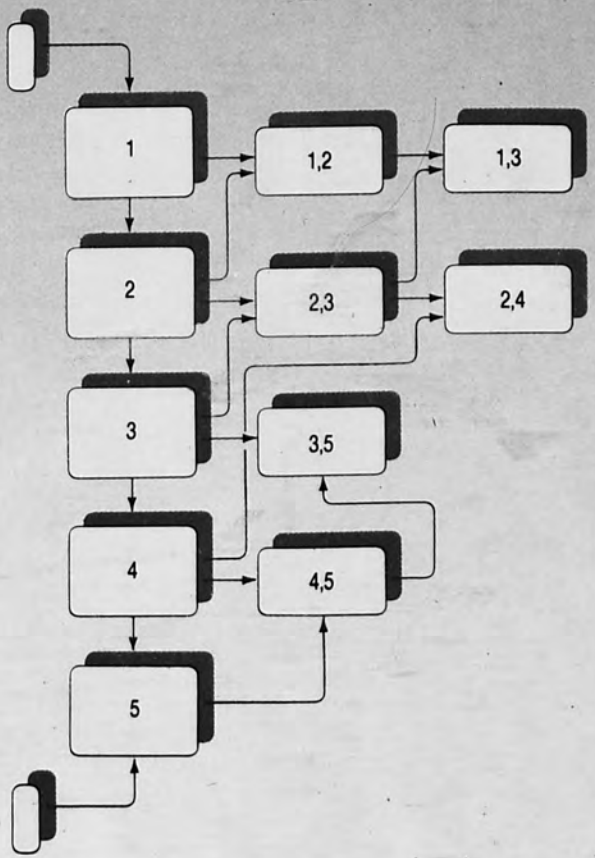
ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ ΜΕ *Pascal*



Θεοδώρου Ζ. Καλαμπούκη

Θεοδώρου Ζ. Καλαμπούκη
ΑΝΑΠΛΗΡΩΤΗ ΚΑΘΗΓΗΤΗ
ΟΙΚΟΝΟΜΙΚΟΥ ΠΑΝΕΠΙΣΤΗΜΙΟΥ ΑΘΗΝΩΝ

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ ΜΕ *Pascal*

ΤΟΜΟΣ Ι

ΕΚΔΟΣΗ 2α ΑΘΗΝΑ 1991

Κάθε γνήσιο αντίγραφο φέρει την υπογραφή του συγγραφέα.

Φιλοτέχνηση εξωφύλλου: Εκτωρ Χαραλάμπους

Στη μνήμη
του πατέρα μου

ΠΡΟΛΟΓΟΣ

Το βιβλίο αυτό αποτελεί το πρώτο μέρος μιάς σειράς μαθημάτων τα οποία δίδαξα κατά τα τελευταία τέσσερα χρόνια στους δευτεροετείς φοιτητές του τμήματος "Στατιστικής και Πληροφορικής" της ΑΣΟΕΕ.

Κατά τα χρόνια αυτά είχα την δυνατότητα να συνεργαστώ με τους φοιτητές που μου έδωσαν την ευκαιρία να επεξεργαστώ κατάλληλα τις λεπτομέρειες, που εμφανίζονται όταν παράλληλα με τη θεωρία ελέγχει κανείς τα συγκεκριμένα προβλήματα στον υπολογιστή. Η έντονη επεξεργασία των διαφόρων θεμάτων καθ'όλη την παραπάνω χρονική διάρκεια συνετέλεσε ώστε τα υπό συζήτηση θέματα να έχουν πάρει σχεδόν μιά μορφή που αντιπροσωπεύει τις πιό σύγχρονες αντιλήψεις στον προγραμματισμό.

Η επιλογή της ύλης έγινε κατά τέτοιο τρόπο, ώστε να καλύπτει όλες σχεδόν τις βασικές δομές δεδομένων και τις τεχνικές για την ανάπτυξη δομημένου προγραμματισμού. Οι τεχνικές αυτές βελτιώνουν οπωσδήποτε την ποιότητα των προγραμμάτων, αλλά και απλουστεύουν τον έλεγχο της ορθότητας και της συντήρησης αυτών. Η παρουσίαση όλων των δομών του βιβλίου έγινε με τη μορφή των Αφηρημένων Τύπων Δεδομένων (Abstract Data Types), παρ'όλο που η γλώσσα Pascal δεν υποστηρίζει τέτοιους τύπους. Ο βασικός σκοπός μου είναι κατά πρώτο λόγο να παρουσιάσω όλες τις δομές και συγχρόνως να περάσω στον αναγνώστη την ιδέα της λογικής αφαίρεσης.

Η περιγραφή της κάθε δομής δεδομένων εισάγεται με τον ορισμό της στη γλώσσα Pascal ακολουθούμενο από τις διαδικασίες που ορίζουν τις βασικές πράξεις της. Με τον τρόπο αυτό ο αναγνώστης έχει μια σύντομη και ολοκληρωμένη εικόνα της δομής και μπορεί εύκολα να τη χρησιμοποιήσει σε πιο σύνθετα προγράμματα. Είναι επίσης σε θέση να συγκρίνει τις διάφορες δομές ως προς την πολυπλοκότητά τους, ώστε να μπορεί να επιλέξει την καταλληλότερη δομή για την κάθε περίπτωση.

Για την κατανόηση του βιβλίου αυτού απαιτούνται μόνο γνώσεις προγραμματισμού με τη γλώσσα Pascal. Για ορισμένες έννοιες, που απαιτούνται ίσως περισσότερες μαθηματικές ή άλλες γνώσεις, ο αναγνώστης μπορεί να εμβαθύνει ανατρέχοντας στη ξένη βιβλιογραφία που προτείνεται στο τέλος του βιβλίου. Για την ευκολότερη ανάγνωση των προγραμμάτων, χρησιμοποιήθηκαν ελληνικά ονόματα για τις μεταβλητές.

Το Κεφάλαιο 1 του βιβλίου δίνει τις απαραίτητες έννοιες και ορισμούς που απαιτούνται για την ανάγνωση των όσων ακολουθούν. Στο Κεφάλαιο 2 εξετάζονται οι πίνακες και οι πυκνωμένοι πίνακες χαρακτήρων. Στο Κεφάλαιο 3 περιγράφονται όλοι οι τύποι γραμμικών δομών, όπως είναι οι απλές λίστες, οι κυκλικές λίστες και οι

λίστες διπλής διεύθυνσης. Στα επόμενα Κεφάλαια 4, 5, 6 και 7 περιγράφονται οι μη γραμμικές δομές, όπως είναι τα δένδρα και τα γραφήματα.

Στο Κεφάλαιο 5 συμπεριλαμβάνονται τα B_δένδρα τα οποία όμως εξετάζονται απλώς σα μια ειδική κατηγορία ισοζυγισμένων δένδρων. Τα δένδρα αυτά χρησιμοποιούνται για την οργάνωση ευρετηρίων με γρήγορη προσπέλαση στις εγγραφές ενός αρχείου. Η μελέτη όμως αυτή αναβάλλεται προς τα παρόν και απλώς εξετάζεται μόνο η δομή του B_δένδρου μαζί με τις πράξεις της. Τέλος στο Κεφάλαιο 8 εισάγουμε την έννοια της αφαίρεσης και του δομημένου προγραμματισμού με τη χρήση των πακέτων της γλώσσας Ada. Αν και το Κεφάλαιο αυτό ίσως απαιτεί ορισμένες γνώσεις της γλώσσας προγραμματισμού Ada, έγινε προσπάθεια να περιορισθούν οι απαιτήσεις αυτές στο ελάχιστο.

Για τη καλύτερη εμπέδωση της θεωρίας στο τέλος κάθε κεφαλαίου δίνονται ασκήσεις. Οι λύσεις ορισμένων ασκήσεων δίνονται στο τέλος του βιβλίου.

Ελπίζω ότι σύντομα θα δημοσιεύσω και το δεύτερο τόμο αυτής της σειράς που θα περιλαμβάνει τις δομές των αρχείων ολοκληρώνοντας έτσι ένα έργο το οποίο πιστεύω ότι θα συμβάλει στην Ελληνική βιβλιογραφία της Επιστήμης των Υπολογιστών συμπληρώνοντας το κενό που υπάρχει.

Η προετοιμασία του βιβλίου έγινε με την βοήθεια υπολογιστή. Για την πληκτρολόγηση των κειμένων οφείλω να ευχαριστήσω ιδιαιτέρως την Αθηνά Σκοπελίτη για την υπομονή της, τόσο στο γράψιμο και τις επανειλλημένες διορθώσεις των αρχικών δοκιμίων, όσο και για τη φιλοτέχνηση των σχημάτων του βιβλίου.

Για τη βοήθειά τους στο γράψιμο διαφόρων προγραμμάτων ευχαριστώ τους φοιτητές μου στο Οικονομικό Πανεπιστήμιο και ιδιαίτερα τους Λευτέρη Χατζάκη, Νάνσυ Πουλούδη και Αντώνη Κοτζαμανίδη.

Για τη βοήθεια στη διεξαγωγή των φροντιστηριακών ασκήσεων και τις λύσεις των ασκήσεων του βιβλίου ευχαριστώ τους Μεταπτυχιακούς υποτρόφους Σπύρο Μάντζαρη και Γιάννη Μήλη.

Τέλος θέλω να ευχαριστήσω ιδιαίτερα το συνάδελφό μου στο Οικονομικό Πανεπιστήμιο Γιάννη Κάβουρα, για τις πολλές υποδείξεις και διορθώσεις που έκανε τόσο στα αρχικά δοκίμια όσο και στην τελική μορφή του κειμένου.

Θόδωρος Καλαμπούκης
Ιούνιος 1989

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΕΦΑΛΑΙΟ 6
ΓΡΑΦΗΜΑΤΑ

ΚΕΦΑΛΑΙΟ 7
ΑΛΓΟΡΙΘΜΟΙ ΓΡΑΦΗΜΑΤΩΝ

ΚΕΦΑΛΑΙΟ 8
ΑΦΗΡΗΜΕΝΟΙ ΤΥΠΟΙ ΔΕΔΟΜΕΝΩΝ

ΚΕΦΑΛΑΙΟ 1

ΤΥΠΟΙ ΚΑΙ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

1.1 Πληροφορίες και δεδομένα

Πληροφορική είναι η επιστήμη που ασχολείται με την απόκτηση, μετάδοση και επεξεργασία πληροφοριών (information). Ενώ είναι εύκολο να δώσει κανείς παραδείγματα πληροφοριών, ένας αυστηρός ορισμός της λέξης "πληροφορία" είναι δύσκολο να δωθεί, καθόσον η ίδια η λέξη είναι μία αφηρημένη έννοια και όχι ένα χειροπιαστό αντικείμενο.

Γενικά πληροφορία είναι η γνώση που αποκτάμε διαβάζοντας ένα κείμενο, κοιτάζοντας μία εικόνα ή παρατηρώντας τον έξω κόσμο. Οτιδήποτε κάνουμε καθημερινά συνδέεται με πληροφορίες. Με το διάβασμα ενός βιβλίου ή το κοίταγμα μιάς φωτογραφίας συλλέγουμε πληροφορίες, δηλαδή η λέξη πληροφορία συνδέεται με μία ή περισσότερες ιδιότητες μιας κατάστασης ή ενός αντικειμένου. Οταν γράφουμε ένα γράμμα ή μιλάμε στο τηλέφωνο τότε μεταδίδουμε πληροφορίες. Τέλος η λύση ενός προβλήματος σημαίνει το μετασχηματισμό κεκτημένων γνώσεων (πληροφοριών) σε μία νέα γνώση, τη λύση του προβλήματος. Η διαδικασία αυτή συνεπάγεται επεξεργασία πληροφοριών.

Οπως βλέπουμε από τα παραπάνω παραδείγματα η Πληροφορική είναι τόσο παλιά όσο η ίδια η ζωή. Ωστόσο, όμως, σαν ξεχωριστή επιστήμη αναπτύχθηκε μόλις τα τελευταία χρόνια χάρις στους ηλεκτρονικούς υπολογιστές. Οι ηλεκτρονικοί υπολογιστές χρησιμοποιούνται σήμερα για την επεξεργασία πληροφοριών οι οποίες κατά πλειοψηφία δεν είναι αριθμητικές. Η χρησιμοποίηση και η μετάδοση τέτοιων θεωρητικών (αφηρημένων) ιδιοτήτων από τον υπολογιστή προϋποθέτει τη δυνατότητα παράστασής τους με κάποιο τρόπο.

Ως **δεδομένο (datum)** θα εννοούμε την παράσταση μιας πληροφορίας. Η παράσταση αυτή θα πρέπει να είναι μοναδική, εκτός αν θεωρήσουμε περισσότερες από μια μεθόδους παράστασης. Για παράδειγμα ο "αριθμός δώδεκα", είναι μια πληροφορία η οποία

μπορεί να παρασταθεί με διάφορους τρόπους ανάλογα με το σύστημα αρίθμησης:

1100	Δυαδικό
12	Δεκαδικό
14	Οκταδικό
XII	Ρωμαικό

Αν και όλες οι παραστάσεις του δώδεκα περιέχουν την ίδια πληροφορία, η υλοποίησή τους εξαρτάται από το σύστημα αρίθμησης που χρησιμοποιούμε κάθε φορά.

Τα δεδομένα αποθηκεύονται στη μνήμη του υπολογιστή η οποία είναι μια γραμμική ακολουθία **ψηφιοσυλλαβών (bytes)**. Για το λόγο αυτό η πιο κατάλληλη μέθοδος για την παράσταση των δεδομένων είναι η **δυαδική (binary representation)**. Για την παράσταση δεδομένων υπάρχουν ειδικοί κώδικες όπως οι ASCII, BCD κ.α.(*).

Γενικά μπορούμε να πούμε ότι η μελέτη οποιουδήποτε θέματος της επιστήμης των υπολογιστών προυποθέτει την αποθήκευση και την επεξεργασία πληροφοριών. Η αποθήκευση δεδομένων στη μνήμη του υπολογιστή απαιτεί γνώση θεμάτων που έχουν σχέση με τα **μέσα αποθήκευσης (storage devices)** και τους **μηχανισμούς διευθυνσιοποίησης (addressing mechanisms)**. Πέραν από τις γνώσεις αυτές του υλικού (hardware) οπωσδήποτε χρειάζονται και γνώσεις λογισμικού (software), όπως, για παράδειγμα του μηχανισμού **διαχείρησης της μνήμης (storage management)**(**). Ο μεταγλωττιστής (compiler) και το λειτουργικό σύστημα (operating system) συμμετέχουν στην καταχώρηση (allocation) και απελευθέρωση (deallocation) περιοχών της μνήμης που χρειάζονται για τη δημιουργία και τη διαγραφή δεδομένων αντίστοιχα. Για την καλή οργάνωση των δεδομένων απαιτούνται επίσης γνώσεις από τη θεωρία επεξεργασίας πληροφοριών. Τέλος, για την επεξεργασία των δεδομένων απαιτούνται γνώσεις από τα μαθηματικά και ειδικότερα από τη θεωρία συνόλων. Το αντικείμενο του βιβλίου αυτού αποτελεί μια τομή όλων των παραπάνω θεμάτων.

1.2 Τύποι Δεδομένων και Δομές Δεδομένων

Η έννοια του **τύπου (type)** αποδίδει ορισμένες χαρακτηριστικές ιδιότητες σε μια κατηγορία αντικειμένων. Οι ιδιότητες αυτές είναι καθοριστικές και διακρίνουν τα αντικείμενα που αφορούν σε μια

(*) Κάβουρα Ι.Κ., Συστήματα Υπολογιστών, Τόμος Ι, Αθήνα 1989, Εκδοση 2α, Κεφ. 1.
(**) Κάβουρα Ι.Κ., Συστήματα Υπολογιστών, Τόμος ΙΙ, Αθήνα 1987.

κατηγορία. Η Pascal, όπως πολλές γλώσσες προγραμματισμού, απαιτεί να έχει το κάθε δεδομένο (μεταβλητή) ενός προγράμματος τον τύπο του. Ο τύπος αυτός καθορίζει:

α) Την περιοχή των τιμών που μπορεί να πάρει η μεταβλητή καθώς και το σύνολο των πράξεων που ορίζονται πάνω σε μεταβλητές αυτού του τύπου.

β) Κάθε **τελεστής (operator)** μιας πράξης (operation) απαιτεί **ορίσματα ή τελεστέους (operands)** κάποιου συγκεκριμένου τύπου και δίνει ένα αποτέλεσμα κάποιου συγκεκριμένου τύπου επίσης.

Με άλλα λόγια, ο όρος **τύπος δεδομένων (data type)**, χρησιμοποιείται στις γλώσσες προγραμματισμού για να δηλώσει το σύνολο των τιμών που μπορεί να πάρει μια μεταβλητή με τις επιτρεπόμενες πράξεις πάνω σ'αυτές τις τιμές. Έτσι μια μεταβλητή τύπου integer μπορεί να πάρει μια οποιαδήποτε ακέραιη τιμή σ'ένα προκαθορισμένο διάστημα [- maxint, + maxint], τα όρια του οποίου καθορίζονται από το υλικό του συστήματος.

Κάθε γλώσσα προγραμματισμού έχει ένα σύνολο από **πρωτογενείς ή στοιχειώδεις (primitive) τύπους**. Το σύνολο αυτό καθορίζει και το πεδίο των προβλημάτων στα οποία μπορεί να εφαρμοστεί η γλώσσα. Στις FORTRAN IV και 77, για παράδειγμα, το σύνολο αυτό αποτελείται από τους τύπους INTEGER, REAL, LOGICAL, CHARACTER και COMPLEX, ανάλογα από την έκδοση της γλώσσας. Στην Pascal το σύνολο των στοιχειωδών τύπων περιλαμβάνει τους τύπους integer, real, boolean και char.

Ο τύπος μιας μεταβλητής προδιαγράφεται από τη δήλωσή της. Η δήλωση της μεταβλητής δίνει την δυνατότητα στο μεταγλωττιστή να ελέγχει τη συμβατότητα (compatibility) και την ορθότητα των παραστάσεων που αφορούν αυτή τη μεταβλητή. Για παράδειγμα η εκχώρηση μιας μεταβλητής τύπου integer σε μια μεταβλητή τύπου boolean μπορεί να εντοπιστεί κατά την μεταγλώττιση του προγράμματος, πριν ακόμη αυτό εκτελεστεί. Αυτό είναι ιδιαίτερα χρήσιμο για την ανάπτυξη ορθών προγραμμάτων και αποτελεί το κυριότερο πλεονέκτημα μια γλώσσας υψηλού επιπέδου έναντι της γλώσσας μηχανής ή μιας συμβολικής γλώσσας. Ενα άλλο επίσης σπουδαίο χαρακτηριστικό των τύπων είναι η διευκόλυνση που παρέχουν στην καταχώρηση της μνήμης.

Μια από τις πρώτες γλώσσες που έκαναν χρήση τύπων ήταν η ALGOL 60. Οι περισσότερες νεότερες γλώσσες προγραμματισμού δίνουν τη δυνατότητα στο χρήστη να ορίσει νέους τύπους δεδομένων. Οι τύποι αυτοί μπορεί να είναι είτε περιορισμοί υπαρχόντων απλών τύπων (π.χ. 1..100, ακέραιοι μεταξύ 1 και 100) είτε συνδυασμοί απλών τύπων (π.χ. πίνακες και εγγραφές). Όλες οι γλώσσες προγραμματισμού παρέχουν την δυνατότητα χρήσης πινάκων, ωστόσο όμως, οι πράξεις που προμηθεύουν για την επεξεργασία των πινάκων είναι διαφορετικές. Η BASIC, για παράδειγμα, διαθέτει την πράξη της πρόσθεσης πινάκων, ενώ η COBOL δε διαθέτει τέτοια πράξη.

Γλώσσες όπως οι Pascal , C και COBOL παρέχουν τη δυνατότητα ορισμού μιας εγγραφής (record) ενώ οι BASIC και FORTRAN δεν έχουν τέτοια δυνατότητα.

Αν τα στοιχεία ενός τύπου δεδομένων αποτελούνται από συσσώρευση απλών τιμών κάποιου άλλου τύπου, που έχει ήδη οριστεί, τότε ο τύπος ονομάζεται **δομημένος τύπος (structured type)**. Μ' άλλα λόγια, οι τιμές αυτών των τύπων δεδομένων μπορούν να διασπαστούν σε τιμές απλών τύπων. Τέλος, ορισμένες γλώσσες όπως η Pascal μας δίνουν τη δυνατότητα να ορίσουμε νέους απλούς (αδόμητους) τύπους. Αυτό γίνεται συνήθως με **απαρίθμηση (enumeration)** των τιμών απο τις οποίες αποτελείται ο νέος τύπος, για παράδειγμα:

```
type ημέρες = (DEY, TRI, TET, PEM, PAR, SAB, KYR)
```

Με τον τρόπο αυτό μπορούμε να ορίσουμε νέους απλούς τύπους, κατάλληλους για το πρόβλημά μας. Οι νέοι τύποι μπορούν να χρησιμοποιηθούν, με τη σειρά τους, για να ορίσουν άλλους δομημένους τύπους κ.ο.κ. Φυσικά για τον ορισμό μιας δομής πρέπει να ορίσουμε και τις πράξεις επι των στοιχείων της. Οι πράξεις αυτές ορίζονται συνήθως υπό μορφή διαδικασιών με τη βοήθεια άλλων βασικών πράξεων και δομικών μεθόδων που διαθέτουν σχεδόν όλες οι γλώσσες προγραμματισμού.

1.3 Αφηρημένοι τύποι δεδομένων (Abstract data types)

Η λογική αφαίρεση (abstraction) και οι αφηρημένοι τύποι δεδομένων παίζουν πολύ σπουδαίο ρόλο στο σύγχρονο προγραμματισμό, τόσο στη σχεδίαση, όσο και στο γράψιμο ορθών και δομημένων προγραμμάτων. Παραδείγματα (λογικής) αφαίρεσης συναντάει κανείς πολύ συχνά στο προγραμματισμό. Όλοι μας χρησιμοποιούμε με άνεση ακεραίους και πραγματικούς αριθμούς στα προγράμματά μας, χωρίς να καταλαβαίνουμε ότι αυτό οφείλεται στη λογική αφαίρεση. Πράγματι όλοι μας χρησιμοποιούμε στα προγράμματά μας τους ακεραίους όπως τους μάθαμε στα Μαθηματικά, ενώ μέσα στον υπολογιστή ένας ακέραιος είναι μια ακολουθία από δυαδικά ψηφία (bits).

Για να καταλάβουμε καλύτερα τι εννοούμε με τον όρο (λογική) αφαίρεση ας υποθέσουμε πως, σ'έναν υπολογιστή, οι αριθμητικές πράξεις γίνονται από το λογισμικό. Στην περίπτωση αυτή όταν μεταγλωττίζεται μια εντολή εκχώρησης της μορφής: X:= 5*Y+1.0, ο μεταγλωττιστής όχι μόνο θα πρέπει να καθορίσει ποιές θέσεις μνήμης θα χρησιμοποιήσει για την αποθήκευση των μεταβλητών X και Y, αλλά θα πρέπει επίσης να παράγει και τις αντίστοιχες εντολές για την πρόσθεση και τον πολλαπλασιασμό. Ωστόσο, όμως, λόγω της

αφαίρεσης "όνομα μεταβλητής" και της αφαίρεσης "πραγματικός αριθμός" που διαθέτουν όλες οι γλώσσες υψηλού επιπέδου, δε χρειάζεται ν'ανησυχούμε για τις λεπτομέρειες της αποθήκευσης και της υλοποίησης όταν γράφουμε μια εντολή εκχώρησης όπως την παραπάνω.

Η αφαίρεση είναι ένας τρόπος με τον οποίο καταλήγουμε σε μια κατάσταση, που λέγεται **απόκρυψη πληροφοριών (information hiding)**, κατά την οποία λεπτομέρειες που αφορούν την αποθήκευση των δεδομένων και την υλοποίηση διαφόρων διαδικασιών που δεν είναι αναγκαίες στον χρήστη αποκρύπτονται από αυτόν ("δεν είναι ορατές" (δεν είναι συνειδητές) από τον χρήστη). Σκοπός ενός Αφηρημένου Τύπου δεδομένων (Α.Τ.Δ.) είναι η απλούστευση του προγραμματισμού, αποκρύπτοντας ορισμένες λεπτομέρειες των πληροφοριών που δεν είναι αναγκαίες στο υπόλοιπο πρόγραμμα. Αυτό βέβαια δε σημαίνει ότι αγνοούμε παντελώς αυτές τις λεπτομέρειες, καθόσον το πρόγραμμα θα πρέπει κάποτε να γραφεί.

Ενας Α.Τ.Δ. είναι ένα είδος μαθηματικού αντικειμένου, μαζί με όλες τις πράξεις που μας επιτρέπουν να δημιουργήσουμε και να χειριστούμε στιγμιότυπα αυτού του αντικειμένου. Μ'άλλα λόγια, ένας Α.Τ.Δ. περιέχει όχι μόνο τον ορισμό του τύπου των δεδομένων, αλλά και τους ορισμούς των πράξεων επί των στοιχείων του τύπου αυτού. Οταν οριστεί και υλοποιηθεί ένας Α.Τ.Δ., τότε, όπως θα δούμε στο κεφάλαιο 8, μπορεί να πάρει την μορφή ενός πακέτου (package, unit). Τέτοια προγράμματα-πακέτα μπορούν να χρησιμοποιηθούν σε άλλα προγράμματα, χωρίς να απαιτείται γι'αυτό η επανάληψη της μεταγλώττισής τους. Δυστυχώς η Pascal δε διαθέτει πακέτα.

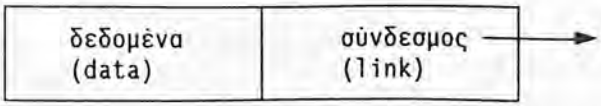
Στα επόμενα θα χρησιμοποιήσουμε έναν ομοιόμορφο τρόπο υλοποίησης των δεδομένων, ακολουθώντας την εξής σειρά:

1. Ορισμός του τύπου δεδομένων σε Pascal και
2. Ορισμός των πράξεων επί των δεδομένων και υλοποίησή των πράξεων σε Pascal υπό μορφή διαδικασιών.

Για την υλοποίηση των δομών δεδομένων υπάρχουν βασικά δύο τρόποι τους οποίους και θα ακολουθήσουμε στα επόμενα.

α) Στατική υλοποίηση, με υπολογισμό της θέσης των δεδομένων (π.χ. πίνακες) και

β) Δυναμική υλοποίηση με δείκτες (pointers). Στη περίπτωση αυτή κάθε στοιχείο ή κόμβος (node) μιας δομής θα παριστάνεται μ'ένα ορθογώνιο (που παριστάνει μια εγγραφή) με δύο βασικά πεδία. Το πεδίο των δεδομένων και το πεδίο της σύνδεσης του κόμβου με άλλους κόμβους της δομής. Σχηματικά αυτό θα παριστάνεται όπως στο παρακάτω σχήμα. Το βέλος στο πεδίο της σύνδεσης δείχνει στον επόμενο κόμβο της δομής.



1.4 Βασικές Δομές

Ενα από τα βασικά στοιχεία μιας δομής δεδομένων είναι η σχέση που υπάρχει μεταξύ των στοιχείων της. Οι κυριότερες από τις σχέσεις που θα εξετάσουμε στα επόμενα κεφάλαια είναι οι εξής:

1) **Γραμμική**, κάθε στοιχείο συνδέεται μ'ένα και μόνο ένα άλλο στοιχείο (πίνακες, λίστες).
2) **Δενδρική ή ιεραρχική**, κάθε στοιχείο συνδέεται με περισσότερα από ένα στοιχεία (δένδρα).
3) **Γραφήματα**, κάθε στοιχείο μπορεί να συνδέεται με οποιοδήποτε άλλο στοιχείο της δομής.

1.5 Αλγόριθμοι και Πολυπλοκότητα

Στα επόμενα οι βασικές πράξεις επί των δομών, που θα εξετάσουμε, θα δίνονται είτε υπό μορφή διαδικασιών ή με την περιγραφή αλγορίθμων. Υπενθυμίζουμε εδώ τον ορισμό του αλγόριθμου που δόθηκε από τον Knuth: Αλγόριθμος είναι ένα πεπερασμένο σύνολο από κανόνες βάσει των οποίων κατασκευάζουμε μια ακολουθία από πράξεις για τη λύση ενός προβλήματος. Κάθε αλγόριθμος έχει τις εξής ιδιότητες:

1) Εχει μια είσοδο για τα δεδομένα (input).
2) Εχει μια έξοδο για τα αποτελέσματα (output).
3) Είναι πεπερασμένος (finite).
4) Είναι αυστηρά ορισμένος (definite).
5) Είναι αποτελεσματικός (efficient) ή αποδοτικός.

Επειδή είναι δυνατόν για τη λύση ενός προβλήματος να χρησιμοποιηθούν περισσότεροι του ενός αλγόριθμοι, είναι αναγκαίο να έχουμε κάποια κριτήρια επιλογής του καλύτερου. Τα κριτήρια αυτά θα πρέπει να είναι τέτοια, ώστε να μπορούμε να κάνουμε μια ποιοτική ανάλυση του αλγόριθμου. Η απόδοση του αλγόριθμου αναλύεται με βάση τις διάφορες παραμέτρους του προβλήματος. Συνήθως τα σπουδαιότερα κριτήρια, μετά φυσικά την ορθότητα του αλγόριθμου, αναφέρονται στην αποτελεσματικότητά του. Η αποτελεσματικότητα είναι ένα μέτρο του χώρου της μνήμης που απαιτεί ο αλγόριθμος και του χρόνου εκτέλεσής του.

Οταν ο χρόνος εκτέλεσης ενός προγράμματος μετριέται σε

δευτερόλεπτα και ο χώρος με το πλήθος των ψηφιοσυλλαβών που καταλαμβάνει το πρόγραμμα στη μνήμη του υπολογιστή, τότε τα αποτελέσματα της ανάλυσης θα εξαρτιούνται άμεσα από τον υπολογιστή που χρησιμοποιούμε. Υπολογιστές με διαφορετική αρχιτεκτονική, και διαφορετικό σύνολο εντολών θα μας δώσουν διαφορετικά αποτελέσματα για κάποιο δεδομένο αλγόριθμο. Για το λόγο αυτό χρησιμοποιούμε για την αξιολόγιση των αλγορίθμων μεθόδους που είναι ανεξάρτητες από τον υπολογιστή. Έτσι, τα αποτελέσματα της απόδοσης των αλγορίθμων εκφράζονται συνήθως ως μια συνάρτηση του μεγέθους των δεδομένων εισόδου του προβλήματος n.

Για παράδειγμα ο χρόνος που απαιτείται για να εκτελεστεί ένας αλγόριθμος συμβολίζεται με T(n) και πολλές φορές αναφέρεται στη βιβλιογραφία ως **πολυπλοκότητα χρόνου (time complexity)**. Για πολλούς αλγόριθμους οι πολυπλοκότητες τους ως προς το χρόνο και το χώρο εξαρτιούνται από ένα ειδικό στοιχείο εισόδου, καθώς επίσης και από το πλήθος όλων των δεδομένων. Για παράδειγμα, ας θεωρήσουμε το πρόβλημα της αναζήτησης μιας συγκεκριμένης ακέραιης τιμής x μέσα σ'έναν πίνακα:

```
A: array [1..N] of integer
```

και έστω πως η αναζήτηση για τον εντοπισμό του στοιχείου x γίνεται ακολουθιακά. Είναι φανερό ότι όταν το x είναι το πρώτο στοιχείο του πίνακα, τότε θα κάνουμε μια μόνο σύγκριση, ενώ όταν το x είναι στο τέλος του πίνακα θα κάνουμε N συγκρίσεις. Για το λόγο αυτό, είναι φανερό, πως θα ήταν καλύτερα να γνωρίζουμε το μέσο όρο των συγκρίσεων που απαιτούνται για τον εντοπισμό ενός στοιχείου. Για το συγκεκριμένο παράδειγμα μπορεί να δειχθεί ότι ο μέσος όρος των συγκρίσεων είναι (N+1)/2. Έτσι, σε κάθε αλγόριθμο διακρίνουμε πολυπλοκότητα για την **καλύτερη περίπτωση (best case)**, τη **χειρότερη περίπτωση (worst case)** και τη **μέση περίπτωση (average case)**.

Για τον υπολογισμό της πολυπλοκότητας του χρόνου μετράμε συνήθως το πλήθος των αριθμητικών πράξεων που εκτελούνται από τον αλγόριθμο. Θα μπορούσαμε ίσως να συμπεριλάβουμε και άλλες πράξεις όπως για παράδειγμα λογικές πράξεις, το χρόνο προσπέλασης στα στοιχεία πινάκων, κ.λπ., αλλά εκείνο που μας ενδιαφέρει είναι η συμπεριφορά του αλγορίθμου όταν το μέγεθος των δεδομένων εισόδου μεγαλώνει. Επιπλέον, ενδιαφερόμαστε περισσότερο για τη συχνότητα μεταβολής της πολυπλοκότητας, παρά για την απόλυτη πολυπλοκότητα ενός συγκεκριμένου προγράμματος. Στο πρόβλημα της αναζήτησης, για παράδειγμα, ενδιαφερόμαστε περισσότερο να γνωρίζουμε ότι η πολυπλοκότητα του χρόνου διπλασιάζεται όταν το πλήθος των δεδομένων του προβλήματος διπλασιάζεται.

Τα αποτελέσματα της ανάλυσης της πολυπλοκότητας ενός αλγορίθμου εκφράζονται με το μαθηματικό συμβολισμό O(f(n)), όπου

f είναι μια συνάρτηση του πλήθους των δεδομένων εισόδου, ή, του προβλήματος. Ενας αλγόριθμος με χρόνο εκτέλεσης O(f(n)) θα λέμε ότι έχει πολυπλοκότητα f(n). Η συνάρτηση f ορίζεται ως εξής:

Ενας αλγόριθμος με χρόνο εκτέλεσης T(n) λέμε ότι έχει πολυπλοκότητα O(f(n)), αν και μόνο αν, υπάρχουν σταθερές c και n_0 τέτοιες ώστε:

$$T(n) \leq c\, f(n) \text{ για κάθε } n \geq n_0.$$

Οπως φαίνεται από τον ορισμό αυτό, ο συμβολισμός O(f(n)) μας δίνει ένα πάνω φράγμα του T(n) για μεγάλα n. Ετσι όταν λέμε ότι ένας αλγόριθμος έχει πολυπλοκότητα O(f(n)), εννοούμε ότι ο χρόνος εκτέλεσής του δεν μπορεί να είναι μεγαλύτερος από μια σταθερά επί f(n). Από τα παραπάνω είναι φανερό ότι ο αλγόριθμος της αναζήτησης του στοιχείου x μέσα στον πίνακα A έχει πολυπλοκότητα O(n).

Για τον καθορισμό ενός φράγματος της πολυπλοκότητας ενός αλγόριθμου, στην καλύτερη περίπτωση, εισάγουμε το συμβολισμό Ω(g(n)). Με τον ίδιο τρόπο,ορίζουμε ότι ένας αλγόριθμος με χρόνο T(n) έχει πολυπλοκότητα Ω(g(n)), όταν υπάρχει μια σταθερά τέτοια ώστε:

$$T(n) \geq c\, g(n) \text{ για άπειρες τιμές του } n.$$

Ο λόγος που η συνάρτηση Ω ορίζεται για άπειρες τιμές του n και όχι για κάθε $n \geq n_0$, οφείλεται στο ότι μερικοί αλγόριθμοι αν και είναι γρήγοροι για ορισμένες τιμές των δεδομένων εισόδου δεν είναι τόσο γρήγοροι για άλλες τιμές.

Οταν έχουμε να επιλέξουμε μεταξύ δύο αλγορίθμων με πολυπλοκότητες Cn^p και Dn^q αντίστοιχα, τότε αν $p < q$ θα λέμε ότι ο πρώτος αλγόριθμος είναι πιο αποτελεσματικός. Οπωσδήποτε όμως και οι συντελεστές C, D παίζουν κάποιο ρόλο στην απόδοση. Αν για παράδειγμα $D < C$, τότε ο πρώτος αλγόριθμος θα είναι καλύτερος από τον δεύτερο, όταν το n είναι πολύ μεγάλο. Πολλές φορές συμβαίνει ένας αλγόριθμος να είναι καλός ως προς το χρόνο εκτέλεσης, όχι όμως και ως προς το χώρο μνήμης που χρειάζεται. Συνεπώς, όταν κανείς γνωρίζει τα μέτρα αυτά της απόδοσης του αλγόριθμου, μπορεί ανάλογα με την περίπτωση να κάνει τη σωστή επιλογή της δομής, που θα χρησιμοποιήσει στο πρόβλημά του.

Εκτός από τα μέτρα που περιγράψαμε, έχουμε ανάλογα μέτρα για το κόστος που απαιτεί γενικά ο αλγόριθμος, πριν εκτελεστεί (preprocessing time). Το κόστος αυτό περιλαμβάνει το γράψιμο του προγράμματος, τον εντοπισμό και τη διόρθωση των σφαλμάτων (debugging), την ενσωμάτωση του σ' ένα υπάρχον σύστημα, καθώς και παράγοντες του περιβάλλοντος, όπως ασφάλεια, προστασία κ.λπ. Η μελέτη τέτοιων παραγόντων ανήκει στα περιεχόμενα του αντικειμένου

διαχείρισης δεδομένων (data management) γι'αυτό και δε θα μας απασχολήσει στα επόμενα.

Ανακεφαλαίωση

Στο κεφάλαιο αυτό δώσαμε τις θεμελιώδεις έννοιες και ορισμούς που είναι απαραίτητα για την ανάγνωση του υπόλοιπου βιβλίου.

Τύπος δεδομένων:	Σύνολο τιμών μαζί με τις πράξεις επί των τιμών αυτών.
Δομές Δεδομένων:	Ενας τύπος του οποίου τα στοιχεία έχουν οριστεί προηγουμένως μαζί με μια σχέση επί των στοιχείων αυτών.
Αφηρημένος τύπος:	Συγκεντρώνει τον ορισμό και τις πράξεις (encapsulation). Οι αφηρημένοι τύποι αποκρύβουν πληροφορίες από το χρήστη, βοηθούν στη γραφή ορθών προγραμμάτων, την δόμηση κατά ενότητες (modularity) και είναι ανεξάρτητοι από την υλοποίηση του αλγόριθμου.
Αλγόριθμος:	Μια μέθοδος λύσης ενός προβλήματος που εκφράζεται μ'ένα πεπερασμένο πλήθος βημάτων, είναι αυστηρά ορισμένη και αποτελεσματική με μια είσοδο για τα δεδομένα και μια έξοδο για τα αποτελέσματα.
Αξιολόγιση αλγορίθμων:	Ως προς τον χρόνο και χώρο (συμβολισμοί O και Ω).

ΚΕΦΑΛΑΙΟ 2

ΠΙΝΑΚΕΣ

2.1 Συνάρτηση απεικόνισης

Το διάνυσμα και ο πίνακας είναι δύο καθαρά μαθηματικές έννοιες. Ενα διάνυσμα x είναι ένα διατεταγμένο σύνολο από στοιχεία $x_1,x_2,\ldots,x_n$. Τα στοιχεία x_i ονομάζονται συντεταγμένες του διανύσματος x σ'ένα n-διάστατο χώρο. Ενας πίνακας είναι ένα διατεταγμένο σύνολο από διανύσματα. Στις γλώσσες προγραμματισμού αναφερόμαστε στα διανύσματα και τους πίνακες με την ειδική λέξη **array**.

Ο πίνακας είναι ένα σύνολο από στοιχεία τα οποία είναι όλα του ίδιου τύπου. Αναφορά σ'ένα στοιχείο του πίνακα γίνεται μ'ένα ή περισσότερους δείκτες, που περικλείονται σε τετραγωνικές αγκύλες, αμέσως μετά τ'όνομα του πίνακα. Το μέγεθος ενός πίνακα είναι το πλήθος των στοιχείων του. Οι πίνακες είναι μια από τις πιο διαδεδομένες δομές δεδομένων και εμφανίσθηκαν για πρώτη φορά στη γλώσσα προγραμματισμού FORTRAN. Τα στοιχεία ενός πίνακα μπορεί να είναι είτε απλά στοιχεία ή άλλες δομές δεδομένων αλλά όλα είναι του ίδιου τύπου.

Ενας πίνακας ορίζεται από τον τύπο των δεικτών του και τον τύπο των στοιχείων του, για παράδειγμα:

```
type   διάνυσμα = array[1..10] of real;
       υπερωρίες = array[εργάσιμη_ημέρα] of integer
```

Ο τύπος του δείκτη πρέπει να είναι τακτικός ή βαθμωτός. Το πλήθος των δεικτών ενός πίνακα είναι γνωστό ως **διάσταση (dimension)** του πίνακα. Η διάσταση και ο τύπος των στοιχείων ενός πίνακα γίνονται γνωστά σ'ένα σύστημα υπολογιστή κατά το χρόνο της μεταγλώττισης ενός προγράμματος (compilation time). Ενας πίνακας απεικονίζεται σε συνεχόμενες θέσεις μνήμης. Η απεικόνιση αυτή είναι μια, ένα προς ένα αντιστοιχία, μεταξύ της λογικής διάταξης ενός στοιχείου και της φυσικής του θέσης στη μνήμη. Για

παράδειγμα αν έχουμε τη δήλωση:

var A : διάνυσμα

τότε το διάνυσμα Α αποθηκεύεται στην κυρία μνήμη του υπολογιστή όπως φαίνεται στον παρακάτω πίνακα:

θέση μνήμης	τιμή δείκτη	τιμή στοιχείου
100	1	2.0
101	2	4.0
102	3	6.0
103	4	8.0
.	.	.
.	.	.
.	.	.
109	10	20.0

Η απεικόνιση αυτή μεταξύ των δεικτών του πίνακα και των θέσεων της μνήμης μας βοηθάει να καθορίσουμε εύκολα τη διεύθυνση της θέσης μνήμης στην οποία αποθηκεύεται ένα στοιχείο του πίνακα (προσπέλαση στα στοιχεία του πίνακα). Ο χρόνος προσπέλασης ενός στοιχείου του πίνακα, όπως θα δούμε, είναι ανεξάρτητος από την τιμή του δείκτη και από το μέγεθος του πίνακα. Για το λόγο αυτό λέμε ότι οι πίνακες είναι δομές **τυχαίας προσπέλασης (random access structures).**

Η απεικόνιση μεταξύ των δεικτών ενός πίνακα και των θέσεων της μνήμης ονομάζεται **Συνάρτηση Απεικόνισης του Πίνακα, Σ.Α.Π., (Array Mapping Function)** και θα τη συμβολίζουμε με F(A,<i>). Η τιμή F(A,<i>) μας δίνει την διεύθυνση της πρώτης ψηφιοσυλλαβής του στοιχείου A[i] στη μνήμη. Στην περίπτωση ενός διανύσματος, η Σ.Α.Π. είναι πολύ απλή και δίνεται από τη σχέση :

$$F(A,<i>) := F(A,<0>)+i-1$$

όπου F(A,<0>) είναι η διεύθυνση της πρώτης ψηφιοσυλλαβής του πρώτου στοιχείου του διανύσματος. Η διεύθυνση αυτή ονομάζεται και **διεύθυνση βάσης (base address)** του πίνακα.

Για παράδειγμα αν F(A,<0>)=100, τότε το στοιχείο A[4] αποθηκεύεται στη θέση:

100+4-1=103.

Τα στοιχεία ενός πίνακα μπορεί να είναι τα ίδια πίνακες, όπως για παράδειγμα στη δήλωση:

B: array[1..10] of διάνυσμα

όπου "διάνυσμα" ορίσθηκε παραπάνω. Αναφορά σ'ένα στοιχείο του πίνακα B γίνεται γράφοντας B[i][j], ή, απλούστερα, B[i,j]. Ο B είναι ένας δισδιάστατος πίνακας. Ενας δισδιάστατος πίνακας μπορεί να ορισθεί επίσης δηλώνοντας τα όρια των δεικτών του, γιά παράδειγμα:

B: array[1..10, 1..10] of real

Με τον ίδιο τρόπο μπορούμε να γενικεύσουμε σε n-διάστατους ή πολυδιάστατους πίνακες. Η Σ.Α.Π. στη περίπτωση του B είναι:

$$F(B,<i,j>) = F(B,<0,0>) + 10(j-1) + i - 1$$

όταν ο πίνακας αποθηκεύεται κατά στήλες, όπως συμβαίνει στη FORTRAN, ή

$$F(B,<i,j>) = F(B,<0,0>) + 10(i-1) + j - 1$$

όταν ο πίνακας αποθηκεύεται κατά γραμμές, όπως στη Pascal.

Φυσικά η θέση του στοιχείου ενός πίνακα στη μνήμη εξαρτάται και από τον τύπο του στοιχείου, τον μεταγλωττιστή και τον υπολογιστή. Ανάλογα με τον τύπο τους τα στοιχεία έχουν ένα ορισμένο μήκος. Οπωσδήποτε όλα τα στοιχεία του πίνακα, εφόσον είναι του ίδιου τύπου, έχουν το ίδιο μήκος (έστω L). Ετσι αν

A : array[l_1..u_1] of τύπος_στοιχείων

τότε οι παραπάνω Σ.Α.Π. στη γενική τους περίπτωση θα είναι:

$$F(A,<i>) = F(A,<0>)+(i-l_1)L = (F(A,<0>)-l_1L)+iL = c_0+c_1i$$

όπου $c_0 = F(A,<0>)-l_1L$ και $c_1 = L$.

Παρατηρούμε ότι η προσπέλαση σ'ένα στοιχείο του πίνακα A απαιτεί μία πρόσθεση και έναν πολλαπλασιασμό. Οι σταθερές c_0 και c_1 υπολογίζονται πριν ο πίνακας χρησιμοποιηθεί από το πρόγραμμα.

Με τον ίδιο τρόπο η Σ.Α.Π. του

B: array[l_1..u_1, l_2..u_2] of τύπος_στοιχείων

είναι

$$F(B,<i,j>) = F(B,<0,0>)+(u_2-l_2+1)(i-l_1)L+(j-l_2)L = c_0+c_1i+c_2j$$

όπου

$$c_2 = L$$
$$c_1 = (u_2-l_2+1)c_2$$
$$c_0 = F(B,<0,0>)-c_1l_1-c_2l_2$$

Η προσπέλαση των στοιχείων ενός δισδιάστατου πίνακα απαιτεί δύο

πολλαπλασιασμούς και δύο προσθέσεις.

Τέλος αν έχουμε ένα n-διάστατο πίνακα:

C: array[$l_1..u_1$, $l_2..u_2$, ..., $l_n..u_n$] of τύπος_στοιχείων

τότε μπορεί να αποδείξει ο αναγνώστης ότι η Σ.Α.Π. δίνεται από τη σχέση:

$$F(C, <i_1, i_2, \ldots i_n>) = c_0+c_1i_1+\ldots+c_ni_n$$

όπου $c_n = L$

$c_{k-1} = (u_k-l_k+1)c_k$ για k=n, n-1, ..., 2 και

$c_0 = F(C, <0, 0, \ldots, 0>)-c_1l_1-c_2l_2-\ldots-c_nl_n$

Στην περίπτωση αυτή η προσπέλαση των στοιχείων του πίνακα απαιτεί **n** πολλαπλασιασμούς και **n** προσθέσεις.

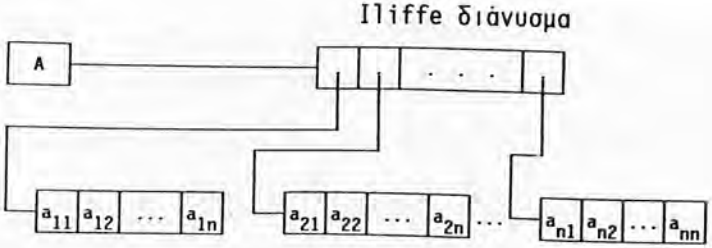
Για να προσπελάσουμε στα στοιχεία ενός πίνακα θα πρέπει να υπολογίσουμε την Σ.Α.Π. του. Τα c_i υπολογίζονται μετά τη δήλωση των ορίων των διαστάσεων του πίνακα και αποθηκεύονται σ'ένα ειδικό **διάνυσμα** που λέγεται, **"χαζό" (dope vector).**

Στο dope διάνυσμα συνήθως αποθηκεύονται ο αριθμός της διάστασης του πίνακα, τα όρια της κάθε διάστασής του, l_i και u_i i=1,...,n, που χρησιμοποιούνται για τον έλεγχο της εγκυρότητας των δεικτών και οι σταθερές c_i, i=0,...,n. Παρ'όλο που το dope διάνυσμα μας επιτρέπει τη γρήγορη προσπέλαση των στοιχείων ενός πίνακα, απαιτεί κάποιο επιπρόσθετο χώρο για την αποθήκευσή του.

Άλλες πράξεις, που συναντιούνται σ'έναν πίνακα, είναι η εισαγωγή και διαγραφή στοιχείων. Οι πράξεις αυτές είναι δύσκολες και χρονοβόρες στους πίνακες, γιατί απαιτούν μετακινήσεις των στοιχείων. Για το λόγο αυτό θα εξετάσουμε στο επόμενο κεφάλαιο άλλες δομές πινάκων που υποστηρίζουν την εισαγωγή και διαγραφή στοιχείων με κάποια ευκολία (βλέπε Ασκηση 8, Κεφ. 3).

Σημείωση:

Οταν ο υπολογιστής κάνει έμμεση προσπέλαση στη μνήμη ή έχει αργή αριθμητική, τότε χρησιμοποιούνται άλλες τεχνικές για την προσπέλαση των στοιχείων ενός πίνακα. Μια τέτοια μέθοδος είναι το Iliffe διάνυσμα. Αυτό είναι ένα βοηθητικό διάνυσμα στο οποίο αποθηκεύεται η διεύθυνση της βάσης των υποδιανυσμάτων του πίνακα. Για παράδειγμα το Iliffe διάνυσμα για ένα δισδιάστατο πίνακα περιέχει τη διεύθυνση της βάσης κάθε γραμμής. Παραστατικά έχουμε:



Είναι φανερό πως η προσπέλαση με Iliffe διανύσματα είναι ταχύτερη από αυτήν που επιτυγχάνεται με dope διανύσματα, αλλά πως απαιτείται πρόσθετος χώρος για την αποθήκευσή του διανύσματος Iliffe.

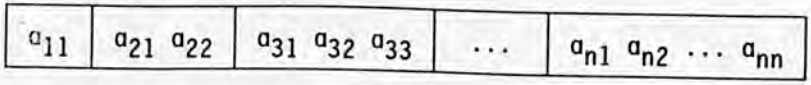
2.2 Ειδικές κατηγορίες πινάκων

Λόγω της μεγάλης χρήσης και της σπουδαιότητάς τους, οι πίνακες υπάρχουν σ'όλες τις γλώσσες προγραμματισμού. Όλα τα συστήματα παρέχουν συναρτήσεις απεικόνισης, όπως αυτές που περιγράψαμε παραπάνω. Για το λόγο αυτό οι προγραμματιστές δε χρειάζεται να ανησυχούν για το πώς τα στοιχεία ενός πίνακα αποθηκεύονται στην μνήμη.

Υπάρχουν όμως ορισμένες κατηγορίες πινάκων για τους οποίους η προσπέλαση των στοιχείων τους μέσω μιας Σ.Α.Π., όπως περιγράψαμε παραπάνω, δεν είναι συμφέρουσα από πλευράς χώρου. Τέτοιοι πίνακες έχουν συνήθως μεγάλο πλήθος στοιχείων μηδέν και τυχαία η κάποια χαρακτηριστική διάταξη των μη μηδενικών τους στοιχείων (για παράδειγμα οι τριγωνικοί (άνω,κάτω), οι τριδιαγώνιοι πίνακες κ.λπ.). Οι πίνακες αυτοί είναι δυνατόν να παρασταθούν αν αποθηκευτούν μόνο τα μη μηδενικά τους στοιχεία στη μνήμη κάνοντας έτσι μεγάλη οικονομία χώρου, όπως θα δούμε παρακάτω.

2.2.1 Κάτω τριγωνικός πίνακας (A[i,j] = 0 για κάθε j>i)

Ενας τέτοιος πίνακας έχει n(n+1)/2 στοιχεία και μπορεί ν'αποθηκευτεί σ'ένα μονοδιάστατο πίνακα B[1..n(n+1)/2], κατά γραμμές, ως εξής :



Με τον τρόπο αυτό εξοικονομούμε (n(n-1)/2) θέσεις της μνήμης. Η συνάρτηση απεικόνισης των στοιχείων του πίνακα B είναι γνωστή. Κάθε στοιχείο A[i,j] απεικονίζεται στη θέση:

B[i(i-1) div 2 + j].

2.2.2 Συμμετρικοί πίνακες (A[i,j] = A[j,i] για i,j=1,...,n)

Ενας συμμετρικός πίνακας μπορεί ν'αποθηκευτεί σε μια από τις τριγωνικές του μορφές. Για παράδειγμα, αν ο πίνακας αποθηκευτεί ως κάτω τριγωνικός πίνακας, τότε η προσπέλαση σ'ενα στοιχείο A[i,j] με i<j, που βρίσκεται στο πάνω τριγωνικό μέρος του πίνακα, ισοδυναμεί με την προσπέλαση στο συμμετρικό του στοιχείο κάτω από τη διαγώνιο. Στη περίπτωση αυτή, η συνάρτηση απεικόνισης είναι η ίδια με την περίπτωση του κάτω τριγωνικού πίνακα που εξετάσαμε για τα στοιχεία με i≥j, ενώ για τα στοιχεία με i<j, ο ρόλος των i και j αλλάζει αμοιβαία.

2.2.3 Αραιοί πίνακες (sparse matrices)

Ενας πίνακας ονομάζεται **αραιός** όταν ένα μεγάλο ποσοστό τον στοιχείων του είναι μηδέν. Δεν υπάρχει ένα συγκεκριμένο ποσοστό μηδενικών στοιχείων που να καθορίζει αν ένας πίνακας είναι αραιός ή όχι. Συνήθως αν περισσότερα του 80% των στοιχείων ενός πίνακα είναι μηδενικά, τότε ο πίνακας χαρακτηρίζεται ως αραιός. Οι αραιοί πίνακες χρειάζονται ειδική μεταχείριση όσον αφορά την αποθήκευση και τη χρήση τους. Παρακάτω θα εξετάσουμε μερικούς τρόπους αποθήκευσης και προσπέλασης των στοιχείων ενός αραιού πίνακα.

α) Μέθοδος με συντεταγμένες

Κάθε στοιχείο ενός πίνακα χαρακτηρίζεται από την τιμή του και τους δείκτες του (συντεταγμένες). Για παράδειγμα σ'ενα δισδιάστατο πίνακα, κάθε στοιχείο χαρακτηρίζεται από την τριάδα (i, j, τιμή στοιχείου στη θέση (i, j)). Ετσι, στη περίπτωση ενός αραιού πίνακα αρκεί ν'αποθηκεύσουμε μόνο τα μη μηδενικά του στοιχεία και τους δείκτες τους. Για παράδειγμα, ο παρακάτω πίνακας ακεραίων B έχει μέγεθος 7x8 και μόνο δέκα μη μηδενικά στοιχεία. Αν N είναι το πλήθος των μη μηδενικών στοιχείων ενός αραιού πίνακα, τότε μπορούμε ν'αποθηκεύσουμε τον πίνακα αυτόν σ'έναν άλλον, έστω A[0..N , 1..3], του ιδίου τύπου όπου στη γραμμή 0 αποθηκεύουμε το πλήθος των γραμμών του πίνακα, το πλήθος

των στηλών και το N, ως εξής:

$$B=\begin{bmatrix} 0 & 5 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 & 0 & 5 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A:

	1	2	3
0	7	8	10
1	1	2	5
2	1	5	6
3	2	6	7
4	3	4	9
5	3	7	11
6	5	2	3
7	5	5	1
8	6	3	14
9	6	8	5
10	7	1	6

Παράδειγμα :
Γράψτε μια διαδικασία που να εκχωρεί το γινόμενο ενός πίνακα A μεγέθους 8x8 μ'ένα διάνυσμα x, σ'ένα πίνακα y.

Αλγόριθμος

```
for  γραμμή:= 1 to πλήθος_γραμμών do
 begin
   y[γραμμή]:= 0.0;
```

```
   if πλήθος_στηλών_γραμμής > 0 then
      for στήλη:= 1 to πλήθος_στηλών_γραμμής do
      y[γραμμή]:= y[γραμμή] + A[γραμμή,3]*x[A[γραμμή,2]]
 end

procedure  AX(a:pinakas; x:dianisma; var y:dianisma; m:grammes);
{ m = a[0, 1] }
var   i,j,k: integer; {i δηλώνει γραμμή και j στήλη}
          t: array[1..m] of integer;
begin
  N:= a[0,3]; {πλήθος μη μηδενικών στοιχείων}
  {υπολόγισε το πλήθος των στηλών κάθε γραμμής}
   for i:= 1 to m do
   t[i]:= 0;
   for i:= 1 to N do
   t[a[i,1]]:= t[a[i,1]] + 1;
  k:= 1;
   for i:= 1 to m do
   begin
     y[i]:= 0.0;
     if t[i] <> 0 then
     for j:= 1 to t[i] do
     begin
       y[i]:= y[i] + a[k,3]*x[a[k,2]];
       k:= k + 1
     end
   end
end {AX}
```

β) Δυαδική μέθοδος

Στη μέθοδο αυτή η θέση ενός μη μηδενικού στοιχείου του πίνακα καθορίζεται από την τιμή '1' (true), ενώ η τιμή ενός μηδενικού στοιχείου από την τιμή '0' (false). Για παράδειγμα η δυαδική παράσταση του πίνακα B παραπάνω αντιπροσωπεύεται από τον πίνακα C μαζί με το διάνυσμα D που περιέχει τις τιμές των μη μηδενικών στοιχείων.

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

D	5	6	7	9	11	3	1	14	5	6

Με την μέθοδο αυτή εξοικονομούμε περισσότερο χώρο όταν η δυαδική παράσταση μιας γραμμής του πίνακα "χωράει" αποτελεσματικά στη μνήμη του υπολογιστή μας. Για έναν πίνακα Β, διαστάσεων mxn με Ν μη μηδενικά στοιχεία, ο χώρος της μνήμης που χρειαζόμαστε είναι ίσος με:

$$N + \frac{m * n}{\beta} \quad \text{λέξεις (words)}$$

όπου β είναι το πλήθος των δυαδικών ψηφίων ανά λέξη.

Η προσπέλαση ενός στοιχείου του πίνακα αυτού γίνεται ως εξής. Αν υπάρχει μονάδα στην αντίστοιχη θέση του πίνακα C, τότε μετράμε τις προηγούμενες μονάδες. Το πλήθος των μονάδων αυτών επί το μήκος που καταλαμβάνει το κάθε στοιχείο στη μνήμη μας δίνει την θέση του ζητούμενου στοιχείου στη μνήμη. Η εισαγωγή και διαγραφή στοιχείων δε γίνεται εύκολα με τη δυαδική μέθοδο. Γενικά με την μέθοδο αυτή ότι κερδίζουμε σε χώρο, το χάνουμε στο χρόνο που απαιτείται για την προσπέλαση των στοιχείων του πίνακα.

2.3 Πίνακες χαρακτήρων

Εστω ότι έχουμε τον πίνακα:

```
C: array[1..N] of char
```

Κάθε χαρακτήρας του πίνακα αυτού αποθηκεύεται σε μια θέση μνήμης και όλος ο πίνακας σε συνεχόμενες θέσεις μνήμης. Αν η κάθε λέξη του υπολογιστή μας έχει μήκος 32 δυαδικών ψηφίων (bit) και χρησιμοποιείται ο κώδικας ASCII για την παράσταση των χαρακτήρων (8 bits ανά χαρακτήρα), τότε ένα μεγάλο μέρος της μνήμης παραμένει ανεκμετάλλευτο. Για το παράδειγμά μας έχουμε 24 bits ανεκμετάλλευτα στη κάθε θέση μνήμης. Τα κενά αυτά λέγονται **παραγεμίσματα (pads)**.

Ο λόγος:

$$u = \frac{\text{χρησιμοποιούμενος χώρος της λέξης}}{\text{μήκος της λέξης}}$$

λέγεται **παράγοντας χρησιμοποίησης ή παράγοντας χρήσης**

(utilization factor).
Για το παράδειγμά μας u=25%. Γενικά στους πίνακες χαρακτήρων μπορεί να έχουμε μεγάλη σπατάλη χώρου, αλλά η προσπέλαση των στοιχείων τους είναι πολύ γρήγορη.

2.4 Πυκνωμένοι πίνακες (Packed arrays)

Στην περίπτωση των πυκνωμένων πινάκων χαρακτήρων τοποθετούνται όσο το δυνατόν περισσότεροι διαδοχικοί χαρακτήρες σε μια λέξη της μνήμης. Για παράδειγμα, έστω ότι έχουμε τη δήλωση:

```
var C: packed array [1..10] of char
```

Αν lw(λέξη) = 32 bits και χρησιμοποιούμε τον (8-bit) κώδικα ASCII, τότε σε κάθε λέξη της μνήμης αποθηκεύονται τέσσερις χαρακτήρες. Ο παράγοντας χρησιμοποίησης είναι 100%. Αν χρησιμοποιούσαμε έναν άλλο κώδικα, για παράδειγμα τον κώδικα BCD (6 bits ανά χαρακτήρα), τότε, η κάθε λέξη, θα μπορούσε να χωρέσει 5 χαρακτήρες και ο παράγοντας χρησιμοποίησης θα ήταν 94% μόνο.

Παράδειγμα

Δίνεται ο πίνακας:

```
C: packed array[1..N] of char
```

Αν υποθέσουμε ότι lw = n bits και πως κάθε χαρακτήρας παρίσταται από m bits, τότε:

α) να υπολογιστεί ο χώρος που καταλαμβάνει ο πίνακας σε λέξεις,
β) να υπολογιστεί ο παράγοντας χρησιμοποίησης για ολόκληρο τον πίνακα και
γ) να γραφεί η συνάρτηση απεικόνησης του πίνακα.

Κάθε λέξη του υπολογιστή μπορεί να χωρέσει k = n div m χαρακτήρες. Συνεπώς ολόκληρος ο πίνακας, (N χαρακτήρες), καταλαμβάνει

$$\lceil N \text{ div } k \rceil \quad \text{λέξεις}$$

Ο παράγοντας χρησιμοποίησης για τον πίνακα C είναι:

$$u = \frac{100(n - (n \bmod m))}{n} \%$$

Η συνάρτηση απεικόνισης υπολογίζεται σε δύο στάδια.

(i) Εντοπισμός της διεύθυνσης της λέξης της μνήμης που περιέχει το ζητούμενο στοιχείο, έστω C[j].

$$F(C,<j>) = F(C,<0>) + (j - 1) \text{ div } k$$

(ii) Εντοπισμός του στοιχείου μέσα στη λέξη. Ο υπολογισμός της θέσης του στοιχείου μέσα στη λέξη αφήνεται ως άσκηση στον αναγνώστη*.

2.5 Εγγραφές (Records)

Η εγγραφή είναι μια δομή, τα στοιχεία της οποίας μπορεί να είναι διαφορετικού τύπου. Μ'άλλα λόγια, τα διαφορετικού τύπου δεδομένα μιας εγγραφής θεωρούνται ως μια λογική ενότητα (οντότητα). Για το λόγο αυτό, πολλές φορές, μια εγγραφή αναφέρεται και ως ετερογενής πίνακας. Για παράδειγμα μια εγγραφή μπορεί να περιέχει τ'όνομα ενός φοιτητή, τον αριθμό μητρώου του, την ημερομηνία εγγραφής του κ.λπ. Μια εγγραφή μπορεί να τύχει επεξεργασίας ως μια οντότητα ή μπορεί να επεξεργαστούμε το κάθε πεδίο της, χωριστά. Αλλη διαφορά μεταξύ των εγγραφών και των πινάκων είναι ότι η αναφορά στα πεδία (στοιχεία) μιας εγγραφής γίνεται με το όνομά τους και όχι με μεταβλητές ή παραστάσεις. Οι εγγραφές αποτελούν την κυριότερη δομή της γλώσσας COBOL.

Η διαδικασία υπολογισμού της διεύθυνσης ενός στοιχείου μιας εγγραφής είναι διαφορετική από αυτή των πινάκων, γιατί τα στοιχεία της εγγραφής είναι διαφορετικού τύπου και συνεπώς το καθένα απαιτεί διαφορετικό χώρο αποθήκευσης στη μνήμη. Οι παράμετροι που χρειάζονται για την προσπέλαση ενός στοιχείου μιας εγγραφής είναι:

- η διεύθυνση της βάσης, δηλαδή η διεύθυνση της πρώτης ψηφιοσυλλαβής της εγγραφής,
- η λίστα των πεδίων και
- το μήκος των πεδίων της εγγραφής

Οι παράμετροι αυτές, εκτός από τη διεύθυνση βάσης, καθορίζονται κατά τη μεταγλώττιση του προγράμματος. Οταν κάνουμε αναφορά σ'ένα πεδίο μιας εγγραφής, τότε θα πρέπει να καθοριστεί ο τύπος του στοιχείου, το μήκος του και η **απόσταση του (offset)** από την διεύθυνση βάσης της εγγραφής .

Η συνάρτηση απεικόνισης μιας εγγραφής δίνει στο σύστημα τη δυνατότητα να καθορίσει τη διεύθυνση βάσης ενός πεδίου. Η

Για την απομόνωση του στοιχείου από τη λέξη βλ., Καβουρας, Ι.Κ. Συστήματα υπολογιστών, Τόμος Ι, Αθήνα 1989, Κεφ.2.

διεύθυνση αυτή προκύπτει αν στη διεύθυνση βάσης της εγγραφής προσθέσουμε την απόσταση (offset) του πεδίου από την αρχή της εγγραφής. Τότε το πεδίο μπορεί να τύχει επεξεργασίας ανάλογα με τον τύπο του.

Ανακεφαλαίωση

Ορισμός :
Πίνακας είναι μια διατεταγμένη ακολουθία στοιχείων που έχουν όλα τον ίδιο τύπο.
Τα στοιχεία αποθηκεύονται σε συνεχόμενες θέσεις μνήμης.
Τα στοιχεία ενός πίνακα έχουν μια, ένα προς ένα, αντιστοιχία με τις θέσεις μνήμης που είναι αποθηκευμένα.

Επιτρεπτές πράξεις :
- Άμεση ή τυχαία προσπέλαση σ'ένα στοιχείο,
- Εισαγωγή στοιχείου,
- Διαγραφή στοιχείου,
- Ταξινόμηση, αναζήτηση στοιχείου.

Ασκήσεις

1. Περιγράψτε την προσπέλαση των στοιχείων ενός κάτω τριγωνικού πίνακα, με χρήση ενός Iliffe διανύσματος.

2. Δώστε τη συνάρτηση απεικόνησης για τον πίνακα A[-1..M,2..N], όταν αυτός αποθηκεύεται (α) κατά στήλες και (β) κατά γραμμές.

3. Δώστε τη συνάρτηση απεικόνησης (α) για έναν τριδιαγώνιο και (β) για έναν πενταδιαγώνιο πίνακα, όταν αυτοί αποθηκεύονται κατά γραμμές.

4. Δώστε τη συνάρτηση απεικόνησης για έναν πάνω τριγωνικό πίνακα όταν αυτός αποθηκεύεται (α) κατά γραμμές και (β) κατά στήλες. Κάνετε το ίδιο για έναν κάτω τριγωνικό πίνακα.

5. Στην Pascal ενας πίνακας αποθηκεύεται κατά γραμμές, ενώ στη FORTRAN ο ίδιος πίνακας αποθηκεύεται κατά στήλες. Εστω ότι ένα πρόγραμμα Pascal δημιουργεί έναν πίνακα και θέλει να τον περάσει ως παράμετρο σε μια διαδικασία γραμμένη σε FORTRAN. Για οικονομία χώρου θέλουμε να χρησιμοποιήσουμε και στα δύο προγράμματα το ίδιο φυσικό "αντίγραφο" του πίνακα. Μια αναφορά σ'ένα στοιχείο A[i,j] στη Pascal αναφέρεται οπωσδήποτε σε διαφορετική φυσική διεύθυνση από την ίδια αναφορά στο πρόγραμμα της FORTRAN. Πως θα κάνουμε τα δύο αυτά

προγράμματα να μπορούν να επικοινωνούν μεταξύ τους; Γράψτε κατάλληλα υποπρογράμματα γι'αυτό.

6. Δίνεται ο πίνακας:

$$A = \begin{bmatrix} a_{11} & 0 & 0 & 0 & a_{15} & 0 & 0 \\ 0 & a_{22} & 0 & a_{24} & 0 & 0 & 0 \\ a_{31} & 0 & 0 & 0 & a_{35} & 0 & a_{37} \\ 0 & 0 & a_{43} & 0 & 0 & 0 & 0 \\ 0 & a_{52} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{65} & 0 & 0 \\ 0 & 0 & a_{73} & 0 & 0 & 0 & a_{77} \end{bmatrix}$$

Ο πίνακας αυτός αποθηκεύεται σ'ένα διάνυσμα:

```
u: array[1..N] of real
```

όπου N είναι το πλήθος των μη μηδενικών στοιχείων συν το πλήθος των γραμμών του πίνακα. Για το παράδειγμα μας N=12+7=19 και το διάνυσμα u περιέχει:

u	a_{11}	a_{15}	0	a_{22}	a_{24}	0	a_{31}	a_{35}	a_{37}	0	a_{43}	. . .

Η κάθε τιμή μηδέν παριστάνει το τέλος μιας γραμμής. Ενα διάνυσμα ακεραίων:

```
j : array[1..N] of integer
```

του ιδίου μεγέθους (N) χρησιμοποιείται για την αποθήκευση του δείκτη που δηλώνει τη στήλη του αντίστοιχου στοιχείου του πίνακα A.
Γράψτε μια διαδικασία στη γλώσσα Pascal που να κάνει τον πολλαπλασιασμό του πίνακα A μ'ένα δεδομένο διάνυσμα:

```
X: array[1..n] of real
```

όπου n είναι ο βαθμός του πίνακα (n=7).

7. Γράψε μια διαδικασία στη γλώσσα Pascal, που να υπολογίζει τον ανάστροφο ενός αραιού πίνακα, όταν ο πίνακας αποθηκεύεται με τη μέθοδο των συντεταγμένων.

8. Δώστε τον ορισμό ενός αραιού πίνακα, στη γλώσσα Pascal, του οποίου τα στοιχεία είναι πραγματικοί αριθμοί.

ΚΕΦΑΛΑΙΟ 3

ΓΡΑΜΜΙΚΕΣ ΛΙΣΤΕΣ

3.1 Γενικά

Μια **γραμμική λίστα (linear list)** είναι μια διατεταγμένη ακολουθία από στοιχεία καθένα των οποίων, εκτός από το πρώτο στοιχείο, έχει ένα προηγούμενο και εκτός του τελευταίου στοιχείου ένα επόμενο.

Ο ορισμός αυτός αποδίδει τη λογική προτεραιότητα που υπάρχει μεταξύ των στοιχείων μιας λίστας. Η φυσική όμως παράσταση (αποθήκευση) μιας λίστας μπορεί να μην ακολουθεί τον ορισμό αυτόν, με την έννοια ότι διαδοχικά στοιχεία μιας λίστας μπορεί να βρίσκονται σε οποιαδήποτε θέση μέσα στη μνήμη και κάποιος άλλος μηχανισμός να καθορίζει το επόμενο στοιχείο της λίστας κάθε φορά. Οι πίνακες που εξετάσαμε στο κεφάλαιο 2 είναι χαρακτηριστική περίπτωση μιας γραμμικής λίστας στην οποία υπάρχει ταύτηση της λογικής και φυσικής τους παράστασης.

Κάθε στοιχείο μιας λίστας μπορεί να είναι ένα απλό αντικείμενο ή μια εγγραφή ή μια ξεχωριστή λίστα. Τα στοιχεία της λίστας είναι συνήθως ταξινομημένα είτε χρονολογικά ή λεξικογραφικά ή μ'άλλα κριτήρια. Ανάλογα με την φυσική τους παράσταση οι γραμμικές λίστες κατατάσσονται σε δύο κατηγορίες, τις στατικές και τις δυναμικές λίστες.

α) Στατικές λίστες

Η υλοποίηση μιας στατικής λίστας γίνεται με χρήση πινάκων. Επειδή τα όρια ενός πίνακα παραμένουν σταθερά από τη δήλωσή του και μετά, προκύπτει και τ'όνομά τους στατικές λίστες.

β) Δυναμικές λίστες

Η υλοποίηση των δυναμικών λιστών γίνεται με δυναμική

καταχώρηση μνήμης, όταν αυτή χρειάζεται, με τη χρήση δεικτών. Το βασικό χαρακτηριστικό μιας δυναμικής λίστας είναι ότι κάθε στοιχείο της περιέχει όχι μόνο τα δεδομένα, αλλά και μια ή περισσότερες θέσεις που περιέχουν τις κατάλληλες πληροφορίες για τα στοιχεία που συσχετίζονται λογικά μεταξύ τους. Οι θέσεις αυτές περιέχουν δείκτες (pointers) που δίνουν τη θέση του επόμενου στοιχείου στη λίστα.

Για κάθε μια από τις κατηγορίες στατικών και δυναμικών λιστών θα εξετάσουμε ειδικές δομές και θα περιγράψουμε είτε υπό μορφή αλγορίθμων, είτε με διαδικασίες γραμμένες σε Pascal ορισμένες χαρακτηριστικές εφαρμογές για την κάθε περίπτωση.

3.2 Στατικές Γραμμικές Λίστες

3.2.1 Στοίβα (Stack)

Στοίβα είναι ένας τύπος δεδομένων που περιγράφεται από την ιδιότητα: **"Το τελευταίο στοιχείο που εισέρχεται στη στοίβα εξέρχεται πρώτο" (Last-In-First-Out, LIFO).** Δηλαδή το στοιχείο που μπορεί να τύχει επεξεργασίας κάθε φορά είναι το τελευταίο που προστέθηκε στη στοίβα. Μερικές από τις χαρακτηριστικές εφαρμογές που χρησιμοποιούν την δομή της στοίβας είναι: Η υλοποίηση υποπρογραμμάτων, σε γλώσσες προγραμματισμού δομημένες με ενότητες (blocks), όπως τις ALGOL, Pascal, και Ada, η διαχείρηση της μνήμης κ.λπ.

Μια στατική στοίβα ορίζεται πλήρως μ'ένα μονοδιάστατο πίνακα και μια μεταβλητή που δείχνει πάντοτε το στοιχείο στην κορυφή της στοίβας, δηλαδή το στοιχείο που εισάχθηκε (προστέθηκε) τελευταία στη στοίβα. Έτσι, λοιπόν, αν τα όρια του πίνακα είναι κάτω_όριο..πάνω_όριο, τότε τα όρια του δείκτη της κορυφής θα είναι κάτω_όριο - 1..πάνω_όριο. Αν η τιμή του δείκτη κάτω_όριο_κορυφής είναι ίση με κάτω_όριο - 1 τότε η στοίβα είναι άδεια. Συνεπώς ο ορισμός του τύπου και η δήλωση μιάς στοίβας είναι:

```
const
          κάτω_όριο = 1;  {έστω}
          πάνω_όριο = 10; {έστω}
    κάτω_όριο_κορυφής = 0; {κάτω_όριο - 1}

type
     όρια_δείκτη_κορυφής = κάτω_όριο_κορυφής .. πάνω_όριο;
```

```
     όρια_στοίβας = κάτω_όριο .. πάνω_όριο;
     τύπος_στοιχείου = ...{συμπληρώνεται από το χρήστη};
     τύπος_στοίβας = record
                       s: array[όρια_στοίβας] of τύπος_στοιχείου;
                       κορυφή: όρια_δείκτη_κορυφής
                     end;
var    στοίβα: τύπος_στοίβας
```

Οι βασικές πράξεις μιας στοίβας είναι:

1. Δημιουργία στοίβας (stack creation)

Η δημιουργία μιας κενής στοίβας πραγματοποιείται με την εκχώριση στο δείκτη κορυφής της τιμής κάτω_όριο_κορυφής.

```
procedure  δημιούργησε_στοίβα (var  στοίβα: τύπος_στοίβας);
begin
  στοίβα.κορυφή:= κάτω_όριο_κορυφής
end
```

2. Ελεγχος αν μια στοίβα είναι κενή (empty stack check)

```
function  κενή (στοίβα: τύπος_στοίβας): boolean;
begin
  κενή:= στοίβα.κορυφή = κάτω_όριο_κορυφής
end
```

3. Ελεγχος αν μια στοίβα είναι γεμάτη (full stack check)

```
function  γεμάτη (στοίβα: τύπος_στοίβας): boolean;
begin
  γεμάτη:= στοίβα.κορυφή = πάνω_όριο
end
```

4. Εισαγωγή στοιχείου σε μια στοίβα (push)

Κατά την εισαγωγή πρέπει πάντα να ελέγχουμε αν η στοίβα είναι γεμάτη. Αν δεν είναι γεμάτη, τότε εισαγάγουμε το νέο στοιχείο στη κορυφή της στοίβας. Ο έλεγχος αυτός (if not γεμάτη (στοίβα)) γίνεται πάντοτε πριν την κλήση της διαδικασίας "εισάγαγε".

```
procedure  εισάγαγε (x: τύπος_στοιχείου;
                     var στοίβα: τύπος_στοίβας);
begin
  with   στοίβα  do
  begin
    κορυφή:= κορυφή + 1;
    s [κορυφή]:= x
  end
end {εισάγαγε}
```

5. Εξαγωγή στοιχείου από μια στοίβα (pop).

Κατά την εξαγωγή πρέπει πάντα να ελέγχουμε αν η στοίβα είναι κενή. Αν δεν είναι κενή, τότε εξάγουμε το στοιχείο από την κορυφή της στοίβας. Ο έλεγχος (if not κενή (στοίβα)) γίνεται πάντοτε πριν την κλήση της διαδικασίας "εξάγαγε".

```
procedure  εξάγαγε (var x: τύπος_στοιχείου;
                    var στοίβα: τύπος_στοίβας);
begin
  with  στοίβα  do
  begin
    x:= s [κορυφή];
    κορυφή:= κορυφή - 1
  end
end {εξάγαγε}
```

3.2.2 Εφαρμογές

Υπολογισμός αριθμητικών παραστάσεων

Αν θεωρήσουμε ότι το σύνολο όλων των επιτρεπτών αριθμητικών παραστάσεων με παρενθέσεις μπορεί να περιέχει ορίσματα (τελεστέους) που είναι μη αρνητικοί αριθμοί και τους αριθμητικούς τελεστές +,-,*,/, τότε ο ορισμός μιας **ενδοθεματικής (infix) παράστασης** δίνεται αναδρομικά ως εξής:

- Μια μεταβλητή που συμβολίζεται μ' ένα γράμμα (A,...,Z) ή ένας μη αρνητικός ακέραιος είναι μια ενδοθεματική παράσταση.

- Αν A και B είναι ενδοθεματικές παραστάσεις, τότε και (A+B), (A-B), (A*B) και (A/B) είναι ενδοθεματικές παραστάσεις.

Ο ορισμός αυτός προϋποθέτει πάντοτε τη χρήση παρενθέσεων. Αυτό δε γίνεται συνήθως στην πράξη, δηλαδή μια παράσταση, έστω η

(A+B)*C, θεωρείται ως ορθή ενδοθεματική παράσταση, αντί της ((A+B)*C) που απαιτεί ο ορισμός. Έτσι αποφεύγονται πολλές παρενθέσεις σε μια παράσταση.

Ο υπολογισμός μιας τέτοιας ενδοθεματικής παράστασης γίνεται ανάλογα με τις προτεραιότητες των τελεστών των πράξεων. Έτσι, για να υπολογίσουμε μια ενδοθεματική παράσταση, πρέπει να τη διατρέξουμε (διασχίσουμε, περάσουμε) πολλές φορές ώστε να εντοπίσουμε τους τελεστές με υψηλότερη προτεραιότητα κάθε φορά. Για ν'αποφευχθεί αυτό, μετατρέπουμε την ενδοθεματική παράσταση σε μια άλλη μορφή για παράδειγμα την επιθεματική (postfix) ή την προθεματική (prefix), που όπως θα δούμε παρακάτω υπολογίζονται ευκολότερα.

Επιθεματική μορφή (Postfix notation) ή Πολωνικός αντίστροφος συμβολισμός (reverse Polish notation)

Η επιθεματική μορφή μιας παράστασης, που οφείλεται στον Πολωνό Μαθηματικό J. Lukasiewicz, είναι μια παράσταση στην οποία οι τελεστές εμφανίζονται μετά τους τελεστέους. Μια τέτοια μορφή έχει το πλεονέκτημα ότι υπολογίζεται εύκολα μ'ένα μόνο "πέρασμά" (ή μία μόνο διάσχιση) της.

Παρακάτω θα δώσουμε πρώτα τον αλγόριθμο μετατροπής μιας ενδοθεματικής παράστασης σε επιθεματική μορφή και μετά τον αλγόριθμο υπολογισμού της τιμής μιας επιθεματικής παράστασης. Για το σκοπό αυτό θα χρησιμοποιήσουμε μια στοίβα για την αποθήκευση των τελεστών και έναν πίνακα από χαρακτήρες για την αποθήκευση της επιθεματικής παράστασης. Η ενδοθεματική παράσταση διαβάζεται από το αρχείο input. Ο αλγόριθμος της μετατροπής μιας παράστασης σε επιθεματική μορφή δίνεται στον πίνακα 3.1. Στον πίνακα 3.2 περιγράφονται τα βήματα του αλγορίθμου για την μετατροπή της παράστασης:

(A+B*(Z-D*E*H+G)*K)/(E*B)

σε επιθεματική μορφή.

```
while not eof (input)  do
begin
  read (input, χαρακτήρας);
  if (ο χαρακτήρας είναι όρισμα) then
        επιπρόσθεσε τον χαρακτήρα στην επιθεματική παράσταση
   else
   if (χαρακτήρας = '(') then
       εισάγαγε (χαρακτήρα, στοίβα_τελεστών)
   else
```

```
if (χαρακτήρας = ')') then
   begin
     while (το στοιχείο στη κορυφή της στοίβας_τελεστών
                                                <> '(') do
       begin
         εξάγαγε (τελεστή, στοίβα_τελεστών);
         επιπρόσθεσε τον τελεστή στην επιθεματική παράσταση
       end;
     εξάγαγε (τελεστή, στοίβα_τελεστών)
   end
else {ο χαρακτήρας είναι τελεστής}
begin
  if not κενή (στοίβα_τελεστών) then
   while (προτεραιότητα(χαρακτήρα) ≤ προτεραιότητα(στοιχείου
   στη κορυφή της στοίβας_τελεστών))
      begin
        εξάγαγε(τελεστή, στοίβα_τελεστών);
        επιπρόσθεσε τον τελεστή στην επιθεματική παράσταση
      end;
    εισάγαγε (χαρακτήρα, στοίβα_τελεστών)
end
end;
άδειασε τη στοίβα_τελεστών στην επιθεματική παράσταση
```

Πίνακας 3.1

Αλγόριθμος υπολογισμού της επιθεματικής παράστασης (πίνακας 3.3)

Η παράσταση υποτίθεται πως είναι αποθηκευμένη στη συμβολοσειρά "παράσταση". Μια στοίβα των μεταβλητών χρησιμοποιείται καθολικά (globally), ως βοηθητική δομή.

```
i:= 1;  χαρακτήρας:= παράσταση [1];
while  (χαρακτήρας <> κενού_χαρακτήρα) do
  begin
    if  (χαρακτήρας είναι αριθμός)  then
       εισάγαγε (αριθμός, στοίβα_μεταβλητών)
    else  {χαρακτήρας είναι τελεστής}
      begin
        εξάγαγε (αριθμός2, στοίβα_μεταβλητών);
        εξάγαγε (αριθμός1, στοίβα_μεταβλητών);
```

Είσοδος δεδομένων	στοίβα τελεστών	επιθεματική μορφή
(	(	
A	(	A
+	(+	A
B	(+	AB
*	(+*	AB
(	(+*(	AB
Z	(+*(	ABZ
-	(+*(-	ABZ
D	(+*(-	ABZD
*	(+*(-*	ABZD
E	(+*(-*	ABZDE
*	(+*(-*	ABZDE*
H	(+*(-*	ABZDE*H
+	(+*(+	ABZDE*H*-
G	(+*(+	ABZDE*H*-G
)	(+*	ABZDE*H*-G+
*	(+ *	ABZDE*H*-G+*
K	(+*	ABZDE*H*-G+*K
)		ABZDE*H*-G+*K*+
/	/	ABZDE*H*-G+*K*+
(	/(	ABZDE*H*-G+*K*+
E	/(	ABZDE*H*-G+*K*+E
*	/(*	ABZDE*H*-G+*K*+E
B	/(*	ABZDE*H*-G+*K*+EB
)	/	ABZDE*H*-G+*K*+EB*
		ABZDE*H*-G+*K*+EB*/

Πίνακας 3.2

```
        {υπολόγισε αποτέλεσμα}
        αποτέλεσμα:= αριθμός1 <τελεστής> αριθμός2;
        εισάγαγε (αποτέλεσμα, στοίβα_μεταβλητών)
      end;
    if  (i < μήκος_επιθεματικής_παράστασης)  then
      begin
        i:= i + 1;
        χαρακτήρας:= παράσταση [i]
      end
    else
       χαρακτήρας:= κενός_χαρακτήρας
  end;
  εξάγαγε (αποτέλεσμα, στοίβα_μεταβλητών)
```

Πίνακας 3.3

Η παρακάτω συνάρτηση υπολογισμός_επιθεματικής_παράστασης αποδίδει τον αλγόριθμο σε Pascal.

```
function   υπολογισμός_επιθεματικής_παράστασης
                          (παράσταση: τύπος_παράστασης): real;
const
        κενό = ' ';
  var
        αριθμός1, αριθμός2, αποτέλεσμα: real;
        i: integer; χαρακτήρας: char;

function  όρισμα (χαρακτήρας: char): boolean;
begin
  όρισμα:= χαρακτήρας in ['0'..'9']
end {όρισμα};

function πράξη (αριθμός1, αριθμός2: real; χαρακτήρας: char):real;
begin
  if  χαρακτήρας in ['+', '-', '*', '/'] then
      case χαρακτήρας of
      '+': πράξη:= αριθμός1 + αριθμός2;
      '-': πράξη:= αριθμός1 - αριθμός2;
      '*': πράξη:= αριθμός1 * αριθμός2;
      '/': if αριθμός2 <> 0.0 then   πράξη:= αριθμός1 / αριθμός2
      end
  else
  writeln ('λάθος τελεστής')
end {πράξη};
```

```
begin (υπολογισμός_επιθεματικής_παράστασης)
   μέγεθος_παράστασης:= length (παράσταση);
   δημιουργία_στοίβας (στοίβα_μεταβλητών);
   for i:= 1 to μέγεθος_παράστασης  do
       begin
         χαρακτήρας:= παράσταση [i];
         if όρισμα(χαρακτήρας) then
          begin
            αποτέλεσμα:= ord(χαρακτήρα) - ord('0');
            εισάγαγε (αποτέλεσμα, στοίβα_μεταβλητών)
          end
         else
            begin
              εξάγαγε (αριθμός2, στοίβα_μεταβλητών);
              εξάγαγε (αριθμός1, στοίβα_μεταβλητών);
              αποτέλεσμα:= πράξη(αριθμός1, αριθμός2, χαρακτήρας);
              εισάγαγε (αποτέλεσμα, στοίβα_μεταβλητών)
            end
      end;
  εξάγαγε (αποτέλεσμα, στοίβα_μεταβλητών);
  υπολογισμός_επιθεματικής_παράστασης:= αποτέλεσμα
end {υπολογισμός_επιθεματικής_παράστασης}
```

3.2.3 Αναδρομή (Recursion)

Αναδρομή είναι η τεχνική με την οποία μπορούμε να περιγράψουμε κάτι "μέσω του εαυτού του". Η αναδρομή μας επιτρέπει να γράψουμε απλά και ευανάγνωστα προγράμματα που λύνουν δύσκολα προβλήματα. Τα αναδρομικά υποπρογράμματα έχουν όμως και τα εξής μειονεκτήματα:

- χρειάζονται περισσότερο χώρο στη μνήμη και
- μπορεί να είναι πιο αργά από τα αντίστοιχα μη αναδρομικά υποπρογράμματα.

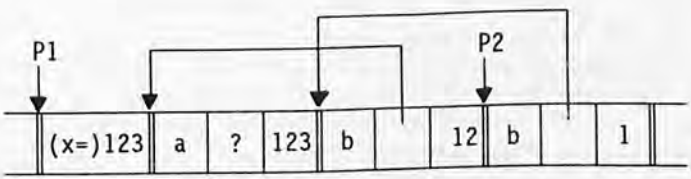
Ενα αναδρομικό υποπρόγραμμα μπορεί να γραφεί χωρίς αναδρομή, χρησιμοποιώντας μια στοίβα, ή σε ορισμένες περιπτώσεις και χωρίς στοίβα όπως θα δούμε στα παρακάτω παραδείγματα. Αν χρησιμοποιήσουμε στοίβα, τότε το μέγεθός της θα είναι ανάλογο του "βάθους" της αναδρομής, δηλαδή του πόσες φορές το υποπρόγραμμα καλεί τον εαυτό του. Στα παρακάτω θα περιγράψουμε τον τρόπο λειτουργίας αναδρομικών υποπρογραμμάτων και θα δώσουμε δύο παραδείγματα αναδρομικών διαδικασιών μαζί με τις αντίστοιχες μη αναδρομικές εκδόσεις τους.

Κατά την εκτέλεση προγραμμάτων της Pascal ο χώρος της μνήμης οργανώνεται ως στοίβα. Κατά την κλήση ενός υποπρογράμματος δημιουργείται μια εγγραφή που ονομάζεται **εγγραφή ενεργοποίησης (activation record)** της διαδικασίας. Η εγγραφή αυτή περιέχει τις τοπικές μεταβλητές του υποπρογράμματος, τις τυπικές παραμέτρους του και δύο άλλες μεταβλητές τύπου δείκτη (pointer). Η πρώτη μας δίνει τη διεύθυνση στην οποία θα πρέπει να επιστρέψει ο έλεγχος του προγράμματος κατά την έξοδο από το υποπρόγραμμα και ονομάζεται **σύνδεσμος διεύθυνσης επιστροφής (return address link)**. Η άλλη μεταβλητή ονομάζεται **σύνδεσμος στοίβας (stack link)** και χρησιμοποιείται κατά την έξοδο από το υποπρόγραμμα για να επαναφέρει τη στοίβα στην ίδια ακριβώς κατάσταση που ήταν πριν την είσοδο στο υποπρόγραμμα.

Παράδειγμα 1: Η παρακάτω διαδικασία "γράψε_ακέραιο" τυπώνει την αριθμοπαράσταση ενός φυσικού ακεραίου.

```
procedure γράψε_ακέραιο (x: φυσικός);
begin
  if x < 10 then
   write (chr(x + ord('0')))
  else
   begin
     γράψε_ακέραιο(x div 10);
{b}  write (chr(x mod 10 + ord('0')))
   end
end {γράψε_ακέραιο}
```

Όταν η διαδικασία αυτή κληθεί με πραγματική παράμετρο τον ακέραιο 123, τότε, μετά την τρίτη κλήση της διαδικασίας η στοίβα της μνήμης θα έχει την μορφή που δείχνει το σχήμα 3.1



Σχήμα 3.1

Στο σχήμα 3.1 η τιμή του x (123) στην πρώτη θέση έχει εκχωρηθεί από το (υπο)πρόγραμμα που καλεί τη διαδικασία

γράψε_ακέραιο. Η εγγραφή ενεργοποίησης περιέχει την τυπική παράμετρο και τους δύο συνδέσμους, που αναφέραμε. Η διεύθυνση a αναφέρεται στην εντολή που ακολουθεί την κλήση της διαδικασίας γράψε_ακέραιο στο καλούν (υπο)πρόγραμμα, ενώ η διεύθυνση b στην εντολή που ακολουθεί την κλήση του εαυτού της, (σημείο {b}) της διαδικασίας. Ο σύνδεσμος της στοίβας στην πρώτη εγγραφή είναι αόριστος, καθόσον η στοίβα δημιουργείται από το καλούν (υπο)πρόγραμμα.

Κατά την είσοδο σ' ένα υποπρόγραμμα πραγματοποιούνται οι εξής πράξεις:

α) εισαγωγή στη στοίβα της διεύθυνσης επιστροφής,
β) εισαγωγή στη στοίβα του δείκτη της στοίβας,
γ) εισαγωγή στη στοίβα των τοπικών μεταβλητών και των τιμών που δίνουν οι πραγματικές παράμετροι και
δ) μεταφορά του ελέγχου στις εντολές του σώματος του υποπρογράμματος.

Κατά την έξοδο από μια διαδικασία πραγματοποιούνται οι αντίστροφες πράξεις δηλαδή:

α) εξαγωγή απο τη στοίβα των τοπικών μεταβλητών,
β) επιστροφή στη θέση που δείχνει ο δείκτης της στοίβας και
γ) επιστροφή του ελέγχου στη διεύθυνση που δείχνει ο σύνδεσμος της διεύθυνσης επιστροφής.

Θα μπορούσαμε να γράψουμε τη διαδικασία γράψε_ακέραιο χωρίς αναδρομή, με χρήση μιάς στοίβας, το μέγεθος της οποίας είναι ανάλογο των ψηφίων που έχει ο ακέραιος που θέλουμε να γράψουμε ως εξής:

```
procedure γράψε_ακέραιο (x: φυσικός);
begin {γράψε_ακέραιο}
  δημιουργησε_στοίβα(στοίβα);
  while x >= 10 do
     begin
       if not γεμάτη(στοίβα) then
       εισάγαγε(x, στοίβα);
       x:= x div 10
     end;
    write (chr(x + ord('0')));
    while not κενή(στοίβα) do
       begin
         εξάγαγε (x, στοίβα);
         write (chr(x mod 10 + ord('0')))
       end
end {γράψε_ακέραιο}
```

Παράδειγμα 2: Η συνάρτηση τύπου boolean, "ανήκει", εξετάζει αν ένα στοιχείο x ανήκει σ'ένα τμήμα A[p]..A[u] ενός πίνακα A.

```
function ανήκει(A: τύπος_πίνακα; p, u: όρια_πίνακα;
                x: τύπος_στοιχείου ): boolean;

begin
   if x = A[p] then ανήκει:= true
   else
   if p = u then ανήκει:= false
   else
      begin
        p:= p + 1;
        ανήκει:= ανήκει (A, p, u, x)
      end
end {ανήκει}
```

Η συνάρτηση αυτή μπορεί να γραφεί εύκολα χωρίς αναδρομή και χωρίς στοίβα (άσκηση προς τον αναγνώστη).

3.2.4 Πολλαπλές στοίβες (multiple stacks)

Οταν έχουμε πολλές στοίβες μπορεί μια από αυτές να γεμίσει ενώ υπάρχει αρκετός χώρος στις άλλες στοίβες. Σπάνια θα υπάρξει περίπτωση να είναι όλες οι στοίβες γεμάτες ταυτόχρονα. Περισσότερες της μιάς στοίβες μπορούν να υλοποιηθούν μ'έναν μόνο πίνακα, τα όρια του οποίου καθορίζουν όλο το διαθέσιμο χώρο. Κάθε στοίβα [i] μπορεί να υλοποιηθεί με δύο δείκτες:

```
το δείκτη_βάσης[i]
και το δείκτη_κορυφής[i]
```

για i=1,...,v όπου v είναι το πλήθος των στοιβών.

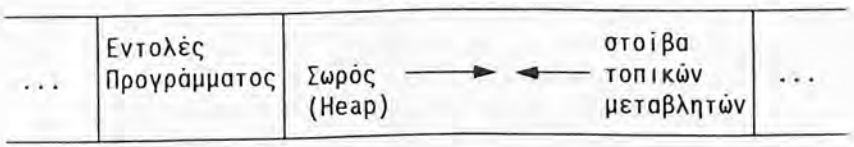
Είναι φανερό πως σε μια τέτοια δομή χρειαζόμαστε έναν αλγόριθμο, που να χειρίζεται τις απαιτήσεις σε μνήμη των στοιβών, έτσι ώστε υπερχείληση να συμβαίνει μόνο όταν **όλος** ο διαθέσιμος χώρος γεμίσει. Για τη διαδικασία εισαγωγής στη στοίβα i, όταν αυτή είναι γεμάτη ο αλγόριθμος πρέπει να ανιχνεύει τις γειτονικές στοίβες που βρίσκονται "δεξιά" και "αριστερά" της στοίβας i, μέχρις ότου βρει διαθέσιμο χώρο. Οταν βρεθεί ο χώρος, τότε αυτός, δίνεται στη στοίβα i με κατάλληλη μετακίνηση των δεικτών των στοιβών, επιτρέποντας έτσι να γίνει η εισαγωγή.

Η απλούστερη περίπτωση είναι αυτή των δύο στοιβών. Στην

περίπτωση αυτή οι βάσεις των στοιβών είναι τα δύο άκρα του πίνακα. Υπερχείλιση συμβαίνει όταν:

κορυφή_στοίβας1 + 1 = κορυφή_στοίβας2.

Η διαθέσιμη μνήμη κατά την εκτέλεση ενός προγράμματος της Pascal οργανώνεται, όπως δείχνει το σχήμα 3.2.



Σχήμα 3.2

Στα δεξιά του σχήματος 3.2 έχουμε τη στοίβα των τοπικών μεταβλητών των υποπρογραμμάτων. Στη στοίβα αυτή εισάγονται οι εγγραφές ενεργοποίησης των υποπρογραμμάτων όταν αυτά καλούνται. Μετά το τέλος της εκτέλεσης των υποπρογραμμάτων οι θέσεις αυτές εξάγονται (ή, μάλλον, "διαγράφονται") από την στοίβα. Στο αριστερό μέρος του σχήματος, που ονομάζεται σωρός (heap), γίνεται η δυναμική καταχώρηση της μνήμης (με την εντολή new). Ο χώρος αυτός δεν αποτελεί στοίβα γιατί μπορούμε να εξαγάγουμε ένα οποιοδήποτε στοιχείο για παράδειγμα με την εντολή dispose, μπορεί όμως να επεκτείνεται ή να πυκνώνει με κάποιο τρόπο ανάλογο μ' αυτόν της στοίβας.

Ο αναγνώστης μπορεί να βρεί περισσότερα στοιχεία για την υλοποίηση πολλαπλών στοιβών στη βιβλιογραφία που δίνεται στο τέλος του βιβλίου.

Ασκηση

Υποθέτουμε ότι μια αριθμητική παράσταση περιέχει (), [], και {}. Να γραφεί μια συνάρτηση που να ελέγχει την ορθότητα της παράστασης ως προς τη στάθμιση (balancing) των () [] και {}. Η παράσταση είναι αποθηκευμένη σε μια συμβολοσειρά.

Θα χρησιμοποιήσουμε μια στοίβα ως βοηθητική δομή.

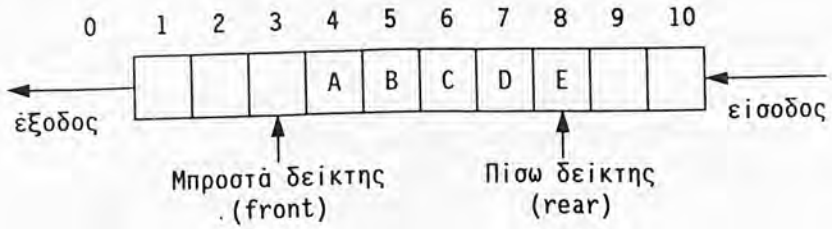
```
function σταθμίζονται (ch1, ch2: char): boolean;
begin
if (ch1 = '(') and (ch2 = ')') or (ch1 = '[') and (ch2 = ']')
 or (ch1 = '{') and (ch2 = '}') then σταθμίζονται:= true
else σταθμίζονται:= false
end; {σταθμίζονται}

function έλεγχος_παράστασης (παράσταση: τύπος_παράστασης):
                                                    boolean;
var i,μέγεθος: integer;
στοιχείο: char;
begin
  έλεγχος_παράστασης:= true;
  δημιούργησε_στοίβα (στοίβα); i:= 1;
  μέγεθος:= length (παράσταση);
  while (i <= μέγεθος) and (not γεμάτη (στοίβα)) do
  begin
    if παράσταση[i] in ['(','[','{'] then
     εισάγαγε (παράσταση [i], στοίβα)
    else
    if παράσταση[i] in [')',']','}'] then
     if κενή (στοίβα) then έλεγχος_παράστασης:= false
     else
      begin
        εξάγαγε (στοιχείο, στοίβα);
        if not σταθμίζονται (στοιχείο, παράσταση [i]) then
                                  έλεγχος_παράστασης:= false
      end;
     i:= i + 1
  end;
   if not κενή (στοίβα) then έλεγχος_παράστασης:= false
end
```

3.2.5 Ουρές (queues)

Ουρά είναι ο τύπος δεδομένων που περιγράφεται από την ιδιότητα: **"Το στοιχείο που εισέρχεται πρώτο στην ουρά εξέρχεται πρώτο" (First-In-First-Out, FIFO)**. Μερικές από τις χαρακτηριστικές εφαρμογές που χρησιμοποιούν τη δομή της ουράς είναι: Ο χρονοπρογραμματισμός (scheduling) των απαιτήσεων εισόδου/εξόδου στους ελεγκτές δίσκων, τα πειράματα προσομοίωσης, που χρησιμοποιούνται για την εκτίμηση της απόδοσης ηλεκτρονικών υπολογιστών και γενικά όλες οι περιπτώσεις όπου πελάτες αναμένουν

για εξυπηρέτηση από κάποια υπηρεσία ή ευκολία (facility).

Η εισοδος των στοιχείων σε μια ουρά γίνεται από το ένα άκρο, ενώ η έξοδος από το άλλο άκρο. Έτσι λοιπόν μια στατική ουρά μπορεί να υλοποιηθεί μ'έναν πίνακα και δύο δείκτες (βλέπε σχήμα 3.3) που δείχνουν αντίστοιχα στο πρώτο (head) και το τελευταίο (tail) στοιχείο της ουράς.



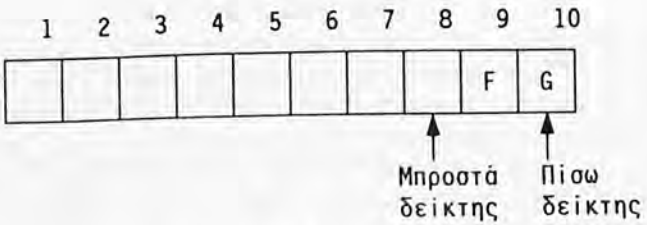
Σχήμα 3.3

Αρχικά και οι δύο δείκτες δείχνουν στη θέση μηδέν. Κατά την εισαγωγή ενός στοιχείου στην ουρά ο πίσω_δείκτης αυξάνεται κατά ένα και το στοιχείο εισάγεται στη νέα θέση, που δείχνει ο πίσω δείκτης. Ανάλογα, κατά την εξαγωγή, αυξάνεται ο μπροστά δείκτης κατά ένα. Συνεπώς ο μπροστά δείκτης της ουράς δείχνει πάντα μια θέση πιό μπροστά από το πρώτο στοιχείο της ουράς, ενώ ο πίσω δείκτης δείχνει πάντα στο τελευταίο στοιχείο της ουράς. Οταν οι δύο δείκτες είναι ίσοι, τότε η ουρά είναι κενή.

Ας θεωρήσουμε τώρα την ουρά του σχήματος 3.3 και τις πράξεις:

(α) εισαγωγή δύο στοιχείων (F, G) και
(β) εξαγωγή πέντε στοιχείων (A, B, C, D, E).

Οταν οι πράξεις αυτές εκτελεστούν η ουρά καταλήγει όπως δείχνει το σχήμα 3.4.

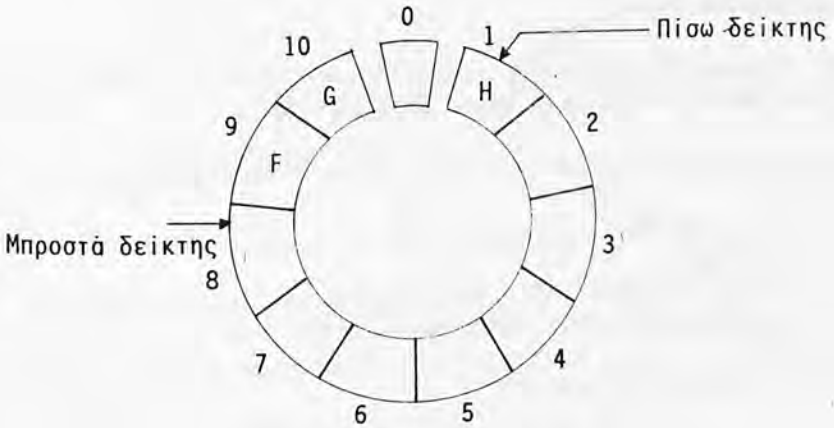


Σχήμα 3.4

Αν προσπαθήσουμε να κάνουμε τώρα εισαγωγή ενός στοιχείου (Η) στην ουρά, παρατηρούμε ότι αυτή δεν είναι δυνατή, γιατί ο πίσω_δείκτης έχει ήδη τη μέγιστη τιμή (10), παρόλο ότι υπάρχουν αρκετές διαθέσιμες θέσεις στην ουρά. Για την αντιμετώπιση τέτοιων καταστάσεων θα πρέπει να έχουμε είτε απεριόριστα μεγάλη μνήμη στη διάθεσή μας, ή θα πρέπει να αναδιοργανώσουμε την ουρά μετά από έναν ορισμένο αριθμό εισαγωγών/εξαγωγών. Το στοιχείο (Η) θα μπορούσε να εισαχθεί στην αρχή της ουράς με την αντίστοιχη μετατόπιση του πίσω δείκτη.

Για την καλύτερη παρακολούθηση της ουράς ας θεωρήσουμε ότι κάμπτουμε τον πίνακα από τα δύο άκρα του έτσι ώστε να σχηματιστεί ένας δακτύλιος όπως δείχνει το σχήμα 3.5. Μια τέτοια ουρά ονομάζεται **κυκλική ουρά (circular queue)**. Σε μια κυκλική ουρά όταν ο πίσω δείκτης δείχνει στο μέγιστο μήκος της ουράς (θέση 10 στο παραπάνω παράδειγμα) και θέλουμε να κάνουμε εισαγωγή, τότε, εφόσον υπάρχουν διαθέσιμες θέσεις, το νέο στοιχείο θα εισαχθεί στη θέση 1.

Μια στατική ουρά ορίζεται πλήρως από έναν μονοδιάστατο πίνακα και δύο μεταβλητές, (δείκτες), που δείχνουν στο τέλος και σε μια θέση πιό μπροστά από την αρχή (κεφαλή) της ουράς



Σχήμα 3.5

αντίστοιχα. Για τη δημιουργία της ουράς εκχωρούμε και στους δύο δείκτες την τιμή 0. Έτσι ο ορισμός του τύπου και η δήλωση μιάς ουράς είναι:

```
const
        κάτω_όριο = 1; {έστω}
        πάνω_όριο = 10; {έστω}
        κάτω_όριο_δείκτη = 0; {κάτω_όριο - 1}

type
        όρια_ουράς = κάτω_όριο.. πάνω_όριο;
        όρια_μπροστά_δείκτη = κάτω_όριο_δείκτη.. πάνω_όριο;
        όρια_πίσω_δείκτη = κάτω_όριο_δείκτη.. πάνω_όριο;
        τύπος_στοιχείου = ... ; {ορίζεται από το χρήστη}

    τύπος_ουράς = record
                    q: array [όρια_ουράς] of τύπος_στοιχείου;
                    μπροστά_δείκτης: όρια_μπροστά_δείκτη;
                    πίσω_δείκτης: όρια_πίσω_δείκτη
                  end;

var
        ουρά: τύπος_ουράς
```

Οι βασικές πράξεις μιας κυκλικής ουράς είναι:

1. Δημιουργία ουράς

Η δημιουργία μιας ουράς πραγματοποιείται με την εκχώριση της τιμής κάτω_όριο_δείκτη στους δείκτες της.

```
procedure δημιούργησε_ουρά (var ουρά: τύπος_ουράς);
 begin
   with ουρά do
   begin
     πίσω_δείκτης:= κάτω_όριο_δείκτη;
     μπροστά_δείκτης:= κάτω_όριο_δείκτη
   end
 end (δημιούργησε_ουρά)
```

2. Ελεγχος αν μια κυκλική ουρά είναι κενή

Οπως ήδη αναφέραμε οι εισαγωγές σε μια ουρά γίνονται από το πίσω μέρος της ουράς. Σε κάθε εισαγωγή μετακινείται ο πίσω_δείκτης. Οταν ο πίσω_δείκτης έχει την τιμή μηδέν, τότε δεν έχει γίνει καμία εισαγωγή και η ουρά είναι κενή. Οταν και οι δύο δείκτες της ουράς δείχνουν στην ίδια (διαφορετική του μηδενός) θέση, τότε η ουρά μπορεί να είναι είτε άδεια, είτε γεμάτη. Οταν

οι δείκτες της ουράς δείχνουν στην ίδια θέση και η ουρά είναι άδεια, τότε σίγουρα αυτό έχει συμβεί μετά από εξαγωγή. Για το λόγο αυτό, στην περίπτωση εξαγωγής ενός στοιχείου από μια ουρά, θα πρέπει να εξετάζουμε αν οι δείκτες είναι ίσοι. [Αν αυτό ισχύει, τότε πρέπει να κάνουμε αρχικοποίηση της ουράς με τη διαδικασία δημιούργησε_ουρά]. Είναι φανερό πως για να εξετάσουμε αν μια ουρά είναι κενή, αρκεί να ελέγξουμε αν ο πίσω δείκτης δείχνει στη θέση κάτω_όριο_δείκτη.

```
function κενή (ουρά: τύπος_ουράς): boolean;
 begin
   κενή:= ουρά.πίσω_δείκτης = κάτω_όριο_δείκτη
 end {κενή}
```

3. Ελεγχος αν η ουρά είναι γεμάτη.

Μια ουρά θα είναι γεμάτη ή όταν ο πίσω_δείκτης ισούται με πάνω_όριο και ο μπροστά_δείκτης ισούται με κάτω_όριο_δείκτη, ή όταν ο πίσω_δείκτης ισούται με τον μπροστά_δείκτης και πίσω_δείκτης ≠0.

```
function γεμάτη (ουρά: τύπος_ουράς): boolean;
 begin
   with ουρά do
   if (πίσω_δείκτης <> κάτω_όριο_δείκτη) then
   γεμάτη:= (μπροστά_δείκτης = πίσω_δείκτης) or
            (μπροστά_δείκτης = κάτω_όριο_δείκτη) and
            (πίσω_δείκτης = πάνω_όριο)
   else γεμάτη:= false
 end {γεμάτη}
```

4. Εισαγωγή στοιχείου σε ουρά (enqueue)

Πριν να εισαγάγουμε ένα στοιχείο σε μια ουρά πρέπει να ελέγχουμε αν η ουρά είναι γεμάτη. Ετσι, η συνάρτηση "γεμάτη" πρέπει πάντα να καλείται πριν από την κλήση της διαδικασίας "εισάγαγε".

```
procedure εισάγαγε (x: τύπος_στοιχείου;
                    var ουρά: τύπος_ουράς);
 begin
   with ουρά do
    begin
      πίσω_δείκτης:= (πίσω_δείκτης mod πάνω_όριο) + 1;
      q [πίσω_δείκτης]:= x
    end
 end {εισάγαγε}
```

5. Εξαγωγή στοιχείου από ουρά (dequeue)

Πριν εξαγάγουμε ένα στοιχείο από μια ουρά πρέπει να ελέγχουμε πάντα αν η ουρά είναι κενή. Έτσι, η συνάρτηση "κενή" πρέπει πάντα να καλείται πριν από την κλήση της διαδικασίας "εξάγαγε".

```
procedure εξάγαγε (var x: τύπος_στοιχείου;
                   var ουρά: τύπος_ουράς);
begin
  with ουρά do
  begin
    μπροστά_δείκτης:= (μπροστά_δείκτης mod πάνω_όριο) + 1;
    x:= q [μπροστά_δείκτης];
    if μπροστά_δείκτης = πίσω_δείκτης then
     δημιούργησε_ουρά (ουρά)
  end
end {εξάγαγε}
```

3.2.6 Ουρές προτεραιότητας - Εφαρμογές

Οι ουρές προτεραιότητας (priority queues) χαρακτηρίζονται από την ιδιότητα **"Το στοιχείο με την υψηλότερη προτεραιότητα εισέρχεται και εξέρχεται πρώτο". Highest-Priority-In - First-Out (HPIFO)**. Χαρακτηριστικό παράδειγμα μια τέτοιας ουράς είναι η ουρά των ασθενών σ'ένα εφημερεύον νοσοκομείο. Σε αντίθεση με τις ουρές FIFO, στις οποίες ο χρόνος άφιξης ενός στοιχείου στην ουρά καθορίζει και την "προτεραιότητά" του, ο χρόνος άφιξης ενός στοιχείου στις ουρές HPIFO δεν έχει καμιά σημασία.

Ο ορισμός μιας ουράς HPIFO πρέπει να περιλαμβάνει για το κάθε στοιχείο ένα επί πλέον πεδίο, την προτεραιότητα του στοιχείου. Η προτεραιότητα αυτή μπορεί να καθορίζεται από το χρήστη. Εκτός από την εισαγωγή και εξαγωγή των στοιχείων, οι πράξεις στις ουρές HPIFO είναι ίδιες όπως στις ουρές FIFO . Από μια ουρά προτεραιότητας εξαγάγουμε πάντα το στοιχείο με τη μεγαλύτερη προτεραιότητα. Όταν συμβεί να έχουν δύο ή περισσότερα στοιχεία την ίδια προτεραιότητα, τότε αυτά τυχαίνουν επεξεργασίας, με LIFO ή FIFO ή με κάποια άλλη τεχνική.

Ενα από τα βασικότερα προβλήματα των **συστημάτων πολυπρογραμματισμού (multiprogramming systems)**, είναι ο χρονοπρογραμματισμός των απαιτήσεων εισόδου/εξόδου για μια συσκευή δίσκων. Η εξυπηρέτηση των απαιτήσεων αυτών μπορεί να δημιουργήσει "μποτιλιάρισμα" (bottleneck) το οποίο μπορεί να

επηρεάσει δυσμενώς την απόδοση του συστήματος. Κάθε χρονική στιγμή ο δίσκος έχει μια ουρά από απαιτήσεις εισόδου/εξόδου που περιμένουν για εξυπηρέτηση και που βέβαια θα πρέπει να εξυπηρετηθούν με κάποια σειρά. Η "πολιτική" εξυπηρέτησης των απαιτήσεων εισόδου/εξόδου από το δίσκο αναφέρεται ως πολιτική χρονοπρογραμματισμού του δίσκου (disk scheduling policy).

Μια μέθοδος χρονοπρογραμματισμού μπορεί να είναι η FIFO, δηλαδή οι απαιτήσεις εισόδου/εξόδου εξυπηρετούνται με τη σειρά που φτάνουν στο δίσκο.

Ο χρόνος εξυπηρέτησης μιας απαίτησης αποτελείται κυρίως από τρεις χρόνους:

(α) **το χρόνο αναζήτησης (seek time)** του ίχνους που περιέχει τα ζητούμενα δεδομένα,
(β) **την περιστροφική καθυστέρηση (rotation delay),** έως ότου η κεφαλή έρθει στο ζητούμενο **τομέα (sector),** και
(γ) **το χρόνο μεταφοράς (transfer time)** των δεδομένων.

Επειδή ο χρόνος αναζήτησης ενός ίχνους είναι πολύ μεγαλύτερος σε σχέση με τους δύο άλλους χρόνους, μια στρατηγική χρονοπρογραμματισμού θα ήταν να εξυπηρετείται εκείνη η απαίτηση εισόδου/εξόδου που είναι πιο κοντά στην κεφαλή του δίσκου. Με τον τρόπο αυτό ελαχιστοποιείται ο χρόνος αναζήτησης. Η μέθοδος αυτή αναφέρεται σαν **"Εξυπηρέτηση με το Μικρότερο Χρόνο Αναζήτησης Πρώτα" (Shortest Seek Time First, SSTF).** Η στρατηγική αυτή εκφράζεται με μια ουρά προτεραιότητας. Πράγματι, η προτεραιότητα μιας απαίτησης καθορίζεται από την απόσταση μεταξύ του ζητουμένου ίχνους και του ίχνους που βρίσκεται η κεφαλή.

$$\text{Προτεραιότητα} = |I_j - I_k|$$

όπου I_j και I_k είναι οι θέσεις στο δίσκο του ζητούμενου ίχνους και του ίχνους που βρίσκεται η κεφαλή αντίστοιχα. Εύκολα όμως μπορείτε να διαπιστώσετε ότι οι προτεραιότητες των στοιχείων μιας τέτοιας ουράς αλλάζουν συνεχώς. Η μέθοδος SSTF έχει και τα μειονεκτήματά της. Για παράδειγμα όταν έχουμε πολλές γειτονικές αιτήσεις στο δίσκο, τότε σύμφωνα με την μέθοδο SSTF, η κεφαλή θα εξυπηρετήσει πρώτα όλες αυτές τις απαιτήσεις . Αυτό έχει ως αποτέλεσμα τη μεγάλη καθυστέρηση εξυπηρέτησης των άλλων απαιτήσεων.

Μια άλλη μέθοδος εξυπηρέτησης χρονοπρογραμματισμού του δίσκου που συνδυάζεται με την κίνηση της κεφαλής βασίζεται στον **αλγόριθμο σάρωσης (scan).** Καθώς η κεφαλή κινείται από το κέντρο

του δίσκου στην περιφέρεια εξυπηρετεί τις απαιτήσεις που συναντάει κατά την κίνησή της και αγνοεί αυτές που βρίσκονται πίσω της. Έτσι έχουμε δύο ουρές προτεραιότητας. Η μια περιλαμβάνει όλες τις απαιτήσεις "μπροστά" από την κεφαλή, σε αύξουσα σειρά του αριθμού του ίχνους των απαιτήσεων. Η άλλη περιλαμβάνει όλες τις απαιτήσεις που βρίσκονται πίσω από την κεφαλή οι οποίες εξυπηρετούνται όταν η κεφαλή αλλάξει φορά κίνησης. Οι προτεραιότητες των απαιτήσεων αυτών είναι σε φθίνουσα σειρά του αριθμού του ίχνους. Οπωσδήποτε και η μέθοδος αυτή είναι δυναμική καθώς συνέχεια φτάνουν απαιτήσεις εισόδου/εξόδου για ίχνη που βρίσκονται είτε μπροστά, είτε πίσω από την κεφαλή.

Ανακεφαλαίωση

Οι στατικές λίστες υλοποιούνται με έναν πίνακα και έναν ή περισσότερους δείκτες.

Στοίβες: Δομή: LIFO
Υλοποίηση: πίνακας και δείκτης κορυφής.
Πράξεις: δημιουργία, κενή, γεμάτη, εισαγωγή (push)
διαγραφή (pop)
Η εισαγωγή/διαγραφή γίνονται μόνο από την κορυφή της στοίβας.

Ουρές: Δομή: FIFO
Υλοποίηση: πίνακας, μπροστά_δείκτης και πίσω_δείκτης.
Πράξεις: δημιουργία, κενή, γεμάτη, εισαγωγή (enqueue)
διαγραφή (dequeue)
Η εισαγωγή στοιχείου γίνεται πάντα από το τέλος και η εξαγωγή από μπροστά.

Ουρές προτεραιότητας: Δομή: HPIFO
Υλοποίηση και πράξεις εκτός της εισαγωγής και διαγραφής όπως στις ουρές FIFO.

Μειονεκτήματα στατικών λιστών: Κίνδυνος υπερχείλισης κατά την εισαγωγή νέου στοιχείου.

Ασκήσεις 3.1

1. Γράψτε μια διαδικασία που να αντιστρέφει τα στοιχεία μιας στοίβας.

2. Γράψτε μια συνάρτηση που να ελέγχει αν μια ακολουθία χαρακτήρων της μορφής:
<ακολουθία χαρακτήρων>*<αντίστροφη ακολουθία χαρακτήρων>$
είναι ορθή.
Για παράδειγμα, η ακολουθία pascal*lacsap$ είναι ορθή.

3. Γράψτε μια συνάρτηση "μήκος_ουράς" η οποία να υπολογίζει το μήκος μιας ουράς. Με τη βοήθεια της συνάρτησης "μήκος_ουράς" γράψτε τη διαδικασία "εισάγαγε_σε_ουρά".

4. Γράψτε τον ορισμό και τις βασικές πράξεις μιας ουράς με δύο άκρα (deque). Η deque χαρακτηρίζεται από την ιδιότητα ότι η εισαγωγή και η εξαγωγή των στοιχείων γίνεται και από τα δύο άκρα της.

5. Δώστε τους ορισμούς σε Pascal για τις δομές: α) ουρά από στοίβες β) στοίβα από ουρές γ) ουρά από ουρές. Γράψτε υποπρογράμματα για την εισαγωγή και την εξαγωγή ενός στοιχείου για κάθε μια από αυτές τις δομές.

6. Δώστε τον ορισμό και τις βασικές πράξεις για τη διπλή στοίβα της παραγράφου 3.2.4.

7. Βελτιώστε τον αλγόριθμο υπολογισμού της επιθεματικής παράστασης, με το να ελέγχετε τη στοίβα των τελεστών κάθε φορά που κάνετε εισαγωγή ή εξαγωγή από αυτήν. Ο αλγόριθμος θα πρέπει επίσης να λαμβάνει υπόψη τυχόν κενά που παρεμβάλονται μεταξύ των ορισμάτων (τελεστέων) και των τελεστών στη παράσταση. Υλοποιείστε τον αλγόριθμό σας σε Pascal.

8. Μια λίστα μπορεί να υλοποιηθεί ως ένας πίνακας από εγγραφές όπου η κάθε εγγραφή περιέχει ένα στοιχείο και έναν ακέραιο που δηλώνει τη θέση του πίνακα στην οποία βρίσκεται το επόμενο στοιχείο της λίστας, ως εξής:

```
λίστες = record
           πλήθος_στοιχείων: 0.. max;
           κεφαλή_λίστας: -1..max;
           κεφαλή_διαθεσίμων: -1..max;
           s: array [0..max] of τύπος_στοιχείων
         end;
var στατική_λίστα: λίστες
```

Η διαγραφή ενός στοιχείου από τη στατική_λίστα θα σημαίνει την εισαγωγή του στοιχείου στη στοίβα των διαθεσίμων θέσεων με κατάλληλη ενημέρωση του δείκτη "κεφαλή_διαθεσίμων". Η

εισαγωγή ενός νέου στοιχείου γίνεται σε θέση που παίρνουμε από την στοίβα των διαθεσίμων θέσεων, εφ'όσον αυτή δεν είναι κενή, διαφορετικά το νέο στοιχείο εισάγεται στο τέλος της λίστας. Παράλληλα ενημερώνουμε τους δείκτες της στατικής_λίστας, έτσι ώστε τα στοιχεία της να παραμένουν ταξινομημένα μετά την εισαγωγή. Γράψτε όλες τις πράξεις για μια στατική λίστα (δημιουργία, κενή, γεμάτη, εισαγωγή_στοιχείου, διαγραφή_στοιχείου).

9. Ενα στενόμακρο γκαράζ με μία μόνο είσοδο/έξοδο μπορεί να χωράει μέχρι 10 αυτοκίνητα. Οταν ένας οδηγός έρχεται να πάρει τ'αυτοκίνητό του, που δεν είναι στην είσοδο, θα πρέπει όλα τ'αυτοκίνητα πριν από αυτό να βγούν από το γκαράζ, μετά να βγει το αυτοκίνητο του οδηγού και μετά τα υπόλοιπα αυτοκίνητα να ξαναμπούν στο γκαράζ με τη σειρά που ήταν πριν. Γράψτε ένα πρόγραμμα που προσομοιώνει τη λειτουργία του γκαράζ για μια ομάδα από δεδομένα. Κάθε γραμμή δεδομένων περιέχει ένα χαρακτήρα που είναι ή 'Α' για άφιξη ή 'Ε' για αναχώρηση και τον αριθμό του αυτοκινήτου. Τ'αυτοκίνητα υποτίθεται ότι έρχονται και φεύγουν με τη σειρά που καθορίζεται από τα δεδομένα εισόδου. Το πρόγραμμα θα πρέπει να δίνει ένα μήνυμα για άφιξη ή αναχώρηση αυτοκινήτου. Οταν το αυτοκίνητο έρχεται θα πρέπει το μήνυμα να μας καθορίζει αν υπάρχει χώρος για το αυτοκίνητο ή όχι. Αν δεν υπάρχει χώρος, τότε τ'αυτοκίνητο φεύγει χωρίς να μπει στο γκαράζ. Οταν ένα αυτοκίνητο φεύγει, το μήνυμα θα πρέπει να αναφέρει πόσες φορές το αυτοκίνητο αυτό μετακινήθηκε έξω από το γκαράζ, για να βγούν άλλα αυτοκίνητα.

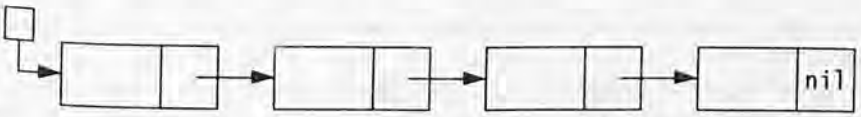
10. Γράψτε τον ορισμό μιας ουράς προτεραιότητας για το χρονοπρογραμματισμό SSTF . Κάθε απαίτηση για είσοδο/έξοδο θα πρέπει να περιέχει ένα κλειδί, τον τύπο της απαίτησης (αν δηλαδή είναι read ή write), το ίχνος και τον τομέα που βρίσκονται τα δεδομένα πάνω στο δίσκο. Υποθέστε ότι η τρέχουσα θέση της κεφαλής είναι μια καθολική μεταβλητή. Γράψτε τις πράξεις που ορίζονται στην ουρά αυτή. Μετά κάθε πράξη εισόδου ή εξόδου υπολογίστε τις νέες προτεραιότητες, που δημιουργούνται με την μετακίνηση της κεφαλής για την εξυπηρέτηση της επόμενης απαίτησης.

11. Γράψτε τους ορισμούς και τις πράξεις των ουρών για τη στρατηγική σάρωσης (scan).

3.3 Δυναμικές Γραμμικές Λίστες

Οπως είδαμε το βασικό μειονέκτημα μιας στατικής λίστας είναι η περίπτωση υπερχείλισης. Στην περίπτωση αυτή είναι αδύνατη η εισαγωγή νέων στοιχείων. Μια λίστα λέγεται **δυναμική (dynamic)**, όταν δεν είναι καθορισμένο από την αρχή το πλήθος των στοιχείων (κόμβων) της. Η Pascal μας δίνει τη δυνατότητα δυναμικής καταχώρησης της μνήμης κατά τη διάρκεια της εκτέλεσης ενός προγράμματος. Ο μεταγλωττιστής της γλώσσας συνδέει με το πρόγραμμα μια ειδική περιοχή της μνήμης που ονομάζεται **σωρός (heap)**, ή **δεξαμενή δυναμικής αποθήκευσης (dynamic storage pool)**. Η διεργασία υποστήριξης της γλώσσας κατά το χρόνο εκτέλεσης, περιλαμβάνει το **διαχειριστή του σωρού της μνήμης (heap manager)**. Αυτός αναλαμβάνει την ευθύνη για τη δυναμική καταχώρηση μνήμης στο σωρό, κατά τη διάρκεια εκτέλεσης ενός προγράμματος της Pascal.

Στην περίπτωση της δυναμικής καταχώρησης οι διαδοχικοί κόμβοι της δομής δεν αποθηκεύονται κατ'ανάγκην σε διαδοχικές θέσεις μνήμης. Τα στοιχεία μιας δυναμικής λίστας υλοποιούνται με μια εγγραφή που περιλαμβάνει δύο πεδία. Το ένα περιέχει τα δεδομένα και το άλλο το σύνδεσμο στη διεύθυνση του επόμενου στοιχείου. Μια λίστα στην οποία κάθε στοιχείο περιέχει τη διεύθυνση του επόμενου στοιχείου λέγεται **(δυναμική) γραμμική λίστα με σύνδεση (linked linear list)** ή **γραμμική λίστα μιας διεύθυνσης, (one-way linear list)**. Μια τέτοια λίστα θα παριστάνεται στα επόμενα όπως δείχνει το σχήμα 3.6.



Σχήμα 3.6

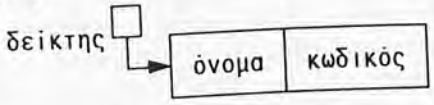
Η δημιουργία δυναμικών εγγραφών στην Pascal γίνεται με τη βοήθεια **δεικτών (pointers)**. Η τιμή ενός δείκτη είναι μια διεύθυνση της μνήμης. Αν αγνοήσουμε τον τρόπο με τον οποίο γίνεται η αναφορά στις διεύθυνσεις της μνήμης (memory addressing) και θεωρήσουμε ότι η μνήμη είναι ένας μονοδιάστατος πίνακας ψηφιοσυλλαβών, τότε η τιμή ενός δείκτη μπορεί να είναι οποιαδήποτε θέση του πίνακα της μνήμης. Οταν ένας δείκτης έχει την τιμή nil, τότε ο δείκτης δε δείχνει πουθενά. Η τιμή της σταθεράς nil εξαρτάται από το μεταγλωττιστή, τη γλώσσα προγραμματισμού και το σύστημα του υπολογιστή που χρησιμοποιείται. Για ένα δείκτη έχουμε ως πεδίο τιμών το σύνολο όλων των διευθύνσεων της διαθέσιμης μνήμης και την τιμή nil. Ενας δείκτης δηλώνεται ως εξής:

```
κόμβος = record
           όνομα: string [10];
           κωδικός: integer
         end;
var δείκτης: ^ κόμβος
```

Το βέλος (^) δηλώνει ότι με τη μεταβλητή "δείκτης" μπορούμε να κάνουμε αναφορά σε εγγραφές του τύπου "κόμβος". Έτσι μια μεταβλητή τύπου δείκτη αναφέρεται σε δεδομένα ενός καθορισμένου τύπου, όπως φαίνεται από την παραπάνω δήλωση. Η τιμή ενός δείκτη μετά την δήλωσή του είναι αόριστη. Η διαδικασία:

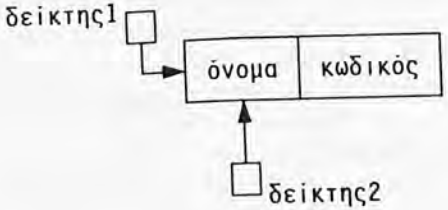
```
new (δείκτης)
```

ορίζει το δείκτη, δηλαδή τον κάνει να δείχνει σε μια εγγραφή τύπου "κόμβος". Έτσι ο τύπος της εγγραφής στην οποία αναφέρεται ο δείκτης εξαρτάται από τον τύπο της παραμέτρου στη διαδικασία new. Σχηματικά έχουμε:



Η διαδικασία new έχει ως αποτέλεσμα την καταχώρηση στο πρόγραμμα θέσεων μνήμης για μια εγγραφή τύπου "κόμβος". Μπορούμε ν' αναφερθούμε στην εγγραφή αυτή με τη μεταβλητή "δείκτης^". Ουσιαστικά η εγγραφή "δείκτης^" είναι η πραγματική δυναμική μεταβλητή, ενώ ο δείκτης όπως δηλώθηκε, είναι μια στατική μεταβλητή.

Ενας δείκτης μπορεί να πάρει μια ορισμένη τιμή και με μια εντολή εκχώρησης. Δύο δείκτες μπορούν να συγκριθούν με τους τελεστές = και <>. Δύο δείκτες μπορούν τέλος να αναφέρονται στην ίδια εγγραφή όπως φαίνεται στο σχήμα 3.7.

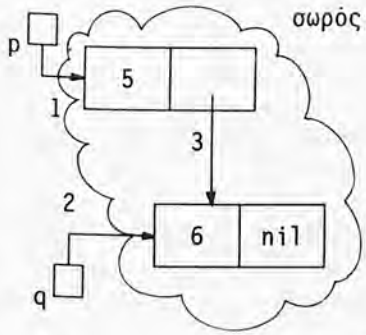


Σχήμα 3.7

Η απουδαιότητα των δεικτών φαίνεται από τη δυνατότητα να έχουμε έναν ή περισσότερους δείκτες μέσα σε μια δυναμική εγγραφή, για παράδειγμα:

```
type κόμβος = record
                περιεχόμενο: integer;
                σύνδεσμος: ^κόμβος
              end;
var δείκτης: ^κόμβος
```

Στο σχήμα 3.8 φαίνεται παραστατικά τι κάνει κάθε μια από τις εντολές του προγράμματος. Εστω οι δηλώσεις: var p,q: ^κόμβος;




```
1:  new (p);
    p^.περιεχόμενο:= 5;
2:  new (q);
    q^.περιεχόμενο:= 6;
    q^.σύνδεσμος:= nil;
3:  p^.σύνδεσμος:= q
```

Σχήμα 3.8

Με την εντολή dispose επιστρέφουμε στο σύστημα τη μνήμη που δε μας χρειάζεται. Αν p είναι μια μεταβλητή τύπου ^κόμβος με τιμή διάφορη της nil, τότε η μνήμη για την εγγραφή p^ απελευθερώνεται και επιστρέφει στο σύστημα με την εντολή dispose(p). Η τιμή του δείκτη p μετά την εκτέλεση της εντολής dispose είναι αόριστη.

Μια δυναμική γραμμική λίστα με σύνδεσμο αποτελεί χαρακτηριστικό παράδειγμα της χρήσης των δεικτών. Θα ορίσουμε εδώ έναν τύπο δείκτη, ώστε να έχουμε τη δυνατότητα να χρησιμοποιήσουμε άλλους δείκτες αυτού του τύπου. Τα στοιχεία στα οποία αναφέρεται ο δείκτης θα είναι εγγραφές με περιεχόμενα κάποια δεδομένα και το σύνδεσμο στο επόμενο στοιχείο της λίστας. Μια τέτοια λίστα ορίζεται ως εξής:

```
type δείκτης_λίστας = ^κόμβος_λίστας;
     τύπος_στοιχείου = ...;
     κόμβος_λίστας = record
                       περιεχόμενο: τύπος_στοιχείου;
                       σύνδεσμος: δείκτης_λίστας
                     end;
     var λίστα: δείκτης_λίστας
```

Οι βασικές πράξεις που ορίζονται σε μια λίστα είναι οι εξής:

α) Δημιουργία λίστας

```
procedure δημιούργησε (var λίστα: δείκτης_λίστας);
begin
  λίστα:= nil
end{δημιούργησε}
```

β) Ελεγχος αν η λίστα είναι κενή

```
function κενή (λίστα: δείκτης_λίστας): boolean;
begin
  κενή:= λίστα = nil
end{κενή}
```

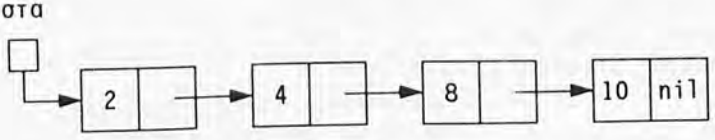
γ) Διάσχιση λίστας (Traverse)

Η παρακάτω διαδικασία επισκέπτεται όλα τα στοιχεία μιας λίστας και τυπώνει τα περιεχόμενά τους.

```
procedure διάσχισε_και_τύπωσε (λίστα: δείκτης_λίστας);
var p: δείκτης_λίστας;
begin
  p:= λίστά;
  while p <> nil do
  begin
    τύπωσε (p^.περιεχόμενο);
    p:= p^.σύνδεσμος
  end
end {διάσχισε_και_τύπωσε}
```

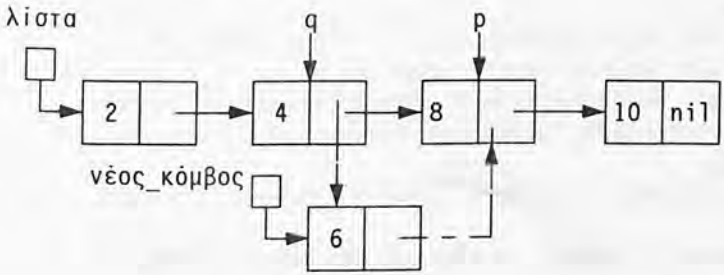
δ) Εισαγωγή νέου στοιχείου.

Εστω ότι θέλουμε να κάνουμε εισαγωγή του στοιχείου 6 στη λίστα του σχήματος 3.9, τα στοιχεία της οποίας είναι ταξινομημένα σε αύξουσα τάξη κάποιου αριθμητικού κλειδιού, έστω.



Σχήμα 3.9

Πρώτα δημιουργούμε ένα νέο κόμβο με περιεχόμενο το 6 και μετά βρίσκουμε τη θέση μέσα στη λίστα που θα πρέπει να τοποθετηθεί ο κόμβος αυτός, έτσι ώστε η νέα λίστα να παραμείνει ταξινομημένη. Οπως φαίνεται από το σχήμα 3.10 ο νέος κόμβος θα πρέπει να εισαχθεί μεταξύ των κόμβων q και p. Μετά πραγματοποιούμε τις συνδέσεις του νέου κόμβου με τη λίστα, όπως φαίνεται από τις διακεκομμένες γραμμές.



Σχήμα 3.10

Η μέθοδος διαφοροποιείται όταν το στοιχείο που θέλουμε να εισαγάγουμε, πρέπει να μπει στην αρχή της λίστας. Ετσι έχουμε τις παρακάτω διαδικασίες εισαγωγής.

```
procedure εισάγαγε_στην_αρχή (x: τύπος_στοιχείου;
                               var λίστα: δείκτης_λίστας);
var νέος_κόμβος: δείκτης_λίστας;
begin
  new (νέος_κόμβος);
  νέος_κόμβος^.περιεχόμενο:= x;
  νέος_κόμβος^.σύνδεσμος:= λίστα;
  λίστα:= νέος_κόμβος
end {εισάγαγε_στην_αρχή};

procedure εισάγαγε (x:τύπος_στοιχείου; var λίστα:δείκτης_λίστας);
var q, p, νέος_κόμβος: δείκτης_λίστας;
                τέλος: boolean;
begin
  {εντόπισε την θέση που πρέπει να μπει ο νέος κόμβος}
  q:= nil;  p:= λίστα; τέλος:= false;
  while (p <> nil) and (not τέλος) do
  if  (p^.περιεχόμενο < x)  then
```

```
  begin
     q:= p;
     p:= p^. σύνδεσμος
   end
   else
   τέλος:= true;
   if  q = nil then εισάγαγε_στην_αρχή (x, λίστα)
    else
    begin
      new (νέος_κόμβος);
      νέος_κόμβος^.περιεχόμενο:= x;
      q^. σύνδεσμος:= νέος_κόμβος;
      νέος_κόμβος^.σύνδεσμος:= p
    end
end {εισάγαγε}
```

ε) Διαγραφή (εξαγωγή) στοιχείου από μία ταξινομημένη λίστα.

Για τη διαγραφή θα χρησιμοποιήσουμε ένα βοηθητικό κόμβο, q, ο οποίος θα δείχνει στον προηγούμενο κόμβο του τρέχοντος κόμβου, έστω p . Οταν εντοπίσουμε τον κόμβο που θέλουμε να διαγράψουμε, τότε κάνουμε το σύνδεσμο του q να δείχνει στον κόμβο που δείχνει ο σύνδεσμος του p. Τέλος επιστρέφουμε στη δεξαμενή δυναμικής αποθήκευσης της μνήμης το κόμβο p.

```
  procedure διάγραψε (var λίστα: δείκτης_λίστας;
                                  x: τύπος_στοιχείου;
                        var έγινε_διαγραφή: boolean);
   var q, p: δείκτης_λίστας;
       βρέθηκε: boolean;
   begin
     if κενή (λίστα) then
       begin
         έγινε_διαγραφή:= false;
         writeln ('η λίστα είναι άδεια')
       end
     else
       begin
         βρέθηκε:= false;
         q:= nil;
         p:= λίστα;
         {διάσχισε τη λίστα για τον εντοπισμό του προς διαγραφήν
                                                        στοιχείου}
         while (p <> nil) and not βρέθηκε  do
           if  (p^. περιεχόμενο < x )  then
```

```
          begin
            q: = p;
            p:= p^.σύνδεσμος
          end
        else
        if p^.περιεχόμενο = x then βρέθηκε:= true
        else {το στοιχείο δεν υπάρχει στη λίστα και η διάσχιση
                                                    σταματάει}
        p:= nil;
        if not βρέθηκε then
         begin
           έγινε_διαγραφή:= false;
           writeln ('το στοιχείο δεν υπάρχει στη λίστα')
         end
          else {το στοιχείο βρέθηκε, κάνε διαγραφή}
         begin
           if (q = nil)  then
               λίστα:= p^.σύνδεσμος
           else
             q^.σύνδεσμος:= p^.σύνδεσμος;
           dispose (p);
           έγινε_διαγραφή:= true
         end
    end
end {διάγραψε}
```

Αλλες πράξεις επί των λιστών είναι η συνένωση (επισύναψη) (concatenation) δύο λιστών και η αντιγραφή (copy) μιας λίστας σε μια άλλη. Η υλοποίηση αυτών αφήνεται ως άσκηση στον αναγνώστη.

Ασκηση. Να γραφεί μια διαδικασία η οποία ν'αντιστρέφει τα στοιχεία μιας λίστας.

```
procedure αντίστρεψε (var λίστα: δείκτης_λίστας);
var προηγούμενος, τρέχων, επόμενος: δείκτης_λίστας;
begin
 if not κενή (λίστα) then
 begin
   επόμενος:= λίστα; τρέχων:= nil;
   while επόμενος <> nil do
   begin
     προηγούμενος:= τρέχων;
     τρέχων:= επόμενος;
     επόμενος:= επόμενος^.σύνδεσμος;
```

```
        τρέχων^.σύνδεσμος:= προηγούμενος;
      end;
      λίστα:= τρέχων
    end
  end {αντίστρεψε}
```

3.4 Ανάλυση της απόδοσης των βασικών πράξεων των γραμμικών λιστών

Οπως είδαμε η εισαγωγή και εξαγωγή ενός στοιχείου από μια λίστα προϋποθέτει τη διάσχιση της λίστας, μέχρις ότου εντοπισθεί η θέση του στοιχείου, ώστε η νέα λίστα να παραμένει ταξινομημένη. Οταν η λίστα δεν είναι ταξινομημένη, τότε η πράξη της εισαγωγής ενός στοιχείου έχει πολυπλοκότητα O(1), ενώ η πράξη της εξαγωγής έχει πολυπλοκότητα O(N). Η ταξινόμηση των στοιχείων μιας λίστας δεν προσφέρει καμμιά βελτίωση στην περίπτωση που κάνουμε ακολουθιακή αναζήτηση και ψάχνουμε για ένα στοιχείο που υπάρχει στη λίστα. Οταν όμως αναζητάμε ένα στοιχείο που δεν υπάρχει στη λίστα, τότε η ταξινόμηση της λίστας βοηθάει στην αναζήτηση, γιατί, όπως θα δούμε παρακάτω, εξοικονομούμε κατά μέσο όρο qN/2 συγκρίσεις, όπου q είναι η πιθανότητα το στοιχείο που ψάχνουμε να μην υπάρχει στη λίστα.

Πράγματι, έστω ότι μια λίστα έχει N διακεκριμένα στοιχεία (κλειδιά), το καθένα με την ίδια πιθανότητα ύπαρξης σε μια θέση της λίστας και έστω ότι ψάχνουμε για ένα κλειδί που έχει πιθανότητα p να υπάρχει στη λίστα και πιθανότητα q=1-p να μην υπάρχει στη λίστα. Τότε το αναμενόμενο πλήθος E(N) των συγκρίσεων για την εντόπιση του κλειδιού αυτού θα είναι:

$$E(N) = p\,E_{\pi}(N) + q\,E_{\alpha}(N)$$

όπου $E_{\pi}(N)$ είναι το αναμενόμενο πλήθος των συγκρίσεων, όταν το στοιχείο υπάρχει στη λίστα και $E_{\alpha}(N)$ το αναμενόμενο πλήθος συγκρίσεων, όταν το στοιχείο δεν υπάρχει στη λίστα. Οταν η λίστα δεν είναι ταξινομημένη, τότε:

$E_{\pi}(N)$ = (πιθανότητα το κλειδί να είναι στη θέση 1) * 1 +
(πιθανότητα το κλειδί να είναι στη θέση 2) * 2 +
....
+(πιθανότητα το κλειδί να είναι στη θέση N) * N

$$= \sum_{i=1}^{N} i \left(\frac{1}{N}\right) = \frac{N+1}{2}$$

Οταν το στοιχείο δεν υπάρχει στη λίστα, τότε διασχίζουμε ολόκληρη τη λίστα για τον εντοπισμό του και το αναμενόμενο πλήθος των συγκρίσεων είναι:

$$E_a(N) = N.$$

Συνεπώς

$$E_1(N) = p\,\frac{N+1}{2} + qN \qquad (1)$$

Οταν η λίστα είναι ταξινομημένη, τότε το αναμενόμενο πλήθος των συγκρίσεων, όταν το στοιχείο υπάρχει στη λίστα, $E_\pi(N)$, είναι το ίδιο με αυτό της περίπτωσης που η λίστα δεν είναι ταξινομημένη και

$E_a(N)$ = (πιθανότητα το κλειδί να είναι μικρότερο του πρώτου στοιχείου) * 1

$+ \sum_{i=2}^{N}$ (πιθανότητα το κλειδί να είναι μεταξύ του (i-1) και (i) στοιχείου) * i + (πιθανότητα το κλειδί να είναι μεγαλύτερο από το N-στό στοιχείο) * N =

$$= 1q_1 + \sum_{i=2}^{N} iq_i + Nq_{N+1} = \frac{(N+1)(N+2)}{(N+1)\,2} - \frac{N}{2} \quad *$$

όταν το N είναι αρκετά μεγάλο.
Στην παραπάνω σχέση που δίνει το $E_a(N)$ έχουμε υποθέσει ότι η πιθανότητα ενός κλειδιού να βρίσκεται μεταξύ του (i-1) και i στοιχείου είναι ίδια για όλα τα i, i=1,2,...,N, δηλαδή $q_i = \frac{1}{N+1}$.

Συνεπώς, όταν η λίστα είναι ταξινομημένη, έχουμε :

$$E_2(N) = p\,\frac{N+1}{2} + q\,\frac{N}{2} \qquad (2)$$

Από τις σχέσεις (1) και (2) φαίνεται ότι η εξοικονόμηση σε συγκρίσεις, που έχουμε όταν η λίστα είναι ταξινομημένη, προκύπτει από την περίπτωση που το κλειδί δεν υπάρχει στη λίστα και είναι:

Εφ' όσον μπορούμε να βρούμε $\frac{1}{N+1} + \frac{2+3+\ldots+N}{N+1} + \frac{N}{N+1} = \frac{1+2+\ldots+N+(N+1)}{N+1} - \frac{1}{N+1} =$

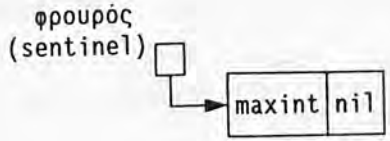
$\frac{N+2}{2} - \frac{1}{N+1} = \frac{(N+2)(N+1)}{(N+1)\,2} - \frac{1}{N+1}$ που πάει $\sim \frac{N}{2}$ για $N >>> M$

$$E_1(N) - E_2(N) = q \cdot N - q \frac{N}{2} = q \frac{N}{2} .$$

Οταν η αναμενόμενη συχνότητα αναζήτησης κλειδιών δεν είναι ίδια, τότε είναι καλό να τοποθετούμε τα κλειδιά στη λίστα σε φθίνουσα σειρά της συχνότητας. Με τον τρόπο αυτόν η αναζήτηση για τα περισσότερα κλειδιά τελειώνει γρήγορα. Οταν η λίστα υλοποιείται με έναν πίνακα και είναι ταξινομημένη, τότε μπορούμε να εντοπίσουμε με δυαδικό ψάξιμο ένα κλειδί σε χρόνο $O(\log_2 N)$. Αυτό δεν μπορεί φυσικά να γίνει σε μια συνδεδεμένη λίστα που χρησιμοποιεί δείκτες. Παρόλο που η αναζήτηση μπορεί να είναι πολύ γρήγορη σε μια στατική και ταξινομημένη λίστα, η εισαγωγή και εξαγωγή στοιχείων είναι χρονοβόρα γιατί απαιτεί μετακίνηση των στοιχείων μετά μια εξαγωγή (και φυσικά υπάρχει πάντοτε το πρόβλημα της υπερχείλισης).

3.5 Κόμβος φρουρός

Οπως είδαμε το κόστος κατά την εισαγωγή και εξαγωγή ενός στοιχείου από μία λίστα ανάγεται στην αναζήτηση, δηλαδή στον εντοπισμό της θέσης του ζητούμενου στοιχείου. Οι διαδικασίες αναζήτησης που παρουσιάσαμε στην προηγούμενη παράγραφο μπορούν ν'απλοποιηθούν, αν θεωρήσουμε ότι σε κάθε λίστα υπάρχει ένας κόμβος φρουρός (sentinel) με κλειδί μεγαλύτερο από τα περιεχόμενα όλων των άλλων κόμβων της λίστας. Για παράδειγμα αν έχουμε μια λίστα που περιέχει ακεραίους, τότε ο κόμβος φρουρός μπορεί να είναι όπως στο σχήμα 3.11.

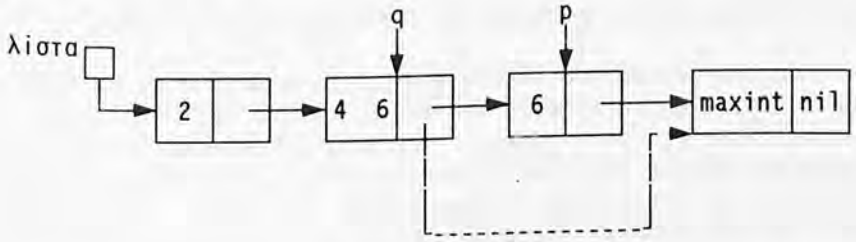


Σχήμα 3.11

Η εισαγωγή ενός νέου στοιχείου σε μια λίστα με κόμβο φρουρό μπορεί τώρα να γίνει απλά, με την παρακάτω διαδικασία:

```
procedure εισάγαγε (var λίστα: δείκτης_λίστας;
                                x: τύπος_στοιχείου);
var p, νέος: δείκτης_λίστας;
begin
  new (νέος);
  p:= λίστα;
  while (x > p^.περιεχόμενο) do
    p:= p^.σύνδεσμος;
  νέος^.περιεχόμενο:= p^.περιεχόμενο;
  νέος^.σύνδεσμος:= p^.σύνδεσμος;
  p^.περιεχόμενο:= x;
  p^.σύνδεσμος:= νέος
end {εισάγαγε}
```

Η διαγραφή στοιχείου από μια λίστα με κόμβο φρουρό μπορεί να γίνει ως εξής. Οταν η λίστα είναι κενή, τότε δεν μπορεί να γίνει διαγραφή. Η λίστα θα είναι κενή όταν περιέχει μόνο τον κόμβο φρουρό. Εστω ότι θέλουμε να διαγράψουμε από τη λίστα του σχήματος 3.12 τον κόμβο με περιεχόμενο το 4.



Σχήμα 3.12

Τότε εντοπίζουμε πρώτα τον κόμβο, έστω q, που θέλουμε να διαγράψουμε, δηλαδή τον κόμβο που περιέχει το 4. Μεταφέρουμε το περιεχόμενο του επόμενου κόμβου έστω p (=q^.σύνδεσμος) στον κόμβο q και πραγματοποιούμε τη σύνδεση όπως φαίνεται στο σχήμα 3.12. Τέλος επιστρέφουμε στη δεξαμενή (pool) δυναμικής αποθήκευσης της μνήμης τον κόμβο p με την εντολή dispose.

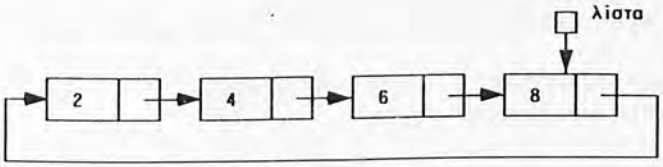
```
procedure διάγραψε (x: τύπος_στοιχείου;
                    var έγινε_διαγραφή: boolean;
                    var λίστα: δείκτης_λίστας);
var
    p, q: δείκτης_λίστας;
    βρέθηκε: boolean;
begin
  if not κενή (λίστα) then
```

```
begin
  q:= λίστα;
  while q^.περιεχόμενο < x do q:= q^.σύνδεσμος;
  βρέθηκε:= q^.περιεχόμενο = x;
  if βρέθηκε then
     begin
       p:= q^.σύνδεσμος;
       q^.περιεχόμενο:= p^.περιεχόμενο;
       q^.σύνδεσμος:= p^.σύνδεσμος;
       έγινε_διαγραφή:= true;
       dispose(p)
     end
  else
    έγινε_διαγραφή:= false
end
 else {κενή(λίστα)}
   έγινε_διαγραφή:= false
end {διάγραψε}
```

3.6 Κυκλικές λίστες ή δακτύλιοι (Circular lists ή rings)

Από τον ορισμό που δώσαμε για μια συνδεδεμένη γραμμική λίστα μιας διεύθυνσης, η εισοδος στη λίστα γίνεται με τη χρήση του δείκτη "λίστα", που δείχνει την αρχή της λίστας. Έτσι, όταν έχουμε να κάνουμε επισύναψη (concatenation) δύο λιστών, θα πρέπει οπωσδήποτε να διασχίσουμε τη μια λίστα και αυτό απαιτεί χρόνο O(N). Θα μπορούσαμε, βεβαίως, σε μια λίστα μιας διεύθυνσης, να κρατάμε ένα δείκτη προς τον τελευταίο κόμβο της λίστας, οπότε η επισύναψη θα ήταν πολύ απλή. Το σημαντικότερο όμως, για μια λίστα μιας διεύθυνσης, είναι ότι δοθέντος ενός δείκτη, που δείχνει σ'έναν κόμβο της λίστας, δεν μπορούμε να επισκεφθούμε τους προηγούμενους κόμβους εκτός εάν διατηρήσουμε το δείκτη της αρχής (κεφαλής) της λίστας. Για τους λόγους αυτούς έχουμε τις κυκλικές λίστες.

Σε μια κυκλική λίστα η σύνδεση του τελευταίου κόμβου, αντί να είναι nil, δείχνει πίσω στον πρώτο κόμβο. Έτσι μπορούμε να φτάσουμε από οποιαδήποτε θέση σε οποιοδήποτε κόμβο θέλουμε. Χρειαζόμαστε όμως έναν εξωτερικό δείκτη, που να μας "εισάγει" στη λίστα. Ο δείκτης αυτός συμφωνούμε να δείχνει προς το δεξιότερο κόμβο της λίστας (τελευταίο κόμβο). Η σύνδεση του κόμβου αυτού δείχνει στον αριστερότερο κόμβο ή πρώτο κόμβο της λίστας όπως φαίνεται στο σχήμα 3.13.



Σχήμα 3.13

Ο ορισμός μιας κυκλικής λίστας χωρίς κόμβο φρουρό, η δημιουργία και ο έλεγχος αν είναι κενή είναι ίδιοι με αυτά μιάς απλής γραμμικής λίστας. Οι υπόλοιπες πράξεις είναι:

(α) Εισαγωγή στοιχείου στην αρχή

```
procedure εισάγαγε_στην_αρχή (var λίστα: δείκτης_λίστας;
                                x: τύπος_στοιχείου);
var νέος: δείκτης_λίστας;
begin
   new (νέος);
   νέος^.περιεχόμενο:= x;
   if not κενή (λίστα) then
   begin
     νέος^.σύνδεσμος:= λίστα^.σύνδεσμος;
     λίστα^.σύνδεσμος:= νέος
   end
   else
   begin
     νέος^.σύνδεσμος:= νέος;
     λίστα:= νέος
   end
end {εισάγαγε_στην_αρχή}
```

(β) Εισαγωγή στοιχείου στο τέλος

Στην περίπτωση αυτή ο δείκτης, που μας εισαγάγει στην κυκλική λίστα, θα δείχνει στον κόμβο με το νέο στοιχείο.

```
procedure  εισάγαγε_στο_τέλος (var λίστα: δείκτης_λίστας;
                                x: τύπος_στοιχείου);
var
   νέος: δείκτης_λίστας;
begin
  new (νέος);
```

```
  νέος^.περιεχόμενο:= x;
  if not κενή (λίστα) then
  begin
      νέος^.σύνδεσμος:= λίστα^.σύνδεσμος;
      λίστα^.σύνδεσμος:= νέος
  end
  else
    νέος^.σύνδεσμος:= νέος;
    λίστα:= νέος
end {εισάγαγε_στο_τέλος}
```

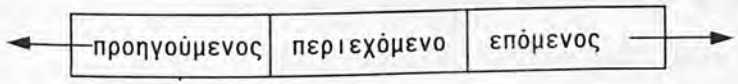
(γ) Διάσχιση κυκλικής λίστας

```
procedure διάσχισε_και_τύπωσε (λίστα: δείκτης_λίστας);
var
   p: δείκτης_λίστας;
  begin
    if not κενή (λίστα) then
      begin
        p:= λίστα^.σύνδεσμος;
        while (p <> λίστα) do
        begin
          τύπωσε (p^.περιεχόμενο);
          p:= p^.σύνδεσμος
        end;
        τύπωσε (p^.περιεχόμενο)
      end
  end {διάσχισε_και_τύπωσε}
```

(δ) Οι υπόλοιπες πράξεις, εξάγαγε_από_την_αρχή, εξάγαγε_από_το_τέλος, εισάγαγε και εξάγαγε αφήνονται ως άσκηση στον αναγνώστη.

3.7 Γραμμικές λίστες δύο συνδέσεων (doubly linked lists)

Στις γραμμικές λίστες με μια σύνδεση, η διάσχιση γίνεται μόνο κατά μια διεύθυνση. Σε πολλές εφαρμογές, όμως, είναι επιθυμητή η διάσχιση της λίστας προς δύο διευθύνσεις, δηλαδή και προς τα "εμπρός" και προς τα "πίσω". Αυτό σημαίνει ότι κάθε κόμβος της λίστας θα έχει δύο δείκτες, "προηγούμενος" και "επόμενος", που θα δείχνουν προς τον προηγούμενο και τον επόμενο κόμβο αντίστοιχα, όπως φαίνεται στο σχήμα 3.14



Σχήμα 3.14

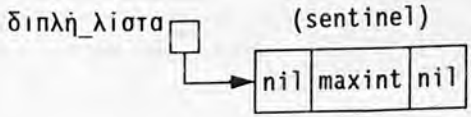
Μια τέτοια λίστα ονομάζεται **λίστα δύο συνδέσεων** ή **δύο διευθύνσεων (doubly linked, two-way linear list).** Σε μια λίστα δύο διευθύνσεων ο επόμενος του προηγούμενου ενός κόμβου p και ο προηγούμενος του επομένου του είναι ο ίδιος ο κόμβος δηλαδή:

```
p^.επόμενος^.προηγούμενος = p
p^.προηγούμενος^.επόμενος = p
```

Ορισμός και βασικές πράξεις λίστας δύο συνδέσεων με κόμβο φρουρό

```
type τύπος_στοιχείου = {δίνεται από τον χρήστη, έστω:} integer;
     δείκτης_διπλής = ^ κόμβος_διπλής;
      κόμβος_διπλής =  record
                         περιεχόμενο: τύπος_στοιχείου;
                         προηγούμενος, επόμενος: δείκτης_διπλής
                       end;
var διπλή_λίστα: δείκτης_διπλής
```

(α) Δημιουργία λίστας δύο συνδέσεων με κόμβο φρουρό. (βλέπε σχήμα 3.15)



Σχήμα 3.15

```
procedure δημιουργία (var διπλή_λίστα: δείκτης_διπλής);
{η διαδικασία αυτή δημιουργεί τον κόμβο φρουρό}
begin
  new (διπλή_λίστα);
  with διπλή_λίστα^ do
  begin
    περιεχόμενο:= maxint;
    προηγούμενος:= nil;
    επόμενος:= nil
  end
end {δημιουργία}
```

(β) Ελεγχος αν η λίστα δύο συνδέσεων είναι κενή

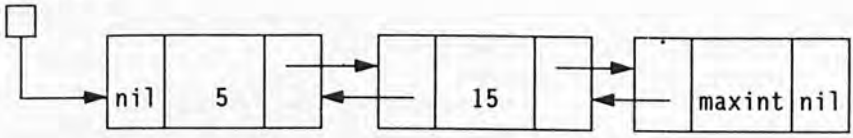
Η λίστα θα είναι κενή όταν ο δείκτης διπλή_λίστα δείχνει στον κόμβο φρουρό.

```
function κενή (διπλή_λίστα: δείκτης_διπλής): boolean;
begin
  κενή:= διπλή_λίστα^.περιεχόμενο = maxint
end
```

(γ) Εισαγωγή νέου στοιχείου

Εστω πως θέλουμε να εισαγάγουμε το στοιχείο 10 στη λίστα του σχήματος 3.16, έτσι ώστε η λίστα που προκύπτει να είναι ταξινομημένη.

διπλή_λίστα



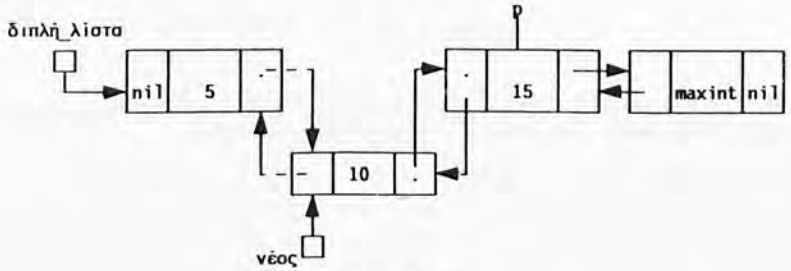
Σχήμα 3.16

Ο αλγόριθμος εισαγωγής περιγράφεται ως εξής:

α) διάσχισε τη λίστα για τον εντοπισμό της θέσης που θα ενταχθεί το νέο στοιχείο,

β) δημιούργησε έναν κόμβο με περιεχόμενο το στοιχείο, έστω 10 και

γ) πραγματοποίησε τις συνδέσεις, όπως φαίνεται με τις διακεκομμένες γραμμές στο σχήμα 3.17.



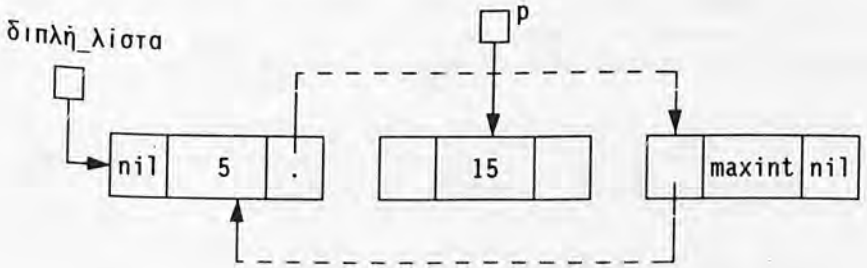
Σχήμα 3.17

```
procedure εισάγαγε (x: τύπος_στοιχείου;
                    var διπλή_λίστα: δείκτης_διπλής);
var
   p, νέος: δείκτης_διπλής;
begin
  {εντόπισε τη θέση που πρέπει να μπει το νέο στοιχείο}
  p:= διπλή_λίστα;
  while  (x > p^.περιεχόμενο)  do
     p:= p^.επόμενος;
  {δημιούργησε ένα νέο κόμβο με περιεχόμενο το νέο στοιχείο}
  new (νέος);
  νέος^.περιεχόμενο:= x;
  {κάνε τις κατάλληλες συνδέσεις}
  νέος^.προηγούμενος:= p^.προηγούμενος;
  νέος^.επόμενος:= p;
  if p = διπλή_λίστα then
     διπλή_λίστα:= νέος
  else
  p^.προηγούμενος^.επόμενος:= νέος;
  p^.προηγούμενος:= νέος
end {εισάγαγε}
```

(δ) Διαγραφή στοιχείου από λίστα δύο συνδέσεων

Ο αλγόριθμος της διαγραφής περιγράφεται ως εξής:

Αν η λίστα δεν είναι κενή, τότε τη διασχίζουμε έως ότου εντοπίσουμε το στοιχείο που θέλουμε να διαγράψουμε, έστω το 15. Πραγματοποιούμε τις συνδέσεις, όπως φαίνεται στο σχήμα 3.18 και επιστρέφουμε στη δεξαμενή δυναμικής καταχώρησης της μνήμης τον κόμβο με την εντολή dispose.



Σχήμα 3.18

Οταν το στοιχείο που θέλουμε να διαγράψουμε δεν υπάρχει στη λίστα, τότε η διαδικασία της διαγραφής δεν κάνει τίποτα.

```
procedure διάγραψε (x: τύπος_στοιχείου;
                    var διπλή_λίστα: δείκτης_διπλής);
var
   p: δείκτης_διπλής;
begin
if not κενή (διπλή_λίστα) then
begin
  {εντόπισε το κόμβο που περιέχει το στοιχείο x}
  p:= διπλή_λίστα;
  while (x > p^.περιεχόμενο)  do
     p:= p^.επόμενος;
  if p^.περιεχόμενο = x then
  begin {το στοιχείο βρέθηκε}
    p^.επόμενος^.προηγούμενος:= p^.προηγούμενος;
    if p = διπλή_λίστα then
       διπλή_λίστα:= p^.επόμενος
    else
       p^.προηγούμενος^.επόμενος:= p^.επόμενος;
    dispose (p)
  end
end
end {διάγραψε}
```

3.8 Αυτο-διοργανούμενες λίστες (self-organized lists)

Οπως είδαμε στην παράγραφο 3.4, η αναζήτηση ενός στοιχείου σε μια λίστα απαιτεί χρόνο O(N), όπου N είναι το μήκος της λίστας. Ωστόσο, στις εφαρμογές, πολύ συχνά η συχνότητα αναζήτησης ορισμένων στοιχείων μιας λίστας είναι μεγαλύτερη από τη συχνότητα των υπόλοιπων στοιχείων της. Αν συμβαίνει να γνωρίζουμε εκ των προτέρων τις συχνότητες αναζήτησης των στοιχείων μιας λίστας, τότε τα στοιχεία με υψηλή συχνότητα πρέπει να εισάγονται στην αρχή, έτσι ώστε το κόστος αναζήτησης γι'αυτά να είναι μικρό. Οι λίστες που οργανώνονται μ'αυτό τον τρόπο είναι γνωστές ως **λίστες ταξινομημένες με τη συχνότητα (frequency ordered lists)**.

Αν γνωρίζουμε τις πιθανότητες με τις οποίες αναζητούνται τα στοιχεία μιας λίστας, τότε το αναμενόμενο πλήθος των συγκρίσεων για μια επιτυχή ακολουθιακή αναζήτηση θα είναι:

$$E = 1p_1 + 2p_2 + \ldots + np_n$$

όπου p_i είναι η πιθανότητα το στοιχείο στη θέση i, i=1,2,..., n να είναι ο στόχος (target) της αναζήτησης. Η παράσταση E ελαχιστοποιείται όταν:

$$p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n.$$

Για το λόγο αυτό, τα στοιχεία θα πρέπει να είναι ταξινομημένα στη λίστα με φθίνουσα τάξη της συχνότητας με την οποία αναζητιούνται. Όταν $p_i = 1/n$, για όλα τα $i = 1,\ldots, n$ τότε η διάταξη των στοιχείων δεν έχει καμιά σημασία.

Όταν όμως, οι συχνότητες αναζητήσης των στοιχείων δεν είναι γνωστές εκ των προτέρων, ή όταν αλλάζουν συναρτήσει του χρόνου, τότε η λίστα θα μπορούσε από μόνη της να αναδιατάσσεται, έτσι ώστε τα στοιχεία με την υψηλότερη συχνότητα να είναι κοντά στην αρχή της. Μια τέτοια λίστα ονομάζεται **αυτο-διοργανούμενη (self organized list)**.

Μια μέθοδος υπολογισμού των συχνοτήτων αναζήτησης των διαφόρων στοιχείων είναι να χρησιμοποιήσουμε τις υπάρχουσες συχνότητες ως πρώτη προσέγγιση. Οι συχνότητες αυτές θεωρούνται αρχικά όλες ίσες με 0. Κάθε φορά που αναζητείται ένα στοιχείο, ενημερώνεται μια μεταβλητή που παριστάνει τη συχνότητα αναζήτησης του στοιχείου αυτού και το στοιχείο μετακινείται προς την αρχή της λίστας. Στην περίπτωση αυτή, για κάθε στοιχείο της λίστας χρειάζεται μια επιπλέον μεταβλητή. Η μεταβλητή αυτή είναι ένας ακέραιος ο οποίος παριστάνει πόσες φορές το στοιχείο ήταν στόχος αναζήτησης. Έτσι, ο ορισμός του τύπου και η δήλωση της λίστας θα είναι:

```
type δεικτης_λιστας = ^κόμβος;
        τύπος_στοιχείου = ...;
        {ορισμένη από τον χρήστη}
        κόμβος = record
              περιεχόμενο: τύπος_στοιχείου;
              συχνότητα: integer;
              σύνδεσμος: δεικτης_λιστας
            end;

var
    αυτοδιοργανούμενη_λίστα: δεικτης_λιστας
```

Η μόνη διαδικασία που διαφοροποιείται από τις βασικές πράξεις των απλών λιστών, είναι η εισαγωγή. Ενας αλγόριθμος εισαγωγής ενός νέου στοιχείου σε μία αυτοδιοργανούμενη λίστα θα μπορούσε να είναι ο εξής:

Αν το στοιχείο που ζητάμε δεν υπάρχει στη λίστα τότε προστίθεται στο τέλος της λίστας με συχνότητα ίση με 1. Αν το στοιχείο υπάρχει ήδη στη λίστα, τότε αυξάνεται η συχνότητά του κατά 1 και το στοιχείο μετακινείται προς τα εμπρός της λίστας, μέχρις ότου περάσει όλα τα στοιχεία πριν από αυτό που έχουν μικρότερη τιμή της συχνότητας.

Παράδειγμα: Να υπολογιστεί η συχνότητα με την οποία εμφανίζονται τα διάφορα γράμματα του αλφαβήτου σ'ένα κείμενο.

Η πιο κατάλληλη δομή για τον αλγόριθμο που περιγράψαμε παραπάνω είναι μια λίστα δύο συνδέσεων. Για κάθε χαρακτήρα που διαβάζεται από το αρχείο input, έστω, καλούμε τη διαδικασία εισάγαγε.

```
procedure εισάγαγε (var αυτοδιοργανούμενη_λίστα: δείκτης_λίστας;
                                                  x: τύπος_στοιχείου);
var p, q, κόμβος: δείκτης_λίστας;
       βρέθηκε: boolean;
begin
  if κενή (αυτοδιοργανούμενη_λίστα) then
  begin
    new (κόμβος);
    κόμβος^.περιεχόμενο:= x;
    κόμβος^.συχνότητα:= 1;
    κόμβος^.επόμενος:= nil;
    κόμβος^.προηγούμενος:= nil;
    αυτοδιοργανούμενη_λίστα:= κόμβος
  end
  else {not κενή αυτοδιοργανούμενη_λίστα}
  begin
    {εντόπισε τη θέση του στοιχείου, αν αυτό υπάρχει στη λίστα}
    p:= αυτοδιοργανούμενη_λίστα;
    while (x <> p^.περιεχόμενο) and (p^.επόμενος <> nil) do
     p:=p^.επόμενος;
    βρέθηκε:= x = p^.περιεχόμενο;
    if βρέθηκε then
    begin
      {αύξησε τη συχνότητα του στοιχείου}
      p^.συχνότητα:= p^.συχνότητα + 1;
       {Μετάφερε το στοιχείο μπροστά στη λίστα έως ότου περάσει
        όλα τα στοιχεία με μικρότερη συχνότητα}
      if (p^.προηγούμενος <> nil) then if (p^.συχνότητα >
                          p^.προηγούμενος^.συχνότητα) then
      begin
        {αποδέσμευσε τον κόμβο που περιέχει το στοιχείο}
        p^.προηγούμενος^.επόμενος:= p^.επόμενος;
        if p^.επόμενος <> nil then p^.επόμενος^.προηγούμενος:=
                                              p^.προηγούμενος;
        q:= p^.προηγούμενος;
        while (q^.συχνότητα < p^.συχνότητα) and (q^.προηγούμενος
                                                  <> nil) do
        q:= q^.προηγούμενος;
```

```
        {πραγματοποίησε τις συνδέσεις του κόμβου p στη νέα του
                                                        θέση}
        if q^.συχνότητα >= p^.συχνότητα then {ενδιάμεση εισαγωγή}
        begin
          p^.επόμενος := q^.επόμενος;
          p^.προηγούμενος := q;
          p^.επόμενος^.προηγούμενος := p;
          q^.επόμενος := p
        end
        else  {εισαγωγή στην αρχή}
        begin
          p^.προηγούμενος := nil;
          p^.επόμενος := q;
          q^.προηγούμενος :=p;
          αυτοδιοργανούμενη_λίστα :=p
        end
    end
    end
    else {δε βρέθηκε, κάνε εισαγωγή του στοιχείου στο τέλος}
    begin
      new (κόμβος);
      κόμβος^.περιεχόμενο:= x;
      κόμβος^.συχνότητα:= 1;
      p^.επόμενος:= κόμβος;
      κόμβος^.προηγούμενος:= p;
      κόμβος^.επόμενος:= nil
    end
  end
end {εισάγαγε}
```

3.9 Εφαρμογές με λίστες

Μια συμβολοσειρά (string) μπορεί να υλοποιηθεί με μια λίστα. Οι πράξεις που ορίζονται στις συμβολοσειρές είναι όμως διαφορετικές από αυτές των λιστών. Για παράδειγμα, η διαδικασία της εισαγωγής μπορεί να εισαγάγει περισσότερα από ένα στοιχεία σε μια συμβολοσειρά. Επί πλέον ο τρόπος υλοποίησης της συμβολοσειράς, δηλαδή πόσους χαρακτήρες θα περιέχει ο κάθε κόμβος της λίστας, διαφοροποιεί ακόμη περισσότερο τις πράξεις αυτές.

Είναι φανερό ότι αν σε κάθε κόμβο της λίστας έχουμε περισσότερους από ένα χαρακτήρα της συμβολοσειράς, τότε οι πράξεις εισαγωγής και διαγραφής γίνονται πολύπλοκες. Για το λόγο αυτό η επιλογή της πιο συμφέρουσας υλοποίησης προϋποθέτει την εκτίμηση του κόστους των πράξεων. Η απλούστερη περίπτωση είναι αυτή στην οποία ο κάθε κόμβος της λίστας περιέχει ένα μόνο χαρακτήρα της συμβολοσειράς.

Ασκηση:

Δίνονται δύο συμβοσειρές s1 και s2. Να γραφεί μια διαδικασία που εισάγει τη συμβολοσειρά s2 μετά τον i-στό χαρακτήρα της s1.

Αλγόριθμος

Εντόπισε τον i χαρακτήρα (κόμβο i) της s1, εφόσον αυτός υπάρχει. Κάνε τη σύνδεση του κόμβου i να δείχνει στην αρχή της s2 και τη σύνδεση του τέλους της s2 να δείχνει στον επόμενο του i-οστού κόμβου της s1.

Επιλογή κατάλληλης δομής

Οπως φαίνεται από τον παραπάνω αλγόριθμο, θα χρειαστεί να κάνουμε πρώτα διάσχιση της s1 μέχρι το i στοιχείο της. Η διάσχιση αυτή απαιτεί i πράξεις. Επί πλέον, όταν χρησιμοποιούμε λίστες μιάς διεύθυνσης, θα χρειαστεί και η διάσχιση της λίστας s2 για να βρούμε τον τελευταίο της κόμβο. Η διάσχιση αυτή απαιτεί κ βήματα όπου κ= length(s2). Συνεπώς ο αλγόριθμος έχει πολυπλοκότητα O(i+κ). Αν όμως χρησιμοποιήσουμε κυκλικές λίστες, για την υλοποίηση των συμβολοσειρών, τότε αποφεύγουμε τη διάσχιση της s2 και ο αλγόριθμος έχει πολυπλοκότητα μόνο O(i). Στη παρακάτω διαδικασία έχουμε υποθέσει ότι ο πρώτος κόμβος της λίστας περιέχει ένα χαρακτήρα, του οποίου ο τακτικός αριθμός ισούται με το μήκος της συμβολοσειράς. Ετσι η διαδικασία length(s) διαβάζει το περιεχόμενο του πρώτου κόμβου της λίστας και επιστρέφει το μήκος της συμβολοσειράς (λίστας) s.

```
procedure  εισάγαγε_συμβολοσειρά (var s1: συμβολοσειρά;
                                   i: integer; s2: συμβολοσειρά);
var   m, j: integer;  p,r:συμβολοσειρά;
begin
  if s1 = nil  then s1:= s2
  else
   if (i >= 0) and (i <= length(s1)) then
   begin
     if s2 <> nil then
      begin
        {υπολόγισε το μήκος της νέας συμβολοσειράς}
        m:= length (s1) + length (s2);
        if m <= 255 then
         begin
           {διάσχισε την s1 μέχρι τον κόμβο i}
           r:= s1^.σύνδεσμος;
           for j:= 1 to i do  r:= r^.σύνδεσμος;
           {αντίγραψε την s2 στην s1}
           p:= s2^.σύνδεσμος^.σύνδεσμος; temp:= r^.σύνδεσμος;
```

```
            while p <> s2^.σύνδεσμος do
              begin
                new (r^.σύνδεσμος);
                r:= r^.σύνδεσμος;
                r^.περιεχόμενο:= p^.περιεχόμενο;
                p:= p^.σύνδεσμος
              end;
            r^.σύνδεσμος:= temp;
            if temp = s1^.σύνδεσμος  then s1:= r;
            {ενημέρωσε τον κόμβο της s1 που περιέχει το μήκος της}
            s1^.σύνδεσμος^.περιεχόμενο:= chr(m)
          end
        else
          writeln ('πολύ μεγάλη συμβολοσειρά')
      end
  end
  else writeln ('Η εισαγωγή απέτυχε: Η τιμή του i είναι εκτός'
                                          ' των ορίων του s2')
end {εισάγαγε_συμβολοσειρά}
```

Ανακεφαλαίωση

Οι βασικές πράξεις επί των λιστών (απλών μιας διεύθυνσης, κυκλικών, δύο διευθύνσεων) είναι: δημιουργία της λίστας, έλεγχος αν η λίστα είναι κενή, αναζήτηση, εισαγωγή και διαγραφή στοιχείου από τη λίστα. Οι πολυπλοκότητες των παραπάνω πράξεων συνοψίζονται στον παρακάτω πίνακα:

Πράξη	Ταξινομημένη λίστα		Μη ταξινομημένη λίστα	
	υλοποίηση με δείκτες	υλοποίηση με πίνακες	υλοποίηση με δείκτες	υλοποίηση με πίνακες
Αναζήτηση	$O(n)$	$O(\log_2 n)$	$O(n)$	$O(n)$
Εισαγωγή	$O(n)$	$O(\log_2 n)$	$O(1)$	$O(1)$
Διαγραφή	$O(n)$	$O(\log_2 n)$	$O(n)$	$O(n)$

Αυτοδιοργανούμενες λίστες με τη συχνότητα: Οι κόμβοι τους ταξινομούνται σε φθίνουσα τάξη της συχνότητας των στοιχείων τους. Οι λίστες αυτές έχουν την δυνατότητα της "αυτόματης" αναδιοργάνωσής τους έτσι ώστε τα στοιχεία με υψηλή συχνότητα αναζήτησης να εμφανίζονται κοντά στην αρχή της λίστας.

Ασκήσεις 3.2

1. Περιγράψτε, για μια δομή τι σημαίνει ακολουθιακή καταχώρηση μνήμης και τι σημαίνει καταχώρηση με δείκτες. Δώστε τα πλεονεκτήματα και μειονεκτήματα του κάθε τρόπου καταχώρησης.

2. Οι κόμβοι μιας λίστας περιέχουν το ονοματεπώνυμο και το υπόλοιπο του λογαριασμού των πελατών μιας τράπεζας. Γράψτε τους κατάλληλους ορισμούς της λίστας αυτής σε Pascal. Γράψτε ένα υποπρόγραμμα που να σχηματίζει τη λίστα "μαύρη_λίστα" όλων των πελατών που το υπόλοιπο του λογαριασμού τους είναι αρνητικό.

3. Γράψτε έναν αλγόριθμο που ν'αλλάζει τις συνδέσεις σε μια γραμμική κυκλική λίστα, έτσι ώστε η σειρά των στοιχείων (κόμβων) στη λίστα να αντιστραφεί.

4. Μια γραμμική λίστα μιας διεύθυνσης "αποθέματα" περιέχει αντικείμενα με τις εξής ιδιότητες:

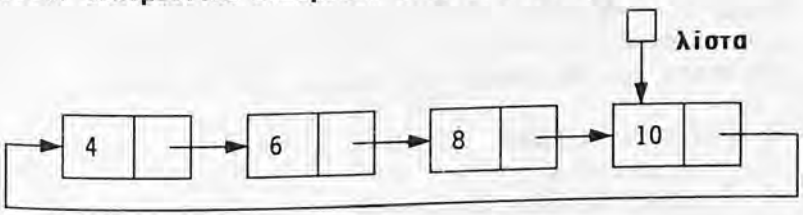
   αριθμός_αποθέματος: ακέραιος που καθορίζει το αντικείμενο,
   στην_αποθήκη
   σημείο_που_παραγγέλονται } ακέραιοι που δηλώνουν ποσότητες.
   έχουν_παραγγελθεί

   Περιγράψτε έναν αλγόριθμο που να δίνει σαν αποτέλεσμα δύο καταστάσεις απογραφής. Η κατάσταση Report_1 να δίνει όλα τ' αντικείμενα για τα οποία "στην_αποθήκη" είναι μεγαλύτερο του "σημείο_που_παραγγέλονται" αλλά το "έχουν_παραγγελθεί" είναι μεγαλύτερο από το μηδέν. Η κατάσταση Report_2 να δίνει όλα τα αντικείμενα για τα οποία "στην_αποθήκη" δεν ξεπερνά το "σημείο_που_παραγγέλονται" και "έχουν_παραγγελθεί" είναι μηδέν. Τυπώστε τις δύο καταστάσεις έτσι ώστε κάθε γραμμή να περιέχει το σχετικό αντικείμενο "αριθμός_αποθέματος" και τις τρεις ποσότητες. Οι καταστάσεις θα πρέπει να σχηματίζονται με μια μόνο διάσχιση της λίστας, όπου η report_1 θα γράφεται κατά τη διάσχιση και η report_2 ν'αποθηκεύεται σε μια κατάλληλη δομή, έτσι ώστε να είναι έτοιμη να γραφτεί στο τέλος. Η εκτύπωση των καταστάσεων θα πρέπει να γίνει με την ίδια σειρά όπως είναι αποθηκευμένα τα αντικείμενα βάσει του ("αριθμός_αποθέματος") στη λίστα.

5. Γράψτε μια διαδικασία:
   delete (var L: δείκτης_λίστας)
   που να διαγράφει ολόκληρη τη λίστα L.

6. Μια συνδεδεμένη γραμμική λίστα μιας διεύθυνσης περιέχει αντικείμενα τύπου "κλειδί". Η λίστα είναι ταξινομημένη με αύξουσα τάξη των τιμών αυτού του τύπου. Χρησιμοποιώντας τις βασικές πράξεις των γραμμικών λιστών, γράψτε μια διαδικασία που να τυπώνει όλες τις τιμές τύπου "κλειδί", που υπάρχουν περισσότερες από μια φορές στη λίστα. Αν δεν υπάρχει επαναλαμβανόμενη τιμή, τότε θα πρέπει να τυπώνεται το μήνυμα "δεν υπάρχει επαναλαμβανόμενη τιμή". Η επαναλαμβανόμενη τιμή θα τυπώνεται μια μόνο φορά, άσχετα από το πόσες φορές επαναλαμβάνεται στη λίστα. Γράψτε πρώτα την λύση πρώτου επιπέδου του αλγόριθμου. Ελέγξτε τον αλγόριθμο προσεκτικά και ειδικά για την περίπτωση που η λίστα έχει ένα μόνο κόμβο.

7. Γράψτε μια αναδρομική διαδικασία που να διατρέχει μια κυκλική λίστα και να τυπώνει τα περιεχόμενα των κόμβων της με αντίστροφη σειρά. ΠΡΟΣΟΧΗ: Δεν πρέπει ν'αντιστρέψετε τη λίστα. **Δοκιμαστικά Δεδομένα**: Οταν δίνεται η λίστα:



Τα **αναμενόμενα αποτελέσματα** θα είναι: 10 8 6 4

8. Δώστε τον ορισμό και τις βασικές πράξεις για τις δομές στοίβα, ουρά μιάς διεύθυνσης και ουρά προτεραιότητας όταν η υλοποίηση αυτών γίνεται με δείκτες.

9. Γράψτε τις διαδικασίες εξάγαγε_από_την_αρχή, εξάγαγε_από_το_τέλος, εισάγαγε και εξάγαγε για μια κυκλική λίστα.

10. Γράψτε τις διαδικασίες "επισύναψε" και "αντίγραψε" για μια γραμμική λίστα μιάς διεύθυνσης και μία κυκλική λίστα.

11. Γράψτε μια αναδρομική διαδικασία που να αντιστρέφει τους δείκτες μιας κυκλικής λίστας.

12. Γράψτε τη διαδικασία εισαγωγής ενός στοιχείου σε μια αυτοδιοργανούμενη λίστα με την εξής μέθοδο. Οταν το στοιχείο δεν υπάρχει στη λίστα, τότε το στοιχείο προστίθεται στο τέλος της λίστας. Οταν το στοιχείο υπάρχει ήδη στη λίστα, τότε το στοιχείο ανταλλάσεται με το προηγούμενό του στη λίστα. Ετσι

τα στοιχεία που εμφανίζονται συχνά μετατίθενται στην αρχή της λίστας.

13. Γράψτε μια διαδικασία (εισάγαγε) που να υπολογίζει τη συχνότητα των γραμμάτων του αλφαβήτου που υπάρχουν σ'ένα κείμενο. Η διαδικασία θα πρέπει να δημιουργεί μια αυτοδιοργανούμενη λίστα ως εξής: Οταν το στοιχείο δεν υπάρχει στη λίστα, τότε εισάγεται στο τέλος, διαφορετικά διαγράφεται το στοιχείο από τη θέση του, εισάγεται στην αρχή της λίστας και αυξάνεται η συχνότητά του.

14. Γράψτε ένα πρόγραμμα που να συγκρίνει αν δύο λίστες L1 και L2 έχουν το ίδιο πλήθος στοιχείων. Το πρόγραμμά σας πρέπει επίσης να διαγράφει και από τις δύο λίστες τα στοιχεία που είναι κοινά σ' αυτές.

15. Γράψτε ένα πρόγραμμα που να διαβάζει από το τερματικό μια σειρά από εγγραφές, που περιέχουν τα στοιχεία των επιβατών μιας αεροπορικής εταιρείας. Κάθε εγγραφή περιέχει τ' όνομα, την πτήση και τον αριθμό καθίσματος ενός επιβάτη.

    Το πρόγραμμα θα πρέπει να κάνει τα εξής:

    α) Εισαγωγή επιβάτη αλφαβητικά στη λίστα.
    β) Αλλαγή της θέσης κάποιου επιβάτη για την ίδια πτήση.
    γ) Διαγραφή επιβάτη.
    δ) Αναφορά της λίστας των επιβατών μιας συγκεκριμένης πτήσης.

16. Ενα αραιό_διάνυσμα παριστάνεται με μια γραμμική λίστα, κάθε κόμβος της οποίας περιέχει έναν πραγματικό αριθμό και έναν ακέραιο. Ο πραγματικός αριθμός παριστάνει την τιμή ενός στοιχείου του αραιού_διανύσματος, ενώ ο ακέραιος παριστάνει τη θέση του στοιχείου. Γράψτε υποπρογράμματα για την πρόσθεση, αφαίρεση και εσωτερικό γινόμενο δύο αραιών_διανυσμάτων.

ΚΕΦΑΛΑΙΟ 4

ΔΕΝΔΡΑ

4.1 Γενικά

Ενα **δένδρο (tree)** είναι μια δομή που χρησιμοποιείται κυρίως για την έκφραση καθαρά ιεραρχικών σχέσεων. Για παράδειγμα η ύλη του μαθήματος Δομές Δεδομένων Ι μπορεί να περιγραφεί όπως φαίνεται στο σχήμα 4.1.

Σχήμα 4.1

Πράγματι παρατηρούμε ότι μεταξύ των κεφαλαίων του μαθήματος υπάρχει μια ιεραρχική, μη γραμμική σχέση. Κάθε κεφάλαιο αποτελείται με τη σειρά του από περισσότερα του ενός υποκεφάλαια και παραγράφους.

Ορισμός: Δένδρο Τ είναι ένα πεπερασμένο σύνολο κόμβων (στοιχείων) με τις εξής ιδιότητες: Υπάρχει ένας ειδικός κόμβος που ονομάζεται

ρίζα (root) του δένδρου. Οι υπόλοιποι κόμβοι χωρίζονται σε m»0 ξένα μεταξύ τους σύνολα $(T_1,\ldots,T_m)$, καθένα από τα οποία είναι ένα δένδρο. Τα δένδρα T_i, i=1,2,...m λέγονται **υπόδενδρα** (subtrees) **της ρίζας.**

Βαθμός κόμβου (node degree) είναι ο αριθμός των υποδένδρων που ξεκινούν από τον κόμβο αυτό.

Βαθμός δένδρου (tree degree) είναι ο μέγιστος βαθμός των κόμβων του. Ανάλογα με τον βαθμό τους τα δένδρα ονομάζονται δυαδικά, τριαδικά, τετραδικά κοκ.

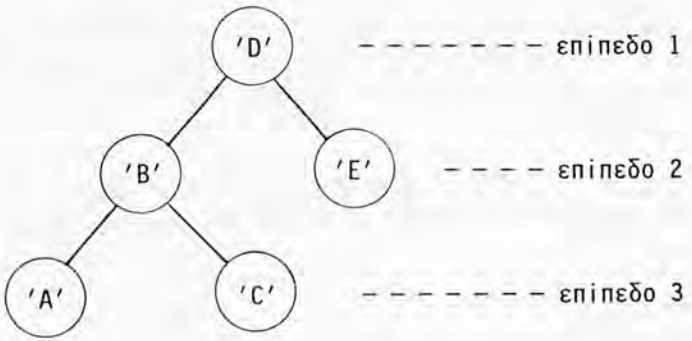
Ενα δένδρο βαθμού d θα ονομάζεται **πλήρες (full)** όταν όλοι οι κόμβοι εκτός από τα φύλλα του έχουν d υπόδενδρα.

Φύλλα ή τερματικοί κόμβοι (leaves, terminal nodes) είναι οι κόμβοι από τους οποίους δεν ξεκινάει άλλο δένδρο. Οι υπόλοιποι κόμβοι του δένδρου λέγονται **εσωτερικοί ή ενδιάμεσοι (internal ή intermediate).**

Ενα δένδρο θα ονομάζεται **διατεταγμένο (ordered tree)** όταν υπάρχει μια σχέση διάταξης μεταξύ των κόμβων του.

Δυαδικό (binary) είναι ένα δένδρο το οποίο είναι είτε κενό είτε έχει έναν κόμβο που ονομάζεται ρίζα του δένδρου και το πολύ δύο ξένα μεταξύ τους δυαδικά υπόδενδρα.

Δυαδικό δένδρο αναζήτησης (binary search tree) είναι ένα δένδρο στο οποίο κάθε κόμβος είναι οργανωμένος, έτσι ώστε, όλοι οι κόμβοι του αριστερού του υποδένδρου να έχουν τιμή μικρότερη από την τιμή του κόμβου αυτού και όλοι οι κόμβοι του δεξιού του υποδένδρου μεγαλύτερη όπως φαίνεται στο σχήμα 4.2.



Σχήμα 4.2

Ιεραρχικές σχέσεις μεταξύ των κόμβων ενός δένδρου αναφέρονται συνήθως για εκείνους τους κόμβους που συνδέονται μεταξύ τους. Οι συνδέσεις αυτές μεταξύ των κόμβων ονομάζονται **κλαδιά (branches)** ή **σύνδεσμοι (ζεύξεις,links)** ή **ακμές (edges).**

Η ακολουθία των κλαδιών που πρέπει να ακολουθηθεί για να φτάσουμε από έναν κόμβο σ'έναν άλλο ονομάζεται **μονοπάτι (path)** μεταξύ των κόμβων αυτών.

Μήκος μονοπατιού (path length) ενός κόμβου Χ, είναι το πλήθος των κλαδιών που πρέπει να διασχίσουμε για να φτάσουμε από τη ρίζα του δένδρου στον κόμβο Χ.

Ο "προηγούμενος" ενός κόμβου λέγεται κόμβος **πατέρας (father) ή γονιός.** Για παράδειγμα στο σχήμα 4.2 ο κόμβος "D" είναι πατέρας των "B" και"E". Ο "επόμενος" ενός κόμβου λέγεται **παιδί (child)** του κόμβου αυτού. Η σχέση αυτή (πατέρας-παιδί), που περιγράφει τα δένδρα, γενικεύεται για τον ορισμό των **προγόνων (ancestors)** και **απογόνων (descendants)** ενός κόμβου.

Εστω Χ ένας κόμβος ενός δένδρου. Αν ο Χ είναι η ρίζα του δένδρου, τότε αυτή δεν έχει προγόνους, διαφορετικά ο πατέρας του Χ και όλοι οι πρόγονοί του είναι επίσης πρόγονοι του Χ. Για παράδειγμα οι πρόγονοι του "C" στο σχήμα 4.2 είναι οι 'B' και 'D'. Μ' άλλα λόγια οι πρόγονοι ενός κόμβου είναι όλοι οι κόμβοι που βρίσκονται σ'ένα μοναδικό απλό μονοπάτι από τη ρίζα προς τον κόμβο. Εστω Υ ένας κόμβος ενός δένδρου. Αν ο Υ είναι φύλλο, τότε δεν έχει απογόνους, διαφορετικά κάθε παιδί του Υ και όλοι οι απόγονοί του είναι απόγονοι του Υ.

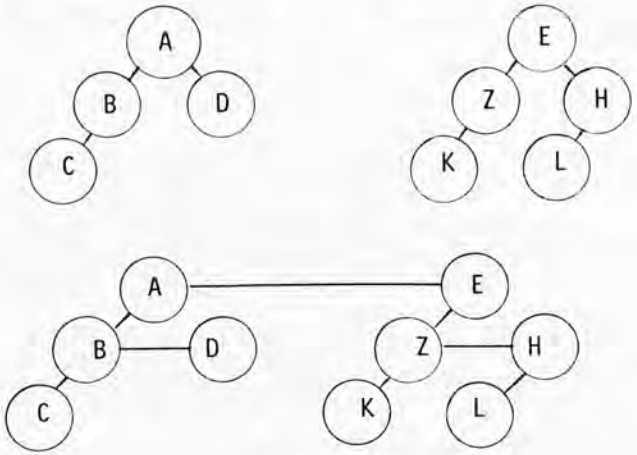
Για τον ορισμό του ύψους ενός δένδρου εισάγουμε την έννοια του **επιπέδου (level)** ενός δένδρου. Κάθε κόμβος ενός δένδρου βρίσκεται σ'ένα επίπεδο. Η ρίζα βρίσκεται στο επίπεδο 1, τα παιδιά της στο επίπεδο 2 κοκ. (βλ. σχ. 4.2). Το μήκος του μονοπατιού ενός κόμβου στο επίπεδο i είναι i - 1.

Υψος (height) ή βάθος (depth) ενός δένδρου είναι ο αριθμός του μεγαλύτερου επιπέδου του. Για παράδειγμα το ύψος του δένδρου του σχήματος 4.2 είναι 3.

Δάσος (forest) είναι ένα διατεταγμένο σύνολο από κανένα ή περισσότερα ξένα μεταξύ τους δένδρα. Ετσι αν από ένα δένδρο διαγράψουμε την ρίζα του, τότε θα έχουμε ένα δάσος και αντίστροφα αν σ'ένα δάσος προσθέσουμε έναν κόμβο, τότε θα έχουμε ένα δένδρο. Επομένως ένας άλλος ορισμός του δένδρου θα ήταν ο εξής: Οι κόμβοι ενός δένδρου εκτός από τη ρίζα αποτελούν ένα δάσος.

Πρόταση. Κάθε δάσος ή δένδρο μπορεί να μετασχηματισθεί σ'ένα δυαδικό δένδρο.

Ο μετασχηματισμός αυτός γίνεται ως εξής: Απομακρύνουμε όλες τις κάθετες συνδέσεις, εκτός από εκείνη του πατέρα προς το πρώτο (αριστερότερο) παιδί και μετά συνδέουμε τα παιδιά που βρίσκονται στο ίδιο επίπεδο, όπως φαίνεται στο σχήμα 4.3.



Σχήμα 4.3

Η παραπάνω διαδικασία μπορεί να αντιστραφεί για να σχηματίσουμε ένα μοναδικό δάσος από ένα δυαδικό δένδρο. Ο μετασχηματισμός που περιγράψαμε λέγεται **φυσική αντιστοιχία (natural correspondence)** μεταξύ δασών και δυαδικών δένδρων.

Μια αυστηρή διατύπωση της φυσικής αντιστοιχίας είναι η εξής: Εστω $F = (T_1,\ldots,T_n)$ ένα δάσος από δένδρα. Το δυαδικό δένδρο B(F) που αντιστοιχεί στο F ορίζεται ως εξής:

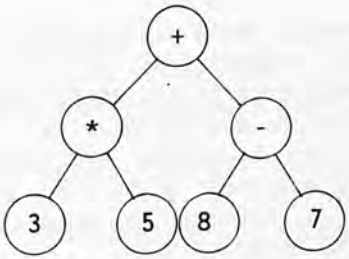
α) Αν $n = 0$, τότε $B(F) = nil$.

β) Αν $n > 0$, τότε η ρίζα του B(F) είναι η ρίζα του T_1. Το αριστερό υπόδενδρο του B(F) είναι:

$$B(T_{11}, T_{12}\ldots T_{1m})$$

όπου $T_{11}, T_{12},\ldots T_{1m}$ είναι m υπόδενδρα της ρίζας του T_1 και το δεξιό υπόδενδρο του B(F) είναι $B(T_2, T_3, \ldots T_n)$.

Ενας από τους λόγους που τα δυαδικά δένδρα παρουσιάζουν μεγαλύτερο ενδιαφέρον είναι ότι κάθε δένδρο ή δάσος μπορεί να παρασταθεί μ'ένα δυαδικό δένδρο. Για το λόγο αυτό παρακάτω θα περιοριστούμε μόνο στα δυαδικά δένδρα. Χαρακτηριστικό παράδειγμα δυαδικού δένδρου είναι τα λεγόμενα **δένδρα παράστασης (expression trees)**, που χρησιμοποιούνται για τον υπολογισμό αριθμητικών παραστάσεων. Για παράδειγμα το δένδρο του σχήματος 4.4 δίνει τον τρόπο υπολογισμού της ενδοθεματικής παράστασης 3 * 5 + (8-7).



Σχήμα 4.4

Πρόταση 1. Ενα αυστηρά πλήρες δένδρο βαθμού d και ύψους h έχει $(d^h-1)/(d-1)$ κόμβους. Η απόδειξη αφήνεται ως άσκηση στον αναγνώστη.

Πρόταση 2. Κάθε πλήρες δυαδικό δένδρο με N φύλλα έχει N-1 εσωτερικούς κόμβους.

Απόδειξη:
Αφού το δένδρο είναι πλήρες, έστω ότι $N=2^k$. Τότε το πλήθος των εσωτερικών κόμβων του δένδρου θα είναι:

$$2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1 = N - 1.$$

Πρόταση 3. Ενα δυαδικό δένδρο με n κόμβους έχει ύψος τουλάχιστον $\lfloor \log_2 n \rfloor$.

Απόδειξη:
Εστω h το ύψος του δένδρου. Τότε $n=2^h-1$ ή $2^h=n+1$ ή $h=\log_2(n+1)$ ή $h=\lfloor \log_2 n \rfloor$. Εφ'όσον υποθέσαμε ότι το δένδρο είναι πλήρες, το ύψος (h) αυτό είναι το ελάχιστο.

Πρόταση 4. Αν n_i είναι το πλήθος των κόμβων με i παιδιά (i=0,1,2) ενός δυαδικού δένδρου που έχει n κόμβους, τότε ισχύει:

$$n_0 = n_2 + 1$$

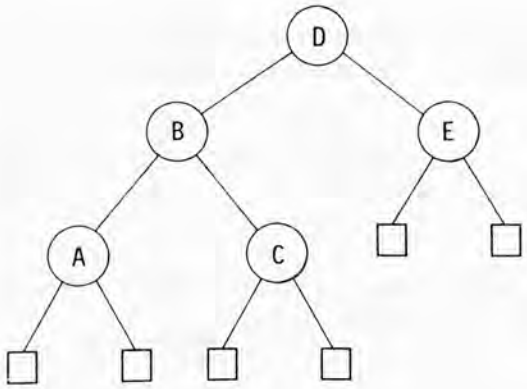
Απόδειξη:
Ισχύει ότι $n_0+n_1+n_2=n$. Ισχύει επίσης ότι το πλήθος των κλαδιών σ'ένα δένδρο με n κόμβους είναι n-1. Πράγματι σε κάθε κόμβο εκτός από τη ρίζα καταλήγει ένα κλαδί. Συνεπώς:

$$n_1 + 2n_2 = n-1$$

Από τις δύο αυτές σχέσεις προκύπτει το ζητούμενο.

Σε θεωρητικές μελέτες βοηθάει μερικές φορές να εξετάζουμε το

επεκτεταμένο δένδρο (extended tree) που αντιστοιχεί σ'ένα δυαδικό δένδρο. Το δένδρο αυτό σχηματίζεται αν αντικαταστήσουμε κάθε κενό υπόδενδρο, μ'έναν **εξωτερικό** ειδικό **κόμβο (external node)**. Οι εξωτερικοί κόμβοι ενός δένδρου αναζήτησης είναι οι κόμβοι στους οποίους φτάνουμε μετά από αναζήτηση ενός στοιχείου, που δεν υπάρχει στο δένδρο. Οι εξωτερικοί κόμβοι θα συμβολίζονται με τετράγωνα. Στο σχήμα 4.5 δίνεται το επεκτεταμένο δένδρο που αντιστοιχεί στο δένδρο του σχήματος 4.2.



Σχήμα 4.5

Σε κάθε εξωτερικό ειδικό κόμβο συνδέουμε την απόστασή του από τη ρίζα. Το άθροισμα E των μηκών (l_i) των μονοπατιών όλων των ειδικών κόμβων ενός δένδρου, ονομάζεται **μήκος εξωτερικού μονοπατιού (external path length)**. Ετσι,

$$E = \sum_{i=1}^{N} l_i$$

όπου N είναι το πλήθος των ειδικών κόμβων. Συνήθως ενδιαφερόμαστε για τον υπολογισμό του $l_{max}=\max_i(l_i)$.

Πρόταση 5. Ισχύει ότι $E = \sum_i s_i(i-1)$, όπου s_i είναι το πλήθος των ειδικών κόμβων στο επίπεδο i.

Με τον ίδιο τρόπο ορίζεται το **εσωτερικό μήκος μονοπατιού (I) (internal path length)** ενός δένδρου, ως το άθροισμα των μηκών των μονοπατιών όλων των κόμβων του. Για το δένδρο του

για παράδειγμα, έχουμε:

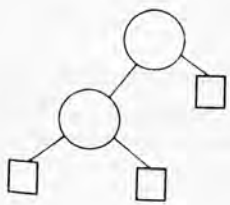
$$I = 0+1+1+2+2 = 1*0+2*1+2*2 = 6$$

Πρόταση 6. Ισχύει ότι $I = \sum_i n_i(i-1)$, όπου n_i είναι το πλήθος των εσωτερικών κόμβων ενός δένδρου στο επίπεδο i.

Πρόταση 7. Σ'ένα δυαδικό δένδρο με n κόμβους ισχύει ότι:

$$E = I + 2n$$

Απόδειξη: Η απόδειξη γίνεται με τελεία επαγωγή. Η σχέση ισχύει για n=1 και n=2. Πράγματι στη περίπτωση n=2 έχουμε:

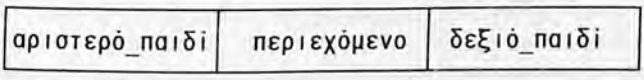


$I = 1$ και $E = 1*1+2*2 = 5$.
Εστω ότι η σχέση ισχύει για $n= k$ δηλαδή $E_k = I_k+2k$, τότε θα αποδείξουμε ότι ισχύει για $n = k + 1$. Αν στο δένδρο με k κόμβους αντικαταστήσουμε έναν εξωτερικό κόμβο στο επίπεδο i μ'έναν εσωτερικό, τότε εμφανίζονται δύο εξωτερικοί κόμβοι στο επίπεδο i+1. Το νέο δένδρο έχει k+1 κόμβους και ισχύουν:

$I_{k+1} = I_k + (i-1)$ και $E_{k+1} = E_k - (i-1) + 2i$ (γιατί;).
Συνεπώς:
$E_{k+1} - I_{k+1} = E_k - I_k + 2 = 2(k+1)$.
Η σχέση ισχύει συνεπώς για κάθε $n \geq 1$.

4.2 Δυναμική υλοποίηση δένδρου

Κάθε κόμβος ενός δένδρου αποτελείται από το περιεχόμενό του και δύο δείκτες που δείχνουν στα παιδιά του, όπως φαίνεται στο σχήμα 4.6.



Σχήμα 4.6

Οι δύο δείκτες αντιστοιχούν στα κλαδιά του δένδρου. Πολλές φορές ένας κόμβος περιέχει κι άλλους δείκτες, για παράδειγμα έναν δείκτη προς τον κόμβο πατέρα, η προς τον επόμενο κόμβο σύμφωνα με

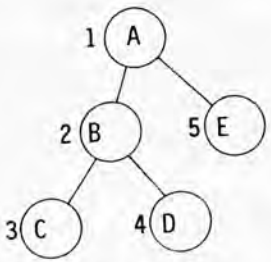
μια μέθοδο διάσχισης του δένδρου κοκ. Η προσθήκη τέτοιων δεικτών κάνει εύκολες ορισμένες πράξεις των δένδρων, οπωσδήποτε όμως απαιτεί σημαντικό επιπλέον χώρο.

Ο ορισμός του τύπου και η δήλωση ενός δένδρου σε Pascal είναι:

```
type τύπος_στοιχείου = {δίνεται από το χρήστη, έστω:} char;
     δείκτης_δένδρου = ^κόμβος_δένδρου;
      κόμβος_δένδρου = record
                          περιεχόμενο: τύπος_στοιχείου;
                          αριστερό_παιδί: δείκτης_δένδρου;
                          δεξιό_παιδί: δείκτης_δένδρου
                       end;
var t: δείκτης_δένδρου
```

4.3 Στατική υλοποίηση δένδρου

Η στατική υλοποίηση ενός δένδρου με τη χρήση πίνακα έχει τα ίδια μειονεκτήματα όπως αυτά της στατικής υλοποίησης των λιστών. Συγκεκριμένα υπάρχει κίνδυνος υπερχείλισης και είναι δύσκολη η εισαγωγή και η διαγραφή κόμβων.



Σχήμα 4.7

Ενας στατικός τρόπος υλοποίησης με χρήση πινάκων είναι ο εξής: Εστω το δένδρο του σχήματος 4.7. Αριθμούμε τους κόμβους του και χρησιμοποιούμε έναν πίνακα από εγγραφές. Κάθε εγγραφή περιέχει το περιεχόμενο του κόμβου και δύο ακεραίους που δείχνουν στις θέσεις των παιδιών του κόμβου μέσα στον πίνακα. Οταν η τιμή ενός δείκτη είναι ίση με μηδέν, τότε ο κόμβος δεν έχει το αντίστοιχο παιδί. Ετσι για το δένδρο του σχήματος 4.7 θα έχουμε:

			l	r
t:	1	A	2	5
	2	B	3	4
	3	C	0	0
	4	D	0	0
	5	E	0	0

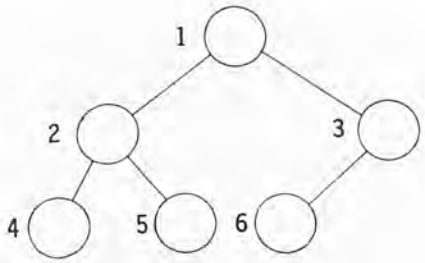
Η θέση t[1] χρησιμοποιείται για τη ρίζα του δένδρου. Ο ορισμός του τύπου και η δήλωση του δένδρου σε Pascal, στη περίπτωση αυτή θα είναι:

```
const μέγεθος_δένδρου = ...; {μέγιστο πλήθος κόμβων}
type τύπος_στοιχείου  = ...; {ορίζεται από το χρήστη}
     τύπος_δείκτη = 0..μέγεθος_δένδρου;
     όρια_δένδρου = 1.. μέγεθος_δένδρου;
     κόμβος_δένδρου = record
                        περιεχόμενο: τύπος_στοιχείου;
                        αριστερό_παιδί: τύπος_δείκτη;
                        δεξιό_παιδί: τύπος_δείκτη
                      end;
var t: array [όρια_δένδρου] of κόμβος_δένδρου
```

Ενας άλλος στατικός τρόπος για την υλοποίηση ενός δένδρου είναι ο εξής: Αριθμούμε τους κόμβους του δένδρου κατά επίπεδα από τ'αριστερά προς τα δεξιά, όπως φαίνεται στο σχήμα 4.8 και χρησιμοποιούμε έναν πίνακα μεγέθους $n=2^h-1$ όπου h είναι το μέγιστο ύψος του δένδρου. Στην περίπτωση αυτή, η δήλωση του δένδρου θα είναι:

```
var t: array [1..n] of τύπος_στοιχείου
```

Κάθε κόμβος i του δένδρου αποθηκεύεται στη θέση t[i] του πίνακα. Η ρίζα του δένδρου αποθηκεύεται στη θέση t[1] και τα παιδιά της στις θέσεις t[2] και t[3]. Γενικά ο κόμβος i αποθηκεύεται στη θέση t[i], το αριστερό του παιδί στη θέση t[2i] και το δεξιό του παιδί στη θέση t[2i+1]. Αν 2i>n ή 2i+1>n, τότε δεν υπάρχει αριστερό ή δεξιό παιδί αντίστοιχα.



Σχήμα 4.8

Παρατηρούμε ότι, με τον τρόπο αυτόν, δε χρειάζονται δείκτες. Κάθε κόμβος μπορεί ν'αποθηκευτεί σε μια μοναδική θέση. Τα μειονεκτήματα της μεθόδου αυτής είναι ότι το μέγεθος του πίνακα θα πρέπει να είναι ίσο με το πλήθος των κόμβων ενός πλήρους δυαδικού δένδρου ύψους h (2^h-1 θέσεις). Συνεπώς, χρησιμοποιούμε επιπλέον χώρο παρ'όλο που αυτός δε χρειάζεται, όταν ένας κόμβος δεν υπάρχει στο δένδρο. Ετσι, στα επόμενα θα χρησιμοποιήσουμε μόνο τη δυναμική παράσταση δένδρων.

Πράξεις επί των δένδρων:

Οι βασικότερες πράξεις επί των δένδρων είναι οι:
1. Δημιουργία δένδρου,
2. Διάσχιση δένδρου,
3. Εισαγωγή νέου κόμβου και
4. Διαγραφή κόμβου.

Αλλες διαδικασίες που θα περιγράψουμε είναι:
5. Υπολογισμός χαρακτηριστικών στοιχείων ενός δένδρου (ύψος, μέγεθος, μήκος εσωτερικού μονοπατιού κ.λπ.),
6. Διαγραφή δένδρου,
7. Αντιγραφή δένδρου και
8. Εκτύπωση περιεχομένων δένδρου.

4.4 Διάσχιση δένδρου (Tree Traversal)

Η διάσχιση ενός δένδρου απαιτεί την επίσκεψη όλων των κόμβων του μια φορά. Για την υλοποίηση της διάσχισης ενός δένδρου θα γράψουμε αναδρομικές διαδικασίες. Αυτό σημαίνει ότι θα πρέπει να μπορούμε να χειριστούμε όλους τους κόμβους του δένδρου με τα υπόδενδρά τους με τον ίδιο ακριβώς τρόπο. Για τον κάθε κόμβο υπάρχουν τρεις δυνατές περιπτώσεις:

1. να επισκεφθούμε τον κόμβο,
2. να επισκεφθούμε το αριστερό του υπόδενδρο και

3. να επισκεφθούμε το δεξιό του υπόδενδρο.

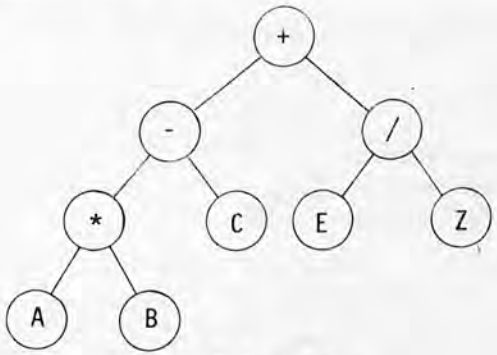
Ανάλογα με τη σειρά που εκτελούνται οι τρεις αυτές διαδικασίες, έχουμε και μια μέθοδο διάσχισης του δένδρου. Συνολικά υπάρχουν έξι τρόποι διάσχισης (3!). Από αυτούς θα εξετάσουμε μόνο τρεις περιπτώσεις, καθόσον αυτές αντιστοιχούν στην ενδοθεματική, επιθεματική και προθεματική μορφή μιας παράστασης.

I. **Ενδοδιατεταγμένη διάσχιση (Inorder traversal)**

Η διάσχιση αυτή περιγράφεται από τα παρακάτω βήματα:

α) Επισκεψη αριστερού υποδένδρου,
β) Επισκεψη ρίζας και
γ) Επισκεψη δεξιού υποδένδρου.

Αν έχουμε το δένδρο του σχήματος 4.9, που αντιστοιχεί στην παράσταση (A*B)-C+E/Z, τότε με τον παραπάνω αλγόριθμο πρώτα θα επισκεφθούμε το αριστερό του υπόδενδρο ({-,*,A,B,C}). Στο σημείο αυτό κάνουμε διάσχιση του υπόδενδρου που έχει ρίζα το -. Με τον ίδιο τρόπο προχωρούμε, μέχρις ότου το αριστερό υπόδενδρο μιας ρίζας να είναι ίσο με nil.



Σχήμα 4.9

Μετά επισκεπτόμαστε τη ρίζα. Για το δένδρο του σχήματος 4.9, ο πρώτος κόμβος που θα επισκεφτούμε θα είναι ο A. Τέλος επαναλαμβάνουμε τη διάσχιση για το δεξιό υπόδενδρο του κόμβου που βρισκόμαστε. Η παρακάτω διαδικασία διασχίζει ένα δένδρο και τυπώνει τα περιεχόμενα των κόμβων του.

```
procedure inorder (t: δεικτης_δένδρου);
begin
  if t <> nil then
```

```
  begin
    inorder (t^.αριστερό_παιδί);
    writeln (t^.περιεχόμενο);
    inorder (t^.δεξιό_παιδί)
  end
end
```

Η σειρά με την οποία επισκεπτόμαστε τους κόμβους του δένδρου είναι η A,*,B,-,C,+,E,/,Z και ταυτίζεται με την **ενδοθεματική (infix)** μορφή της παράστασης.

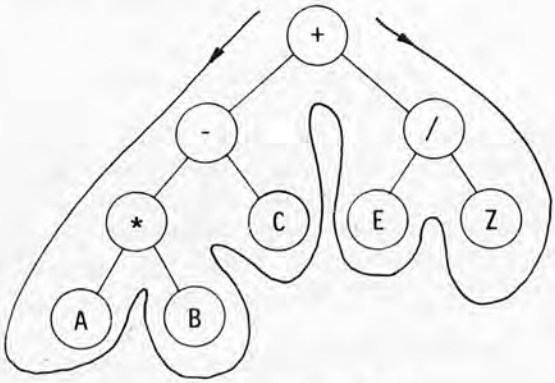
II. **Προδιατεταγμένη διάσχιση (Preorder traversal)**

Η διάσχιση αυτή περιγράφεται από τα βήματα:

α) Επισκεψη της ρίζας,
β) Επισκεψη του αριστερού υποδένδρου και
γ) Επισκεψη του δεξιού υποδένδρου.

Στην προδιατεταγμένη διάσχιση επισκεπτόμαστε έναν κόμβο πριν επισκεφθούμε τα παιδιά του. Η διαδικασία εκτύπωσης των περιεχομένων των κόμβων ενός δένδρου με προδιατεταγμένη διάσχιση είναι:

```
procedure preorder (t: δεικτης_δένδρου);
begin
  if t <> nil then
  begin
    writeln (t^.περιεχόμενο);
    preorder (t^.αριστερό_παιδί);
    preorder (t^.δεξιό_παιδί)
  end
end
```



Σχήμα 4.10

Η σειρά με την οποία θα τυπωθεί το δένδρο του σχήματος 4.9 είναι η +,-,*,A,B,C,/,E,Z. Η σειρά αυτή ταυτίζεται με την **προθεματική (prefix)** μορφή της παράστασης, A*B-C+E/Z. Η σειρά με την οποία γίνεται η επίσκεψη στους κόμβους του δένδρου περιγράφεται με τον εξής μνημονικό κανόνα. Περιβάλλουμε το δένδρο με μια γραμμή όπως φαίνεται στο σχήμα 4.10. Αν διασχίσουμε τη γραμμή κατά την φορά του βέλους, τότε επισκεπτόμαστε έναν κόμβο, όταν τον συναντάμε αριστερά κατά την κίνησή μας.

III. **Μεταδιατεταγμένη διάσχιση (Postorder traversal)**

Η διάσχιση αυτή περιγράφεται από τα βήματα:

α) Επίσκεψη του αριστερού υποδένδρου,
β) Επίσκεψη του δεξιού υποδένδρου και
γ) Επίσκεψη της ρίζας.

Η διαδικασία εκτύπωσης ενός δένδρου με τη μεταδιατεταγμένη διάσχιση είναι:

```
procedure postorder (t: δεικτης_δένδρου);
begin
  if t <> nil then
  begin
    postorder (t^.αριστερό_παιδί);
    postorder (t^.δεξιό_παιδί);
    writeln (t^.περιεχόμενο)
  end
end
```

Η σειρά με την οποία θα τυπωθεί το δένδρο του σχήματος 4.9 είναι η A,B,*,C,-,E,Z,/,+ δηλαδή ταυτίζεται με την **επιθεματική (postfix)** μορφή της παράστασης A*B-C+E/Z. Στην περίπτωση αυτή επισκεπτόμαστε πρώτα τα παιδιά ενός κόμβου και μετά τον κόμβο.

Οι διαδικασίες διάσχισης που περιγράψαμε αν και φαίνονται πολύ εύκολες και ευανάγνωστες έχουν, όπως όλες οι αναδρομικές διαδικασίες, τα μειονεκτήματά τους. Πράγματι το βάρος για την εκτέλεση των διαδικασιών αυτών ανατίθεται στο μεταγλωττιστή της γλώσσας, ο οποίος θα κάνει την μετατροπή των αναδρομικών κλήσεων της κάθε διαδικασίας στη μη αναδρομική γλώσσα μηχανής. Η απόδοση του αναδρομικού αλγορίθμου εξαρτάται συνεπώς από το πόσο καλός είναι ο μεταγλωττιστής.

Μια μη αναδρομική διαδικασία, παρ'όλο που είναι πολύ δυσκολότερο να προγραμματιστεί, είναι περισσότερο αποτελεσματική από την αντίστοιχη της αναδρομική. Πράγματι, στην περίπτωση που κάνουμε διάσχιση ενός υπόδενδρου το οποίο είναι nil, τότε στο αναδρομικό πρόγραμμα θα δημιουργηθεί μια εγγραφή ενεργοποίησης με όλες τις τοπικές μεταβλητές. Η εγγραφή αυτή θα εισαχθεί στη μνήμη στοίβα, παρ'όλο που στην πράξη αυτό δε χρειάζεται.

Παρακάτω δίνουμε την ενδοδιατεταγμένη διάσχιση με μια μη αναδρομική διαδικασία και τη χρήση στοίβας. Η στοίβα είναι υλοποιημένη με δείκτες και τα στοιχεία της είναι τύπου δείκτης_δένδρου.

```
procedure inorder (t: δείκτης_δένδρου);
var τέλος: boolean;
begin
  εισάγαγε (nil, στοίβα);
  while t <> nil do
  begin
    while t^.αριστερό_παιδί <> nil do
      begin
        εισάγαγε (t, στοίβα);
        t:= t^.αριστερό_παιδί
      end;
      τέλος:= false;
      repeat
        write (t^.περιεχόμενο);
        if t^.δεξιό_παιδί = nil then
         begin
           if not κενή (στοίβα) then
           εξάγαγε (t, στοίβα);
           if t = nil then τέλος:= true
         end
        else
         begin
           t:= t^.δεξιό_παιδί;
           τέλος:= true
         end
      until τέλος
  end
end
```

4.5 Δυαδικά δένδρα με κλωστές (threaded binary trees)

Πρόταση. Κάθε δυαδικό δένδρο με n κόμβους έχει (n+1) συνδέσμους

που είναι ίσοι με nil.

Απόδειξη:
Οι σύνδεσμοι που είναι nil είναι:

$$N = 2n_0 + n_1 \text{ (γιατί;)}$$

Ισχύουν επίσης οι σχέσεις $n_1+2n_2=n-1$ και $n_0=n_2+1$ (πρόταση 4). Συνεπώς:

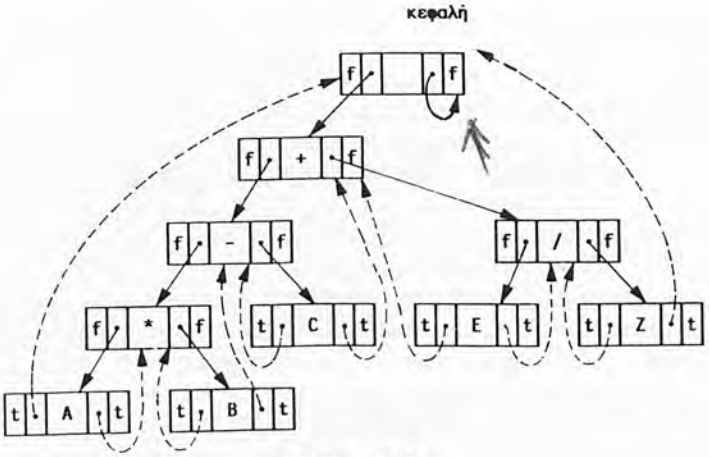
$$N = 2n_0+n_1 = 2(n_2+1)+n_1 = 2n_2+n_1+2 = n-1+2 = n+1.$$

Οπως επίσης γνωρίζουμε, ένα δυαδικό δένδρο με n κόμβους έχει (n-1) κλαδιά, δηλαδή σ'ένα δυαδικό δένδρο με n κόμβους υπάρχουν (n-1) σύνδεσμοι διάφοροι του nil. Επομένως σε κάθε δυαδικό δένδρο οι σύνδεσμοι που είναι nil είναι περισσότεροι από αυτούς που δεν είναι nil. Για να εκμεταλλευτούμε τους (n+1) συνδέσμους που είναι ίσοι με nil, τους κάνουμε να δείχνουν σε άλλους κόμβους, κατάλληλα επιλεγμένους, έτσι ώστε να διευκολύνουν τη λύση ορισμένων προβλημάτων. Τα προβλήματα αυτά περιλαμβάνουν την προσπέλαση των στοιχείων ενός δένδρου, την εύρεση του επόμενου ή προηγούμενου κόμβου σύμφωνα με μια διάταξη κ.ά. Ενας σύνδεσμος που ήταν nil και τον κάναμε να δείχνει σ'έναν άλλο κόμβο, ονομάζεται **κλωστή (thread)** και το δένδρο, στη περίπτωση αυτή, ονομάζεται **δυαδικό δένδρο με κλωστές (threaded binary tree)**.

Εστω ότι θέλουμε να υπολογίσουμε τον ενδοδιατεταγμένο επόμενο ενός κόμβου, δηλαδή τον επόμενο ενός κόμβου όταν κάνουμε ενδοδιατεταγμένη διάσχιση. Τότε, επειδή η σειρά με την οποία επισκεπτόμαστε τους κόμβους του δένδρου του σχήματος 4.9, είναι η A,*,B,-,C,+,E,/,Z, οι κλωστές θα δείχνουν όπως φαίνεται στο σχήμα 4.11 με τις διακεκομμένες συνδέσεις. Κάθε κόμβος του δένδρου πρέπει τώρα να περιέχει δύο επιπλέον μεταβλητές τύπου boolean (αριστερή_κλωστή, δεξιά_κλωστή), οι οποίες δηλώνουν αν ο αντίστοιχος σύνδεσμος είναι κλωστή ή όχι.
Ο ορισμός του τύπου και η δήλωση ενός δένδρου στην περίπτωση αυτή θα είναι:

```
type
  τύπος_στοιχείου = char; {έστω}
  δείκτης_δένδρου = ^κόμβος_δένδρου;
  κόμβος_δένδρου = record
                   αριστερή_κλωστή, δεξιά_κλωστή: boolean;
                   αριστερό_παιδί, δεξιό_παιδί: δείκτης_δένδρου;
                   περιεχόμενο: τύπος_στοιχείου
                   end;
var t: δείκτης_δένδρου
```

Επειδή ο κόμβος με περιεχόμενο A δεν έχει ενδοδιατεταγμένο προηγούμενο κι επειδή ο κόμβος με περιεχόμενο Z δεν έχει ενδοδιατεταγμένο επόμενο, χρησιμοποιούμε, για ευκολία, ένα



Σχήμα 4.11

βοηθητικό κόμβο (κεφαλή), στον οποίο δείχνουν αντίστοιχα αυτοί οι κόμβοι. Όταν το δένδρο είναι nil, τότε ο κόμβος "κεφαλή" θα είναι όπως φαίνεται στο παρακάτω σχήμα.

Αν σ'ένα δένδρο έχουμε πραγματοποιήσει τους συνδέσμους κλωστές, τότε είναι εύκολο να γράψουμε μια συνάρτηση η οποία, δοθέντος ενός δείκτη που δείχνει σ'έναν κόμβο του δένδρου να μας δίνει τον επόμενο κόμβο, που θα επισκεφτούμε κατά την ενδοδιατεταγμένη διάσχιση.

```
function ενδοδιατεταγμένος_επόμενος (t: δείκτης_δένδρου):
                                                δείκτης_δένδρου;
var p: δείκτης_δένδρου;
begin
  if (t^.δεξιά_κλωστή) then
  ενδοδιατεταγμένος_επόμενος:= t^.δεξιό_παιδί
```

```
  else
      begin
        p:= t^.δεξιό_παιδί;
        while (not p^.αριστερή_κλωστή) do
        p:= p^.αριστερό_παιδί;
        ενδοδιατεταγμένος_επόμενος:= p
      end
end {ενδοδιατεταγμένος_επόμενος}
```

Από την παραπάνω συνάρτηση προκύπτει πως ο ενδοδιατεταγμένος επόμενος ενός κόμβου Χ βρίσκεται ακολουθώντας ένα μονοπάτι από συνδέσμους αριστερό_παιδί. Το μονοπάτι αυτό ξεκινάει από τον κόμβο δεξιό_παιδί του Χ και τελειώνει όταν φτάσουμε σ'ένα κόμβο με αριστερή_κλωστή ίση με true. Με τη βοήθεια της παραπάνω συνάρτησης μπορούμε τώρα να κάνουμε μια ενδοδιατεταγμένη διάσχιση ενός δένδρου, χωρίς αναδρομή ή τη χρήση στοίβας.

Ασκηση

Για το δένδρο του σχήματος 4.9 σημειώστε τις κλωστές όταν κάνουμε προδιατεταγμένη διάσχιση. Γράψτε μια συνάρτηση, "επόμενος" η οποία, όταν δίνεται ένας δείκτης που δείχνει σ'έναν κόμβο του δένδρου, να επιστρέφει τον επόμενο κόμβο, με την προδιατεταγμένη έννοια του επόμενου. Γράψτε μια διαδικασία, που να κάνει διάσχιση του δένδρου με την προδιατεταγμένη μέθοδο χρησιμοποιώντας τη συνάρτηση, "επόμενος", και να τυπώνει το περιεχόμενο του κάθε κόμβου με τη σειρά που επισκέπτεται.

Λύση

```
function επόμενος (t: δείκτης_δένδρου): δείκτης_δένδρου;
begin
    if (not t^.αριστερή_κλωστή) then
      επόμενος:= t^.αριστερό_παιδί
    else
      επόμενος:= t^.δεξιό_παιδί
end {επόμενος};

procedure διάσχισε (t: δείκτης_δένδρου);
var p: δείκτης_δένδρου;
begin
  if (t^. αριστερό_παιδί <> t) then
  begin
  p:= t^.αριστερό_παιδί;
  writeln (p^.περιεχόμενο);
  while (p^.δεξιό_παιδί <> t) do
    begin
```

```
      p:= επόμενος (p);
      writeln (p^.περιεχόμενο)
    end
  end
end {διάσχισε}
```

4.6 Αλλες πράξεις επί των δένδρων

Στη συνέχεια θα δώσουμε υπό μορφή διαδικασιών τις πράξεις "διαγραφή", "αντιγραφή", "εκτύπωση" και "υπολογισμού των χαρακτηριστικών" (ύψος, πλήθος κόμβων του δένδρου, μήκος εσωτερικού μονοπατιού) ενός δένδρου.

α. Διαγραφή δένδρου

Η διαγραφή ενός δένδρου σημαίνει την επιστροφή στο σωρό της μνήμης όλων των κόμβων του δένδρου, με τη χρήση της διαδικασίας dispose. Για το σκοπό αυτό θα πρέπει να χρησιμοποιήσουμε μεταδιατεταγμένη διάσχιση, καθόσον ένας κόμβος μπορεί να διαγραφεί μόνο όταν έχουμε επισκεφθεί το αριστερό και δεξιό υπόδενδρό του.

```
procedure  διάγραψε (var t: δείκτης_δένδρου);
begin
  if t <> nil then
    begin
      διάγραψε (t^.αριστερό_παιδί);
      διάγραψε (t^.δεξιό_παιδί);
      dispose (t)
    end
end {διάγραψε}
```

β. Αντιγραφή δένδρου

Η αντιγραφή ενός δένδρου t_1 σημαίνει τη δημιουργία ενός δένδρου t_2 όμοιου με το t_1. Για το σκοπό αυτό θα χρησιμοποιήσουμε μια προδιατεταγμένη μέθοδο που περιγράφεται από τα βήματα:

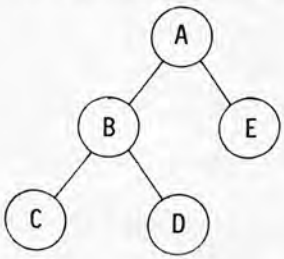
1. Δημιουργία νέου κόμβου (ρίζας),
2. Αντιγραφή αριστερού υποδένδρου και
3. Αντιγραφή δεξιού υποδένδρου.

```
function αντίγραψε (t1: δείκτης_δένδρου): δείκτης_δένδρου;
var t2: δείκτης_δένδρου;
begin
```

```
  if t1 = nil then t2:= nil
  else
  begin
    new (t2);
    t2^.περιεχόμενο:= t1^.περιεχόμενο;
    t2^.αριστερό_παιδί:= αντίγραψε (t1^.αριστερό_παιδί);
    t2^.δεξιό_παιδί:= αντίγραψε (t1^.δεξιό_παιδί)
  end;
  αντίγραψε:= t2
end {αντίγραψε}
```

γ. Εκτύπωση δένδρου

Για την εκτύπωση ενός δένδρου μπορούμε να χρησιμοποιήσουμε οποιαδήποτε μέθοδο διάσχισης. Μια εύκολη περίπτωση είναι η εκτύπωση ενός δένδρου με ενδοδιατεταγμένη διάσχιση, το αποτέλεσμα της οποίας φαίνεται στο σχήμα 4.12. Το δένδρο εκτυπώνεται πλαγίως χωρίς, φυσικά, τα κλαδιά του.



C
B
D
A
E

Σχήμα 4.12

```
procedure εκτύπωσε (t: δείκτης_δένδρου; h: integer);
var i: integer;
begin
  if t <> nil then
    begin
      εκτύπωσε (t^.αριστερό_παιδί, h+1);
      for i:= 1 to h do
      write ('   ');
      writeln (t^.περιεχόμενο);
      εκτύπωσε (t^.δεξιό_παιδί, h+1)
    end
end {εκτύπωσε}
```

δ. Υπολογισμός χαρακτηριστικών ενός δένδρου

Ο υπολογισμός των χαρακτηριστικών ενός δένδρου περιλαμβάνει

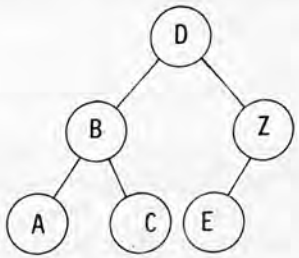
τον υπολογισμό του ύψους του δένδρου, του μεγέθους του (πλήθους των κόμβων του) και του μέσου μήκους των εσωτερικών μονοπατιών του.

```
procedure χαρακτηριστικά (t: δεικτης_δένδρου; var ύψος, μέγεθος:
                          integer; var μέσο_μήκος_μονοπατιού: real);
var μήκος_μονοπατιού: integer;  h: integer;

procedure inorder (t: δεικτης_δένδρου; h: integer);
begin
    if t <> nil then
    begin
      inorder (t^.αριστερό_παιδί, h + 1);
      μέγεθος:= μέγεθος + 1;
      μήκος_μονοπατιού:= μήκος_μονοπατιού + h;
      if ύψος < h + 1 then  ύψος:= h + 1;
      inorder (t^.δεξιό_παιδί, h + 1)
    end
end {inorder};

begin
  ύψος:= 0; μέγεθος:= 0; μήκος_μονοπατιού:= 0;
  if t <> nil then
  begin
    inorder (t,0);
    μέσο_μήκος_μονοπατιού:= μήκος_μονοπατιού / μέγεθος
  end
end {χαρακτηριστικά}
```

4.7 Δυαδικά δένδρα αναζήτησης (Binary search trees)

Μια από τις βασικές πράξεις σ'ένα δένδρο είναι η αναζήτηση (search) ενός κόμβου, που περιέχει ένα συγκεκριμένο στοιχείο (κλειδί). Εστω ότι θέλουμε να βρούμε τον κόμβο, "στόχο" (target), που περιέχει ένα δεδομένο κλειδί. Ξεκινάμε την αναζήτηση από τη ρίζα και ελέγχουμε αν είναι ο κόμβος στόχος. Αν πράγματι η ρίζα περιέχει το κλειδί, τότε τελειώσαμε. Διαφορετικά πρέπει ν'αποφασίσουμε σε ποιό υπόδενδρο να συνεχίσουμε την αναζήτηση, στο αριστερό ή στο δεξιό; Αυτό που χρειαζόμαστε είναι ένα δένδρο στο οποίο, ξεκινώντας από τη ρίζα κι ακολουθώντας ένα μοναδικό απλό μονοπάτι να εντοπίζουμε τον κόμβο-στόχο. Σ'ένα τέτοιο δένδρο θα πρέπει να γνωρίζουμε, για τον κάθε κόμβο του, σε ποιό υπόδενδρο θα πρέπει να κινηθούμε, ώστε να εντοπίσουμε το ζητούμενο στοιχείο. Τα δυαδικά δένδρα αναζήτησης είναι ακριβώς ότι χρειαζόμαστε. Σ'ένα δυαδικό δένδρο αναζήτησης η θέση ενός

κόμβου, σε ποιό δηλαδή υπόδενδρο ανήκει, καθορίζεται από την τιμή του στοιχείου (κλειδιού), που περιέχει ο κόμβος.



Σχήμα 4.13

Στο σχήμα 4.13 δίνεται ένα δένδρο αναζήτησης του οποίου τα στοιχεία είναι χαρακτήρες του λατινικού αλφάβητου. Ενα πλήρες δυαδικό δένδρο με n κόμβους έχει ύψος $\log_2 n$. Αν το δένδρο αυτό είναι και δένδρο αναζήτησης, τότε η ανίχνευση ενός στοιχείου απαιτεί το πολύ $\log_2 n$ συγκρίσεις. Ο αλγόριθμος αναζήτησης περιγράφεται ως εξής:

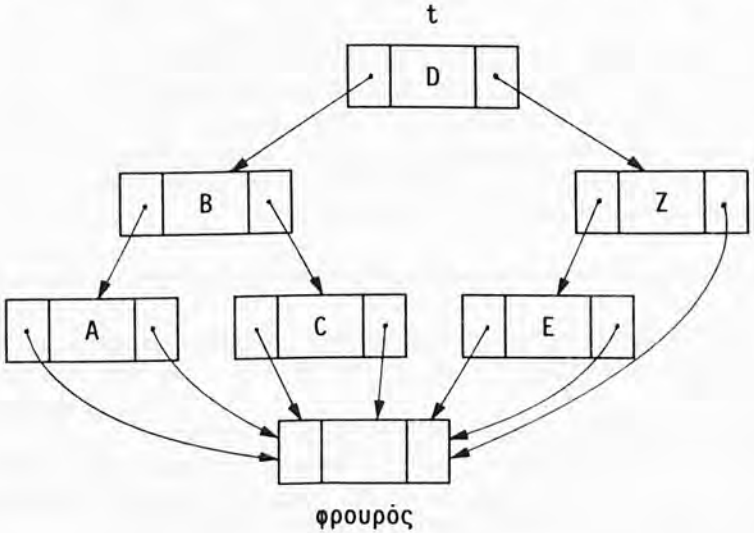
```
while (δένδρο <> nil) and (ο κόμβος στόχος δεν έχει βρεθεί) do
if το στοιχείο_στόχος είναι < από το στοιχείο του τρέχοντος
                                                    κόμβου then
συνέχισε την αναζήτηση στο αριστερό υπόδενδρο
else if το στοιχείο_στόχος > από το στοιχείο του τρέχοντος
                                                    κόμβου
then συνέχισε την αναζήτηση στο δεξιό υπόδενδρο
else
το στοιχείο στόχος έχει βρεθεί
```

Η παρακάτω επαναληπτική συνάρτηση επιστρέφει τον κόμβο, έστω x, που περιέχει το ζητούμενο κλειδί.

```
function βρες_στόχο (t: δεικτης_δένδρου; x: τύπος_στοιχείου):
                                                    δεικτης_δένδρου;
var βρέθηκε: boolean;
begin
  βρέθηκε:= false;
  while (t<>nil) and (not βρέθηκε) do
```

```
  begin
    if t^.περιεχόμενο = x then βρέθηκε:= true
    else
    if t^.περιεχόμενο < x then t:= t^.δεξιό_παιδί
      else   t:= t^.αριστερό_παιδί
  end;
  βρες_στόχο:= t
end {βρες_στόχο}
```

Οταν το κλειδί που ζητάμε δεν υπάρχει στο δένδρο, τότε η διαδικασία επιστρέφει την τιμή nil. Οπως και στις γραμμικές λίστες, η διαδικασία της αναζήτησης απλοποιείται με τη χρήση ενός κόμβου φρουρού. Στον κόμβο φρουρό θα δείχνουν όλοι οι σύνδεσμοι του δένδρου που είναι nil, όπως φαίνεται στο σχήμα 4.14.



Σχήμα 4.14

Ετσι η συνάρτηση βρες_στόχο είναι:

```
function βρές_στόχο (t: δείκτης_δένδρου; x: τύπος_στοιχείου):
                                                  δείκτης_δένδρου;
begin
  φρουρός^.περιεχόμενο:= x;
  while t^.περιεχόμενο <> x do
    if x < t^.περιεχόμενο then t:= t^.αριστερό_παιδί
    else t:= t^.δεξιό_παιδί;
```

```
    βρες_στόχο:= t
end {βρες_στόχο}
```

Στη περίπτωση που το κλειδί δεν υπάρχει στο δένδρο, η συνάρτηση επιστρέφει τον κόμβο φρουρό.

4.8 Εισαγωγή ενός νέου στοιχείου σε δένδρο αναζήτησης

Ο αλγόριθμος της εισαγωγής ενός νέου στοιχείου σ'ένα δένδρο αναζήτησης μπορεί να περιγραφεί από την αναδρομική εντολή:

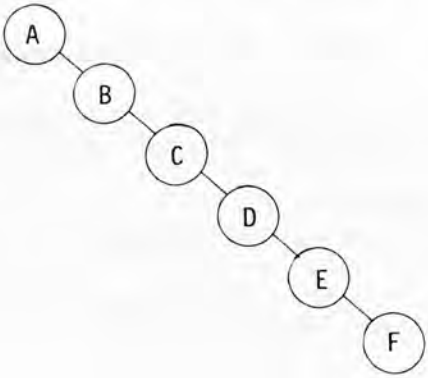
```
if δένδρο = nil then
   δημιούργησε νέο κόμβο με περιεχόμενο το νέο στοιχείο και
   συνδέσεις nil
else κάνε διάσχιση για τον εντοπισμό της θέσης στην οποία θα
     πρέπει να εισαχθεί το νέο στοιχείο
```

Ο αλγόριθμος υλοποιείται σε Pascal με την παρακάτω αναδρομική διαδικασία:

```
procedure εισάγαγε (x: char; var t: δείκτης_δένδρου);
begin
  if (t=nil) then
    begin
      new (t);
      t^.περιεχόμενο:= x;
      t^.αριστερό_παιδί:= nil;
      t^.δεξιό_παιδί:= nil
    end
  else {αναζήτηση για τον εντοπισμό της θέσης του νέου
        στοιχείου}
    if (x < t^.περιεχόμενο) then
      εισάγαγε (x, t^.αριστερό_παιδί)
    else
    if (x > t^.περιεχόμενο) then
      εισάγαγε (x, t^.δεξιό_παιδί)
    else
      writeln ('Το στοιχείο υπάρχει ήδη στο δένδρο')
end {εισάγαγε}
```

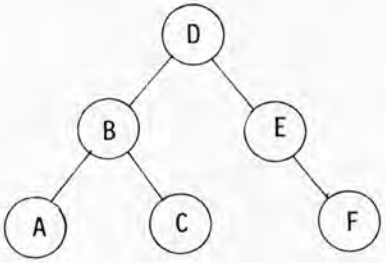
Ο αλγόριθμος αυτός απαιτεί $O(\log_2 n)$ πράξεις, καθόσον είναι παρόμοιος με τον αλγόριθμο της αναζήτησης. Υπάρχει ωστόσο μια βασική διαφορά μεταξύ των δύο αλγορίθμων, η οποία μπορεί να κάνει την απόδοση της εισαγωγής γραμμική, δηλαδή $O(n)$. Αυτό συμβαίνει όταν οι περισσότεροι κόμβοι του δένδρου έχουν ένα παιδί.

Πράγματι, ας υποθέσουμε ότι σ'ένα δένδρο, που αρχικά είναι nil, εισάγονται τα στοιχεία A,B,C,D,E,F. Εφαρμόζοντας τον αλγόριθμο της εισαγωγής προκύπτει το δένδρο του σχήματος 4.15.



Σχήμα 4.15

Το δένδρο αυτό είναι μια ακραία περίπτωση δένδρου εκφυλισμένου σε λίστα. Στο δένδρο αυτό η αναζήτηση και συνεπώς η εισαγωγή, είναι ανάλογη του πλήθους των στοιχείων του δένδρου n. Αν η εισαγωγή των κόμβων γινόταν με τη σειρά D,B,E,C,A,F, τότε θα είχαμε το δένδρο του σχήματος 4.16.



Σχήμα 4.16

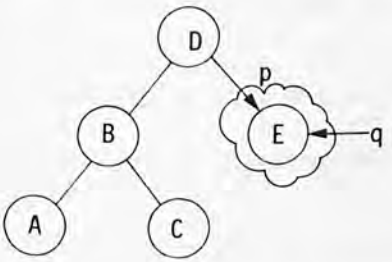
Είναι φανερό πως η σειρά με την οποία εισάγουμε τα νέα στοιχεία σ'ένα δυαδικό δένδρο αναζήτησης επηρεάζει το "σχήμα" του δένδρου που κατασκευάζεται. Έτσι, όσο περισσότερο πλήρες είναι το δένδρο αναζήτησης, τόσο καλύτερη είναι η απόδοση των αλγορίθμων της αναζήτησης και της εισαγωγής . Για την κατασκευή ενός δένδρου αναζήτησης με το μικρότερο ύψος θα ασχοληθούμε στο επόμενο κεφάλαιο. Οπως στην περίπτωση της αναζήτησης, η διαδικασία της εισαγωγής απλοποιείται όταν χρησιμοποιούμε έναν κόμβο φρουρό.

Στην περίπτωση αυτή ένα κενό δένδρο δεν είναι ίσο με nil, αλλά θα περιέχει τον κόμβο φρουρό. Πριν την αναζήτηση του κλειδιού στο δένδρο αναζήτησης θα πρέπει πρώτα να τοποθετήσουμε το κλειδί στον κόμβο φρουρό.

```
procedure εισάγαγε (x:τύπος_στοιχείου; var t: δείκτης_δένδρου);
begin
  φρουρός^.περιεχόμενο:= x;
  if x < t^.περιεχόμενο then εισάγαγε (x, t^.αριστερό_παιδί)
  else
  if x > t^.περιεχόμενο then εισάγαγε (x, t^.δεξιό_παιδί)
  else
  if t <> φρουρός then writeln ('Ο κόμβος υπάρχει στο δένδρο')
  else
    begin
      new (t);
      t^.περιεχόμενο:= x;
      t^.αριστερό_παιδί:= φρουρός;
      t^.δεξιό_παιδί:= φρουρός
    end
end {εισάγαγε}
```

4.9 Διαγραφή ενός κόμβου από δένδρο αναζήτησης

Για τη διαγραφή ενός κόμβου από ένα δενδρο αναζήτησης διακρίνουμε τρεις περιπτώσεις:
(α) Ο κόμβος δεν έχει παιδιά. Τότε απλά τον διαγράφουμε και εκχωρούμε στο δείκτη p, που δείχνει στον προς διαγραφήν κόμβο, την τιμή nil, όπως φαίνεται στο σχήμα 4.17.

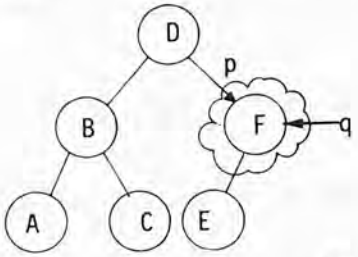



```
q:= p;
p:= nil;
dispose(q)
```

Σχήμα 4.17

(β) Ο κόμβος έχει ένα παιδί, όπως δείχνει το σχήμα 4.18. Τότε εκχωρείται στο δείκτη p η τιμή q^.αριστερό_παιδί ή η τιμή q^.δεξιό_παιδί, ανάλογα με το αν το παιδί του προς διαγραφήν

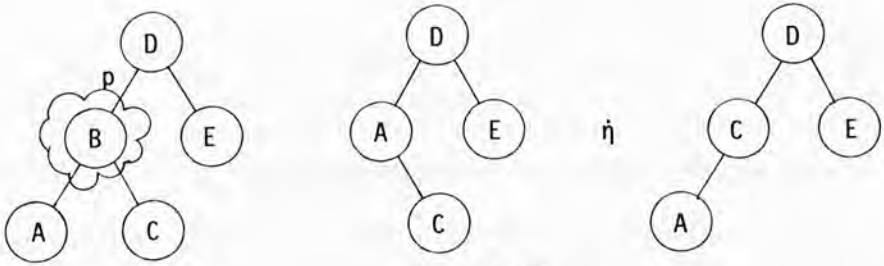
κόμβου είναι αριστερό ή δεξιό, αντίστοιχα, και μετά διαγράφουμε τον κόμβο.




```
q:= p;
p:= q^.αριστερό_παιδί;
dispose(q)
```

Σχήμα 4.18

(γ) Ο κόμβος έχει δύο παιδιά. Τότε αντικαθιστούμε τον κόμβο που θέλουμε να διαγράψουμε με το αριστερότερο κόμβο του δεξιού του υποδένδρου, ή το δεξιότερο κόμβο του αριστερού του υποδένδρου, όπως φαίνεται στο σχήμα 4.19.



Σχήμα 4.19

```
procedure διάγραψε (x: τύπος_στοιχείου; var t: δείκτης_δένδρου);
var p, q, r: δείκτης_δένδρου;
begin
  if t <> nil then
  begin
    if (x < t^.περιεχόμενο) then διάγραψε (x, t^.αριστερό_παιδί)
    else
    if (x > t^.περιεχόμενο) then διάγραψε (x, t^.δεξιό_παιδί)
    else
      begin
        p:= t;
        {εξέτασε αν ο κόμβος έχει ένα παιδί}
        if t^.αριστερό_παιδί = nil then t:= p^.δεξιό_παιδί
        else
        if t^.δεξιό_παιδί = nil then t:= p^.αριστερό_παιδί
        else {ο κόμβος έχει δύο παιδιά}
          begin {αντικατάστησε τον κόμβο με το δεξιότερο παιδί
```

```
                του αριστερού του υποδένδρου}
            q:= p^.αριστερό_παιδί;   r:= nil;
            while q^.δεξιό_παιδί <> nil do
            begin
              r:= q;
              q:= q^.δεξιό_παιδί
            end;
            p:= q;
            t^.περιεχόμενο:= p^.περιεχόμενο;
            if  r = nil then
              t^.αριστερό_παιδί:= p^.αριστερό_παιδί
              else r^.δεξιό_παιδί:= p^.αριστερό_παιδί
          end;   dispose(p)
      end;
      writeln('Το στοιχείο δεν υπάρχει στο δένδρο')
  end
end {διάγραψε}
```

4.10 Εφαρμογές δένδρων - Κώδικες Huffman

Οπως είναι γνωστό η κωδικοποίηση των χαρακτήρων γίνεται σύμφωνα με κάποιο κώδικα (ASCII, EBCD κ.ά.). Οταν μια ακολουθία από χαρακτήρες μεταφέρεται μέσω μιάς γραμμής επικοινωνίας, θα θέλαμε το μήκος της κωδικοποιημένης ακολουθίας να είναι όσο το δυνατόν μικρότερο, έτσι ώστε να μεγιστοποιηθεί η **ρυθμαπόδοση (throughput)** της μετάδοσης των χαρακτήρων. Επιπλέον μιά τέτοια ακολουθία θα πρέπει να μπορεί σχετικά εύκολα να αποκωδικοποιηθεί στο άλλο άκρο της γραμμής.

Αν χρησιμοποιήσουμε έναν κώδικα σταθερού μήκους, τότε για μια ακολουθία n χαρακτήρων απαιτούνται τουλάχιστον $\log_2 n$ δυαδικά ψηφία. Αν για παράδειγμα έχουμε να κωδικοποιήσουμε τους χαρακτήρες a, b, c, d, e, τότε χρειαζόμαστε τουλάχιστον 3 bits για τον κάθε χαρακτήρα ως εξής:

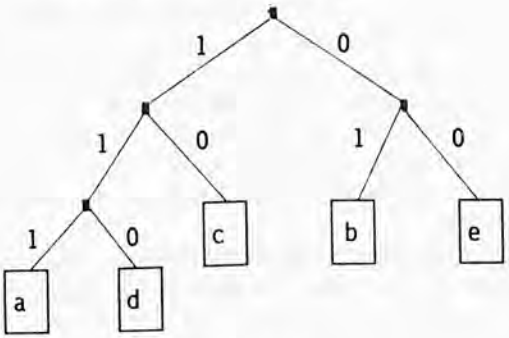
a	000
b	001
c	010
d	011
e	111

Στην περίπτωση αυτή το μέσο μήκος της ακολουθίας των bits ανά χαρακτήρα είναι 3. Αν όμως γνωρίζουμε τις πιθανότητες εμφάνισης των χαρακτήρων, θα μπορούσαμε να τους κωδικοποιήσουμε μ'έναν κώδικα μεταβλητού μήκους, έτσι ώστε οι χαρακτήρες με μεγάλη πιθανότητα εμφάνισης να κωδικοποιούνται με λιγότερα bits. Σ'αυτή τη περίπτωση θα πρέπει όμως να σιγουρευτούμε ότι δεν υπάρχει κώδικας που να είναι πρόθεμα ενός άλλου κώδικα. Αν, για

παράδειγμα, οι κώδικες για τους χαρακτήρες a, b, c, είναι 01, 010 και 101, αντίστοιχα, τότε το μήνυμα 01010101 μπορεί να μεταφραστεί σαν aaaa ή abc ή bca. Ενας κώδικας μεταβλητού μήκους θα μπορούσε να είναι ο:

a	111
b	01
c	10
d	110
e	00

Αν οι πιθανότητες εμφάνισης των χαρακτήρων a, b, c, d και e είναι 0.3, 0.05, 0.3, 0.25 και 0.1, αντίστοιχα, τότε το ερώτημα είναι κατά πόσο ο κώδικας αυτός είναι ο βέλτιστος. Την απάντηση στο ερώτημα αυτό δίνει ο αλγόριθμος του Huffman που περιγράφεται παρακάτω. Ας δούμε όμως πρώτα πως μπορούμε να κατασκευάσουμε τους κώδικες των χαρακτήρων, έτσι ώστε να μην υπάρχει κάποιος κώδικας ως πρόθεμα ενός άλλου. Ας υποθέσουμε ότι οι εξωτερικοί κόμβοι ενός δένδρου αντιστοιχούν στους χαρακτήρες a, b, c, d, και e, όπως δείχνει το σχήμα 4.20:



Σχήμα 4.20

Για κάθε κόμβο του δένδρου σημειώνουμε το σύνδεσμο του αριστερού του υποδένδρου με 1 και του δεξιού υποδένδρου με 0. Τότε ο κώδικας για έναν εξωτερικό κόμβο θα αντιστοιχεί σε μια ακολουθία από ψηφία 0 και 1 πάνω στο μονοπάτι από την ρίζα του δένδρου στον εξωτερικό κόμβο. Για το λόγο αυτό δεν μπορεί να υπάρχει ένας κώδικας, που να είναι συγχρόνως πρόθεμα κάποιου άλλου. Ο αλγόριθμος του Huffman λύνει το πρόβλημα της κατασκευής του βέλτιστου κώδικα το οποίο περιγράφεται ως εξής:

Πρόβλημα: Δοθέντων m βαρών w_1, w_2,..., w_m να κατασκευαστεί ένα δένδρο με τα βάρη αυτά ως εξωτερικούς κόμβους, έτσι ώστε το

εξωτερικό μήκος μονοπατιού για το δένδρο να είναι ελάχιστο, δηλαδή:

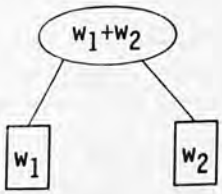
$$\sum_i w_i l_i = \min$$

όπου l_i είναι το μήκος του μονοπατιού του κόμβου i.

Αλγόριθμος Huffman

Ας υποθέσουμε πως η ακολουθία w_i είναι ταξινομημένη έτσι ώστε $w_1 \leq w_2 \leq, \ldots, \leq w_m$. Τότε το δένδρο κατασκευάζεται ως εξής:

1. Εκλέγουμε τα δύο μικρότερα βάρη w_1 και w_2.
2. Αντικαθιστούμε τα w_1, w_2 με το w_1+w_2 στην κατάλληλη θέση, έτσι ώστε η νέα ακολουθία των βαρών να είναι ταξινομημένη και κατασκευάζουμε τον κόμβο:



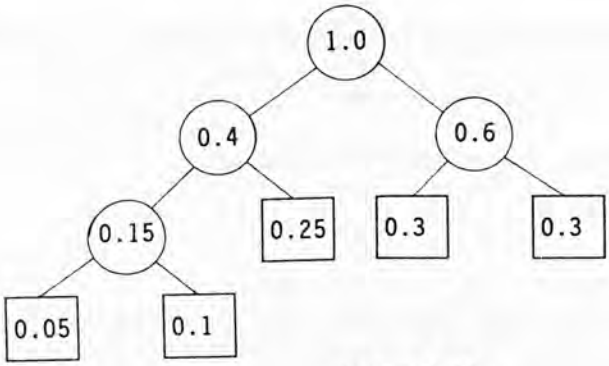
3. Λύνουμε μετά το πρόβλημα για τα (m - 1) βάρη και η διαδικασία επαναλαμβάνεται, μέχρις ότου η ακολουθία των βαρών καταλήξει σ'ένα βάρος $w = \sum_i wi$.

Ο αλγόριθμος αυτός αποδεικνύεται ότι κατασκευάζει το δένδρο που ελαχιστοποιεί το $\Sigma w_i l_i$. Για το παράδειγμά μας, το δένδρο που κατασκευάζεται φαίνεται στο σχήμα 4.21. Συνεπώς ο βέλτιστος κώδικας είναι ο:

a	01
b	111
c	00
d	10
e	110

Ο μέσος όρος του μήκους του κώδικα είναι μόνο:
3 (0.05 + 0.1) + 2(0.25 + 0.3 + 0.3) = 2.15

Για την υλοποίηση του αλγόριθμου Huffman θα χρησιμοποιήσουμε



Σχήμα 4.21

μια λίστα οι κόμβοι της οποίας έχουν περιεχόμενα τύπου "δείκτης_δένδρου".

αριστερό_παιδί	στοιχείο	δεξιό_παιδί	σύνδεσμος

Αρχικά το στοιχείο του κάθε κόμβου της λίστας είναι ίσο με ένα από τα δοθέντα βάρη και οι σύνδεσμοι "αριστερό_παιδί" και "δεξιό_παιδί" είναι nil, δηλαδή οι αρχικοί κόμβοι της λίστας είναι οι εξωτερικοί κόμβοι του δένδρου Huffman . Ο ορισμός του τύπου και η δήλωση της λίστας αυτής σε Pascal είναι:

```
type
  δείκτης_δένδρου = ^ κόμβος_δένδρου;
  κόμβος_δένδρου  = record
                      περιεχόμενο: real;
                      αριστερό_παιδί, δεξιό_παιδί: δείκτης_δένδρου
                    end;
  δείκτης_λίστας =^ κόμβος_λίστας;
  κόμβος_λίστας  = record
                      στοιχείο: δείκτης_δένδρου;
                      σύνδεσμος: δείκτης_λίστας
                   end;
var    λίστα_huffman: δείκτης_λίστας
```

Παρακάτω θα υποθέσουμε ότι η λίστα_huffman είναι ταξινομημένη σε αύξουσα τάξη των βαρών. Η διαδικασία δένδρο_huffman χρησιμοποιεί τη συνάρτηση με επικεφαλίδα:

```
function μικρότερο_βάρος (var h: δείκτης_λίστας): δείκτης_δένδρου
```

που εξάγει από τη λίστα h το πρώτο στοιχείο και τη διαδικασία με επικεφαλίδα:

```
procedure εισάγαγε (t:δεικτης_δένδρου; var h:δεικτης_λίστας)
```

που εισάγει έναν κόμβο t στην λίστα h, έτσι ώστε η νέα λίστα να είναι ταξινομημένη σύμφωνα με τα βάρη.

```
function μικρότερο_βάρος(var h: δεικτης_λίστας): δεικτης_δένδρου;
var q: δεικτης_λίστας;
begin
  μικρότερο_βάρος:= h^. στοιχείο;
  q:= h;
  h:= h^. σύνδεσμος;
  dispose(q)
end {μικρότερο_βάρος};

function    δένδρο_huffman (var  h: δεικτης_λίστας; m: integer):
                                                    δεικτης_δένδρου;

var t: δεικτης_δένδρου;
i: integer;
begin
  for i:=1 to m - 1    do
    begin
      new (t);
      t^.αριστερό_παιδί:= μικρότερο_βάρος (h);
      t^.δεξιό_παιδί:= μικρότερο_βάρος (h);
      t^.περιεχόμενο:= t^.αριστερό_παιδί^.περιεχόμενο +
                                        t^.δεξιό_παιδί^.περιεχόμενο;

      εισάγαγε (t, h)
    end;
    δένδρο_huffman:= μικρότερο_βάρος (h)
end {δένδρο_huffman}
```

Ανακεφαλαίωση

Στο κεφάλαιο αυτό εισαγάγαμε τη βασική ορολογία και τις πράξεις επί των δυαδικών δένδρων. Εμφαση δόθηκε στα δυαδικά δένδρα αναζήτησης. Παρακάτω συνοψίζουμε τις πολυπλοκότητες των πράξεων στα δυαδικά δένδρα αναζήτησης:
αναζήτηση στοιχείου με το κλειδί: $O(\log_2 n)$
Διάσχιση: $O(n)$
Εισαγωγή: $O(\log_2 n)$

Διαγραφή: $O(\log_2 n)$
Αναζήτηση στοιχείου με την θέση του: $O(n)$.

Οπως είδαμε οι πολυπλοκότητες αυτές εξαρτώνται από το ύψος του δένδρου. Το ύψος αυτό επηρεάζεται από τη σειρά με την οποία γίνονται οι εισαγωγές. Για το λόγο αυτό οι πολυπλοκότητες των πράξεων αναζήτησης στοιχείου με το κλειδί, εισαγωγής και διαγραφής στοιχείου στη χειρότερη περίπτωση μπορεί να είναι $O(n)$. Σ'έναν ταξινομημένο πίνακα ο αλγόριθμος της αναζήτησης ενός στοιχείου, με βάση την τιμή του ή τη θέση του, έχουν πολυπλοκότητα $O(\log_2 n)$ και $O(1)$ αντίστοιχα. Ετσι όταν για κάποιο πρόβλημα έχουμε να κάνουμε αναζήτηση πολύ συχνά και εισαγωγή ή διαγραφή σπάνια, τότε θα πρέπει να χρησιμοποιήσουμε πίνακα και όχι δυαδικό δένδρο. Αντίθετα όταν κάνουμε πολλές εισαγωγές/διαγραφές, τότε η υλοποίηση με δυαδικό δένδρο αναζήτησης είναι καλύτερη.Η αναζήτηση ενός στοιχείου με βάση την θέση του σ' ένα δυαδικό δένδρο αναζήτησης, για παράδειγμα αν είναι το k-ιοστό στοιχείο, σημαίνει ενδοδιατεταγμένη διάσχιση, μέχρις ότου φτάσουμε στον k κόμβο του. Η πολυπλοκότητα αυτή μπορεί να βελτιωθεί και να γίνει $O(\log_2 n)$, όταν σε κάθε κόμβο του δένδρου αποθηκεύσουμε το πλήθος των κόμβων του αριστερού του υποδένδρου. Ενας από τους λόγους που τα δυαδικά δένδρα αναζήτησης αποτελούν μια πολύ εύχρηστη και σημαντική δομή, είναι ότι αυτά κάνουν ευκολότερη την υλοποίηση των πράξεων εισαγωγής και διαγραφής ενός στοιχείου.

Ασκήσεις 4.1

1. Δίνεται ένα δυαδικό δένδρο αναζήτησης t. Γράψτε μια συνάρτηση PROGONOI η οποία, για κάποιο δεδομένο κόμβο P του δένδρου, να σχηματίζει τη λίστα όλων των προγόνων του.

2. Γράψε μια διαδικασία που να τυπώνει το αριστερό υπόδενδρο ενός δοθέντος δένδρου t.

3. Γράψε μια διαδικασία ONELEVEL, που κάνει διάσχιση ενός δυαδικού δένδρου κατά την έννοια "ρίζα-αριστερά-δεξιά" και τυπώνει τις τιμές των κλειδιών που βρίσκονται στο ίδιο επίπεδο, GIVEN_LEVEL. Η υλοποίηση να γίνει με δείκτες. Το GIVEN_LEVEL θεωρείται ότι δίνεται καθολικά (globaly). Δώστε την εντολή κλήσης αυτής της διαδικασίας.

4. Γράψτε διαδικασίες σε Pascal που τοποθετούν τις κλωστές ενός δυαδικού δένδρου με α) την ενδοδιατεταγμένη και β) την προδιατεταγμένη έννοια του επόμενου.

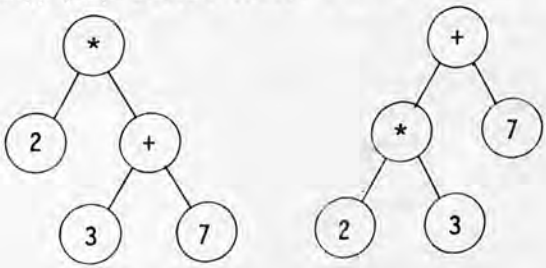
5. Γράψτε μια διαδικασία που να τυπώνει το περιεχόμενο των κόμβων ενός δυαδικού δένδρου κατά επίπεδο από αριστερά προς τα δεξιά.

6. Γράψτε μια διαδικασία που μετρά τα φύλλα ενός δυαδικού δένδρου. Υπολογίστε την πολυπλοκότητα του αλγόριθμου σας ως προς τον χρόνο.

7. (α) Γράψτε μια διαδικασία "FYLLA" που κάνει διάσχιση ενός δυαδικού δένδρου κατά την έννοια "ρίζα-αριστερά-δεξιά" και τυπώνει τις τιμές των κλειδιών που βρίσκονται στα φύλλα του δένδρου.
   (β) Γράψτε μια εντολή κλήσης αυτής της διαδικασίας.

8. Γράψτε μια διαδικασία που να επιστρέφει τον πατέρα του κόμβου με περιεχόμενα x σ'ένα δυαδικό δένδρο.

9. Ενα δένδρο στο οποίο το περιεχόμενο όλων των ενδιάμεσων κόμβων είναι τελεστής πράξεως και το περιεχόμενο των τερματικών κόμβων είναι ένας αριθμός λέγεται δένδρο παράστασης (expression tree). Για παράδειγμα τα:



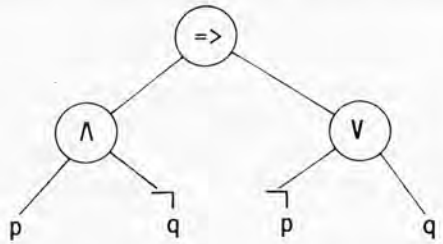
είναι δύο δένδρα παράστασης. Αν υποθέσουμε ότι υπάρχουν μόνο δύο τελεστές + και * και ότι όλες οι τιμές είναι ακέραιοι, τότε μπορούμε να χρησιμοποιήσουμε μια μέθοδο διάσχισης για τον υπολογισμό της τιμής του δένδρου παράστασης. Για παράδειγμα, η τιμή του δένδρου παράστασης (α) είναι 20 και του (β) είναι 13. Γράψτε μια αναδρομική συνάρτηση που να επιστρέφει την τιμή ενός δένδρου παράστασης. Η παράμετρος της συνάρτησης σας θα πρέπει να είναι το δένδρο. Θα πρέπει να ορίσετε όλους τους τύπους που θα χρησιμοποιήσετε, παίρνοντας κατάλληλες αποφάσεις για τον τρόπο που οι τελεστές και οι τιμές αποθηκεύονται στο δένδρο.

Υπόδειξη: Αν η ρίζα είναι αριθμός, τότε η τιμή του δένδρου βρέθηκε, διαφορετικά αν η ρίζα είναι '+', τότε η τιμή του δένδρου είναι:

(τιμή αριστερού υποδένδρου) + (τιμή δεξιού υποδένδρου)
διαφορετικά αν η ρίζα είναι '*', τότε η τιμή του δένδρου είναι:
(τιμή αριστερού υποδένδρου) * (τιμή δεξιού υποδένδρου).

10. Δίνονται δύο δένδρα t και s. Γράψτε μια συνάρτηση που να επιστρέφει την τιμή true, αν τα περιεχόμενα των αντίστοιχων κόμβων των δύο δένδρων είναι ίσα. Αν τα δένδρα έχουν διαφορετικό σχήμα ή δεν συμπίπτουν τα περιεχόμενα κάποιων κόμβων τους, τότε η συνάρτηση θα επιστρέφει την τιμή false.

11. Γράψτε μια διαδικασία που να υπολογίζει την τιμή μιας λογικής παράστασης από ένα λογικό δένδρο παράστασης όπως αυτό του σχήματος:



Οι μεταβλητές έχουν τιμές true ή false και οι πράξεις είναι:

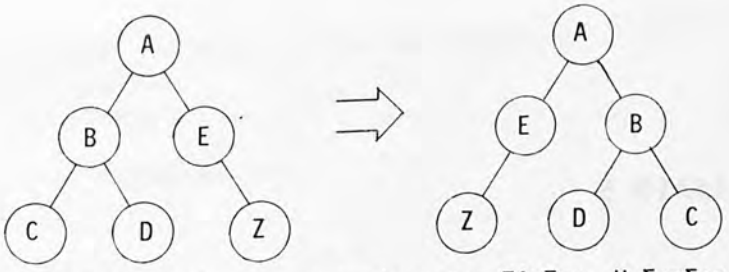
¬	(NOT)	(μονομερής πράξη)
Λ	(AND)	(διμερής πράξη)
V	(OR)	(διμερής πράξη)
=>		(συνεπάγεται)

Οι ορισμοί αυτοί των πράξεων δίνονται από τους παρακάτω πίνακες αληθείας:

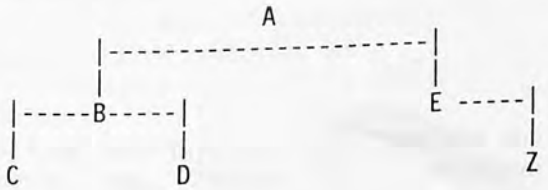
p	q	¬p	p Λ q	p V q	p => q
F	F	T	F	F	T
F	T	T	F	T	T
T	F	F	F	T	F
T	T	F	T	T	T

12. Γράψτε μια διαδικασία που να διατρέχει ένα δυαδικό δένδρο και να γράφει τα περιεχόμενα των κόμβων που έχουν ένα μόνο παιδί.

13. Γράψτε μια διαδικασία SWAP που να ανταλλάζει αμοιβαία τα κλειδιά των κόμβων ενός δυαδικού δένδρου, όπως φαίνεται στο παρακάτω σχήμα:



14. Γράψτε μια διαδικασία που να τυπώνει ένα δένδρο. Η διαδικασία πρέπει να τυπώνει και τα κλαδιά του δένδρου όπως φαίνεται στο παρακάτω σχήμα.



15. Γράψτε μια αναδρομική διαδικασία "βρες_αδελφό" η οποία, δοθέντος ενός κόμβου p ενός δυαδικού δένδρου, να επιστρέφει τον αδελφό του κόμβο, έστω q. Υποτίθεται ότι ο κόμβος p είναι διαφορετικός από τη ρίζα του δένδρου. Αν δεν υπάρχει κόμβος αδελφός, τότε η διαδικασία θα πρέπει να επιστρέφει την τιμή nil.

ΚΕΦΑΛΑΙΟ 5

ΙΣΟΖΥΓΙΣΜΕΝΑ ΔΕΝΔΡΑ

5.1 Γενικά

Οπως είδαμε από τον αλγόριθμο εισαγωγής ενός στοιχείου σ'ένα δυαδικό δένδρο αναζήτησης, το "σχήμα" του δένδρου, που κατασκευάζεται, εξαρτάται απο τη σειρά με την οποία γίνονται οι εισαγωγές των στοιχείων. Είναι φανερό ότι για δεδομένο πλήθος κόμβων n, το δένδρο με το μικρότερο ύψος θα έχει τον καλύτερο χρόνο αναζήτησης, $\log_2 n$. Για να κατασκευάσουμε ένα τέτοιο δένδρο, θα πρέπει σε κάθε επίπεδο, εκτός ίσως από το τελευταίο, το δένδρο να είναι πλήρες. Ενα δένδρο θα λέγεται **τέλεια ισοζυγισμένο** ή **ισοσταθμισμένο (perfectly balanced)**, όταν για κάθε κόμβο του το αριστερό και το δεξιό του υπόδενδρο διαφέρουν το πολύ κατά έναν κόμβο.

5.2 Κατασκευή ισοζυγισμένου δένδρου

Εστω ότι θέλουμε να κατασκευάσουμε ένα τέλεια ισοζυγισμένο δένδρο με n κόμβους. Δημιουργούμε τον κόμβο-ρίζα και ισομοιράζουμε τους υπόλοιπους κόμβους στα δύο υπόδενδρά του. Ετσι το κάθε υπόδενδρο θα πάρει k = n div 2 και n-k-1 κόμβους αντίστοιχα. Η διαδικασία αυτή επαναλαμβάνεται αναδρομικά, μέχρις ότου φτάσουμε στην περίπτωση όπου έχουμε εξαντλήσει όλους τους κόμβους. Ο αλγόριθμος αυτός μπορεί να περιγραφεί με τα εξής βήματα:

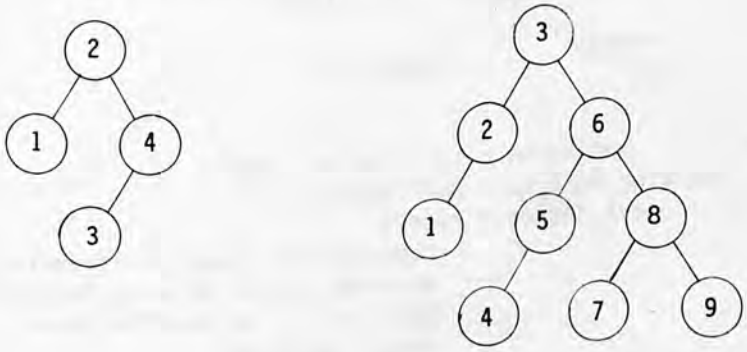
1. Δημιούργησε τη ρίζα.
2. Κατασκεύασε το αριστερό υπόδενδρο με k = n div 2 κόμβους.
3. Κατασκεύασε το δεξιό υπόδενδρο με n-k-1 κόμβους.

Η παρακάτω διαδικασία δημιουργεί ένα τέλεια ισοζυγισμένο δένδρο:

```
function ισοζύγισε (n: integer): δεικτης_δένδρου;
var p: δεικτης_δένδρου;
    x, nl, nr: integer;
begin
  if n = 0 then ισοζύγισε:= nil
  else
    begin
      nl:= n div 2;
      nr:= n-nl-1;
      read (x);
      new (p);
      p^.περιεχόμενο:= x;
      p^.αριστερό_παιδί:= ισοζύγισε (nl);
      p^.δεξιό_παιδί:= ισοζύγισε (nr)
    end; ισοζυγισε:= p
end {ισοζύγισε}
```

5.3 AVL - δένδρα

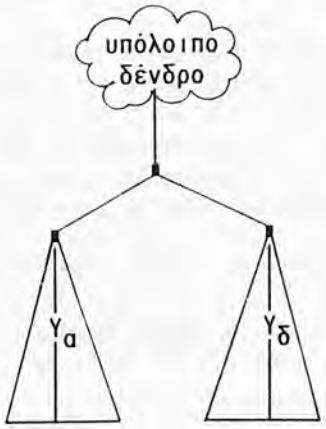
Η πολυπλοκότητα αναζήτησης ενός στοιχείου σ'ένα δυαδικό δένδρο αναζήτησης εξαρτάται από το ύψος του δένδρου μάλλον, παρά από το μέγεθός του. Το ύψος αυτό, όπως αναφέραμε, κυμαίνεται από $\log_2 n$ έως n. Το τελευταίο ισχύει στην περίπτωση που το δένδρο είναι εκφυλισμένο σε λίστα. Ευτυχώς, έχει αποδειχτεί πειραματικά, πως το ύψος ενός τυχαίου δυαδικού δένδρου αναζήτησης, δηλαδή ενός δένδρου που προκύπτει μετά από έναν αριθμό τυχαίων εισαγωγών/διαγραφών, είναι κατά μέσο όρο του ίδιου βαθμού, $O(\log_2 n)$, με το ύψος ενός πλήρους δυαδικού δένδρου. Στη χειρότερη όμως περίπτωση η αναζήτηση σ'ένα τυχαίο δυαδικό δένδρο έχει την ίδια πολυπλοκότητα όπως η ακολουθιακή αναζήτηση (O(n)).



Σχήμα 5.1

Οπωσδήποτε όταν κάνουμε αναζήτηση σ'ένα δένδρο θα θέλαμε το ύψος του να είναι $\log_2 n$. Για το λόγο αυτό θα εξετάσουμε στην παράγραφο αυτή την περίπτωση εισαγωγής ενός στοιχείου σ'ένα δένδρο αναζήτησης, έτσι ώστε το ύψος του δένδρου να παραμένει $O(\log_2 n)$. Για το σκοπό αυτό εισάγουμε εδώ μια ειδική κατηγορία ισοζυγισμένων δένδρων αναζήτησης τα **AVL-δένδρα**. Ενα δένδρο θα ονομάζεται AVL ή 1-ισοζυγισμένο δένδρο, όταν για κάθε κόμβο του, η διαφορά του ύψους μεταξύ των υποδένδρων του είναι το πολύ 1. Στο σχήμα 5.1 δίνονται παραδείγματα AVL-δένδρων. Γενικά όταν Y_α και Y_δ είναι τα ύψη του αριστερού και δεξιού υποδένδρου για έναν κόμβο, τότε $|Y_\alpha - Y_\delta| \leq 1$. Σχηματικά αυτό δίνεται στο σχήμα 5.2.

Για την εισαγωγή ενός στοιχείου σ'ένα AVL δένδρο θα πρέπει να γνωρίζουμε το ύψος του υποδένδρου στο οποίο θα προστεθεί ο νέος κόμβος. Αυτό μπορεί να γίνει εύκολα αν χρησιμοποιήσουμε για τον κάθε κόμβο του δένδρου ένα επιπλέον πεδίο, που θα περιέχει πληροφορίες για τα ύψη των υποδένδρων του κόμβου αυτού. Ετσι ο ορισμός του τύπου και η δήλωση ενός AVL-δένδρου σε Pascal είναι:



Σχήμα 5.2

```
type        ισοζύγιση = (συν, μηδέν, πλην);
       δεικτης_δένδρου = ^avl_κόμβος;
            avl_κόμβος = record
                           περιεχόμενο: τύπος_περιεχομένου;
                           αριστερό_παιδί: δεικτης_δένδρου;
                           δεξιό_παιδί: δεικτης_δένδρου;
                           ύψος: ισοζύγιση
                         end;
    var avl: δεικτης_δένδρου
```

Η σχέση ύψος=συν σημαίνει ότι το δεξιό υπόδενδρο είναι κατά ένα ψηλότερο από το αριστερό. Η σχέση ύψος=πλην σημαίνει ότι το αριστερό υπόδενδρο είναι κατά ένα ψηλότερο. Τέλος όταν ύψος=μηδέν τα δύο υπόδενδρα έχουν το ίδιο ύψος. Αποδεικνύεται ότι σ'ένα AVL δένδρο με n κόμβους το ύψος του δεν ξεπερνά το $1.45\log_2 n$(*).

5.4 Εισαγωγή σε AVL - δένδρο

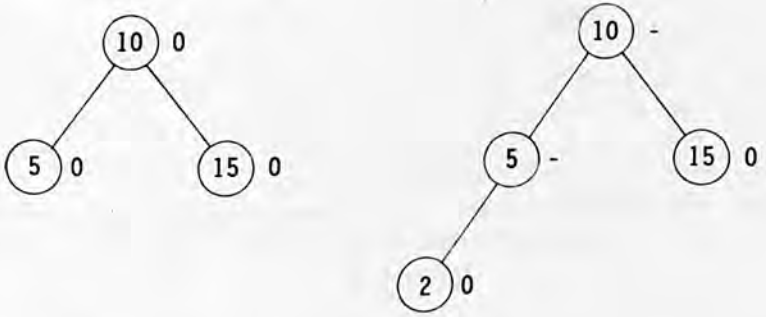
Η εισαγωγή ενός νέου στοιχείου σ'ένα AVL-δένδρο γίνεται σε δύο φάσεις, ως εξής:

α) Το στοιχείο εισάγεται, όπως ακριβώς σ'ένα απλό δένδρο αναζήτησης.
β) Το δένδρο, που προκύπτει, επανα-ζυγίζεται (rebalanced), έτσι ώστε, μετά την εισαγωγή να παραμένει AVL-δένδρο.

Ως κόμβο **"οδηγό" (pivot)** ονομάζουμε τον κόμβο εκείνο, πάνω στο μονοπάτι αναζήτησης, του οποίου το ύψος είναι συν ή πλην και που βρίσκεται πλησιέστερα προς το νέο κόμβο. Για την εισαγωγή θα διακρίνουμε τρεις περιπτώσεις.

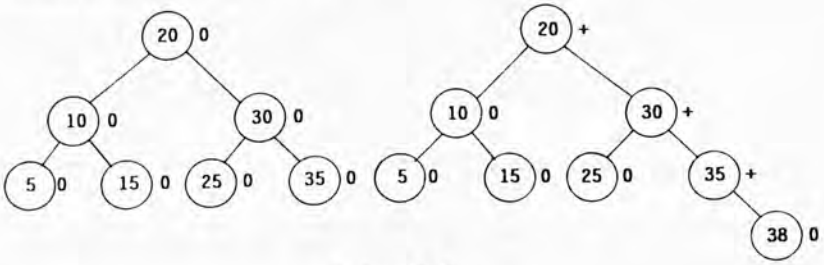
(1) Δεν υπάρχει κόμβος-οδηγός. Αυτό σημαίνει ότι το δένδρο είναι ισοζυγισμένο από τη ρίζα. Στην περίπτωση αυτή, ο νέος κόμβος προστίθεται όπως στην περίπτωση του δένδρου αναζήτησης κανονικά στη θέση του, και το δένδρο αναπροσαρμόζεται, αλλάζοντας τις τιμές της μεταβλητής "ύψος" σε όλους τους κόμβους πάνω στο μονοπάτι αναζήτησης ξεκινώντας από τη ρίζα. Στο σχήμα 5.3 δίνονται παραδείγματα εισαγωγής. Με 0, +, - συμβολίζουμε τις τιμές της μεταβλητής, "ύψος", για κάθε κόμβο.

α) Εισαγωγή του στοιχείου 2:



[Knuth D. E. The art of computer programming, Τόμος 1, Κεφ. 6]

β) Εισαγωγή του στοιχείου 38:

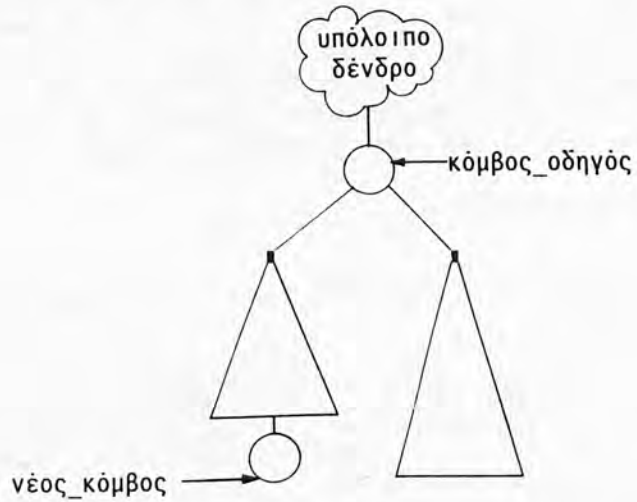


Σχήμα 5.3

Από τα παραπάνω παραδείγματα παρατηρούμε ότι η μεταβλητή "ύψος" για κάθε κόμβο αλλάζει ως εξής:

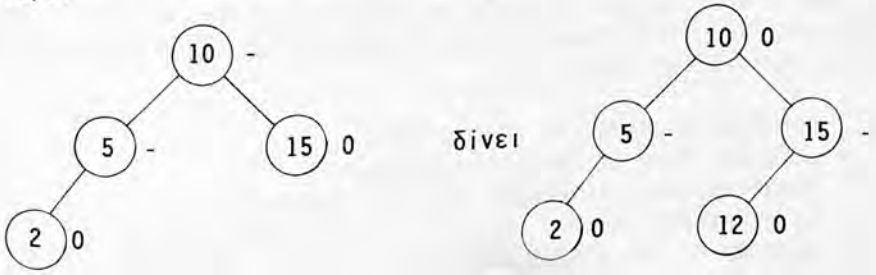
Οταν κινούμαστε στο μονοπάτι αναζήτησης προς τα δεξιά , τότε το ύψος "αυξάνει" δηλαδή το - γίνεται 0 και το 0 γίνεται +. Οταν κινούμαστε στο μονοπάτι αναζήτησης προς τ'αριστερά, το ύψος "μειώνεται", με την ίδια έννοια.

(2) Ο κόμβος-οδηγός υπάρχει και το υπόδενδρό του στο οποίο θα προστεθεί ο νέος κόμβος, έχει μικρότερο (ή ίσο) ύψος όπως φαίνεται στο σχήμα 5.4.

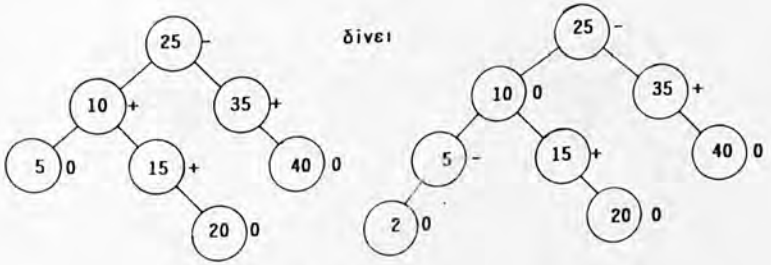


Σχήμα 5.4

Τότε ο νέος κόμβος προστίθεται και πάλι κανονικά, όπως στην περίπτωση ενός απλού δένδρου αναζήτησης, και η αναπροσαρμογή του δένδρου γίνεται πάνω στο μονοπάτι αναζήτησης από τον κόμβο οδηγό μέχρι το νέο κόμβο (βλέπε σχήμα 5.5). Το υπόλοιπο δένδρο δεν επηρεάζεται από την εισαγωγή, καθόσον το ύψος του υπόδενδρου, με ρίζα τον κόμβο οδηγό, δεν αλλάζει μετά την εισαγωγή του νέου στοιχείου. Για παράδειγμα, η εισαγωγή του στοιχείου 12 στο δένδρο:



Ενώ η εισαγωγή του στοιχείου 2 του δένδρου



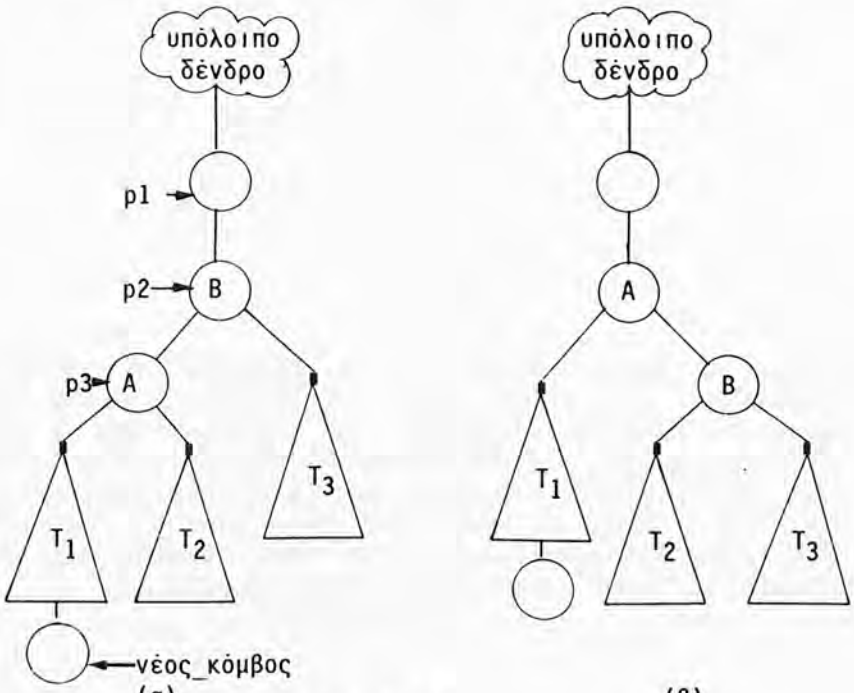
Σχήμα 5.5

(3) Υπάρχει ο κόμβος οδηγός και το υπόδενδρο, στο οποίο θα προστεθεί ο νέος κόμβος, έχει το μεγαλύτερο ύψος. Στην περίπτωση αυτή αν προσθέσουμε το νέο κόμβο όπως σε απλό δένδρο αναζήτησης, τότε το δένδρο που προκύπτει δε θα είναι πλέον AVL. Για το λόγο αυτό, μετά από μια τέτοια εισαγωγή, θα πρέπει ν' αλλάξει η δομή του δένδρου. Στην περίπτωση αυτή διακρίνουμε δύο υποπεριπτώσεις που είναι γνωστές ως απλή (single) και διπλή (double) περιστροφή (rotation).

3α) **Απλή περιστροφή** (βλέπε σχήμα 5.6)

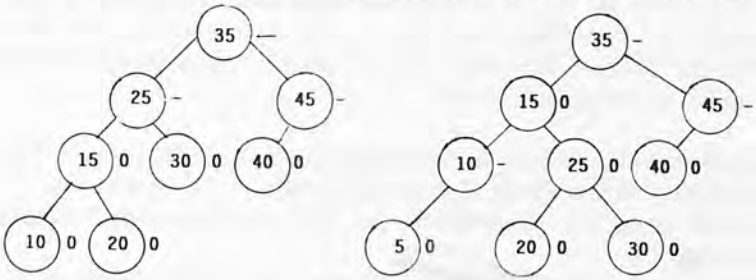
Στην περίπτωση αυτή, ο κόμβος οδηγός είναι αυτός με περιεχόμενο το Β. Τα δένδρα T_1, T_2, T_3 είναι αυθαίρετα, αλλά έχουν το ίδιο ύψος και όλοι οι κόμβοι πάνω στο μονοπάτι αναζήτησης, από τον κόμβο Α μέχρι το νέο κόμβο, είναι ισοζυγισμένοι, έχουν δηλαδή ύψος ίσο με μηδέν. Το υπόλοιπο του δένδρου δεν επηρεάζεται από την εισαγωγή του νέου κόμβου, καθόσον το ύψος του υπόδενδρου με ρίζα τον κόμβο οδηγό δεν αλλάζει, όπως μπορούμε να διαπιστώσουμε από το σχήμα 5.6β. Ο αλγόριθμος της απλής περιστροφής είναι:

```
A1: εντόπισε τη θέση του νέου κόμβου;
    εισάγαγε όπως σε δένδρο αναζήτησης;
    {αναδιοργάνωσε το δένδρο ώστε να παραμένει AVL}
    p1:= πατέρας (κόμβου οδηγού);   p2:= κόμβος_οδηγός;
    p3:= παιδί του κόμβου οδηγού στο μονοπάτι αναζήτησης;
    p1^."παιδί":= p3 {παιδί είναι το κλαδί που βρίσκεται στο
                      μονοπάτι αναζήτησης};
    p3^.δεξιό_παιδί:= p2;
    p2^.αριστερό_παιδί = ρίζα (T2)
```



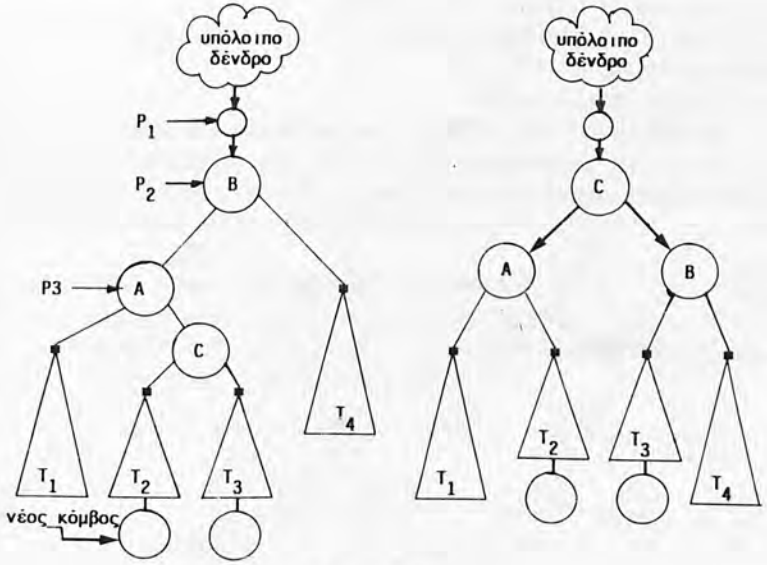
Σχήμα 5.6

Ο αλγόριθμος της απλής περιστροφής θα πρέπει να λαμβάνει υπόψη του και τη συμμετρική περίπτωση του σχήματος 5.6. Στο σχήμα 5.7 δίνεται ένα παράδειγμα εισαγωγής του στοιχείου 5 στο δένδρο:



Σχήμα 5.7

(3β) **Διπλή περιστροφή** (βλέπε σχήμα 5.8)

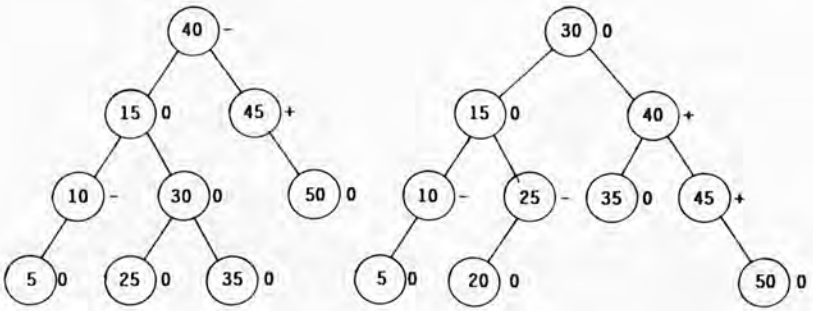


Σχήμα 5.8

Στην περίπτωση αυτή το μονοπάτι αναζήτησης από τον κόμβο με περιεχόμενο το Α, μέχρι το νέο κόμβο αποτελείται από ισοζυγισμένους κόμβους μόνο. Τα δένδρα T_1, T_2, T_3, T_4 είναι αυθαίρετα αλλά τα T_2, T_3 έχουν το ίδιο ύψος το οποίο είναι κατά 1 μικρότερο από το ύψος των T_1, T_4 που έχουν επίσης το ίδιο ύψος. Το υπόλοιπο δένδρο δεν επηρεάζεται από την εισαγωγή. Ο αλγόριθμος της διπλής περιστροφής είναι:

```
εντόπισε τη θέση του νέου κόμβου;
εισάγαγε όπως σε δένδρο αναζήτησης;
{αναδιοργάνωσε το δένδρο ώστε να παραμένει AVL και μετά την
εισαγωγή}
p1:= πατέρας (κόμβου οδηγού);
p2:= κόμβος_οδηγός;
p3:= p2^.αριστερό_παιδί;
p4:= p3^.δεξιό_παιδί;
p1^."παιδί":= p4 {παιδί είναι εκείνο το κλαδί που βρίσκεται
                                        στο μονοπάτι αναζήτησης};
p2^.αριστερό_παιδί:= ρίζα (T3);
p3^.δεξιό_παιδί:= ρίζα (T2);
p4^.αριστερό_παιδί:= p3;
p4^.δεξιό_παιδί:= p2
```

Ο αλγόριθμος της διπλής περιστροφής θα πρέπει να λαμβάνει υπόψη του και τη συμμετρική περίπτωση του σχήματος 5.8. Στο σχήμα 5.9 δίνεται ένα παράδειγμα εισαγωγής του στοιχείου 20 στο δένδρο:



Σχήμα 5.9

Αλγόριθμος εισαγωγής στοιχείου σε AVL-δένδρο

Για την περιγραφή του αλγόριθμου εισαγωγής ενός στοιχείου θα χρησιμοποιήσουμε τρία υποπρογράμματα:

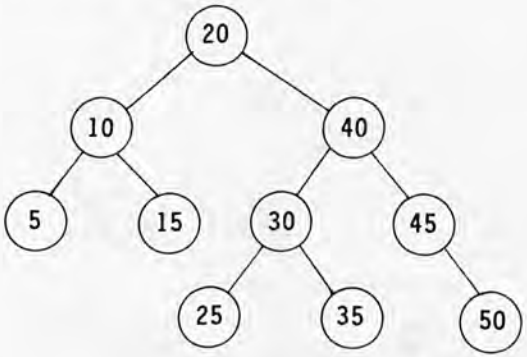
α) **τοποθέτησε_δείκτες:** διαδικασία, που βάζει τους δείκτες P_1, P_2, P_3, P_4,

β) **μικρό:** συνάρτηση τύπου boolean, που ελέγχει αν ο νέος κόμβος θα προστεθεί στο κοντότερο ή στο μακρύτερο υπόδενδρο του κόμβου οδηγού και

γ) **Κάνε αναπροσαρμογή_υψών (p: δείκτης_δένδρου):** διαδικασία που ενημερώνει το πεδίο "ύψος" κάθε κόμβου πάνω στο μονοπάτι αναζήτησης, από τον κόμβο p^ μέχρι το νέο κόμβο, μετά την εισαγωγή.

Αλγόριθμος εισαγωγής:

```
procedure avl_εισαγωγή (x: τύπος_περιεχομένου; var
                                        t: δείκτης_δένδρου);
begin
  εισάγαγε το νέο στοιχείο σαν να ήταν απλό δυαδικό
  δένδρο αναζήτησης;
  τοποθέτησε_δείκτες;
  if p2 = nil then κάνε_αναπροσαρμογή_υψών (ρίζα)
  else
  if μικρό then κάνε_αναπροσαρμογή_υψών (p2)
    else  έλεγξε ποια υπο-περίπτωση εισαγωγής είναι;
    if  (υποπερίπτωση 3.α)  then  εφάρμοσε απλή περιστροφή
      else  εφάρμοσε διπλή περιστροφή
end
```



Σχήμα 5.10

Ασκηση: Εστω ότι τα δεδομένα 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 διαβάζονται από το αρχείο εισόδου. Εφαρμόζοντας τον αλγόριθμο της εισαγωγής, δείξτε ότι κατασκευάζεται το AVL-δένδρο του σχήματος 5.10

5.5 Σωροί (heaps)

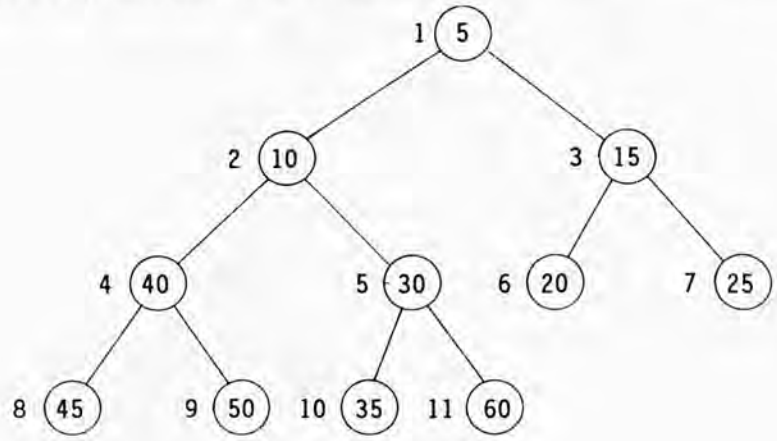
Σωρός είναι μια ακολουθία x[i] στοιχείων για τα οποία ισχύει: x[i] ≤ x[2i] και x[i] ≤ x[2i+1] για όλα τα i. Για παράδειγμα η ακολουθία των ακεραίων:

5, 10, 15, 40, 30, 20, 25, 45, 50, 35, 60

αποτελεί ένα σωρό.

Από τον παραπάνω ορισμό παρατηρούμε ότι ένας σωρός μπορεί να παρασταθεί μ'ένα δυαδικό δένδρο στο οποίο η συνθήκη του ορισμού εφαρμόζεται μεταξύ του πατέρα και των παιδιών του, όπως φαίνεται στο σχήμα 5.11. Ενας σωρός, λοιπόν, μπορεί να θεωρηθεί ως ένα πλήρες δυαδικό δένδρο στο οποίο το περιεχόμενο κάθε κόμβου είναι μικρότερο από τα περιεχόμενα των παιδιών του. Στο τελευταίο επίπεδο του δένδρου-σωρού όλοι οι κόμβοι θα πρέπει να βρίσκονται στο αριστερότερο άκρο του δένδρου.

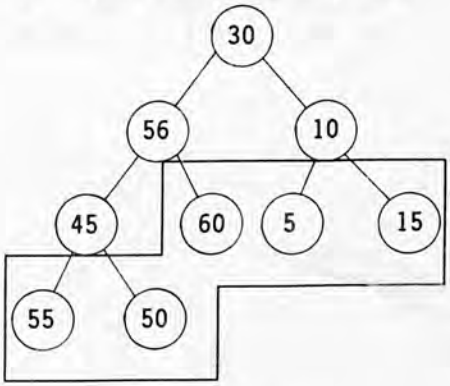
Από τον ορισμό του σωρού παρατηρούμε επίσης ότι οποιοδήποτε μονοπάτι ακολουθήσουμε από τη ρίζα τα στοιχεία των κόμβων θα είναι σε αύξουσα τάξη. Η παρατήρηση αυτή θα μας βοηθήσει στην κατασκευή ενός σωρού. Τέλος, πρέπει να σημειώσουμε πως: Ενα πλήρες δυαδικό δένδρο, στο οποίο το περιεχόμενο κάθε κόμβου, εκτός από τα φύλλα, είναι μεγαλύτερο ή ίσο από τα περιεχόμενα των παιδιών του, αποτελεί επίσης ένα σωρό.



Σχήμα 5.11

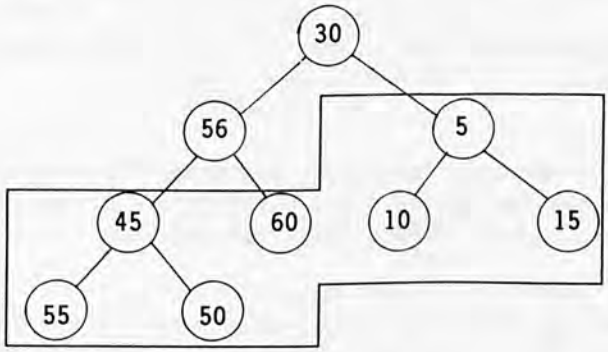
5.6 Κατασκευή σωροῦ

Εστω ότι έχουμε το δένδρο του σχήματος 5.12.



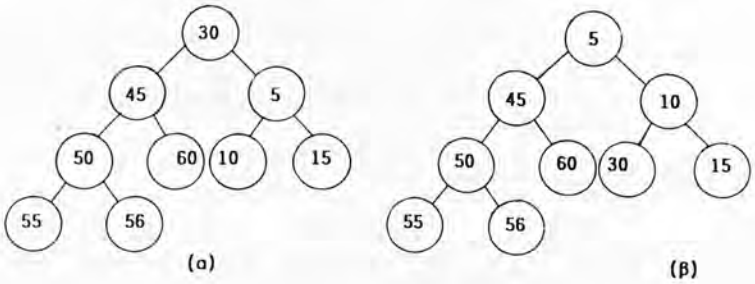
Σχήμα 5.12

Παρατηρούμε ότι οι τερματικοί κόμβοι πληρούν τη συνθήκη του σωροῦ, γιατί δεν έχουν παιδιά. Εστω πως στο σύνολο των κόμβων αυτών {55, 50, 60, 5, 15} θέλουμε να εισαγάγουμε τον κόμβο 45, έτσι ώστε να πληρούται η συνθήκη του σωροῦ. Εξετάζουμε τα παιδιά του (50, 55), γιατί μόνο αυτά μπορούν να επηρεασθούν από την εισαγωγή. Παρατηρούμε ότι και τα δύο παιδιά έχουν κλειδιά μεγαλύτερα του 45 και συνεπώς δεν επηρεάζονται από την εισαγωγή του 45. Μετά συνεχίζουμε με την εισαγωγή του 10. Στην περίπτωση αυτή ένα παιδί έχει περιεχόμενο μικρότερο του 10. Ανταλλάσσουμε το παιδί αυτό με τον πατέρα του και το δένδρο γίνεται όπως δείχνει το σχήμα 5.13.



Σχήμα 5.13

Μετά εισάγουμε τον κόμβο 56. Στην περίπτωση αυτή επηρεάζονται όλα τα στοιχεία, που βρίσκονται σ'ένα μονοπάτι από το 56 μέχρι τα φύλλα του δένδρου. Το μονοπάτι αυτό αποτελείται από τα κλαδιά, που ενώνουν πατέρα με το μικρότερο παιδί του. Έτσι προκύπτει το δένδρο του σχήματος 5.14α. Τέλος μετά την εισαγωγή του 30 έχουμε το δένδρο του σχήματος 5.14β.



Σχήμα 5.14

Παρατηρήσεις

1. Από τον ορισμό του σωρού φαίνεται ότι, για την υλοποίησή του, συμφέρει η στατική μέθοδος που περιγράψαμε στην παράγραφο 4.3.

2. Από την πρόταση 2 της παραγράφου 4.1, σε συνδυασμό με τον αλγόριθμο κατασκευής του σωρού, όπως τον περιγράψαμε, προκύπτει ότι τα (n div 2 + 1) κλειδιά, που αντιστοιχούν στα φύλλα του δένδρου, μπορούν να τοποθετηθούν στον πίνακα, που παριστάνει το σωρό απ'ευθείας. Αρκεί συνεπώς τα υπόλοιπα κλειδιά να τοποθετηθούν στον πίνακα με τέτοιο τρόπο, ώστε να έχουμε ένα σωρό.

Η παρακάτω διαδικασία δημιουργεί ένα σωρό. Υποτίθεται ότι όλα τα κλειδιά βρίσκονται αρχικά σ'ένα μονοδιάστατο πίνακα "heap" μεγέθους ίσου με το μέγιστο πλήθος (max) των κλειδιών.

```
procedure αναπροσάρμοσε_σωρό (υπόλοιπα: integer);
var i, j, temp: integer;
    τέλος: boolean;
begin
  i:= υπόλοιπα;
  j:= 2*i;
  temp:= heap[i];
  τέλος:= false;
  {Οσο υπάρχουν παιδιά και η κανονική θέση δεν έχει βρεθεί}
  while (j <= n) and (not τέλος) do
  begin
     {αν υπάρχουν δύο παιδιά διάλεξε το μικρότερο}
     if j < n then
     if heap[j] > heap[j+1] then
     j:= j+1;
     {αν η θέση του στοιχείου βρέθηκε, τότε τέλος, διαφορετικά
     επαναλαμβάνουμε για το επόμενο επίπεδο του δένδρου}
     if temp <= heap [j] then
     τέλος:= true
     else
     begin
       heap[i]:= heap[j];
       i:= j;
       j:=2*i
     end
   end;
   heap[i]:= temp
end  {αναπροσάρμοσε_σωρό};

procedure δημιούργησε_σωρό (var heap: διάνυσμα);
var υπόλοιπα: integer;
begin
    υπόλοιπα:= n div 2;
    while υπόλοιπα > 0 do
    begin
      αναπροσάρμοσε_σωρό (υπόλοιπα);
      υπόλοιπα:= υπόλοιπα - 1
    end
end {δημιούργησε_σωρό}
```

Από τη διαδικασία δημιούργησε_σωρό παρατηρούμε ότι η διαδικασία αναπροσάρμοσε_σωρό εκτελείται n/2 φορές. Η διαδικασία αυτή ακολουθεί κάθε φορά ένα μονοπάτι. Στη χειρότερη περίπτωση το μονοπάτι αυτό επεκτείνεται από τη ρίζα μέχρι τα φύλλα του

δένδρου, δηλαδή ελέγχει το πολύ $\log_2 n$ στοιχεία. Ετσι η πολυπλοκότητα του αλγόριθμου κατασκευής ενός σωρού είναι:

$$(n/2)\log_2 n.$$

5.7 Σωροί και ουρές προτεραιότητας

Ο σωρός είναι η καλύτερη δομή για την υλοποίηση μιας ουράς προτεραιότητας. Πράγματι, όταν μια ουρά υλοποιείται με μια λίστα ή μ' έναν πίνακα, τότε οι διαδικασίες εισαγωγής και διαγραφής ενός στοιχείου απαιτούν $O(n)$ συγκρίσεις. Οταν χρησιμοποιήσουμε όμως σωρό οι ίδιες διαδικασίες απαιτούν μόνο $O(\log_2 n)$ συγκρίσεις. Στην περίπτωση του σωρού το στοιχείο με την υψηλότερη προτεραιότητα θα βρίσκεται πάντα στην κορυφή του σωρού.

Ορισμός τύπου και δήλωση μιάς ουράς προτεραιότητας

```
const max = ...;
type προτεραιότητα = integer;
     τύπος_κλειδιού = ...;
     τύπος_στοιχείου = record
                         κλειδί: τύπος_κλειδιού;
                         προτ  : προτεραιότητα
                       end;
     τύπος_ουράς = record
                     q: array [0..max] of τύπος_στοιχείου;
                     μέγεθος: 0..max
                   end;
     var ουρά: τύπος_ουράς
```

Οι βασικές πράξεις μιας ουράς προτεραιότητας είναι:

1. **Δημιουργία ουράς**

```
procedure δημιούργησε (var ουρά: τύπος_ουράς);
begin
  ουρά.μέγεθος:= 0
end
```

2. **Ελεγχος αν η ουρά είναι κενή**

```
function κενή (ουρά: τύπος_ουράς): boolean;
begin
  κενή:= ουρά.μέγεθος = 0
end
```

3. **Ελεγχος αν η ουρά είναι γεμάτη**

```
function γεμάτη (ουρά: τύπος_ουράς): boolean;
begin
  γεμάτη:= ουρά.μέγεθος = max
end
```

4. **Εισαγωγή στοιχείου σε ουρά προτεραιότητας**

Το στοιχείο εισάγεται κανονικά στον πίνακα, στο τέλος της ουράς και κατόπιν μετακινείται προς τα πάνω, αν η προτεραιότητα του νέου στοιχείου είναι μεγαλύτερη από την προτεραιότητα του κόμβου_πατέρα.

```
procedure εισάγαγε (x: τύπος_κλειδιού; p: προτεραιότητα;
                                        var ουρά: τύπος_ουράς);
var j,k: integer;
begin
  with ουρά do
  begin
    μέγεθος:= μέγεθος+1;
    q[μέγεθος].κλειδί:= x;
    q[μέγεθος].προτ:= p;
    q[0]:= q[μέγεθος];
    k:= μέγεθος;
    j:= k div 2;
    while q[j].προτ < p do
    begin
      q[k]:= q[j];
      k:= j;
      j:= j div 2
    end;
    q[k]:= q[0]
  end
end {εισάγαγε}
```

5. **Εξαγωγή στοιχείου από ουρά προτεραιότητας**

Το στοιχείο που εξάγεται είναι το πρώτο στοιχείο, q[1]. Κατόπιν τοποθετούμε στη θέση q[1] το στοιχείο q[μέγεθος] και ελαττώνουμε το μέγεθος της ουράς κατά 1. Στη συνέχεια το στοιχείο q[μέγεθος] μετακινείται προς τα κάτω στην ουρά, αν η προτεραιότητά του είναι μικρότερη από την προτεραιότητα ενός τουλάχιστον των παιδιών του.

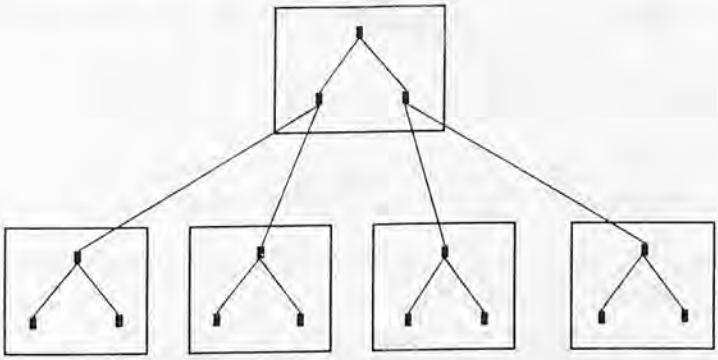
```
procedure εξάγαγε(var x: τύπος_κλειδιού; var p: προτεραιότητα;
                                          var ουρά: τύπος_ουράς);
var i,j: integer;
    τέλος: boolean;
begin
  with ουρά do
    begin
      x:= q[1].κλειδί;
      p:= q[1].προτ;
      q[1]:= q[μέγεθος];
      μέγεθος:= μέγεθος-1;
      i:= 1;
      j:= 2*i;
      q[0] := q[i];
      τέλος:= false;
      while (j <= μέγεθος) and (not τέλος) do
      begin
        if j < μέγεθος then
        if (q[j].προτ < q[j+1].προτ) then
        j:= j+1;
        if (q[i].προτ >= q[j].προτ) then
        τέλος:= true
        else
        begin
          q[i]:= q[j];
          i:= j;
          j:= 2*i
        end
      end;
      q[i]:= q[0];
    end
end {εξάγαγε}
```

5.8 Αλλες κατηγορίες ισοζυγισμένων δένδρων

Στο κεφάλαιο αυτό θα εισαγάγουμε μια νέα κατηγορία ισοζυγισμένων δένδρων. Ας υποθέσουμε ότι ένα δυαδικό δένδρο είναι αποθηκευμένο σ'ένα δίσκο. Είναι γνωστό ότι η μεταφορά των δεδομένων από ένα δίσκο γίνεται κατά ομάδες (blocks), το μέγεθος των οποίων είναι συνήθως 512, 1024 ή 2048 bytes. Η αναζήτηση (seek) μιας ομάδας πάνω στο δίσκο είναι μια μηχανική, χρονοβόρα διαδικασία, που εξαρτάται από τη θέση, της ομάδας και την τρέχουσα θέση της κεφαλής του βραχίονα γραψίματος/διαβάσματος στο δίσκο.

Οταν έχουμε αποθηκεύσει ένα δυαδικό δένδρο σ'ένα δίσκο, τότε

η προσπέλαση σ'ένα κόμβο του δένδρου είναι δυνατόν ν'απαιτεί την προσπέλαση μιας ομάδας του δίσκου. Αυτό σημαίνει ότι απαιτούνται $\log_2 n$ προσπελάσεις ομάδων για τον εντοπισμό ενός κόμβου, όπου n είναι το πλήθος των κόμβων του δένδρου. Επειδή όμως το μέγεθος των στοιχείων ενός κόμβου είναι συνήθως μικρό, είναι δυνατόν οι κόμβοι ενός δένδρου να ομαδοποιηθούν, έτσι ώστε να ελαττωθεί το ύψος του δένδρου και κατά συνέπεια το πλήθος των προσπελάσεων. Στο δένδρο του σχήματος 5.15 έχουμε ομαδοποιήσει τους 15 κόμβους του με τέτοιο τρόπο, ώστε να χρειάζονται μόνο δύο προσπελάσεις του δίσκου για τον εντοπισμό ενός κλειδιού, αντί για τέσσερις που χρειαζόμαστε για το αρχικό δυαδικό δένδρο.



Σχήμα 5.15

Κάθε ομάδα περιέχει 3 κόμβους και συνδέεται με 4 άλλους κόμβους-ομάδες. Αν επεκτείνουμε το δένδρο σ'ένα ακόμη επίπεδο, τότε μπορούν να προστεθούν 16 νέοι κόμβοι-ομάδες. Με τον τρόπο αυτό μπορούμε να εντοπίσουμε έναν κόμβο_ομάδα του δένδρου με πολύ λίγες προσπελάσεις στο δίσκο. Η αναζήτηση του κλειδιού, που ψάχνουμε μέσα σ'ένα κόμβο-ομάδα, γίνεται στην κύρια μνήμη με ακολουθιακό ή δυαδικό ψάξιμο. Με την χρήση μεγάλων ομάδων οπωσδήποτε ελαχιστοποιούμε το πλήθος των προσπελάσεων αλλά, κάθε προσπέλαση ομάδας απαιτεί τη μεταφορά περισσοτέρων του ενός στοιχείων, από τα οποία πολλά δε χρησιμοποιούνται. Ωστόσο όμως, η αύξηση του **παράγοντα ομαδοποίησης (blocking factor)**, δηλαδή του πλήθους των στοιχείων, που περιέχονται σε μία ομάδα, αξίζει τον κόπο, όπως θα διαπιστώσουμε στα επόμενα.

Ενας τρόπος ομαδοποίησης που διατηρεί το δένδρο ισοζυγισμένο μετά από εισαγωγές/διαγραφές στοιχείων χρησιμοποιείται σε μιά ειδική κατηγορία δένδρων που ονομάζονται B-δένδρα. Πριν όμως εξετάσουμε τα B-δένδρα λεπτομερειακά, θα περιγράψουμε μια γενικότερη δομή δένδρου, τα m-κατευθυνόμενα δένδρα.

5.9 m-κατευθυνόμενα δένδρα (m-way trees)

Ορισμός: Ενα δένδρο ονομάζεται **m-κατευθυνόμενο**, όταν όλοι οι κόμβοι του έχουν βαθμό μικρότερο ή ίσο του m.

Ορισμός: Ενα δένδρο T είναι ένα **m-κατευθυνόμενο δένδρο αναζήτησης**, αν T = nil **ή** αν T ≠ nil και έχει τις εξής ιδιότητες:

(i) Κάθε κόμβος του δένδρου έχει την μορφή:

$$n, A_0, (K_1, A_1)\ (K_2, A_2) \ldots (K_n, A_n)$$

όπου A_i, $0 \le i \le n$ είναι δείκτες στα υπόδενδρα του κόμβου και K_i, $1 \le i \le n$ είναι οι τιμές των κλειδιών ($1 \le n < m$)

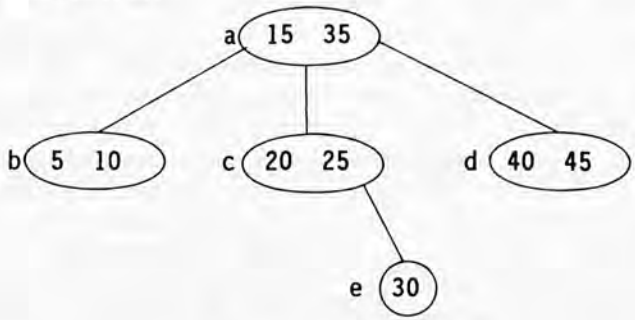
(ii) $K_i < K_{i+1}$, $\quad 1 \le i \le n - 1$

(iii) Ολες οι τιμές των κλειδιών στο υπόδενδρο A_i είναι μικρότερες από την τιμή του κλειδιού K_{i+1}, $0 \le i < n$.

(iv) Ολες οι τιμές των κλειδιών στο υπόδενδρο A_n είναι μεγαλύτερες από K_n και

v) Τα υπόδενδρα A_i, $0 \le i \le n$ είναι m-κατευθυνόμενα δένδρα αναζήτησης.

Στο σχήμα 5.16 δίνεται ένα παράδειγμα 3-κατευθυνόμενου δένδρου αναζήτησης.



Σχήμα 5.16

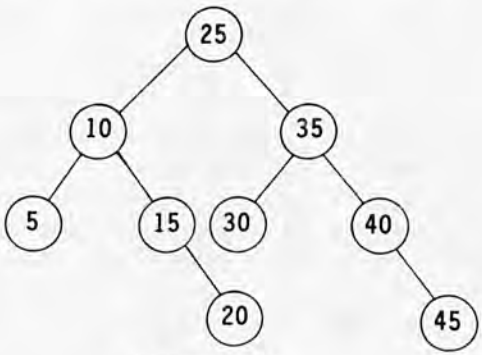
Σύμφωνα με τον παραπάνω ορισμό οι κόμβοι του δένδρου του σχήματος 5.16 περιγράφονται ως εξής:

Κόμβος	Τύπος
a	2, b, (15, c), (35, d)
b	2, nil, (5, nil), (10, nil)
c	2, nil, (20, nil), (25, e)
d	2, nil, (40, nil), (45, nil)
e	1, nil, (30, nil)

Ο αλγόριθμος της αναζήτησης ενός στοιχείου (κλειδιού) σ'ένα m-κατευθυνόμενο δένδρο περιλαμβάνει:

α) τον εντοπισμό του κόμβου που περιέχει το κλειδί και
β) την αναζήτηση του κλειδιού μέσα στον κόμβο αυτό.

Η μέθοδος αναζήτησης του κλειδιού μέσα στον κόμβο εξαρτάται από το πλήθος των κλειδιών. Αν το πλήθος των κλειδιών είναι μεγάλο, τότε κάνουμε δυαδική αναζήτηση. Ωστόσο η διαδικασία αυτή δεν είναι τόσο χρονοβόρα, όσο είναι ο εντοπισμός του κόμβου, που περιέχει το κλειδί. Αν, για παράδειγμα, ψάχνουμε για το κλειδί που έχει τιμή ίση με 30, τότε θα πρέπει να ελέγξουμε τρείς κόμβους του δένδρου. Αυτό σημαίνει ότι χρειάζονται τρείς προσπελάσεις για την ανάκληση στη μνήμη των κόμβων a, c, και e, όταν το δένδρο είναι αποθηκευμένο σ'ένα δίσκο. Το καλύτερο δυαδικό δένδρο αναζήτησης με τα ίδια κλειδιά (Σχήμα 5.17) απαιτεί, στη χειρότερη περίπτωση, 4 αναζητήσεις/προσπελάσεις για τον εντοπισμό των κλειδιών, για παράδειγμα K = 20 και K = 45.



Σχήμα 5.17

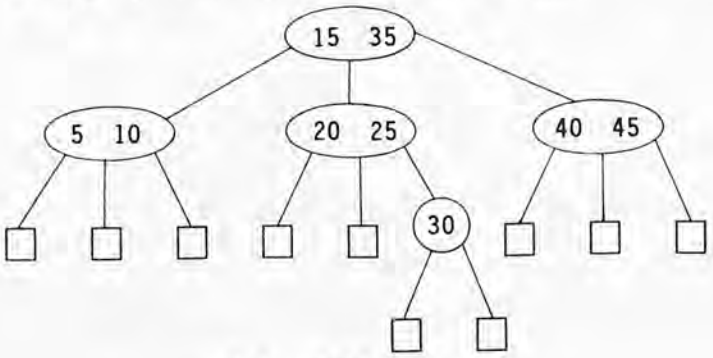
Για τον εντοπισμό του κόμβου που περιέχει ένα κλειδί, το πλήθος των ελέγχων (προσπελάσεων) είναι ίσο με το ύψος του δένδρου. Οπως αναφέραμε ο χρόνος προσπέλασης, ενός δίσκου είναι πολύ μεγαλύτερος από τον χρόνο αναζήτησης του κλειδιού μέσα στον κόμβο. Για το λόγο αυτό προσπαθούμε να ελαχιστοποιήσουμε το ύψος του δένδρου.

Οπως είναι γνωστό σ'ένα δένδρο βαθμού m και ύψους $h > 1$, το μέγιστο πλήθος των κόμβων του είναι $(m^h - 1)/(m - 1)$. Εφ'όσον κάθε κόμβος σ'ένα m-κατευθυνόμενο δένδρο έχει το πολύ m - 1 κλειδιά, το μέγιστο πλήθος των κλειδιών του θα είναι m^h-1. Για παράδειγμα, ένα δυαδικό δένδρο με h = 3 έχει 7 κόμβους (κλειδιά), ενώ ένα 100-κατευθυνόμενο δένδρο με h = 3 έχει: $100^3 - 1 = 999999$ κλειδιά. Για να ελαχιστοποιήσουμε όμως το πλήθος των προσπελάσεων, δηλαδή να επιτύχουμε την καλύτερη δυνατή απόδοση για ένα δεδομένο πλήθος κλειδιών, το m-κατευθυνόμενο δένδρο αναζήτησης θα πρέπει να είναι και ισοζυγισμένο. Μια ειδική κατηγορία m-κατευθυνομένων και ισοζυγισμένων δένδρων είναι τα B-δένδρα.

5.10 B-δένδρα

Για τον ορισμό του B-δένδρου εισάγουμε πρώτα την έννοια του **κόμβου αποτυχίας (failure node).** Ενας κόμβος στον οποίο μπορούμε να φτάσουμε κατά την αναζήτηση της τιμής ενός κλειδιού, όταν η τιμή αυτή δεν υπάρχει στο δένδρο, ονομάζεται κόμβος αποτυχίας. Κάθε υπόδενδρο A_i = nil είναι ένας κόμβος αποτυχίας γιατί είναι ένα σημείο στο οποίο φτάνουμε μόνο όταν το κλειδί που ζητάμε δεν υπάρχει στο δένδρο.

Τους κόμβους αποτυχίας θα τους παριστάνουμε με □. Στην πραγματικότητα δεν υπάρχουν αυτοί οι κόμβοι παρά μόνο η τιμή nil, όπου "υπάρχει" ένας τέτοιος κόμβος. Οι κόμβοι αποτυχίας είναι οι μόνοι κόμβοι που δεν έχουν παιδιά. Στο σχήμα 5.18 φαίνονται οι κόμβοι αποτυχίας του δένδρου του σχήματος 5.16.

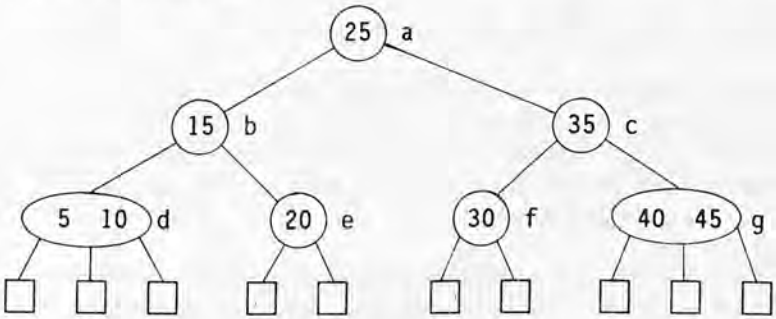


Σχήμα 5.18

Ορισμός: Ενα B-δένδρο, t, βαθμού m, είναι ένα m-κατευθυνόμενο δένδρο αναζήτησης το οποίο είναι είτε κενό είτε είναι ύψους h ≥ 1 και ικανοποιεί τις ιδιότητες:

(i) Ο κόμβος ρίζα έχει τουλάχιστον δύο παιδιά.
(ii) Ολοι οι κόμβοι εκτός από τη ρίζα και τους κόμβους αποτυχίας (F) έχουν τουλάχιστον $\lceil m/2 \rceil$ παιδιά.
(iii) Ολοι οι κόμβοι αποτυχίας βρίσκονται στο ίδιο επίπεδο.

Στο σχήμα 5.19 δίνεται το B-δένδρο βαθμού 3, με τα ίδια κλειδιά όπως του δένδρου του σχήματος 5.16. Από το σχήμα 5.19 παρατηρούμε ότι παρ' όλο που το πλήθος των ενδιάμεσων κόμβων έχει αυξηθεί, απλουστεύονται, όπως, θα δούμε, οι πράξεις εισαγωγής και διαγραφής ενός κλειδιού.



Σχήμα 5.19

Σημείωση.

Τα B-δένδρα είναι μια πρόσφατη δομή δεδομένων. Ανακοινώθηκαν μόλις το 1972 από τους R. Bayer και E. McGreight και έχουν τεράστιες εφαρμογές σήμερα στη **δεικτοδοτημένη ακολουθιακή οργάνωση αρχείων (index sequential file organization)**. Για το λόγο αυτό, μερικοί ορισμοί, όπως για παράδειγμα ο ορισμός του βαθμού ενός B-δένδρου, δεν είναι ομοιόμορφοι στη βιβλιογραφία.

Ο ορισμός που χρησιμοποιούμε εμείς προτάθηκε από τον Knuth. Σύμφωνα με τον ορισμό αυτό, ο βαθμός ενός B-δένδρου είναι το μέγιστο πλήθος παιδιών, που μπορεί να έχει ένας κόμβος. Το πλήθος των κλειδιών (στοιχείων) κάθε κόμβου είναι κατά ένα μικρότερο από το πλήθος των παιδιών του κόμβου. Οσο αφορά τ'όνομα B δένδρο υπάρχουν πολλές εκδοχές, όπως οι: "Balanced", "Bayers" και "Boeing" (το όνομα της εταιρίας που εργάζονταν οι Bayers-McGreight όταν τα πρότειναν).

Στη συνέχεια θα προσπαθήσουμε ν'απαντήσουμε σε δύο βασικά

ερωτήματα που αφορούν τα B-δένδρα:

E1. Ποιό είναι το ελάχιστο πλήθος των κλειδιών ενός B-δένδρου βαθμού m και ύψος h;

E2. Ποιό είναι το μέγιστο ύψος, h, ενός B-δένδρου βαθμού m με N κλειδιά;

Από τον ορισμό του B-δένδρου προκύπτει ότι:

στο επίπεδο 1 υπάρχει ένας κόμβος, η ρίζα,

στο επίπεδο 2 υπάρχουν τουλάχιστον 2 κόμβοι,

στο επίπεδο 3 υπάρχουν τουλάχιστον $2\lceil m/2 \rceil$ και, γενικά,

στο επίπεδο i υπάρχουν τουλάχιστον $2\lceil m/2 \rceil^{i-2}$ κόμβοι.

Αν το πλήθος των κλειδιών είναι N και $K_1 < K_2 <, \ldots, < K_N$, τότε το πλήθος των κόμβων αποτυχίας είναι $(N + 1)$ (γιατί;). Αυτό συνεπάγεται ότι υπάρχουν $N + 1$ διαφορετικοί κόμβοι στους οποίους θα μπορούσε να φτάσει κανείς, για κάποια τιμή κλειδιού, που δεν υπάρχει στο δένδρο. Ετσι,

$$N + 1 = \text{πλήθος κόμβων αποτυχίας}$$
$$= \text{πλήθος κόμβων στο επίπεδο } h + 1$$
$$\geq 2 \left\lceil \frac{m}{2} \right\rceil^{h-1}$$

ή
$$N \geq 2 \left\lceil \frac{m}{2} \right\rceil^{h-1} - 1$$

Από την ίδια σχέση προκύπτει επίσης ότι:

$$h \leq \log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right) + 1$$

Δηλαδή το πλήθος των προσπελάσεων για τον εντοπισμό ενός κόμβου (που περιέχει ένα κλειδί), είναι το πολύ $\log_{\lceil m/2 \rceil}((N+1)/2)+1$. Ας δούμε όμως καλύτερα τι σημαίνει αυτό σε αριθμούς.

Αν $m = 200$ και $N = 2 \times 10^8 - 1$, τότε

$$h \leq \left\lceil \log_{100} \left(\frac{2 \times 10^8}{2} \right) + 1 \right\rceil = 5!$$

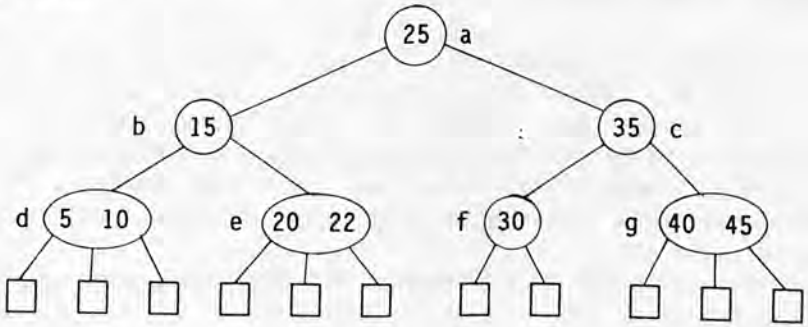
Συνεπώς η χρησιμοποίηση ενός Β-δένδρου μεγάλου βαθμού έχει ως αποτέλεσμα να κάνουμε πολύ λίγες προσπελάσεις στους κόμβους του δένδρου, ακόμη κι όταν το πλήθος των στοιχείων του είναι πολύ μεγάλο. Οπωσδήποτε όμως το μέγεθος του m δεν μπορεί να μεγαλώνει απεριόριστα, γιατί περιορίζεται από το μέγεθος της ομάδας (block). Στα επόμενα θα περιγράψουμε τους αλγόριθμους εισαγωγής και διαγραφής στοιχείου σε Β-δένδρο θεωρώντας ότι το δένδρο βρίσκεται στην κυρία μνήμη.

5.11 Εισαγωγή στοιχείου σε Β-δένδρο.

Η εισαγωγή ενός νέου στοιχείου σε Β-δένδρο θα πρέπει να γίνεται κατά τέτοιο τρόπο, ώστε το δένδρο που προκύπτει να παραμένει Β-δένδρο. Ο αλγόριθμος της εισαγωγής χρησιμοποιεί αρχικά μια διαδικασία αναζήτησης, για τον εντοπισμό του κόμβου στον οποίο θα πρέπει να εισαχθεί το νέο στοιχείο. Μετά διακρίνουμε δύο περιπτώσεις:

1. Ο κόμβος όπου πρέπει να εισαχθεί το νέο στοιχείο έχει χώρο γι'αυτό, δηλαδή περιέχει λιγότερα από m-1 στοιχεία.

Εστω ότι στο δένδρο του σχήματος 5.19 θέλουμε να κάνουμε εισαγωγή του στοιχείου K = 22. Από τη διαδικασία αναζήτησης προκύπτει ότι το 22 θα πρέπει να μπει στον κόμβο e. Ο κόμβος αυτός έχει ένα μόνο κλειδί, επομένως έχει χώρο για ένα ακόμη. Έτσι μετά την εισαγωγή το δένδρο θα έχει τη μορφή που δείχνει το σχήμα 5.20



Σχήμα 5.20

2. Ο κόμβος, έστω p όπου πρέπει να εισαχθεί το νεο στοιχείο

είναι ήδη γεμάτος, δηλαδή περιέχει m-1 στοιχεία (κλειδιά).

Στην περίπτωση αυτή "εισαγάγουμε" το νέο κλειδί στον κόμβο p, στην κατάλληλη θέση, έτσι ώστε τα κλειδιά του κόμβου να είναι ταξινομημένα και **αναδιοργανώνουμε (reogranize)** το δένδρο. Η αναδιοργάνωση γίνεται με τέτοιο τρόπο ώστε το δένδρο να παραμένει B_δένδρο μετά την εισαγωγή, δηλαδή κάθε κόμβος να έχει το πολύ m-1 κλειδιά. Ας υποθέσουμε ότι ο κόμβος p μετά την εισαγωγή του νέου στοιχείου είναι:

p: $\boxed{m,\ A_0,\ (K_1,\ A_1),\ldots\ (K_m,\ A_m)}$

→ όπου $K_i < K_{i+1}$, $1 \le i \le m-1$.

Κατασκευάζουμε δύο κόμβους p και p_1 χρησιμοποιώντας τους εξής τύπους:

p: $\boxed{\lceil m/2 \rceil - 1,\ A_0,\ (K_1, A_1)\ ,\ldots,\ (K_{\lceil m/2 \rceil -1},\ A_{\lceil m/2 \rceil -1})}$

p_1: $\boxed{m - \lceil m/2 \rceil,\ A_{\lceil m/2 \rceil},\ (K_{\lceil m/2 \rceil +1}, A_{\lceil m/2 \rceil +1})\ ,\ \ldots\ ,\ (K_m,\ A_m)}$

Η τιμή του κλειδιού $K_{\lceil m/2 \rceil}$, που απομένει, και ο νέος κόμβος p_1 σχηματίζουν το ζεύγος:

$$(K_{\lceil m/2 \rceil},\ p_1)$$

Το ζεύγος αυτό προσπαθούμε να το εισαγάγουμε στον κόμβο πατέρα του p, εφόσον αυτό είναι δυνατόν. Αν ο κόμβος πατέρας είναι ήδη γεμάτος, τότε εφαρμόζουμε την ίδια διαδικασία για τον κόμβο αυτό. Έτσι στη χειρότερη περίπτωση η διαδικασία προχωρεί προς τα πάνω, μέχρις ότου φτάσει στη ρίζα του δένδρου, οπότε δημιουργείται μια καινούργια ρίζα και το ύψος του δένδρου αυξάνεται κατά ένα.

Εστω ότι θέλουμε να εισαγάγουμε στο δένδρο του σχήματος 5.20 τα κλειδιά 17, και 14:

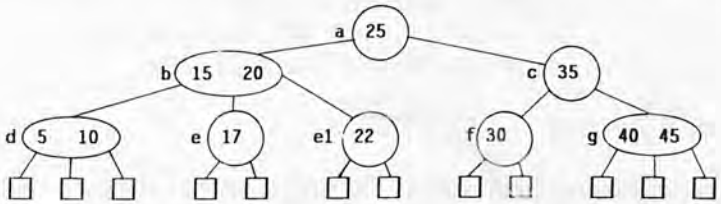
α. εισαγωγή K = 17

Με την αναζήτηση εντοπίζουμε τον κόμβο e στον οποίο θα

πρέπει να προστεθεί το 17. Ο κόμβος αυτός είναι ήδη γεμάτος. Ετσι σύμφωνα με την παραπάνω περιγραφή, ο κόμβος e σπάει σε δύο κόμβους:

e: (17) e_1: (22)

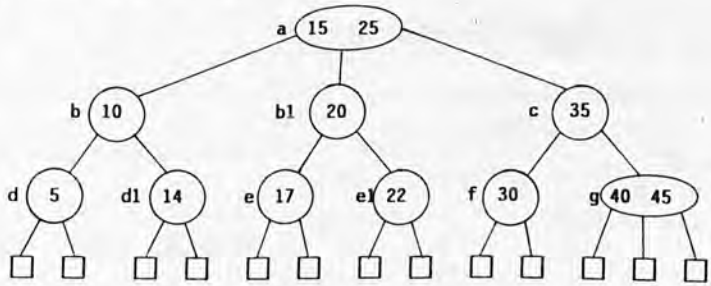
και το στοιχείο (20, e_1) ανεβαίνει στον πατέρα του e, εφόσον υπάρχει χώρος. Το δένδρο που προκύπτει φαίνεται στο σχήμα 5.21.



Σχήμα 5.21

β. Εισαγωγή K = 14

Το K θα πρέπει να εισαχθεί στον κόμβο d, που είναι ήδη γεμάτος. Ετσι ο κόμβος d σπάει σε δύο κόμβους (d με περιεχόμενο το 5 και d_1 με περιεχόμενο 14) και το στοιχείο K = 10 ανεβαίνει στον πατέρα του d. Ο κόμβος αυτός (b), που είναι επίσης γεμάτος, σπάει σε δύο κόμβους (b με περιεχόμενο 10 και b_1 με περιεχόμενο 20) και το στοιχείο 15 ανεβαίνει στη ρίζα. Το αποτέλεσμα των παραπάνω πράξεων φαίνεται στο σχήμα 5.22.



Σχήμα 5.22

Παρακάτω δίνουμε τον ορισμό του τύπου και τη δήλωση ενός Β-δένδρου και τις διαδικασίες αναζήτησης και εισαγωγής σε Pascal:

```
const   m = ... {βαθμός του Β-δένδρου};
type
   τύπος_κλειδιού = integer;
   Β_δείκτης = ^ Β_κόμβος;
   Β_κόμβος = record
                πλήθος_κλειδιών: 1..m-1;
                k: array [1..m-1] of τύπος_κλειδιού;
                A: array [0..m-1] of Β_δείκτης
              end;
var t: Β_δείκτης
```

Στην παρακάτω διαδικασία "Β_αναζήτησε", όταν η μεταβλητή τύπου boolean "βρέθηκε" γίνει true, θα σημαίνει ότι το στοιχείο x υπάρχει ήδη στο δένδρο, διαφορετικά η διαδικασία επιστρέφει τον κόμβο στον οποίο θα πρέπει να γίνει η εισαγωγή του νέου στοιχείου x.

```
procedure Β_αναζήτησε (x: τύπος_κλειδιού; t: Β_δείκτης;
                       var p: Β_δείκτης; var βρέθηκε: boolean);

var i, n: integer;
      q: Β_δείκτης;
begin
  p:= t;
  q:= nil;  {q είναι ο κόμβος πατέρας του p}
  βρέθηκε:= false;
  while (p <> nil) and (not βρέθηκε) do
  begin
    n:= p^.πλήθος_κλειδιών;
    if  x < p^.k[1] then
      begin
        q:= p;
        p:= p^.A[0]
      end
    else
     if x > p^.k[n] then
     begin
       q:= p;
       p:= p^.A[n]
     end

     else
      begin
```

```
        i:= 1;
        while (i <= n) and (not βρέθηκε) do
        begin
          if x = p^.k[i] then
           βρέθηκε:= true
          else
          if (x > p^.k[i]) and (x < p^.k[i + 1]) then
          begin
            q:= p;
            p:= p^.A[i]
          end;
          i = i + 1
        end
      end {else}
  end; {while p <> nil}
  p:= q
end {B_αναζήτησε}
```

Τέλος η διαδικασία της εισαγωγής περιγράφεται με τη βοήθεια των παρακάτω διαδικασιών και συναρτήσεων, που αφήνονται ως άσκηση στον αναγνώστη.

```
function πατέρας (p: B_δείκτης): B_δείκτης;
{επιστρέφει τον κόμβο πατέρα του p}

procedure ταξινόμησε (p: B_δείκτης; x: τύπος_κλειδιού);
{τοποθετεί το νέο στοιχείο (x, nil) στη κατάλληλη θέση, με
τέτοιο τρόπο, ώστε τα στοιχεία του p να είναι ταξινομημένα)

procedure  δημιούργησε_νέο_κόμβο  (var p, p1: B_δείκτης);
{χωρίζει τον κόμβο p σε δύο κόμβους p και p1, σύμφωνα με τον
αλγόριθμο που περιγράψαμε παραπάνω}

procedure B_εισάγαγε (x:τύπος_κλειδιού; var t: B_δείκτης;
                                  έγινε_εισαγωγή: boolean);
var D, p1, r, p: B_δείκτης;
    βρέθηκε, τέλος: boolean;

begin
  D:= nil; {(x, D) είναι το στοιχείο που πρέπει να εισαχθεί}
  B_αναζήτησε (x, t, p, βρέθηκε);
  if not βρέθηκε  then
    begin
      τέλος:= false;
      while (p <> nil) and (not τέλος) do
      begin
```

```
        ταξινόμησε (p, x);
        if p^.πλήθος_κλειδιών ≤ m - 1 then
        {υπάρχει χώρος στο κόμβο p για το νέο στοιχείο}
        τέλος:= true
        else
        begin
          δημιούργησε_νέο_κόμβο (p, p1);
          x:= p^. k[m/2 + 1];  { K[m/2] }
          D:= p1;
          p:= πατέρας (p)
        end
      end {while};
      if not τέλος then
       begin
         {νέα ρίζα θα πρέπει να δημιουργηθεί}
         new (r);
         r^.πλήθος_κλειδιών:= 1;
         r^.k[1]:= x;
         r^.A[0]:= t;
         r^.A[1]:= D;
         t:= r
       end;
       έγινε_εισαγωγή:= true
   end
   else {if not βρέθηκε}
     έγινε_εισαγωγή:= false
end {B_εισάγαγε}.
```

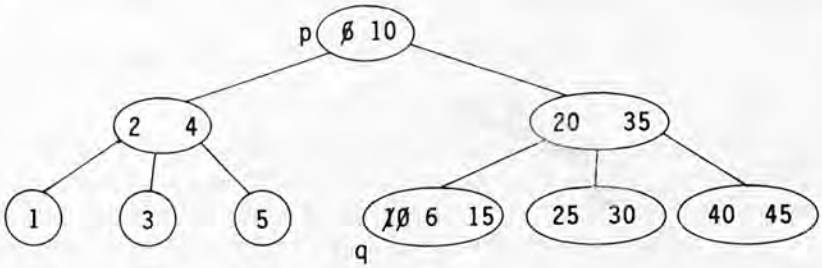
Οπως αναφέραμε, όταν ένα B-δένδρο βρίσκεται αποθηκευμένο σ'ένα δίσκο, η προσπέλαση του δίσκου είναι η πιο χρονοβόρα διαδικασία . Για τον λόγο αυτό, στην ανάλυση του αλγορίθμου της εισαγωγής μετράμε το πλήθος των προσπελάσεων του δίσκου. Ο χρόνος αναζήτησης του κλειδιού μέσα στον κόμβο θεωρείται αμελητέος, γιατί η διαδικασία αυτή γίνεται στη κύρια μνήμη. Αν υποθέσουμε ότι το δένδρο έχει ύψος h, τότε για τον εντοπισμό του κόμβου, που θα μπει το νέο στοιχείο, απαιτούνται τουλάχιστον h προσπελάσεις. Επιπλέον απαιτούνται δύο προσπελάσεις κάθε φορά, που ένας κόμβος χωρίζεται σε δύο. Αν συνεπώς το πλήθος των κόμβων, που χωρίζονται είναι i, τότε το συνολικό πλήθος των προσπελάσεων είναι:

$$h + 2i + 1 \leq 3h + 1 = O(h).$$

5.12 Διαγραφή ενός στοιχείου από B-δένδρο

Αρχικά εντοπίζουμε τον κόμβο p, που περιέχει το προς διαγραφήν στοιχείο, έστω K, με την διαδικασία αναζήτησης. Ο

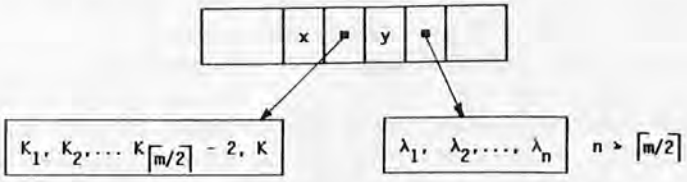
κόμβος p μπορεί να είναι κόμβος_φύλλο ή όχι. Αν ο p δεν είναι φύλλο, τότε ανταλλάσουμε αμοιβαία το προς διαγραφήν στοιχείο με το μικρότερο στοιχείο του επόμενου κόμβου σύμφωνα, με την ενδοδιατεταγμένη έννοια του επόμενου. Αυτό μπορεί να γίνει, όπως στα δυαδικά δένδρα, ακολουθώντας το μονοπάτι, που οδηγεί στο αριστερότερο παιδί του δεξιού παιδιού του K, μέχρις ότου βρεθεί ένας κόμβος αποτυχίας, έστω q.



Σχήμα 5.23

Στο σχήμα 5.23 δίνεται ένα παράδειγμα για την διαγραφή του στοιχείου 6. Το στοιχείο 6 ανταλλάσεται αμοιβαία με το μικρότερο στοιχείο (10) του κόμβου q και μετά διαγράφεται το 6 από τον κόμβο q. Ετσι το πρόβλημα ανάγεται πάντα στη διαγραφή ενός στοιχείου απο έναν κόμβο-φύλλο. Αν ο κόμβος q περιέχει περισσότερα από ($\lceil m/2 \rceil$ -1) στοιχεία, τότε δέν υπάρχει λόγος **επανα-ισοζύγισης (rebalance)** του δένδρου, έτσι ώστε να πληρεί τις ιδιότητες του B δένδρου και ο αλγόριθμος τελειώνει. Οταν όμως ο q περιέχει ακριβώς ($\lceil m/2 \rceil$ - 1) στοιχεία, τότε θα πρέπει να κάνουμε αναπροσαρμογή του δένδρου. Αυτό μπορεί να γίνει με δύο τρόπους:

1. Με δανεισμό στοιχείων από ένα γειτονικό κόμβο (αριστερό ή δεξιό), όταν το πλήθος των στοιχείων του κόμβου είναι μεγαλύτερο ή ίσο του $\lceil m/2 \rceil$, όπως δείχνει το σχήμα 5.24.

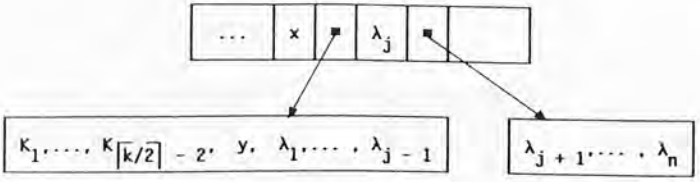


Σχήμα 5.24

Στην περίπτωση αυτή κατασκευάζουμε τη διατεταγμένη ακολουθία:

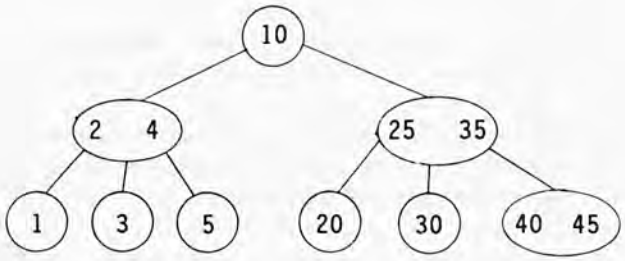
$$K_1, K_2, \ldots, K_{\lceil m/2 \rceil} - 2, y, \lambda_1, \lambda_2, \ldots, \lambda_n$$

Το μεσαίο στοιχείο λ_j $(1 \leq j \leq n)$ της ακολουθίας αυτής αντικαθιστάει το στοιχείο y στον κόμβο πατέρα, ενώ τα στοιχεία $K_1, \ldots, K_{\lceil m/2 \rceil} - 2$, y, $\lambda_1, \ldots, \lambda_{j-1}$ πηγαίνουν στον κόμβο q και τα υπόλοιπα στοιχεία $\lambda_{j+1}, \ldots, \lambda_n$ στον γειτονικό του κόμβο, όπως φαίνεται στο σχήμα 5.25.



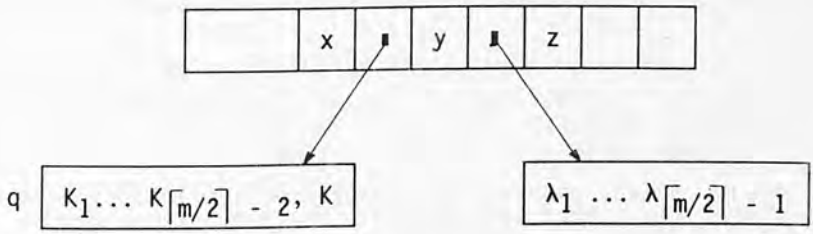
Σχήμα 5.25

Εστω, για παράδειγμα, πως θέλουμε να διαγράψουμε το στοιχείο 15 από το δένδρο του σχήματος 5.23. Επειδή μετά την διαγραφή του στοιχείου 6 ο κόμβος q περιέχει μόνο το 15, η διαγραφή γίνεται με δανεισμό στοιχείων από γειτονικό κόμβο, εφ'όσον υπάρχει τέτοιος κόμβος, οπότε προκύπτει το δένδρο του σχήματος 5.26.



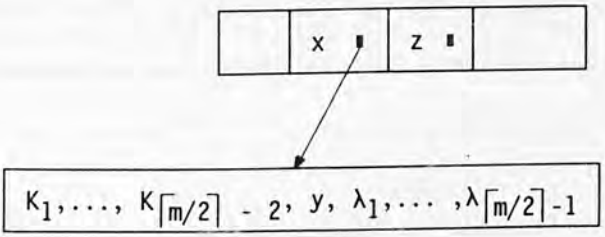
Σχήμα 5.26

2. Ο γειτονικός κόμβος του q έχει ακριβώς $\lceil m/2 \rceil - 1$ στοιχεία (βλ. σχήμα 5.27), οπότε δεν μπορούμε να δανειστούμε στοιχεία από αυτόν. Στη περίπτωση αυτή συγχωνεύουμε τους δύο κόμβους μαζί, έτσι ώστε οι κόμβοι να φτιάξουν ένα νέο κόμβο.



Σχήμα 5.27

Η συγχώνευση γίνεται με δανεισμό ενός στοιχείου από τον κόμβο πατέρα. Τα $(\lceil m/2 \rceil - 2)$ στοιχεία του q με τα $\lceil m/2 \rceil - 1$ του γείτονά του και το ένα στοιχείο από τον κόμβο πατέρα σχηματίζουν έναν κόμβο με m-1 στοιχεία, όπως φαίνεται στο σχήμα 5.28



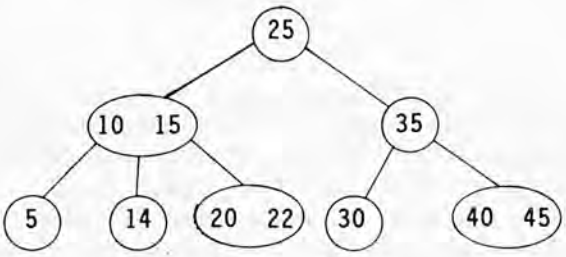
Σχήμα 5.28

Αυτό σημαίνει ότι το στοιχείο y θα πρέπει να μεταφερθεί από τον κόμβο πατέρα. Αν ο κόμβος πατέρας απομείνει με λιγότερα από $\lceil m/2 \rceil - 1$ στοιχεία, τότε θα πρέπει να επαναλάβουμε τη διαδικασία της αναπροσαρμογής και για τον κόμβο αυτόν. Στη χειρότερη περίπτωση, η διαδικασία της συγχώνευσης (2) μπορεί να συνεχιστεί μέχρι τη ρίζα του δένδρου. Αν η ρίζα περιέχει ένα στοιχείο, τότε αυτό συγχωνεύεται μ'ένα της παιδί και το ύψος του δένδρου ελαττώνεται κατά ένα. Έτσι, ο αλγόριθμος της διαγραφής συνοψίζεται ως εξής:

1. Αν το κλειδί που διαγράφεται δε βρίσκεται σε φύλλο, τότε άλλαξέ το αμοιβαία με το αμέσως "επόμενό" του κλειδί, που βρίσκεται σε φύλλο.
2. Διάγραψε το κλειδί.
3. Αν το φύλλο απομένει με τουλάχιστον ($\lceil m/2 \rceil - 1$) κλειδιά, τότε ο αλγόριθμος τελειώνει.
4. Αν το φύλλο απομένει με λιγότερα από $\lceil m/2 \rceil - 1$ κλειδιά, τότε εξέτασε τον αριστερό και δεξιό του γείτονα:
   4.1 Αν ένας γείτονας έχει περισσότερα κλειδιά από το ελάχιστο πλήθος ($\lceil m/2 \rceil - 1$), τότε κάνε δανεισμό στοιχείων από το γείτονα αυτό.
   4.2 Αν κανένας γείτονας δεν έχει περισσότερα κλειδιά από το ελάχιστο πλήθος, τότε συγχώνευσε τα δύο φύλλα και το μεσαίο (medium) κλειδί από τον κόμβο πατέρα σ'έναν κόμβο.
5. Αν τα φύλλα συγχωνευθούν, τότε εφάρμοσε τα βήματα 3-6 στον κόμβο πατέρα.
6. Αν το τελευταίο κλειδί από τη ρίζα του B_δένδρου μετακινηθεί τότε το ύψος του δένδρου ελαττώνεται κατά ένα.

Ασκηση

Δείξτε ότι μετά τη διαγραφή του στοιχείου K = 17 από το δένδρο του σχήματος 5.22 προκύπτει το παρακάτω δένδρο:



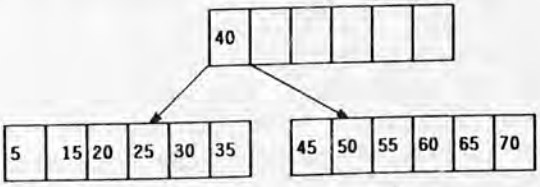
5.13 B* δένδρα

Το 1973 ο Knuth σε μια ανασκόπιση, που έκανε στα B_δένδρα, επέκτεινε τον αλγόριθμο της αναπροσαρμογής του δένδρου κατά την εισαγωγή ενός νέου στοιχείου και περιέλαβε νέους κανόνες για τη διάσπαση (splitting) ενός κόμβου. Οπως αναφέραμε στα B_δένδρα, η

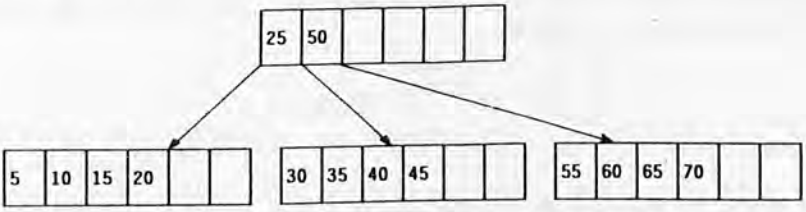
εισαγωγή ενός στοιχείου σ'έναν κόμβο που είναι ήδη γεμάτος (δηλαδή περιέχει m-1 κλειδιά), γίνεται με τη διάσπαση του κόμβου σε δύο άλλους κόμβους, οι οποίοι περιέχουν $\lceil m/2 \rceil - 1$ και $(m - \lceil m/2 \rceil)$ κλειδιά αντίστοιχα, δηλαδή οι νέοι κόμβοι είναι κατά το ήμισυ γεμάτοι. Με τον τρόπο αυτόν μετά από ένα ορισμένο αριθμό από εισαγωγές, ένα μεγάλο ποσοστό του χώρου, που καταλαμβάνει το δένδρο, παραμένει ανεκμετάλευτο.

Η μέθοδος διάσπασης ενός κόμβου που δόθηκε από τον Knuth περιγράφεται ως εξής: Κατά την εισαγωγή του νέου στοιχείου μπορούμε να καθυστερήσουμε τη διάσπαση του κόμβου, κάνοντας μια ανακατανομή των στοιχείων σε γειτονικούς κόμβους. Οταν όμως θέλουμε να διασπάσουμε ένα διάφορο της ρίζας κόμβο, ο οποίος έχει ένα γείτονα που είναι επίσης γεμάτος, τότε μπορούμε να διασπάσουμε δύο κόμβους σε τρεις (2 - 3 split).

Το σπουδαίο αποτέλεσμα αυτής της διάσπασης είναι ότι οι κόμβοι, που προκύπτουν, είναι κατά τα 2/3 γεμάτοι, αντί κατά το ήμισυ που ήταν πριν. Στο σχήμα 5.29 δίνεται ένα παράδειγμα εισαγωγής τού στοιχείου 10 σ'ένα δένδρο βαθμού 5. Ο αλγόριθμος της εισαγωγής διαφοροποιείται όταν πρόκειται να διασπαστεί η ρίζα. Για την περίπτωση αυτή, ο Knuth υπέθετε ότι η ρίζα μπορεί να μεγαλώσει περισσότερο από τους άλλους κόμβους (m-1), έτσι ώστε όταν χωριστεί να δημιουργήσει δύο κόμβους γεμάτους κατά τα 2/3.



Μετά την εισαγωγή του στοιχείου 10:



Σχήμα 5.29

Στη περίπτωση αυτή το δένδρο που προκύπτει λέγεται **B*_δένδρο βαθμού m**. Ενα δένδρο B*_δένδρο βαθμού m χαρακτηρίζεται από τις ιδιότητες:

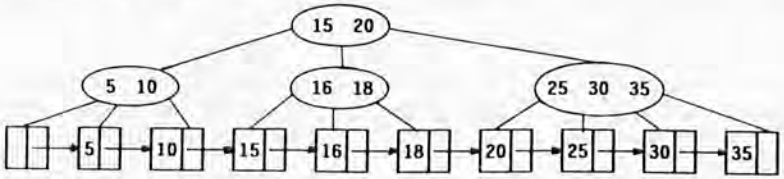
1. Κάθε κόμβος του δένδρου έχει το πολύ m παιδιά.
2. Κάθε κόμβος εκτός από τη ρίζα και τα φύλλα έχει τουλάχιστον (2m - 1)/3 παιδιά.
3. Η ρίζα έχει τουλάχιστον δύο παιδιά.
4. Ολα τα φύλλα του δένδρου είναι στο ίδιο επίπεδο.
5. Ενας κόμβος με k παιδιά έχει k - 1 κλειδιά.
6. Ενα φύλλο περιέχει τουλάχιστον $\lceil (2m - 1)/3 \rceil$ και όχι περισσότερα από m - 1 κλειδιά.

Η νέα ιδιότητα των B*_δένδρων οπωσδήποτε επηρεάζει τους αλγόριθμους εισαγωγής και διαγραφής ενός στοιχείου, που γίνονται περισσότερο πολύπλοκοι.

5.14 B+ δένδρα

Η δομή ενός B_δένδρου επιτρέπει την επεξεργασία των στοιχείων του με **τυχαία σειρά (random order)**. Αν θέλουμε να κάνουμε ακολουθιακή επεξεργασία των στοιχείων, τότε μια απλή μέθοδος θα ήταν να κάνουμε μια ενδοδιατεταγμένη διάσχιση του δένδρου. Η διάσχιση όμως αυτή απαιτεί τη χρησιμοποίηση μιας στοίβας. Για ν'αποφύγουμε τη στοίβα, χρησιμοποιούμε μια δομή που ονομάζεται B+_δένδρο. Το ίδιο πρόβλημα ακόμη πιο έντονο εμφανίζεται όταν το B_δένδρο είναι στη δευτερεύουσα μνήμη. Στην περίπτωση αυτή η ακολουθιακή επεξεργασία του δένδρου, με ενδοδιατεταγμένη διάσχιση, είναι απαγορευτική, γιατί το πλήθος των αναζητήσεων (seeks) που κάνουμε είναι πολύ μεγάλο.

Τα πραγματικά στοιχεία ενός B+ δένδρου αποθηκεύονται στα φύλλα του. Ολοι οι ενδιάμεσοι κόμβοι περιέχουν μόνο τα κλειδιά των στοιχείων. Τα πραγματικά στοιχεία στα φύλλα μπορούν να συνδεθούν με μια γραμμική λίστα. Ετσι επιτυγχάνεται γρήγορη ακολουθιακή προσπέλαση σ'όλα τα στοιχεία. Στο σχήμα 5.30 δίνεται ένα παράδειγμα B+ δένδρου.



Σχήμα 5.30

Ενα πλεονέκτημα των B^+ δένδρων είναι ότι ο κάθε κόμβος τους μπορεί να χωρέσει πολύ περισσότερα στοιχεία, γιατί μόνο τα κλειδιά των στοιχείων του αποθηκεύονται στους ενδιάμεσους κόμβους. Αυτό σημαίνει ότι ο βαθμός m του δένδρου μπορεί να είναι πολύ μεγάλος και κατά συνέπεια, το ύψος του B^+ δένδρου είναι πολύ μικρότερο από το ύψος του αντίστοιχου B_δένδρου.

Ανακεφαλαίωση

Στο κεφάλαιο αυτό εξετάσαμε ειδικές κατηγορίες ισοζυγισμένων δένδρων αναζήτησης. Μεγαλύτερη έμφαση δόθηκε στα B_δένδρα, λόγω των τεράστιων εφαρμογών τους στην οργάνωση ευρετηρίων (indexes) ή καταλόγων (catalogues) με γρήγορη προσπέλαση στις εγγραφές ενός αρχείου. Τα B^+ δένδρα, που εισαγάγαμε στη παράγραφο 5.14, αποτελούν την πλέον αποτελεσματική δομή για την ανάπτυξη δεικτοδοτημένων ακολουθιακών αρχείων. Τα αρχεία αυτά έχουν γρήγορη προσπέλαση στις εγγραφές τους και μπορούν ταυτόχρονα να τύχουν ακολουθιακής επεξεργασίας π.χ. συγχώνευση αρχείων. Οι ορισμοί για την κάθε δομή, που δόθηκαν στο κεφάλαιο αυτό, μαζί με τις αντίστοιχες πράξεις τους θεωρούν ότι η δομή βρίσκεται στη κύρια μνήμη. Αυτό έγινε εσκεμμένα, γιατί ο σκοπός μας εδώ ήταν να δώσουμε μόνο τους ορισμούς των δομών αυτών, μαζί με τις αντίστοιχες πράξεις τους. Αλλωστε οι πράξεις αυτές δε διαφέρουν, όταν οι δομές βρίσκονται στη δευτερεύουσα μνήμη. Εκτενέστερη μελέτη των θεμάτων αυτών με τις αντίστοιχες εφαρμογές τους ανήκουν στο αντικείμενο του μαθήματος Δομές Αρχείων.

Ασκήσεις 5.1

1. Κατασκευάστε τα AVL δένδρα, που σχηματίζονται από τα παρακάτω στοιχεία:

   α. 5, 8, 15, 4, 2, 3, 9, 7, 6
   β. 1, 2, 3, 4, 5, 6, 7, 8, 9

2. Γράψτε μία διαδικασία, σε Pascal, για την εισαγωγή στοιχείου σε AVL δένδρο.

3. Κατασκευάστε το B_δένδρο βαθμού 3, που σχηματίζεται με συνεχείς εισαγωγές των στοιχείων:
   12, 100, 81, 19, 5, 6, 75, 16, 66, 110

4. Κατασκευάστε το B^+ δένδρο από το B_δένδρο της προηγούμενης άσκησης.

5. Υλοποιήστε τον αλγόριθμο της διαγραφής ενός στοιχείου απο Β_δένδρο σε Pascal.

6. Βρείτε μια σχέση μεταξύ του ύψους ενός Β_δένδρου βαθμού d και του μεγέθους του n. Υποθέστε ότι το δένδρο έχει: α) μέγιστο ύψος και β) ελάχιστο ύψος.

7. Εστω ότι θέλουμε να διαγράψουμε ένα στοιχείο από έναν κόμβο q ενός Β_δένδρου κι ας υποθέσουμε ότι ο πλησιέστερος δεξιός γείτονας περιέχει ακριβώς $\lceil m/2 \rceil$ στοιχεία. Αυτό σημαίνει πως δεν μπορεί να γίνει δανεισμός από το γείτονα αυτόν. Μπορούμε να κάνουμε μόνο συνένωση. Ωστόσο όμως ένας γειτονικός κόμβος του q από αριστερά περιέχει περισσότερα από $\lceil m/2 \rceil$ στοιχεία. Τι θα κάνετε, συγχώνευση ή δανεισμό;

8. Βελτιώστε τον αλγόριθμο της άσκησης 5 παραπάνω σύμφωνα με την άσκηση 7.

9. Εξηγήστε γιατί το δένδρο του σχήματος 5.11, που προκύπτει μετά τη διαγραφή του στοιχείου 6, είναι πράγματι Β_δένδρο.

10. Δείξτε ότι όλα τα δένδρα βαθμού δύο είναι πλήρη δυαδικά δένδρα.

ΚΕΦΑΛΑΙΟ 6

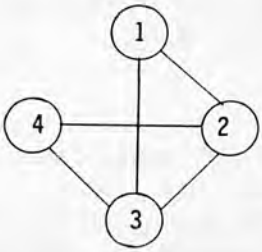
ΓΡΑΦΗΜΑΤΑ

6.1 Ορισμοί

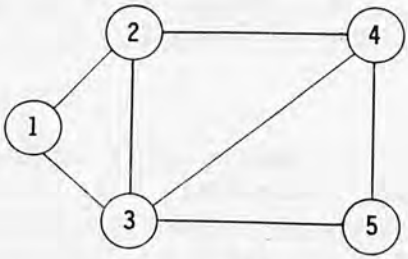
Στις δομές που περιγράψαμε μέχρι τώρα, υπάρχει μια σχέση προτεραιότητας (προηγούμενος-επόμενος για λίστες) ή ιεραρχίας (πατέρας-παιδιά για δένδρα) μεταξύ των στοιχείων τους. Ενα γράφημα είναι μια γενικευμένη δομή στην οποία κάθε στοιχείο μπορεί να συνδέεται με οποιοδήποτε άλλο στοιχείο της δομής.

Ορισμός: Γράφημα (graph) είναι μια δομή που αποτελείται από δύο σύνολα V και E, όπου V είναι ένα πεπερασμένο $\neq \emptyset$ σύνολο, **κόμβων** ή **κορυφών (vertices)** και E είναι ένα σύνολο από ζεύγη κόμβων που ονομάζονται **ακμές (edges)**.

Ενα γράφημα θα παριστάνεται με G=(V,E) και τα σύνολα των κόμβων του και των ακμών του με V(G) και E(G) αντίστοιχα. Στο σχήμα 6.1 δίνονται παραδείγματα γραφημάτων:



(α)



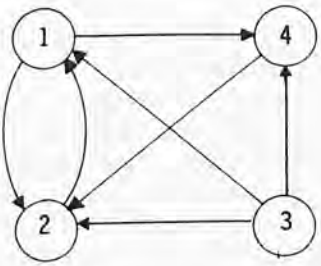
(β)

V = {1,2,3,4}
E = {(1,2)(2,3)(1,3)(2,4)(3,4)}

V = {1,2,3,4,5}
E = {(1,2)(1,3)(2,3)(2,4)(3,4)
(4,5)(3,5)}

Σχήμα 6.1

Ενα **γράφημα** ονομάζεται **μη διευθυνόμενο (undirected graph)** όταν τα ζεύγη των κόμβων, που παριστάνουν τις ακμές του, δεν είναι διατεταγμένα, δηλαδή όταν τα (v_i,v_j) και (v_j,v_i) παριστάνουν την ίδια ακμή. Αντίθετα, ένα γράφημα ονομάζεται **διευθυνόμενο (directed graph)** όταν τα ζεύγη που παριστάνουν τις ακμές του είναι διατεταγμένα. Αν $(v_i,v_j) \in E(G)$ και G είναι ένα διευθυνόμενο γράφημα, τότε (v_i,v_j) δηλώνει αφ'ενός ότι οι κόμβοι v_i και v_j συνδέονται και επιπλέον τη φορά σύνδεσης που είναι από τον κόμβο v_i προς τον κόμβο v_j. Ο κόμβος v_i ονομάζεται **ουρά (tail)** της ακμής, ενώ ο κόμβος v_j **κεφαλή (head)**. Για να ξεχωρίσουμε τις δύο περιπτώσεις θα συμβολίζουμε την ακμή ενός διευθυνόμενου γραφήματος με $<v_i,v_j>$ και σχηματικά θα την παριστάνουμε μ'ένα βέλος όπως φαίνεται στο σχήμα 6.2.



$V(G) = \{1,2,3,4\}$
$E(G) = \{<1,2>,<2,1>,<1,4>,<3,4>,<3,2>,<3,1>,<4,2>\}$

Σχήμα 6.2

Σ'ένα γράφημα με n κόμβους το πλήθος των διαφορετικών μη διατεταγμένων ζευγών (v_i, v_j) με $v_i \neq v_j$ είναι ίσο με:

$$n(n-1)/2$$

Ενα γράφημα με n κόμβους και $n(n-1)/2$ ακμές ονομάζεται **πλήρες (complete)**. Το μέγιστο πλήθος των ακμών ενός διευθυνόμενου γραφήματος είναι $n(n-1)$. Οταν $(v_i,v_j) \in E(G)$, τότε λέμε ότι οι κόμβοι v_i, v_j είναι **διπλανοί ή γειτονικοί (adjacent)** και η ακμή (v_i,v_j) θα ονομάζεται **περιστατικό (incident)** των κόμβων v_i, v_j. Οταν $<v_i, v_j>$ είναι μια διατεταγμένη ακμή, τότε λέμε ότι ο κόμβος v_i είναι **γειτονικός στον (adjacent to)** v_j, ενώ ο v_j θα ονομάζεται **γειτονικός από τον (adjacent from)** v_i.

Ενα γράφημα G' θα ονομάζεται **υπογράφημα (subgraph)** ενός γραφήματος G, όταν ισχύουν οι σχέσεις: $V(G') \subseteq V(G)$ και $E(G') \subseteq E(G)$.

Οταν $v_p, v_q \in V(G)$, τότε το **μονοπάτι (path)** που συνδέει τους κόμβους αυτούς είναι μια ακολουθία από κόμβους v_p, v_{i1}, v_{i2},..., v_{in}, v_q τέτοια ώστε οι ακμές (v_p, v_{i1}) $(v_{i1}, v_{i2}),\ldots,(v_{in}, v_q)$ ανήκουν στο σύνολο E(G). Οταν το γράφημα είναι διευθυνόμενο, τότε το μονοπάτι αποτελείται από τις διατεταγμένες ακμές $\langle v_p, v_{i1}\rangle$, $\langle v_{i2}, v_{i3}\rangle$ ..., $\langle v_{in}, v_q\rangle$. Το **μήκος** ενός **μονοπατιού (path length)** είναι το πλήθος των ακμών από τις οποίες αυτό αποτελείται. Ενα **μονοπάτι** ονομάζεται **απλό (simple path)** όταν όλοι οι κόμβοι του, εκτός ίσως από τον πρώτο και τον τελευταίο, είναι διαφορετικοί. Ενας **κύκλος (cycle)** είναι ένα απλό μονοπάτι στο οποίο ο πρώτος και ο τελευταίος κόμβος ταυτίζονται.

Από τους παραπάνω ορισμούς είναι φανερό ότι ένα δένδρο είναι ένα γράφημα χωρίς κύκλους **(ακυκλικό γράφημα, acyclic graph)**.

Δύο κόμβοι ενός γραφήματος ονομάζονται **συνδεδεμένοι (connected)**, αν υπάρχει τουλάχιστον ένα μονοπάτι που τους ενώνει. Ενα γράφημα ονομάζεται **συνδεδεμένο**, όταν για κάθε ζεύγος κόμβων v_i, $v_j \in V(G)$, τέτοιο ώστε $v_i \neq v_j$, υπάρχει μονοπάτι που τους ενώνει.

Ενα διευθυνόμενο γράφημα G είναι **ισχυρά συνδεδεμένο (strongly connected)**, όταν για κάθε ζεύγος v_i, $v_j \in V(G)$, $v_i \neq v_j$, υπάρχει ένα διευθυνόμενο μονοπάτι από το v_i στο v_j και αντίστροφα.

Ο **βαθμός** ενός **κόμβου (node degree)** είναι το πλήθος των ακμών που είναι περιστατικά του κόμβου. Οταν το γράφημα είναι διευθυνόμενο, τότε για κάθε κόμβο έχουμε το **μέσα-βαθμό (in-degree)**, που είναι το πλήθος των ακμών που φτάνουν στον κόμβο, και τον **έξω-βαθμό (out-degree)**, που είναι το πλήθος των ακμών που ξεκινάνε από τον κόμβο αυτόν προς άλλους κόμβους.

Αν d_i είναι ο βαθμός ενός κόμβου v_i ενός γραφήματος G με n κόμβους, τότε το πλήθος των ακμών (e) του γραφήματος είναι ίσο με:

$$e = \frac{1}{2} \sum_{i=1}^{n} d_i$$

Στην ξένη βιβλιογραφία ένα διευθυνόμενο γράφημα αναφέρεται ως **digraph** (από τις λέξεις **d**irected **graph**).

6.2. Υλοποίηση γραφήματος με πίνακα

Οταν το πλήθος των ακμών ενός γραφήματος είναι πολύ μεγάλο, τότε η υλοποίηση του γραφήματος γίνεται με χρήση πινάκων. Η υλοποίηση με πίνακες είναι σχετικά εύκολη και, όπως θα δούμε

παρακάτω, ορισμένες αλγεβρικές πράξεις επί των πινάκων μας δίνουν διάφορα χαρακτηριστικά του γραφήματος. Για να παραστήσουμε ένα δεδομένο διευθυνόμενο γράφημα G=(V, E) μ'έναν πίνακα, θα πρέπει να θεωρήσουμε μια μορφή διάταξης μεταξύ των κόμβων του γραφήματος. Για το λόγο αυτό η παράσταση του γραφήματος θα εξαρτάται από τη διάταξη αυτή.

Εστω ότι το σύνολο των κόμβων $V(G) = \{v_1, \ldots, v_n\}$ ενός γραφήματος G είναι ταξινομημένο (από v_1 μέχρι v_n). Τότε ο πίνακας A με στοιχεία a_{ij}, τύπου boolean, τέτοια ώστε:

$$a_{ij} = \begin{cases} 1 \text{ αν } (v_i, v_j) \in E(G) \\ 0 \text{ διαφορετικά} \end{cases}$$

ονομάζεται **μήτρα γειτνίασης (adjacency matrix)** του γραφήματος. Για παράδειγμα η μήτρα γειτνίασης του γραφήματος του σχήματος 6.2 παριστάνεται με τον εξής πίνακα:

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	1	0	0	0
v_3	1	1	0	1
v_4	0	1	0	0

Οταν το γράφημα είναι μη διευθυνόμενο, τότε ο πίνακας A που παριστάνει τη μήτρα γειτνίασης είναι συμμετρικός. Στην περίπτωση αυτή αρκούν n(n-1)/2 δυαδικά ψηφία για την παράστασή του.

Σε κάθε γραμμή i του πίνακα γειτνίασης ενός γραφήματος, η θέση της κάθε μονάδας καθορίζει μιαν ακμή, που ξεκινάει από τον κόμβο v_i. Συνεπώς το άθροισμα των στοιχείων της i-γραμμής μας δίνει τον έξω-βαθμό του κόμβου v_i:

$$d_{out}(v_i) = \sum_{j=1}^{n} a_{ij}$$

Με τον ίδιο τρόπο το άθροισμα των στοιχείων της j-στήλης του πίνακα γειτνίασης μας δίνει το μέσα-βαθμό (in-degree) του κόμβου v_j:

$$d_{in}(v_j) = \sum_{i=1}^{n} a_{ij}$$

Ο πίνακας γειτνίασης ενός γραφήματος δεν είναι μονοσήμαντα ορισμένος, αλλά εξαρτάται από τη διάταξη που δίνουμε στους κόμβους του γραφήματος. Όταν έχουμε δύο γραφήματα κι ο πίνακας γειτνίασης του ενός προκύπτει από τον πίνακα γειτνίασης του άλλου, μετά εναλλαγή των γραμμών ή στηλών, τότε λέμε ότι τα γραφήματα είναι **ισοδύναμα (equivalent)**. Οπως αναφέραμε, η ύπαρξη μονάδας στη θέση (i,j) του πίνακα γειτνίασης A δηλώνει την ύπαρξη της ακμής (v_i, v_j), δηλώνει δηλαδή την ύπαρξη ενός μονοπατιού μήκους 1 από τον κόμβο v_i στον κόμβο v_j. Ας θεωρήσουμε τον πίνακα A^2 όπου:

$$a_{ij}^{(2)} = \sum_{k=1}^{n} a_{ik}a_{kj}.$$

Είναι φανερό ότι συνεισφορά στο άθροισμα έχει κάθε όρος για τον οποίον ισχύει $a_{ik}*a_{kj}=1$, για κάποιο σταθερό k. Η σχέση αυτή ισχύει όταν $a_{ik}=1$ και $a_{kj}=1$, όταν δηλαδή $(v_i,v_k) \in E(G)$ και $(v_k,v_j) \in E(G)$, που σημαίνει ότι υπάρχει ένα μονοπάτι μήκους 2 από τον v_i στον v_j. Κατά συνέπεια το $a_{ij}^{(2)}$ ισούται με το πλήθος όλων των διαφορετικών μονοπατιών μήκους 2 από τον κόμβο v_i στον κόμβο v_j. Τα διαγώνια στοιχεία $a_{ii}^{(2)}$ του πίνακα A^2 μας δίνουν το πλήθος των κύκλων μήκους 2 για τον κόμβο v_i. Τα παραπάνω γενικεύονται ως εξής:

Αν A είναι ο πίνακας γειτνίασης ενός γραφήματος G, τότε το στοιχείο $a_{ij}^{(n)}$ του A^n $(n > 1)$ μας δίνει το πλήθος των μονοπατιών μήκους n από τον κόμβο i στον κόμβο j. Από τον πίνακα A μπορούμε συνεπώς να ελέγξουμε αν υπάρχει ακμή από τον κόμβο v_i στον v_j και από τον πίνακα A^r, $r>1$ αν υπάρχει μονοπάτι μήκους r από τον v_i στον v_j. Από τα παραπάνω προκύπτει ότι ο πίνακας:

$$B_p = A + A^2 + \ldots + A^p$$

μας δίνει το πλήθος των μονοπατιών μήκους $\leq p$, από τον κόμβο v_i στον v_j. Ετσι, όταν θέλουμε να εξακριβώσουμε αν μπορούμε να φτάσουμε από τον κόμβο v_i στον κόμβο v_j, θα πρέπει να υπολογίσουμε δυνάμεις του πίνακα A. Η μέθοδος όμως αυτή ούτε πρακτική είναι, αλλά, όπως θα δούμε, ούτε αναγκαία.

Σ'ένα διευθυνόμενο γράφημα με n κόμβους ένα απλό μονοπάτι ή ένας κύκλος δεν μπορεί να έχει μήκος μεγαλύτερο του n. Αν θέλουμε να εξετάσουμε αν υπάρχει μονοπάτι από τον κόμβο v_i στον κόμβο v_j, αρκεί να εξετάσουμε όλα τα απλά μονοπάτια μήκους $\leq n-1$. Στην περίπτωση που $v_i = v_j$, αρκεί να εξετάσουμε όλους τους απλούς κύκλους μήκους $\leq n$. Τα μονοπάτια αυτά ή οι κύκλοι καθορίζονται από τον πίνακα:

$$B_n = A + A^2 + \ldots + A^n$$

έτσι ώστε, όταν $b_{ij}^{(n)} \neq 0$, τότε υπάρχει κάποιο μονοπάτι από τον v_i στον v_j.

Όταν δε μας ενδιαφέρει το πλήθος των μονοπατιών από τον κόμβο v_i στον κόμβο v_j, αλλά απλώς θέλουμε να ξέρουμε αν μπορούμε να φτάσουμε στον κόμβο v_j, ξεκινώντας από τον v_i, τότε αρκεί να δείξουμε ότι υπάρχει ένα τέτοιο μονοπάτι. Έτσι για ένα γράφημα G=(V,E) ορίζουμε τον **πίνακα μονοπατιών** P ως εξής:

$$P_{ij} = \begin{cases} 1 \text{ όταν υπάρχει μονοπάτι από τον } v_i \text{ στον } v_j \\ 0 \text{ διαφορετικά.} \end{cases}$$

Ο πίνακας P δείχνει την ύπαρξη ή όχι **ενός τουλάχιστον** μονοπατιού από τον v_i στον v_j. Συνεπώς ο πίνακας αυτός μπορεί να υπολογισθεί από τον πίνακα B_n ως εξής:

$$P_{ij} = \begin{cases} 1 \text{ όταν } b^{(n)}_{ij} \neq 0 \\ 0 \text{ όταν } b^{(n)}_{ij} = 0 \end{cases}$$

Ο πίνακας P μπορεί να υπολογιστεί από τον πίνακα B_{n-1}, γιατί ένα μονοπάτι μήκους n δεν μπορεί να είναι απλό. Η μόνη διαφορά του P, όταν υπολογισθεί από τον B_{n-1} αντί από τον B_n, βρίσκεται στα διαγώνια στοιχεία του. Μπορεί όμως να υποτεθεί ότι κάθε κόμβος συνδέεται με τον εαυτό του.

Ο τρόπος υπολογισμού του P από τους A, $A^2, \ldots, A^n$ δεν είναι αποτελεσματικός. Θα περιγράψουμε εδώ έναν άλλο τρόπο υπολογισμού του P, που βασίζεται στην ίδια ιδέα, αλλά είναι πιο αποτελεσματικός στην πράξη. Για το σκοπό αυτό θα χρειαστεί να κάνουμε πράξεις με πίνακες τύπου boolean. Αν A και B είναι δύο n x n πίνακες τύπου boolean, τότε το άθροισμά τους, A V B, είναι ένας πίνακας C τύπου boolean τέτοιος ώστε:

$$c_{ij} = a_{ij} \vee b_{ij}$$

Το γινόμενο, A Λ B, των δύο πινάκων είναι επίσης ένας πίνακας D τύπου boolean, τέτοιος ώστε:

$$d_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}$$

Ισχύει ότι:

$$d_{ij} = \begin{cases} 1 \text{ όταν } a_{ik}=1 \text{ και } b_{kj}=1, \text{ για κάποιο } k \\ 0 \text{ διαφορετικά.} \end{cases}$$

Οι δυνάμεις ενός πίνακα A τύπου boolean ορίζονται από τον αναδρομικό τύπο:

$$A^{(r)} = A \wedge A^{(r-1)}$$

Η διαφορά μεταξύ των A^r και $A^{(r)}$ είναι ότι ο $A^{(r)}$ είναι ένας λογικός πίνακας, τέτοιος ώστε, όταν $a_{ij}^{(r)}=1$, να υπάρχει τουλάχιστον ένα μονοπάτι μήκους r από τον κόμβο v_i στον κόμβο v_j, ενώ το a_{ij}^{r} δίνει το πλήθος των μονοπατιών μήκους r από τον v_i στον v_j. Ο πίνακας P δίνεται από τη σχέση:

$$P = A \vee A^{(2)} \vee \ldots \vee A^{(n)} = \bigvee_{k=1}^{n} A^{(k)}$$

Για το γράφημα του σχήματος 6.2 έχουμε:

$$P = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Η μέθοδος υπολογισμού του πίνακα P για ένα διευθυνόμενο γράφημα περιγράφεται με τον αλγόριθμο του Warshall.

Αν A είναι ο πίνακας γειτνίασης ενός γραφήματος τότε ο **αλγόριθμος του Warshall** περιγράφεται από τη διαδικασία:

```
procedure warshall (var A, P: πινακας; N: integer);
var i, j, k:  integer;
begin
  for i:= 1 to N do
  for j:= 1 to N do
   P[i, j]:= A [i, j];
  for k:= 1 to N do
    for i:= 1 to N do
      for j:= 1 to N do
       P[i, j]:= P[i, j] or (P[i, k] and P[k, j])
end (warshall).
```

Ο πίνακας που υπολογίζεται από τις εντολές for-i και for-j, για κάποιο σταθερό k, έχει στοιχεία P[i, j] ίσα με 1, αν και μόνο αν υπάρχει μια ακμή από τον κόμβο v_i στον κόμβο v_j ή αν υπάρχει

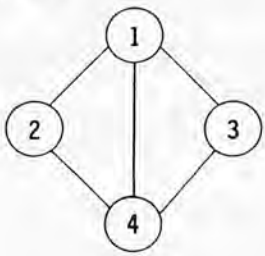
μονοπάτι από τον v_1 στον v_j διαμέσου των κορυφών $v_1, \ldots, v_k$. Με τον ίδιο τρόπο, για την επόμενη τιμή του k ισχύει P[i, j] = 1, ή όταν P[i, j] = 1 από προηγούμενο βήμα, ή όταν υπάρχει μονοπάτι από τον v_i στον v_j δια μέσου των κορυφών $v_1, \ldots, v_k, v_{k+1}$. Αυτό τελικά σημαίνει ότι P[i, j] = 1 όταν υπάρχει μονοπάτι από τον v_i στον v_j.

6.3 Υλοποίηση γραφήματος με λίστες

Όλοι οι αλγόριθμοι, που είναι σχετικοί με γραφήματα, και που χρησιμοποιούν τον πίνακα γειτνίασης, έχουν πολυπλοκότητα $O(|V|^2)$, όπου $|..|$ συμβολίζει τον πληθικό αριθμό ενός συνόλου. Όταν το γράφημα είναι αραιό, τότε τα περισσότερα στοιχεία του πίνακα γειτνίασης είναι μηδέν και μπορούν να αγνοηθούν. Στην περίπτωση αυτή η πολυπλοκότητα των αλγορίθμων θα είναι $O(|E| + |V|)$, όπου $|E| \leq |V|^2/2$. Η πολυπλοκότητα αυτή μπορεί να επιτευχθεί με τη χρήση συνδεδεμένων λιστών που περιέχουν μόνο εκείνες τις ακμές που ανήκουν στο γράφημα. Έτσι έχουμε τις παρακάτω μεθόδους υλοποίησης ενός γραφήματος με λίστες.

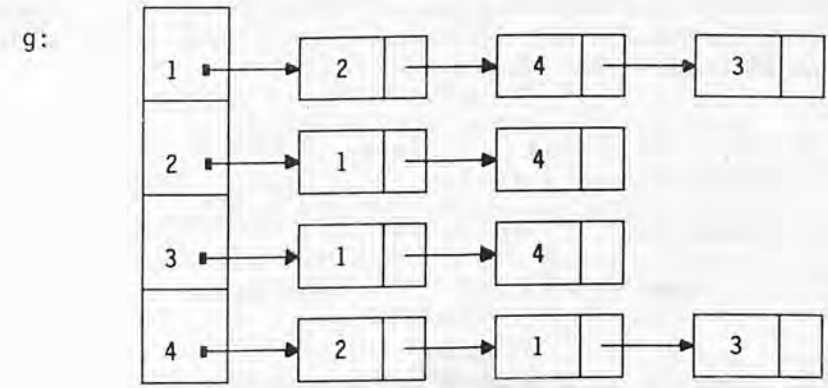
Μέθοδος 1

Στη μέθοδο αυτή οι γραμμές του πίνακα γειτνίασης αντικαθιστούνται με απλές γραμμικές λίστες μιάς διεύθυνσης. Στην κάθε κορυφή του γραφήματος αντιστοιχεί μια λίστα που περιέχει όλους τους γειτονικούς κόμβους της κορυφής. Εστω, για παράδειγμα, το γράφημα του σχήματος 6.3:



Σχήμα 6.3

Για την υλοποίηση του γραφήματος αυτού απαριθμούμε τους κόμβους του (1,... , n) και χρησιμοποιούμε έναν πίνακα μεγέθους n, τα στοιχεία του οποίου παριστάνουν δείκτες. Ο δείκτης στη θέση i του πίνακα δείχνει στη λίστα που περιέχει όλους τους γειτονικούς κόμβους του i, όπως φαίνεται στο σχήμα 6.4.



Σχήμα 6.4

Ο ορισμός ενός τέτοιου γραφήματος σε Pascal θα είναι:

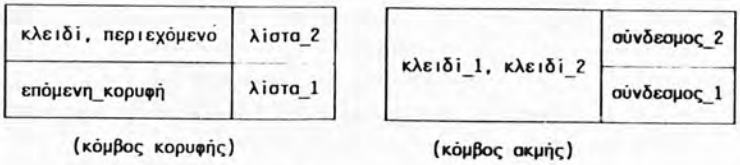
```
type τύπος_κλειδιού = integer;
     δείκτης_λίστας = ^ κόμβος_λίστας;
     κόμβος_λίστας = record
                       κλειδί: τύπος_κλειδιού;
                       σύνδεσμος: δείκτης_λίστας
                     end;
     var g:  array [1.. n] of δείκτης_λίστας
```

Οπως παρατηρούμε κι από το σχήμα 6.4, με την μέθοδο αυτή κάθε ακμή του γραφήματος υπάρχει σε δύο λίστες. Αυτό κάνει πολύ εύκολο τον υπολογισμό των γειτόνων ενός κόμβου. Σαν αντάλλαγμα, όμως, πληρώνουμε σε χώρο, γιατί αποθηκεύουμε το διπλάσιο πλήθος ακμών του γραφήματος. Η πολυπλοκότητα σε χώρο για την αποθήκευση της δομής με τη μέθοδο αυτή είναι $O(|V| + 2|E|)$.

Μέθοδος 2

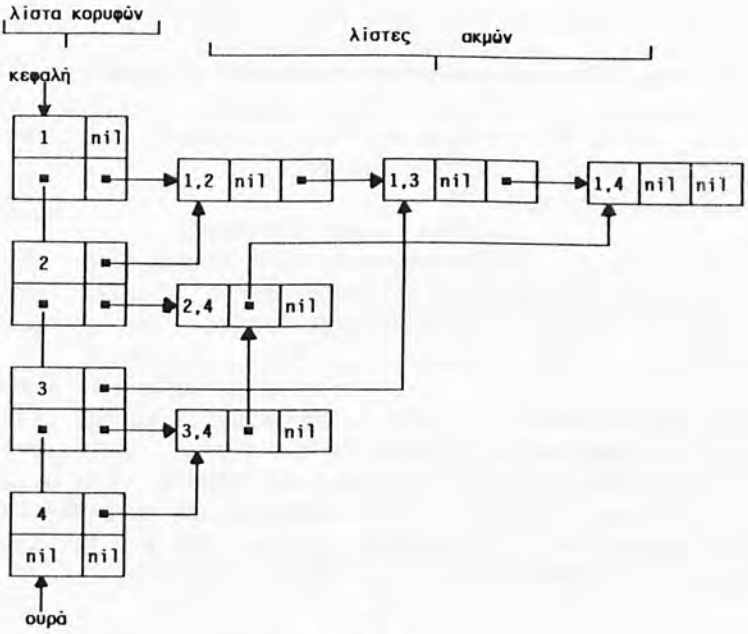
Στη μέθοδο αυτή κάθε ακμή του γραφήματος παριστάνεται με ένα μόνο κόμβο ακμής. Για το σκοπό αυτό χρησιμοποιούμε μια πολλαπλή λίστα. Κάθε ακμή του γραφήματος είναι κοινή σε δύο λίστες. Κάθε κορυφή του γραφήματος παριστάνεται μ'έναν κόμβο στη λίστα των κορυφών και κάθε ακμή μ'έναν κόμβο που είναι κοινός στις δύο λίστες των κορυφών, που συνδέει. Από κάθε κόμβο κορυφής ξεκινούν δύο λίστες ακμών. Στη μια λίστα το κλειδί του κόμβου κορυφής είναι το πρώτο από το ζεύγος των κλειδιών, που υπάρχουν σε κάθε

κόμβο ακμής, ενώ στην άλλη λίστα, το κλειδί του κόμβου κορυφής είναι το δεύτερο στη λίστα των ακμών. Σχηματικά οι κόμβοι κορυφών και ακμών δίνονται στο σχήμα 6.5.



Σχήμα 6.5

Η παράσταση του γραφήματος του σχήματος 6.3 δίνεται από το σχήμα 6.6.



Σχήμα 6.6

Η μέθοδος αυτή ελαχιστοποιεί την **επιβάρυνση χώρου (space overhead)** που οφείλεται στους δείκτες. Ωστόσο όμως, ορισμένοι αλγόριθμοι, όπως, για παράδειγμα, αυτός του υπολογισμού των γειτόνων ενός κόμβου, γίνονται πολύ πιο δύσκολοι.

Ο ορισμός του γραφήματος στην περίπτωση αυτή είναι:

```
type τύπος_κλειδιού = integer;
     τύπος_περιεχομένου = ... ; {ορίζεται από το χρήστη}
     τύπος_στοιχείου = record
                         κλειδί:  τύπος_κλειδιού;
                         περιεχόμενο: τύπος_περιεχομένου
                       end;
          δείκτης_κορυφής = ^ κόμβος_κορυφής;
          δείκτης_ακμής = ^ κόμβος_ακμής;

     κόμβος_κορυφής = record
                        στοιχείο:  τύπος_στοιχείου;
                        επόμενος_κόμβος:  δείκτης_κορυφής;
                        λίστα_1, λίστα_2:  δείκτης_ακμής
                      end;
     κόμβος_ακμής = record
                      κλειδί_1, κλειδί_2:  τύπος_κλειδιού;
                      σύνδεσμος_1, σύνδεσμος_2:  δείκτης_ακμής
                    end;
       γράφημα = record
                   κεφαλή, ουρά, φρουρός, τρέχουσα_κορυφή:
                                          δείκτης_κορυφής;
                   τρέχουσα_ακμή:  δείκτης_ακμής
                 end;
var   g : γράφημα
```

6.4 Πράξεις επί των γραφημάτων

1. Δημιουργία γραφήματος

```
procedure δημιούργησε (var g: γράφημα);
begin
  with g do
  begin
    κεφαλή:= nil;
    ουρά:= nil;
    τρέχουσα_κορυφή:= nil;
    τρέχουσα_ακμή:= nil;
    new (φρουρός)
  end
end {δημιούργησε}.
```

2. Αναζήτηση κορυφής με δεδομένο κλειδί k

```
procedure αναζήτησε_κορυφή (k:τύπος_κλειδιού; var g:γράφημα;
                                    var βρέθηκε: boolean);
var p:  δείκτης_κορυφής;
begin
  βρέθηκε:= false;
  with g do
  begin
    if κεφαλή <> nil then
       begin
         p:= κεφαλή;
         {βάλε στον κόμβο φρουρό το κλειδί του κόμβου που
          ζητάμε}
         φρουρός ^.στοιχείο.κλειδί:= k;
         {σύνδεσε τον κόμβο φρουρό}
         ουρά^.επόμενη_κορυφή:= φρουρός;
         {αναζήτησε στη λίστα των κορυφών}
         while k <> p^.στοιχείο.κλειδί do
         p:= p^.επόμενη_κορυφή;
         {αποσύνδεσε τον κόμβο φρουρό}
         ουρά^.επόμενη_κορυφή:= nil;
         if p <> φρουρός then
          begin
            τρέχουσα_κορυφή:= p;
            βρέθηκε:= true
          end
       end
  end {with}
end {αναζήτησε_κορυφή}
```

Ο αλγόριθμος της αναζήτησης κορυφής έχει πολυπλοκότητα $O(|V|)$.

3. Αναζήτηση της ακμής που συνδέει δύο κορυφές με κλειδιά k1 και k2 αντίστοιχα.

```
procedure αναζήτησε_ακμή (k1,k2: τύπος_κλειδιού; var g:γράφημα;
                                  var βρέθηκε_ακμή: boolean);
var e: δείκτης_ακμής; βρέθηκε_κορυφή: boolean;
begin
  βρέθηκε_ακμή:= false;
  αναζήτησε_κορυφή (k1, g, βρέθηκε_κορυφή);
  if βρέθηκε_κορυφή  then
   begin
     with g do
```

```
        begin
          e:= τρέχουσα_κορυφή^.λίστα_1;
          while (e <> nil) and (not βρέθηκε_ακμή) do
          if (k1 = e^.κλειδί_1) and (k2 = e^.κλειδί_2) then
          βρέθηκε_ακμή:= true
          else
          e:= e^.σύνδεσμος_1;
          if βρέθηκε_ακμή then τρέχουσα_ακμή:= e
        end
    end
  end {αναζήτησε_ακμή}
```

Η διαδικασία αυτή ισχύει στην περίπτωση που το γράφημα είναι διευθυνόμενο. Οταν το γράφημα δεν είναι διευθυνόμενο, τότε η αναζήτηση της ακμής εξαρτάται από τη διαδικασία εισαγωγής της ακμής. Για το λόγο αυτό θα πρέπει να ψάξουμε και στη λίστα_1 της κορυφής με κλειδί k2. Η πολυπλοκότητα του αλγόριθμου αυτού είναι $O(|V| + |E|)$.

4. Εισαγωγή νέας κορυφής

Η εισαγωγή μιας νέας κορυφής γίνεται στην ουρά της λίστας των κορυφών του γραφήματος.

```
procedure εισάγαγε_κορυφή (x: τύπος_στοιχείου; var g: γράφημα);
var p: δείκτης_κορυφής;
begin
  new (p);
  with g do
  begin
    τρέχουσα_κορυφή:= p;
    with p^ do
      begin
        στοιχείο:= x;
        επόμενη_κορυφή:= nil;
        λίστα_1:= nil;
        λίστα_2:= nil
      end;  { with p^}
    if κεφαλή = nil then κεφαλή:= p
    else
    ουρά^.επόμενη_κορυφή:= p;
    ουρά:= p
  end {with g}
end {εισάγαγε_κορυφή}
```

Η πολυπλοκότητα του αλγόριθμου εισαγωγής νέας κορυφής είναι $O(1)$.

5. Εισαγωγή ακμής με κλειδιά k1 και k2

Ο αλγόριθμος της εισαγωγής ακμής σ'ένα γράφημα περιγράφεται ως εξής. Εντοπίζουμε στη λίστα των κορυφών τους κόμβους που έχουν κλειδιά τα k1 και k2, αντίστοιχα. Εφ'όσον υπάρχουν οι κόμβοι αυτοί, η εισαγωγή της ακμής γίνεται στην αρχή της λίστας_1 του κόμβου κορυφής με κλειδί το k1 και στην αρχή της λίστας_2 για τον κόμβο κορυφής με κλειδί το k2.

```
procedure  εισάγαγε_ακμή (k1, k2 : τύπος_κλειδιού;
                    var g: γράφημα; var έγινε_εισαγωγή: boolean);
var p1, p2 : δείκτης_κορυφής; e: δείκτης_ακμής;
        βρέθηκε_κορυφήk1, βρέθηκε_κορυφήk2 : boolean;
begin
  αναζήτησε_κορυφή (k1, g, βρέθηκε_κορυφήk1);
  if βρέθηκε_κορυφήk1  then
    with g do
    begin
      p1:= τρέχουσα_κορυφή;
      αναζήτησε_κορυφή (k2, g, βρέθηκε_κορυφήk2);
      if βρέθηκε_κορυφήk2  then
      begin
        p2:= τρέχουσα_κορυφή;
        έγινε_εισαγωγή:= true;
        new (e);
        with e^ do
        begin
          κλειδί_1:= k1;
          κλειδί_2:= k2;
          σύνδεσμος_1:= p1^.λίστα_1;
          p1^.λίστα_1:= e;
          σύνδεσμος_2:= p2^.λίστα_2;
          p2^.λίστα_2:= e
        end {with e^}
      end
      else έγινε_εισαγωγή:= false
    end {with g}
    else έγινε_εισαγωγή:= false
end {εισάγαγε_ακμή}
```

Ο αλγόριθμος αυτός έχει πολυπλοκότητα $O(|V|)$.

6. Διαγραφή ακμής με κλειδιά k1 και k2

Ο αλγόριθμος της διαγραφής ακμής από ένα γράφημα ισοδυναμεί με τη διαγραφή της ακμής από τις δύο λίστες στις οποίες ανήκει η ακμή. Η πολυπλοκότητα του αλγόριθμου αυτού είναι $O(|V| + |E|)$.

```
procedure διάγραψε_ακμή (k1, k2: τύπος_κλειδιού; var g: γράφημα;
                         var απέτυχε: boolean);
var r, e:  δείκτης_ακμής;
    βρέθηκε_ακμή: boolean; βρέθηκε_κορυφή : boolean;
begin
  αναζήτησε_κορυφή (k1, g, βρέθηκε_κορυφή);
  if βρέθηκε_κορυφή then
  begin
    βρέθηκε_ακμή:= false;
    with g do
    begin
      r:= nil; e:= τρέχουσα_κορυφή^.λίστα_1;
      while (e <> nil) and (not βρέθηκε_ακμή)  do
      if (e^.κλειδί_1 = k1) and (e^.κλειδί_2 = k2) then
      βρέθηκε_ακμή:= true
      else
      begin
        r:= e;
        e:= e^.σύνδεσμος_1
      end;
      if βρέθηκε_ακμή  then
      begin
        if r = nil then τρέχουσα_κορυφή^.λίστα_1:= e^.σύνδεσμος_1
        else
        r^.σύνδεσμος_1:= e^.σύνδεσμος_1;
        αναζήτησε_κορυφή (k2, g, βρέθηκε_κορυφή);
        if βρέθηκε_κορυφή then
        begin
          r:= nil; βρέθηκε_ακμή:= false;
          e:= τρέχουσα_κορυφή^.λίστα_2;
          while (e <> nil) and (not βρέθηκε_ακμή) do
          if (e^.κλειδί_1 = k1) and (e^.κλειδί_2 =k2) then
          βρέθηκε_ακμή:= true
          else
          begin
            r:=e;
            e:= e^.σύνδεσμος_2
          end;
          if r=nil then τρέχουσα_κορυφή^.λίστα_2:= e^.σύνδεσμος_2
```

```
          else r^.σύνδεσμος_2:= e^.σύνδεσμος_2
        end {if βρέθηκε κορυφή2};
        dispose (e)
      end  {if βρέθηκε ακμή}
      else απέτυχε:= true
    end {with g}
  end {if βρέθηκε_κορυφή1}
end {διάγραψε_ακμή}
```

7. Διαγραφή κορυφής που έχει κλειδί k

Ο αλγόριθμος της διαγραφής από ένα γράφημα της κορυφής με κλειδί k περιγράφεται ως εξής. Εντοπίζουμε την κορυφή που έχει κλειδί k και διαγράφουμε τις λίστες, λίστα_1 και λίστα_2, που ξεκινούν από την κορυφή αυτή. Τέλος διαγράφουμε την κορυφή που έχει κλειδί k, κάνοντας τις κατάλληλες συνδέσεις. Ο αλγόριθμος αυτός έχει πολυπλοκότητα $O(|V| + |E|)$.

```
procedure διάγραψε_κορυφή (k: τύπος_κλειδιού; var g: γράφημα;
                                  var απέτυχε: boolean);
var q, p: δείκτης_κορυφής;
e: δείκτης_ακμής; fail: boolean;
begin
  with g do
  begin
    if κεφαλή <> nil then
    begin
      απέτυχε:= false;
      p:= κεφαλή; q:= nil;
      {εισάγαγε την προς διαγραφήν κορυφή στον κόμβο φρουρό}
      φρουρός^. στοιχείο. κλειδί:= k;
      ουρά^. επόμενη_κορυφή:= φρουρός;
      {αναζήτησε την κορυφή με κλειδί k}
      while p^.στοιχείο.κλειδί <> k do
      begin
        q:= p;
        p:= p^.επόμενη_κορυφή
      end;
      {αποσύνδεσε τον κόμβο φρουρό}.
      ουρά^.επόμενη_κορυφή:= nil;
      if p <> φρουρός then
      begin
        e:= p^.λίστα_1;
        while e <> nil do
        begin
```

```
            διάγραψε_ακμή (e^.κλειδί_1,
                           e^.κλειδί_2, g, fail);
            e:= e^.σύνδεσμος_1
          end;
          e:= p^.λίστα_2;
          while e <> nil do
          begin
            διάγραψε_ακμή (e^.κλειδί_1,
                           e^.κλειδί_2, g, fail);
            e:= e^.σύνδεσμος_2
          end;
          if q = nil then
          {διάγραψε την  πρώτη κορυφή}
          κεφαλή:= κεφαλή^.επόμενη_κορυφή
          else
          q^.επόμενη_κορυφή:= p^.επόμενη_κορυφή;
          if ουρά = p then ουρά:= q;
          dispose (p)
          end {if p <> φρουρός}
          else
          απέτυχε:= true
      end {if κεφαλή <> nil}
      else
      απέτυχε:= true
    end {with g}
end {διάγραψε_κορυφή}
```

6.5 Διάσχιση γραφήματος

Διάσχιση ενός γραφήματος σημαίνει τη συστηματική επίσκεψη όλων των κορυφών του. Οι τεχνικές διάσχισης, που συνήθως χρησιμοποιούνται, αποτελούν γενίκευση των αντίστοιχων μεθόδων που χρησιμοποιούνται στα δένδρα και αποτελούν, όπως και στα δένδρα, ένα ισχυρότατο εργαλείο για την ανάπτυξη πολλών αλγορίθμων. Στα επόμενα θα περιγράψουμε δύο βασικούς αλγόριθμους που βασίζονται στις:

α) **αναζήτηση πλάτους πρώτα (breadth first search, bfs)** και
β) **αναζήτηση βάθους πρώτα (depth first search, dfs).**

Για την ευκολία περιγραφής των αλγορίθμων θα εισαγάγουμε για τον κάθε κόμβο κορυφής την έννοια της **κατάστασης (status)**. Κάθε κόμβος μπορεί να βρίσκεται ή σε κατάσταση αναμονής, ή να είναι έτοιμος για επεξεργασία, ή να έχει ήδη τύχει επεξεργασίας. Ορίζουμε λοιπόν τον τύπο:

```
type κατάσταση = (αναμονή, έτοιμος, επεξεργάστηκε).
```

6.5.1 Αναζήτηση κατά πλάτος πρώτα

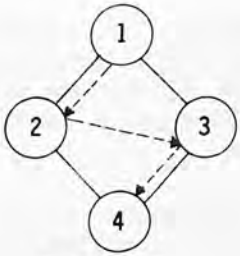
Στην μέθοδο αυτή επισκεπτόμαστε και επεξεργαζόμαστε όλους τους γειτονικούς κόμβους w_1, w_2,... , w_k ενός ορισμένου κόμβου w, πριν επισκεφθούμε τους γειτόνους των w_i i = 1,... , k.

Αρχικά όλοι οι κόμβοι του γραφήματος βρίσκονται σε κατάσταση αναμονής. Αυτό πραγματοποιείται με τη διαδικασία "δώσε_αρχικές_τιμές". Κάθε κόμβος που έχει τύχει επίσκεψης σημειώνεται ότι έχει υποστεί επεξεργασία και προετοιμάζονται για επίσκεψη όλοι οι γείτονες των κόμβων που βρίσκονται σε κατάσταση αναμονής. Ο αλγόριθμος χρησιμοποιεί μια ουρά στην οποία αποθηκεύονται οι κόμβοι του γραφήματος που είναι έτοιμοι για επεξεργασία. Η μέθοδος περιγράφεται με τον παρακάτω αλγόριθμο bfs.

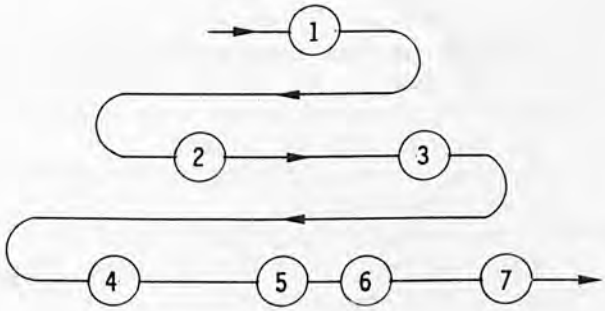
```
procedure bfs;
 procedure επίσκεψη (w: κόμβος);
 begin
   εισάγαγε_σε_ουρά (w, ουρά);
   άλλαξε την κατάσταση του w σε "έτοιμος";
   while not κενή (ουρά) do
     begin
       εξάγαγε_από_ουρά (p, ουρά);
       επεξεργάσου τον κόμβο p;
       for κάθε (p, v) ∈ E(G) do
       if  ο v είναι σε κατάσταση "αναμονής" then
       begin
         εισάγαγε_σε_ουρά (v, ουρά);
         άλλαξε την κατάσταση του v σε "έτοιμος"
       end
     end
 end;
 begin
   δώσε_αρχικές_τιμές;
   for κάθε κόμβο u ∈ V(G) do
   if (ο u είναι σε κατάσταση "αναμονής") then
   επίσκεψη (u)
 end {bfs}
```

Στον αλγόριθμο bfs πρώτα τυχαίνει επεξεργασία ένας κόμβος, μετά όλοι οι γείτονές του, μετά οι γείτονες των γειτόνων του κοκ., όπως φαίνεται από την διακεκομμένη γραμμή στο σχήμα 6.7 .

Αν το γράφημα ήταν ένα δένδρο, τότε η επίσκεψη των κόμβων του γίνεται όπως φαίνεται στο σχήμα 6.8



Σχήμα 6.7



Σχήμα 6.8

Ο έλεγχος της κατάστασης ενός κόμβου με περιεχόμενο Κ στην διαδικασία "επισκεψη" συνεπάγεται τη διάσχιση της λίστας των κορυφών του γραφήματος, μέχρις ότου εντοπίσουμε την κορυφή με περιεχόμενο το Κ. Για ν'αποφύγουμε τις διασχίσεις αυτές προσθέτουμε σε κάθε κόμβο ακμής δύο ακόμη δείκτες, δ1 και δ2. Οι δείκτες αυτοί μας δίνουν τη δυνατότητα της απ'ευθείας προσπέλασης στους κόμβους που συνδέονται με αυτή την ακμή. Τέλος η ακεραία μεταβλητή "σειρά" χρησιμοποιείται για να δηλώσει τη σειρά με την οποία γίνεται η επισκεψη των κόμβων του γραφήματος. Ετσι οι ορισμοί των κόμβων των κορυφών και των ακμών τροποποιούνται ως εξής:

```
κόμβος_κορυφής = record
                   στοιχείο: τύπος_στοιχείου;
                   state: κατάσταση;
                   σειρά: 0..maxint;
                   επόμενη_κορυφή: δείκτης_κορυφής;
                   λίστα_1, λίστα_2: δείκτης_ακμής
                 end;
```

```
  κόμβος_ακμής = record
                   κλειδί_1, κλειδί_2: τύπος_κλειδιού;
                   δ1, δ2: δείκτης_κορυφής;
                   σύνδεσμος_1, σύνδεσμος_2: δείκτης_ακμής
                 end;

procedure bfs (g: γράφημα);
var p: δείκτης_κορυφής;  μετρητής: integer;

procedure επισκεψη (q: δείκτης_κορυφής);
var k: τύπος_κλειδιού; e: δείκτης_ακμής;
begin
  with g do
    begin
      εισάγαγε_σε_ουρά (q, ουρά); {enqueue}
      while not  κενή (ουρά) do
       begin
         εξάγαγε_από_ουρά (q, ουρά);
         μετρητής:= μετρητής + 1;
         q^.σειρά:= μετρητής;
         q^.state:= επεξεργάστηκε;
         e:= q^.λίστα_1;
         while e <> nil do
           begin
             if e^.δ2^.state = αναμονή then
                begin
                  εισάγαγε_σε_ουρά (e^.δ2, ουρά);
                  e^.δ2^.state:= έτοιμος
                end;
             e:= e^.σύνδεσμος_1
           end;
           e:= q^.λίστα_2;
           while e <> nil do
            begin
              if e^.δ1^.state = αναμονή then
               begin
                 εισάγαγε_σε_ουρά (e^.δ1, ουρά);
                 e^.δ1^.state:= έτοιμος
               end;
               e:= e^.σύνδεσμος_2
            end
       end
    end {with g}
end;
begin
  μετρητής:= 0;
```

```
    δώσε_αρχικές_τιμές;
    with g do
      begin
        p:= κεφαλή;
        while p <> nil do
         begin
           if p^.state = αναμονή then
            επισκεψη (p);
           p:= p^.επόμενη_κορυφή;
         end
      end {with g}
  end {bfs}
```

Από την παραπάνω διαδικασία bfs είναι φανερό ότι η σειρά με την οποία γίνεται η επισκεψη των κόμβων του γραφήματος εξαρτάται από τη σειρά που εμφανίζονται οι ακμές στις λίστες των ακμών. Ο παραπάνω αλγόριθμος έχει πολυπλοκότητα $O(|V||E| + |V|)$.

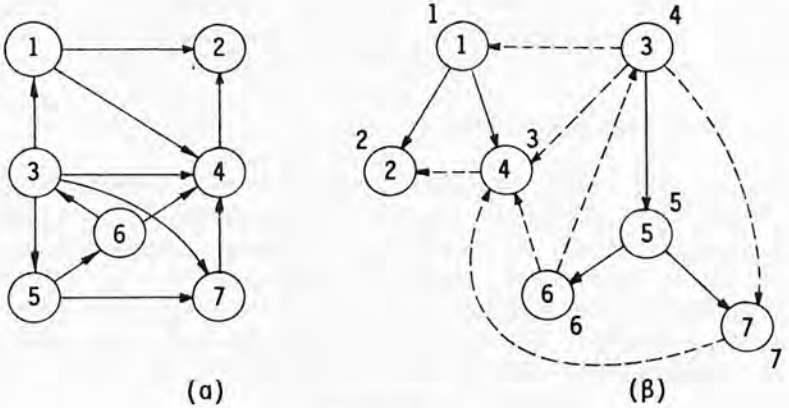
6.5.2 Αναζήτηση βάθους πρώτα

Η μέθοδος αυτή αποτελεί γενίκευση της προδιατεταγμένης διάσχισης ενός δυαδικού δένδρου. Αρχικά όλοι οι κόμβοι του γραφήματος βρίσκονται σε κατάσταση αναμονής. Κάθε κόμβος που βρίσκεται σε κατάσταση αναμονής υφίσταται με την σειρά του επεξεργασία και στη συνέχεια όλοι οι γείτονές του, που είναι στην κατάσταση αναμονής, τυχαίνουν επισκέψεως με τον ίδιο τρόπο. Η μέθοδος περιγράφεται από τον παρακάτω αλγόριθμο dfs.

```
procedure dfs;
procedure επισκεψη (w: κόμβος_κορυφής);
begin
  εισάγαγε_σε_στοίβα (w, στοίβα);
  while not κενή(στοίβα) do
  begin
    εξάγαγε_από_στοίβα (p, στοίβα);
    επεξεργάσου τον κόμβο p;
    for  κάθε (p, v) ∈ E(G)  do
    if κατάσταση του v είναι "αναμονή" then
     begin
       εισάγαγε_σε_στοίβα (v, στοίβα);
       άλλαξε την κατάσταση του v   σε "έτοιμος"
     end
  end
end; {επισκεψη}
  begin
    δώσε_αρχικές_τιμές;
```

```
        for κάθε κόμβο u ∈ V(G) do
        if (κατάσταση του u είναι αναμονή) then
        επισκεψη (u)
    end {dfs}
```

Σύμφωνα με τον αλγόριθμο αυτόν, μετά την επεξεργασία ενός κόμβου, τυχαίνει επεξεργασίας ένας από τους γειτόνους του και οι υπόλοιποι που δεν έχουν ακόμη τύχει επισκέψεως τοποθετούνται σε μια στοίβα. Στο σχήμα 6.9(β) ο ακέραιος δίπλα σε κάθε κόμβο μας δίνει τη σειρά με την οποία γίνεται η επισκεψη των κόμβων του γραφήματος του σχήματος 6.9(α)



Σχήμα 6.9

Για την υλοποίηση του αλγόριθμου θα συμβολίσουμε την κατάσταση αναμονής ενός κόμβου με την τιμή μηδέν για τη μεταβλητή "σειρά".

```
procedure dfs (g: γράφημα);
var p: δεικτης_κορυφής; μετρητής: integer;

procedure επισκεψη (q: δεικτης_κορυφής);
var e: δεικτης_ακμής; k: τύπος_κλειδιού;

begin
  with g do
  begin
    μετρητής:= μετρητής + 1;
    q^. σειρά:= μετρητής;
```

```
        e:= q^.λίστα_1;
        while e <> nil do
        begin
          if e^.δ2^.σειρά = 0 then
          επίσκεψη (e^.δ2);
          e:= e^.σύνδεσμος_1
        end;
        e:= q^.λίστα_2;
        while e <> nil do
        begin
          if e^.δ1^.σειρά = 0 then
          επίσκεψη (e^.δ1);
          e:= e^.δ2
        end
      end {with g}
    end;

    begin
      μετρητής:= 0;
      δώσε_αρχικές_τιμές; {στη μεταβλητή "σειρά" όλων των κόμβων
                                          εκχωρείται η τιμή μηδέν}
      with g do
        begin
          p:= κεφαλή;
          while p <> nil do
          if p^.σειρά = 0 then
          επίσκεψη (p)
          else
          p:= p^.επόμενη_κορυφή
        end
    end {dfs}
```

Στο σχήμα 6.9(β) οι ακμές που οδηγούν σ'έναν κόμβο, που βρίσκεται ακόμη σε κατάσταση αναμονής, σημειώνονται με μια συνεχόμενη γραμμή, ενώ όλες οι άλλες ακμές σημειώνονται με διακεκομμένη γραμμή. Οπως παρατηρούμε από το σχήμα αυτό, οι συνεχόμενες γραμμές σχηματίζουν ένα σύνολο από δένδρα. Κάθε ένα από τα δένδρα αυτά ονομάζεται **dfs-επικαλυπτικό δένδρο (dfs-spanning tree, dfst)** και οι ακμές του ονομάζονται **δενδρικές ακμές (tree arcs).** Το σύνολο των δένδρων αυτών αναφέρεται ως **dfs-επικαλυπτικό δάσος (dfs-forest).**

Οι διακεκομμένες ακμές χωρίζονται σε τρεις κατηγορίες ως εξής:

1) **πίσω ακμές (back arcs)**, ακμές από έναν κόμβο σ'ένα γνήσιο πρόγονό του στο ίδιο dfs δένδρο, για παράδειγμα η ακμή (6,3),

2) **ακμές προς τα εμπρός (forward arcs)**, ακμές από έναν κόμβο σ'ένα γνήσιο απόγονό του στο ίδιο dfs δένδρο, για παράδειγμα η ακμή (3,7) και
3) **διασταυρούμενες ακμές (cross arcs)**, ακμές από έναν κόμβο σ'έναν άλλο κόμβο ενός άλλου dfs δένδρου, για παράδειγμα οι ακμές (6,4), (7,4), (3,4).

Σημείωση: Ενας αλγόριθμος, που χρησιμοποιεί ως βοηθητική δομή μια ουρά προτεραιότητας, ονομάζεται **αλγόριθμος αναζήτησης προτεραιότητας πρώτα (priority first search, pfs).** Οταν η προτεραιότητα του κόμβου, που πρόκειται να εισαχθεί στην ουρά, είναι μικρότερη από την προτεραιότητα του κόμβου που εισήχθη τελευταία στην ουρά, τότε ο αλγόριθμος pfs ταυτίζεται με τον bfs. Στην αντίθετη περίπτωση, ο αλγόριθμος ταυτίζεται με τον dfs.

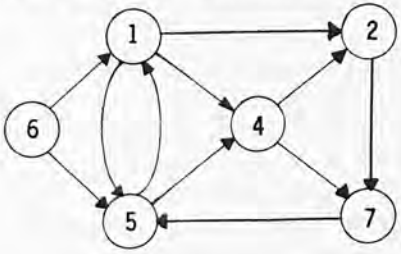
Ανακεφαλαίωση

Γράφημα	:	Μια δομή που αποτελείται από ένα πεπερασμένο μη κενό σύνολο V κορυφών και ένα σύνολο E ζευγών κόμβων, (ακμών).
Υλοποίηση γραφήματος	:	Στατική με τον πίνακα γειτνίασης και δυναμική με λίστες.
Πράξεις επί των γραφημάτων	:	Δημιουργία (creation), αναζήτηση κορυφής με δεδομένο κλειδί, αναζήτηση ακμής, εισαγωγή νέας κορυφής, εισαγωγή ακμής, διαγραφή κορυφής και διάσχιση γραφήματος (αλγόριθμοι bfs και dfs)

Ασκήσεις 6.1

1. Για το γράφημα του παρακάτω σχήματος:
   α) υπολογίστε το μέσα και τον έξω βαθμό του κάθε κόμβου,
   β) κατασκευάστε τον πίνακα γειτνίασης,
   γ) δώστε την παράσταση του γραφήματος με λίστες

δ) απαριθμήστε τα ισχυρά συνδεδεμένα στοιχεία του.



2. Γράψτε μια διαδικασία που να επιστρέφει τα κλειδιά όλων των γειτονικών κόμβων ενός δοθέντος κόμβου.

3. Περιγράψτε τη σειρά με την οποία γίνεται η επίσκεψη στους κόμβους ενός δένδρου με τους αλγόριθμους bfs και dfs.

4. Γράψτε τον αλγόριθμο dfs χρησιμοποιώντας αναδρομή, όταν το γράφημα ορίζεται με τον πίνακα γειτνίασης.

5. Γράψτε έναν αλγόριθμο που να παίρνει ως δεδομένα την παράσταση ενός γραφήματος με πίνακα και να κατασκευάζει την παράσταση του γραφήματος με λίστες.

6. Για το γράφημα της άσκησης 6.1 κατασκευάστε την παράσταση που χρησιμοποιεί λίστες.

7. Υπολογίστε όλα τα μονοπάτια μήκους 3 του γραφήματος της άσκησης 6.1.

8. Δώστε τους αλγόριθμους και το τελικό πρόγραμμα σε Pascal για τις πράξεις των γραφημάτων, όταν η υλοποίησή τους γίνεται με τη μέθοδο 1 που περιγράψαμε στο Κεφάλαιο 6.1.

9. Γράψτε ένα σύνολο (φιλικών προς το χρήστη) υποπρογραμμάτων Pascal για το διάβασμα και την επεξεργασία γραφημάτων.

10. (α) Βελτιώστε τον αλγόριθμο (5) της εισαγωγής μιας ακμής με κλειδιά k1 και k2 σ'ένα γράφημα, έτσι ώστε να ελέγχει αν η ακμή υπάρχει ήδη στο γράφημα.
    (β) Γράψτε τη διαδικασία της εισαγωγής ακμής με κλειδιά τα k1 και k2 σ'ένα μη διευθυνόμενο γράφημα.
    (γ) Γράψτε τη διαδικασία της διαγραφής ακμής με κλειδιά k1 και k2.

ΚΕΦΑΛΑΙΟ 7

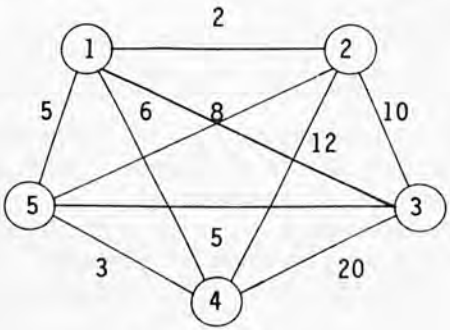
ΑΛΓΟΡΙΘΜΟΙ ΓΡΑΦΗΜΑΤΩΝ

7.1 Γενικά

Οι αλγόριθμοι όλων σχεδόν των προβλημάτων τα οποία αναφέρονται σε γραφήματα, βασίζονται στις τεχνικές διάσχισης bfs και dfs, που περιγράψαμε στο προηγούμενο κεφάλαιο. Στο κεφάλαιο αυτό θα εξετάσουμε μερικούς από τους βασικότερους αλγόριθμους που αναφέρονται σε γραφήματα ή δίκτυα και που έχουν τεράστιες εφαρμογές, όπως είναι η εύρεση του πλησιέστερου μονοπατιού μεταξύ δύο κορυφών ενός γραφήματος, ο υπολογισμός του επικαλυπτικού δένδρου ενός γραφήματος, ο υπολογισμός του κρίσιμου μονοπατιού ή κρίσιμης διαδρομής (critical path) για την αξιολόγιση μεγάλων έργων κ.α. Για εκτενέστερη μελέτη ο αναγνώστης παραπέμπεται στις αναφορές [21, 12, 7, 18] της βιβλιογραφίας που βρίσκεται στο τέλος του βιβλίου.

7.2 Επικαλυπτικό δένδρο γραφήματος (Spanning tree)

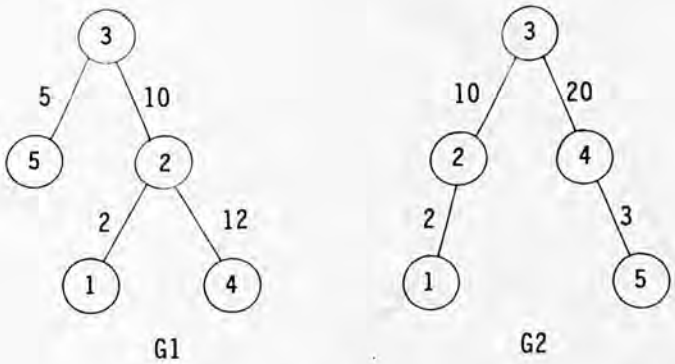
Οι κόμβοι και οι ακμές ενός γραφήματος μπορεί να περιέχουν και άλλες πληροφορίες εκτός από αυτές που είδαμε στο προηγούμενο κεφάλαιο.



Σχήμα 7.1

Για παράδειγμα, οι ακμές μπορεί να χαρακτηρίζονται από ένα βάρος (weight), που αντιστοιχεί στο κόστος μετάβασης μεταξύ των κόμβων που συνδέει. Ενα τέτοιο γράφημα φαίνεται στο σχήμα 7.1 και ονομάζεται **δίκτυο (network)** ή **γράφημα με βάρη (weighted graph)**.

Εστω G = (V, E) ένα συνδεδεμένο μη διευθυνόμενο γράφημα με βάρη. Ενα **επικαλυπτικό δένδρο (ΕΔ)**, του G είναι ένα υπογράφημα G′= (V, E′) του G, (E′⊆E) που είναι συνδεδεμένο και δεν περιέχει κύκλους. Μ'άλλα λόγια, ένα επικαλυπτικό δένδρο ενός γραφήματος G είναι ένα δένδρο που περιέχει όλους τους κόμβους του G. Στο σχήμα 7.2 δίνονται δύο επικαλυπτικά δένδρα για το γράφημα του σχήματος 7.1.



Σχήμα 7.2

Το **κόστος ενός επικαλυπτικού δένδρου** είναι το άθροισμα των βαρών των ακμών του. Ετσι το κόστος του G1 είναι 29 ενώ του G2 είναι 35.

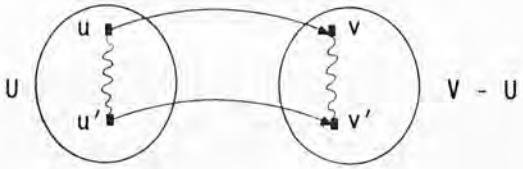
Στα επόμενα θα ασχοληθούμε με το πρόβλημα της εύρεσης του **επικαλυπτικού δένδρου με το ελάχιστο κόστος, ΕΔΕΚ, (Minimum cost spanning tree)**, ενός γραφήματος. Το πρόβλημα αυτό έχει πολλές εφαρμογές για παράδειγμα, σ'ένα δίκτυο υπολογιστών μπορεί να θέλουμε να συνδέσουμε τους υπολογιστές κατά τέτοιο τρόπο, ώστε όλοι να επικοινωνούν μεταξύ τους, είτε απ'ευθείας, είτε μέσω άλλων υπολογιστών, με τέτοιο τρόπο ώστε το κόστος του δικτύου να είναι το ελάχιστο δυνατό. Στη περίπτωση ενός τέτοιου δικτύου, τα βάρη των συνδέσεων μεταξύ δύο κόμβων-υπολογιστών αντιστοιχούν στο κόστος μετάβασης των δεδομένων (ποσότητα δεδομένων επί απόσταση). Είναι φανερό ότι ο υπολογισμός του ΕΔΕΚ παρέχει το φτηνότερο δίκτυο. Πριν περιγράψουμε τον αλγόριθμο κατασκευής ενός ΕΔΕΚ, θα αποδείξουμε ένα θεώρημα στο οποίο στηρίζεται ο αλγόριθμος αυτός.

Θεώρημα 7.1

Εστω ένα γράφημα G = (V, E) και U ένα γνήσιο υποσύνολο του V ($U \subset V$, $U \neq V$). Αν η ακμή (u, v) έχει ελάχιστο κόστος μεταξύ όλων εκείνων των ακμών για τις οποίες $u \in U$ και $v \in V-U$, τότε υπάρχει ένα ΕΔΕΚ το οποίο περιέχει την ακμή (u, v).

Απόδειξη

Η απόδειξη γίνεται με εις άτοπο επαγωγή. Ας υποθέσουμε ότι δεν υπάρχει ΕΔΕΚ το οποίο να περιέχει την ακμή (u, v).



Σχήμα 7.3

Οταν δίνεται ένα ΕΔΕΚ, έστω M, θα κατασκευάσουμε ένα άλλο, M',το οποίο θα περιέχει την ακμή (u, v). Επειδή το M είναι συνδεδεμένο, υπάρχει μία ακμή (u', v'), τέτοια ώστε $u' \in U$ και $v' \in V-U$, όπως φαίνεται στο σχήμα 7.3. Εφ'όσον το M είναι επικαλυπτικό δένδρο (ΕΔ) υπάρχει ένα μονοπάτι (διαδρομή) από τον u στον v, έστω το:

$$u \longrightarrow \ldots \longrightarrow u' \longrightarrow v' \longrightarrow \ldots \longrightarrow v$$

Για να κατασκευάσουμε ένα άλλο ΕΔ, έστω M', απομακρύνουμε την ακμή (u', v') από το M και αντ' αυτής προσθέτουμε την (u, v) στο M. Το δένδρο M' είναι συνδεδεμένο και εξ ορισμού δεν περιέχει κύκλους. Είναι συνεπώς ένα ΕΔ. Το κόστος του M' είναι μικρότερο ή ίσο του κόστους του M, γιατί το βάρος της ακμής (u, v) είναι μικρότερο ή ίσο από το βάρος της (u', v').

Αρα έχουμε κατασκευάσει ένα νέο ΕΔ με κόστος μικρότερο ή ίσο του κόστους του M. Επειδή το M είναι ένα ΕΔΕΚ, συνεπάγεται ότι και το M'είναι επίσης ένα ΕΔΕΚ. Από την αρχική μας υπόθεση αυτό είναι όμως άτοπο και συνεπώς υπάρχει ένα ΕΔΕΚ το οποίο περιέχει την ακμή (u, v).

7.3 Αλγόριθμος του Prim

Ο αλγόριθμος του Prim χρησιμοποιεί το παραπάνω θεώρημα 7.1 για την κατασκευή ενός ΕΔΕΚ. Εστω G = (V, E) ένα γράφημα και V = {1, 2,... ,n) το σύνολο των κόμβων του γραφήματος αυτού. Σαν αρχικό σύνολο U επιλέγουμε το σύνολο U = {i}, για κάποιον κόμβο i του G, $1 \leq i \leq n$. Τότε μια ακμή (u, v) με το μικρότερο βάρος, τέτοια ώστε $u \in U$ και $v \in V-U$, σύμφωνα με το θεώρημα, θα ανήκει στο ΕΔΕΚ του G. Θέτουμε λοιπόν:

$$U = U \cup \{v\}$$

και επαναλαμβάνουμε την ίδια διαδικασία. Έτσι κάθε φορά το σύνολο U περιέχει ακμές οι οποίες ανήκουν σ'ένα ΕΔΕΚ. Σε κάθε επανάληψη μια νέα ακμή προστίθεται στην τελική λύση. Ο αλγόριθμος τελειώνει όταν U = V, μετά (n - 1) επαναλήψεις. Εστω A το σύνολο των ακμών που συνδέουν τους κόμβους από το U στο V - U. Τότε για το γράφημα του σχήματος 7.1 ο αλγόριθμος έχει ως εξής:

Αρχικά υποθέτουμε ότι U = {1}, οπότε το σύνολο A περιέχει εκείνες τις ακμές που είναι περιστατικά του κόμβου 1, όπως φαίνεται στο σχήμα 7.4. Μεταξύ των ακμών αυτών η (1, 2) έχει το ελάχιστο βάρος, w = 2, και συνεπώς η ακμή αυτή προστίθεται στο ΕΔΕΚ. Έτσι ο κόμβος 2 προστίθεται στο σύνολο U και το σύνολο A κατασκευάζεται εκ νέου. Τώρα οι ακμές (1, 5) και (2, 4) έχουν το ελάχιστο βάρος, w = 5. Σε μια τέτοια περίπτωση διαλέγουμε μια από τις ακμές τυχαία, έστω την (1, 5). Η ακμή (1, 5) προστίθεται στο ΕΔΕΚ και ο κόμβος 5 εισάγεται στο σύνολο U. Η ακμή με το ελάχιστο βάρος είναι τώρα η (5, 4) με w = 3, συνεπώς η ακμή αυτή εισάγεται στο ΕΔΕΚ και ο κόμβος 4 στο U.

u	A	ΕΔΕΚ
{1}	{(1,2),(1,3),(1,4),(1,5)}	{(1,2)}
{1,2}	{(1,3),(1,4),(1,5),(2,3),(2,4),(2,5)}	{(1,2),(1,5)}
{1,2,5}	{(1,3),(1,4),(2,3),(2,4),(5,3),(5,4)}	{(1,2),(1,5),(5,4)}
{1,2,5,4}	{(1,3),(2,3),(5,3),(4,3)}	{(1,2),(1,5),(5,4),(5,3)}
{1,2,5,4,3}	{ }	

Σχήμα 7.4

Με τον ίδιο τρόπο στο επόμενο βήμα εκλέγουμε την ακμή (5,3) με w = 10. Η ακμή αυτή εισάγεται στο ΕΔΕΚ και ο κόμβος 3 στο U. Το U τώρα ταυτίζεται με το V και έτσι ο αλγόριθμος τελειώνει με

ΕΔΕΚ = ((1,2), (1,5), (5,4), (5,3)) και κόστος 20.

Από την περιγραφή του αλγορίθμου παρατηρούμε ότι το δένδρο που κατασκευάζεται δεν είναι μοναδικό.

Αλγόριθμος του Prim

```
ΕΔΕΚ = [];
U = [1];
A = [(u,v): (u,v) ∈ E(G) και u = 1 η v = 1];
for i:= 1 to n - 1 do
 begin
   {εκλέγουμε από το A την ακμή a με το ελάχιστο κόστος}
   εξάγαγε (a,A);
   εισάγαγε (a,ΕΔΕΚ);
   εισάγαγε (v,U);
   κατασκεύασε νέο A
 end
```

Μια κατάλληλη δομή για την υλοποίηση του συνόλου A είναι ο σωρός. Επειδή το σύνολο A δεν μπορεί να περιέχει περισσότερες από $|E|$ ακμές, η διαδικασία της εξαγωγής ενός στοιχείου από το A απαιτεί χρόνο $O(\log_2|E|)$. Όταν εξάγουμε από το A μια ακμή (u, v), τότε η διαδικασία "κατασκεύασε νέο A" σημαίνει την απομάκρυνση από το A όλων των ακμών (v, x), όπου $x \in U$, και την εισαγωγή στο A των ακμών (v, y), όπου $y \in V\text{-}U$. Τέτοιες πράξεις δεν υποστηρίζονται όμως αποτελεσματικά από το σωρό. Μια εναλλακτική λύση είναι η υλοποίηση του A με μια γραμμική λίστα. Στη περίπτωση αυτή οι παραπάνω διαδικασίες απαιτούν χρόνο $O(|V||E|) \approx O(|V|^3)$.

Μια άλλη δομή για την υλοποίηση του συνόλου A, που βελτιώνει την πολυπλοκότητα του αλγόριθμου ως προς το χρόνο, είναι ο πίνακας. Στον πίνακα αποθηκεύονται κάθε φορά μόνο οι ακμές (i, j) για τις οποίες, για κάποιο $i \in V\text{-}U$, η ακμή (i, j) έχει το μικρότερο κόστος μεταξύ των ακμών (i,k), για όλα τα $k \in U$. Δηλαδή, μόνο οι ακμές με το μικρότερο βάρος που αντιστοιχούν σε κάθε κόμβο i αποθηκεύονται στο σύνολο A. Αυτό σημαίνει ότι το σύνολο A δεν μπορεί να περιέχει περισσότερες από $|V| = n$ ακμές. Έτσι το σύνολο A υλοποιείται με έναν πίνακα:

```
τύπος_στοιχείων = record
                    i: integer;
                    w: real
                  end;
A: array [1..n] of τύπος_στοιχείων
```

Το στοιχείο A[i] = (j, w), όπου $i \in V\text{-}U$ και $j \in U$ δηλώνει την ακμή (i, j) με το μικρότερο βάρος w, δηλαδή ο κόμβος $j \in U$ είναι ο πλησιέστερος κόμβος του $i \in V\text{-}U$. Με την χρήση του πίνακα A οι διαδικασίες της κατασκευής του νέου πίνακα A και της

επιλογής της ακμής με το ελάχιστο κόστος από το A έχουν πολυπλοκότητα $O(|V|)$. Όταν μια ακμή δεν υπάρχει στο γράφημα, τότε θέτουμε ως βάρος της ένα μεγάλο αριθμό L(=maxint). Ο αλγόριθμος του Prim στην περίπτωση αυτή γίνεται:

```
ΕΔΕΚ:= [];     U = [1];
A[i]:= [j,w}: κόστος (i,j) ≤ κόστος (i,k), k ∈ U]
for i:= 1  to  |V|-1 do
begin
  1: Βρες το στοιχείο A[y] = (x,w) με το
     μικρότερο βάρος στο A;
     εισάγαγε ((y,x), ΕΔΕΚ);
     θέσε A[y]:= (x,L) {ώστε να μην ξαναχρησιμοποιήσουμε το y};

  2: Επαναδιοργάνωσε τον πίνακα A μετά την αποχώρηση του y
     από το V-U και την εισαγωγή του στο U;
     έλεγξε τον πίνακα για να διαπιστώσεις αν η πρόσθεση του  y
     στο U δημιούργησε μια ακμή με μικρότερο βάρος σε κάποιο
     κόμβο  του V-U;
     for i:= 1 to n do {A[j] ≡ (k,w')}
     if (βάρος (y,j) < w') and (w' < L) then
     A[j]:= (y, βάρος (y,j))
end.
```

Τα βήματα 1 και 2 απαιτούν χρόνο $O(|V|)$. Συνεπώς ο αλγόριθμος απαιτεί χρόνο $O|V|(|V|-1) = O(|V|^2)$. Στο σχήμα 7.5 φαίνεται πως αλλάζει ο πίνακας A για το γράφημα του σχήματος 7.1.

	1	2	3	4	5	ΕΔΕΚ
A:	(1,L)	(1,2) (1,L)	(1,12) (2,10) (2,L)	(1,6) (2,5) (2,L)	(1,5) (4,3) (4,L)	(1,2) (2,4) (5,4) (3,2)

Σχήμα 7.5

7.4 Βραχύτερο μονοπάτι (Shortest path)

Στην παράγραφο αυτή θ' ασχοληθούμε με το πρόβλημα της εύρεσης των μονοπατιών μεταξύ των κόμβων ενός διευθυνόμενου γραφήματος ή δικτύου. Οι λύσεις που θα περιγράψουμε ισχύουν και για μη διευθυνόμενα γραφήματα. Τα προβλήματα εύρεσης μονοπατιών

σ'ένα δίκτυο περιλαμβάνουν τα εξής:

Π1. Εύρεση του μονοπατιού από έναν κόμβο πηγής (source) s, σ'έναν κόμβο στόχο ή κόμβο προορισμού (target, ή terminal node), t.

Π2. Εύρεση του πλησιέστερου μονοπατιού από τον s στον t.

Π3. Εύρεση του μονοπατιού με το ελάχιστο βάρος από τον κόμβο s στον κόμβο t.

Π4. Εύρεση, σ'ένα γράφημα, όλων των μονοπατιών με ελάχιστο βάρος (shortest weighed paths), από τον κόμβο s σ'όλους τους κόμβους του γραφήματος.

Στα επόμενα θα ασχοληθούμε μόνο με το Π4. Τα υπόλοιπα προβλήματα μπορούν να θεωρηθούν ως ειδικές περιπτώσεις του Π4 και αφήνονται ως ασκήσεις για τον αναγνώστη.

7.4.1 Αλγόριθμος του Dijkstra

Ο αλγόριθμος του Dijkstra προσφέρει μια λύση του προβλήματος Π4. Η μέθοδος βασίζεται στη διαίρεση του συνόλου των κόμβων V(G) ενός γραφήματος G σε δύο σύνολα: Ενα **μόνιμο (permanent)** σύνολο P, που περιέχει τους κόμβους του γραφήματος με σταθερό μήκος, του πλησιέστερου μονοπατιού με βάρος (shortest weighted path length). Το δεύτερο σύνολο, F, περιέχει όλους τους **συνοριακούς (frontier)** κόμβους, δηλαδή περιέχει τους κόμβους εκείνους που δεν ανήκουν στο σταθερό σύνολο, αλλά είναι γειτονικοί των κόμβων του συνόλου αυτού. Με T_x θα συμβολίζουμε το μήκος του πλησιέστερου μονοπατιού με βάρος για έναν κόμβο x από τον κόμβο s.

Αρχικά το σταθερό σύνολο περιέχει μόνο τον κόμβο s, καθόσον το μικρότερο μήκος του μονοπατιού του είναι γνωστό $T_s = 0$.

Αν ένας κόμβος $y \in F$ είναι γειτονικός με τους κόμβους x_1, x_2,... του P, τότε προσωρινά θέτουμε:

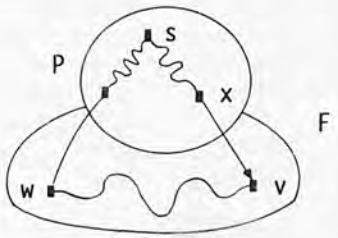
$$T_y = \min \{T_{x1} + \text{βάρος}(x_1, y), T_{x2} + \text{βάρος}(x_2, y), \ldots\}$$

Το T_y αναφέρεται ως το **προσωρινό μήκος μονοπατιού με βάρος (temporary weighted path length)**. Πριν περιγράψουμε τον αλγόριθμο του Dijkstra θα αποδείξουμε το παρακάτω θεώρημα στο οποίο βασίζεται ο αλγόριθμος.

Θεώρημα:
Αν ένας κόμβος $w \in F$ έχει το μικρότερο προσωρινό μήκος μονοπατιού με βάρος (shortest temporary weighted path length) T_w, μεταξύ όλων των κόμβων του F, τότε το T_w μπορεί να γίνει μόνιμο.

Απόδειξη:
Η απόδειξη γίνεται με εις άτοπο απαγωγή. Εστω ότι το T_w δεν

μπορεί να γίνει το μόνιμο μικρότερο μονοπάτι με βάρος για τον κόμβο w. Τότε πρέπει να υπάρχει ένα άλλο μονοπάτι με μικρότερο βάρος από τον s στον w , όπως φαίνεται στο σχήμα 7.6.



Σχήμα 7.6

Εστω ότι το νέο αυτό μονοπάτι είναι το:

$$s \longrightarrow x \longrightarrow v \longrightarrow \ldots \longrightarrow w$$

με:

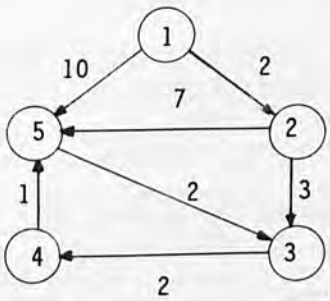
$$T_y + \text{βάρος\_μονοπατιού } (v, w) < T_w.$$

Η ανισότητα αυτή όμως δεν είναι δυνατόν να ισχύει, καθόσον ισχύει ότι:

$$\text{βάρος\_μονοπατιού } (v, w) \geq 0$$

Συνεπώς το T_w είναι το μόνιμο πλησιέστερο μονοπάτι με βάρος για το κόμβο w. Η εφαρμογή του θεωρήματος αυτού στον αλγόριθμο του Dijkstra σημαίνει ότι ο κόμβος w θα πρέπει να μεταφερθεί στο σύνολο P και να κατασκευσθεί ένα νέο σύνολο F. Ο αλγόριθμος τελειώνει όταν το F = ø.

Παράδειγμα. Για το γράφημα του σχήματος 7.7



Σχήμα 7.7

τα βήματα του αλγόριθμου του Dijkstra δίνονται στον παρακάτω πίνακα:

P	F	Προσωρινό μήκος μονοπατιού	Σταθερό μήκος μονοπατιού
{ }	{1}	T_1=0	T_1=0
{1}	{2,5}	T_2=2, T_5=10	T_2=2
{1,2}	{5,3}	T_5=9, T_3=5	T_3=5
{1,2,3}	{5,4}	T_4=7	T_4=7
{1,2,3,4}	{5}	T_5=8	T_5=8
{1,2,3,4,5}	{ }		

7.4.2 Περιγραφή του αλγορίθμου του Dijkstra

```
for κάθε v ∈ V do
T_v = maxint; T_s = 0;  P = ø;  F = {s}
1: while F <> ø do
    begin
2:    εντόπισε τον κόμβο w ∈ F με το μικρότερο T_w και διάγραψέ
      τον;
      εισάγαγε το w στο σύνολο P;
3:    for κάθε (w, x) ∈ E(G) do
      begin
4:      T_x:= min (T_x, T_w + βάρος (w,x));
5:      if x ∉ F then
        εισάγαγε το x στο F
      end
    end
```

Για την υλοποίηση του συνόλου F θα πρέπει να επιλέξουμε μια δομή που να υποστηρίζει αποτελεσματικά τις διαδικασίες:

i) διαγραφής του κόμβου w με min T_w (βήμα 2)
ii) αλλαγής του βάρους T_x (βήμα 4)
iii) ελέγχου του αν w ∈ F και
iv) εισαγωγής του x στο F (βήμα 5)

Το F είναι ένα σύνολο από κόμβους μαζί με τα αντίστοιχα βάρη τους T_x. Μια δομή για την υλοποίηση του F, που να υποστηρίζει τα βήματα 2 και 4, είναι ο σωρός. Ωστόσο όμως οι διαδικασίες του βήματος 5 απαιτούν πολυπλοκότητα χρόνου $O(|V|)$. Για το λόγο αυτό

χρησιμοποιούμε, για την υλοποίηση του F, ένα σωρό, μαζί μ'ένα σύνολο δεικτών προς τον σωρό αυτό, που βελτιώνει τις διαδικασίες αλλαγής του βάρους και του ελέγχου αν w ∈ F του βήματος 5. Έτσι ο αλγόριθμος γίνεται:

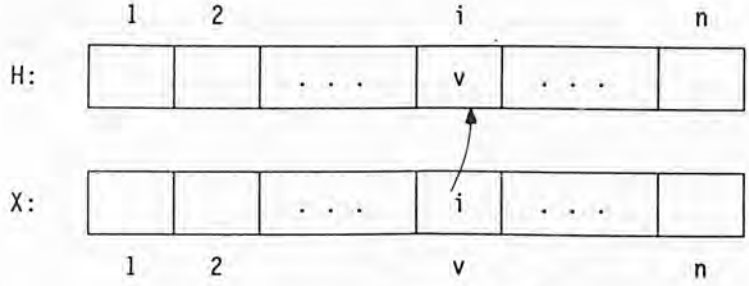
```
procedure Dijkstra;
const n =... ; {κάποια ακέραιη θετική σταθερά}
type κόμβος = 1..n;
var F: συνοριακό_σύνολο;
    T: array [κόμβος] of integer;
    x, w: κόμβος;
begin
  for  κάθε v ∈ V do T[v]:= maxint;
  T[s]:= 0;  F:= [s];
  while  F <> ø do
   begin
     for κάθε (w,x) ∈ E(G) do
     if T[x] > T[w] + βάρος (w,x) then
      begin
        T[x]:= T[w] + βάρος (w,x);
        if x ∉ F then
        εισάγαγε (x,F)
        else άλλαξε_βάρη (x,F)
      end
   end
end
```

Το σύνολο F μπορεί να υλοποιηθεί μ'ένα σωρό H κι έναν πίνακα X από δείκτες προς τον σωρό αυτό.

```
type συνοριακό_σύνολο = record
                          H,X: array [1..n] of κόμβος;
                          μέγεθος: 0..n
                        end;
var F: συνοριακό_σύνολο
```

Ο πίνακας X χρησιμοποιείται για την προσπέλαση των στοιχείων του σωρού. Όταν δίνεται ένας κόμβος v, τότε το X[v] = i μας δίνει τη θέση του κόμβου v στον σωρό H[i] όπως φαίνεται στο σχήμα 7.8. Αν X[v] = 0, τότε ο κόμβος v δεν υπάρχει στον σωρό. Έτσι η θέση ενός κόμβου στο σωρό καθορίζεται σε χρόνο O(1).

Για τον υπολογισμό της πολυπλοκότητας του αλγορίθμου του Dijkstra θα υπολογίσουμε ξεχωριστά την πολυπλοκότητα κάθε μιας των διαδικασιών που χρησιμοποιεί ο αλγόριθμος.



Σχήμα 7.8

εισάγαγε (x,F): Το βάρος T[x] χρησιμοποιείται ως η προτεραιότητα του κόμβου x. Η εισαγωγή γίνεται στον πίνακα H. Κάθε φορά που αλλάζει ένα στοιχείο του H (με την εντολή H[i]:= j), ενημερώνουμε τον πίνακα X των δεικτών ανάλογα (με την εντολή X[j]:= i). Ο χρόνος που απαιτείται γι'αυτό είναι $O(\log_2 n)$.

διάγραψε τον κόμβο x με ελάχιστο T_x: Επειδή το x διαγράφεται από το F, έχουμε X[x]=0 και κάθε άλλη αλλαγή στα υπόλοιπα στοιχεία του H πρέπει να συνοδεύεται με ενημέρωση του πίνακα των δεικτών X. Η πολυπλοκότητα του χρόνου που απαιτείται γι'αυτό είναι $O(\log_2 n)$.

x ανήκει στο F: (ανήκει(x,F)) Αν X[x] = 0, τότε το x δεν ανήκει το F, διαφορετικά το x ανήκει στο F. Ο χρόνος για τον έλεγχο αυτό είναι O(1).

άλλαξε_βάρη (x,F): Το νέο βάρος του x δίνεται από το T[x] στο βήμα 4. Η πολυπλοκότητα χρόνου γι'αυτό είναι $O(\log_2 n)$.

Από τα παραπάνω έπεται ότι η συνολική πολυπλοκότητα του αλγόριθμου του Dijkstra είναι:

$$O\ (n+n\log_2 n+e+\max(n\log_2 n,\ e\log_2 n)) \approx O(\max(n,e)\log_2 n)$$

όπου $e = |E|$ και $n=|V|$. Επειδή συνήθως $e > n$ προκύπτει τελικά ότι η πολυπλοκότητα είναι $O(e\log_2 n)$.

Ο αλγόριθμός που περιγράψαμε μέχρι εδώ υπολογίζει το μήκος των μικροτέρων μονοπατιών για τον κάθε κόμβο. Για να έχουμε και

τα αντίστοιχα μονοπάτια, θα πρέπει να κρατάμε πάντα τον προηγούμενο κόμβο. Αυτό μπορεί να γίνει με τη χρήση ενός πίνακα "προηγούμενος_κόμβος" και την προσθήκη στον αλγόριθμο της εντολής:

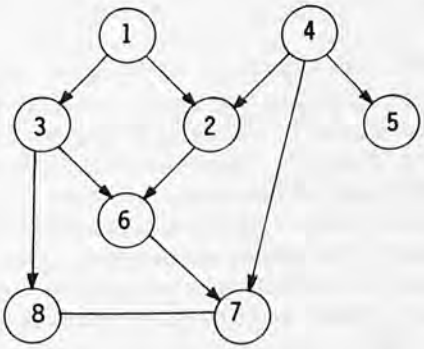
προηγούμενος_κόμβος [x]:= w

μετά τον υπολογισμό του νέου κόμβου x. Έτσι το μονοπάτι για έναν κόμβο x υπολογίζεται ως εξής:

```
procedure βρές_μονοπάτι (x: κόμβος);
begin
  if x <> s then
   begin
     μονοπάτι (προηγούμενος_κόμβος [x];
     write (προηγούμενος_κόμβος [x])
   end
end {βρές_μονοπάτι}
```

7.5 Τοπολογική Ταξινόμηση (Topological sorting)

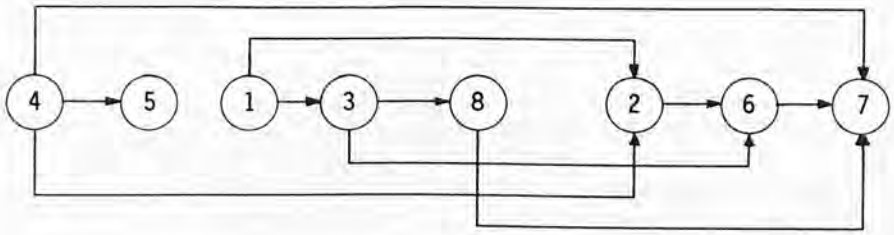
Το πρόβλημα της τοπολογικής ταξινόμησης σημαίνει τη μερική διάταξη των κόμβων ενός **διευθυνόμενου ακυκλικού γραφήματος, (Directed Acyclic Graph) ΔΑΓ** σε μια ακολουθία, έτσι ώστε, αν (v, w) ∈ E(G), τότε ο κόμβος v εμφανίζεται πριν από τον κόμβο w στην ακολουθία. Ενα χαρακτηριστικό παράδειγμα τοπολογικής ταξινόμησης είναι το πρόβλημα των προαπαιτουμένων μαθημάτων ενός προγράμματος σπουδών. Η τοπολογική ταξινόμηση του γραφήματος των μαθημάτων μας δίνει τη σειρά με την οποία θα πρέπει να επιλέξει ένας φοιτητής τα μαθήματα στο πρόγραμμα των σπουδών του, έτσι



Σχήμα 7.9

ώστε να ικανοποιούνται οι σχέσεις μεταξύ προαπαιτούμενων και υποχρεωτικών μαθημάτων. Για παράδειγμα στο σχήμα 7.9 η σχέση του προαπαιτούμενου μεταξύ δύο μαθημάτων v και w καθορίζεται από την φορά της ακμής (v, w).

Στο σχήμα αυτό το μάθημα 7 έχει προαπαιτούμενα τα 4, 6 και 8. Το μάθημα 4 δεν έχει προαπαιτούμενα. Μια δυνατή λύση του προβλήματος δίνεται στο σχήμα 7.10.



Σχήμα 7.10

Η βασική ιδέα του αλγόριθμου της τοπολογικής ταξινόμησης είναι η εξής. Εντοπίζουμε έναν κόμβο χωρίς απογόνους, τον απομακρύνουμε από το γράφημα και τον προσθέτουμε σε μια στοίβα. Η διαδικασία συνεχίζεται μέχρις ότου όλοι οι κόμβοι του γραφήματος εισαχθούν στη στοίβα. Η εξαγωγή από τη στοίβα όλων των στοιχείων της μας δίνει τελικά μια τοπολογική τους ταξινόμηση.

Ο αλγόριθμος που περιγράψαμε υλοποιείται με διάσχιση του γραφήματος κατά βάθος πρώτα. Εστω ότι για έναν κόμβο v του γραφήματος υπάρχει ένα μονοπάτι:

$$v \longrightarrow v_1 \longrightarrow v_2 \longrightarrow \ldots v_k \longrightarrow w, \quad \text{όπου } k \geq 0$$

Τότε, σύμφωνα με τον ορισμό, ο κόμβος v_k θα πρέπει να εμφανίζεται πριν από τον κόμβο w, ο v_{k-1} πριν από τον v_k κ.ο.κ. Σύμφωμα με τον αλγόριθμο dfs, όταν η διάσχιση έχει φτάσει στη επίσκεψη ενός κόμβου v, τότε θα πρέπει όλοι οι επόμενοί του κόμβοι να έχουν ήδη τύχει επισκέψεως. Μ'άλλα λόγια η ιδιότητα "βάθος πρώτα" του αλγόριθμου dfs, μας εξασφαλίζει ότι όλοι οι προηγούμενοι κόμβοι έχουν ήδη τύχει επισκέψεως, οπότε εισαγάγουμε τον κόμβο σε μια στοίβα. Στο τέλος μια ακολουθία από εξαγωγές από την στοίβα (pop) θα μας δώσει τη ζητούμενη τοπολογική ταξινόμηση.

Αλγόριθμος Τοπολογικής Ταξινόμησης

```
δημιούργησε (στοίβα);
δώσε_αρχικές_τιμές; {όλοι οι κόμβοι του γραφήματος
                     χαρακτηρίζονται πως "δεν έχουν τύχει
                     επισκέψεως" unvisited (not visited)}
for κάθε v ∈ V do
if not visited [v] then dfs (v);
{άδειασε τη στοίβα}
while not κενή (στοίβα) do
   begin
     εξάγαγε (στοιχείο, στοίβα);
     write (στοιχείο)
   end
```

Στην περίπτωση αυτή ο αλγόριθμος dfs γράφεται εξής:

```
      visited (v):= true;
      for (v,w) ∈ E(G) do
      if not visited(w) then dfs(w);
      {στο σημείο αυτό όλοι οι προγόνοι του v έχουν εισαχθεί στη
      στοίβα.  Έτσι εισάγουμε και τον κόμβο v στη στοίβα}
      εισάγαγε (v, στοίβα)
    end
```

Ο αλγόριθμος της τοπολογικής ταξινόμησης έχει πολυπλοκότητα χρόνου $O(|E|)$.

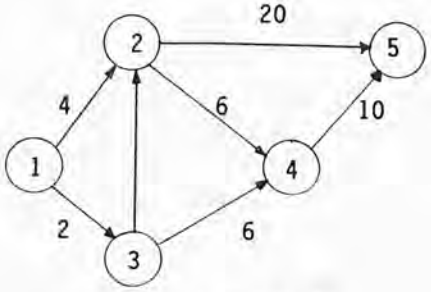
7.6 Εφαρμογές γραφημάτων, Δίκτυα PERT - Κρίσιμο Μονοπάτι

Τα διευθυνόμενα γραφήματα χρησιμοποιούνται συχνά για την παράσταση και την ανάλυση συνθέτων **έργων (projects)** τα οποία αποτελούνται από πολλές αλληλοσχετιζόμενες **δραστηριότητες (activities).** Υπάρχουν διάφορες τεχνικές για την ανάλυση έργων με γραφήματα. Από αυτές οι πιο γνωστές είναι οι **PERT (Project Evaluation and Review Technique) και CPM (Critical Path Method, Μέθοδος Κρισίμου Μονοπατιού).**

Ενα γράφημα PERT είναι ένα πεπερασμένο γράφημα με βάρη, χωρίς κύκλους, στο οποίο υπάρχει ένας **κόμβος-πηγή (source)** με μέσα-βαθμό μηδέν (indegree=0) και ένας **κόμβος-αποδέκτης (sink)** με έξω-βαθμό (outdegree = 0) ίσο με μηδέν. Οι διευθυνόμενες ακμές σ'ένα τέτοιο γράφημα παριστάνουν δραστηριότητες και οι αντίστοιχοι κόμβοι, που συνδέει μια ακμή, παριστάνουν τον αρχικό και τον τελικό χρόνο της δραστηριότητας αυτής. Το βάρος της ακμής παριστάνει το χρόνο που απαιτείται για να ολοκληρωθεί η

παριστάνει το χρόνο που απαιτείται για να ολοκληρωθεί η δραστηριότητα.

Σ'ένα γράφημα PERT υπάρχουν συνήθως αλληλοεξαρτήσεις μεταξύ των δραστηριοτήτων. Για παράδειγμα μια δραστηριότητα μπορεί να είναι αδύνατο να ξεκινήσει, αν δεν έχει τελειώσει κάποια άλλη δραστηριότητα. Οταν γνωρίζουμε αυτές τις πληροφορίες για ένα έργο, τότε μπορούμε να το παραστήσουμε μ'ένα γράφημα (χρονογράφημα). Ας θεωρήσουμε για παράδειγμα το γράφημα του σχήματος 7.11.



Σχήμα 7.11

Το γράφημα αυτό έχει επτά δραστηριότητες που έχουν μια σχέση διάταξης με την έννοια ότι ορισμένες δεν μπορούν να ξεκινήσουν, αν δεν έχουν τελειώσει άλλες. Για παράδειγμα πριν ξεκινήσει η δραστηριότητα (2,5), θα πρέπει να έχουν τελειώσει οι (1,2) και (3,2). Ο κόμβος 1 είναι ο κόμβος-πηγή και ο 5 είναι ο κόμβος-αποδέκτης ή τελικός κόμβος του έργου. Οι αριθμοί σε κάθε ακμή παριστάνουν το χρόνο που απαιτεί η κάθε δραστηριότητα. Κάθε κόμβος του γραφήματος ονομάζεται **γεγονός (event)** και παριστάνει μια χρονική στιγμή. Ειδικά ο κόμβος-πηγή παριστάνει την αρχή του έργου και ο κόμβος-αποδέκτης το τέλος του. Στα επόμενα θα ασχοληθούμε με τον υπολογισμό του κρίσιμου μονοπατιού για ένα διευθυνόμενο γράφημα.

Κρίσιμο μονοπάτι μεταξύ δύο κόμβων ενός διευθυνόμενου γραφήματος με βάρη είναι το μονοπάτι με το μεγαλύτερο μήκος. Το κρίσιμο μονοπάτι είναι ένα πολύ χρήσιμο εργαλείο για την εκτίμηση έργων με πολλές εφαρμογές, όπως είναι ο χρονικός προγραμματισμός ενός έργου κ.α. Ο καθορισμός του κρίσιμου μονοπατιού με την τεχνική PERT γίνεται, σε δύο βήματα, ως εξής:

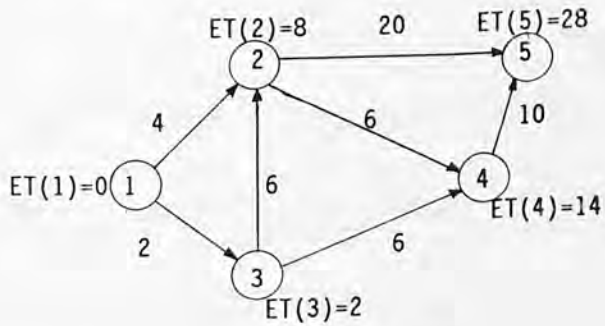
I. Για κάθε δραστηριότητα υπολογίζουμε τον **ενωρίτερο χρόνο ολοκλήρωσής της (earliest completion time)** με τον περιορισμό ότι, πριν ξεκινήσει μια δραστηριότητα, θα πρέπει να έχουν τελειώσει όλες οι άλλες από τις οποίες εξαρτάται. Σε κάθε

κόμβο i εκχωρούμε μια τιμή ET(i), που αντιστοιχεί στη χρονική διάρκεια που απαιτείται για να τελειώσουν όλες οι δραστηριότητες κατά μήκος του μεγαλύτερου μονοπατιού, από την αρχή του γραφήματος μέχρι τον κόμβο i. Μ'άλλα λόγια, σε κάθε κόμβο εκχωρούμε τη μεγίστη τιμή μεταξύ όλων των ακμών που φτάνουν στον κόμβο αυτόν συν το βάρος του αρχικού κόμβου για την κάθε ακμή. Αν 1,2,...,k είναι οι ακμές που φτάνουν στον κόμβο i, τότε:

ET(1) = 0
ET(i) = max {(ET(1) + βάρος (1,i)),... , (ET(k) + βάρος (k,i))

Η τιμή που εκχωρείται στον τελικό κόμβο αντιστοιχεί στον ενωρίτερο χρόνο πραγματοποίησης όλου του έργου. Οι τιμές αυτές για το γράφημα του σχήματος 7.11 δίνονται στο σχήμα 7.12

II. Για κάθε κόμβο υπολογίζουμε το **βραδύτερο χρόνο ολοκλήρωσής του (latest completion time)**. Ο χρόνος αυτός είναι ο μέγιστος χρόνος στον οποίο μπορεί να εκτελεστεί μια δραστηριότητα, χωρίς να δημιουργήσει καθυστέρηση στον ενωρίτερο χρόνο πραγματοποίησης ολόκληρου του έργου. Είναι δηλαδή οι βραδύτεροι χρόνοι ολοκλήρωσης για κάθε δραστηριότητα οι χρόνοι οι οποίοι δεν αυξάνουν τον χρόνο ET(5) του τελικού κόμβου. Ο βραδύτερος χρόνος ολοκλήρωσης που εκχωρείται σ'έναν κόμβο είναι ο μεγαλύτερος χρόνος ο οποίος επιτρέπει στην κάθε δραστηριότητα που ξεκινάει από τον κόμβο αυτόν να τελειώνει,

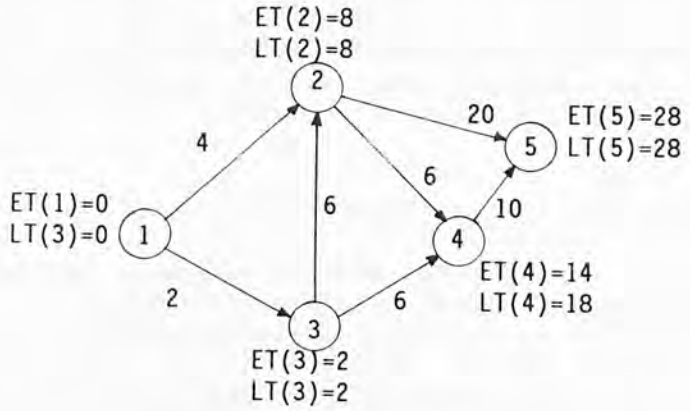


Σχήμα 7.12

χωρίς ν'αυξηθεί ο συνολικός χρόνος πραγματοποίησης του έργου. Έτσι στον κάθε κόμβο i του γραφήματος δίνουμε τιμές, LT(i), που υπολογίζονται ως εξής:

LT(n) = ET(n)
LT(j) = min{LT(1) - e_{j1}),... , (LT(k) - e_{jk})}

όπου $e_{j1} \ldots e_{jk}$ είναι όλες οι δραστηριότητες με αρχή τον κόμβο j (βλέπε για παράδειγμα το σχήμα 7.13)



Σχήμα 7.13

Το κρίσιμο μονοπάτι είναι εκείνο το μονοπάτι, από τον κόμβο-πηγή στον τελικό κόμβο, που αν μια δραστηριότητα καθυστερήσει κατά χρόνο t, ολόκληρο το έργο θα καθυστερήσει κατά χρόνο t. Κάθε κόμβος (i) του κρισίμου μονοπατιού έχει ET(i)=LT(i). Για το παράδειγμά μας, το κρίσιμο μονοπάτι είναι το (1,3,2,5). Για να επιταχύνουμε την εκτέλεση όλου του έργου αρκεί να επιταχύνουμε μόνο εκείνες τις δραστηριότητες που βρίσκονται πάνω στο κρίσιμο μονοπάτι.

Υλοποίηση

Κάθε κόμβος του γραφήματος είναι μια εγγραφή που έχει τη μορφή:

περιεχόμενο	ET	LT	επόμενος_κόμβος	λίστα1	λίστα2

Το πεδίο "περιεχόμενο" περιέχει την ετικέτα του κόμβου και τυχόν άλλες πληροφορίες. Το πεδίο "λίστα1" περιέχει όλες τις ακμές που ξεκινάνε από τον κόμβο, ενώ το πεδίο "λίστα2" όλες τις ακμές που φτάνουν στον κόμβο. Κάθε κόμβος ακμής περιέχει τα παρακάτω πεδία:

κλειδί1	κλειδί2	βάρος	σύνδεσμος1	σύνδεσμος2

Για την κατασκευή του γραφήματος PERT του σχήματος 7.13 θα υποθέσουμε ότι οι κόμβοι του γραφήματος είναι τοπολογικά ταξινομημένοι σε αύξουσα σειρά της ετικέτας με κόμβο-πηγή τον κόμβο 1 και κόμβο-αποδέκτη τον κόμβο 5 που αντιστοιχεί στο τέλος του έργου. Για να καθορίσουμε το κρίσιμο μονοπάτι θα πρέπει να υπολογίσουμε για τον κάθε κόμβο τις τιμές ET και LT. Ενας αλγόριθμος που επιτυγχάνει αυτό είναι ο εξής:

```
ET[1]:= 0
for τρέχων_κόμβος:= 1 to αριθμός_κόμβων do

for κάθε ακμή e[i, τρέχων_κόμβος] του τρέχοντος κόμβου do
{υπολόγισε την τιμή ET}
ET[τρέχοντος_κόμβου]:= max {ET[i] + weight (e[i,τρέχων_κόμβος])};
                          i
LT[αριθμός_κόμβων]:= ET[αριθμός_κόμβων];
 for τρέχων_κόμβος:= πλήθος_κόμβων down to 1 do
   for κάθε ακμή e[τρέχων_κόμβος, j]  do
   {υπολόγισε την τιμή LT}
 LT[τρέχοντος_κόμβου]:= min{LT[j] - weight(e[τρέχων_κόμβος, j]);
                            j
 τύπωσε τους κόμβους που βρίσκονται στο κρίσιμο μονοπάτι
```

Αντί ανακεφαλαίωσης

Στο κεφάλαιο αυτό εξετάσαμε μερικούς από τους βασικότερους αλγόριθμους επί των γραφημάτων.Οπως είδαμε όλοι οι αλγόριθμοι βασίζονται στους αλγόριθμους διάσχισης ενός γραφήματος.
Αν και οι αλγόριθμοι αυτοί έχουν τεράστιες εφαρμογές σε πολλές επιστήμες και ιδιαίτερα στην επιστήμη των υπολογιστών, με την καθιέρωση των δικτύων υπολογιστών η εκτενέστερη μελέτη τους θα ξέφευγε από τους σκοπούς για τους οποίους γράφτηκε αυτό το βιβλίο. Για περισσότερη μελέτη ο ενδιαφερόμενος αναγνώστης παραπέμπεται στις αναφορές 7 και 18 στο τέλος του βιβλίου.

Ασκήσεις

1. Προσαρμόστε τον αλγόριθμο του Dijkstra για τον υπολογισμό: (i) του πλησιέστερου μονοπατιού από έναν κόμβο s σ'έναν κόμβο t και (ii) του μικρότερου μονοπατιού με βάρος από τον κόμβο s στον κόμβο t.

2. Ως ακτίνα ενός δένδρου ορίζουμε την μεγίστη απόσταση από τη ρίζα προς τα φύλλα του. Δοθέντος ενός συνδεδεμένου μη διευθυνόμενου γραφήματος γράψτε έναν αλγόριθμο που να υπολογίζει το επικαλυπτικό δένδρο με την ελάχιστη ακτίνα.

3. Γράψτε ένα πρόγραμμα σε Pascal που να υλοποιεί τον αλγόριθμο του Prim. Ελέγξτε το πρόγραμμα σας χρησιμοποιώντας το γράφημα του σχήματος 7.1

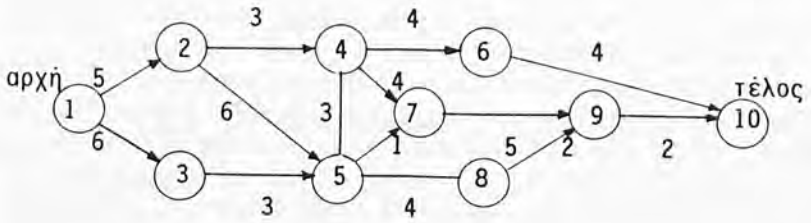
4. Γράψτε έναν αλγόριθμο που να καθορίζει αν δύο κόμβοι με κλειδιά k1 και k2, αντίστοιχα, είναι συνδεδεμένοι.

5. Γράψτε ένα πρόγραμμα σε Pascal που να υλοποιεί τον αλγόριθμο της τοπολογικής ταξινόμησης.

6. Γράψτε έναν αλγόριθμο και το αντίστοιχο πρόγραμμα σε Pascal για τον υπολογισμό του πλησιέστερου μονοπατιού μεταξύ δύο κόμβων, όταν η υλοποίηση του γραφήματος γίνεται με λίστες.

7. Κατασκευάστε το διάγραμμα δραστηριοτήτων για το πρόβλημα των προαπαιτουμένων μαθημάτων του προγράμματος σπουδών σας, που να περιέχει μόνο τα μαθήματα της ειδίκευσης της πληροφορικής. Γράψτε την τοπολογικά ταξινομημένη ακολουθία των μαθημάτων αυτών.

8. Ενα έργο παριστάνεται με το παρακάτω δίκτυο δραστηριοτήτων:



Υπολογίστε τον ενωρίτερο χρόνο για να τελειώσει το έργο.

9. Γράψτε το πρόγραμμα για τον αλγόριθμο του Dijkstra, όταν το γράφημα ορίζεται με πίνακα γειτνίασης.

ΚΕΦΑΛΑΙΟ 8

ΑΦΗΡΗΜΕΝΟΙ ΤΥΠΟΙ ΔΕΔΟΜΕΝΩΝ

8.1 Γενικά

Το κόστος της ανάπτυξης (development) και της συντήρησης (maintenance) του σύγχρονου λογισμικού είναι πολύ υψηλό, σε αντίθεση με την ποιότητα των παραγομένων προγραμμάτων που είναι πολύ χαμηλή. Τα προβλήματα αυτά είναι οξύτερα στα μεγάλα και πολύπλοκα προγράμματα που έχουν μεγάλη διάρκεια ζωής (lifetime) και χρήση (use). Σε τέτοια προγράμματα συνήθως εμπλέκονται πολλοί προγραμματιστές, όχι μόνο κατά τη διάρκεια της ανάπτυξής τους, αλλά και για τη συντήρηση που αυτά χρειάζονται μετά την αρχική τους έκδοση (initial release). Έτσι το κόστος και η ποιότητα του λογισμικού επηρεάζονται όχι μόνο από την διαχείριση (management) της ομάδας παραγωγής του, αλλά και από τις διάφορες μεθόδους της τεχνολογίας λογισμικού (software engineering). Ο έλεγχος της ανάπτυξης και της συντήρησης του λογισμικού συνεπάγεται επίσης τη διαχείριση της διανοητικής πολυπλοκότητας (intellectual complexity), των προγραμμάτων. Αυτό σημαίνει ότι πολλοί άνθρωποι θα πρέπει να καταλάβουν ένα πρόγραμμα για να μπορέσουν με οποιοδήποτε τρόπο να επέμβουν σ'αυτό, είτε για να το δοκιμάσουν, είτε για να το συντηρήσουν ή για να το επεκτείνουν.

Μια από τις σημαντικότερες εξελίξεις των μεθοδολογιών και των γλωσσών προγραμματισμού είναι η ανάπτυξη εργαλείων, που έχουν να κάνουν με **λογική αφαίρεση (abstraction)**. Λογική αφαίρεση είναι η απλοποιημένη **περιγραφή ή προδιαγραφή (specification)** ενός συστήματος, που δίνει έμφαση σε ορισμένες μόνο λεπτομέρειες ή ιδιότητες του συστήματος αυτού, ενώ αποκρύπτει άλλες. "Καλή" αφαίρεση είναι εκείνη η οποία τονίζει ιδιαίτερα εκείνες τις ιδιότητες του συστήματος, που είναι σημαντικές για τον χρήστη, ενώ αποκρύπτει άλλες ιδιότητες, που δεν είναι αναγκαίο να τις γνωρίζει ο χρήστης. Ενα απλό παράδειγμα αφαίρεσης είναι οι ακέραιοι και οι πραγματικοί αριθμοί. Ολοι μας χρησιμοποιούμε με άνεση ακεραίους ή πραγματικούς αριθμούς στα προγράμματά μας,

αλλά όπως γνωρίζουμε εσωτερικά στον υπολογιστή αυτοί παριστάνονται από δυαδικά ψηφία (bits).

Η ιδέα της λογικής αφαίρεσης πρωτοεμφανίστηκε στη δεκαετία του 70. Κατά την περίοδο αυτήν η έρευνα πάνω σε αφαιρετικές τεχνικές επικεντρώθηκε στις γλώσσες προγραμματισμού και σε θέματα των προδιαγραφών των τύπων δεδομένων. Όπως στο δομημένο προγραμματισμό, η μεθοδολογία των αφηρημένων τύπων δεδομένων (abstract data types) δίνει έμφαση στη τοπικότητα (locality) ενοτήτων, που περιέχουν συσχετιζόμενες πληροφορίες. Έτσι δόθηκε έμφαση στην οργάνωση των προγραμμάτων κατά **ενότητες** (modules) **με** τέτοιο τρόπο ώστε οι λεπτομέρειες της υλοποίησής τους να κρατούνται όσο το δυνατόν **τοπικά (locally)** σε κάθε ενότητα. Η προσοχή όμως τώρα συγκεντρώθηκε στα δεδομένα μάλλον παρά στον έλεγχο και η στρατηγική ήταν να κατασκευαστούν ενότητες, που να ορίζουν πλήρως μια δομή δεδομένων μαζί με τις πράξεις της. Ο σκοπός ήταν να μπορεί κανείς να χρησιμοποιήσει τις ενότητες αυτές με τον ίδιο τρόπο, όπως τους ακεραίους, τους πραγματικούς αριθμούς κ.λπ. Αυτό σημαίνει ότι η γλώσσα προγραμματισμού θα πρέπει να διαθέτει τα κατάλληλα χαρακτηριστικά για τις δηλώσεις, τους ορισμούς των ενδοθεματικών τελεστών (infix operators) κ.λπ.

Το αποτέλεσμα των αφηρημένων τύπων δεδομένων (ΑΤΔ) είναι η επέκταση του συνόλου των τύπων, που διατίθενται από τη γλώσσα προγραμματισμού, καθώς και η επεξήγηση του νέου συνόλου των τιμών, που μπορούν να πάρουν οι μεταβλητές ενός νέου τύπου. Στην αφαίρεση ενός τύπου δεδομένων θα πρέπει, λοιπόν, αφού καθορίσουμε τις συναρτησιακές ιδιότητες της δομής δεδομένων και τις πράξεις επί των στοιχείων της, να υλοποιήσουμε αυτές σε μία γλώσσα προγραμματισμού. Όταν στη συνέχεια χρησιμοποιούμε την αφαίρεση, έχουμε να κάνουμε μόνο με το νέο τύπο, ανάλογα με την προδιαγραφή του.

Αν και οι ιδέες πίσω από μοντέρνες αφαιρετικές τεχνικές μπορούν να εξεταστούν ανεξάρτητα από τη γλώσσα προγραμματισμού, η μελέτη τους σε σχέση με μια πραγματική γλώσσα είναι πολύ σημαντική. Γιά το λόγο αυτό, οι γλώσσες προγραμματισμού είναι το κύριο εργαλείο για να εκφράσουμε τις ιδέες αυτές. Η ανάπτυξη γλωσσών έχει επηρεάσει τον τρόπο με τον οποίο σκεπτόμαστε τους αλγόριθμους, κάνοντας έτσι τις δομές των προγραμμάτων ευκολότερες. Αν και οι περισσότερες γλώσσες έχουν τεχνικά τις ίδιες περίπου δυνατότητες, εν τούτοις, μικρές διαφορές μεταξύ τους μπορούν να επηρεάσουν σημαντικά την πρακτική τους χρήση. Στις περισσότερες γλώσσες, που έχουν τη δυνατότητα ορισμού ενός Αφηρημένου Τύπου Δεδομένων (ΑΤΔ), ο ορισμός ενός τέτοιου τύπου αποτελεί μια ενότητα του προγράμματος, που περιλαμβάνει τα παρακάτω :

1. Το **μέρος** που είναι **ορατό (visible part)** στο χρήστη, για

παράδειγμα τ'όνομα του τύπου και οι επικεφαλίδες των υποπρογραμμάτων, που επιτρέπεται να χρησιμοποιησουν τον τύπο αυτόν και

2. Το μέρος που είναι **κρυφό από το χρήστη (information hiding)**. Το μέρος αυτό περιλαμβάνει την υλοποίηση του νέου τύπου με τη βοήθεια άλλων τύπων δεδομένων, τα σώματα των υποπρογραμμάτων (οι επικεφαλίδες των οποίων υπάρχουν στο ορατό μέρος) και τέλος άλλα "κρυμμένα" υποπρογράμματα, που μπορούν να κληθούν μόνο μέσα από την ενότητα.

8.2 Pascal

Η Pascal είναι μια απλή αξιωματική γλώσσα σχεδιασμένη για να ικανοποιήσει τρεις κυρίως σκοπούς:

1. Την ανάπτυξη μεθοδολογίας σύγχρονου προγραμματισμού,
2. Την διδασκαλία σε φοιτητές, γιατί είναι αρκετά απλή και
3. Τη χρήση της ακόμη και σε προσωπικούς υπολογιστές, γιατί είναι μικρή σε μέγεθος.

Πράγματι η Pascal έχει σήμερα πετύχει και τους τρεις αυτούς σκοπούς και χρησιμοποιείται ευρύτατα. Οπως γνωρίζουμε η γλώσσα περιέχει όλες εκείνες τις δομές ελέγχου που υποστηρίζουν το δομημένο προγραμματισμό. Διαθέτει επίσης εκείνα τα χαρακτηριστικά που χρειάζονται για τον ορισμό αφαιρέσεων. Αυτά περιλαμβάνουν, απαριθμητούς τύπους δεδομένων, εγγραφές, δείκτες, καθώς και τη δυνατότητα να ορίσει κανείς ένα νέο τύπο δεδομένων. Ωστόσο όμως οι πράξεις, που συνοδεύουν μια δομή, δεν αποτελούν μια ενότητα μαζί με τον ορισμό που δίνεται από το χρήστη. Για παράδειγμα, όταν σ'ένα πρόβλημα, που χρησιμοποιεί μια στοίβα με στοιχεία ακεραίους, θέλουμε ν'αλλάξουμε τον τύπο των στοιχείων της στοίβας σε πραγματικούς αριθμούς, τότε θα πρέπει να επέμβουμε στο πρόγραμμα για ν'αλλάξουμε τον τύπο από integer σε real και να κάνουμε εκ' νέου μεταγλώττιση ολοκλήρου του προγράμματος. Αλλα βασικά μειονεκτήματα της Pascal είναι:

- η αδυναμία να υποστηρίζει μεγάλα προγράμματα. Πράγματι, η μόνη δυνατότητα που έχουμε για να γράψουμε ανεξάρτητες ενότητες στην Pascal, είναι με εμφωλιασμένες διαδικασίες και
- η αδυναμία για ανεξάρτητη μεταγλώττιση ενοτήτων του ίδιου προγράμματος.

Παρά τις αδυναμίες της Pascal, στο βιβλίο αυτό προσπαθήσαμε να κρατήσουμε, όσο ήταν δυνατό, τους ορισμούς των νέων δομών σε

Pascal με τέτοιο τρόπο, ώστε να συμφωνούν με τους ΑΤΔ. Για κάθε νέα δομή δίναμε τον ορισμό της στην Pascal και αμέσως μετά όλες τις πράξεις που ορίζονταν πάνω στη δομή αυτή υπό μορφή υποπρογραμμάτων.

8.3 Ada

Η Ada, επηρεασμένη από την επιτυχία της Pascal, περιέχει όχι μόνο πολλά χαρακτηριστικά της γλώσσας αυτής, αλλά και πολλές σημαντικές αλλαγές και επεκτάσεις που την κάνουν να διαφέρει πολύ από την Pascal. Μερικά από τα κυριότερα χαρακτηριστικά της Ada είναι:

- Λογική αφαίρεση
- Αυστηρότητα στους ορισμούς των τύπων δεδομένων (strongly typed). Ο μεταφραστής δεν κάνει καμία αλλαγή προς διευκόλυνση του χρήστη. Έτσι τα περισσότερα λάθη ελέγχονται κατά τη μετάφραση του προγράμματος.
- Ειδικός μηχανισμός **εξαιρέσεων (exceptions)** για τον έλεγχο της κανονικής ροής του προγράμματος, όταν συμβεί κάποιο λάθος.
- Οργάνωση σε ενότητες (modularity). Η γλώσσα διαθέτει τέσσερα είδη μονάδων προγράμματος (program units) ως εξής.

1. υποπρογράμματα (subprograms),
2. πακέτα (packages),
3. διεργασίες (tasks), για παράλληλη επεξεργασία και
4. γενικά πακέτα (generic packages).

Ενας ορισμός τύπου generic επιτρέπει τη δημιουργία πολλών ομοίων αφαιρέσεων.

Στα παρακάτω θα ασχοληθούμε μόνο με τα πακέτα γιατί αυτά χρησιμοποιούνται για ορισμούς ΑΤΔ.

Ενα πακέτο, στην Ada, αποτελείται από δύο μέρη. Τον ορισμό ή **προδιαγραφή (specification)** του, που είναι ορατός από άλλες ενότητες του προγράμματος, και το **σώμα (body)** του,το περιεχόμενο του οποίου δεν είναι ορατό από άλλες μονάδες του προγράμματος. Έτσι οι πληροφορίες ενός πακέτου είναι γνωστές μόνο στο ίδιο το πακέτο και με τον τρόπο αυτό διαφυλλάσεται το απόρρητο του πακέτου. Ο χρήστης μπορεί να χρησιμοποιήσει μόνο τις δηλώσεις που εμφανίζονται στο τμήμα του ορισμού του πακέτου. Η συντακτική δομή ενός πακέτου είναι:

```
package όνομα_πακέτου is
-- προδιαγραφή, ορατό μέρος
δηλώσεις;
επικεφαλίδες υποπρογραμμάτων;
end όνομα_πακέτου;

package body όνομα_πακέτου is
-- σώμα
τοπικές δηλώσεις μεταβλητές και υποπρογράμματα;
begin
  εντολές;
end όνομα_πακέτου;
```

Παράδειγμα

Εστω ότι θέλουμε να ορίσουμε μια δομή που να παριστάνει τα κλάσματα. Για το σκοπό αυτό θα χρησιμοποιήσουμε μια εγγραφή που περιέχει δύο πεδία: τον αριθμητή και τον παρανομαστή του κλάσματος, όπου ο αριθμητής είναι ένας ακέραιος και ο παρανομαστής είναι θετικός ακέραιος. Οι πράξεις που ορίζονται στα κλάσματα είναι οι +, -, *, /, όπως τις μάθαμε στο δημοτικό σχολειο. Επειδή η Ada επιτρέπει επικάλυψη στους ορισμούς των ενδοθεματικών τελεστών, αντί να χρησιμοποιήσουμε μια συνάρτηση, sum, έστω, για το άθροισμα, μπορούμε να χρησιμοποιήσουμε το ίδιο το σύμβολο της πρόσθεσης. Η ιδιότητα αυτή της επικάλυψης των τελεστών ονομάζεται **υπερφόρτωση (overloading)**. Η αναγνώριση του συμβόλου + γίνεται από το μεταγλωττιστή σύμφωνα με τον τύπο των ορισμάτων του. Αλλες πράξεις είναι η δημιουργία ενός κλάσματος, οι συγκρίσεις (=, <, >= κ.λπ.). Ο ορισμός του πακέτου των κλασμάτων στην Ada θα μπορούσε να είναι είναι ως εξής:

```
package κλάσματα is
type κλάσμα is
 record
  αριθμ: integer;
  παρον: positive;
end record;

function δημιουργία_κλάσματος (N, D: integer) return κλάσμα;
function αριθμητής (x: κλάσμα) return integer;
function παρανομαστής (x: κλάσμα) return integer;
-- οι διαδικασίες αυτές επιστρέφουν τον αριθμητή ή
-- τον παρανομαστή ενός κλάσματος
-- υπερφόρτωση αριθμητικών ενδοθεματικών τελεστών
function '+' (x, y: κλάσμα) return κλάσμα;
function '-' (x, y: κλάσμα) return κλάσμα;
```

```
function '*' (x, y: κλάσμα) return κλάσμα;
function '/' (x, y: κλάσμα) return κλάσμα;
-- υπερφόρτωση τελεστών σύγκρισης
function ισα (x, y: κλάσμα) return boolean;
function '<' (x, y: κλάσμα) return boolean;
function '>=' (x, y: κλάσμα) return boolean;
function αναγωγή (χ: κλάσμα) return κλάσμα;
end κλάσματα;

package body κλάσματα is
-- σώμα της συνάρτησης αριθμητής
function αριθμητής (x: κλάσμα) return integer is
begin
  return x.αριθμ;
end αριθμητής;

-- σώμα την συνάρτησης παρονομαστής
function παρονομαστής (x: κλάσμα) return integer is
begin
  return x.παρον;
end παρονομαστής;

function '+' (x, y: κλάσμα) return κλάσμα is
N: integer
D: positive;
begin
  N:= αριθμητής(x) * παρονομαστής(y) + αριθμητής(y) *
                                       παρονομαστής(x);
  D:= παρονομαστής(x) * παρονομαστής(y);
  return αναγωγή (δημιουργία_κλάσματος(N, D));
end '+';

-- ακολουθούν τα σώματα των συναρτήσεων '-', '*', '/'

function ισα (x, y: κλάσμα) return boolean is
begin
  return αριθμητής(x) * παρονομαστής(y) = αριθμητής(y) *
                                          παρονομαστής (x);
end ισα;

-- ακολουθούν τα σώματα των συναρτήσεων '<', '>=', και αναγωγή

end κλάσματα;
```

Επειδή ο ορισμός του πακέτου είναι ορατός από το χρήστη, δηλαδή ο χρήστης γνωρίζει ότι η υλοποίηση του κλάσματος γίνεται

με μια εγγραφή, αυτός μπορεί να γράψει εντολές, όπως την:

```
n:= y.αριθμ;
```

αντί να χρησιμοποιήσει τη συνάρτηση "αριθμητής", που δίνεται στον ορισμό του πακέτου.

Η γνώση από το χρήστη του τρόπου με τον οποίο υλοποιείται μία δομή μπορεί όμως να δημιουργήσει προβλήματα. Για παράδειγμα, σε περίπτωση που η δομή μπορούσε ν'αλλάξει υλοποίηση, είναι πιθανό να θέλαμε ν'αλλάξουμε μόνο τον ορισμό και τις πράξεις της δομής, χωρίς να επέμβουμε στο πρόγραμμα που τη χρησιμοποιεί. Ευτυχώς, η Ada διαθέτει για το σκοπό αυτό έναν ειδικό τύπο, που ονομάζεται **ιδιωτικός (private)**. Στο ορατό μέρος του πακέτου μπορούμε να δηλώσουμε ότι ένας τύπος είναι private, χωρίς να δώσουμε αμέσως τις λεπτομέρειες της υλοποίησής του και στο δεύτερο μέρος των ορισμών, που ονομάζεται ιδιωτικό μέρος, δίνουμε τις λεπτομέρειες που ορίζουν τον τύπο αυτόν. Ο ορισμός του πακέτου "κλάσματα" στην περίπτωση αυτή μπορεί να γίνει ως εξής:

```
package κλάσματα is

type κλάσμα is private;

function δημιουργία_κλάσματος (N, D: integer) return integer;
function αριθμητής (x: κλάσμα) return integer;
function παρονομαστής (x: κλάσμα) return integer;
function '+' (x, y: κλάσμα) return κλάσμα;
function '-' (x, y: κλάσμα) return κλάσμα;
function '*' (x, y: κλάσμα) return κλάσμα;
function '/' (x, y: κλάσμα) return κλάσμα;
function ίσα (x, y: κλάσμα) return boolean;
function '<' (x, y: κλάσμα) return boolean;
function '>=' (x, y: κλάσμα) return boolean;
function αναγωγή (x: κλάσμα) return κλάσμα;

private
   type  κλάσμα is
     record
       αριθμ: integer;
       παρον: positive;
     end record;

end κλάσματα;
```

Στην περίπτωση αυτή, όταν ο χρήστης του πακέτου γράφει στο πρόγραμμά του μια εντολή, όπως την y.αριθμ, τότε αυτή θα θεωρηθεί

ως λανθασμένη, κατά τη διάρκεια μεταγλώττισης του προγράμματος. Έτσι, παρ'όλο που η δομή είναι "ορατή" από τον ορισμό του πακέτου και γνωρίζουμε τον τρόπο υλοποίησής της, δεν έχουμε τη δυνατότητα να κάνουμε χρήση αυτής της γνώσης. Στην περίπτωση αυτή, ο μόνος τρόπος να έχουμε την τιμή y.αριθμ είναι με τη βοήθεια της συνάρτησης αριθμητής(y).

Όταν ένας τύπος είναι δηλωμένος ως private, τότε οι μόνες πράξεις που μπορεί να χρησιμοποιήσει ο χρήστης είναι η εκχώρηση, το =, το /= και αυτές που περιλαμβάνονται υπό μορφή υποπρογραμμάτων στο ορατό μέρος του πακέτου. Μπορούμε τέλος να δηλώσουμε σταθερές ενός ιδιωτικού τύπου. Στις σταθερές αυτές δεν μπορούμε να δώσουμε όμως αρχικές τιμές, καθόσον δεν είναι ακόμη γνωστός ο τρόπος υλοποίησής τους. Η εκχώρηση αρχικής τιμής σε σταθερές γίνεται στο ιδιωτικό μέρος του ορισμού του πακέτου.

Οι πράξεις πάνω σ'έναν ιδιωτικό τύπο μπορούν επίσης να περιοριστούν σ'αυτές ακριβώς που υπάρχουν στο ορατό μέρος του ορισμού του τύπου. Αυτό γίνεται ορίζοντας το τύπο ως **περιορισμένο (limited)** και **ιδιωτικό (private)**. Στη περίπτωση αυτή, οι πράξεις, εκχώρηση, = και /= δεν μπορούν να χρησιμοποιηθούν έξω από το πακέτο. Η απουσία της εκχώρησης για έναν περιορισμένο τύπο σημαίνει πως δεν μπορούμε να δώσουμε αρχική τιμή στον ορισμό ενός **αντικειμένου (object)**. Αυτό συνεπάγεται πως δεν μπορούμε να δηλώσουμε μια σταθερά έξω από το πακέτο. Οπωσδήποτε, όμως, στο πακέτο μπορούμε να ορίσουμε μια συνάρτηση '=', αν οι δύο παράμετροί της είναι του ίδιου περιορισμένου τύπου (limited type). Τέλος όταν ένας τύπος είναι limited private, τότε ο προγραμματιστής έχει πλήρη έλεγχο πάνω στα στοιχεία του τύπου αυτού.

Τελειώνοντας θα περιγράψουμε συνοπτικά πως αντιμετωπίζει η γλώσσα Ada την περίπτωση κάποιου σφάλματος **(error handling)** κατά την εκτέλεση ενός προγράμματος. Όπως γνωρίζουμε, όταν παραβιάσουμε τους κανόνες μιας γλώσσας, τότε το πρόγραμμα σταματάει. Σχετικά παραδείγματα,περιλαμβάνουν τη περίπτωση που ο δείκτης ενός πίνακα έχει ξεπεράσει τα όριά του, τη διαίρεση με μηδέν, την υπερχείλιση, την υποχείλιση κ.λπ.

Για τις περιπτώσεις αυτές, όταν μπορούμε να προβλέψουμε ότι μπορεί να συμβεί μια **"εξαίρεση" (exception)** σ'ένα μέρος ενός προγράμματος, τότε μπορούμε να χρησιμοποιήσουμε έναν ειδικό μηχανισμό για να ξεπεράσουμε τη δυσκολία αυτή, χωρίς να σταματίσει το πρόγραμμά μας. Αυτό γίνεται ως εξής:

```
begin
-- ακολουθία από εντολές;
exception
 when σφάλμα =>
 -- αντιμετώτπισε το σφάλμα
end;
```

Αν στο παραπάνω απόσπασμα προγράμματος συμβεί ένα σφάλμα στην ακολουθία των εντολών μεταξύ των ειδικών λέξεων begin και exception, τότε η ροή του προγράμματος διακόπτεται και ο έλεγχος μεταφέρεται στην ακολουθία των εντολών μετά το =>. Η εντολή when λέγεται **διαχειριστής της εξαίρεσης (exception handler)**. Μια εξαίρεση δηλώνεται με τον ίδιο τρόπο όπως οι μεταβλητές π.χ.

σφάλμα: exception;

8.4 Υλοποίηση στοίβας με πίνακες.

Με όλα τα νέα στοιχεία της Ada, που πολύ συνοπτικά περιγράψαμε παραπάνω, θα προσπαθήσουμε να ορίσουμε έναν ΑΤΔ για στοίβες.

```
-- Ορισμός στοίβας:
package στοίβες is
στοίβα_γεμάτη, στοίβα_κενή: exception;
type στοίβα is limited private;
procedure create (s: in out στοίβα);
function empty (s: in στοίβα) return boolean;
function full (s: in στοίβα) return boolean;
procedure push (s: in out στοίβα; x: in integer);
procedure pop (s: in out στοίβα; x: out integer);
function '=' (s, t: στοίβα) return boolean;
private
  max: constant:= 100;
  type διάνυσμα is
   array (integer range <>) of integer;
   type στοίβα is
    record
        s: διάνυσμα (1..max);
        κορυφή: integer range 0..max:= 0;
    end record;
end στοίβες;
```

Η μεταβλητή "κορυφή" έχει ως τιμή ορισμού (default) το μηδέν. Αυτό σημαίνει ότι με τη δήλωσή της η στοίβα είναι κενή. Η συνάρτηση '=' επιστρέφει την τιμή true, μόνον όταν δύο στοίβες έχουν το ίδιο πλήθος στοιχείων και τα αντίστοιχα στοιχεία τους είναι ίσα. Θα ήταν λάθος να συγκρίνει κανείς όλες τις εγγραφές των διανυσμάτων με τα οποία υλοποιούνται οι δύο στοίβες, γιατί στη περίπτωση αυτή θα λάβαινε υπόψη του όλα τα στοιχεία που υπάρχουν στα δύο διάνυσματα, ακόμη και αυτά που δε χρησιμοποιούνται από τις στοίβες.

```
-- σώμα πακέτου στοίβες
package body στοίβες is
procedure create (s: in out στοίβα) is
begin
  s.κορυφή:= 0;
end create;

function empty (s: in στοίβα) return boolean is
begin
  return s.κορυφή = 0;
end empty;

function full (s: in στοίβα) return boolean is
begin
  return s.κορυφή = max;
end;

procedure push (s: in out στοίβα; x in integer) is
begin
  if full(s)  then
    raise στοίβα_γεμάτη;
    else
    s.κορυφή:= s.κορυφή + 1;
    s.s(s.κορυφή):= x;
  end if;
end push;

procedure pop (s: in out στοίβα; x out integer) is
begin
  if empty (s) = 0 then raise στοίβα_κενή;
  else
  x:= s.s(s.κορυφή);
  s.κορυφή:= s.κορυφή-1;
  end if;
end pop;

function '=' (s, t: στοίβα) return boolean is
begin
  if s.κορυφή /= t.κορυφή then
     return false;
  end if;
 for i in 1.. s.κορυφή
 loop
   if s.s(i) /= t.s(i) then
     return false;
   end if;
```

```
    end loop;
     return true;
    end '=';
   end στοιβες;
```

Αυτός είναι και ο λόγος που ο τύπος στοιβα έχει δηλωθεί ως limited private και όχι απλώς ως private. Αν η "στοιβα" έχει δηλωθεί μόνο ως private, τότε δε θα μπορούσαμε να ορίσουμε το '=' ώστε ν'αποδίδει τη σωστή του έννοια. Αυτό είναι ένα τυπικό παράδειγμα μιας δομής δεδομένων της οποίας η "τιμή" δεν αποτελείται από το "άθροισμα" των στοιχείων της. Πράγματι η τιμή του διανύσματος s εξαρτάται από την τιμή της μεταβλητής "κορυφή".

Μια τελευταία παρατήρηση είναι ότι η μεταβλητή s έχει χρησιμοποιηθεί με δύο διαφορετικές έννοιες: ως τ'όνομα της τυπικής παραμέτρου, δηλώνοντας μια στοιβα, και ως το διάνυσμα μέσα στην εγγραφή του ορισμού της στοιβας. Αυτό όμως δε δημιουργεί πρόβλημα, γιατί, αν και τα πεδία επιρροής (scope) τους επικαλύπτονται, οι περιοχές της "ορατότητας" (regions of visibility) τους δεν επικαλύπτονται. Μπορούμε να γράψουμε τώρα μια ενότητα προγράμματος ως εξής:

```
declare
  use στοιβες;
begin
  ...
  push(s, m);
  ...
  y:= pop(s, x);
  ...
  exception
   when στοιβα_γεμάτη =>
            -- η στοιβα είναι γεμάτη
            .
            .
            .
   when στοιβα_κενή =>
            -- η στοιβα είναι κενή
            .
            .
            .
   when others =>
            -- κάτι άλλο δεν πήγε καλά
            .
            .
            .
end;
```

8.5 Γενικά πακέτα (Generic packages)

Το πρόβλημα με το πακέτο "στοίβες" παραπάνω, είναι πως μπορεί να χρησιμοποιηθεί μόνο όταν τα στοιχεία της στοίβας είναι ακέραιοι, αν και η ίδια λογική εφαρμόζεται και για στοιχεία οποιουδήποτε άλλου τύπου. Αυτό ακριβώς συμβαίνει και στην Pascal, όπου ο μόνος τρόπος είναι να επεμβουμε στο πρόγραμμα, ν'αλλάξουμε τον τύπο από integer σε real, για παράδειγμα και να ξαναμεταγλωττίσουμε ολόκληρο το πρόγραμμα. Η Ada όμως μας δίνει τη δυνατότητα να γράψουμε γενικά ή δημιουργικά (generic) πακέτα. Αυτό γίνεται ως εξής:

```
generic
  max: positive;
  type τύπος_στοιχείου is private;
  package στοίβες is
     .
     .
     .
  end στοίβες;
  package body στοίβες is
     .
     .
     .
  end στοίβες;
```

Από το γενικό αυτό πακέτο μπορούμε να ορίσουμε τώρα μια νέα στοίβα με συγκεκριμένο μέγεθος και τύπο στοιχείων ως εξής:

```
declare
 package νέα_στοίβα is new στοίβες (100, real);
 use νέα_στοίβα
begin
   .
   .
   .
 push(s, x);
   .
   .
end;
```

Το πακέτο νέα_στοίβα συμπεριφέρεται ακριβώς σαν ένα νέο πακέτο. Η χρήση της εντολής use μας επιτρέπει να χρησιμοποιήσουμε απλά τα αναγνωριστικά push και pop χωρίς να χρειάζεται να γράφουμε νέα_στοίβα.push και νέα_στοίβα.pop.

Συμπεράσματα

Αν και η ανάγνωση αυτού του κεφαλαίου προυποθέτει ίσως γνώσεις από τη γλώσσα Ada, σκοπός μας ήταν να δείξουμε μερικές από τις δυνατότητες της γλώσσας αυτής για τον ορισμό ΑΤΔ. Πράγματι τα πακέτα της Ada είναι ίσως ο καλύτερος μηχανισμός, που μπορεί να ευρεθεί μεταξύ των πλέον διαδεδομένων γλωσσών προγραμματισμού για την υλοποίηση ενός ΑΤΔ. Ενα άλλο επίσης σημαντικό πλεονέκτημα των πακέτων είναι ότι μπορούν να μεταγλωττιστούν ξεχωριστά και να τοποθετηθούν σε μια βιβλιοθήκη για μελλοντική χρήση. Ο ορισμός και το σώμα ενός πακέτου μπορούν να μεταγλωττιστούν ξεχωριστά το ένα από το άλλο. Αυτό σημαίνει ότι όταν αλλάξει κάτι στο σώμα του πακέτου, μόνο αυτό χρειάζεται να ξαναμεταγλωττιστεί κι όχι ολόκληρο το πρόγραμμα και το ίδιο ισχύει όταν αλλάξει κάτι στον ορισμό του πακέτου. Δυστυχώς συχνότερα αλλάζει ο ορισμός του πακέτου,ο οποίος περιέχει τον τύπο του ΑΤΔ, πράγμα που σημαίνει ότι το πρόγραμμα θα πρέπει να ξαναμεταγλωττιστεί.

ΛΥΣΕΙΣ ΑΣΚΗΣΕΩΝ ΚΑΤ'ΕΠΙΛΟΓΗ

Ασκήσεις 3.1

2.
```
function  legal (s: strg): boolean;
var x, ch: char; correct: boolean;
begin
  ch:= s[1]; i:=1;
  while (ch <> '*') and (ch <> '$') and not full(stack) do
  begin
    push (stack, ch);
    i:= i + 1;
    ch:= s[i]
  end;
  if (ch = '$') or full (stack) and (ch <> '*') then
  legal:= false
  else
  begin
    i:= i + 1; correct:= true;
    ch:= s[i];
    while (ch <> '$') and correct and not empty (stack) do
    begin
      pop (stack, x);
      correct:= x = ch;
      i:= i + 1; ch:= s[i]
    end
   if correct and empty(stack) and (ch = '$') then
    legal:= true
    else legal := false
   end
end
```

3.
```
function mikos_ouras (oura: typos_ouras): integer;
var  megethos_ouras: integer;
begin
  with oura do
    begin
      megethos_ouras:= panw_orio - katw_orio + 1;
      if pisw_deiktis > empros_deiktis then
       mikos_ouras:= pisw_deiktis - empros_deiktis
      else
       if pisw_deiktis < empros_deiktis  then
           mikos_ouras:= megethos_ouras - empros_deiktis +
                                                  pisw_deiktis
```

```
            else
             if pisw_deiktis = katw_orio - 1  then
                 mikos_ouras:= 0
            else
            mikos_ouras:= megethos_ouras
      end
  end
```

8.
```
  Program Vector_List (input, output);
  uses crt;
  const
    max = 20;
  type
   list_pointer = -1..max;
   stack_pointer = -1..max;
    list_limits = 0..max;
     data_type = integer;
        node = record
                  data: data_type;
                  link: list_pointer
                end;
   list_type = array [list_limits] of node;
   lists = record
             head1: list_pointer;
             head2: stack_pointer;
    total_records: list_limits
                s: list_type
       end;

 var  list: lists;
data_in: data_type;
     ch: char;

 procedure create (var list: lists);
 begin
   with list do
    begin
      head2:= -1; {Στοίβα διαθεσίμων κενή}
      head1:= -1; {Λίστα κενή}
      total_records:= 0
    end
 end; {create}

 procedure push (var list: lists; incomer: list_limits);
 begin
```

```
 with list do
  begin
    s[incomer].link:= head2;
    head2:= incomer
  end
end; {push}

procedure pop (var list: lists; var outcomer: list_limits);
begin
  with list do
   begin
     outcomer:= head2;
     head2:= s[outcomer].link
   end
end; {pop}

function empty_stack (list: lists): boolean;
begin
  empty_stack:= list. head2 = -1
end;{empty_stack}

function empty_list (list: lists): boolean;
begin
  empty_list:= list.head1 = -1 {ή empty_list:= list.
                                total_records = 0}

end;{empty_list}

function full_list (list: lists): boolean;
begin
  full_list:= list.total_records = max
end;{full_list}

procedure insert (var list: lists; newcomer: data_type);
var
    thesi_in: list_limits;
          j: list_pointer;
begin
  with list do
   begin
     {Βρες μεταξύ ποιών κόμβων της λίστας μπαίνει η νέα
                                                 εγγραφή}

     j:= head1;
     if j <> -1 then
        while (s[j].data < newcomer) and (s[j].link <> -1) do
            j:= s[j].link;
     {Βρες σε ποιά θέση του πίνακα μπαίνει η νέα εγγραφή}
```

```
      if empty_stack(list) then thesi_in:= total_records
      else pop(list, thesi_in);
      {Βάλε την νέα εγγραφή και κάνε τις σωστές συνδέσεις}
      if j <> -1 then
       if s[j].data < newcomer then {πρέπει να μπει τελευταία}
        begin
          s[thesi_in].data:= newcomer;
          s[thesi_in].link:= -1;
          s[j].link:= thesi_in
        end
       else {πρέπει να μπει πρώτη ή ενδιάμεσα}
        begin
          s[thesi_in].data:= s[j].data;
          s[j].data:= newcomer;
          s[thesi_in].link:= s[j].link;
          s[j].link:= thesi_in;
        end
     else {j = -1}
      begin
        s[thesi_in].data:= newcomer;
        s[thesi_in].link:= -1;
        head1:= thesi_in
      end;
      {Αύξησε τις ολικές εγγραφές}
      total_records:= total records + 1
    end
end; {insert}

procedure delete(var list: lists; out: data_type);
var
  prev, j: list_limits;
begin
  with list do
   begin
     {Βρες την εγγραφή που θα διαγραφεί}
     j:= head1;
     while (s[j].data < out) and (s[j].link <> -1) do
      begin
        prev:= j;
        j:= s[j].link
      end;
     if s[j].data = out then {βρέθηκε στην λίστα}
      begin
        total_records:= total_records - 1;
```

```
          if j <> head1 then {είναι ενδιάμεσα ή στο τέλος}
           begin
             s[prev].link:= s[j].link;
             push (list, j)
           end
          else {είναι το πρώτο}
           begin
             head1:= s[j].link;
             push (list, j)
           end
        end
       else {s[j].data <> out}
        writeln ('Not found', out)
     end
 end; {delete}

procedure print_list (list: lists);
var
   j: list_pointer;
begin
  with list do
   begin
     j:= head1;
     writeln('     ********************** Λίστα: ********* ');
     writeln('     Κεφαλή λίστας ', j);
     while j <> -1 do
      begin
        write ('          θέση Πίνακα: ', j);
        writeln(' Περιεχόμενο: ', s[j].data,' σύνδεση με:
                                              's[j].link);
        j:= s[j].link
      end;
     writeln('    ****************************************** ')
   end
end; {print_list}

procedure print_stack(list: lists);
var
   j: stack_pointer;
begin
  with list do
   begin
     writeln('       Κεφαλή στοίβας:', head2);
     write('        Στοίβα διαθεσίμων:');
     if head2 <> -1 then
      begin
```

```
          j:= head2;
          while  j <> -1 do
           begin
             write(' ', j);
             j:= s[j].link
           end
        end
       else write(' κενή ');
     end;
     writeln
  end; {print_stack}

  begin {main}
    clrscr; checkeof:= true;
    create(list);
    write('Εντολή: ');
    while not eof do
     begin
       readln(ch, data_in);
       case ch of
        'i', 'I': if not full_list(list) then
                  insert (list, data_in)
                  else writeln('Η λίστα είναι ήδη γεμάτη !!!');
        'd', 'D': if not empty_list(list) then
                  delete(list, data_in)
                  else writeln('Τι να διαγράψω από μια άδεια
                                                      στοίβα;')
        else writeln('Εντολή ', ch',δεν αναγνωρίζεται από το
                                                    πρόγραμμα')
        end;
        print_list(list);
        print_stack(list);
        writeln;
        write('Εντολή:')
       end
     end {vector_List}


9. {γκαράζ και αποθήκη είναι δύο στοίβες}
    δημιουργία (γκαράζ); δημιουργία (αποθήκη);
    read (γεγονός); {Η μεταβλητή γεγονός σημαίνει άφιξη/αναχώρηση
                                    αυτοκινήτου στο/από το γκαράζ}
     while not eof (input) do
     begin
       readln (αριθμός_αυτοκινήτου);
       if γεγονός είναι άφιξη then
```

```
        if not γεμάτο (γκαράζ) then
        begin
          with αυτοκίνητο  do
           begin
             αριθμός:= αριθμός_αυτοκινήτου;
             μετακινήσεις:= 0
           end
          εισάγαγε (αυτοκίνητο, γκαράζ)
        end
        else
        writeln ('Το γκαράζ είναι γεμάτο')
      else {γεγονός είναι αποχώρηση}
      begin
        βρέθηκε_αυτοκίνητο:= false;
        while not άδειο (γκαράζ) and
                 (not βρέθηκε_αυτοκίνητο) do
        begin
          εξάγαγε (αυτοκίνητο, γκαράζ);
          βρέθηκε_αυτοκίνητο:= αυτοκίνητο.αριθμός =
                                      αριθμός_αυτοκινήτου;
          if not βρέθηκε_αυτοκίνητο  then
          εισάγαγε (αυτοκίνητο, αποθήκη)
        end;
         if  βρέθηκε αυτοκίνητο then
          writeln ('έφυγε το', αυτοκίνητο, '.'
             'Ο αριθμός μετακινήσεων ήταν', μετακινήσεις)
         else
         writeln ('Το αυτοκίνητο', αριθμός_αυτοκινήτου,'δεν
                                    υπάρχει στο γκαράζ');
         while not άδεια (αποθήκη) do
          begin
            εξάγαγε (αυτοκίνητο, αποθήκη);
            αυτοκίνητο.μετακινήσεις:= αυτοκίνητο.μετακινήσεις + 1;
            εισάγαγε (αυτοκίνητο, γκαράζ)
          end
      end; {αποχώρησης αυτοκινήτου}
      read (γεγονός)
    end {not eof}
```

Ασκήσεις 3.2

```
6. procedure  epanalamvanomenes_times(p: list_pointer);
   var epanalipsi, synexise: boolean;
         prohgoumeni_timi: kleidi;
```

```
begin
  epanalipsi:= false;
  if p <> nil then
   begin
     prohgoumeni_timi:= p^.data;
     p:= p^.next;
     while p <> nil do
     if p^.data = prohgoumeni_timi then
       begin
         epanalipsi:= true;
         writeln (p^.data);
         p:= p^.next; synexise:= true;
         while (p <> nil) and synexise do
         if p^.data = prohgoumeni_timi then
          p:= p^.next
         else
          synexise:= false;
         if p <> nil then
         begin
           prohgoumeni_timi:= p^.data;
           p:= p^.next
         end
       end
     else
     begin
       prohgoumeni_timi:= p^.data;
       p:= p^.next
     end
   end;
  if not epanalipsi then
  writeln ('Δεν υπάρχει επαναλαμβανόμενο στοιχείο')
end
```

12.

```
procedure insert (var lista: listpointer;
                          x: element_type);
  var p, prohgoumenos: listpointer;
  begin
    p:= lista;  prohgoumenos:= nil;
    if p <> nil then
     begin
       while (p^.data <> x) and (p^.next <> nil) do
        begin
          prohgoumenos:= p;
          p:= p^.next
        end;
```

```
        if p^.data = x then
        begin
        if prohgoumenos <> nil then
         begin
           p^.data:= prohgoumenos^.data;
           prohgoumenos^.data:= x
         end {διαφορετικά το στοιχείο x είναι ήδη στην αρχή
                                                της λίστας}
        end
         else
       begin
         new (p^.next);   p^.next^.data:= x;
         p^.next^.next:= nil
       end
     end {p <> nil}
      else {η λίστα είναι κενή}
      begin
        new (lista);    lista^.data:= x;
        lista^.next:= nil
      end
end
```

14.
```
function equal (var l1, l2:  listpointer): boolean;
var p1, p2: listpointer;
    m1, m2: integer;
procedure delete (var head, l: listpointer);
var p: listpointer;
begin
  if head = l   then
   begin
     head:= l^. next; dispose (l);
     l:= head
   end
 else
  if l^.next = nil then
  begin
    p:= head;
    while p^.next <> l do   p:= p^.next;
    p^.next:= nil;
    dispose (l);
    l:= nil
  end
 else
  begin
    p:= l^.next; l^:= l^.next^;
```

```
      l:= l^.next; dispose(p)
    end
  end

  begin
    p1:= l1; p2:= l2; m1:= 0; m2:= 0;
    while (p1 <> nil) and (p2 <> nil) do
    if p1^.data = p2^.data then
    begin
      delete (l1, p1);
      delete (l2, p2)
    end
    else
    if p1^.data < p2^.data then
    begin
      p1:= p1^.next;
      m1:= m1 + 1
    end
    else
     begin
       p2:= p2^.next; m2:= m2 + 1
     end;
     while p1 <> nil do
     begin
       p1:= p1^.next; m1:= m1 + 1
     end;
     while p2 <> nil do
      begin
        p2:= p2^.next; m2:= m2 + 1
      end
     equal:= m1 = m2
  end
```

16.
```
  function sum (a, b: listpointer): listpointer;
  var head, p: listpointer;
  begin
    new (head); p:= head;
    while (a <> nil) and (b <> nil) do
     begin
       new (p^.next); p:= p^.next;
       if a^.thesi = b^.thesi  then
        begin
          p^.thesi:= a^.thesi;
          p^.data:= a^.data + b^.data
```

```
          a:= a^.next;
          b:= b^.next;

        end
       else
       if a^.thesi < b^.thesi then
        begin
          p^.thesi:= a^.thesi;
          p^.data:= a^.data;
          a:= a^.next
        end
      else
       begin
         p^.thesi:= b^.thesi;
         p^.data:= b^.data;
         b:= b^.next
       end
    end;
while a <> nil do
  begin
    new (p^.next); p:= p^.next;
    p^.data:= a^.data;
    p^.thesi:= a^.thesi;
    a:= a^.next
  end;
  while b <> nil do
   begin
     new (p^.next); p:= p^.next;
     p^.data:= b^.data;
     p^.thesi:= b^.thesi;
     b:= b^.next
   end;
   p^.next:= nil;
   sum:= head^.next;
   dispose (head)
end


function inner_product (a, b: listpointer): real;
var s: real;

begin
  s:= 0.0;
  while (a <> nil) and (b <> nil) do
  if a^.thesi = b^.thesi then
   begin
```

```
        s:= s + a^.data * b^.data;
        a:= a^.next;
        b:= b^.next
      end
    else
       if a^.thesi < b^.thesi then
        a:= a^.next
       else
        b:= b^.next;
   inner_product:= s
end {inner_product}
```

Ασκήσεις 4.1

3.
```
procedure onelevel (level: integer; t: treepointer);
begin
  if t <> nil then
    if level = given_level then
     write (t^.data, ' ')
  else
   begin
     onelevel (level + 1, t^.left_child);
     onelevel (level + 1, t^.right_child)
   end
end
```

κλήση: onelevel (1, root)

8.
```
procedure findparent (t, p: treepointer; var father:
                      treepointer; var found: boolean);
begin
if t <> nil then
  if t^.data = x then    {το x είναι ορισμένο καθολικά}
  begin
    father:= p;
    found:= true
  end
  else
   begin
     findparent (t^.leftchild, t, father, found);
     if not found then
     findparent (t^.rightchild, t, father, found)
   end
end
```

κλήση: findparent (root, nil, father, found)

```
10. function equal (p, q: treepointer): boolean;
    var w: boolean;
    begin
      if (p = nil) and (q = nil) then equal:= true
      else
      begin
        if (p <> nil) and (q <> nil) then
        if p^.data = q^.data then
        begin
          w:= equal (p^.leftchild, q^.leftchild);
          if w then w:= equal (p^.rightchild, q^.rightchild);
          equal:= w
        end
        else
          equal:= false {διαφορετικό περιεχόμενο στους κόμβους}
        else
          equal:= false {διαφορετικό σχήμα δένδρου}
      end
    end

11. type operators = (mynot, myand, myor, conclude);
          datum = record
                    case typos: boolean of
                    true: (data: boolean);
                    false: (operator: operators)
                  end;
          treepointer = ^treenode;
           treenode = record
                        element: datum;
                        leftchild, rightchild: treepointer
                      end;
        function conclude (x, y: boolean): boolean;
        begin
          if  (x and not y) then  conclude:= false
          else    conclude:= true
        end;

        procedure logical_expression (t: treepointer;
                                      var result: boolean);
        var temp1, temp2: boolean;
        begin
          if t^.element.typos then result:=t^.element.data
          else
```

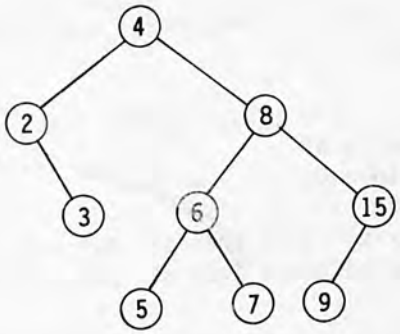
```
      case t^.element.operator of
      mynot: begin
               logical_expression (t^.leftchild, temp1);
               result:= not (temp1)
             end;
      myand: begin
               logical_expression (t^.leftchild, temp1);
               logical_expression (t^.rightchild, temp2);
               result:= temp1 and temp2
             end;
      myor : begin
               logical_expression (t^.leftchild, temp1)
               logical_expression (t^.rightchild, temp2);
               result:= temp1 or temp2
             end;

      conclude: begin
                  logical_expression (t^.leftchild, temp1);
                  logical_expression (t^.rightchild, temp2);
                  result:= conclude (temp1, temp2)
                end
      end {case}
    end
```

15.
```
procedure brother (p: treepointer; var found: boolean;
                                   var q: treepointer);
  begin
    if p^.leftchild = t  then
      begin
        q:= p^.rightchild;
        found:= true
      end
    else
      if p^.rightchild = t then
        begin
          q:= p^.leftchild;
          found:= true
        end
      else
        begin
          if (p^.leftchild <> nil) and not found  then
            brother(p^.leftchild, found, q);
          if (p^.rightchild <> nil) and not found  then
            brother(p^.rightchild,found, q)
        end
  end
```

Αρχικές τιμές: found:= false
Κλήση: brother(root,found,q)

Ασκήσεις 5.1

1α.



2.

```
program avl_trees_insertion;

{$A-}

type

balance=-1..1;
pointer=^node;
node=record
       data:integer;
       bal:balance;
       left,right:pointer;
     end;

var

tree,tree1,p1,p2, p3,p4 :pointer;
x:integer;
ch:char;
brethike,found1:boolean;


procedure reset_balances(var t:pointer;x1,choice:integer);

{ Η διαδικασία αυτή υπολογίζει τις νέες τιμές της μεταβλητής  )
```

```
{ bal:balance ανάλογα με το είδος περιστροφής }

var
   temp:pointer;
   found:boolean;

begin
  if ((choice=4) or (choice=5)) and (x1=p4^.data)  then
  p2^.bal:=0
  else
  begin
    case choice of
      1:begin end;
      2,3:p2^.bal:=0;
      4:if x>p4^.data then p3^.bal:=-1 else p2^.bal:=1;
      5:if x>p4^.data then p2^.bal:=-1 else p3^.bal:=1;
    end;
    temp:=t;
    temp^.bal:=0;
    found:=false;
    if x1>temp^.data   then     temp:=temp^.right
    else  temp:=temp^.left;
    while (temp<>nil) and (not found) do
    begin
      if temp^.data<x1 then  {εισαγωγή στα δεξιά του στοιχείου}
      begin
        temp^.bal:=1;
        temp:=temp^.right;
      end
      else
      if temp^.data>x1 then {εισαγωγή στ'αριστερά του στοιχείου}
      begin
        temp^.bal:=-1;
        temp:=temp^.left;
      end
      else
      found:=true;
    end;
  end;
end;


procedure set_pointers(choice:integer);

{ Η διαδικασία αυτή βάζει τους απαραίτητους δείκτες ανάλογα με }
{ το είδος της περιστροφής }
```

```
begin

  case choice of
     2:p3:=p2^.left; { left single rotation }
     3:p3:=p2^.right;{ right single rotation }
     4: begin
          p3:=p2^.left; { left double rotation }
          p4:=p2^.left^.right;
        end;
     5: begin
          p3:=p2^.right; { right double rotation }
          p4:=p2^.right^.left;
        end;
  end;
end;


procedure insert(var t,p,fp:pointer;x2:integer;var exists:boolean);

{ Η διαδικασια αυτή κάνει εισαγωγή του στοιχειου στο δένδρο και }
{ επιστρέφει τους δεικτες p2 και p1. Αν το στοιχείο υπάρχει ήδη }
{ στο δένδρο,δεν κάνει εισαγωγή και η μεταβλητή exists έχει τιμή}
{true}
var
    r,q,s:pointer;

begin
  r:=t;
  exists:=false;
  p:=nil;
  fp:=nil;
  if  r=nil   then   { άδειο δέντρο }
  begin
    new(s);
    s^.data:=x2;
    s^.bal:=0;
    s^.left:=nil;
    s^.right:=nil;
    t:=s;
  end
  else
  begin
    if r^.bal<>0  then   p:=r;
    while (r<>nil) and (not exists) do
    begin
      q:=r;
```

```
      if r^.data<x2   then   r:=r^.right
      else
      if r^.data>x2  then   r:=r^.left
      else
      exists:=true;
      if   r<>nil   then
      if r^.bal<>0 then
      begin
        fp:=Q;
        p:=r;
      end;
    end;{ while }
    if not exists  then
    begin
      new(s);
      s^.data:=x2;
      s^.bal:=0;
      s^.left:=nil;
      s^.right:=nil;
      { σύνδεση με το υπόλοιπο δέντρο }
      if   x2>q^.data   then   q^.right:=s   else   q^.left:=s;
    end;
  end; { else if r<>nil }
end;


procedure single_rotation(var t,p1,p2,p3:pointer;
                                  aristera:boolean);
{ το p1 δείχνει στο father_of_pivot }
{ το p2 δείχνει στο pivot   }
{ και το p3 δείχνει στο pivot_child }

begin
 if p1<>nil then
  begin
   if p1^.data<x  then
     p1^.right:=p3
   else
     p1^.left:=p3;
  end
 else
  t:=p3;
  if aristera  then
    begin
     p2^.left:=p3^.right;
     p3^.right:=p2;
```

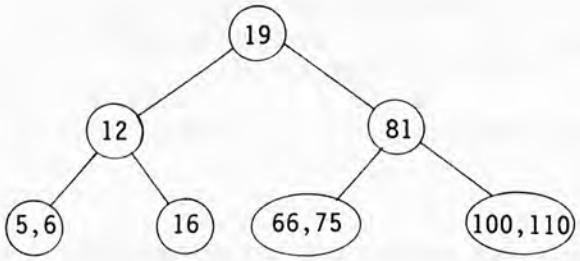
```
    end
   else
    begin
     p2^.right:=p3^.left;
     p3^.left:=p2;
    end;
end;


procedure double_rotation(var t,p1,p2,p3,p4:pointer;
                                    aristera:boolean);
{ το p4 δειχνει το p_c_child }

begin
  if p1<>nil  then
  begin
    if p1^.data>x then p1^.left:=p4 else
    if p1^.data<x then p1^.right:=p4
  end
  else
  t:=p4;
  if aristera  then
  begin
    p3^.right:=p4^.left;
    p2^.left:=p4^.right;
    p4^.right:=p2;
    p4^.left:=p3;
  end
  else
  begin
    p3^.left:=p4^.right;
    p2^.right:=p4^.left;
    p4^.left:=p2;
    p4^.right:=p3;
  end;
end;


procedure avl_insert;

{ Κύρια διαδικασία,που υλοποιεί τον αλγόριθμο εισαγωγής σε }
{ AVL δένδρο,χρησιμοποιώντας τις άλλες βοηθητικές ρουτίνες.}
{ Η διαδικασία κάνει εισαγωγή του στοιχείου στο δένδρο και }
{ αποφασίζει για το αν θα γίνει περιστροφή ή όχι στο       }
{ δένδρο, καθώς και το είδος της περιστροφής.              }
```

```
var
   temp:pointer;

begin
  insert(tree,p2,p1,x,brethike);
  if not brethike  then
  begin
    if p2=nil   then
    { δεν υπάρχει pivot άρα όλα τα balances είναι μηδέν }
    begin
      temp:=tree;
      while temp^.data<>x do
      begin
        if temp^.data<X  then
        begin
          temp^.bal:=1;
          temp:=temp^.right;
        end
        else
        begin
          temp^.bal:=-1;
          temp:=temp^.left;
        end;
      end;
    end
    else
    if ((p2^.bal=-1) and (x>p2^.data)) or ((p2^.bal=1)
    and (x<p2^.data))  then (*εισαγωγή στο μικρό υπόδεντρο*)
    reset_balances(p2,x,1)
    else
    if (p2^.bal=-1) and (x<p2^.data) and (x<p2^.left^.data)  then
    { left single rotation }
    begin
      set_pointers(2); write('mp');
      single_rotation(tree,p1,p2,p3,true);
      reset_balances(p3,x,2);
    end
    else
    if (p2^.bal=1) and (x>p2^.data) and (x>p2^.right^.data)  then
    { right single rotation }
    begin
      set_pointers(3);
      single_rotation(tree,p1,p2,p3,false);
      reset_balances(p3,x,3);
    end
    else
```

```
    if (p2^.bal=-1) and (x<p2^.data) and (x>p2^.left^.data)  then
    { left double rotation }
    begin
      set_pointers(4);
      double_rotation(tree,p1,p2,p3,p4,true);
      if x>p4^.data then reset_balances(p2,x,4)
      else reset_balances(p3,x,4);
    end
    else
    if (p2^.bal=1) and (x>p2^.data) and (x<p2^.right^.data)  then
    begin
      set_pointers(5);
      double_rotation(tree,p1,p2,p3,p4,false);
      if x>p4^.data   then  reset_balances(p3,x,5)
      else    reset_balances(p2,x,5);
    end;
  end { if not brethike }
  else
  begin
    gotoxy(30,16);
    writeln('Η ΤΙΜΗ ΥΠΑΡΧΕΙ ΗΔΗ ΣΤΟ ΔΕΝΤΡΟ');
  end;
end;


procedure print_tree(t:pointer;h:integer);
var
    i:integer;

begin
  if t<>nil then
  begin
    print_tree(t^.left,h+1);
    for i:=0 to h do   write('    ');
    writeln(t^.data);
    print_tree(t^.right,h+1);
  end;
end;


begin   { main }
  clrscr;
  gotoxy(20,13);
  write('ΔΩΣΕ ΕΝΑ ΑΚΕΡΑΙΟ 9999 ΓΙΑ ΤΕΛΟΣ:');  readln(x);
  while x<>9999 do
  begin
```

```
    avl_insert;
    clrscr;
    gotoxy(20,13);
    write('ΔΩΣΕ ΕΝΑ ΑΚΕΡΑΙΟ 9999  ΓΙΑ ΤΕΛΟΣ:'); readln(x);
  end;{ while }
  clrscr;
  print_tree(tree,0);
end.
```

3.



Ασκήσεις 6.1

4. Δηλώσεις

```
var a: array [1..v, 1..v] of boolean
{ο πινακας γειτνιασης του γραφήματος}

procedure dfs;
  var  k: integer;
     vis: array [1..v] of boolean;
  procedure visit (k: integer);
  var t: integer;
  begin
    vis[k]:= true;
    for t:= 1 to v do
    if a[k, t] and not vis[t] then visit(t);
  end {visit}

begin
  for k:= 1 to v do vis[k]:= false
  for k:= 1 to v do
  if not vis[k] then visit(k)
end {dfs}
```

Ασκήσεις 7.1

```
2. program min_radius_spanning_ tree;
     {το πρόγραμμα αυτό υπολογίζει το επικαλυπτικό δένδρο}
     {ενός γραφήματος με την ελάχιστη ακτίνα}

     const   v=...;  {κορυφές}
             e=...;  {ακμές}
     var
         w: array [1..v, 1..v] of  boolean;  {πινακας γειτνιασης}
         adj: array [1..v] of integer;
         {adj[i] δηλώνει το πλήθος γειτονικών κορυφών}
         {της κορυφής i}

         vis: array[1..v] of boolean; {κορυφές που έχουν
                                                 επισκεφθεί}
         act: array [1..v] of boolean; {κορυφές σε αναμονή να
                                                 επεξεργαστούν}

         parent: array[1..v] of integer;
         {parent[i] περιέχει τον πατέρα της κορυφής i}
         {στο επικαλυπτικό δένδρο}
         i, j, x, y, k, max, r: integer;

     begin
       {input γραφήματος}
       readln (v, e);
       for i:= 1 to v do
       for j:= 1 to v do  w[i,j]:= false;
       for j:= 1 to e do
        begin
          readln(x, y);
          w[x, y]:= true;
          w[y, x]:= true;
        end;
     { υπολογισμός των γειτονικών κορυφών για κάθε κορυφή }
     { και της κορυφής με τον μεγαλύτερο πλήθος γειτόνων }

     for i:= 1 to v do
     begin
       max:= 0;
       adj[i]:= 0;
       for j:= 1 to v do
       if w[i,j] then adj[i]:= adj[i] + 1;
       if adj[i] > max then
```

```
    begin
      max:= adj[i];
      k:= i
    end;
    {αρχικές συνθήκες}
    vis[i]:= false;
    act[i]:= false;
    parent[i]:= 0
  end;
  { Διάσχιση του γραφήματος με την μέθοδο bfs. Η κορυφή που }
  { επισκέπτεται πρώτη είναι εκείνη με το μεγαλύτερο βαθμό }
  { δηλαδή η κορυφή με το μεγαλίτερο πλήθος γειτόνων οι     }
  { οποίοι δεν έχουν ακόμη επισκεφθεί                       }

  vis[k]:= true;
  r:= 1; {ακτινα}
  repeat
    r:= r + 1;
    for i:= 1 to v do
    if w[k, i] and not vis[i] then
     begin
       vis[i]:= true;
       act[i]:= true;
       adj[i]:= adj[i] - 1;
       for j:= 1 to v do
       if w[i, j] and act[j] then
        begin
          adj[i]:= adj[i] - 1;
          adj[j]:= adj[j]-1
        end;
       parent[i]:= k;
     end;
     {υπολογισμός του κόμβου με τον μεγαλίτερο βαθμό}
     max:= 0;
     for i:= 1 to v do
     if act[i] and (adj[i] > max) then
      begin
        max:= adj[i];
        k:= i
      end;
     act[k]:= false
     until max = 0
  end {min_radius_spanning_tree}
```

```
4.  funcion connect (v1, v2: integer): boolean;
    var v: integer;
        visited: array [1..n] of boolean;
    procedure visit (v: integer);
    var t: listpointer;
    begin
      visited[v]:= true;
      t:= adj[v];
      while t <> nil do
      begin
        if not visited [t^.key] then
         visit (t^.key);
        t:= t^.next
      end
    end

    begin {connect}
      connect:= false;
      for v:= 1 to n do
       visited[v]:= false;
       visit(v1);
       if visited[v2] then connect:= true
     end {connect}

     όπου καθολικά έχουν δηλωθεί:
       const n= ...;
        type list pointer=^ node;
             node = record
                      key: integer;
                      next: listpointer
                    end;
       var adj: array[1..n] of listpointer
```

```
9. program  Dijkstra ;
     var w:array [1..12,1..12] of integer;
         val,parent:array [1..12] of integer;
         vis,act:array[1..12] of boolean;
         n,k,next,t,x,y,y1,min:integer;
         ch:char;    st:string[8];
     begin
     (*-------------------  input graph --------------------*)
       write ('read in number of nodes  ');
       readln(n);
```

```
for  x:= 1  to  n  do
for y:= 1  to   n  do    w[x,y]:=0;
repeat
  write( 'x = ?  ','y = ?  ','w[',x:2,',',y:2,'] = ?   ');
  readln(x,y,w[x,y]);
until (x=999) or (y=999);
(*----------------- initial values --------------------*)
for  k:= 1  to  n  do
begin
  val[k]:= 0;  parent[k]:= 0;
  vis[k]:= false;   act[k]:= false
end;
write('read in initial node  ');
readln(k);
vis[k]:= true;
(*------  shortest paths from a node to all other -----*)
repeat
  for  t:= 1  to  n  do
  if (not vis[t]) and (w[k,t] > 0) then
   if (val[t] = 0) or (val[t] > val[k] + w[k,t]) then
   begin
     val[t]:= val[k] + w[k,t];
     act[t]:= true;
     parent[t]:= k;
   end;
  min:= maxint;
  k:= 0;
  for  t:= 1  to  n  do
  if  (act[t]) and (val[t] < min)  then
  begin
    min:= val[t];
    k:= t;
  end;
  vis[k]:= true;
  act[k]:= false;
until k=0;
(*----------------- print results  -------------------*)
for  y:= n  downto  1  do
begin
  x:= parent[y];
  write('cost of path  ',val[y]:3);
  write ('  path :',y:3);
  while x<>0 do
  begin
    write(x:3);
    y1:= x;  x:= parent[y1];
```

```
    end;
    writeln
  end;
end.
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. A.V. Aho, J.E. Hopcroft, J.D. Ullman The Design and analysis of Computer Atgorithms, Addison Wesley 1974.
2. A.V. Aho, J.E. Hopcroft, J.D. Ullman, Data sructures and Algorithms, Addison Wesley 1983.
3. M. Azmoodeh, Abstract data types and Algorithms, Macmillan Education LTD, 1988.
4. J. G.P. Barnes, Programming in Ada, International computer science series, 1984.
5. P. Berlioux, P. Bizard, Algorithms, The Construction, Proof and analysis of Programs, J. Wiley, 1986.
6. G. Booch, Software components with Ada, Benjamin/Cummings Publishing Company, 1987.
7. N. Christofides, Graph theory: An algorithmic approach, Academic Press, 1975.
8. M.B. Feldman, Data structures with Ada, Reston Publishing Company, 1985.
9. W. Findlay, D. Watt, Pascal, An introduction to methodical programming, Pitman, 1981.
10. C.C Gotlied, L.R. Gotlieb. Data types and structures, Prentice-Hall 1978.
11. G.H. Gonnet, Handbook of algorithms and Data structures, International Computer Science series, 1984.
12. F. Harary, Graph theory, Addisson Wesley, 1969.
13. P. Hibbard et al. Studies in Ada style, 2nd Ed., Springer-Verlag, 1983.
14. E. Horowitz, S. Sahmi, Fundamentals of data structures in Pascal, Computer Science Press, 1984.
15. D.E. Knuth, The art of Computer programming, Fundamentals algorithms Vol. 1, Addison Wesley, 1973.
16. K. Mehlhorn, Data structures and Algorithms 1,2,3, Springer-Verlang, 1984.
17. E.S. Page, L.B. Wilson, Information Representation and manipulation in a Computer, Cambridge University Press 1973.
18. C.H Papadimitriou, K. Steigtitz, Combinatorial optimization: Algorithms and Complexity, Prentice Hall, Englewood Cliffs, NJ, 1982.
19. E.M. Reingold, W.J. Hansen, Data structures, Little Brown Boston Mass.
20. J.S. Rohl, Recursion via Pascal, Cambridge University Press, 1984.
21. R. Sedgewick, Algorithms, Addison Wesley 1983.
22. T.A. Standish, Data structure techniques, Addison-Wesley 1980.
23. D.F. Stubbs, N.W. Webre, Data structures with Abstract data

types and Pascal, Brooks/Colè Publishing Company, 1985.
24. R.E. Tarjan, Data structures and Network Algorithms, SIAM, CBMS-NSF 44, Ed.5, 1988.
25. A.M. Tenenbaum, M.J. Augenstein, Data structures using Pascal, Prentice Hall 1981.
26. J.P. Tremblay, P.G. Sorenson, An introduction to data structures with applications, Mc Craw_Hill, 1984.
27. N. Wirth, Algorithms + Data srectures = programs, Prentice-Hall 1976.

ΕΛΛΗΝΙΚΗ ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Ι.Κ. Κάβουρα, Δομημένος προγραμματισμός με Pascal, Τόμος Ι, Αθήνα, 1989.
2. Ι.Κ. Κάβουρα, Δομημένος προγραμματισμός με Pascal, Τόμος ΙΙ, Αθήνα, 1988.
3. Ι.Κ. Κάβουρα, Συστήματα Υπολογιστών, Τόμος Ι, Αθήνα 1989.
4. Ι.Κ. Κάβουρα, Συστήματα Υπολογιστών, Τόμος ΙΙ, Αθήνα 1987.
5. Ε.Α. Κιουντούζη, Διοικητικός προγραμματισμός έργων πληροφορικής, 1988.
6. Γ. Κόλλια, Δομές Δεδομένων, 1984.
7. Μ. Λουκάκη, Δομές Δεδομένων-Αλγόριθμοι, Τόμος Α, Θεσσαλονίκη 1987.

ΕΥΡΕΤΗΡΙΟ ΟΡΩΝ