως ακολουθίες κόμβων

Και οι τέσσερις μέθοδοι διάσχισης που περιγράψαμε διατάσσουν τους κόμβους ενός δυαδικού δέντρου σε μια γραμμή, παράγουν δηλαδή μια ακολουθία. Γνωρίζουμε ωστόσο, πως σε κάθε ακολουθία

κάθε όρος εκτός του πρώτου έπεται ενός άλλου όρου και κάθε όρος εκτός του τελευταίου προηγείται ενός άλλου κόμβου. Έχει ενδιαφέρουν να εξερευνήσουμε τις ιδιότητες των ακολουθιών κόμβων που παράγουν οι μέθοδοι διάσχισης ενός δυαδικού δέντρου και ειδικά τις ακολουθίες που παράγει η ενδοδιατεταγμένη διάσχιση η οποία ονομάζεται και συμμετρική διάσχιση.

Αρχικά ας εντοπίσουμε τον πρώτο και τον τελευταίο κόμβο που επισκεπτόμαστε κατά τη συμμετρική διάσχιση. Με λίγο πειραματισμό σε διαφορετικά δέντρα, θα παρατηρήσουμε πως ο πρώτος κόμβος που επισκεπτόμαστε είναι ο αριστερότερος κόμβος του δέντρου. Πράγματι, επειδή η συμμετρική διάσχιση πηγαίνει συνεχώς προς τα κάτω και αριστερά μέχρι να συναντήσει ένα εξωτερικό κόμβο, ο πρώτος όρος που θα επισκεφτεί είναι εκείνο το φύλλο του δέντρου που βρίσκεται πιο αριστερά από τα υπόλοιπα. Πολύ εύκολα προκύπτει με ανάλογη ανάλυση πως ο τελευταίος κόμβος που επισκέπτεται η ενδοδιατεταγμένη διάσχιση είναι το δεξιότερο φύλλο του δέντρου. Αλήθεια είναι πως αν δεν συνέβαινε κάτι τέτοιο, ο χαρακτηρισμός συμμετρική θα ήταν λίγο άστοχος.

Ας δούμε τώρα πως μπορούμε να εντοπίσουμε τον επόμενο κόμβο έστω $\mathrm{succ}\,(p)$ ενός κόμβου p. Αν το δεξί παιδί του p δεν είναι εξωτερικός κόμβος, τότε ο αλγόριθμος διάσχισης αφού επισκεφτεί τον p θα συνεχίσει με το δεξί του παιδί, κινούμενος συνεχώς προς τα κάτω και αριστερά. Άρα ο $\mathrm{succ}\,(p)$ δεν μπορεί να είναι άλλος από το αριστερότερο παιδί του δεξιού του υπόδεντρου. Αν τώρα το δεξί παιδί του p είναι εξωτερικός κόμβος, ο αλγόριθμος διάσχισης έχοντας ολοκληρώσει με τόσο με το αριστερό παιδί του p, αλλά και τον ίδιο τον p, θα κινηθεί προς τα πάνω, στο μονοπάτι από τη ρίζα προς τον p. Οι πρόγονοι του p που συναντά ο αλγόριθμος και οι οποίοι είναι αριστερότερα του p, δεν μπορεί να έπονται του p καθώς ο p βρίσκεται στο δεξί τους υπόδεντρο. Επομένως ο $\mathrm{succ}\,(p)$ είναι ο πρώτος πρόγονος του p στο μονοπάτι απ' τον p προς τη ρίζα, ο οποίος έχει τον p στο αριστερό του υπόδεντρο. Αν τέτοιος κόμβος δεν υπάρχει τότε ο p είναι ο τελευταίος κόμβος της ενδοδιατεταγμένης ακολουθίας.

Ακολουθώντας ανάλογο συλλογισμό, συμπεραίνουμε πως ο κόμβος $\mathrm{pred}\,(p)$, που προηγείται του κόμβου p στην ενδοδιατεταγμένη ακολουθία κόμβων ενός δυαδικού δέντρου, είναι το δεξιότερο παιδί του αριστερού του υπόδεντρου, εφόσον αυτό δεν είναι εξωτερικός

κόμβος, ή ο πρώτος πρόγονος του p στο μονοπάτι από τον p προς τη ρίζα, ο οποίος έχει τον p στο δεξί του υπόδεντρο. Αν δεν υπάρχει τέτοιος κόμβος, τότε ο p δεν έχει προηγούμενο, είναι ο πρώτος κόμβος στην ενδοδιατεταγμένη ακολουθία.

Ασκήσεις

5-12 Δώστε τις ακολουθίες κλήσεων της προδιατεταγμένης, ενδοδιατεταγμένης και μεταδιατεταγμένης διάσχισης του δέντρου του Σχήματος 5-4 στη σελίδα 143.

5-13 Δώστε την ακολουθία επίσκεψης των κόμβων του δέντρου του Σχήματος 5-5 στη σελίδα 143, κατά την προδιατεταγμένη, ενδοδιατεταγμένη, μεταδιατεταγμένη και κατά επίπεδα διάσχιση.

5-14 Υλοποιείστε τις κλάσεις `ResursiveInorder` και `RecursivePostorder` στα πρότυπα της κλάσης `RecursivePreorder`.

5-15 Υλοποιήστε τη διεπαφή `Enumerator` για ένα δυαδικό δέντρο, χρησιμοποιώντας τον παρακάτω αλγόριθμο επαναληπτικής ενδοδιατεταγμένης διάσχισης.

```
begin
   if the tree is an external node return;
   set p = root of tree;
   repeat
      set n = p
      while n is not an external node do
         push n to s
         set n = left child of n
      end while
      if s is empty return
      pop from s to p
      visit p
      set p = right child of p
   end repeat
end
```

5-16 Υλοποιήστε τη διεπαφή `Enumerator` για ένα δυαδικό δέντρο, χρησιμοποιώντας διάσχιση κατά επίπεδα.

5-17 Υλοποιήστε τη μέθοδο `BinaryTree.Node succ(BinaryTree.Node)` η οποία επιστρέφει τον επόμενο ενός κόμβου στην ενδοδιατεταγμένη ακολουθία ενός δυαδικού δέντρου.

5-18 Ορίστε τον επόμενο και προηγούμενο ενός κόμβου δυαδικού δέντρου στην προδιατεταγμένη και μεταδιατεταγμένη ακολουθία.

5-19 Υλοποιήστε τη διεπαφή `Enumerator` για ένα δυαδικό δέντρο, χωρίς να χρησιμοποιήσετε βοηθητική δομή δεδομένων.

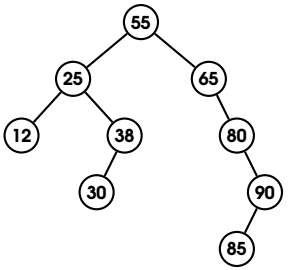
ΚΕΦΑΛΑΙΟ 6

Δέντρα αναζήτησης

Οι μηχανισμοί οργάνωσης δεδομένων που έχουμε παρουσιάσει μέχρι στιγμής δεν παρέχουν τη δυνατότητα αποτελεσματικής υλοποίησης και των τριών βασικών λειτουργιών (εισαγωγή, αναζήτηση, διαγραφή) ενός δυναμικού συνόλου. Η χρήση συστοιχίας ή λίστας παρέχει πολύ γρήγορη εισαγωγή αλλά αργή αναζήτηση και διαγραφή. Η χρήση ταξινομημένης συστοιχίας παρέχει γρήγορη αναζήτηση αλλά αργή εισαγωγή. Στο κεφάλαιο αυτό θα παρουσιάσουμε μερικές υλοποιήσεις οι οποίες βασίζονται στην οργάνωση των δυαδικών δέντρων. Ξεκινώντας από την απλούστερη τέτοια οργάνωση, τα δυαδικά δέντρα αναζήτησης, θα καταλήξουμε σε δομές στις οποίες η εκτέλεση των βασικών λειτουργιών μιας συλλογής N στοιχείων απαιτεί χρόνο $O(\lg N)$ στη χειρότερη περίπτωση.

6.1 Δυαδικά δέντρα αναζήτησης

Τα δυαδικά δέντρα αναζήτησης είναι ένας εξαιρετικά απλός και σε αρκετές περιπτώσεις πολύ αποτελεσματικός μηχανισμός οργάνωσης δεδομένων. Πολλές υλοποιήσεις αφηρημένων τύπων δεδομένων, βασίζονται σε δυαδικά δέντρα αναζήτησης καθώς αυτά συνδυάζουν αποτελεσματικότητα των περισσοτέρων λειτουργιών, απλότητα υλοποίησης και μέτριες απαιτήσεις μνήμης.



ΠΔΤ	55 25 12 38 30 65 80 90 85
ΕΔΤ	12 25 30 35 55 65 80 85 90
ΜΔΤ	12 30 38 25 85 90 80 65 55
ΚΕΠ	55 25 65 12 38 80 30 90 85

Σχήμα 6-1: Δυαδικό δέντρο αναζήτησης.
Το αντικείμενο που είναι αποθηκευμένο σε κάθε κόμβο είναι μεγαλύτερο από τα αντικείμενα που είναι αποθηκευμένα στους κόμβους του αριστερού του υποδέντρου και μικρότερο από τα αντικείμενα που είναι αποθηκευμένα στους κόμβους του δεξιού του υποδέντρου. Η ενδοδιατεταγμένη ακολουθία κόμβων ενός δυαδικού δέντρου αναζήτησης είναι ταξινομημένη σε αύξουσα διάταξη όπως φαίνεται στον πίνακα στο κάτω μέρος του σχήματος.

6.1.1 Βασικές έννοιες και λειτουργίες

Ορισμός 6-1. Ένα *δυαδικό δέντρο αναζήτησης* είναι ένα δυαδικό δέντρο στο οποίο το αντικείμενο που είναι αποθηκευμένο σε κάθε κόμβο είναι μεγαλύτερο από τα αντικείμενα που είναι αποθηκευμένα στους κόμβους του αριστερού του υποδέντρου και μικρότερο από τα στοιχεία που βρίσκονται αποθηκευμένα στους κόμβους του δεξιού του υποδέντρου.

Για τον ορισμό ενός δυαδικού δέντρου αναζήτησης απαιτείται μια σχέση διάταξης προκειμένου να μπορούμε να ορίσουμε τις έννοιες "μικρότερο" και "μεγαλύτερο". Ο Ορισμός 6-1 έχει μια πολύ ενδιαφέρουσα συνέπεια: η ενδοδιατεταγμένη ακολουθία των κόμβων ενός δυαδικού δέντρου αναζήτησης είναι ταξινομημένη σε αύξουσα διάταξη.

Θεώρημα 6-1. *Η ενδοδιατεταγμένη ακολουθία κόμβων ενός δυαδικού δέντρου αναζήτησης είναι ταξινομημένη σε αύξουσα διάταξη.*

Απόδειξη. Όλοι οι κόμβοι που προηγούνται ενός κόμβου v στην ενδοδιατεταγμένη ακολουθία, βρίσκονται στο αριστερό του υπόδεντρο και κατά συνέπεια του Ορισμού 6-1 αποθηκεύουν αντικείμενα που είναι μικρότερα από το αποθηκευμένο στον v αντικείμενο. Ομοίως οι κόμβοι που έπονται ενός κόμβου v στην ενδοδιατεταγμένη ακολουθία, βρίσκονται στο δεξί του υπόδεντρο και επομένως τα αντικείμενα που αποθηκεύουν είναι μεγαλύτερα από το αντικείμενο που είναι αποθηκευμένο στον v. □

Αναζήτηση. Σε αντίθεση με τα συνηθισμένα δυαδικά δέντρα, στα οποία προκειμένου να εντοπίσουμε τον κόμβο στον οποίο είναι αποθηκευμένο ένα αντικείμενο, πρέπει να επισκεφτούμε όλους τους κόμβους, τα δυαδικά δέντρα αναζήτησης προσφέρουν ένα πολύ πιο αποτελεσματικό τρόπο. Ο αλγόριθμος αναζήτησης περιλαμβάνει τα παρακάτω βήματα.

i). Όπως συνήθως, ξεκινάμε την αναζήτηση από τη ρίζα του δέντρου.

ii). Αν το ζητούμενο αντικείμενο είναι ίσο με το αντικείμενο στη ρίζα, η αναζήτηση τερματίζει επιτυχώς.

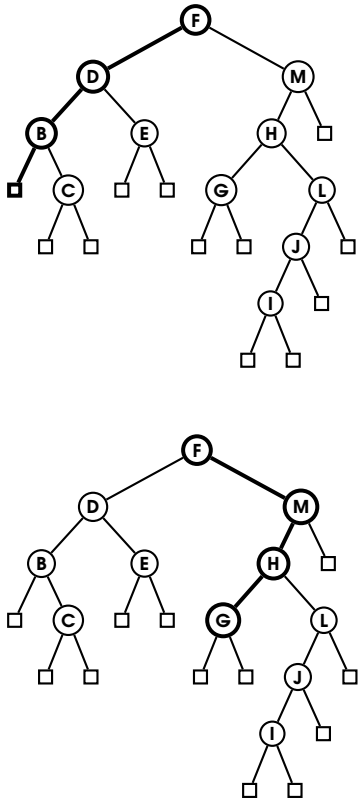
iii). Αν το ζητούμενο αντικείμενο είναι μικρότερο από το αντικείμενο που βρίσκεται αποθηκευμένο στη ρίζα, τότε η αναζήτηση

συνεχίζεται στο αριστερό υπόδεντρο, καθώς το όλα τα αντικείμενα που βρίσκονται στο δεξί υπόδεντρο είναι μεγαλύτερα από το ζητούμενο αντικείμενο. Αν αντίθετα το ζητούμενο αντικείμενο είναι μεγαλύτερο από το αντικείμενο που είναι αποθηκευμένο στη ρίζα, η αναζήτηση συνεχίζεται στο δεξί υπόδεντρο.

iv). Αν η αναζήτηση καταλήξει σ' ένα εξωτερικό κόμβο, τότε η αναζήτηση είναι ανεπιτυχής. Το ζητούμενο αντικείμενο δεν είναι στοιχείο του δέντρου.

Ο αλγόριθμος αναζήτησης σε δυαδικό δέντρο αναζήτησης παρουσιάζεται γραφικά στο Σχήμα 6-2. Οι έντονες γραμμές υποδεικνύουν το *μονοπάτι αναζήτησης*, την ακολουθία κόμβων δηλαδή που εξετάζονται καθώς και τις ακμές ακμή που διασχίζονται κάθε φορά. Παρατηρούμε πως σε κάθε επίπεδο του δέντρου λαμβάνεται μια απόφαση η οποία εξαλείφει την ανάγκη αναζήτησης στο ένα από τα δύο υπόδεντρα.

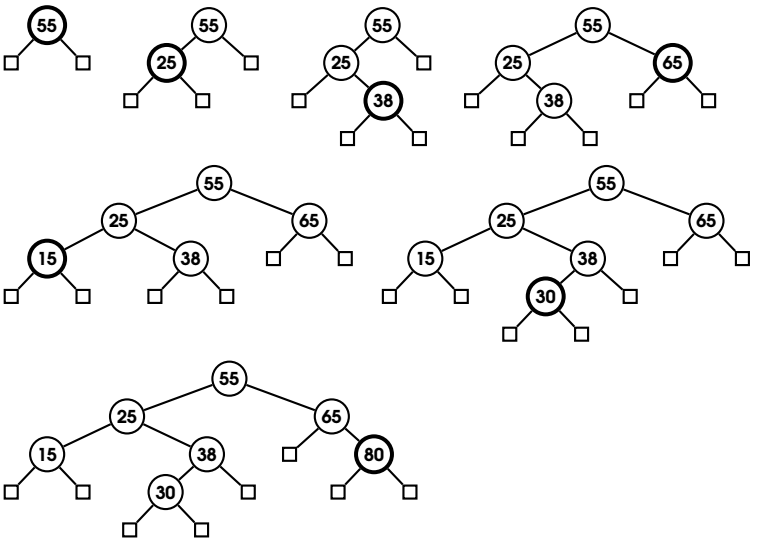
Αυτός ο αλγόριθμος θυμίζει έντονα τον αλγόριθμο δυαδικής αναζήτησης καθώς σε κάθε επανάληψη όσα αντικείμενα βρίσκονται στα αριστερά ή στα δεξιά του "μεσαίου" αντικειμένου, απορρίπτονται με μία και μόνη σύγκριση αντικειμένων. Βεβαίως στην δυαδική αναζήτηση ακριβώς τα μισά αντικείμενα απορρίπτονται σε κάθε επανάληψη. Αυτό δεν ισχύει απαραίτητα σε ένα δυαδικό δέντρο αναζήτησης, καθώς το πλήθος των κόμβων του δέντρου δεν είναι κατ' ανάγκη κατανεμημένο ομοιόμορφα στα υπόδεντρα κάθε επιπέδου. Το σχήμα δηλαδή του δέντρου είναι και ο βασικός παράγοντας που καθορίζει την αποτελεσματικότητα της αναζήτησης.



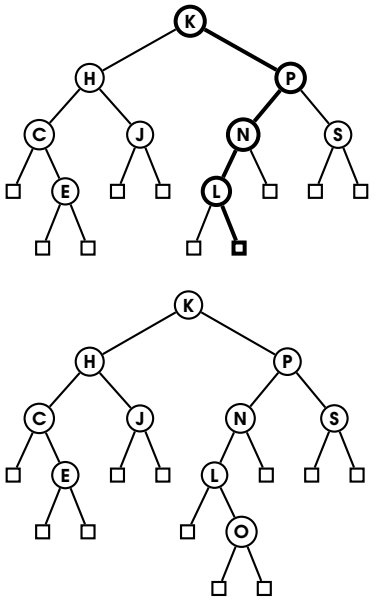
Σχήμα 6-2: Αναζήτηση αντικειμένου σε δυαδικό δέντρο αναζήτησης.
Σε κάθε επίπεδο του δέντρο αποκλείεται ένα από τα δύο υπόδεντρα του εξεταζόμενου κόμβου. Πάνω η διαδρομή που ακολουθεί ο αλγόριθμος αναζήτησης προσπαθώντας να εντοπίσει το αντικείμενο **A**, ενώ δεξιά η διαδρομή που ακολουθείται για τον εντοπισμό του αντικειμένου **G**. Στην πρώτη περίπτωση η αναζήτηση τερματίζει ανεπιτυχώς ενώ στη δεύτερη επιτυχώς.

Εισαγωγή. Ο απλούστερος τρόπος εισαγωγής ενός κόμβου σε ένα δυαδικού δέντρου είναι με την αντικατάσταση οποιουδήποτε εξωτερικού κόμβου από ένα νέο φύλλο όπως κάνει η κλάση `BinaryTree` (βλ. Κώδικα 5.2 στη σελίδα 148). Αυτός ο τρόπος εισαγωγής μπορεί να χρησιμοποιηθεί και σε δυαδικά δέντρα αναζήτησης με την προϋπόθεση πως ο εξωτερικός κόμβος που θα αντικατασταθεί θα επιλεγεί έτσι ώστε το νέο δυαδικό δέντρο που θα προκύψει θα ικανοποιεί τη συνθήκη του Ορισμού 6-1.

Ας δούμε τον αλγόριθμο εισαγωγής με ένα παράδειγμα προσπαθώντας να εισάγουμε ένα κόμβο με το αντικείμενο **o** στο δυαδικό δέντρο αναζήτησης του Σχήματος 6-4. Ξεκινάμε προσπαθώντας να



Σχήμα 6-3: Κατασκευή δυαδικού δέντρου αναζήτησης με διαδοχικές εισαγωγές αντικειμένων.
Οι σχεδιασμένοι με έντονες γραμμές κόμβοι υποδεικνύουν το αντικείμενο που εισάγεται κάθε φορά.



Σχήμα 6-4: Εισαγωγή κόμβου σε δυαδικό δέντρο αναζήτησης.
Επάνω, φαίνεται το μονοπάτι αναζήτησης. Κάτω ο αλγόριθμος αντικαθιστά τον εξωτερικό κόμβο στον οποίο τερματίστηκε ανεπιτυχώς η αναζήτηση με ένα νέο φύλλο.

εντοπίσουμε ένα κόμβο με το στοιχείο αυτό. Η αναζήτηση θα τερματίσει ανεπιτυχώς στο δεξί παιδί του κόμβου που περιέχει το αντικείμενο **ι** αφού διασχίσει το μονοπάτι αναζήτησης που υποδεικνύεται με έντονες γραμμές. Αν τώρα αντικαταστήσουμε τον εξωτερικό αυτό κόμβο με ένα φύλλο που περιέχει το αντικείμενο **ο**, το δέντρο που προκύπτει είναι δυαδικό δέντρο αναζήτησης. Η τροποποίηση που κάναμε περιορίζεται στο υπόδεντρο με ρίζα τον κόμβο που περιέχει το αντικείμενο **ι** και εφόσον αυτό είναι δυαδικό δέντρο αναζήτησης, το ίδιο ισχύει και ολόκληρο το δέντρο. Ανακεφαλαιώνοντας, τα βήματα του αλγορίθμου εισαγωγής έχουν ως εξής.

i). Ξεκινώντας από τη ρίζα του δέντρου αναζητούμε ένα κόμβο που περιέχει το αντικείμενο που πρόκειται να εισαχθεί.

ii). Αν η αναζήτηση τερματίσει επιτυχώς, το αντικείμενο περιέχεται ήδη στο δέντρο και η εισαγωγή τερματίζει.

iii). Αν η αναζήτηση τερματίσει χωρίς επιτυχία, αντικαθιστούμε τον τελευταίο κόμβο του μονοπατιού αναζήτησης με ένα νέο φύλλο που περιέχει το προς εισαγωγή αντικείμενο.

Το σχήμα ενός δυαδικού δέντρου αναζήτησης που κατασκευάζεται με διαδοχικές εισαγωγές στοιχείων εξαρτάται απολύτως από τη διάταξη της ακολουθίας στοιχείων που εισάγονται. Σε μερικές περιπτώσεις, όπως για παράδειγμα όταν η ακολουθία αντικειμένων είναι

ταξινομημένη, το το δέντρου που προκύπτει έχει ένα κόμβο σε κάθε επίπεδο. Ένα τέτοιο δέντρο είναι στην ουσία μια ταξινομημένη συνδεδεμένη λίστα και η αποτελεσματικότητα αναζήτησης είναι φτωχή.

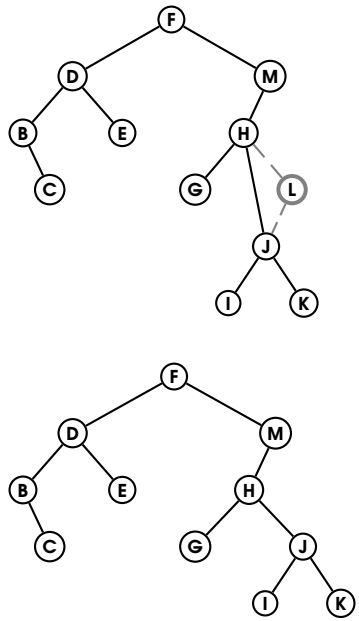
Διαγραφή. Ας εξετάσουμε τώρα με ποιο τρόπο μπορούμε να εξασφαλίσουμε ότι ένα δυαδικό δέντρο αναζήτησης εξακολουθεί να ικανοποιεί τη συνθήκη ορισμού όταν διαγράψουμε ένα τυχαίο κόμβο.

Αν για παράδειγμα θέλουμε να διαγράψουμε τον κόμβο που αποθηκεύει το αντικείμενο **30** στο τελικό δέντρο του Σχήματος 6-3. Επειδή ο κόμβος αυτός είναι φύλλο, η αφαίρεσή του από το δέντρο, η αντικατάστασή του δηλαδή με ένα εξωτερικό κόμβο, δεν επηρεάζει τη συνθήκη ορισμού.

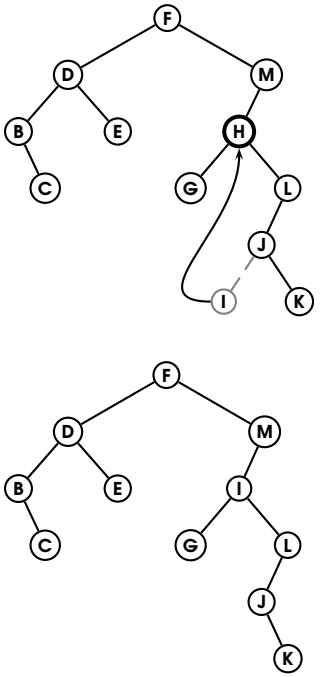
Αν ωστόσο θέλουμε να διαγράψουμε τον κόμβο που αποθηκεύει το αντικείμενο **65**, τότε πρέπει να βρούμε ένα νέο πατέρα για το δεξί του παιδί. Αν κάνουμε το δεξί παιδί του κόμβου προς διαγραφή, δεξί παιδί του πατέρα του πατέρα του, η συνθήκη ορισμού δεν παραβιάζεται.

Αν τώρα θέλουμε να διαγράψουμε τον κόμβο που αποθηκεύει το αντικείμενο **65**, τότε δυστυχώς δεν μπορούμε να εργαστούμε όπως προηγουμένως, καθώς ο κόμβος αυτός έχει δύο παιδιά. Ας φέρουμε στο μυαλό μας τον τρόπο με τον οποίο διαγράφουμε ένα αντικείμενο στην κλάση `ArrayCollection` μεταφέροντας το τελευταίο αντικείμενο στη θέση του αντικειμένου που πρέπει να διαγραφεί και στη συνέχεια διαγράφουμε το τελευταίο αντικείμενο. Κάτι αντίστοιχο μπορεί να γίνει και εδώ. Αρκεί να βρούμε ένα αντικείμενο το οποίο μπορεί να αντικαταστήσει το αντικείμενο του κόμβου προς διαγραφή χωρίς να παραβιάζεται η συνθήκη ορισμού, και στη συνέχεια να αποσυνδέσουμε από το δέντρο τον κόμβο στον οποίο εκείνο το αντικείμενο ήταν αρχικά αποθηκευμένο. Βέβαια το αντικείμενο αυτό, εφόσον υπάρχει, πρέπει να βρίσκεται σε κόμβο ο οποίος να έχει το πολύ ένα παιδί, διαφορετικά η διαδικασία που περιγράψαμε θα πρέπει να επαναληφθεί.

Υπάρχουν δύο κόμβοι τα αντικείμενα των οποίων μπορούν να αντικαταστήσουν το το αντικείμενο ενός δοσμένου κόμβου p, έτσι ώστε το δέντρο να παραμένει δυαδικό δέντρο αναζήτησης. Επιπλέον οι κόμβοι αυτοί έχουν το πολύ ένα παιδί. Πρόκειται για τον προηγούμενο και τον επόμενο κόμβο του κόμβου p στην ενδοδιατεταγμένη



Σχήμα 6-5: Διαγραφή κόμβου από δυαδικό δέντρο αναζήτησης.
Αν ο κόμβος έχει ένα μόνο παιδί, αυτό συνδέεται στο κατάλληλο υπόδεντρο του πατέρα του.
Στο επάνω μέρος του σχήματος παρουσιάζονται οι τροποποιήσεις που πρέπει να γίνουν για τη διαγραφή του κόμβου που περιέχει το αντικείμενο **L**. Κάτω παρουσιάζεται το δυαδικό δέντρο αναζήτησης που προκύπτει.



Σχήμα 6-6: Διαγραφή κόμβου από δυαδικό δέντρο αναζήτησης.
Αν ο κόμβος έχει δύο παιδιά τότε το αντικείμενο που περιέχει αντικαθίσταται από το αντικείμενο του επόμενου (ή προηγούμενου) κόμβου στην ενδοδιατεταγμένη ακολουθία, έστω r. Στη συνέχεια, διαγράφεται από το δέντρο ο κόμβος r ο οποίος έχει το πολύ ένα παιδί.
Στο επάνω μέρος του σχήματος παρουσιάζονται οι τροποποιήσεις που πρέπει να γίνουν για τη διαγραφή του κόμβου που περιέχει το αντικείμενο **H**. Το αντικείμενο του κόμβου αυτού αντικαθίσταται από αυτό του ενδοδιατεταγμένου επόμενου (ο κόμβος με το αντικείμενο **I**) ο οποίος στη συνέχεια διαγράφεται από το δέντρο.
Κάτω παρουσιάζεται το δυαδικό δέντρο αναζήτησης που προκύπτει.

ακολουθία. Αν ο κόμβος p έχει και αριστερό και δεξί υπόδεντρο, οι κόμβοι αυτοί είναι αντίστοιχα το *δεξιότερο φύλλο του αριστερού υποδέντρου* και το *αριστερότερο φύλλο του δεξιού υποδέντρου*.

Συνοψίζοντας τα παραπάνω με τη βοήθεια των Σχημάτων 6-5 και 6-6, ο αλγόριθμος για την διαγραφή ενός κόμβου από ένα δυαδικό δέντρο αναζήτησης περιλαμβάνει τις παρακάτω περιπτώσεις.

- Αν ο κόμβος προς διαγραφή είναι φύλλο απλώς αποσυνδέεται από το δέντρο (αντικαθίσταται από εξωτερικό κόμβο).
- Αν ο κόμβος προς διαγραφή έχει ένα και μοναδικό παιδί, τότε ο κόμβος αυτός αντικαθιστά τον πατέρα του (βλ. Σχήμα 6-5).
- Αν ο κόμβος έχει δύο παιδιά τότε το στοιχείο του αντικαθίσταται από το στοιχείο του επόμενου κόμβου στην ενδοδιατεταγμένη ακολουθία, ο οποίος στη συνέχεια διαγράφεται όπως στην πρώτη ή δεύτερη περίπτωση (βλ. Σχήμα 6-6).

Αν τώρα θέλουμε να διαγράψουμε ένα κόμβο που περιέχει ένα συγκεκριμένο αντικείμενο, δεν έχουμε παρά να αναζητήσουμε τον κόμβο αυτό. Αν βρεθεί, η διαγραφή του γίνεται με βάση τον αλγόριθμο που μόλις περιγράψαμε.

6.1.2 Αποτελεσματικότητα

Οι βασική λειτουργία ενός δυαδικού δέντρου αναζήτησης είναι ο εντοπισμός του κόμβου στον οποίο βρίσκεται αποθηκευμένο ένα αντικείμενο. Η εισαγωγή νέου κόμβου βασίζεται στην αναζήτηση προκειμένου να εντοπίσει το σημείο εισαγωγής. Η διαγραφή βασίζεται επίσης στην αναζήτηση για τον εντοπισμό του κόμβου προς διαγραφή.

Έχουμε ήδη αναφέρει πως η αποτελεσματικότητα της αναζήτησης εξαρτάται από το σχήμα του δέντρου και συγκεκριμένα από τον αριθμό των επιπέδων που αυτό έχει. Σε ένα δυαδικό δέντρο αναζήτησης με N κόμβους το οποίο έχει βέλτιστο ύψος απαιτούνται $\lfloor \lg N \rfloor + 1$ συγκρίσεις στη χειρότερη περίπτωση, όταν δηλαδή το ζητούμενο αντικείμενο δεν είναι αποθηκευμένο στο δέντρο. Αν αντίθετα το ύψος του δέντρου είναι το μέγιστο δυνατό, απαιτούνται N συγκρίσεις αντικειμένων στη χειρότερη περίπτωση. Σε ένα τυχαίο δέντρο ωστόσο, το ύψος είναι κοντά στο βέλτιστο. Ο αναμενόμενος αριθμός συγκρίσεων σε ένα τέτοιο δέντρο είναι περίπου $1.4 \lg N$.

Θεώρημα 6-2. *Η αναζήτηση ενός αντικειμένου σε ένα τυχαίο δυαδι-*

κό δέντρο αναζήτησης απαιτεί περίπου $2\ln N \approx 1.4\lg N$ *συγκρίσεις αντικειμένων κατά μέσο όρο.*

Απόδειξη. Έστω C_N και C'_N ο αναμενόμενος αριθμός συγκρίσεων που απαιτούνται για την επιτυχή και ανεπιτυχή αντίστοιχα αναζήτηση, ενός αντικειμένου σε ένα δυαδικό δέντρο αναζήτησης με N κόμβους. Οι δύο αυτές ποσότητες σχετίζονται άμεσα με το μήκος εσωτερικού και εξωτερικού μονοπατιού. Συγκεκριμένα, αν I και E είναι αντίστοιχα το εσωτερικό και το εξωτερικό μήκος μονοπατιού ενός δέντρου με N κόμβους τότε

$$C_N = 1 + \frac{I}{N} \tag{6.1}$$

και

$$C'_N = \frac{E}{N+1}. \tag{6.2}$$

Σχεδιάζοντας τις σχέσεις (6.1) και (6.2) με βάση το Θεώρημα 5-3 αποδεικνύεται εύκολα (βλέπε Άσκηση 6-4) ότι

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1. \tag{6.3}$$

Εξάλλου, ο αριθμός συγκρίσεων που απαιτούνται για τον εντοπισμό ενός αντικειμένου σε ένα τυχαίο δυαδικό δέντρο αναζήτησης είναι κατά 1 μεγαλύτερος από τον αριθμό συγκρίσεων που απαιτήθηκαν κατά την εισαγωγή του αντικειμένου αυτού στο ίδιο δέντρο. Εφόσον μιλάμε για τυχαία δέντρα αναζήτησης, το αντικείμενο αυτό μπορεί να είχε ισοπίθανα εισαχθεί στο δέντρο πρώτο, δεύτερο, τρίτο, τελευταίο. Η εισαγωγή ενός αντικειμένου σε ένα δέντρο i κόμβων απαιτεί C'_i συγκρίσεις και κατά συνέπεια ο αναμενόμενος αριθμός συγκρίσεων για τον εντοπισμό του απαιτεί

$$C_N = 1 + \frac{C'_0 + C'_1 + \cdots + C'_{N-1}}{N} \tag{6.4}$$

συγκρίσεις. Αντικαθιστώντας την (6.3) στην (6.4) και απλουστεύοντας προκύπτει η σχέση

$$(N+1)C'_N = 1 + C'_0 + C'_1 + \cdots + C'_{N-1} \tag{6.5}$$

ή αλλιώς

$$NC'_{N-1} = 1 + C'_0 + C'_1 + \cdots + C'_{N-2} \tag{6.6}$$

Αφαιρώντας κατά μέλη την (6.6) από την (6.5) λαμβάνουμε την αναδρομική σχέση

$$C'_N = C'_{N-1} + \frac{2}{N} \tag{6.7}$$

με $C'_0 = 0$. Επιλύοντας με την τηλεσκοπική μέθοδο, λαμβάνουμε τελικά

$$C'_N = 2(H_{N+1} - 1). \tag{6.8}$$

Η (6.8) καλύπτει την περίπτωση στην οποία το ζητούμενο αντικείμενο δεν βρίσκεται στο δέντρο. Συνδυάζοντάς τη ωστόσο με την (6.3) λαμβάνουμε

$$C_N = 2\frac{N+1}{N}H_N - 3 \tag{6.9}$$

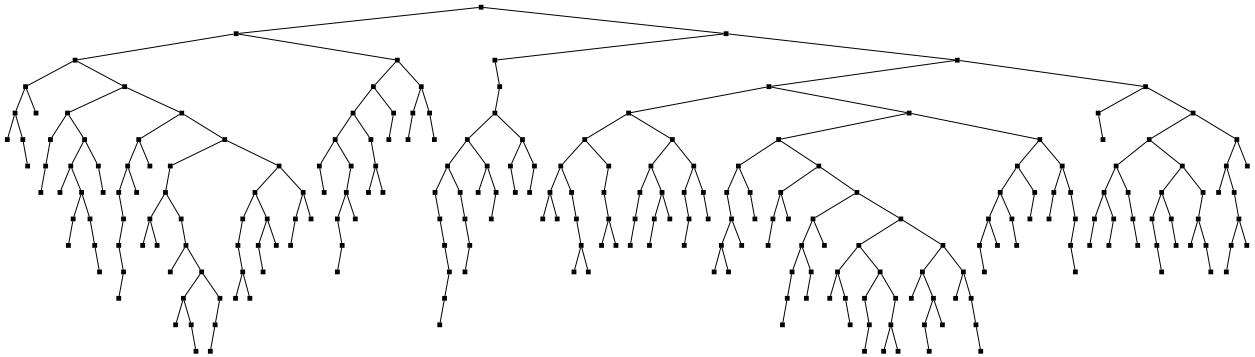
καλύπτοντας και την περίπτωση στην οποία το ζητούμενο στοιχείο βρίσκεται στο δέντρο. □

Αξίζει να παρατηρήσουμε εδώ ότι ένα δυαδικό δέντρο αναζήτησης που προκύπτει από μια ακολουθία εισαγωγών και διαγραφών τυχαίων αντικειμένων είναι ένα τυχαίο δέντρο αναζήτησης. Ένα παράδειγμα τέτοιου δέντρου παρουσιάζεται στο Σχήμα 6-7.

Για την εισαγωγή ενός κόμβου, εκτός από την αναζήτηση απαιτούνται μερικές εντολές για την σύνδεση του κόμβου. Απαιτούν ωστόσο σταθερό χρόνο και κατά συνέπεια δεν επηρεάζουν παρά ελάχιστα την απόδοση του αλγορίθμου. Η διαγραφή αντίθετα, απαιτεί σε ορισμένες περιπτώσεις τον εντοπισμό του επόμενου ή προηγούμενου κόμβου στην ενδοδιατεταγμένη ακολουθία. Στη χειρότερη περίπτωση, όταν δηλαδή ο κόμβος προς διαγραφή είναι η ρίζα του δέντρου και έχει δύο παιδιά, αυτό σημαίνει πως πρέπει να διασχίσουμε το μονοπάτι από τη ρίζα ως ένα φύλλο.

Συνοψίζοντας θα λέγαμε πως τα δυαδικά δέντρα αναζήτησης είναι ένας πολύ ικανοποιητικός μηχανισμός οργάνωσης δεδομένων εφόσον έχουμε σοβαρούς λόγους να υποθέσουμε πως είναι αρκετά τυχαία. Σε εφαρμογές στις οποίες αυτό δεν συμβαίνει, η χρήση τους οδηγεί σε χαμηλή αποτελεσματικότητα.

Επιπλέον, η χρήση δυαδικών δέντρων αναζήτησης δεν συνίσταται για εφαρμογές στις οποίες απαιτείται εγγυημένο άνω φράγμα σημαντικά χαμηλότερο από $O(N)$ χωρίς παραδοχές για την τυχαιότητα των δεδομένων. Αν για παράδειγμα μια εφαρμογή απαιτεί η

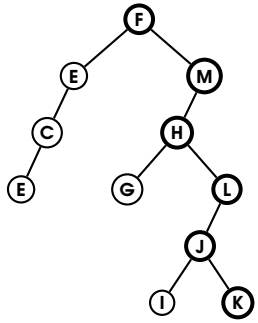


Σχήμα 6-7: Ένα δυαδικό δέντρο αναζήτησης με 256 κόμβους.
Το δέντρο αυτό έχει κατασκευαστεί με την διαδοχική εισαγωγή τυχαίων αριθμών. Στη χειρότερη περίπτωση απαιτούνται 14 συγκρίσεις για τον εντοπισμό ενός αντικειμένου.

αναζήτηση αντικειμένου να μην εκτελεί περισσότερες από $O\left(\sqrt{N}\right)$ συγκρίσεις αντικειμένων, τότε τα δυαδικά δέντρα αναζήτησης δεν είναι καλή επιλογή, καθώς χωρίς παραδοχές για την τυχαιότητα των δεδομένων δεν μπορεί να αποκλειστεί το ενδεχόμενο η αναζήτηση να εκτελεί αριθμό συγκρίσεων αντικειμένων ο οποίος είναι ανάλογος του N.

6.1.3 Επιλογή κόμβου με βάση τη θέση του.

Μέχρι στιγμής έχουμε ασχοληθεί με αναζήτηση κόμβων ενός δυαδικού δέντρου αναζήτησης με κριτήριο το αντικείμενο το οποίο αποθηκεύουν. Σε αρκετές εφαρμογές ωστόσο απαιτείται η αναζήτηση ενός κόμβου με βάση τη θέση του στην ενδοδιατεταγμένη ακολουθία κόμβων. Έστω $v_0, v_1 \ldots, v_{n-1}$ η ενδοδιατεταγμένη ακολουθία κόμβων ενός δυαδικού δέντρου αναζήτησης στην οποία θέλουμε να εντοπίσουμε τον κόμβο v_k, $0 \leq k < n$. Ένας προφανής τρόπος είναι να εντοπίσουμε τον v_0 ο οποίος είναι ο αριστερότερος κόμβος του δέντρου και στην συνέχεια καλώντας διαδοχικά τη μέθοδο succ να φτάσουμε στον κόμβο v_k. Η υλοποίηση αυτού του αλγορίθμου δίνεται στο παρακάτω απόσπασμα κώδικα.



Σχήμα 6-8: Επιλογή κόμβου με βάση τη θέση του στην ενδοδιατεταγμένη ακολουθία δυαδικού δέντρου αναζήτησης.
Το σχήμα παρουσιάζει τους κόμβους που εξετάζει ο αλγόριθμος επιλογής κατά την αναζήτηση του ένατου κόμβου της ενδοδιατεταγμένης ακολουθίας του δέντρου.

Κώδικας 6.1: Εντοπισμός κόμβου δυαδικού δέντρου με βάση τη θέση του στην ενδοδιατεταγμένη ακολουθία

```
public Object select(int k) {
   if (k < 0 || k >= size) throw new IndexOutOfBoundsException();
   Node p = root;
```

```
   for (; p.left != null; p = p.left);
   for (int i=0; i < k; ++i, p = succ(p));
   return p.element;
}
```

Ο αλγόριθμος αυτός ωστόσο δεν είναι ο αποτελεσματικότερος που θα μπορούσαμε να σκεφτούμε καθώς δεν λαμβάνει υπόψη τις ιδιότητες των δυαδικών δέντρων αναζήτησης. Ας πάρουμε για παράδειγμα το δυαδικό δέντρο αναζήτησης του Σχήματος 6-8. Ο κόμβος που περιέχει το αντικείμενο **κ** είναι ο κόμβος v_8 της ενδοδιατεταγμένης ακολουθίας. Επειδή ωστόσο το αριστερό υπόδεντρο της ρίζας του δέντρου έχει 3 κόμβους, δεν υπάρχει λόγος να ασχοληθούμε με το αριστερό υπόδεντρο. Ο κόμβος που αναζητούμε είναι στο δεξί υπόδεντρο και συγκεκριμένα είναι ο κόμβος $v_{k-3-1} = v_4$ της ενδοδιατεταγμένης ακολουθίας του υποδέντρου αυτού.

Αναλυτικά ο αλγόριθμος εντοπισμού του κόμβου v_k της ενδοδιατεταγμένης ακολουθίας ενός δυαδικού δέντρου αναζήτησης έχει ως εξής.

- Ξεκινάμε από τη ρίζα του δέντρου και συμβολίζουμε με m το πλήθος των κόμβων στο αριστερό υπόδεντρο της ρίζας.
- Αν $m = k$ τότε ο κόμβος που ψάχνουμε είναι η ρίζα.
- Αν $m > k$ τότε ο κόμβος που ψάχνουμε βρίσκεται στο αριστερό υπόδεντρο. Συνεχίζουμε με τον ίδιο τρόπο από τη ρίζα του αριστερού υποδέντρου.
- Αν $m < k$ τότε ο ζητούμενος κόμβος βρίσκεται στο δεξί υπόδεντρο. Συνεχίζουμε με τον ίδιο τρόπο από τη ρίζα του δεξιού υποδέντρου, αναζητώντας τον κόμβο v_{k-m-1} της ενδοδιατεταγμένης ακολουθίας του δεξιού υποδέντρου.

Η υλοποίηση αυτού του αλγορίθμου η οποία δίνεται παρακάτω (Κώδικας 6.2) προϋποθέτει πως κάθε κόμβος του δυαδικού δέντρου αναζήτησης έχει μια μεταβλητή `ncount` η οποία αποθηκεύει το πλήθος των κόμβων του δέντρου του οποίου ο κόμβος αυτός αποτελεί ρίζα. Επιπλέον απαιτείται η ενημέρωση της μεταβλητής αυτής κατά την εισαγωγή και διαγραφή κόμβων από το δέντρο (βλ. Άσκηση 6-5).

Κώδικας 6.2: Εντοπισμός κόμβου δυαδικού δέντρου αναζήτησης με βάση τη θέση του στην ενδοδιατεταγμένη ακολουθία

```
public Object select(int k) {
   if (k < 0 || k >= size) throw new IndexOutOfBoundsException();
   for (Node p = root;;) {
```

```
      int m = p.left != null ? p.left.ncount : 0;
      if (m == k) {
         return p.element;
      }
      else if (m > k) {
         p = p.left;
      }
      else {
         k = k-m-1;
         p = p.right;
      }
   }
}
```

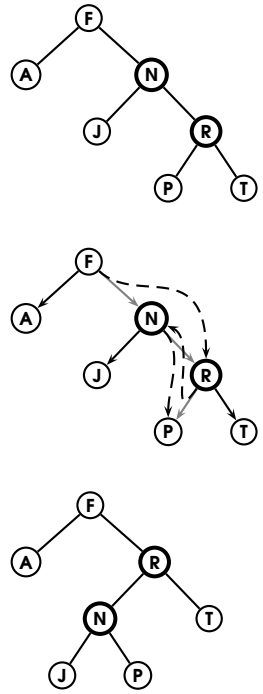
Σε εφαρμογές που η ελαχιστοποίηση των απαιτήσεων μνήμης είναι σημαντικότερη από την ταχύτητα, ο ακολουθιακός αλγόριθμος (Κώδικας 6.1) είναι προτιμότερη επιλογή.

6.1.4 Περιστροφές και εισαγωγή στη ρίζα

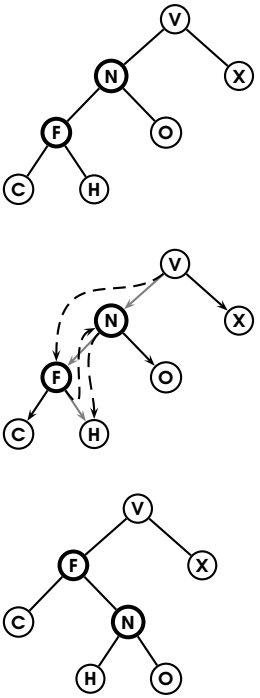
Καθώς το σχήμα ενός δυαδικού δέντρου αναζήτησης όπως αυτό διαμορφώνεται με την εισαγωγή και διαγραφή κόμβων με βάση τους αλγορίθμους που είδαμε στην Ενότητα 6.1.1 καθορίζει αποφασιστικά την αποτελεσματικότητά του, είναι αναγκαίο να εισάγουμε δομικούς μετασχηματισμούς που βελτιώνουν το σχήμα του.

Όλοι οι αλγόριθμοι δομικής τροποποίησης δυαδικών δέντρων αναζήτησης βασίζονται στη θεμελιώδη λειτουργία της *περιστροφής κόμβου*, ενός απλού τοπικού μετασχηματισμού που δεν παραβιάζει τη συνθήκη ορισμού των δυαδικών δέντρων αναζήτησης. Η εύστοχη χρήση του μετασχηματισμού αυτού σε ορισμένες περιπτώσεις απλώς βελτιώνει την αναμενόμενη αποτελεσματικότητα των λειτουργιών αναζήτησης εισαγωγής και διαγραφής, ενώ σε ορισμένες άλλες περιπτώσεις εγγυάται τη βέλτιστη αποτελεσματικότητά τους.

Η αριστερή περιστροφή ενός κόμβου p εμπλέκει τον ίδιο και το δεξί του παιδί, έστω r. Το αριστερό υπόδεντρο του κόμβου r γίνεται δεξί παιδί του κόμβου p και ο p γίνεται αριστερό παιδί του r. Η δεξιά περιστροφή ενός κόμβου p εμπλέκει τον p και το αριστερό του παιδί, έστω l. Το δεξί υπόδεντρο του κόμβου l γίνεται αριστερό παιδί του κόμβου p και ο p γίνεται δεξί παιδί του l. Οι μετασχηματισμοί αυτοί παρουσιάζονται στα Σχήματα 6-9 και 6-10 αντίστοιχα.



Σχήμα 6-9: Αριστερή περιστροφή κόμβου.
Στο επάνω μέρος του σχήματος φαίνεται το δέντρο πριν από την περιστροφή του κόμβου που περιέχει το αντικείμενο **N**. Στο κεντρικό μέρος του σχήματος υποδεικνύονται οι συνδέσεις που πρέπει να τροποποιηθούν, ενώ στο κάτω μέρος του σχήματος απεικονίζεται το δέντρο που προκύπτει μετά την περιστροφή.



Σχήμα 6-10: Δεξιά περιστροφή κόμβου. Στο επάνω μέρος του σχήματος φαίνεται το δέντρο πριν από την περιστροφή του κόμβου που περιέχει το αντικείμενο **N**. Στο κεντρικό μέρος του σχήματος υποδεικνύονται οι συνδέσεις που πρέπει να τροποποιηθούν, ενώ στο κάτω μέρος του σχήματος απεικονίζεται το δέντρο που προκύπτει μετά την περιστροφή.

Σε κάθε περίπτωση το αποτέλεσμα της περιστροφής είναι να ανεβάσουμε το δεξί (αν πρόκειται για αριστερή περιστροφή) ή το αριστερό (αν πρόκειται για δεξιά περιστροφή) παιδί του *αξονικού κόμβου* ένα επίπεδο πλησιέστερα στη ρίζα του δέντρου. Ταυτόχρονα ο αξονικός κόμβος (pivot node) κατεβαίνει ένα επίπεδο προς τα κάτω.

Εκτελώντας κατάλληλες συνεχόμενες περιστροφές μπορούμε με αυτό τον τρόπο να μεταφέρουμε οποιοδήποτε κόμβο στη ρίζα του δυαδικού δέντρου αναζήτησης. Αν εκτελέσουμε αυτή τη διαδικασία για ένα κόμβο n που μόλις έχει εισαχθεί σε ένα δυαδικό δέντρο αναζήτησης, τότε μιλάμε για εισαγωγή του n στη ρίζα του δέντρου. Ο αλγόριθμος περιλαμβάνει τα παρακάτω βήματα.

- Εισάγουμε τον νέο κόμβο n αντικαθιστώντας ένα εξωτερικό κόμβο.
- Θέτουμε `p = n.parent`.
- Εφόσον `p != null`, περιστρέφουμε τον p αριστερά (αν είναι δεξί παιδί του πατέρα του) ή δεξιά (αν είναι αριστερό παιδί του πατέρα του) και θέτουμε `p = p.parent`. Επαναλαμβάνουμε το βήμα αυτό μέχρις ότου `p == null` οπότε ο κόμβος n θα βρίσκεται στη ρίζα του δέντρου.

Η υλοποίηση του αλγορίθμου περιλαμβάνεται στην κλάση `BinarySearchTree`.

6.1.5 Υλοποίηση

Κώδικας 6.3: Η κλάση `BinarySearchTree`

```
1  package aueb.util.imp;
2
3  import aueb.util.api.AbstractCollection;
4  import aueb.util.api.Comparator;
5  import aueb.util.api.DefaultComparator;
6  import aueb.util.api.Enumerator;
7
8  /**
9   * Collection implementation using a binary search tree structure.
10  */
11 public class BinarySearchTree extends AbstractCollection {
12     /**
13      * Each node holds four references; left, right, parent and element.
14      */
15     static class Node {
```

```
16        public Node left, parent, right;
17        public Object element;
18        protected Node(Object element) {
19           this.element = element;
20        }
21        protected void unlink() {
22           element = null;
23           parent = left = right = null;
24        }
25     }
26     /**
27      * Tree enumerator. Relies on {@link BinarySearchTree#succ(Node)}
28      * and {@link BinarySearchTree#pred(Node)}
29      */
30     protected class BSTEnumerator implements Enumerator {
31        /**
32         * The node that will be returned by {@link #next()}.
33         */
34        private Node node;
35        /**
36         * Default constructor. Positions the new instance on
37         * the leftmost leaf of the enclosing tree.
38         */
39        public BSTEnumerator() {
40           if (root == null) return;
41           for (node = root; node.left != null; node = node.left);
42        }
43        /**
44         * @see aueb.util.api.Enumerator#hasNext()
45         */
46        public boolean hasNext() {
47           return node != null;
48        }
49        /**
50         * @see aueb.util.api.Enumerator#next()
51         */
52        public Object next() {
53           if (node == null) throw new IllegalStateException();
54           Node p = node;
55           node = succ(node);
56           return p.element;
57        }
58     }
59     /**
60      * The root of the tree.
61      */
```

```
62    protected Node root;
63    /**
64     * The number of nodes in the tree.
65     */
66    protected int size;
67    /**
68     * The comparison function.
69     */
70    protected Comparator cmp;
71    /**
72     * Default constructor.
73     */
74    public BinarySearchTree() {
75       this(new DefaultComparator());
76    }
77    /**
78     * Parametrized constructor that uses a given comparison function.
79     * @param cmp The comparison function to use.
80     */
81    public BinarySearchTree(Comparator cmp) {
82       this.size = 0;
83       this.cmp = cmp;
84    }
85    /**
86     * @see aueb.util.api.Collection#add(java.lang.Object)
87     */
88    public boolean add(Object element) {
89       return position(element) != null;
90    }
91    /**
92     * @see aueb.util.api.Collection#remove(java.lang.Object)
93     */
94    public boolean remove(Object element) {
95       Node n = locate(element);
96       if (n == null) return false;
97       disconnect(n);
98       return true;
99    }
100   /**
101    * @see aueb.util.api.Collection#size()
102    */
103   public int size() {
104      return size;
105   }
106   /**
107    * @see aueb.util.api.Collection#contains(java.lang.Object)
```

```
108      */
109     public boolean contains(Object element) {
110        return locate(element) != null;
111     }
112     /**
113      * @see aueb.util.api.Collection#clear()
114      */
115     public void clear() {
116        root = null;
117        size = 0;
118     }
119     /**
120      * @see aueb.util.api.Collection#enumerator()
121      */
122     public Enumerator enumerator() {
123        return new BSTEnumerator();
124     }
125     /**
126      * Locates the node a given element is stored at.
127      * @param element The element to search for.
128      * @return The node that hosts element, or null if element wasn't found.
129      */
130     protected Node locate(Object element) {
131        Node p = root;
132        while (p != null) {
133           // Compare element with the element in the current subtree
134           int result = cmp.compare(element, p.element);
135           if (result == 0) {
136              break;
137           }
138           // Go left or right based on comparison result
139           p = result < 0 ? p.left : p.right;
140        }
141        return p;
142     }
143     /**
144      * Inserts an element in the tree and returns the new node.
145      * @param element The element to insert
146      * @return The node that hosts element, or null if element wasn't found.
147      */
148     protected Node position(Object element) {
149        if (element == null) throw new IllegalArgumentException();
150        Node n = root;
151        Node p = null;
152        int result = 0;
153        while (n != null) {
```

```
154             // Compare element with the element in the current subtree
155             result = cmp.compare(element, n.element);
156             if (result == 0) return null;
157             // Go left or right based on comparison result
158             // Keep a reference in the last non-null node encountered
159             p = n;
160             n = result < 0 ? n.left : n.right;
161          }
162          // Create and connect a new node
163          Node node = newNodeInstance(element);
164          node.parent = p;
165          // The new node must be a left child of p
166          if (result < 0) {
167             p.left = node;
168          }
169          // The new node must be a right child of p
170          else if (result > 0) {
171             p.right = node;
172          }
173          // The tree is empty; root must be set
174          else {
175             root = node;
176          }
177          ++size;
178          return node;
179       }
180       /**
181        * Removes a given node from the tree.
182        * @throws NullPointerException if p is null.
183        * @param p The node to remove.
184        */
185       protected void disconnect(Node p) {
186          // If p has two children locate its successor, then remove it
187          if (p.left != null && p.right != null) {
188             Node succ = succ(p);
189             p.element = succ.element;
190             p = succ;
191          }
192          Node parent = p.parent;
193          Node child = p.left != null ? p.left : p.right;
194          // The root-node is being removed
195          if (parent == null) {
196             root = child;
197          }
198          // Bypass p
199          else if (p == parent.left) {
```

```
200            parent.left = child;
201         }
202         else {
203            parent.right = child;
204         }
205         if (child != null) {
206            child.parent = parent;
207         }
208         // Dispose p
209         p.unlink();
210         --size;
211      }
212      /**
213       * Performs a left rotation.
214       * @param pivot The node to rotate.
215       */
216      protected void rotateLeft(Node pivot) {
217         Node parent = pivot.parent;
218         Node child = pivot.right;
219         if (parent == null) {
220            root = child;
221         }
222         else if (parent.left == pivot) {
223            parent.left = child;
224         }
225         else {
226            parent.right = child;
227         }
228         child.parent = pivot.parent;
229         pivot.parent = child;
230         pivot.right = child.left;
231         if (child.left != null) child.left.parent = pivot;
232         child.left = pivot;
233      }
234      /**
235       * Performs a right rotation.
236       * @param pivot The node to rotate.
237       */
238      protected void rotateRight(Node pivot) {
239         Node parent = pivot.parent;
240         Node child = pivot.left;
241         if (parent == null) {
242            root = child;
243         }
244         else if (parent.left == pivot) {
245            parent.left = child;
```

```
246         }
247         else {
248            parent.right = child;
249         }
250         child.parent = pivot.parent;
251         pivot.parent = child;
252         pivot.left = child.right;
253         if (child.right != null) child.right.parent = pivot;
254         child.right = pivot;
255      }
256      /**
257       * Locates the inorder successor of a ginen node.
258       * @param q The node whose successor to locate
259       * @throws NullPointerException if q is null
260       * @return The successor of q, or null if q is the last node
261       */
262      protected Node succ(Node q) {
263         // The successor is the leftmost leaf of q's right subtree
264         if (q.right != null) {
265            Node p = q.right;
266            while (p.left != null) p = p.left;
267            return p;
268         }
269         // The successor is the nearest ancestor on the right
270         else {
271            Node p = q.parent;
272            Node ch = q;
273            while (p != null && ch == p.right) {
274               ch = p;
275               p = p.parent;
276            }
277            return p;
278         }
279      }
280      /**
281       * Locates the inorder predecessor of a ginen node.
282       * @param q The node whose predecessor to locate
283       * @return The predecessor of q, or null if q is the first node
284       */
285      protected Node pred(Node q) {
286         // The successor is the rightmost leaf of q's left subtree
287         if (q.left != null) {
288            Node p = q.left;
289            while (p.right != null) p = p.right;
290            return p;
291         }
```

```
292         // The successor is the nearest ancestor on the right
293         else {
294            Node p = q.parent;
295            Node ch = q;
296            while (p != null && ch == p.left) {
297               ch = p;
298               p = p.parent;
299            }
300            return p;
301         }
302      }
303      /**
304       * Factory method that creates node instances. May be overriden
305       * by subclasses to provide with the appropriate node type.
306       * @param element The element to store in the new node.
307       * @throws IllegalArgumentException if element is null.
308       * @return The node created.
309       */
310      protected Node newNodeInstance(Object element) {
311         if (element == null) throw new IllegalArgumentException();
312         return new BinarySearchTree.Node(element);
313      }
314   }
```

Ασκήσεις

6-1 Σχεδιάστε το δυαδικό δέντρο αναζήτησης που προκύπτει από την εισαγωγή των αντικειμένων **S F G E L V N G E A** σε ένα κενό δυαδικό δέντρο αναζήτησης.

6-2 Εκτελέστε τις λειτουργίες **-H -G -W -J +P +O -D +V -W +Z** —το σύμβολο **+** υποδεικνύει εισαγωγή ενώ το σύμβολο - διαγραφή αντικειμένου— στο δέντρο του Σχήματος 6-10 και σχεδιάστε το δυαδικό δέντρο αναζήτησης που προκύπτει.

6-3 Περιγράψτε και στη συνέχεια υλοποιείστε ένα αναδρομικό αλγόριθμο εισαγωγής αντικειμένου σε ένα δυαδικό δέντρο αναζήτησης.

6-4 Αποδείξτε ότι

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1$$

όπου C_N και C'_N είναι ο αναμενόμενος αριθμός συγκρίσεων για μια επιτυχή και ανεπιτυχή αντίστοιχα αναζήτηση αντικειμένου σε ένα τυχαίο δέντρο αναζήτησης.

6-5 Περιγράψτε και στη συνέχεια υλοποιείστε τους αλγορίθμους εισαγωγής και διαγραφής σε δυαδικό δέντρο αναζήτησης όταν κάθε κόμβος

p περιλαμβάνει το πεδίο ncount το οποίο αποθηκεύει το πλήθος των κόμβων που περιλαμβάνει το δέντρο με ρίζα τον κόμβο p.

6-6 Βελτιώστε τον Κώδικα 6.2 ώστε να εκτελεί μονάχα μία σύγκριση των μεταβλητών m και index σε κάθε εκτέλεση του βρόχου while.

6-7 Αποδείξτε τυπικά πως η περιστροφή ενός κόμβου δυαδικού δέντρου αναζήτησης, παράγει ένα δυαδικό δέντρο αναζήτησης.

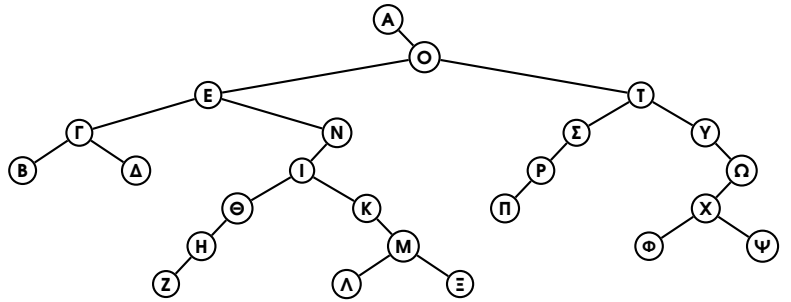
6-8 Υλοποιείστε ένα δρομέα για την κλάση BinarySearchTree.

6.2 Προσαρμοστικά δέντρα αναζήτησης

Σε πολλές εφαρμογές ορισμένα αντικείμενα τείνουν να αναζητούνται με σημαντικά μεγαλύτερη συχνότητα από τα υπόλοιπα. Τυπικό παράδειγμα τέτοιας εφαρμογής αποτελεί το σύνολο δεσμευμένων λέξεων μιας γλώσσας προγραμματισμού. Στα περισσότερα προγράμματα Java για παράδειγμα, λέξεις όπως public, private, final έχουν μεγαλύτερη συχνότητα εμφάνισης από λέξεις όπως transient, synchronized, switch, volatile. Ένα άλλο παράδειγμα είναι τα γράμματα του αλφαβήτου μιας φυσικής γλώσσας τα οποία εμφανίζουν σημαντικά ανομοιόμορφη κατανομή συχνότητας. Το ίδιο συμβαίνει και με ένα μικρό υποσύνολο 30-50 λέξεων κάθε φυσικής γλώσσας οι οποίες απαντώνται στα κείμενα με σημαντικά υψηλότερη συχνότητα από τις υπόλοιπες λέξεις.

Σε τέτοιες περιπτώσεις μπορούμε να επιτύχουμε βέλτιστο χρόνο αναζήτησης αν κατασκευάσουμε ένα *βέλτιστο δυαδικό δέντρο αναζήτησης*. Ένας αλγόριθμος για την κατασκευή του βέλτιστου δυαδικού δέντρου αναζήτησης παρουσιάζεται στο [7] και βασίζεται σε δυναμικό προγραμματισμό. Δεν θα μας απασχολήσει εδώ καθώς η τεχνική απαιτεί τα αντικείμενα και οι συχνότητες εμφάνισής τους είναι γνωστά εκ των προτέρων. Για το λόγο αυτό δεν ενδείκνυται για υλοποιήσεις γενικού σκοπού.

Αν τα αντικείμενα και οι συχνότητα εμφάνισης καθενός ήταν γνωστά τότε μια σημαντική βελτίωση της αποτελεσματικότητας της αναζήτησης θα μπορούσε να επιτευχθεί αν κατασκευάζαμε ένα δυαδικό δέντρο εισάγοντας τα αντικείμενα σε φθίνουσα διάταξη της συχνότητας αναζήτησης. Έτσι το αντικείμενο με τη μεγαλύτερη συχνότητα εμφάνισης θα είναι στη ρίζα του δέντρου, το επόμενο αντικείμενο θα είναι παιδί της ρίζας κ.ο.κ. Ένα τέτοιο δυαδικό δέντρο



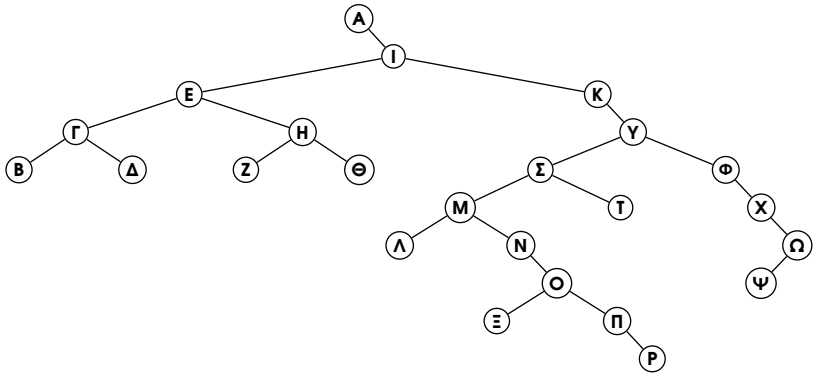
Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ
11,7	0,7	2,1	1,6	8,1	0,3	3,1	3,2	7,7	4,4	2,9	4,4
Ν	**Ξ**	**Ο**	**Π**	**Ρ**	**Σ**	**Τ**	**Υ**	**Φ**	**Χ**	**Ψ**	**Ω**
7,8	0,6	9,7	4,1	5,1	5,2	8,8	4,2	1,2	1,4	0,1	1,6

Σχήμα 6-11: Ένα δυαδικό δέντρο αναζήτησης με βάση τη συχνότητα εμφάνισης των γραμμάτων του ελληνικού αλφαβήτου.
Τα γράμματα έχουν εισαχθεί στο δέντρο κατά φθίνουσα διάταξη της συχνότητας εμφάνισης με βάση τον πίνακα στο κάτω μέρος του σχήματος. Οι συχνότητες εμφάνισης που αναφέρονται στον πίνακα έχουν προκύψει από ανάλυση μέρους του κειμένου αυτού.
Αν και το σχήμα του δέντρου δεν είναι βέλτιστο, στη συγκεκριμένη εφαρμογή έχει μεγαλύτερη αναμενόμενη αποτελεσματικότητα.

αναζήτησης παρουσιάζεται στο Σχήμα 6-11 για τα γράμματα του ελληνικού αλφαβήτου. Αν και ένα τέτοιο δέντρο δεν έχει κατ' ανάγκη βέλτιστη αποτελεσματικότητα, επιτυγχάνει καλύτερη επίδοση αναζήτησης από ότι ένα τυχαίο δυαδικό δέντρο αναζήτησης. Το πρόβλημα της εκ των προτέρων γνώσης της κατανομής των πιθανοτήτων αναζήτησης μπορεί μάλιστα να παρακαμφθεί χωρίς επιπλοκές.

Αν εκ των προτέρων γνωρίζουμε πως η κατανομή πιθανότητας αναζήτησης για τα αντικείμενα του δέντρου δεν είναι ομοιόμορφη, τότε μπορούμε να αναγκάσουμε τα αντικείμενα τα οποία αναζητούνται συχνά να συγκεντρωθούν κοντά στη ρίζα του δέντρου και έτσι να επιτύχουμε τον αρχικό μας σκοπό. Αρκεί κάθε φορά που έχουμε μια επιτυχή αναζήτηση, να μεταφέρουμε μέσω μιας περιστροφής τον αντίστοιχο κόμβο ένα επίπεδο πλησιέστερα στη ρίζα. Έπειτα από μερικές αναζητήσεις, το δέντρο θα έχει προσαρμόσει το σχήμα του ώστε να συγκλίνει σε αυτό που περιγράψαμε προηγουμένως χωρίς να υπάρχει απαίτηση να γνωρίζουμε εκ των προτέρων τη συχνότητα αναζήτησης κάθε αντικειμένου.
Η προσαρμογή ενός δυαδικού δέντρου έχει έδαφος για εφαρμογή και στις αποτυχημένες προσπάθειες αναζήτησης. Στις περιπτώσεις αυτές μπορούμε να μεταφέρουμε ένα επίπεδο προς τα πάνω, τον τελευταίο εσωτερικό κόμβο του μονοπατιού αναζήτησης (βλ. Άσκηση 6-10).

Σε ορισμένες άλλες εφαρμογές, μερικά αντικείμενα είναι πολύ



Σχήμα 6-12: Πειραματική μελέτη της συμπεριφοράς προσαρμοστικού δέντρου αναζήτησης.
Το δέντρο του σχήματος έχει προκύψει από την αναζήτηση των χαρακτήρων ενός μικρού αποσπάσματος κειμένου. Αρχικά τα γράμματα είχαν εισαχθεί στο δέντρο σε αύξουσα διάταξη ώστε πριν από οποιαδήποτε αναζήτηση το δέντρο να έχει τη χειρότερη δυνατή μορφή. Παρά το γεγονός αυτό, παρατηρούμε ότι μετά από τις αναζητήσεις, το σχήμα του δέντρου έχει προσαρμοστεί σε σημαντικό βαθμό.

δημοφιλή για ένα σχετικά μικρό χρονικό διάστημα μόνο. Τέτοια χαρακτηριστικά έχουν για παράδειγμα οι εφαρμογές επεξεργασίας δοσοληψιών. Ένα προσαρμοστικό δέντρο είναι μια πιθανή λύση για τέτοιου είδους εφαρμογές, αν εξασφαλίσουμε πως η η προσαρμογή γίνεται με περισσότερο δραστικό τρόπο. Αντί να μεταφέρουμε τον κόμβο στον οποίο σημειώθηκε μια επιτυχής αναζήτηση ένα επίπεδο πλησιέστερα στη ρίζα του δέντρου, μπορούμε να τον κάνουμε ρίζα του δέντρου, χωρίς να περιμένουμε το σχήμα του δέντρου να προσαρμοστεί βαθμιαία.

Η αναζήτηση με βάση αυτή την τεχνική απαιτεί περισσότερο χρόνο λόγω του μεγαλύτερου αριθμού περιστροφών που είναι απαραίτητες σε κάθε επιτυχή αναζήτηση, και κατά συνέπεια πρέπει να υπάρχουν πολύ ισχυρές ενδείξεις για την καταλληλότητα της χρήσης της.

Ασκήσεις

6-9 Υπολογίστε τον αναμενόμενο αριθμό συγκρίσεων αντικειμένων που απαιτείται για την επιτυχή αναζήτηση ενός γράμματος του ελληνικού αλφαβήτου στα δέντρα των σχημάτων 6-11 και 6-12. Συγκρίνετε με ένα πλήρως ισοζυγισμένο δυαδικό δέντρο αναζήτησης.

6-10 Υλοποιήστε ένα προσαρμοστικό δέντρο αναζήτησης στο οποίο η αναζήτηση λειτουργεί με βάση τους παρακάτω κανόνες.

i) Κατά τον επιτυχή εντοπισμού ενός αντικειμένου, να μεταφέρει

τον κόμβο στον οποίο αυτό εντοπίστηκε, ένα επίπεδο πλησιέστερα στη ρίζα του δέντρου.

ii) Κατά την ανεπιτυχή αναζήτηση ενός αντικειμένου, να μεταφέρει τον κόμβο στον οποίο τερματίστηκε η αναζήτηση, ένα επίπεδο πλησιέστερα στη ρίζα του δέντρου.

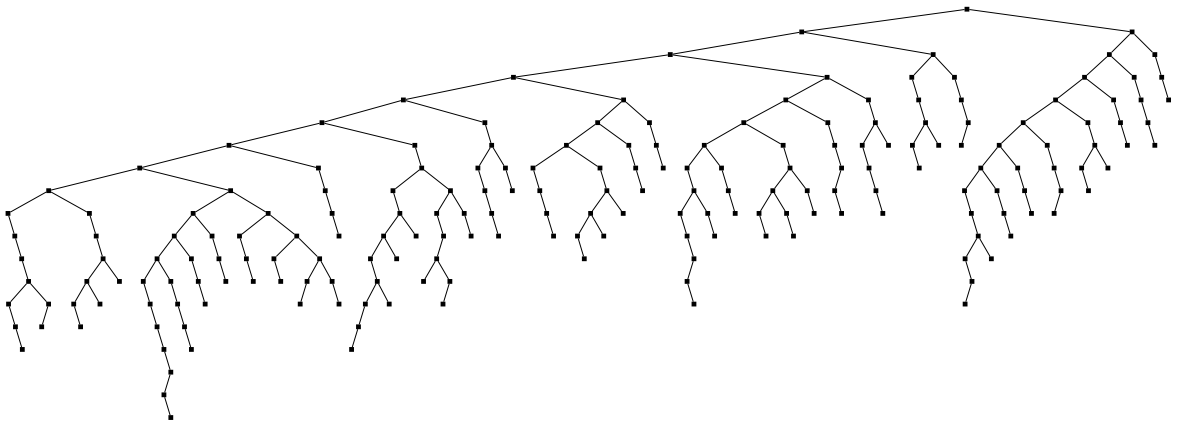
Φροντίστε ώστε η υλοποίησή σας να λειτουργεί ορθά ακόμη και σε ακραίες περιπτώσεις όπως όταν π.χ., το δέντρο είναι κενό, ή έχει ένα και μοναδικό κόμβο.

6-11 Βελτιώστε την υλοποίηση της Άσκησης 6-10 έτσι ώστε ο χρήστης να μπορεί να ενεργοποιήσει πιο δραστική προσαρμογή με τη χρήση της μεθόδου `makeRoot`.

6.3 Τυχαία δέντρα αναζήτησης

Η απόδοση ενός τυχαίου δυαδικού δέντρου αναζήτησης είναι εξαιρετική για μια πολύ ευρεία οικογένεια εφαρμογών. Μέχρι στιγμής έχουμε βασίσει την τυχαιότητα ενός δυαδικού δέντρου αναζήτησης στη σειρά με την οποία γίνονται οι εισαγωγές αντικειμένων κατά την κατασκευή του. Τα δεδομένα είναι τυχαία όταν οποιοδήποτε αντικείμενο είναι εξίσου πιθανό να βρεθεί στη ρίζα του δέντρου. Σε περιπτώσεις που δεν μπορούμε να εγγυηθούμε την τυχαιότητα των δεδομένων, μπορούμε να τροποποιήσουμε τον αλγόριθμο εισαγωγής ώστε να εισάγει τυχαιότητα στη κατασκευή του δέντρου. Η ιδέα είναι πολύ απλή: Αν εισάγοντας ένα αντικείμενο σε ένα δυαδικό δέντρο αναζήτησης με N κόμβους εξασφαλίσουμε πως το αντικείμενο έχει πιθανότητα $1/N + 1$ αν είναι η ρίζα του δέντρου και αυτό ισχύει και για τα υπόδεντρα, τότε ο αλγόριθμος εισαγωγής είναι τυχαίος.

Η υλοποίηση είναι επίσης απλή. Η μόνη διαφορά με τον κλασικό αλγόριθμο εισαγωγής είναι πως σε κάθε επίπεδο του δέντρου λαμβάνουμε μια τυχαία απόφαση για την εισαγωγή του νέου κόμβου στη ρίζα του υποδέντρου με πιθανότητα $1/m + 1$ όπου m είναι το πλήθος των κόμβων στο συγκεκριμένο υπόδεντρο. Αν το αποτέλεσμα είναι αρνητικό, τότε προχωρούμε στο αριστερό υπόδεντρο αν το αντικείμενο είναι μικρότερο από το αντικείμενο στη ρίζα του υποδέντρου ή στο δεξί υπόδεντρο αν το αντικείμενο είναι μεγαλύτερο. Αν το αποτέλεσμα είναι θετικό τότε εκτελούμε εισαγωγή του κόμβου στη ρίζα του συγκεκριμένου υποδέντρου. Η λήψη των αποφάσεων

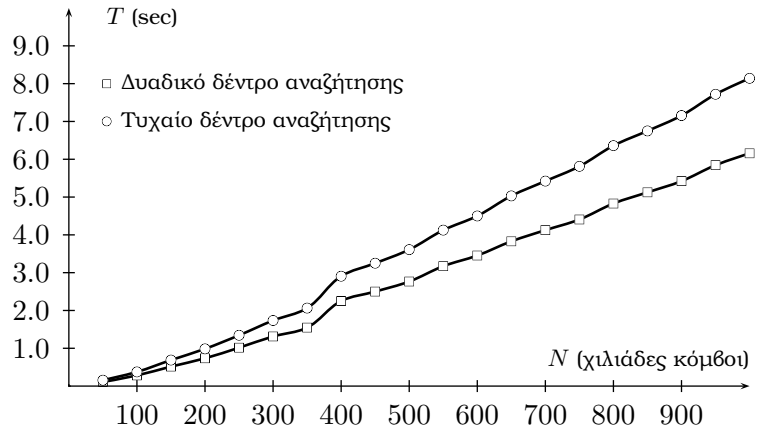


Σχήμα 6-13: Κατασκευή τυχαίου δυαδικού δέντρου αναζήτησης.
Το δέντρο του σχήματος έχει κατασκευαστεί με την εισαγωγή των αριθμών 1 ως 200 σε αύξουσα διάταξη.

μπορεί να γίνει με τη χρήση μιας γεννήτριας ψευδοτυχαίων αριθμών σε συνδυασμό με την τιμή m.

Υπάρχουν ωστόσο δύο επιπλοκές για την πρακτική εφαρμογή αυτής της μεθόδου εισαγωγής. Η πρώτη είναι πως πρέπει να γνωρίζουμε το πλήθος των κόμβων σε κάθε υπόδεντρο. Αυτό σημαίνει πως πρέπει να συντηρούμε μια μεταβλητή σε κάθε κόμβο και να την ενημερώνουμε κατάλληλα έπειτα από κάθε εισαγωγή και διαγραφή κόμβου. Η δεύτερη επιπλοκή σχετίζεται με τη γεννήτρια ψευδοτυχαίων αριθμών: Χρειαζόμαστε ένα τυχαίο αριθμό σε κάθε κόμβο που συναντάμε στο μονοπάτι αναζήτησης, και αυτό μπορεί να συνεπάγεται σημαντικό κόστος καθώς η παραγωγή ψευδοτυχαίων αριθμών δεν είναι δωρεάν.

Ευτυχώς μπορούμε να ξεπεράσουμε τα προβλήματα αυτά με ένα απλό τέχνασμα. Ας υποθέσουμε πως κατά την εισαγωγή ενός αντικειμένου σε δυαδικό δέντρο με τον κλασικό αλγόριθμο εισαγωγής συναντούμε k κόμβους στο μονοπάτι αναζήτησης. Μετά την εισαγωγή έχουμε $k + 1$ ρίζες υποδέντρων στις οποίες θα μπορούσαμε να εισάγουμε τον νέο κόμβο. Μπορούμε να λάβουμε αυτή την απόφαση παράγοντας ένα τυχαίο αριθμό i στην περιοχή $[0, k + 1)$ και στη συνέχεια να εκτελώντας i περιστροφές να φέρουμε τον νεοεισαχθέντα κόμβο στην επιθυμητή θέση. Με τη μέθοδο αυτή δεν απαιτείται να γνωρίζουμε το πλήθος των κόμβων σε κάθε υπόδεντρο του μονοπατιού αναζήτησης ενώ χρειαζόμαστε ένα μόνο τυχαίο αριθμό ανά



Σχήμα 6-14: Πειραματική συγκριτική μελέτη δυαδικών δέντρων αναζήτησης και τυχαίων δέντρων αναζήτησης.
Το σχήμα παρουσιάζει το χρόνο που απαιτείται για την εισαγωγή 1.000.000 τυχαίων αριθμών σε δυαδικό δέντρο αναζήτησης και σε τυχαίο δέντρο αναζήτησης.

εισαγωγή.

Το δέντρο του Σχήματος 6-7 επιδεικνύει τη λειτουργία ενός τυχαίου δέντρου αναζήτησης το οποίο λειτουργεί με βάση την τεχνική αυτή. Είναι φανερό πως το δέντρο που προκύπτει είναι σαφέστατα βελτιωμένο σε σχέση με τον κλασικό αλγόριθμο εισαγωγής. Επιπλέον μπορούμε να εισάγουμε τυχαιότητα και στον αλγόριθμο διαγραφής κόμβου. Στην περίπτωση που ο κόμβος προς διαγραφή έχει δύο παιδιά, μπορούμε να πάρουμε με τυχαίο τρόπο την απόφαση για το αν το αντικείμενο που είναι αποθηκευμένο στον κόμβο αυτό θα αντικατασταθεί από το αντικείμενο του επόμενου ή προηγούμενου κόμβου στην ενδοδιατεταγμένη ακολουθία.

Φυσικά τα πλεονεκτήματα των τυχαίων δέντρων αναζήτησης δεν είναι χωρίς κόστος. Ο αλγόριθμος εισαγωγής σε τυχαίο δέντρο αναζήτησης απαιτεί περισσότερη δουλειά από τον κλασικό αλγόριθμο όπως φαίνεται και στο Σχήμα 6-14. Αντίθετα η επιβάρυνση κατά τη διαγραφή είναι αμελητέα.

Ασκήσεις

6-12 Υλοποιείστε ένα τυχαίο δέντρο αναζήτησης χρησιμοποιώντας τις τεχνικές που παρουσιάζονται στην Ενότητα 6.3.

6-13 Συγκρίνετε την απόδοση της υλοποίησης της Άσκησης 6-12 με δύο άλλες υλοποιήσεις τυχαίων δέντρων αναζήτησης. Η πρώτη υλοποίηση χρησιμοποιεί το πλήθος m των κόμβων του τρέχοντος υποδέντρου

σε συνδυασμό με μια γεννήτρια ψευδοτυχαίων αριθμών προκειμένου να αποφασίσει ή όχι εισαγωγή στη ρίζα του τρέχοντος υποδέντρου. Η δεύτερη υλοποίηση είναι μια παραλλαγή της πρώτης η οποία δεν χρησιμοποιεί ψευδοτυχαίους αριθμούς αλλά την ισχύ της σχέσης $111m \bmod = 3$ για να αποφασίσει εισαγωγή στη ρίζα του τρέχοντος υποδέντρου.

ΚΕΦΑΛΑΙΟ 7

Ουρές προτεραιότητας

Στο Κεφάλαιο 2 περιγράψαμε τον τρόπο λειτουργίας των ουρών αναμονής, ενώ στα Κεφάλαια 3 και 4 μελετήσαμε τρόπους για την αποτελεσματική υλοποίησή τους. Στο κεφάλαιο αυτό θα περιγράψουμε ένα γενικότερο και ισχυρότερο αφηρημένο τύπο δεδομένων που ονομάζεται *ουρά προτεραιότητας* (priority queue) και θα αναλύσουμε εναλλακτικούς μηχανισμούς για την υλοποίησή του.

Μια ουρά προτεραιότητας είναι μια ουρά αναμονής από την οποία τα αντικείμενα εξάγονται κατά φθίνουσα σειρά *προτεραιότητας*. Η προτεραιότητα κάθε αντικειμένου είναι ένας μη-αρνητικός ακέραιος αριθμός. Οι βασικές λειτουργίες μιας ουράς προτεραιότητας είναι, όπως και σε μια ουρά αναμονής η εισαγωγή και η εξαγωγή αντικειμένου. Όταν ωστόσο εξάγουμε από μια ουρά προτεραιότητας, το αντικείμενο που διάγραφεται είναι εκείνο με τη μεγαλύτερη προτεραιότητα.

Αν και χρησιμοποιήσαμε την ουρά αναμονής στον ορισμό της ουράς προτεραιότητας, η δεύτερη δεν είναι μια ειδική περίπτωση της πρώτης· το αντίθετο μάλιστα. Μια ουρά προτεραιότητας μπορεί να λειτουργήσει ως ουρά αναμονής ή ως στοίβα ανάλογα με τον ορισμό της προτεραιότητας ενός αντικειμένου. Μπορούμε για παράδειγμα να προσομοιώσουμε τη λειτουργία μιας στοίβας χρησιμοποιώντας μια ουρά προτεραιότητας, θεωρώντας πως η προτεραιότητα κάθε αντικειμένου είναι η χρονική στιγμή κατά την οποία αυτό εισάγεται στη δομή. Με ανάλογο τρόπο, μπορούμε να προσαρμόσουμε μια ουρά προτεραιότητας ώστε να λειτουργεί ως ουρά αναμονής.

Εξαιτίας της γενικότητάς της, οι εφαρμογές αυτής της δομής δεδομένων είναι σχεδόν αμέτρητες. Ο χρονοπρογραμματισμός εκτέλεσης διεργασιών και νημάτων σε ένα υπολογιστικό σύστημα, οι εξυπηρέτηση περιφερειακών συσκευών, ο προγραμματισμός χειρουργικών τραπεζιών είναι μερικές μόνο εφαρμογές στις οποίες οι ουρές προτεραιότητας αποτελούν βασικό εργαλείο. Οι εφαρμογές των ουρών προτεραιότητας δεν εξαντλούνται ωστόσο στα συστήματα εξυπηρέτησης. Στο κεφάλαιο αυτό θα δούμε ότι μπορούμε χρησιμοποιώντας μια ουρά προτεραιότητας να ταξινομήσουμε N αντικειμένα σε χρόνο όχι περισσότερο από $O(N \lg N)$. Επίσης θα χρησιμοποιήσουμε ουρές προτεραιότητας στην υλοποίηση ενός κλασικού αλγορίθμου συμπίεσης δεδομένων.

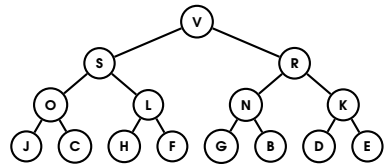
Οι βασικές λειτουργίες μιας ουράς προτεραιότητας είναι η εισαγωγή και διαγραφή αντικειμένων. Στην πράξη ωστόσο, και εξαιτίας του πλήθους των εφαρμογών που βασίζονται στη χρήση τους, οι ουρές προτεραιότητας παρέχουν ένα πιο διευρημένο ρεπερτόριο λειτουργιών:

- εισαγωγή αντικειμένου,
- εξαγωγή του αντικειμένου με τη μέγιστη (ή ελάχιστη) προτεραιότητα,
- τροποποίηση της προτεραιότητας ενός αντικειμένου,
- διαγραφή ενός αντικειμένου,
- αντικατάσταση ενός αντικειμένου από ένα άλλο,
- οργάνωση μιας συλλογής αντικειμένων σε ουρά προτεραιότητας (κατασκευή),
- συνένωση δύο ουρών προτεραιότητας σε μία.

Η αποτελεσματική υποστήριξη του συνόλου των παραπάνω λειτουργιών δεν είναι πάντα εφικτή. Αν για παράδειγμα χρησιμοποιήσουμε μια διπλοουρά, η εισαγωγή, η κατασκευή και η συνένωση υλοποιούνται πολύ αποτελεσματικά, δεν συμβαίνει το ίδιο όμως για την τροποποίηση της προτεραιότητας ενός αντικειμένου ή την εξαγωγή του αντικειμένου με τη μέγιστη προτεραιότητα. Στις επόμενες ενότητες θα παρουσιάσουμε δύο μηχανισμούς οργάνωσης που εξασφαλίζουν την αποτελεσματική υλοποίηση των περισσοτέρων λειτουργιών.

7.1 Δυαδικοί σωροί

Ο δυαδικός σωρός είναι μια οργάνωση δεδομένωνπου παρουσιάζει σημαντικά πλεονεκτήματα σε σχέση με όσες δομές έχουμε εξετάσει μέχρι στιγμής. Κατ' αρχήν υλοποιείται με πολύ συμπαγή τρόπο. Επιπλέον, οι λειτουργίες εισαγωγής και διαγραφής εκτελούνται σε χρόνο $O(\lg N)$, όπου N είναι το πλήθος των αντικειμένων της δομής.



Σχήμα 7-1: Δυαδικός σωρός.
Το αντικείμενο στη ρίζα κάθε υποδέντρου είναι μεγαλύτερο των αντικειμένων που βρίσκονται στις ρίζες του αριστερού και δεξιού υποδέντρου.

Ορισμός 7-1. Ένα δυαδικό δέντρο έχει διάταξη *δυαδικού σωρού* όταν το αντικείμενο κάθε κόμβου είναι μεγαλύτερο (έπεται σε μια δοσμένη σχέση διάταξης) των αντικειμένων που βρίσκονται στη ρίζα του αριστερού και του δεξιού του υποδέντρου.

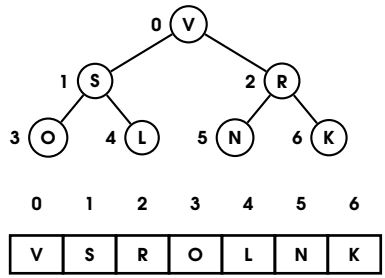
Ένα δέντρο με διάταξη δυαδικού σωρού ονομάζεται *δυαδικός σωρός* (binary heap) ή απλά σωρός. Το Σχήμα 7-1 παρουσιάζει 15 αντικείμενα οργανωμένα σε ένα δυαδικό σωρό. Η βασική ιδιότητα της οργάνωσης είναι πως το μεγαλύτερο (ή ενδεχομένως το μικρότερο, αν αντιστρέψουμε τη σημασιολογία της σχέσης διάταξης) αντικείμενο βρίσκεται στη ρίζα του δέντρου. Αυτό καθιστά τους δυαδικούς σωρούς μια ελκυστική οργάνωση για την υλοποίηση ουρών προτεραιότητας.

Αν και δενδρικές δομές, οι δυαδικοί σωροί σχεδόν ποτέ δεν υλοποιούνται με συνδεδεμένη οργάνωση κόμβων, αλλά με συστοιχία. Αυτό οφείλεται στο γεγονός ότι τα πλεονεκτήματα της συνδεδεμένης οργάνωσης, όπως για παράδειγμα η μετακίνηση ενός υποδέντρου με δύο ή τρεις αναθέσεις, δεν προσφέρουν τίποτε στους αλγορίθμους δυαδικών σωρών όπως θα δούμε παρακάτω. Έτσι, και δεδομένου ότι η συνδεδεμένη οργάνωση έχει σημαντική επίπτωση στην κατανάλωση μνήμης, μια συστοιχία είναι καταλληλότερος μηχανισμός.

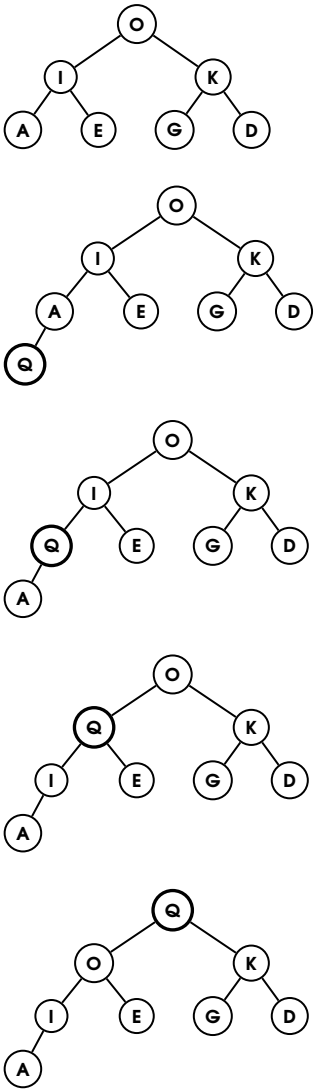
Μπορούμε να αποθηκεύσουμε ένα δυαδικό δέντρο σε μια συστοιχία εκτελώντας μια πλήρη διάσχιση κατά επίπεδα και αποθηκεύοντας τον πρώτο κόμβο στη θέση 0, το δεύτερο στη θέση 1 κ.ο.κ. όπως φαίνεται στο Σχήμα 7-2. Αν θέλουμε να είμαστε περισσότερο αυστηροί, οι κόμβοι ενός δυαδικού δέντρου αποθηκεύονται σε συνεχόμενες θέσεις μνήμης με βάση τους παρακάτω κανόνες:

i). Κάθε θέση της συστοιχίας αντιστοιχεί σε ένα κόμβο του δέντρου.

ii). Η ρίζα του δέντρου αποθηκεύεται στη θέση 0 της συστοιχίας.



Σχήμα 7-2: Οργάνωση δυαδικού σωρού σε συστοιχία.
Το αριστερό και το δεξιό υπόδεντρο του κόμβου στη θέση i (εξαιρουμένων των κόμβων του τελευταίου επιπέδου) βρίσκονται αποθηκευμένα στις θέσεις $2i+1$ και $2i+2$ αντίστοιχα. Ο πατέρας του κόμβου που βρίσκεται στη θέση $i > 0$ της συστοιχίας, είναι αποθηκευμένος στη θέση $\lfloor (i-1)/2 \rfloor$.



Σχήμα 7-3: Ανάδυση αντικειμένου σε δυαδικό σωρό.
Το σχήμα παρουσιάζει τη λειτουργία του αλγορίθμου shift up κατά την εισαγωγή του αντικειμένου **Q**.

iii). Αν ένας κόμβος του δέντρου βρίσκεται αποθηκευμένος στη θέση i της συστοιχίας, οι ρίζες του αριστερού και δεξιού του υποδέντρου βρίσκονται αποθηκευμένες στις θέσεις $2i+1$ και $2i+2$ αντίστοιχα.

iv). Ο πατέρας του κόμβου που βρίσκεται στη θέση $i > 0$ της συστοιχίας, βρίσκεται στη θέση $\lfloor (i-1)/2 \rfloor$.

Η οργάνωση αυτή, υποστηρίζει αποτελεσματικά τις περισσότερες από τις λειτουργίες μιας ουράς προτεραιότητας.

Εισαγωγή. Όπως και στα δυαδικά δέντρα αναζήτησης, όταν προσθέτουμε ένα αντικείμενο σε ένα δυαδικό σωρό πρέπει να φροντίσουμε ώστε η δομή να εξακολουθεί να ικανοποιεί τις ιδιότητες ενός δυαδικού σωρού. Η εισαγωγή αντικειμένου σε δυαδικό σωρό χωρητικότητας $2^k - 1$ αντικειμένων ο οποίος περιέχει N αντικείμενα, περιλαμβάνει δύο στάδια. Αρχικά τοποθετούμε το αντικείμενο στη θέση N και στη συνέχεια εκτελούμε τον αλγόριθμο shift up για τη θέση N.

Αλγόριθμος 7-1 (`ShiftUp`). Ανάδυση αντικειμένου σε δυαδικό σωρό. Η μεταβλητή `i` προσδιορίζει τον κόμβο του σωρού στον οποίο βρίσκεται το αντικείμενο.

1). Αν η θέση `i` είναι η ρίζα του σωρού, αν δηλαδή `i == 0`, τερματίζουμε.

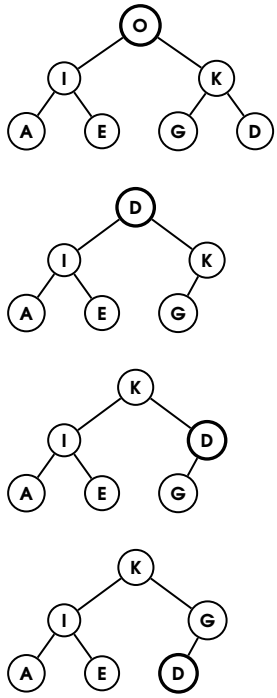
2). Θέτουμε `p = (i-1)/2` και συγκρίνουμε τα αντικείμενα στις θέσεις `p` και `i`. Αν το αντικείμενο στη θέση `i` είναι μικρότερο (ή ίσο) του αντικειμένου στη θέση `p`, τερματίζουμε, διαφορετικά συνεχίζουμε από το βήμα 3.

3). Ανταλλάσουμε αμοιβαία τα αντικείμενα στις θέσεις `i` και `p`, θέτουμε `i = p` και συνεχίζουμε από το βήμα 1.

Στο πρώτο στάδιο της διαδικασίας εισαγωγής είναι πιθανό να παραβιαστεί η ιδιότητα του σωρού. Αυτό θα συμβεί αν το νέο αντικείμενο είναι μεγαλύτερο από αντικείμενα που βρίσκονται σε κόμβους πάνω στο μονοπάτι από τη ρίζα μέχρι τον κόμβο N. Στο δεύτερο στάδιο, ο αλγόριθμος shift up μεταφέρει, εφόσον αυτό χρειάζεται, το νέο αντικείμενο προς τα πάνω μέχρι να διαπιστωθεί ότι η ιδιότητα του δυαδικού σωρού ικανοποιείται. Αν το αντικείμενο του νέου κόμ-

βου είναι μικρότερο ή ίσο του αντικειμένου του πατέρα του, τότε ο αλγόριθμος τερματίζει. Διαφορετικά ανταλλάζει αμοιβαία τα δύο αντικείμενα και συνεχίζει με τον ίδιο τρόπο προς τη ρίζα του σωρού. Κάποια στιγμή είτε το νέο αντικείμενο θα φτάσει στη ρίζα του σωρού (αν είναι μεγαλύτερο από όλα τα υπόλοιπα αντικείμενα) ή θα βρεθεί ένας κόμβος με μεγαλύτερο αντικείμενο. Ο αλγόριθμος θα τερματίσει και στις δύο περιπτώσεις καθώς ικανοποιείται η συνθήκη ορισμού του δυαδικού σωρού. Ένα παράδειγμα εκτέλεσης του αλγορίθμου εισαγωγής σε δυαδικό σωρό, ο οποίος ονομάζεται και swim καθώς ανεβάζει τον νέο κόμβο μέχρι την "επιφάνεια" (κορυφή) του σωρού εφόσον χρειαστεί, δίνεται στο Σχήμα 7-3.

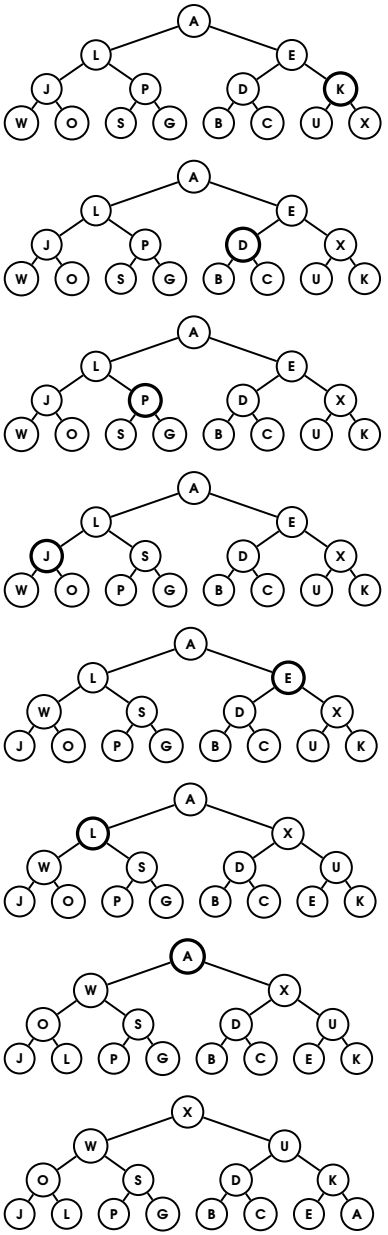
Διαγραφή. Για τη διαγραφή ενός κόμβου από δυαδικό σωρό πρέπει να αντιστρέψουμε τη διαδικασία εισαγωγής. Αρχικά αντικαθιστούμε το αντικείμενο που βρίσκεται στη ρίζα του δέντρου με το αντικείμενο που βρίσκεται στο "δεξιότερο" φύλλο του σωρού. Στη συνέχεια μεταφέρουμε το αντικείμενο αυτό προς το τελευταίο επίπεδο του δέντρου ώστε να αποκαταστήσουμε τη διάταξη σωρού που έχει πλέον παραβιαστεί. Ο Αλγόριθμος 7-2 περιγράφει τη διαδικασία για την "κατάδυση" του αντικειμένου που βρίσκεται σε ένα δοσμένο κόμβο του σωρού.



Σχήμα 7-4: Διαγραφή αντικειμένου από δυαδικό σωρό.
Το σχήμα παρουσιάζει τη λειτουργία του αλγορίθμου shift κατά τη διαγραφή του αντικειμένου **O**.

Αλγόριθμος 7-2 (`ShiftDown`). Κατάδυση αντικειμένου σε δυαδικό σωρό. Η μεταβλητή `i` προσδιορίζει τον κόμβο του σωρού στον οποίο βρίσκεται το αντικείμενο ενώ η μεταβλητή `size` το πλήθος των αντικειμένων που περιλαμβάνει ο σωρός.

1). Αν η θέση `i` είναι φύλλο, αν δηλαδή `2*i + 1 >= size`, τερματίζουμε.
2). Υπολογίζουμε τη θέση `m` του κόμβου στον οποίο βρίσκεται το μεγαλύτερο από τα παιδιά του κόμβου `i`. Αν το αντικείμενο αυτό είναι μικρότερο ή ίσο του αντικειμένου στη θέση `i`, τερματίζουμε, διαφορετικά συνεχίζουμε από το βήμα 3.
3). Αντικαθιστούμε το αντικείμενο που βρίσκεται στον κόμβο `i` με το αντικείμενο που βρίσκεται στον κόμβο `m`, θέτουμε `i = m` και συνεχίζουμε από το βήμα 1.



Σχήμα 7-5: Κατασκευή δυαδικού σωρού από κάτω προς τα πάνω.

Όπως και κατά την εισαγωγή, το πρώτο στάδιο του αλγορίθμου διαγραφής μπορεί να παραβιάσει την ιδιότητα του σωρού. Έτσι το αντικείμενο που μεταφέρθηκε στη ρίζα, βυθίζεται σταδιακά ώσπου να διαπιστωθεί η αποκατάσταση της ιδιότητας του σωρού. Ένα παράδειγμα εκτέλεσης του αλγορίθμου διαγραφής αντικειμένου από δυαδικό σωρό, ο οποίος είναι γνωστός και ως sink, παρουσιάζεται στο Σχήμα 7-4.

Αντικατάσταση. Οι αλγόριθμοι 7-1 και 7-2 μπορούν να χρησιμοποιηθούν και για την αντικατάσταση ενός αντικειμένου του δυαδικού σωρού από ένα νέο αντικείμενο. Αν το νέο αντικείμενο είναι μικρότερο από το αντικείμενο που αντικαθίσταται, τότε χρησιμοποιούμε τον αλγόριθμο shift down για να το μεταφέρουμε σε χαμηλότερα επίπεδα του σωρού. Αντίστοιχα, αν το νέο αντικείμενο είναι μεγαλύτερο από το αντικείμενο που αντικαθίσταται, το μεταφέρουμε σε υψηλότερα επίπεδα του σωρού χρησιμοποιώντας τον αλγόριθμο shift up.

Κατασκευή δυαδικού σωρού. Ο απλούστερος τρόπος για να κατασκευάσουμε ένα σωρό N αντικειμένων είναι να ξεκινήσουμε με ένα κενό σωρό και να εισάγουμε διαδοχικά σε αυτόν τα αντικείμενα. Αν τα αντικείμενα είναι αποθηκευμένα σε μια λίστα ή ένα δυαδικό δέντρο, δεν έχουμε άλλη επιλογή. Αν ωστόσο τα αντικείμενα βρίσκονται ήδη σε μια συστοιχία, τότε εκτός του ότι μπορούμε να αποφύγουμε τη σπατάλη μνήμης, μπορούμε επιπλέον να επιταχύνουμε σημαντικά τη διαδικασία κατασκευής του σωρού αν εργαστούμε με διαφορετικό τρόπο. Η διαδικασία παρουσιάζεται στο Σχήμα 7-5 και βασίζεται στο γεγονός ότι τα φύλλα ενός δυαδικού δέντρου ικανοποιούν την ιδιότητα του σωρού καθώς δεν έχουν υπόδεντρα. Επομένως δεν υπάρχει λόγος να ασχοληθούμε με τους κόμβους αυτούς. Ξεκινάμε λοιπόν από τη θέση $\lfloor (N-2)/2 \rfloor$, όπου N είναι το πλήθος των αντικειμένων, η οποία αντιστοιχεί στο δεξιότερο κόμβο του προτελευταίου επιπέδου. Αν εκτελέσουμε τον αλγόριθμο shift down στον κόμβο αυτό, θα έχουμε μετατρέψει το αντίστοιχο δέντρο σε σωρό. Συνεχίζοντας με τον ίδιο τρόπο, από τα δεξιά προς τα αριστερά και από κάτω προς τα πάνω, δημιουργώντας όλο και μεγαλύτερους σωρούς, θα καταλήξουμε στη ρίζα ολοκληρώνοντας την κατασκευή.

Μηχανισμός οργάνωσης	Εισαγωγή	Διαγραφή	Μνήμη
Μη ταξινομημένη συστοιχία	$O(1)$	$O(N)$	N
Ταξινομημένη συστοιχία	$O(N)$	$O(1)$	N
Μη ταξινομημένη συνδεδεμένη λίστα	$O(1)$	$O(N)$	$\Theta(N)$
Ταξινομημένη συνδεδεμένη λίστα	$O(N)$	$O(1)$	$\Theta(N)$
Ισοζυγισμένο δυαδικό δέντρο αναζήτησης	$O(\lg N)$	$O(\lg N)$	$\Theta(N)$
Δυαδικό δέντρο αναζήτησης	$O(N)$	$O(N)$	$\Theta(N)$
Δυαδικός σωρός	$O(\lg N)$	$O(\lg N)$	N

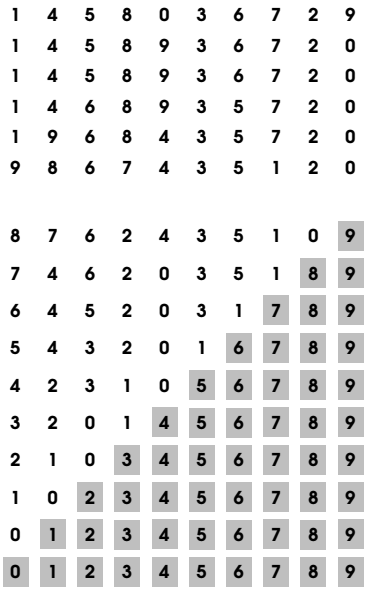
Πίνακας 7-1:

Αποτελεσματικότητα. Οι αλγόριθμοι 7-1 και 7-2 εργάζονται κατά μήκος ενός μονοπατιού από τη ρίζα προς κάποιο φύλλο. Καθώς ένας δυαδικός σωρός είναι ένα πλήρες δυαδικό δέντρο, το ύψος των εξωτερικών κόμβων είναι το πολύ $\lg N$, όπου N το πλήθος των κόμβων του σωρού (βλέπε Θεώρημα 5-4 στη σελίδα 143). Επομένως στη χειρότερη περίπτωση, τόσο η εισαγωγή όσο και η διαγραφή αντικειμένου, απαιτούν χρόνο $O(\lg N)$.

Η κατασκευή σωρού απαιτεί χρόνο $O(N \lg N)$ όταν γίνεται με διαδοχικές εισαγωγές. Ο αριθμός συγκρίσεων C_N για την κατασκευή δυαδικού σωρού N στοιχείων, δίνεται από την αναδρομική σχέση $C_N = C_{N-1} + \lg N$, με $C_1 = 0$ και με βάση την Άσκηση 1-23 είναι $C_N = O(N \lg N)$. Όταν η κατασκευή του σωρού γίνεται από κάτω προς τα πάνω, ο χρόνος που απαιτείται είναι $O(N)$ καθώς ο βρόχος εκτελείται περίπου $N/2$ φορές και σε κάθε επανάληψη εκτελείται σταθερός αριθμός συγκρίσεων.

Επιπλέον, η ποσότητα μνήμης που απαιτείται για την οργάνωση N αντικειμένων σε δυαδικό σωρό είναι ελάχιστη $-4(N+1)$ ψηφιοσυλλαβές— αφού δεν χρειάζεται να συντηρούμε αναφορές για τα παιδιά και τον πατέρα κάθε κόμβου όπως θα οφείλαμε σε ένα τυπικό δυαδικό δέντρο.

Το μειονέκτημα βεβαίως της οργάνωσης είναι η αδυναμία γρήγορου εντοπισμού ενός αντικειμένου. Αν για παράδειγμα θέλουμε να αντικαταστήσουμε ένα αντικείμενο με ένα άλλο, πρέπει πρώτα να το εντοπίσουμε ώστε να χρησιμοποιήσουμε στη συνέχεια τους αλγορίθμους shift up και shift down. Ο εντοπισμός του αντικειμένου ωστόσο απαιτεί $O(N)$ συγκρίσεις κατά μέσο όρο.



Σχήμα 7-6: Ο αλγόριθμος heap sort. Το σχήμα παρουσιάζει τον τρόπο με τον οποίο λειτουργεί ο αλγόριθμος heap sort για την ταξινόμηση της συστοιχίας ακεραίων **1 4 5 8 0 3 6 7 2 9**. Στο επάνω μέρος του σχήματος φαίνεται η διαδικασία κατασκευής σωρού, ενώ στο κάτω η διαδικασία ταξινόμησης. Η γκρι περιοχή δείχνει το τμήμα της συστοιχίας που έχει ταξινομηθεί έπειτα από κάθε επανάληψη του αλγορίθμου.

Ταξινόμηση με σωρό. Η ταξινόμηση με επιλογή είναι μια κατηγορία αλγορίθμων ταξινόμησης που ακολουθούν το παρακάτω "αλγοριθμικό σχήμα".

Αλγόριθμος 7-3 (`SelectionSort`). Ταξινόμηση συστοιχίας με επιλογή μέγιστης τιμής. Η μεταβλητή `array` είναι η συστοιχία προς ταξινόμηση.

1). Εκτελούμε το βήμα 2 για `i = array.size - 1`, ..., 1.

2). Εντοπίζουμε το μέγιστο των στοιχείων `array[0]` ... `array[i]` και το ανταλλάζουμε αμοιβαία με το στοιχείο `array[i]`.

Το κρίσιμο σημείο για την αποτελεσματικότητα κάθε αλγορίθμου που βασίζεται σε αυτό το σχήμα είναι η επιλογή του μεγίστου στοιχείου. Στην κλασική του εκδοχή, ο αλγόριθμος selection sort χρησιμοποιεί μια ακολουθιακή προσέγγιση για τον εντοπισμό του μεγίστου στοιχείου. Έτσι, ο απαιτούμενος χρόνος για την εκτέλεση του αλγορίθμου είναι $T_N = T_{N-1} + N = O\left(N^2\right)$. Αν όμως επιβάλλουμε διάταξη σωρού στην συστοιχία, ο εντοπισμός του μεγίστου είναι δωρεάν, ενώ μετά την αμοιβαία αναταλλαγή θα πρέπει να αποκαταστήσουμε τη διάταξη σωρού, διαδικασία που απαιτεί χρόνο $O\left(\lg N\right)$. Έτσι ο αλγόριθμος ταξινόμησης συνολικά, απαιτεί χρόνο $T_N = T_{N-1} + \lg N = O\left(N \lg N\right)$. Η ταξινόμηση με χρήση δυαδικού σωρού ονομάζεται heap sort και υλοποιείται από τον Αλγόριθμο 7-4. Ένα παράδειγμα εκτέλεσης του αλγορίθμου παρουσιάζεται στο Σχήμα 7-6.

Αλγόριθμος 7-4 (`HeapSort`). Ταξινόμηση με δυαδικό σωρό. Η μεταβλητή `array` είναι η συστοιχία αντικειμένων που πρέπει να ταξινομηθεί.

1). Για `i = (array.size-2)/2, ..., 0` εκτελούμε `ShiftDown(i)` (κατασκευή δυαδικού σωρού).

2). Για `i = array.size-1, ..., 1` εκτελούμε το βήμα 3.

3). Ανταλλάζουμε αμοιβαία τα στοιχεία `a[0]` και `a[i]`, και εκτελούμε `ShiftDown(0)`.

Κώδικας 7.1: Η κλάση `BinaryHeap`.

```
1 package aueb.util.imp;
2
3 import aueb.util.api.Comparator;
4 import aueb.util.api.DefaultComparator;
```

```
5  /**
6   * Implementation of a binary heap.
7   */
8  public class BinaryHeap {
9     /**
10     * Array-based heap representation
11     */
12    Object[] heap;
13    /**
14     * The number of objects in the heap
15     */
16    int size;
17    /**
18     * Comparator.
19     */
20    Comparator cmp;
21    /**
22     * Creates heap with a given capacity and default comparator.
23     * @param capacity The capacity of the heap being created.
24     */
25    public BinaryHeap(int capacity) {
26       this(capacity, new DefaultComparator());
27    }
28    /**
29     * Creates heap with a given capacity and comparator.
30     * @param capacity The capacity of the heap being created.
31     * @param cmp The comparator that will be used.
32     */
33    public BinaryHeap(int capacity, Comparator cmp) {
34       if (capacity < 1) throw new IllegalArgumentException();
35       this.heap = new Object[capacity];
36       this.cmp = cmp;
37    }
38    /**
39     * Creates a heap from an array of objects.
40     * @param array The array of objects.
41     */
42    public BinaryHeap(Object[] array) {
43       this(array, new DefaultComparator());
44    }
45    /**
46     * Creates a heap from an array using a given comparator.
47     * @param array The array of objects.
48     * @param c The comparator that will be used.
49     */
50    public BinaryHeap(Object[] array, Comparator c) {
```

```
51        size = array.length;
52        heap = (Object[]) array.clone();
53        cmp = c;
54        heap(heap, size, cmp);
55     }
56     /**
57      * Inserts an object in this heap.
58      * @throws IllegalStateException if heap capacity is exceeded.
59      * @param object The object to insert.
60      */
61     public void add(Object object) {
62        // Ensure object is not null
63        if (object == null) throw new IllegalArgumentException();
64        // Check available space
65        if (size == heap.length) throw new IllegalStateException();
66        // Place object at the next available position
67        heap[size] = object;
68        // Let the newly added object swim
69        swim(heap, size++, cmp);
70     }
71     /**
72      * Removes the object at the root of this heap.
73      * @throws IllegalStateException if heap is empty.
74      * @return The object removed.
75      */
76     public Object remove() {
77        // Ensure not empty
78        if (size == 0) throw new IllegalStateException();
79        // Keep a reference to the root object
80        Object object = heap[0];
81        // Replace root object with the one at rightmost leaf
82        if (size > 1) heap[0] = heap[size-1];
83        // Dispose the rightmost leaf
84        heap[--size] = null;
85        // Sink the new root element
86        sink(heap, size, 0, cmp);
87        // Return the object removed
88        return object;
89     }
90     /**
91      * Sorts an array of objects in O(N lg N) time using heap-sort;
92      * N is the array length.
93      * @throws NullPointerException If any array element is null.
94      * @param array The array to sort.
95      */
96     public static void sort(Object[] array) {
```

```
97          // Use the default comparator
98          Comparator cmp = new DefaultComparator();
99          // Arrange array elements in heap order
100         heap(array, array.length, cmp);
101         // Repeatedly replace the root element and sink it
102         for (int j = array.length-1; j > 0; --j) {
103            swap(array, 0, j);
104            sink(array, j, 0, cmp);
105         }
106      }
107      /**
108       * In-place, bottom-up heap construction.
109       */
110      static void heap(Object[] a, int size, Comparator c) {
111         for (int i = (size-2)/2; i >= 0; --i) sink(a, size, i, c);
112      }
113      /**
114       * Shift up.
115       */
116      static void swim(Object[] heap, int i, Comparator c) {
117         while (i > 0) {
118            int p = (i - 1) / 2;
119            int result = c.compare(heap[i], heap[p]);
120            if (result <= 0) return;
121            swap(heap, i, p);
122            i = p;
123         }
124      }
125      /**
126       * Shift down.
127       */
128      static void sink(Object[] heap, int size, int i, Comparator c){
129         int l = 2*i+1, r = l+1, m = l;
130         // If 2*i+1 >= size, node i is a leaf
131         while (l < size) {
132            // Determine the largest children of node i
133            if (r < size) {
134               m = c.compare(heap[l], heap[r]) < 0 ? r : l;
135            }
136            // If the heap condition holds, stop. Else swap and go on.
137            if (c.compare(heap[i], heap[m]) >= 0) return;
138            swap(heap, i, m);
139            i = m; l = 2*i+1; r = l+1; m = l;
140         }
141      }
142      /**
```

```
143      * Interchanges two array elements.
144      */
145     static void swap(Object[] array, int i, int j) {
146        Object tmp = array[i];
147        array[i] = array[j];
148        array[j] = tmp;
149     }
150  }
```

Ασκήσεις

7-1 Σχεδιάστε ένα δυαδικό σωρό 11 αντικειμένων.

7-2 Σχεδιάστε το δυαδικό σωρό που προκύπτει από την διαδοχική εισαγωγή των αντικειμένων **2 E A S Y Q U E S T I O N** σε ένα κενό δυαδικό σωρό.

7-3 Σχεδιάστε το δυαδικό σωρό που προκύπτει από την εκτέλεση των εντολών **E A S Y Q - - - U E - S T - - I O N**, σε ένα κενό δυαδικό σωρό. Ένα γράμμα σημαίνει εισαγωγή του αντίστοιχου αντικειμένου, ενώ το σύμβολο - υποδεικνύει διαγραφή της ρίζας του σωρού.

7-4 Αν αντιστρέψουμε τη σημασιολογία της διάταξης δυαδικού σωρού, το αντικείμενο κάθε κόμβου πρέπει να είναι μικρότερο ή ίσο των αντικειμένων των παιδιών του κόμβου αυτού. Διατυπώστε ανάλογα τους αλγορίθμους 7-1, 7-2 και 7-4.

7-5 Υλοποιήστε ένα δυαδικό σωρό χρησιμοποιώντας οργάνωση δυαδικού δέντρου.

Ασκήσεις

7-6 Υλοποιήστε μια ουρά προτεραιότητας χρησιμοποιώντας δυαδικό σωρό με οργάνωση συστοιχίας.

ΚΕΦΑΛΑΙΟ 8

Πίνακες Κατακερματισμού

Ας υποθέσουμε πως χρειαζόμαστε ένα μηχανισμό αποθήκευσης και αναζήτησης θετικών ακεραίων αριθμών που ανήκουν στο διάστημα $[0, 2^{31}]$. Θέλουμε δηλαδή, μια υλοποίηση της διεπαφής `Collection`, εξειδικευμένη στον χειρισμό θετικών ακεραίων αριθμών. Ένας ταχύτατος μηχανισμός θα ήταν να χρησιμοποιήσουμε μια συστοιχία 2^{31} ακεραίων και να θεωρήσουμε πως ο ακέραιος i υπάρχει στη συλλογή, αν η θέση i της συστοιχίας έχει τιμή i, ενώ δεν υπάρχει όταν η θέση i της συστοιχίας έχει την τιμή 0. Έτσι η εισαγωγή θα απαιτούσε μία σύγκριση και ενδεχομένως μια ανάθεση, ενώ η αναζήτηση μια σύγκριση και η διαγραφή, μία ανάθεση. Η ταχύτητα ωστόσο του μηχανισμού αυτού, επιτυγχάνεται σε βάρος της γενικότητας και με μεγάλη κατανάλωση μνήμης.

i) Τα στοιχεία της συλλογής πρέπει να είναι θετικοί ακέραιοι αριθμοί, διαφορετικά δεν μπορούν να χρησιμοποιηθούν για τη διευθυνσιοδότηση της συστοιχίας.

ii) Ανεξάρτητα από το πλήθος των στοιχείων που θέλουμε να αποθηκεύσουμε, είμαστε αναγκασμένοι να δεσμεύσουμε μια συστοιχία $2.147.483.648$ στοιχείων, δηλαδή 8 GB μνήμης.

Αν και μπορούμε να ελαττώσουμε κατά πολύ τις απαιτήσεις σε μνήμη (βλέπε Άσκηση 3-9) κάτι τέτοιο δεν θα έχει ιδιαίτερη αξία εκτός αν περιοριστούμε στη χρήση του μηχανισμού για φυσικούς αριθμούς. Ωστόσο χρησιμοποιώντας τη βασική ιδέα αυτού του απλού μηχανισμού, μπορούμε να σχεδιάσουμε εξαιρετικά αποτελεσματικούς, γενικούς, αφηρημένους τύπους δεδομένων που βασίζονται

στη χρήση συστοιχιών. Ο μηχανισμός που θα περιγράψουμε είναι γνωστός ως *πίνακας κατακερματισμού*, και αποτελεί το αντικείμενο αυτού του κεφαλαίου.

8.1 Εισαγωγή

Ένας πίνακας κατακερματισμού υλοποιείται με τη χρήση μιας συστοιχίας. Αν γνωρίζουμε, έστω και κατά προσέγγιση, το μέγιστο πλήθος (έστω M) αντικειμένων που θα χρειαστεί να αποθηκευτούν στον πίνακα κατακερματισμού, μπορούμε να χρησιμοποιήσουμε αντικείμενα οποιουδήποτε τύπου για τη *διευθυνσιοδότηση* μιας συστοιχίας M στοιχείων, αν ορίσουμε μια συνάρτηση από το σύνολο των αντικειμένων έστω O, στο σύνολο $[0, M)$. Η τεχνική για τον ορισμό τέτοιων συναρτήσεων, ονομάζεται κατακερματισμός (hashing). Έχοντας στη διάθεσή μας μια συνάρτηση κατακερματισμού $h : O \rightarrow [0, M)$, μπορούμε να αποθηκεύσουμε κάθε αντικείμενο o στη θέση $h(o)$ μιας συστοιχίας, εφόσον η h είναι αμφιμονοσήμαντη, εφόσον δηλαδή

$$h(o_1) \neq h(o_2)\ \forall o_1, o_2 \in O,\ o_1 \neq o_2$$

Με αυτόν τον τρόπο εξαλείφουμε τα δύο προβλήματα που αναφέρθηκαν παραπάνω.

Δυστυχώς αμφιμονοσήμαντες (ή αλλιώς *τέλειες συναρτήσεις κατακερματισμού*, είναι κατά κανόνα δύσκολο να οριστούν χωρίς παραδοχές για τις ιδιότητες του συνόλου O. Επιπλέον οι τέλειες συναρτήσεις κατακερματισμού απαιτούν κατά κανόνα, σημαντικό χρόνο για τον υπολογισμό τους, τόσο σημαντικό μάλιστα, που σε πολλές περιπτώσεις εξαλείφει τα υπόλοιπα πλεονεκτήματα των πινάκων κατακερματισμού. Ένα επιπλέον πρόβλημα που εμπλέκεται στη σχεδίαση τέλειων συναρτήσεων κατακερματισμού είναι η χωρητικότητα του πίνακα κατακερματισμού, η οποία πρέπει να είναι ίση με τον πληθικό αριθμό του πεδίου τιμών της συνάρτησης κατακερματισμού. Έτσι, η χρήση τέλειων συναρτήσεων κατακερματισμού, περιορίζεται σε περιπτώσεις που το πλήθος των στοιχείων του συνόλου O είναι σχετικά μικρό και γνωστό εκ των προτέρων.

Η περίπτωση κατά την οποία, δύο αντικείμενα $o_1, o_2 \in O$ για τα οποία ισχύει $o_1 \neq o_2$, έχουν την ίδια τιμή κατακερματισμού $x = h(o_1) = h(o_2)$, ονομάζεται *σύγκρουση*. Αν έχουμε στη διάθεσή

μας μέσα για την διαχείριση συγκρούσεων, τότε μπορούμε να κατασκευάσουμε πίνακες κατακερματισμού, χωρίς να απαιτείται μια τέλεια συνάρτηση κατακερματισμού.

Στην πράξη η διαδικασία της μετατροπής ενός αντικειμένου σε μια διεύθυνση πίνακα, διακρίνεται, για λόγους που αναλύουμε στην ενότητα 8.3, σε δύο βήματα. Το πρώτο, ο κατακερματισμός, αντιστοιχίζει ένα αντικείμενο, όχι σε ένα θετικό ακέραιο στο διάστημα $[0, M)$ όπως αναφέραμε παραπάνω, αλλά σε ένα αυθαίρετο ακέραιο σύμφωνα με τα πρότυπα της γλώσσας προγραμματισμού που χρησιμοποιείται. Το δεύτερο βήμα, που ονομάζεται *συμπίεση διεύθυνσης* (address compression), είναι αυτό που μετατρέπει το αποτέλεσμα του πρώτου βήματος, την τιμή κατακερματισμού δηλαδή, σε μια διεύθυνση πίνακα. Συνοψίζοντας λοιπόν, πρέπει να πούμε πως προκειμένου να υλοποιήσουμε ένα πίνακα κατακερματισμού πρέπει να έχουμε στη διάθεσή μας τα παρακάτω εργαλεία.

i). Ένα μηχανισμό κατακερματισμού.
ii). Ένα μηχανισμό συμπίεσης διεύθυνσης
iii). Ένα μηχανισμό διαχείρισης πιθανών συγκρούσεων.

8.2 Κατακερματισμός

Στην Java, όπως και σε πολλές άλλες γλώσσες προγραμματισμού, υπάρχουν σημαντικές διαφορές υλοποίησης μεταξύ των πρωτογενών και των αφηρημένων τύπων δεδομένων. Έτσι θα αναλύσουμε ξεχωριστά τις μεθόδους κατακερματισμού σε κάθε μία από τις δύο αυτές περιπτώσεις.

8.2.1 Κατακερματισμός πρωτογενών τύπων δεδομένων

Οι πρωτογενείς τύποι δεδομένων χαρακτηρίζονται από δύο ιδιότητες, το μορφότυπο και το μήκος (σε δυαδικά ψηφία). Λαμβάνοντας υπόψη αυτές τις ιδιότητες μπορούμε να αντιστοιχίσουμε κάθε τιμή ενός βασικού τύπου σε μια τιμή τύπου `int`. Για τους τύπους που έχουν μικρότερο μήκος από τον τύπο `int`, η αντιστοιχία θα είναι αμφιμονοσήμαντη (τέλεια), σε αντίθεση με τους τύπους με μεγαλύτερο μήκος.

Κατακερματισμός τιμών τύπου `boolean`. Στις δύο τιμές του τύπου `boolean` μπορούμε να αντιστοιχίσουμε δύο αυθαίρετες τιμές τύπου `int`, όπως για παράδειγμα, 1231 και 1237.

Κατακερματισμός τιμών τύπου `char`. Οι τιμές τύπου `char` κατακερματίζονται εύκολα και αμφιμονοσήμαντα, καθώς σε κάθε χαρακτήρα αντιστοιχεί ένας μοναδικός θετικός ακέραιος, το κωδικοσημείο Unicode.

Κατακερματισμός τιμών τύπου `byte` **και** `short`. Ο τύπος `byte` είναι υποσύνολο του τύπου `int`. Έτσι κάθε τιμή του τύπου `byte` είναι ταυτόχρονα και τιμή τύπου `int`. Το ίδιο ισχύει βεβαίως και για τον τύπο `short`.

Κατακερματισμός τιμών τύπου `float`. Ο τύπος `float` αν και έχει διαφορετικό πρότυπο από τον τύπο `int`, έχει το ίδιο μήκος (32 δυαδικά ψηφία), οπότε κάθε τιμή του μπορεί εύκολα να μετατραπεί αμφιμονοσήμαντα σε μια τιμή τύπου `int`. Η κλάση `java.lang.Float` παρέχει τη στατική μέθοδο `floatToIntBits(float)` η οποία αντιγράφει τα δυαδικά ψηφία ενός αριθμού κινητής υποδιαστολής σε ένα ακέραιο.

Κατακερματισμός τιμών τύπου `long`. Αντίθετα με τον τύπο `float`, ο τύπος `long` έχει το ίδιο πρότυπο με τον τύπο `int` αλλά διπλάσιο μήκος. Μπορούμε για κάθε τιμή τύπου `long` να υπολογίσουμε μια τιμή τύπου `int` αν παίρναμε τα 32 λιγότερο σημαντικά δυαδικά ψηφία. Υπάρχει ωστόσο το μειονέκτημα ότι λαμβάνουμε υπόψη μόνο τη μισή πληροφορία. Ένας καλύτερος τρόπος για να κατακερματίσουμε ένα μεγάλο ακέραιο είναι να υπολογίσουμε την λογική σύζευξη ή αποκλειστική διάζευξη των δύο τιμών τύπου `int` από τις οποίες αποτελείται μια τιμή τύπου `long`.

Κατακερματισμός τιμών τύπου `double`. Ο τύπος `double` έχει μήκος 64 δυαδικά ψηφία όπως και ο τύπος `long`. Μπορούμε λοιπόν να μετατρέψουμε κάθε τιμή του σε τιμή τύπου `long` (χρησιμοποιώντας τη στατική μέθοδο `doubleToLongBits(double)` της κλάσης `java.lang.Double`) και αυτή με τη σειρά της σε τιμή τύπου

`int` όπως έχουμε ήδη περιγράψει. Τα παραπάνω συνοψίζονται στον Πίνακα 8-1.

8.2.2 Κατακερματισμός αντικειμένων

Σε αντίθεση με τους πρωτογενείς τύπους, οι αφηρημένοι τύποι δεδομένων δεν έχουν προκαθορισμένες τιμές. Αν και αυτό με την πρώτη ματιά καθιστά δύσκολη την αναλυτική περιγραφή μιας γενικής μεθόδου κατακερματισμού, τα πράγματα δεν είναι ακριβώς έτσι. Ας θεωρήσουμε προς στιγμή, πως τα αντικείμενα της κλάσης `C` έχουν k το πλήθος πεδία τύπου `int`. Από μαθηματική σκοπιά, τα αντικείμενα αυτά δεν είναι παρά διατεταγμένες πλειάδες

$$o = (x_0, \ldots, x_{k-1}) \qquad x_i \in \mathbb{Z}.$$

Μπορούμε να υπολογίσουμε μια τιμή κατακερματισμού για κάθε τέτοιο αντικείμενο, επιλέγοντας μια θετική σταθερά a και υπολογίζοντας την παράσταση (8.1).

$$h(x_0)a^{k-1} + h(x_1)a^{k-2} + \cdots + h(x_{k-2})a + h(x_{k-1}) \tag{8.1}$$

Η (8.1) είναι ένα πολυώνυμο της μεταβλητής a, βαθμού $k-1$. Οι συντελεστές του πολυωνύμου είναι οι τιμές κατακερματισμού των συνιστωσών x_i, $i = 0, \ldots, k-1$, δηλαδή των πεδίων της κλάσης `C`. Εφαρμόζοντας τον κανόνα του Horner, μπορούμε να γράψουμε την (8.1) ως

$$h(x_{k-1}) + a\Big(h(x_{k-2}) + a\Big(h(x_{k-3}) + \cdots + a\big(h(x_1) + ah(x_0)\big)\cdots\Big)\Big) \tag{8.2}$$

Αυτή η τεχνική κατακερματισμού ονομάζεται *πολυωνυμικός κατακερματισμός*. Η προσεκτική επιλογή της τιμής a παίζει σημαντικό ρόλο στην αποτελεσματικότητά της. Θεωρητικές και πειραματικές μελέτες έχουν δείξει, πως επιλέγοντας ένα σχετικά μικρό πρώτο αριθμό, όπως για παράδειγμα 31, 37 ή 41, επιτυγχάνουμε πολύ ικανοποιητική αποτελεσματικότητα. Φυσικά δεν υπάρχει κανένας λόγος να περιοριστούμε μονάχα σε αντικείμενα των οποίων τα πεδία είναι ακέραιοι. Αφού γνωρίζουμε πως να κατακερματίζουμε όλους τους πρωτογενείς τύπους δεδομένων της γλώσσας Java, μπορούμε να υπολογίζουμε τιμές κατακερματισμού για κάθε αντικείμενο του οποίου τα πεδία είναι μεταβλητές πρωτογενών τύπων δεδομένων. Η μέθοδος `hash` για παράδειγμα, στο παρακάτω απόσπασμα κώδικα,

Σχήμα 8-1: Το μπλοκ C0 του προτύπου Unicode.
Το μπλοκ C0 περιλαμβάνει τους χαρακτήρες ελέγχου, και το λατινικό αλφάβητο. Το κωδικοσημείο κάθε χαρακτήρα στο δεκαεξαδικό σύστημα προκύπτει αν ενώσουμε τον αριθμό κάθε στήλης με τον αριθμό κάθε γραμμής. Έτσι για παράδειγμα ο χαρακτήρας A (LATIN CAPITAL LETTER A) αντιστοιχεί στο κωδικοσημείο 41.

	000	001	002	003	004	005	006	007
0	NUL 000	DLE 016	SP 032	0 048	@ 064	P 080	` 096	p 112
1	SOH 001	DC1 017	! 033	1 049	A 065	Q 081	a 097	q 113
2	STX 002	DC2 018	" 034	2 050	B 066	R 082	b 098	r 114
3	ETX 003	DC3 019	# 035	3 051	C 067	S 083	c 099	s 115
4	EOT 004	DC4 020	$ 036	4 052	D 068	T 084	d 100	t 116
5	ENQ 005	NAK 021	% 037	5 053	E 069	U 085	e 101	u 117
6	ACK 006	SYN 022	& 038	6 054	F 070	V 086	f 102	v 118
7	BEL 007	ETB 023	' 039	7 055	G 071	W 087	g 103	w 119
8	BS 008	CAN 024	(040	8 056	H 072	X 088	h 104	x 120
9	HT 009	EM 025	) 041	9 057	I 073	Y 089	i 105	y 121
A	LF 010	SUB 026	* 042	: 058	J 074	Z 090	j 106	z 122
B	VT 011	ESC 027	+ 043	; 059	K 075	(091	k 107	{ 123
C	FF 012	FS 028	, 044	< 060	L 076	\ 092	l 108	| 124
D	CR 013	GS 029	- 045	= 061	M 077	) 093	m 109	} 125
E	SO 014	RS 030	. 046	> 062	N 078	^ 094	n 110	~ 126
F	SI 015	US 031	/ 047	? 063	O 079	_ 095	o 111	DEL 127

ΤΥΠΟΣ ΔΕΔΟΜΕΝΩΝ	ΣΥΝΑΡΤΗΣΗ ΚΑΤΑΚΕΡΜΑΤΙΣΜΟΥ
boolean	value ? 1231 : 1237
byte	(int) value
char	(int) value
short	(int) value
float	Float.floatToIntBits(value)
long	(int)(value ^ (value >>> 32))
double	(int)(v ^ (v >>> 32))
	όπου v είναι Double.doubleToLongBits(value)

Πίνακας 8-1: Κατακερματισμός πρωτογενών τύπων δεδομένων.
Η αριστερή στήλη του πίνακα δίνει τον τύπο της μεταβλητής value, ενώ η δεξιά, την αντίστοιχη συνάρτηση κατακερματισμού.

υπολογίζει μια τιμή κατακερματισμού για το αντικείμενο s, τύπου char[], χρησιμοποιώντας πολυωνυμικό κατακερματισμό.

Κώδικας 8.1: Κατακερματισμός συμβολοσειρών.

```
1 public static int hash(char[] s) {
2    int h = 0;
3    int l = s.length;
4    for (int i = 0; i < l; ++i) {
5       h = 31 * h + (int) s[i];
6    }
7    return h;
8 }
```

Η μέθοδος hashCode

Τι γίνεται ωστόσο όταν μια κλάση, όπως για παράδειγμα η κλάση SparePart του Κώδικα 8.2, έχει πεδία των οποίων ο τύπος είναι μια κλάση αντί ενός βασικού τύπου δεδομένων όπως έχουμε θεωρήσει μέχρι στιγμής; Στη γλώσσα Java, κάθε κλάση κληρονομεί τη μέθοδο

```
public int hashCode()
```

από την κλάση Object. Η μέθοδος αυτή είναι υπεύθυνη για την παραγωγή της τιμής κατακερματισμού. Χρησιμοποιώντας τη μέθοδο hashCode, μπορούμε να υπολογίζουμε τιμές κατακερματισμού για αντικείμενα οποιασδήποτε κλάσης.

Κώδικας 8.2: Υλοποίηση της hashCode με πολυωνυμικό κατακερματισμό.

```
1 public final class SparePart {
2    private int serialNumber;
3    private String name;
```

```
4     public int hashCode() {
5        int hash = 0;
6        hash = 31 * hash + serialNumber;
7        hash = 31 * hash + name.hashCode();
8        return hash;
9     }
10    public boolean equals(Object o) {
11       if (this == o) return true;
12       if (o == null) return false;
13       if (o.getClass() != SparePart.class) return false;
14       SparePart p = (SparePart) o;
15       return serialNumber == p.serialNumber && name.equals(p.name);
16    }
17 }
```

Στον Κώδικα 8.2 θεωρούμε πως κάθε αντικείμενο της κλάσης SparePart είναι ένα ζεύγος (i, s), όπου i είναι η τιμή κατακερματισμού του πεδίου serialNumber και s είναι η τιμή κατακερματισμού του πεδίου name. Για το πεδίο serialNumber η τιμή κατακερματισμού συμπίπτει με την τιμή του πεδίου, ενώ την τιμή κατακερματισμού του πεδίου name, μπορούμε να την πάρουμε από τη μέθοδο hashCode της κλάσης String[1]. Αυτό που απομένει για τη μέθοδο hashCode της κλάσης SparePart, είναι απλώς να υπολογίσει την πολυωνυμική τιμή κατακερματισμού με βάση την (8.2) για $a = 31$.

Κάθε φορά που σχεδιάζουμε μια νέα κλάση πρέπει να παρέχουμε μια υλοποίηση της μεθόδου hashCode εκτός αν κάποια υπερκλάση παρέχει μια κατάλληλη υλοποίηση. Η υλοποίηση της hashCode στην κλάση Object, επιστρέφει τη διεύθυνση μνήμης στην οποία είναι αποθηκευμένο το αντικείμενο μεσω του οποίου η μέθοδος καλείται. Αν αποφασίσουμε να ακυρώσουμε αυτή τη συμπεριφορά πρέπει να λάβουμε υπόψη τις παρακάτω ιδιότητες που πρέπει να εξασφαλίζει μια υλοποίηση της hashCode.

Συνέπεια ως προς την μέθοδο equals. Αν x και y είναι αναφορές διάφορες του null και x.equals(y) == true τότε πρέπει να ισχύει x.hashCode() == y.hashCode(). Το αντίστροφο ωστόσο δεν απαιτείται: Αν x.equals(y) == false δεν είναι απαραίτητο να ισχύει x.hashCode() == y.hashCode(). Με άλλα λόγια δεν απαιτείται, για

[1] Η υλοποίηση της μεθόδου hashCode στην κλάση String είναι πανομοιότυπη με αυτήν της μεθόδου hash στον Κώδικα 8.1.

λόγους που πρέπει να είναι πλέον κατανοητοί, η υλοποίηση να είναι μια τέλεια συνάρτηση κατακερματισμού.

Ντετερμινιστικότητα. Κάθε φορά που η μέθοδος `hashCode` καλείται στο ίδιο αντικείμενο κατά τη διάρκεια εκτέλεσης μιας εφαρμογής, πρέπει να επιστρέφει την ίδια τιμή, εφόσον η πληροφορία που χρησιμοποιεί η `equals` δεν έχει αλλάξει μεταξύ των κλήσεων.
Συνδυάζοντας τα παραπάνω με όσα έχουμε αναφέρει για την `equals`, καταλήγουμε στον κανόνα: Όταν ακυρώνεται η `equals`, πρέπει να ακυρώνεται και η `hashCode` ώστε να είναι συνεπής με την νέα υλοποίηση της `equals`. Επιπλέον η υλοποίηση της `hashCode`, πρέπει να χρησιμοποιεί για τον υπολογισμό της τιμής κατακερματισμού τα ίδια ακριβώς πεδία που χρησιμοποιεί και η `equals`.

8.3 Συμπίεση διεύθυνσης

Μέχρι στιγμής έχουμε αναλύσει τρόπους για την αντιστοίχιση τιμών ή αντικειμένων ενός τύπου, σε τιμές τύπου `int`. Γνωρίζουμε ωστόσο πως στη γλώσσα Java, η διευθυνσιοδότηση των θέσεων μιας συστοιχίας απαιτεί μη αρνητικούς ακεραίους στο διάστημα $[0, 2^{31} - 1]$. Για την υλοποίηση ενός πίνακα κατακερματισμού, εκτός των συναρτήσεων κατακερματισμού, απαιτείται ένας τρόπος για την απεικόνιση της τιμής κατακερματισμού ενός αντικειμένου σε μια θέση του πίνακα κατακερματισμού. Η απεικόνιση αυτή ονομάζεται *συμπίεση διεύθυνσης*.

Σε πολλές περιπτώσεις η συμπίεση διεύθυνσης αντιμετωπίζεται ως μέρος της ίδιας της συνάρτησης κατακερματισμού. Αυτή η θεώρηση είναι ωστόσο λανθασμένη, καθώς ενώ μια τεχνική κατακερματισμού αναπτύσσεται πάντα σε σχέση με τις ιδιαιτερότητες ενός τύπου δεδομένων, η συμπίεση διεύθυνσης είναι μια λεπτομέρεια υλοποίησης του ίδιου του πίνακα κατακερματισμού. Με άλλα λόγια, ο κατακερματισμός έχει να κάνει με τα χαρακτηριστικά των δεδομένων, ενώ η συμπίεση έχει να κάνει με τα χαρακτηριστικά του πίνακα κατακερματισμού. Κατά συνέπεια, την ευθύνη για την παραγωγή της τιμής κατακερματισμού για ένα αντικείμενο πρέπει να την έχει ο τύπος του αντικειμένου, ενώ την ευθύνη της συμπίεσης διεύθυνσης πρέπει να την έχει ο τύπος του πίνακα κατακερματισμού. Έτσι

k	p_k	$2^k - p_k$
3	7	1
4	13	3
5	31	1
6	61	3
7	127	1
8	251	5
9	509	3
10	1.021	3
11	2.039	9
12	4.093	3
13	8.191	1
14	16.381	3
15	32.749	19
16	65.521	15
17	131.071	1
18	262.139	5
19	524.287	1
20	1.048.573	3
21	2.097.143	9
22	4.194.301	3
23	8.388.593	15
24	16.777.213	3
25	33.554.393	39
26	67.108.859	5
27	134.217.689	39
28	268.435.399	57
29	536.870.909	3
30	1.073.741.789	35
31	2.147.483.647	1

Πίνακας 8-2: Πρώτοι αριθμοί που προσεγγίζουν δυνάμεις του 2.
Με p_k συμβολίζουμε τον μεγαλύτερο πρώτο ακέραιο που είναι μικρότερος του 2^k.

ανεξαρτητοποιούμε τα δεδομένα από το μηχανισμό δόμησής τους.

Στις επόμενες παραγράφους θα περιγράψουμε τρεις μεθόδους απεικόνισης μιας ακέραιας τιμής κατακερματισμού h στις θέσεις 0 ως $M-1$ ενός πίνακα κατακερματισμού. Όπως οι συναρτήσεις κατακερματισμού πρέπει να κατανέμουν ομοιόμορφα τα αντικείμενα ενός τύπου στον άξονα των ακεραίων, έτσι και οι συναρτήσεις συμπίεσης διεύθυνσης πρέπει να κατανέμουν με ομοιόμορφο τρόπο, τις ακέραιες τιμές κατακερματισμού στις θέσεις του πίνακα κατακερματισμού, προκειμένου να ελαχιστοποιείται η πιθανότητα εμφάνισης συγκρούσεων.

Η μέθοδος της διαίρεσης. Η μέθοδος αυτή χρησιμοποιεί τη συνάρτηση

$$c(h) = |h| \bmod M \tag{8.3}$$

για την παραγωγή μιας διεύθυνσης στο διάστημα $[0, M)$. Προκειμένου η (8.3) να είναι αποτελεσματική, το μήκος του πίνακα κατακερματισμού M, πρέπει να είναι πρώτος αριθμός. Καθώς ο γρήγορος υπολογισμός πρώτων αριθμών δεν είναι τετριμμένη διαδικασία, στην πράξη χρησιμοποιούνται γνωστοί πρώτοι αριθμοί. Οι ακέραιοι της μορφής $2^k - 1$ για παράδειγμα, είναι πρώτοι, εφόσον το k ανήκει στο σύνολο $\{2, 3, 5, 7, 13, 17, 19, 31\}$. Πρόκειται για τους αριθμούς Mersenne. Σε περιπτώσεις που οι αριθμοί Mersenne δεν είναι αποδεκτή λύση –επειδή για παράδειγμα υπάρχει αρκετά μεγάλη απόσταση μεταξύ μερικών από αυτούς, μπορεί να χρησιμοποιηθεί ο Πίνακας 8-2 ο οποίος παρουσιάζει τους πρώτους αριθμούς που είναι πολύ κοντά σε δυνάμεις του 2.

Η μέθοδος MAD. Όταν η συνάρτηση κατακερματισμού τείνει να παράγει πολλές τιμές της μορφής $pM + q$, η μέθοδος της διαίρεσης, ακόμη και με μια καλή επιλογή του μήκους του πίνακα κατακερματισμού, θα τείνει να δημιουργεί συγκρούσεις. Μπορούμε ωστόσο να αμυνθούμε έναντι μιας τέτοιας συνάρτησης κατακερματισμού, αν χρησιμοποιήσουμε την συνάρτηση συμπίεσης

$$c(h) = |ah + b| \bmod M \tag{8.4}$$

Όπως και με τη μέθοδο της διαίρεσης, έτσι και εδώ, ο M πρέπει να είναι πρώτος αριθμός. Οι παράμετροι a και b πρέπει να είναι

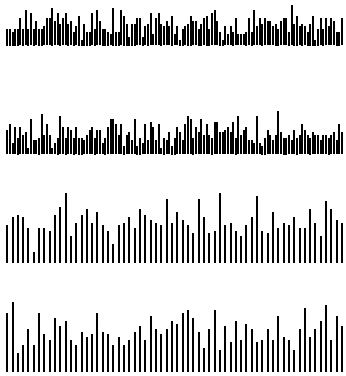
θετικοί ακέραιοι, ενώ θα πρέπει επίσης $aM \bmod \neq 0$.

Συμπίεση με αποκοπή. Ο πιο γρήγορη τεχνική συμπίεσης διεύθυνσης, είναι η χρήση των k λιγότερο σημαντικών δυαδικών ψηφίων της τιμής κατακερματισμού h, όταν το μήκος του πίνακα κατακερματισμού είναι $M = 2^k - 1$. Στην περίπτωση αυτή, η συνάρτηση συμπίεσης ορίζεται ως

$$c(h) = h \ \& \ M - 1 \tag{8.5}$$

όπου & είναι ο τελεστής σύζευξης δυαδικών ψηφίων. Η μοναδική προϋπόθεση που θέτει αυτή η τεχνική, είναι το μήκος του πίνακα κατακερματισμού να είναι δύναμη του 2. Στην πραγματικότητα αυτό δεν είναι περιορισμός αλλά μάλλον πλεονέκτημα σε σχέση με τις μεθόδους της διαίρεσης και του πολλαπλασιασμού. Η μέθοδος ωστόσο, δεν παύει να είναι ευάλωτη σε απόδοση υλοποιήσεις της μεθόδου hashCode. Ένας τρόπος άμυνας είναι να ανακατέψουμε λίγο τα δυαδικά ψηφία της τιμής κατακερματισμού h όπως κάνει για παράδειγμα, η παρακάτω μέθοδος.

```
static int hash(Object x) {
   int h = x.hashCode();
   h += ~(h << 9);
   h ^= (h >>> 14);
   h += (h << 4);
   h ^= (h >>> 10);
   return h;
}
```



Σχήμα 8-2: Συγκριτική παρουσίαση των τεχνικών συμπίεσης διεύθυνσης. Το σχήμα παρουσιάζει την κατανομή 1.000 τυχαίων ακεραίων σε ένα πίνακα 128 θέσεων με τις τεχνικές της συμπίεσης διεύθυνσης που παρουσιάζονται στην ενότητα 8.3. Το πρώτο διάγραμμα (πάνω) παρουσιάζει τη μέθοδο της διαίρεσης. Το δεύτερο διάγραμμα παρουσιάζει τη μέθοδο MAD με $a = 13$ και $b = 3$. Στο τρίτο και τέταρτο διάγραμμα παρουσιάζεται η μέθοδος της αποκοπής η οποία είναι αισθητά κατώτερη από τις άλλες δύο τεχνικές. Στο τελευταίο διάγραμμα, όπου έχουμε ανακατέψει τα δυαδικά ψηφία πριν την αποκοπή, υπάρχει μια μικρή βελτίωση σε σχέση με το τρίτο διάγραμμα.

8.4 Διαχείριση συγκρούσεων

Με βάση τα όσα έχουμε περιγράψει μέχρι τώρα, είναι προφανές πως συγκρούσεις μπορούν να προκύψουν είτε από τη συνάρτηση κατακερματισμού ενός τύπου δεδομένων, είτε λόγω της συνάρτησης συμπίεσης σε συνδυασμό με το μέγεθος του πίνακα κατακερματισμού. Οι τεχνικές για τη διαχείριση συγκρούσεων χωρίζονται σε δύο ομάδες, γνωστές ως *αλυσίδωση* και *ανοικτή διευθυνσιοδότηση*.

8.4.1 Αλυσίδωση

Ένας πίνακας κατακερματισμού μεγέθους M, που υλοποιείται με την τεχνική της αλυσίδωσης χρησιμοποιεί M συνδεδεμένες λίστες· μία για κάθε θέση του πίνακα κατακερματισμού. Όσα αντικείμενα συγκρούονται στη θέση i του πίνακα κατακερματισμού, συνδέονται στην αντίστοιχη λίστα L_i. Πρόκειται για μια απλή και πολύ αποτελεσματική τεχνική διαχείρισης συγκρούσεων.

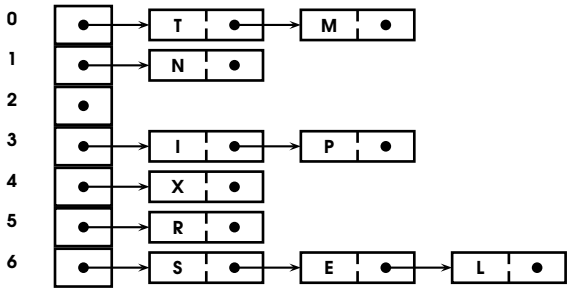
Κατά την εισαγωγή ενός αντικειμένου στον πίνακα κατακερματισμού, καλούμε την συνάρτηση κατακερματισμού του αντικειμένου, δηλαδή τη μέθοδο `hashCode`, και έπειτα συμπιέζουμε το αποτέλεσμα σε μια τιμή $i \in [0, M)$. Έτσι εντοπίζουμε τη λίστα L_i στην οποία πρέπει να αποθηκευτεί το νέο αντικείμενο. Όπως συνήθως, πρέπει να εξασφαλίσουμε πως δεν υπάρχει ήδη αποθηκευμένο αντικείμενο ίσο με το αντικείμενο που εισάγεται. Έτσι, διασχίζουμε την L_i, συγκρίνοντας το αντικείμενο σε κάθε κόμβο με το εισαγόμενο αντικείμενο. Αν κάποια σύγκριση επιστρέψει `true`, η εισαγωγή τελειώνει ανεπιτυχώς. Διαφορετικά κατασκευάζουμε ένα νέο κόμβο, αποθηκεύουμε σε αυτόν μια αναφορά στο νέο αντικείμενο, και τον συνδέουμε στην λίστα L_i.

Για τη διαγραφή ενός αντικειμένου από τον πίνακα κατακερματισμού, εντοπίζουμε όπως και κατά την εισαγωγή τη λίστα L_i στην οποία πρέπει να είναι αποθηκευμένο το αντικείμενο και συγκρίνουμε το αντικείμενο κάθε κόμβου με το αντικείμενο προς διαγραφή. Αν έχουμε επιτυχία σε κάποιο κόμβο, τότε τον αποσυνδέουμε από τη λίστα, διαφορετικά η διαγραφή ολοκληρώνεται ανεπιτυχώς.

8.4.2 Ανοικτή διευθυνσιοδότηση.

Η ανοικτή διευθυνσιοδότηση δεν βασίζεται σε λίστες. Υποθέτουμε πως το μέγεθος του πίνακα κατακερματισμού M είναι μεγαλύτερο από το μέγιστο αριθμό αντικειμένων, έστω N, που θα χρειαστεί να αποθηκευτούν. Με αυτό το μηχανισμό διαχείρισης συγκρούσεων, ένα αντικείμενο που συγκρούεται με κάποιο άλλο αντικείμενο, στην θέση i του πίνακα κατακερματισμού, αποθηκεύεται σε κάποια άλλη θέση.

Κάθε θέση του πίνακα κατακερματισμού αποθηκεύει μια αναφορά σε ένα αντικείμενο το οποίο ονομάζεται *κάδος* (bucket). Κάθε



I	N	S	E	R	T	X	M	P	L
73	78	83	69	82	84	88	77	80	76
3	1	6	6	5	0	4	0	3	6

Σχήμα 8-3: Εισαγωγή σε πίνακα κατακερματισμού με αλυσίδωση. Το σχήμα παρουσιάζει ένα πίνακα κατακερματισμού μετά την εισαγωγή των αντικειμένων τύπου `Character` **I N S E R T X M P L**. Οι τιμές κατακερματισμού και οι διευθύνσεις που παράγονται από αυτές με τη μέθοδο της διαίρεσης δίνονται στο κάτω μέρος του σχήματος.

κάδος μπορεί να αποθηκεύσει ένα αντικείμενο μαζί με πληροφορία που προσδιορίζει την κατάσταση του κάδου: Θα λέμε πως ένας κάδος είναι *κενός*, όταν ποτέ δεν έχει αποθηκευτεί αντικείμενο σε αυτόν, ενώ είναι *κατειλημμένος* καθόσον αποθηκεύει κάποιο αντικείμενο. Όταν ένα αντικείμενο διαγραφεί από ένα κατειλημμένο κάδο, λέμε πως ο κάδος είναι *ελεύθερος*. Τέλος ένας κάδος είναι *διαθέσιμος* αν είναι κενός ή ελεύθερος.

Γραμμική δοκιμή (linear probing). Κατά την εισαγωγή ενός αντικειμένου, αρχικά εντοπίζουμε τον κάδο i στον οποίο πρέπει να αποθηκευτεί το αντικείμενο με βάση την συμπίεση της τιμής κατακερματισμού του. Αν αυτός ο κάδος είναι κενός, τότε αποθηκεύουμε το αντικείμενο σε αυτόν και τερματίζουμε την εισαγωγή επιτυχώς. Αν ο κάδος είναι κατειλημμένος από αντικείμενο ίσο με το εισαγόμενο, τότε τερματίζουμε. Αν ωστόσο ο κάδος είναι κατειλημμένος από διαφορετικό αντικείμενο (σύγκρουση) πρέπει να αποθηκεύσουμε το αντικείμενο σε άλλο κάδο. Έτσι δοκιμάζουμε τον κάδο $i + 1$. Το ίδιο πρέπει να κάνουμε και όταν ο κάδος i είναι ελεύθερος. Αυτό συμβαίνει διότι υπάρχει πιθανότητα να έχει ήδη εισαχθεί ένα αντικείμενο ίσο με το προς εισαγωγή αντικείμενο, ενώ ο κάδος i ήταν κατειλημμένος. Αν αυτό έχει πράγματι συμβεί, τότε δεν αποκλείεται το αντικείμενο να είναι ακόμη αποθηκευμένο σε κάποιο άλλο κάδο. Τέλος, αν η ακολουθία

$$i, i+1, i+2, \ldots, M-1, 0, 1, \ldots, i-1$$

E	X	A	M	P	L	E
69	88	65	77	80	76	69
4	10	0	12	2	11	4

0	1	2	3	4	5	6	7	8	9	10	11	12
				E								
				E						X		
A				E						X		
A				E						X		M
A		P		E						X		M
A		P		E						X	L	M
A		P		E						X	L	M

Σχήμα 8-4: Εισαγωγή αντικειμένων σε πίνακα κατακερματισμού με ανοικτή διευθυνσιοδότηση γραμμικής δοκιμής. Το σχήμα παριστάνει τα περιεχόμενα του πίνακα αμέσως μετά την εισαγωγή καθενός των στοιχείων τύπου `char`: **EXAMPLE**. Οι τιμές κατακερματισμού και οι διευθύνσεις που παράγονται με τη μέθοδο της διαίρεσης δίνονται στο επάνω μέρος του σχήματος.

εξαντληθεί χωρίς αποτέλεσμα, σημαίνει πως ο πίνακας κατακερματισμού είναι γεμάτος, επομένως η εισαγωγή δεν μπορεί να πραγματοποιηθεί. Στο Σχήμα 8-4 παρουσιάζεται η λειτουργία ενός πίνακα κατακερματισμού που χρησιμοποιεί την μέθοδο της γραμμικής δοκιμής.

Στη μέθοδο γραμμικής δοκιμής η ακολουθία θέσεων δοκιμής ορίζεται από τη συνάρτηση

$$i + k,\ k = 0, 1, 2, \ldots$$

και τείνει να συγκεντρώνει τα αντικείμενα σε συνεχόμενες θέσεις του πίνακα κατακερματισμού. Το φαινόμενο αυτό που ονομάζεται *συσσώρευση* και παρουσιάζεται στο Σχήμα 8-5, επηρεάζει την αποτελεσματικότητα της αναζήτησης και κατά συνέπεια της εισαγωγής και διαγραφής αντικειμένων. Στον πίνακα κατακερματισμού του σχήματος 8-5 για παράδειγμα, ο εντοπισμός του αντικειμένου **G** απαιτεί 5 δοκιμές, στις θέσεις 6, 7, 8, 9, και 10. Παρατηρούμε πως η εισαγωγή ενός αντικειμένου, στο συγκεκριμένο παράδειγμα του αντικειμένου **T**, μπορεί να επηρεάσει σημαντικά την ακολουθία δοκιμών που απαιτούνται για την εισαγωγή ενός άλλου στοιχείου γεγονός που καθιστά τον πίνακα κατακερματισμού, αρκετά ευάλωτο στην σειρά με την οποία εισάγονται τα αντικείμενα. Αυτό οφείλεται στο γεγονός πως έπειτα από κάθε σύγκρουση η επόμενη θέση η οποία θα δοκιμαστεί είναι η ίδια για όλα τα αντικείμενα.

C	L	U	S	T	E	R	I	N	G
67	76	85	83	84	69	82	73	78	71
2	11	7	5	6	4	4	8	0	6

0	1	2	3	4	5	6	7	8	9	10	11	12
		C										
		C									L	
		C					U				L	
		C			S		U				L	
		C			S	T	U				L	
		C		E	S	T	U				L	
		C		E	S	T	U	R			L	
		C		E	S	T	U	R	I		L	
N		C		E	S	T	U	R	I		L	
N		C		E	S	T	U	R	I	G	L	

Σχήμα 8-5: Το φαινόμενο της συσσώρευσης σε πίνακες κατακερματισμού που χρησιμοποιούν γραμμική δοκιμή. Το σχήμα παριστάνει τα περιεχόμενα ενός πίνακα κατακερματισμού μήκους 13, αμέσως μετά την εισαγωγή καθενός των στοιχείων τύπου char: **C L U S T E R I N G**.

Διπλός κατακερματισμός. Ένας τρόπος για να μετριάσουμε τη συσσώρευση στοιχείων σε συνεχόμενες θέσεις του πίνακα κατακερματισμού είναι να βρούμε ένα τρόπο ώστε η ακολουθία δοκιμών, να εξαρτάται από κάθε αντικείμενο που εισάγεται. Θα μπορούσαμε να χρησιμοποιήσουμε μια συνάρτηση d και να ορίσουμε την ακολουθία δοκιμών ως

$$(i + kd\,(h)) \bmod M,\ k = 0, 1, 2, \ldots$$

όπου h είναι η τιμή κατακερματισμού του αντικειμένου που εισάγεται, και M το μήκος του πίνακα κατακερματισμού. Η συνάρτηση d πρέπει ασφαλώς να επιλεγεί με προσοχή: Αφενός το μηδέν πρέπει να αποκλειστεί από το πεδίο τιμών της, διαφορετικά έπειτα από μια σύγκρουση θα μπορούσε να ακολουθήσει άπειρη ανακύκλωση. Αφετέρου οι τιμές της d που είναι μεγαλύτερες της μονάδας δεν

πρέπει να διαιρούν το M, διαφορετικά η ακολουθία θέσεων δοκιμής ενδέχεται να μην περιλαμβάνει κάποιες θέσεις του πίνακα κατακερματισμού. Μπορούμε εύκολα να εξασφαλίσουμε τα παραπάνω, αν το M είναι πρώτος, επιλέγοντας την d ώστε το πεδίο τιμών της να είναι το σύνολο $(0, M)$. Ένα παράδειγμα θα μπορούσε να είναι η συνάρτηση

$$d(h) = 1 + h \bmod 61$$

εφόσον το μήκος του πίνακα κατακερματισμού είναι ένας πρώτος αριθμός μεγαλύτερος του 61. Αν έχουμε επιλέξει το μήκος του πίνακα κατακερματισμού με βάση τον Πίνακας 8-2, αν δηλαδή $M = p_i$, θα μπορούσαμε αντί της τιμής 61 να έχουμε την τιμή p_{i-1}. Η τεχνική αυτή, που ονομάζεται *διπλός κατακερματισμός*, περιορίζει το φαινόμενο της συσσώρευσης όπως φαίνεται και στο Σχήμα 8-6.

C	L	U	S	T	E	R	I	N	G
67	76	85	83	84	69	82	73	78	71
2	11	7	5	6	4	4	8	0	6
5	7	2	7	1	7	6	4	2	2

0	1	2	3	4	5	6	7	8	9	10	11	12
		C										
		C									L	
		C					U				L	
		C			S		U				L	
		C			S	T	U				L	
		C		E	S	T	U				L	
		C		E	S	T	U			R	L	
		C		E	S	T	U	I		R	L	
N		C		E	S	T	U	I		R	L	
N		C		E	S	T	U	I		R	L	G

Σχήμα 8-6: Εισαγωγή αντικειμένων σε πίνακα κατακερματισμού με ανοικτή διευθυνσιοδότηση διπλού κατακερματισμού.
Το σχήμα απεικονίζει το ίδιο σενάριο με αυτό του Σχήματος 8-5, με τη διαφορά ότι η επίλυση συγκρούσεων γίνεται με την τεχνική του διπλού κατακερματισμού. Η συνάρτηση d έχει οριστεί ως $d(h) = h7 \bmod +1$ και οι τιμές της για κάθε αντικείμενο, δίνονται κάτω από την αντίστοιχη συμπιεσμένη τιμή κατακερματισμού, στο πάνω μέρος του σχήματος. Η πρώτη σύγκρουση παρουσιάζεται κατά την εισαγωγή του αντικειμένου **R**: το αντικείμενο πρέπει να εισαχθεί στη θέση 4 η οποία είναι κατειλημμένη. Η επόμενη θέση δοκιμής είναι η θέση 10 στην οποία και γίνεται η εισαγωγή. Κατά την εισαγωγή του αντικειμένου **G** έχουμε σύγκρουση στη θέση 6, όπως και στις δύο επόμενες θέσεις δοκιμής 8 και 10, και τελικά το αντικείμενο εισάγεται στη θέση 12.

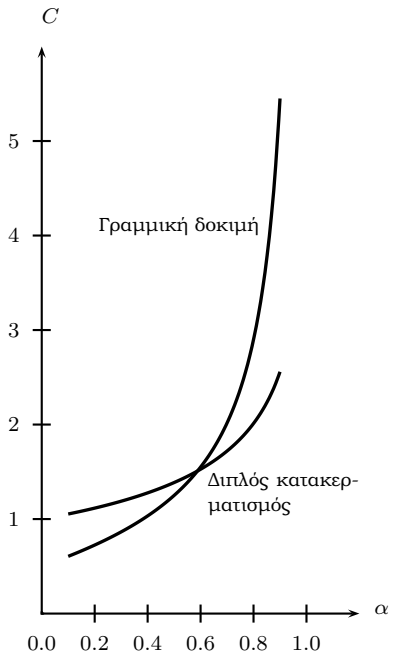
8.5 Αποτελεσματικότητα

Η αποτελεσματικότητα των περισσοτέρων δομών δεδομένων είναι συνάρτηση του πλήθους των αποθηκευμένων αντικειμένων. Στους πίνακες κατακερματισμού αντίθετα, το χαρακτηριστικό που επηρεάζει αποφασιστικά την αποτελεσματικότητα αναζήτησης είναι ο λόγος

$$\alpha = \frac{N}{M} \tag{8.6}$$

του πλήθος των αποθηκευμένων στοιχείων προς τη χωρητικότητα του πίνακα κατακερματισμού.

Ας εξετάσουμε διαισθητικά την τεχνική αλυσίδωσης για ένα πίνακα κατακερματισμού με 100 λίστες και 5.000 αποθηκευμένα αντικείμενα. Εφόσον η κατανομή των αντικειμένων στις θέσεις του πίνακα είναι αρκετά ομοιόμορφη, πρέπει να αναμένουμε περίπου $N/M = 50$ αντικείμενα ανά λίστα. Αν οι λίστες δεν είναι ταξινομημένες, απαιτούνται κατά μέσο όρο 25 συγκρίσεις αντικειμένων για μια επιτυχή αναζήτηση και 50 συγκρίσεις αντικειμένων για μια ανεπιτυχή αναζήτηση. Αξίζει να σημειώσουμε πως ακόμη και στην ακραία περίπτωση που όλα τα αντικείμενα έχουν αποθηκευτεί στην ίδια λίστα, ο μέσος αριθμός αντικειμένων ανά λίστα παραμένει N/M καθώς $(0+0+\cdots+N)/M = N/M$. Αυτό ωστόσο που κάνει αποτελεσματική την αναζήτηση σε πίνακες κατακερματισμού με αλυσίδωση



Σχήμα 8-7: Αποτελεσματικότητα επιτυχούς αναζήτησης σε πίνακες κατακερματισμού.
Στον οριζόντιο άξονα ο συντελεστής φορτίου και στον κάθετο ο αναμενόμενος αριθμός δοκιμών (συγκρίσεων αντικειμένων) που απαιτούνται για τον εντοπισμό ενός αντικειμένου στον πίνακα κατακερματισμού.

είναι η πιθανότητα να φράσσεται το μήκος μιας αλυσίδας από ένα μικρό πολλαπλάσιο του α, η οποία προσεγγίζει τη μονάδα.

Συγκεκριμένα, η πιθανότητα να περιλαμβάνει μια λίστα περισσότερα από $k\alpha$ αντικείμενα, είναι μικρότερη από

$$\left(\frac{\alpha e}{k}\right)^k e^{-\alpha}.$$

Αυτό σημαίνει ότι αν στο παραπάνω παράδειγμα, χρησιμοποιούσαμε ένα πίνακα 5.000 θέσεων, η πιθανότητα να συναντήσουμε μια λίστα με περισσότερα από 3 θα ήταν μικρότερη του 27%, πράγμα που πρακτικά σημαίνει αναζήτηση σε σταθερό χρόνο.

Στην περίπτωση που η διαχείριση συγκρούσεων γίνεται με ανοικτή διευθυνσιοδότηση, ο λόγος α εκφράζει το ποσοστό των θέσεων του πίνακα κατακερματισμού που είναι κατειλημμένες και φυσικά είναι μικρότερος ή ίσος της μονάδας, κάτι που δεν συμβαίνει απαραίτητα στη διαχείριση συγκρούσεων με αλυσίδωση. Σε πολλές περιπτώσεις ο λόγος α ονομάζεται *συντελεστής φορτίου* (load factor) στην ανοικτή διευθυνσιοδότηση. Η διαίσθηση δείχνει πως η επιρροή του λόγου α στην αποτελεσματικότητα είναι και πάλι καθοριστική: Καθώς το πλήθος των κατειλημμένων κάδων τείνει να προσεγγίσει το πλήθος των διαθέσιμων, η πιθανότητα σύγκρουσης αυξάνει οδηγώντας στη μείωση της αποτελεσματικότητας αναζήτησης και εισαγωγής αντικειμένων.

Όταν χρησιμοποιείται γραμμική δοκιμή, μια επιτυχής αναζήτηση σε πίνακα κατακερματισμού με συντελεστή φορτίου α απαιτεί

$$\frac{1}{2}\left(\alpha + \frac{1}{1-\alpha}\right) \tag{8.7}$$

δοκιμές, ενώ θα πρέπει να αναμένουμε

$$\frac{1}{2}\left(\alpha + \frac{1}{(1-\alpha)^2}\right) \tag{8.8}$$

δοκιμές κατά μέσο όρο προκειμένου να ολοκληρωθεί μια ανεπιτυχής αναζήτηση.

Αν αντί της γραμμικής δοκιμής, χρησιμοποιηθεί διπλός κατακερματισμός, μια επιτυχής αναζήτηση απαιτεί

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha} \tag{8.9}$$

δοκιμές, ενώ μια ανεπιτυχής αναζήτηση απαιτεί

$$\frac{1}{1-\alpha} \tag{8.10}$$

δοκιμές κατά μέσο.

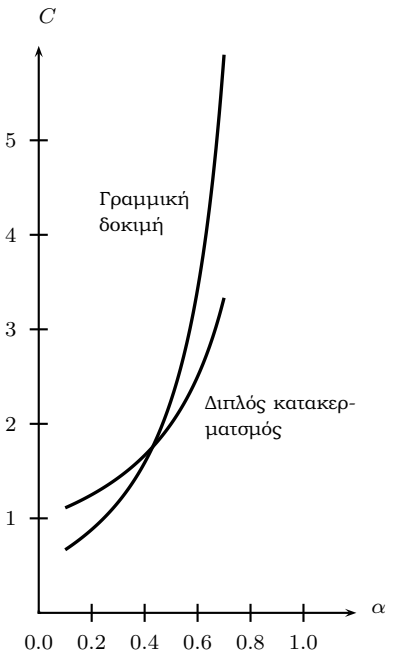
Με βάση τις παραπάνω σχέσεις[2] οφείλουμε να καταλήξουμε στο πρακτικό συμπέρασμα πως πρέπει να φροντίζουμε να διατηρούμε το συντελεστή φορτίου κάτω από τα $3/4$ της χωρητικότητας του πίνακα κατακερματισμού όταν διαχειριζόμαστε συγκρούσεις με ανοικτή διευθυνσιοδότηση.

8.6 Δυναμικοί πίνακες κατακερματισμού

Η ενότητα 8.5 αφήνει μια μικρή απογοήτευση σχετικά με την ισχύ της τεχνικής του κατακερματισμού στην υλοποίηση γρήγορων δομών δεδομένων. Χρησιμοποιώντας αλυσίδωση μπορούμε να έχουμε αρκετά μια γρήγορη δομή δεδομένων της οποίας η απόδοση ελαττώνεται γραμμικά καθώς ο λόγος α αυξάνεται. Όταν χρησιμοποιούμε ανοικτή διευθυνσιοδότηση, η ταχύτητα αναζήτησης μειώνεται με δραματικούς ρυθμούς καθώς ο συντελεστής φορτίου προσεγγίζει τη μονάδα. Και στις δύο περιπτώσεις, προκειμένου να έχουμε μια ικανοποιητική απόδοση, είναι απαραίτητη μια καλή εκτίμηση του μέγιστου πλήθους των αντικειμένων που πρόκειται να αποθηκευτούν. Όταν αυτό δεν είναι εφικτό, το μήκος του πίνακα κατακερματισμού πρέπει να αυξηθεί, ώστε ο λόγος $\alpha = N/M$ να παραμείνει σε αποδεκτά όρια. Μια λύση είναι ο διπλασιασμός της χωρητικότητας του πίνακα κατακερματισμού, όταν διαπιστωθεί πως ο λόγος α αυξάνει πέρα από τα αποδεκτά όρια.

Οι πίνακες κατακερματισμού που υποστηρίζουν αυτό το χαρακτηριστικό, ονομάζονται *δυναμικοί*. Βεβαίως η επέκταση της χωρητικότητας ενός πίνακα κατακερματισμού δεν επιτυγχάνεται χωρίς κόστος. Όταν η τιμή της παραμέτρου M αλλάξει, ορισμένα από τα αποθηκευμένα αντικείμενα θα χρειαστεί να μεταφερθούν σε μια

[2] Η απόδειξη των σχέσεων (8.7)–(8.10) είναι πέραν του αντικειμένου αυτού του κειμένου. Για τους αναγνώστες που ενδιαφέρονται για την περαιτέρω μελέτη τους, η σχετική ανάλυση στο [7] αποτελεί ένα αξεπέραστο παράδειγμα κομψότητας στην ανάλυση αλγορίθμων.



Σχήμα 8-8: Αποτελεσματικότητα ανεπιτυχούς αναζήτησης σε πίνακες κατακερματισμού.
Στον οριζόντιο άξονα ο συντελεστής φορτίου και στον κάθετο ο αναμενόμενος αριθμός δοκιμών (συγκρίσεων αντικειμένων) που απαιτούνται στην περίπτωση που το ζητούμενο αντικείμενο δεν υπάρχει στον πίνακα κατακερματισμού.

νέα θέση. Εφόσον δεν έχουμε στη διάθεσή μας κανένα τρόπο να γνωρίζουμε ποια είναι αυτά τα αντικείμενα, δεν απομένει παρά να τα ελέγξουμε ένα προς ένα. Η διαδικασία αυτή είναι γνωστή ως *επανακατακερματισμός* (rehashing).

Φυσικά οι δυναμικοί πίνακες κατακερματισμού δεν μπορούν να εγγυηθούν ένα ικανοποιητικό φράγμα στο πλήθος των συγκρίσεων που απαιτούνται για την εισαγωγή ενός αντικειμένου, καθώς μια εισαγωγή που θα προκαλέσει επέκταση του πίνακα κατακερματισμού θα οδηγήσει στην επανεισαγωγή όλων των αντικειμένων. Σε περιπτώσεις που απαιτείται ένα ικανοποιητικό τέτοιο φράγμα, οι δυναμικοί πίνακες κατακερματισμού δεν είναι ικανοποιητική λύση. Στις περιπτώσεις ωστόσο που ενδιαφερόμαστε για την μέση αποτελεσματικότητα, οι δυναμικοί πίνακες κατακερματισμού αποτελούν ένα εξαιρετικό εργαλείο, καθώς μπορούμε να επιμερίσουμε επιμερισμός το κόστος επέκτασης σε κάθε εισαγωγή αντικειμένου, επιτυγχάνοντας πρακτικά σταθερό χρόνο εισαγωγής και αναζήτησης.

8.7 Υλοποίηση

Στον Κώδικα 8.3 παρουσιάζεται ο σκελετός για την υλοποίηση συλλογών χρησιμοποιώντας πίνακα κατακερματισμού ανοικτής διευθυνσιοδότησης. Οι κάδοι αποθηκεύονται σε μια συστοιχία και αριθμούνται με βάση τη θέση της συστοιχίας στην οποία βρίσκονται. Η κλάση `AbstractHashTable` βασίζει τη λειτουργία της στην υλοποίηση της μεθόδου `indexOf(Object)` η οποία εντοπίζει τον κάδο στον οποίο βρίσκεται αποθηκευμένο ένα αντικείμενο. Αν το ζητούμενο αντικείμενο δεν βρίσκεται αποθηκευμένο στον πίνακα κατακερματισμού και πρέπει να τοποθετηθεί στον κάδο i, η μέθοδος πρέπει να επιστρέψει την τιμή $-i-1$. Για να υλοποιήσουμε ένα πίνακα κατακερματισμού γραμμικής δοκιμής ή διπλού κατακερματισμού, πρέπει να κληρονομήσουμε την κλάση `AbstractHashTable` και να υλοποιήσουμε κατάλληλα τη μέθοδο `indexOf` (βλέπε Ασκήσεις 8-4 και 8-4). Τέλος για να ολοκληρωθεί η υλοποίηση, θα πρέπει να υλοποιηθεί και η μέθοδος `enumerator` (βλέπε Άσκηση 8-10).

Κώδικας 8.3: Κατακερματισμός με ανοικτή διευθυνσιοδότηση.

```
1 package aueb.util.imp;
2 import aueb.util.api.AbstractCollection;
```

```
3  /**
4   * Abstract implementation of a collection that uses open addressing
5   * hash table. The table is an array of buckets. Each bucket stores an
6   * object and it is always in one of the following states:
7   * empty, occupied, released.
8   * Subclasses must provide an implementation of {@link #indexOf(Object)}.
9   */
10 abstract class AbstractHashTable extends AbstractCollection {
11    /**
12     * Each collection object is stored in a bucket. A bucket is
13     * always in one of the following states:
14     * Empty: The bucket does not store an object.
15     * Occupied: The bucket stores an object.
16     * Released: The bucket stored an object, but is currently empty.
17     * </ul>
18     * @author mms
19     */
20    protected static class Bucket {
21       /**
22        * Indicates an empty bucket.
23        */
24       private static final int S_EMPTY = 0;
25       /**
26        * Indicates a released bucket.
27        */
28       private static final int S_RELEASED = 1;
29       /**
30        * Indicates an occupied bucket.
31        */
32       private static final int S_OCCUPIED = 2;
33       /**
34        * The state of this bucket
35        */
36       private int state;
37       /**
38        * The stored object
39        */
40       private Object object;
41       /**
42        * Default constructor. Creates a bucket in empty state.
43        */
44       public Bucket() {
45          state = S_EMPTY;
46       }
47       /**
48        * Stores an object in this bucket.
```

```
49          * @param object The object to store.
50          */
51         public void store(Object object) {
52            this.object = object;
53            state = S_OCCUPIED;
54         }
55         /**
56          * Releases this bucket.
57          */
58         public void release() {
59            object = null;
60            state = S_RELEASED;
61         }
62         /**
63          * Returns the object stored in this bucket. <br/>
64          * This method may not be called when {@link #isOccupied()}
65          * is false.
66          */
67         public Object object() {
68            return object;
69         }
70         /**
71          * Returns true iff this bucket is empty.
72          */
73         public boolean isEmpty() {
74            return state == S_EMPTY;
75         }
76         /**
77          * Returns true iff this bucket is released.
78          */
79         public boolean isReleased() {
80            return state == S_RELEASED;
81         }
82         /**
83          * Returns true iff this bucket is occupied.
84          */
85         public boolean isOccupied() {
86            return state == S_OCCUPIED;
87         }
88         /**
89          * Returns true iff this bucket is not occupied.
90          */
91         public boolean isAvailable() {
92            return !isOccupied();
93         }
94      }
```

```
95    /**
96     * Default table capacity.
97     */
98    protected static final int DEFAULT_CAPACITY = 17;
99    /**
100    * Default maximum value of load factor.
101    */
102   protected static final float DEFAULT_LOAD = 0.67f;
103   /**
104    * The capacity of this hash table.
105    */
106   protected int capacity;
107   /**
108    * Number of objects in this hash table.
109    */
110   protected int size;
111   /**
112    * The bucket array.
113    */
114   protected Bucket[] buckets;
115   /**
116    * The maximum value of load factor.
117    */
118   protected float maxLoad;
119   /**
120    * Default constructor. Assigns default capacity and load factor.
121    */
122   public AbstractHashTable() {
123      this(DEFAULT_CAPACITY, DEFAULT_LOAD);
124   }
125   /**
126    * Parametrised constructor.
127    * @param capacity The initial capacity of this hash table.
128    * @param load The maximum load factor allowed.
129    */
130   public AbstractHashTable(int capacity, float load) {
131      buckets = new Bucket[this.capacity = capacity];
132      init(buckets);
133      this.maxLoad = load;
134      size = 0;
135   }
136   public boolean add(Object object) {
137      if (object == null) throw new IllegalArgumentException();
138      // If overloaded, rehash
139      if ((1.0 + size) / capacity > maxLoad) {
140         Bucket[] tmp = buckets;
```

```
141             buckets = new Bucket[capacity *= 2];
142             init(buckets);
143             for (int i = 0; i < tmp.length; ++i) {
144                 if (tmp[i].isAvailable()) continue;
145                 int p = indexOf(tmp[i].object());
146                 buckets[-p-1].store(tmp[i].object);
147             }
148         }
149         // Locate the bucket object should be stored at.
150         int p = indexOf(object);
151         // The object is already in the collection
152         if (p >= 0) return false;
153         // The object must be stored in bucket -p-1
154         buckets[-p-1].store(object);
155         ++size;
156         return true;
157     }
158     public boolean remove(Object object) {
159         if (object == null) throw new IllegalArgumentException();
160         // Locate the bucket object should be stored at.
161         int p = indexOf(object);
162         // The object is already in the collection; return.
163         if (p < 0) return false;
164         // The object is in bucket p; release the bucket.
165         buckets[p].release();
166         --size;
167         return true;
168     }
169     public boolean contains(Object object) {
170         return object == null ? false : indexOf(object) >= 0;
171     }
172     public int size() {
173         return size;
174     }
175     public boolean isEmpty() {
176         return size == 0;
177     }
178     public void clear() {
179         init(buckets);
180         size = 0;
181     }
182     protected final void init(Bucket[] buckets) {
183         for (int i=0; i < buckets.length; ++i) buckets[i] = new Bucket();
184     }
185     /**
186      * Returns the index of the bucket an object is stored in.
```

```
187       * If the object is not in the table, this method must return
188       * -i-1, where i is the index of the
189       * bucket the object should be stored.
190       * @param object the object to search for.
191       * @return The position the object is (or should be) stored at.
192       */
193      protected abstract int indexOf(Object object);
194  }
```

Ασκήσεις

8-1 Εκτελέστε το σενάριο του Σχήματος 8-5 χρησιμοποιώντας αλυσίδωση για την επίλυση συγκρούσεων.

8-2 Σχεδιάστε τον πίνακα κατακερματισμού που προκύπτει μετά την εισαγωγή των στοιχείων **12 33 44 17 21 7 9 56 88 66 94 11 89 101**. Το μήκος του πίνακα κατακερματισμού είναι 16 και η συμπίεση διεύθυνσης γίνεται με τη χρήση των τεσσάρων λιγότερο σημαντικών δυαδικών ψηφίων της τιμής κατακερματισμού.

8-3 Επαναλάβετε την Άσκηση 8-2 χρησιμοποιώντας γραμμική δοκιμή για την επίλυση συγκρούσεων.

8-4 Υλοποιήστε ένα πίνακα κατακερματισμού που διαχειρίζεται συγκρούσεις με τη μέθοδο της γραμμικής δοκιμής.

8-5 Επαναλάβετε την Άσκηση 8-2 για ένα πίνακα κατακερματισμού μήκους 11, χρησιμοποιώντας διπλό κατακερματισμό με τη χρήση της συνάρτησης $d(h) = 1 + h \bmod 7$.

8-6 Σε ένα πίνακα κατακερματισμού μήκους 13 οι θέσεις που διαιρούνται από το 5 καθώς και όλες οι περιττές θέσεις είναι κατειλημμένες. Ορίστε την ακολουθία θέσεων δοκιμής, για ένα αντικείμενο με τιμή κατακερματισμού -516 το οποίο δεν είναι ήδη αποθηκευμένο στον πίνακα. Η συμπίεση διεύθυνσης γίνεται με τη μέθοδο της διαίρεσης και η επίλυση συγκρούσεων με γραμμική δοκιμή.

8-7 Οι θέσεις 0, 1, 2, 7 και 8 ενός πίνακα κατακερματισμού μήκους 13 είναι κατειλημμένες. Ορίστε την ακολουθία θέσεων δοκιμής, για την αναζήτηση ενός αντικειμένου με τιμή κατακερματισμού 132 το οποίο δεν είναι ήδη αποθηκευμένο στον πίνακα. Η συμπίεση διεύθυνσης γίνεται με τη μέθοδο της διαίρεσης και η επίλυση συγκρούσεων με διπλό κατακερματισμό με τη χρήση της συνάρτησης $d(h) = 1 + h \bmod 7$.

8-8 Υλοποιήστε ένα πίνακα κατακερματισμού που διαχειρίζεται συγκρούσεις με τη μέθοδο του διπλού κατακερματισμού.

8-9 Υλοποιήστε ένα πίνακα κατακερματισμού που διαχειρίζεται συγκρούσεις με τη μέθοδο της αλυσίδωσης.

8-10 Υλοποιήστε ένα απαριθμητή συλλογής που υλοποιείται με πίνακα κατακερματισμού που χρησιμοποιεί αλυσίδωση για την επίλυση συγκρούσεων.

8-11 Σχεδιάστε και υλοποιήστε ένα πρόγραμμα το οποίο συγκρίνει πειραματικά την αποτελεσματικότητα των υλοποιήσεων πινάκων κατακερματισμού που παρουσιάστηκαν σε αυτό το κεφάλαιο.

8-12 Υλοποιήστε ένα απαριθμητή συλλογής που υλοποιείται με πίνακα κατακερματισμού ανοικτής διευθυνσιοδότησης.

ΚΕΦΑΛΑΙΟ 9

Απαντήσεις ασκήσεων

΄Ασκηση 1-2. Έστω S_N το άθροισμα των αριθμών στο διάστημα $[1, N]$, το οποίο δίνεται από τη σχέση (1.9) στη σελίδα 6. Υπολογίσουμε το άθροισμα των αριθμών του πίνακα· έστω ότι αυτό είναι s. Ο ζητούμενος αριθμός είναι $S_N - s$.

΄Ασκηση 1-18. Αν ο n είναι πρώτος, τότε είναι ένα τετριμένο γινόμενο πρώτων. Διαφορετικά ο n έχει παράγοντες εξ΄ ορισμού: $n = m_1 \times m_2 \times \cdots \times m_k$ με $k \geq 2$ και $m_i < n$. Αφού οι παράγοντες m_i είναι μικρότεροι του n, τότε μπορούν να γραφούν ως γινόμενα πρώτων αριθμών, και κατά συνέπεια, το ίδιο και ο n.

΄Ασκηση 1-23. Αρχικά επιλύουμε με την τηλεσκοπική μέθοδο.

$$\begin{aligned} C_N &= C_{N-1} + \ln N \\ &= C_{N-2} + \ln(N-1) + \ln N \\ &= C_{N-3} + \ln(N-2) + \ln(N-1) + \ln N \\ &= \vdots \\ &= C_0 + \ln 2 + \ln 3 + \cdots + \ln N \\ &= \sum_{i=1}^{N} \ln i. \end{aligned}$$

Στη συνέχεια προσεγγίζουμε το τελευταίο άθροισμα μέσω ολοκληρώματος και έχουμε τελικά

$$\sum_{i=1}^{N} \ln i \approx \int_{1}^{N} \ln x \, dx = [x \ln x - x]_1^N = N \ln N - N + 1.$$

΄Ασκηση 1-25. Μπορούμε να αντικαταστήσουμε τον έλεγχο `i < a.length` στη γραμμή 3 με τον έλεγχο `a[i] != x`. Όταν η τιμή της τοπικής μεταβλητής `i` γίνει ίση με `a.length` θα σημανθεί μια εξαίρεση τύπου `IndexOutOfBoundsException` την οποία ωστόσο μπορούμε να παγιδεύσουμε και να επιστρέψουμε την τιμή -1.

```
public static final int indexOf(int x, int[] a) {
   int i = 0;
   try {
      while (a[i] != x) ++i;
   }
   catch (IndexOutOfBoundsException e) {
      return -1;
   }
   return i;
}
```

΄Ασκηση 1-28. Δεδομένου ότι η ακολουθία είναι ταξινομημένη, δεν χρειάζεται να εξαντλούμε την συστοιχία. Μόλις εντοπίσουμε ένα στοιχείο με τιμή μεγαλύτερη από την τιμή της παραμέτρου `x`, μπορούμε να τερματίσουμε τον αλγόριθμο.

```
public static final int indexOf(int x, int[] a) {
   int i = 0;
   try {
      while (a[i] < x) ++i;
   }
   catch (IndexOutOfBoundsException e) {
      return -1;
   }
   return a[i] == x ? i : -1;
}
```

Στην καλύτερη και στην χειρότερη περίπτωση ο παραπάνω αλγόριθμος θα εκτελέσει τον ίδιο αριθμό συγκρίσεων με τον Κώδικα 1.2. Σε ορισμένες περιπτώσεις όμως, θα τερματίσει κάνοντας λιγότερες συγκρίσεις, όπως προκύπτει από την παρακάτω ανάλυση.

Αν η ζητούμενη τιμή υπάρχει στη συστοιχία, τότε θεωρώντας ότι η πιθανότητα να βρίσκεται σε οποιαδήποτε θέση είναι $p = \frac{1}{N}$, ο αναμενόμενος αριθμός συγκρίσεων είναι

$$A_1(N) = \frac{1}{N} \sum_{i=0}^{N-1} (i+2) = \frac{N+1}{2}. \tag{9.1}$$

Προκειμένου να υπολογίσουμε τον αναμενόμενο αριθμό συγκρίσεων στην περίπτωση κατά την οποία η ζητούμενη τιμή δεν βρίσκεται στη συστοιχία, ορίζουμε ως δ_i το σύνολο των ακεραίων y για τους οποίους ισχύει $a_{i-1} < y < a_i$, $i \in [1, N-1]$. Ορίζουμε επίσης ως δ_0 το σύνολο των ακεραίων y οι οποίοι είναι μικρότεροι από το πρώτο στοιχείο της συστοιχίας και δ_N το σύνολο των ακεραίων y οι οποίοι είναι μεγαλύτεροι από το τελευταίο στοιχείο της συστοιχίας. Εφόσον η ζητούμενη τιμή δεν υπάρχει στη συστοιχία, αυτή θα ανήκει σε ένα από τα σύνολα δ_i, $i \in [0, N]$. Ο παρακάτω πίνακας δίνει τον αριθμό συγκρίσεων που απαιτούνται σε κάθε περίπτωση.

δ_0	δ_1	δ_2	$\cdots$	δ_{N-2}	δ_{N-1}	δ_N
2	3	4	$\cdots$	N	$N+1$	$N+1$

Έτσι ο αναμενόμενος αριθμός συγκρίσεων στην περίπτωση που η ζητούμενη τιμή δεν υπάρχει στη συστοιχία είναι:

$$\begin{aligned} A_2(N) &= \frac{1}{N+1} \sum_{i=0}^{N-1} (i+2) + \frac{1}{N+1} (N+1) \\ &= \frac{N}{N+1} \frac{N+3}{2} + 1. \end{aligned} \tag{9.2}$$

Συνδυάζοντας τις (9.1) και (9.1) και δεδομένου πως η πιθανότητα το ζητούμενο στοιχείο να μην υπάρχει στη συστοιχία είναι $q = 1 - p$, λαμβάνουμε τον αναμενόμενο αριθμό συγκρίσεων

$$A(N) = p\frac{N+1}{2} + q\left(\frac{N}{N+1}\frac{N+3}{2} + 1\right). \tag{9.3}$$

Στην περίπτωση που η συστοιχία δεν είναι ταξινομημένη σε αύξουσα διάταξη, ο αναμενόμενος αριθμός συγκρίσεων είναι

$$A(N) = p\frac{N+1}{2} + qN. \tag{9.4}$$

΄Ασκηση 1-38. Η μέθοδος δεν εξασφαλίζει ότι κάθε αναδρομική κλήση λύνει ένα μικρότερο πρόβλημα καθώς αν η τιμή της τυπικής παραμέτρου n δεν είναι 1 ή πολλαπλάσιο του 5, η επόμενη αναδρομική κλήση ενδεχομένως θα πρέπει να λύσει ένα μεγαλύτερο πρόβλημα.

΄Ασκηση 2-14. Το σύνολο $\mathbb{C}$ των μιγαδικών αριθμών περιλαμβάνει τους αριθμούς της μορφής $x = a + b\,i$ όπου $a, b \in \mathbb{R}$ και i συμβολίζει τη φανταστική μονάδα η οποία ορίζεται ως $i = \sqrt{-1}$. Οι πραγματικοί a και b ονομάζονται πραγματικό και φανταστικό μέρος του x αντίστοιχα. Επιπλέον στο σύνολο $\mathbb{C}$, ισχύουν οι παρακάτω σχέσεις.

$$\begin{aligned}
(a + b\,i) + (c + d\,i) &= (a + c) + (b + d)\,i \\
(a + b\,i) - (c + d\,i) &= (a - c) + (b - d)\,i \\
(a + b\,i)(c + d\,i) &= (ac - bd) + (ad + bc)\,i \\
\frac{a + b\,i}{c + d\,i} &= \frac{(ac + bd) + (bc - ad)\,i}{c^2 + d^2} \\
\overline{x} &= (a, -b) \\
|x| &= a^2 + b^2 \\
\mathrm{re}\,(x) &= a \\
\mathrm{im}\,(x) &= b
\end{aligned}$$

Αν $x = a + b\,i$, ο μιγαδικός αριθμός $\overline{a - b\,i}$ ονομάζεται συζυγής (conjugate) του x, ο πραγματικός αριθμός $|x|$ ονομάζεται μέτρο (norm) του x, ενώ η ποσότητα $\sqrt{|x|}$ ονομάζεται απόλυτη τιμή του x. Ο αφηρημένος τύπος δεδομένων `Complex` παρέχει τις αντίστοιχες λειτουργίες. Τα αντικείμενα του τύπου αυτού είναι μεταβλητά καθώς οι μέθοδοι που υλοποιούν την αριθμητική μιγαδικών αριθμών (`add`, `subtract` κτλ.) τροποποιούν την κατάσταση του αντικειμένου μέσω του οποίου καλούνται.

```
1 /**
2  * A complex number x = a + b * i is a pair (a, b) where a is
3  * called the real part (re), b is called the imaginery part (im)
4  * an i represents the square root of -1.
5  */
6 public final class Complex implements Cloneable {
7    /**
8     * The real part of this complex number.
```

```
9       */
10      private double re;
11      /**
12       * The imaginary part of this complex number.
13       */
14      private double im;
15      /**
16       * Factory method that creates a random complex number.
17       * @return A random complex.
18       */
19      public static Complex newRandomInstance() {
20         int sign = Math.random() > 0.5 ? 1 : -1;
21         double x = Math.random() * Short.MAX_VALUE;
22         double y = Math.random() * Short.MAX_VALUE;
23         return new Complex(sign * x, y);
24      }
25      /**
26       * Default constructor.
27       */
28      public Complex() {
29         this(0.0, 0.0);
30      }
31      /**
32       * Copy constructor.
33       * Creates an instance by copying another instance.
34       * @param c The complex number to copy.
35       */
36      public Complex(Complex c) {
37         this(c.re, c.im);
38      }
39      /**
40       * Creates a new complex number.
41       * @param re The real part.
42       * @param im The imaginary part.
43       */
44      public Complex(double re, double im) {
45         this.re = re;
46         this.im = im;
47      }
48      /**
49       * Returns the real part of this complex number.
50       * @return The real part of this complex number.
51       */
52      public double re() {
53         return re;
54      }
```

```
55    /**
56     * Returns the imaginary part of this complex number.
57     * @return The imaginary part of this complex number.
58     */
59    public double im() {
60       return im;
61    }
62    /**
63     * Returns the conjugate of this complex number.
64     * @return The conjugate of this complex number.
65     */
66    public Complex conjugate() {
67       return new Complex(re, -im);
68    }
69    /**
70     * Adds a complex number to this complex number.
71     * @param c The complex number to add.
72     * @return A new complex which is the result of addition.
73     */
74    public Complex add(Complex c) {
75       re += c.re;
76       im += c.im;
77       return this;
78    }
79    /**
80     * Subtract a complex number from this complex number.
81     * @param c The number to subtract.
82     * @return A new complex which is the result of subtraction.
83     */
84    public Complex subtract(Complex c) {
85       re -= c.re;
86       im -= c.im;
87       return this;
88    }
89    /**
90     * Multiplies this complex number by another complex number.
91     * @param c The number to multiply by.
92     * @return A new complex which is the result of multiplication.
93     */
94    public Complex multiply(Complex c) {
95       re = re*c.re - im*c.im;
96       im = re*c.im + im*c.re;
97       return this;
98    }
99    /**
100    * Divides this complex number by another complex number.
```

```
101      * @param c The number to divide by.
102      * @return A new complex which is the result of division.
103      */
104     public Complex divide(Complex c) {
105        double d = c.re*c.re + c.im*c.im;
106        re = (re*c.re + im*c.im)/d;
107        im = (im*c.re - re*c.re)/d;
108        return this;
109     }
110     /**
111      * Changes the sign of this complex number.
112      * @return The opposite of this complex.
113      */
114     public Complex minus() {
115        re *= -1.0;
116        im *= -1.0;
117        return this;
118     }
119     /**
120      * Returns the absolute value of this complex number.
121      * @return The absolute value of this complex number.
122      */
123     public double abs() {
124        return Math.sqrt(re * re + im * im);
125     }
126     /**
127      * Returns the norm (a.k.a. the absolute square) of this complex number.
128      * @return The norm of this complex number.
129      */
130     public double norm() {
131        return re*re + im*im;
132     }
133     /**
134      * @see Object#equals(java.lang.Object)
135      */
136     public boolean equals(Object o) {
137        if (this == o) return true;
138        if (!(o instanceof Complex)) return false;
139        Complex c = (Complex) o;
140        return re == c.re && im == c.im;
141     }
142     /**
143      * @see Object#hashCode()
144      */
145     public int hashCode() {
146        long v = Double.doubleToLongBits(abs());
```

```
147         return (int) (v ^ (v >>> 32));
148      }
149      /**
150       * @see java.lang.Object#clone()
151       */
152      protected Object clone() {
153         try {
154            return super.clone();
155         }
156         catch (CloneNotSupportedException e) {
157            throw new InternalError();
158         }
159      }
160      /**
161       * @see java.lang.Object#toString()
162       */
163      public String toString() {
164         StringBuffer buffer = new StringBuffer();
165         buffer.append(Double.toString(re));
166         buffer.append("+");
167         buffer.append(Double.toString(im));
168         buffer.append("i");
169         return buffer.toString();
170      }
171   }
```

΄Ασκηση 2-16. Η κλάση CharArrayComparator, διατάσσει δύο αντικείμενα τύπου char[] με βάση τους λεξικογραφικούς κανόνες.

```
1  import aueb.util.api.Comparator;
2
3  public class CharArrayComparator implements Comparator {
4     public int compare(Object x, Object y) {
5        char[] a = (char[]) x;
6        char[] b = (char[]) y;
7        for (int i=0, j=0; i < a.length && j < b.length; i++, j++) {
8           if (a[i] != b[j]) {
9              return a[i] - a[j];
10          }
11       }
12       return a.length - b.length;
13    }
14 }
```

΄Ασκηση 2-17. Η υλοποίηση είναι γενικά όπως έχει περιγραφεί στο Κεφάλαιο 2. Αξίζει ωστόσο κανείς να προσέξει τη διαφοροποίηση στην υλοποίηση της μεθόδου `clone()`. Για το ρόλο της μεθόδου `hashCode()` θα πρέπει κανείς να ανατρέξει στο Κεφάλαιο 8.

```
1  /**
2   * A Point is a pair of integer coordinates.
3   */
4  public class Point implements Comparable, Cloneable {
5     /**
6      * The x-coordinate of this point.
7      */
8     private int x;
9     /**
10     * The y-coordinate of this point.
11     */
12    private int y;
13    /**
14     * Factory method that creates a random point.
15     * @return A random point.
16     */
17    public static Point newRandomInstance() {
18       int x = (int) (Math.random() * Short.MAX_VALUE);
19       int y = (int) (Math.random() * Short.MAX_VALUE);
20       return new Point(x, y);
21    }
22    /**
23     * Default constructor.
24     * Creates a new Point with x- and y-coordinates set to zero.
25     */
26    public Point() {
27       super();
28    }
29    /**
30     * Copy constructor.
31     * Creates a Point instance by copying a given point.
32     * @param p The point to copy
33     */
34    public Point(Point p) {
35       this(p.x, p.y);
36    }
37    /**
38     * Creates a new Point instance with given coordinates.
39     * @param x The x-coordinate.
40     * @param y The y-coordinate.
41     */
```

```
42    public Point(int x, int y) {
43       super();
44       this.x = x;
45       this.y = y;
46    }
47    /**
48     * Reads the value of the x-coordinate of this point.
49     * @return The x-coordinate of this point.
50     */
51    public int getX() {
52       return x;
53    }
54    /**
55     * Sets this point's x-coordinate.
56     * @param x The x-coordinate.
57     */
58    public void setX(int x) {
59       this.x = x < 0 ? 0 : x;
60    }
61    /**
62     * Reads the value of the y-coordinate of this point.
63     * @return The y-coordinate of this point.
64     */
65    public int getY() {
66       return y;
67    }
68    /**
69     * Sets this point's y-coordinate.
70     * @param y The y-coordinate.
71     */
72    public void setY(int y) {
73       this.y = y < 0 ? 0 : y;
74    }
75    /**
76     * Calculates the distance between this point and a given point.
77     * The result is rounded to the nearest integer.
78     * @param p The given point.
79     * @return The distance between this point and a given point.
80     */
81    public int distance(Point p) {
82       int dx = x - p.x;
83       int dy = y - p.y;
84       return (int) (Math.sqrt(dx * dx + dy * dy) + 0.5);
85    }
86    /**
87     * Moves this point by given offsets. Parameters dx and dy
```

```
88      * may be negative but the resulting coordinates may not.
89      * @param dx The x-offset.
90      * @param dy The y-offset.
91      * @return This point.
92      */
93     public Point move(int dx, int dy) {
94        setX(x + dx);
95        setY(y + dy);
96        return this;
97     }
98     /**
99      * @see Object#equals(java.lang.Object)
100     */
101    public boolean equals(Object o) {
102       if (this == o) {
103          return true;
104       }
105       if (o == null) {
106          return false;
107       }
108       if (getClass() != o.getClass()) {
109          return false;
110       }
111       Point p = (Point) o;
112       return this.x == p.x && this.y == p.y;
113    }
114    /**
115     * @see Object#hashCode()
116     */
117    public int hashCode() {
118       return x + 17 * y;
119    }
120    /**
121     * @see java.lang.Object#clone()
122     */
123    protected Object clone() {
124       try {
125          return super.clone();
126       }
127       catch (CloneNotSupportedException e) {
128          throw new InternalError();
129       }
130    }
131    /**
132     * @see java.lang.Object#toString()
133     */
```

```
134     public String toString() {
135        StringBuffer buffer = new StringBuffer();
136        buffer.append('(');
137        buffer.append(Integer.toString(x));
138        buffer.append(',');
139        buffer.append(Integer.toString(y));
140        buffer.append(')');
141        return buffer.toString();
142     }
143     /**
144      * @see java.lang.Comparable#compareTo(java.lang.Object)
145      */
146     public int compareTo(Object o) {
147        Point p = (Point) o;
148        return x != p.x ? x - p.x : y - p.y;
149     }
150  }
```

΄Ασκηση 6-10. Αρκεί να υλοποιήσουμε την κλάση AdaptiveSearchTree η οποία κληρονομεί την κλάση BinarySearchTree και ακυρώνει την μέθοδο locate(Object), έτσι ώστε να εκτελεί μια περιστροφή με άξονα τον πατέρα του κόμβου στον οποίο διαπιστώνεται η επιτυχία ή η αποτυχία της αναζήτησης.

```
1  package aueb.util.imp;
2
3  import aueb.util.imp.BinarySearchTree;
4
5  public class AdaptiveBinarySearchTree extends BinarySearchTree {
6     protected Node locate(Object element) {
7        Node p = root, q = null;
8        while (p != null) {
9           int result = cmp.compare(element, p.element);
10          if (result == 0) {
11             pushup(p);
12             return p;
13          }
14          q = p;
15          p = result < 0 ? p.left : p.right;
16       }
17       pushup(q);
18       return p;
19    }
20    private final void pushup(Node node) {
```

```
21          if (node == null || node.parent == null) {
22              return;
23          }
24          if (node == node.parent.left) {
25              rotateRight(node.parent);
26          }
27          else {
28              rotateLeft(node.parent);
29          }
30      }
31  }
```

Η υλοποίηση της μεθόδου `locate(Object)` χρησιμοποιεί μια επιπλέον αναφορά `q` η οποία δείχνει τον κόμβο στον οποίο διαπιστώνεται αποτυχία της αναζήτησης. Η περιστροφή του κόμβου `p` ή `q` γίνεται με μια κλήση στη βοηθητική μέθοδο `pushup`.

Η κλάση `AdaptiveSearchTree` υλοποιεί ένα δυαδικό δέντρο αναζήτησης το οποίο έχει την τάση να μεταφέρει κοντά στη ρίζα του δέντρου, τα αντικείμενα εκείνα τα οποία αναζητούνται πιο συχνά. Μια τέτοια δομή δεδομένων είναι πολύ χρήσιμη σε εφαρμογές στις οποίες η κατανομή της συχνότητας αναζήτησης αντικειμένων δεν είναι ομοιόμορφη.

Βιβλιογραφία

[1] Arnold, K., Gosling, J. and Holmes, D. *The Java Programming Language Third Edition*. Addison Wesley, 2000.

[2] Baase, S. *Computer Algorithms: Introduction to Design and Analysis Second Edition*. Addison Wesley, 1989.

[3] Bell, J. and Gupta, G. *An Evaluation of Self-adjusting Binary Search Tree Techniques*. Software—Practice and Experience, Vol. 23. No. 4, April 1993, pp. 369–382.

[4] Gamma E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*. Addison Wesley, 1995.

[5] Goodrich, T. and Tamassia, R. *Data Structures and Algorithms in Java Third Edition*. John Wiley & Sons, Inc., 2004.

[6] Knuth, D. *The Art of Computer Programming Volume 1: Fundamental Algorithms Second Edition*. Addison Wesley, 1973.

[7] Knuth, D. *The Art of Computer Programming Volume 3: Sorting and Searching Second Edition*. Addison Wesley, 1998.

[8] Sedgewick, R. *Algorithms in Java Third Edition*. Addison Wesley, 2003.

[9] Sleator, D. and Tarjan, R. *Self-Adjusting Binary Search Trees*. Journal of the Association for Computing Machinery, Vol. 32, No. 3, July 1985, pp. 652–686.

[10] Sun Microsystems, Inc. *The Java Collections Framework*. `http://java.sun.com/j2se/1.4.2/docs/guide/collections/index.html`

Ευρετήριο

C

D

L

M

O

P

Q

R

S

T

W

A

Γ

Δ

E

Ο

Π

Ρ

Σ

Τ

Υ

Φ

Ψ