ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ
ΚΕΝΤΡΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ
ΣΧΟΛΗ
ΤΜΗΜΑ

Αλγόριθμοι
&
Δομές Δεδομένων

Ευάγγελος Γ. Ούτσιος (BSc, MSc)
Καθηγητής Εφαρμογών

Τομέας Υπολογιστικών τεχνικών και Συστημάτων

Σεπτέμβριος 2015

Άδειες Χρήσης

Χρηματοδότηση

Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.

Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο ΤΕΙ Κεντρικής Μακεδονίας**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.

Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

ΠΕΡΙΕΧΟΜΕΝΑ

I. ΑΛΓΟΡΙΘΜΟΙ

1.1 Εισαγωγή…… 1-1

1.1.1 Ανάλυση – Σύνθεση Προβλήματος…… 1-1

1.2 Βασικές έννοιες αλγορίθμων……. 1-3

1.2.1 Περιγραφή και αναπαράσταση αλγορίθμων… 1-4

1.2.2 Στοιχεία ψευδοκώδικα……. 1-5

1.3 Βασικές αλγοριθμικές δομές…….. 1-6

1.3.1 Δομή Ακολουθίας…….. 1-6

1.3.2 Δομή Επιλογής…….. 1-7

1.3.3 Δομή Επανάληψης…… 1-9

1.4 Ανάλυση αλγορίθμων…… 1-12

1.4.1 Επίδοση αλγορίθμων…… 1-12

1.4.2 Ορθότητα αλγορίθμων……. 1-12

1.4.3 Πολυπλοκότητα αλγορίθμων……. 1-13

1.4.4 Είδη αλγορίθμων…….. 1-15

ΙΙ. ΠΙΝΑΚΕΣ

2.1 Εισαγωγή……. 2-1

2.2 Βασικές έννοιες πινάκων…….. 2-2

2.3 Αποθήκευση πινάκων……. 2-6

2.4 Ειδικές μορφές πινάκων…….. 2-7

ΙΙΙ. ΑΝΑΔΡΟΜΗ

3.1 Εισαγωγή…… 3-1

3.2 Υπολογισμός παραγοντικού……... 3-1

3.3 Υπολογισμός δύναμης…… 3-3

IV. ΑΝΑΖΗΤΗΣΗ

4.1 Εισαγωγή..... 4-1

4.2 Σειριακή Αναζήτηση..... 4-1

4.3 Δυαδική Αναζήτηση..... 4-3

V. ΤΑΞΙΝΟΜΗΣΗ

5.1 Εισαγωγή……... 5-1

5.2 Ταξινόμηση με απευθείας επιλογή…….... 5-2

5.3 Ταξινόμηση με απευθείας εισαγωγή…... 5-3

5.4 Ταξινόμηση φυσαλίδας……. 5-4

5.5 Γρήγορη ταξινόμηση……..… 5-5

VI. ΓΡΑΜΜΙΚΕΣ ΛΙΣΤΕΣ

6.1 Εισαγωγή..... 6-1

6.2 Σειριακές Λίστες..... 6-1

6.2.1 Στοίβα..... 6-1

6.2.2 Ουρά..... 6-3

6.3 Συνδεδεμένες Λίστες..... 6-6

6.3.1 Απλή συνδεδεμένη λίστα..... 6-7

6.3.2 Στοίβα ως συνδεδεμένη λίστα..... 6-11

6.3.3 Ουρά ως συνδεδεμένη λίστα..... 6-13

VII. ΔΕΝΔΡΑ

7.1 Εισαγωγή..... 7-1

7.2 Δυαδικά δένδρα..... 7-3

7.3 B-trees..... 7-10

7.4 Tries..... 7-13

VIII. ΓΡΑΦΟΙ

8.1 Εισαγωγή..... 8-1

8.2 Μέθοδοι Αναπαράστασης γράφων.. 8-4

8.3 Μέθοδοι Διάσχισης γράφων..... 8-6
8.3.1 Αναζήτηση με προτεραιότητα Βάθους 8-6
8.3.2 Αναζήτηση με προτεραιότητα Πλάτους 8-8

8.4 Το πρόβλημα του συντομότερου μονοπατιού..... 8-10

ΙΧ. ΠΙΝΑΚΕΣ ΚΑΤΑΚΕΡΜΑΤΙΣΜΟΥ

9.1 Εισαγωγή..... 9-1

9.2 Συγκρούσεις..... 9-3

9.3 Ανοιχτή διευθυνσιοδότηση..... 9-3

9.4 Ξεχωριστή σύνδεση..... 9-4

ΠΙΝΑΚΑΣ ΣΧΗΜΑΤΩΝ

Σχήμα 1.1 *Στάδια αντιμετώπισης ενός προβλήματος* …….…….. 1-3
Σχήμα 1.2 *Γεωμετρικά σχήματα διαγράμματος ροής*……... 1-5
Σχήμα 6.1 *LIFO (Last In First Out)*…… 6-2
Σχήμα 6.2 *Ουρά (queue)*…….. 6-4
Σχήμα 6.3 *Εικονική υπερχείλιση*…….. 6-5
Σχήμα 6.4 *Λύση Εικονικής υπερχείλισης*…….. 6-5
Σχήμα 6.5 *Κυκλική ουρά*…….. 6-6
Σχήμα 6.6 *Συνδεδεμένες Λίστες*…… 6-6
Σχήμα 6.7 *Στοίβα ως Συνδεδεμένη λίστα*……. 6-12
Σχήμα 6.8 *Ουρά ως Συνδεδεμένη Λίστα*……. 6-13
Σχήμα 7.1 *Δένδρο*…… 7-1
Σχήμα 7.2 *Δυαδικό Δένδρο Αναζήτησης*…… 7-2
Σχήμα 7.3 *Η Αλγεβρική έκφραση A*B+C*……. 7-3
Σχήμα 7.4 *Διάσχιση δυαδικού δένδρου*…… 7-4
Σχήμα 7.5 *Ισορροπημένα Δένδρα*…… 7-9
Σχήμα 7.6 *Trie*…… 7-15
Σχήμα 8.1 *Γράφοι*…… 8-2
Σχήμα 8.2 *Αναζήτηση με προτεραιότητα βάθους*……. 8-6

ΠΙΝΑΚΑΣ ΠΙΝΑΚΩΝ

Πίνακας 1.1 *Πίνακας λογικών πράξεων* 1-9

Πίνακας 2.1 *Άθροισμα στοιχείων πίνακα*..... 2-6

Πίνακας 4.1 *Αριθμός συγκρίσεων με Δυαδική Αναζήτηση*..... 4-5

ΠΑΡΑΡΤΗΜΑ I

Πίνακας χαρακτηριστικών Δομών Δεδομένων…. I-1

ΒΙΒΛΙΟΓΡΑΦΙΑ

Ι. ΑΛΓΟΡΙΘΜΟΙ..... 1-1

1.1 Εισαγωγή..... 1-1

1.1.1 Ανάλυση – Σύνθεση Προβλήματος..... 1-1

1.2 Βασικές έννοιες αλγορίθμων..... 1-3

1.2.1 Περιγραφή και αναπαράσταση αλγορίθμων... 1-4

1.2.2 Στοιχεία ψευδοκώδικα..... 1-5

1.3 Βασικές αλγοριθμικές δομές..... 1-6

1.3.1 Δομή Ακολουθίας..... 1-6

1.3.2 Δομή Επιλογής..... 1-7

1.3.3 Δομή Επανάληψης..... 1-9

1.4 Ανάλυση αλγορίθμων..... 1-12

1.4.1 Επίδοση αλγορίθμων..... 1-12

1.4.2 Ορθότητα αλγορίθμων..... 1-12

1.4.3 Πολυπλοκότητα αλγορίθμων..... 1-13

1.4.4 Είδη αλγορίθμων..... 1-15

1.1 Ε Ι Σ Α Γ Ω Γ Η

1.1.1 ΑΝΑΛΥΣΗ – ΣΥΝΘΕΣΗ ΠΡΟΒΛΗΜΑΤΟΣ

Το πρόβλημα αποτελεί έννοια που απαντάται σε όλες τις επιστήμες και τους κλάδους τους, αλλά παράλληλα και στην καθημερινή μας ζωή.

Με τον όρο **πρόβλημα** εννοείται μια κατάσταση η οποία χρήζει αντιμετώπισης, απαιτεί λύση, η δε λύση της δεν είναι γνωστή, ούτε προφανής.

Τόσο η αντιμετώπιση, όσο και η διατύπωση ενός προβλήματος, αποτελούν διαδικασίες που απαιτούν ιδιαίτερες αναλυτικές και συνθετικές ικανότητες, ορθολογική σκέψη, αλλά και σωστό και εμπεριστατωμένο χειρισμό της φυσικής γλώσσας.

Η οποιαδήποτε προσπάθεια αντιμετώπισης ενός προβλήματος είναι καταδικασμένη σε αποτυχία αν προηγουμένως δεν έχει γίνει απόλυτα κατανοητό το πρόβλημα που τίθεται. Η κατανόηση ενός προβλήματος αποτελεί συνάρτηση δύο παραγόντων, της σωστής διατύπωσης εκ μέρους του δημιουργού του και της αντίστοιχα σωστής ερμηνείας από τη μεριά εκείνου που καλείται να το αντιμετωπίσει.

Η κατανόηση του προβλήματος είναι βασική προϋπόθεση για να γίνει στη συνέχεια δυνατή η σωστή αποτύπωση της δομής του. Η καταγραφή της δομής ενός προβλήματος σημαίνει αυτόματα ότι έχει αρχίσει η διαδικασία ανάλυσης του προβλήματος σε άλλα απλούστερα. Με τη σειρά τους τα νέα προβλήματα μπορούν να αναλυθούν σε άλλα, ακόμη πιο απλά. Η διαδικασία αυτή της ανάλυσης μπορεί να συνεχιστεί μέχρις ότου τα επιμέρους προβλήματα που προέκυψαν θεωρηθούν αρκετά απλά και η αντιμετώπισή τους χαρακτηριστεί ως δυνατή.

Με το όρο δομή ενός προβλήματος αναφερόμαστε στα συστατικά του μέρη, στα επιμέρους τμήματα που το αποτελούν καθώς επίσης και στον τρόπο που αυτά τα μέρη συνδέονται μεταξύ τους.

Η δυσκολία αντιμετώπισης των προβλημάτων ελαττώνεται όσο περισσότερο προχωράει η ανάλυσή τους σε απλούστερα προβλήματα. Ο κατακερματισμός

ενός προβλήματος σε άλλα απλούστερα είναι μια από τις διαδικασίες που ενεργοποιούν και αμβλύνουν τόσο τη σκέψη, αλλά κυρίως την αναλυτική ικανότητα του ατόμου.

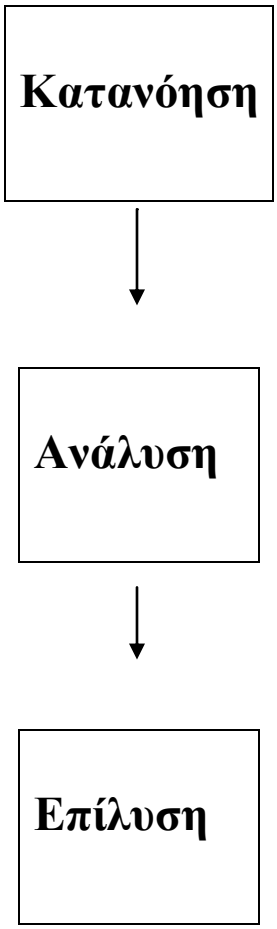
Η σωστή επίλυση ενός προβλήματος προϋποθέτει τον επακριβή προσδιορισμό των **δεδομένων** που παρέχει το πρόβλημα. Απαιτεί επίσης την λεπτομερειακή καταγραφή των **ζητούμενων** που αναμένονται σαν αποτελέσματα της επίλυσης του προβλήματος.

Θα πρέπει να δοθεί μεγάλη προσοχή στην ανίχνευση των δεδομένων ενός προβλήματος. Επισημαίνεται πως δεν είναι πάντοτε εύκολο να διακρίνει κάποιος τα δεδομένα. Υπάρχουν πολλές περιπτώσεις προβλημάτων που τα δεδομένα θα πρέπει να 'ανακαλυφθούν' μέσα στα λεγόμενα του προβλήματος. Η διαδικασία αυτή απαιτεί προσοχή, συγκέντρωση και σκέψη. Μεθοδολογία προσδιορισμού των δεδομένων ενός προβλήματος δεν υπάρχει, ούτε και μεθοδολογία εντοπισμού και αποσαφήνισης των ζητούμενων ενός προβλήματος.

Το ίδιο προσεκτικά θα πρέπει να αποσαφηνιστούν και τα ζητούμενα ενός προβλήματος. Δεν είναι πάντοτε ιδιαίτερα κατανοητό τι ακριβώς ζητάει ένα πρόβλημα. Σε μια τέτοια περίπτωση θα πρέπει να θέτονται μια σειρά από ερωτήσεις με στόχο την διευκρίνιση πιθανών αποριών σχετικά με τα ζητούμενα, τον τρόπο παρουσίασής τους, το εύρος τους κλπ. Οι ερωτήσεις αυτές μπορούν να απευθύνονται είτε στο δημιουργό του προβλήματος, είτε στον ίδιο μας τον εαυτό αν εμείς καλούμαστε να αντιμετωπίσουμε το πρόβλημα.

Συμπερασματικά από όλα τα παραπάνω διαφαίνεται πως τα στάδια αντιμετώπισης ενός προβλήματος είναι τρία:

- **Κατανόηση**, όπου απαιτείται η σωστή και πλήρης αποσαφήνιση των δεδομένων και των ζητούμενων του προβλήματος
- **Ανάλυση**, όπου το αρχικό πρόβλημα διασπάται σε άλλα επιμέρους απλούστερα προβλήματα
- **Επίλυση**, όπου υλοποιείται η λύση του προβλήματος, μέσω της λύσης των επιμέρους προβλημάτων.



Σχήμα 1.1 *Στάδια αντιμετώπισης ενός προβλήματος*

1.2 ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ ΑΛΓΟΡΙΘΜΩΝ

Αλγόριθμος είναι μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος.

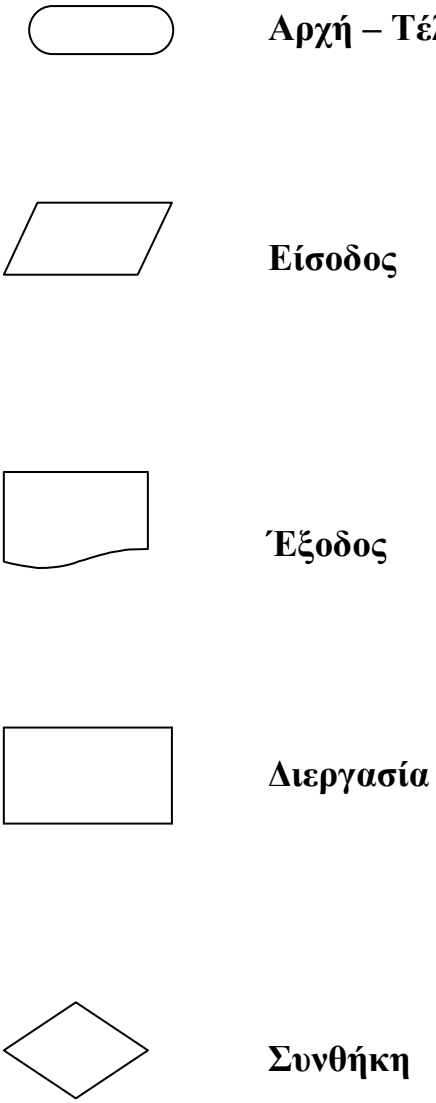
Κάθε αλγόριθμος απαραίτητα ικανοποιεί τα επόμενα κριτήρια:

- **Είσοδος** (input). Καμία, μία ή περισσότερες τιμές δεδομένων πρέπει να δίνονται ως είσοδοι στον αλγόριθμο. Η περίπτωση που δεν δίνονται τιμές δεδομένων εμφανίζεται, όταν ο αλγόριθμος δημιουργεί και επεξεργάζεται κάποιες πρωτογενείς τιμές με τη βοήθεια συναρτήσεων παραγωγής τυχαίων αριθμών ή με τη βοήθεια άλλων απλών εντολών.

- **Έξοδος** (output). Ο αλγόριθμος πρέπει να δημιουργεί τουλάχιστον μία τιμή δεδομένων ως αποτέλεσμα προς το χρήστη ή προς έναν άλλο αλγόριθμο.
- **Καθοριστικότητα** (definiteness). Κάθε εντολή πρέπει να καθορίζεται χωρίς καμία αμφιβολία για τον τρόπο εκτέλεσής της. Π.χ. η εντολή διαίρεσης πρέπει να θεωρεί και την περίπτωση, όπου ο διαιρέτης λαμβάνει μηδενική τιμή.
- **Περατότητα** (finiteness). Ο αλγόριθμος να τελειώνει μετά από πεπερασμένα βήματα εκτέλεσης των εντολών του.
- **Αποτελεσματικότητα** (effectiveness). Κάθε μεμονωμένη εντολή του αλγορίθμου να είναι απλή. Αυτό σημαίνει ότι μία εντολή δεν αρκεί να έχει ορισθεί, αλλά πρέπει να είναι και εκτελέσιμη.

1.2.1 Περιγραφή και αναπαράσταση αλγορίθμων

- **Ελεύθερο κείμενο**. Αποτελεί τον πιο ανεπεξέργαστο και αδόμητο τρόπο παρουσίασης αλγορίθμου. Έτσι εγκυμονεί τον κίνδυνο ότι μπορεί εύκολα να οδηγήσει σε μη εκτελέσιμη παρουσίαση παραβιάζοντας το τελευταίο χαρακτηριστικό των αλγορίθμων, δηλ. την αποτελεσματικότητα.
- **Φυσική γλώσσα** κατά βήματα. Στην περίπτωση αυτή χρειάζεται προσοχή, γιατί μπορεί να παραβιασθεί το τρίτο βασικό χαρακτηριστικό ενός αλγορίθμου, όπως προσδιορίσθηκε προηγουμένως, δηλ. το κριτήριο του καθορισμού.
- **Διάγραμμα ροής**. Αποτελείται από ένα σύνολο γεωμετρικών σχημάτων, όπου το καθένα δηλώνει μία συγκεκριμένη ενέργεια ή λειτουργία. Τα γεωμετρικά σχήματα ενώνονται μεταξύ τους με βέλη, που δηλώνουν τη σειρά εκτέλεσης των ενεργειών αυτών. Τα κυριότερα χρησιμοποιούμενα γεωμετρικά σχήματα είναι τα εξής:



Σχήμα 1.2 *Γεωμετρικά σχήματα διαγράμματος ροής*

- **Ψευδοκώδικας**. Χρησιμοποιείται μία δομημένη μορφή ψευδογλώσσας, που στηρίζεται στις βασικές αλγοριθμικές δομές και τις αρχές του δομημένου προγραμματισμού και μπορεί εύκολα σχετικά να προγραμματισθεί σε οποιαδήποτε γλώσσα προγραμματισμού.

1.2.2 Στοιχεία ψευδοκώδικα

- **Σταθερές** (constants). Με τον όρο αυτό αναφερόμαστε σε προκαθορισμένες τιμές που παραμένουν αμετάβλητες σε όλη τη διάρκεια της εκτέλεσης ενός αλγορίθμου.

- **Μεταβλητές** (variables). Μια μεταβλητή είναι ένα γλωσσικό αντικείμενο, που χρησιμοποιείται για να παραστήσει ένα στοιχείο δεδομένου. Στη μεταβλητή εκχωρείται μια τιμή, η οποία μπορεί να αλλάζει κατά τη διάρκεια εκτέλεσης ενός αλγορίθμου. Ανάλογα με το είδος της τιμής που μπορούν να λάβουν, οι μεταβλητές διακρίνονται σε
    - Αριθμητικές
    - αλφαριθμητικές
    - και λογικές.

- **Τελεστές** (operators). Πρόκειται για τα γνωστά σύμβολα που χρησιμοποιούνται σε διάφορες πράξεις. Οι τελεστές διακρίνονται σε
    - **Αριθμητικούς: +, -, *, /, %**
    - **Συγκριτικούς: >, <, =, >=, <=, !=**
    - **Λογικούς:** και (&&), ή (||), όχι (!)

- **Εκφράσεις** (expressions). Οι εκφράσεις διαμορφώνονται από τους τελεστέους (operands), που είναι σταθερές και μεταβλητές και από τους τελεστές. Η διεργασία αποτίμησης μιας έκφρασης συνίσταται στην απόδοση τιμών στις μεταβλητές και στην εκτέλεση των πράξεων. Η τελική τιμή μιας έκφρασης εξαρτάται από την ιεραρχία των πράξεων και τη χρήση των παρενθέσεων. Μια έκφραση μπορεί να αποτελείται από μια μόνο μεταβλητή ή σταθερά μέχρι μια πολύπλοκη μαθηματική παράσταση.

1.3 ΒΑΣΙΚΕΣ ΑΛΓΟΡΙΘΜΙΚΕΣ ΔΟΜΕΣ

1.3.1 Δομή Ακολουθίας

Η ακολουθιακή δομή εντολών (σειριακών βημάτων) χρησιμοποιείται πρακτικά για την αντιμετώπιση απλών προβλημάτων, όπου είναι δεδομένη η σειρά εκτέλεσης ενός συνόλου ενεργειών. Εντολές της δομής της ακολουθίας

είναι συνήθως οι εντολές εισόδου, εντολές εξόδου και εντολές υπολογισμού και εκχώρησης τιμής.

Π.χ.

Να διαβασθούν δυο αριθμοί, να υπολογισθεί και να εκτυπωθεί το άθροισμά τους.

ΑΛΓΟΡΙΘΜΟΣ Άθροισμα	**C:**
ΑΡΧΗ	**{**
ΔΙΑΒΑΣΕ a	**scanf(“%d”,&a);**
ΔΙΑΒΑΣΕ b	**scanf(“%d”,&b);**
c = a + b	**c = a + b;**
ΕΜΦΑΝΙΣΕ c	**printf(“%d”,c);**
ΤΕΛΟΣ	**}**

Μετά την ανάγνωση των τιμών των μεταβλητών **a** και **b** γίνεται ο υπολογισμός του αθροίσματος με την εντολή:

c = a + b

Η εντολή αυτή αποκαλείται εντολή **εκχώρησης τιμής** (assignment statement). Η γενική μορφή της είναι:

Μεταβλητή = Έκφραση

και η λειτουργία της είναι «γίνονται οι πράξεις στην έκφραση και το αποτέλεσμα αποδίδεται, μεταβιβάζεται, εκχωρείται στη μεταβλητή».

1.3.2 Δομή επιλογής

Στην πραγματικότητα λίγα προβλήματα μπορούν να επιλυθούν με τον προηγούμενο τρόπο της σειριακής/ακολουθιακής δομής ενεργειών. Συνήθως τα προβλήματα έχουν κάποιες ιδιαιτερότητες και δεν ισχύουν τα ίδια βήματα για κάθε περίπτωση. Η πλέον συνηθισμένη περίπτωση είναι να λαμβάνονται κάποιες αποφάσεις με βάση κάποια δεδομένα κριτήρια που μπορεί να είναι διαφορετικά για κάθε διαφορετικό στιγμιότυπο ενός προβλήματος.

Γενικά η διαδικασία της επιλογής περιλαμβάνει τον έλεγχο κάποιας συνθήκης που μπορεί να έχει δύο τιμές (Αληθής ή Ψευδής) και ακολουθεί η απόφαση εκτέλεσης κάποιας ενέργειας με βάση την τιμή της λογικής αυτής συνθήκης.

Στην παράσταση αλγορίθμων με ψευδοκώδικα η επιλογή υλοποιείται με την εντολή ΑΝ…ΤΟΤΕ. Η σύνταξη της εντολής είναι:

Απλή επιλογή

ΑΝ συνθήκη ΤΟΤΕ	**if (expression)**
Εντολές	**Statements;**
ΤΕΛΟΣ ΑΝ	

Σύνθετη επιλογή

ΑΝ συνθήκη ΤΟΤΕ	**if (expression)**
Εντολές 1	**Statements 1;**
ΑΛΛΙΩΣ	**else**
Εντολές 2	**Statements 2;**
ΤΕΛΟΣ ΑΝ	

Εμφωλευμένη επιλογή

ΑΝ συνθήκη 1 ΤΟΤΕ	**if (expression 1)**
ΑΝ συνθήκη 2 ΤΟΤΕ	**if (expression 2)**
Εντολές 1	**Statements 1;**
ΑΛΛΙΩΣ	**else**
Εντολές 2	**Statements 2;**
ΤΕΛΟΣ ΑΝ	**else**

```
ΑΛΛΙΩΣ                          if (expression 3)
    ΑΝ συνθήκη 3 ΤΟΤΕ                Statements 3;
        Εντολές 3                else
    ΑΛΛΙΩΣ                           Statements 4;
        Εντολές 4
    ΤΕΛΟΣ ΑΝ
ΤΕΛΟΣ ΑΝ
```

Σε πολλές περιπτώσεις η συνθήκη εμπεριέχει αποφάσεις που πιθανόν βασίζονται σε περισσότερα από ένα κριτήρια. Ο συνδυασμός των κριτηρίων αυτών καθορίζει και τις 'λογικές' πράξεις που μπορούν να γίνουν μεταξύ διαφορετικών συνθηκών. Πολύ συχνά στην καθημερινή ζωή κάποιες αποφάσεις βασίζονται σε συνδυασμούς κριτηρίων και λογικών πράξεων. Η λογική πράξη **ή** είναι αληθής όταν οποιαδήποτε από δύο προτάσεις είναι αληθής. Η λογική πράξη **και** είναι αληθής όταν και οι δύο προτάσεις είναι αληθείς, ενώ η λογική πράξη **όχι** είναι αληθής όταν η πρόταση που την ακολουθεί είναι ψευδής. Ο επόμενος πίνακας δίνει τις τιμές των τριών λογικών πράξεων για όλους τους συνδυασμούς τιμών:

Πρόταση Α	Πρόταση Β	Α ή Β	Α και Β	όχι Α
ΑΛΗΘΗΣ	ΑΛΗΘΗΣ	ΑΛΗΘΗΣ	ΑΛΗΘΗΣ	ΨΕΥΔΗΣ
ΑΛΗΘΗΣ	ΨΕΥΔΗΣ	ΑΛΗΘΗΣ	ΨΕΥΔΗΣ	ΨΕΥΔΗΣ
ΨΕΥΔΗΣ	ΑΛΗΘΗΣ	ΑΛΗΘΗΣ	ΨΕΥΔΗΣ	ΑΛΗΘΗΣ
ΨΕΥΔΗΣ	ΨΕΥΔΗΣ	ΨΕΥΔΗΣ	ΨΕΥΔΗΣ	ΑΛΗΘΗΣ

Πίνακας 1.1 *Πίνακας λογικών πράξεων*

1.3.3 Δομή Επανάληψης

Η διαδικασία της επανάληψης είναι ιδιαίτερα συχνή, αφού πλήθος προβλημάτων μπορούν να επιλυθούν με κατάλληλες επαναληπτικές διαδικασίες. Η λογική των επαναληπτικών διαδικασιών εφαρμόζεται στις

περιπτώσεις όπου μια ακολουθία εντολών πρέπει να εφαρμοσθεί σε ένα σύνολο περιπτώσεων, που έχουν κάτι κοινό. Οι επαναληπτικές διαδικασίες μπορεί να έχουν διάφορες μορφές και συνήθως εμπεριέχουν και συνθήκες επιλογών. Γενικά υπάρχουν τρία σχήματα δομών επανάληψης, που υποστηρίζονται από τις περισσότερες γλώσσες προγραμματισμού.

- Επαναληπτικό σχήμα με έλεγχο επανάληψης στην αρχή

ΟΣΟ συνθήκη ΕΠΑΝΕΛΑΒΕ **while (expression)**
Εντολές **Statements;**
ΤΕΛΟΣ ΕΠΑΝΑΛΗΨΗΣ

- Επαναληπτικό σχήμα με έλεγχο επανάληψης στο τέλος

ΑΡΧΗ ΕΠΑΝΑΛΗΨΗΣ **do**
Εντολές **Statements;**
ΜΕΧΡΙΣ ΟΤΟΥ συνθήκη **while (expression);**

- Επαναληπτικό σχήμα ορισμένων φορών επανάληψης

ΓΙΑ μεταβλητή ΑΠΟ αρχική τιμή ΜΕΧΡΙ τελική τιμή
Εντολές
ΤΕΛΟΣ ΕΠΑΝΑΛΗΨΗΣ

for (variable = initial value; variable<= final value; step)
Statements;

Το πρώτο σχήμα επαναληπτικής δομής (while) είναι το πιο γενικό και μπορεί να εκτελεσθεί από καθόλου μέχρι όσες φορές θέλουμε.

Το δεύτερο σχήμα της επαναληπτικής δομής (do-while) έχει το χαρακτηριστικό να εκτελείται οπωσδήποτε μία φορά, επειδή ο έλεγχος της συνθήκης τερματισμού γίνεται στο τέλος, γι' αυτό και δεν χρησιμοποιείται πολύ.

Το τρίτο σχήμα επαναληπτικής δομής (for) χρησιμοποιείται όταν το πλήθος των επαναλήψεων είναι εκ των προτέρων γνωστό.

1.4 ΑΝΑΛΥΣΗ ΑΛΓΟΡΙΘΜΩΝ

Η καταγραφή των μεγεθών που επηρεάζουν την **επίδοση** ενός αλγορίθμου είναι μια σημαντική ενέργεια για την κατανόηση της **αποδοτικότητας** ενός αλγορίθμου. Για το λόγο αυτό, είναι απαραίτητο να βρεθεί η βασική λειτουργία και η δομή του αλγορίθμου όπου κυρίως δαπανώνται οι υπολογιστικοί πόροι.

1.4.1 Επίδοση αλγορίθμων

Η χειρότερη περίπτωση ενός αλγορίθμου αφορά στο μέγιστο κόστος εκτέλεσης του αλγορίθμου. Το κόστος αυτό πολλές φορές κρίνει την επιλογή και τον σχεδιασμό ενός αλγορίθμου. Για να εκφρασθεί αυτή η χειρότερη περίπτωση, χρειάζεται κάποιο μέγεθος σύγκρισης και αναφοράς που να χαρακτηρίζει τον αλγόριθμο. Μια συνηθισμένη πρακτική είναι το πλήθος των βασικών πράξεων που εκτελεί ο αλγόριθμος.

Οι βρόγχοι επανάληψης αποτελούν το κρίσιμο σημείο για τον χαρακτηρισμό της επίδοσης ενός αλγορίθμου. Αν η λύση ενός προβλήματος επιτυγχάνεται με τη χρήση δύο ή περισσοτέρων αλγορίθμων, χρειάζεται να γίνει η επιλογή του καταλληλότερου με βάση την αποδοτικότητά τους. Βέβαια, όταν συγκρίνονται δύο αλγόριθμοι, θα πρέπει να συγκρίνονται με χρήση των ίδιων δεδομένων και κάτω από τις ίδιες συνθήκες. Γενικά, ο χρόνος εκτέλεσης κάθε αλγορίθμου εξαρτάται από ένα σύνολο παραγόντων, όπως:

- Τύπος Η/Υ που θα εκτελέσει το πρόγραμμα του αλγορίθμου
- Γλώσσα προγραμματισμού που θα χρησιμοποιηθεί
- Δομή προγράμματος και δομές δεδομένων που χρησιμοποιούνται
- Είδος συστήματος, ενός χρήστη ή πολλών χρηστών

1.4.2 Ορθότητα αλγορίθμων

Η επίλυση ενός προβλήματος με τη χρήση κάποιου αλγορίθμου, έχει μεγαλύτερη ισχύ, όταν υπάρχει κάποια τυποποιημένη ένδειξη ή απόδειξη για την ορθότητα του προτεινόμενου αλγορίθμου. Οι τεχνικές απόδειξης της ορθότητας ενός αλγορίθμου συνδέονται άμεσα με τον τρόπο της αρχικής

σχεδίασης και ανάλυσης του συγκεκριμένου αλγορίθμου. Είναι λάθος να επιχειρείται ο έλεγχος των σφαλμάτων και της ορθότητας ενός αλγορίθμου μετά την σχεδίαση και την τυποποίησή του. Γενικά, δεν μπορεί να υπάρξει απόλυτη ασφάλεια στην εκτίμηση των πιθανών σφαλμάτων ενός αλγορίθμου. Η απόδειξη της ορθότητας ενός αλγορίθμου θα πρέπει να περιλαμβάνει τις εξής δύο συνθήκες:

- Απόδειξη ότι σε κάθε περίπτωση ο τερματισμός της εκτέλεσης του αλγορίθμου οδηγεί σε αποδεκτά αποτελέσματα
- Απόδειξη ότι θα υπάρξει τερματισμός της εκτέλεσης του αλγορίθμου

1.4.3 Πολυπλοκότητα αλγορίθμων

Ο απλούστερος τρόπος μέτρησης της επίδοσης ενός αλγορίθμου είναι ο **εμπειρικός**. Δηλαδή, ο αλγόριθμος υλοποιείται και εφαρμόζεται σε ένα σύνολο δεδομένων, για να υπολογισθεί ο απαιτούμενος χρόνος επεξεργασίας (processing time) και η χωρητικότητα μνήμης (memory space). Ο τρόπος όμως αυτός παρουσιάζει τα εξής μειονεκτήματα:

- Είναι δύσκολο να προβλεφθεί η συμπεριφορά του αλγορίθμου για κάποιο άλλο σύνολο δεδομένων
- Ο χρόνος επεξεργασίας εξαρτάται από το υλικό, τη γλώσσα προγραμματισμού και τον μεταφραστή και φυσικά την ικανότητα του προγραμματιστή

Έτσι, μπορεί να βγουν λανθασμένες εκτιμήσεις για την επίδοση του αλγορίθμου. Ένας άλλος τρόπος εκτίμησης της επίδοσης ενός αλγορίθμου είναι ο **θεωρητικός**. Εισάγεται μία μεταβλητή n, που εκφράζει το μέγεθος του προβλήματος, έτσι ώστε η μέτρηση της αποδοτικότητας του αλγορίθμου να ισχύει για οποιοδήποτε σύνολο δεδομένων και ανεξάρτητα από υποκειμενικούς παράγοντες. Η σημασία της μεταβλητής αυτής εξαρτάται από το πρόβλημα που πρόκειται να επιλυθεί.

Στη συνέχεια, ο χρόνος επεξεργασίας και ο απαιτούμενος χώρος μνήμης εκτιμώνται με τη βοήθεια μιας συνάρτησης f(n), που εκφράζει τη **χρονική**

πολυπλοκότητα (time complexity) και της g(n), που εκφράζει την **πολυπλοκότητα χώρου** (space complexity).

Σε πολλές περιπτώσεις, όμως, δεν ενδιαφέρουν οι επακριβείς τιμές, αλλά μόνο η γενική συμπεριφορά των αλγορίθμων, δηλαδή η τάξη του αλγορίθμου. Για το λόγο αυτό, εισάγεται ο λεγόμενος **συμβολισμός O** (O – notation), από την αγγλική λέξη order.

Σχεδόν οι περισσότεροι αλγόριθμοι πρακτικού ενδιαφέροντος έχουν χρονική πολυπλοκότητα που ανήκει σε μία από τις επόμενες κατηγορίες:

- **O**(1)

  Κάθε εντολή του προγράμματος εκτελείται μία ή μερικές μόνο φορές
- **O**(log n)

  Ο αλγόριθμος είναι λογαριθμικής πολυπλοκότητας
- **O**(n)

  Γραμμική πολυπλοκότητα.

  Είναι η καλύτερη επίδοση για έναν αλγόριθμο που πρέπει να εξετάσει ή να δώσει n στοιχεία στην έξοδο.
- **O**(n.log n)

  Στην κατηγορία αυτή ανήκει μια πολύ σπουδαία οικογένεια αλγορίθμων ταξινόμησης.
- **O**(n^2)

  Τετραγωνική πολυπλοκότητα.

  Πρέπει να χρησιμοποιείται μόνον στα προβλήματα μικρού μεγέθους.
- **O**(n^3)

  Κυβική πολυπλοκότητα.

  Πρέπει να χρησιμοποιείται μόνον στα προβλήματα μικρού μεγέθους.
- **O**(2^n)

  Εκθετική πολυπλοκότητα.

  Σπάνια χρησιμοποιείται στην πράξη.

1.4.4 Είδη αλγορίθμων

Το αντικείμενο των αλγορίθμων είναι η βάση όπου στηρίζονται όλοι σχεδόν οι τομείς της Πληροφορικής. Για το λόγο αυτό, έχουν αναπτυχθεί πολλά είδη και κατηγορίες αλγορίθμων, όπως για παράδειγμα οι αναδρομικοί και οι επαναληπτικοί αλγόριθμοι. Στη συνέχεια θα αναφερθούμε σε μερικές νέες έννοιες σχετικά με τις κατηγορίες αλγορίθμων.

Υπάρχουν, σήμερα, Η/Υ που αποτελούνται από πολλούς επεξεργαστές, πολλές κύριες μνήμες και πολλές δευτερεύουσες μνήμες. Στους Η/Υ αυτούς είναι δυνατόν ένας αλγόριθμος να κατατμηθεί σε μικρότερα κομμάτια, που εκτελούνται παράλληλα, αφού απαιτούν διαφορετικούς υπολογιστικούς πόρους. Σε αυτά τα υπολογιστικά συστήματα, ο χρόνος εκτέλεσης ενός αλγορίθμου είναι ασύγκριτα μικρότερος. Η ανάπτυξη **παράλληλων** (parallel) αλγορίθμων είναι μία σημαντική περιοχή που απασχολεί πλήθος επιστημόνων.

Επειδή η βελτίωση των αλγορίθμων είναι ζωτικής σημασίας, στην Πληροφορική διερευνούνται οι αλγόριθμοι από την αναλυτική άποψη και ταξινομούνται ανάλογα με την επίδοσή τους. Έτσι, συνεχώς νέοι αλγόριθμοι αναπτύσσονται, ενώ άλλοι εγκαταλείπονται ως μη αποτελεσματικοί.

Ένας αλγόριθμος λέγεται **βέλτιστος** (optimal) αν αποδειχθεί ότι είναι τόσο αποτελεσματικός, ώστε δεν μπορεί να κατασκευασθεί καλύτερος.

Πολυωνυμικοί (polynomial) λέγονται οι αλγόριθμοι με πολυπλοκότητα που φράσσεται από επάνω με μία πολυωνυμική έκφραση. Για παράδειγμα, πολυωνυμικοί είναι οι αλγόριθμοι τάξης $O(n)$, $O(n^2)$, κλπ. Συνήθως αυτοί δεν απαιτούν μεγάλη υπολογιστική προσπάθεια σε αντίθεση με τους αλγορίθμους πολυπλοκότητας τάξης $O(2^n)$, $O(n.2^n)$, κλπ. που ονομάζονται **μη πολυωνυμικοί** ή **εκθετικοί**.

Υπάρχουν πολλά προβλήματα που δεν μπορούν να επιλυθούν με κάποιο αποτελεσματικό (δηλαδή πολυωνυμικό) αλγόριθμο, αλλά πρέπει να δοκιμασθούν όλες οι δυνατές περιπτώσεις, για να επιλεγεί η καλύτερη. Τα προβλήματα αυτά ονομάζονται **δυσχείριστα** (intractable). Π.χ. το πρόβλημα της συντομότερης διαδρομής που πρέπει να ακολουθήσει ένας περιοδεύων

πωλητής (traveling salesman), ώστε να περάσει μία μόνο φορά από κάθε πόλη και να επιστρέψει στην αρχική.

Η αδυναμία της επιστήμης να προτείνει αποτελεσματικούς αλγορίθμους για πολλά δυσχείριστα προβλήματα, έχει αναγκαστικά οδηγήσει στην ανάπτυξη **προσεγγιστικών** (approximate) αλγορίθμων για την επίλυσή τους, έτσι ώστε να επιτυγχάνεται μία αποδεκτή λύση σε λογικό χρόνο.

Ευριστικός (heuristic) είναι ο αλγόριθμος που είτε μπορεί να οδηγήσει σε μία καλή ή ακόμα και βέλτιστη λύση ενός προβλήματος ή στο άλλο ενδεχόμενο, σε μία λύση που απέχει πολύ από τη βέλτιστη. Οι ευριστικοί αλγόριθμοι δεν είναι τυποποιημένοι και στηρίζονται σε κάποιες τεχνικές ή εμπειρικές παρατηρήσεις ή εμπνεύσεις του προγραμματιστή. Συνήθως, οι προσεγγιστικοί αλγόριθμοι είναι ευριστικοί, αλλά όμως υπάρχουν πολλοί ευριστικοί αλγόριθμοι που δεν είναι προσεγγιστικοί.

ΙΙ. ΠΙΝΑΚΕΣ

2.1 Εισαγωγή…….. 2-1

2.2 Βασικές έννοιες πινάκων…….. 2-2

2.3 Αποθήκευση πινάκων……. 2-6

2.4 Ειδικές μορφές πινάκων…….. 2-7

2.1 ΕΙΣΑΓΩΓΗ

Εκτός από τους αλγορίθμους, σημαντική έννοια για την Πληροφορική είναι και η έννοια των δεδομένων. Τα δεδομένα αποθηκεύονται στον υπολογιστή με τη βοήθεια των λεγόμενων **δομών δεδομένων**. Θεωρώντας τους αλγόριθμους και τις δομές δεδομένων μια αδιάσπαστη ενότητα μπορεί να λεχθεί, ότι η ενότητα αυτή τελικά αποτελεί τη βάση ενός προγράμματος που επιλύει ένα πρόβλημα.

Αλγόριθμοι + Δομές Δεδομένων = Προγράμματα

Τα δεδομένα ενός προβλήματος αποθηκεύονται στον Η/Υ, είτε στην κύρια μνήμη του ή στη δευτερεύουσα μνήμη του. Η αποθήκευση αυτή δε γίνεται κατά ένα τυχαίο τρόπο αλλά συστηματικά, δηλαδή χρησιμοποιώντας μία δομή. Η έννοια της δομής δεδομένων (data structure) είναι σημαντική για την Πληροφορική και ορίζεται με τον ακόλουθο τυπικό ορισμό.

Ορισμός

Δομή Δεδομένων είναι ένα σύνολο αποθηκευμένων δεδομένων που υφίστανται επεξεργασία από ένα σύνολο λειτουργιών.

Κάθε μορφή δομής δεδομένων αποτελείται από ένα σύνολο **κόμβων** (nodes). Οι βασικές λειτουργίες επί των δομών δεδομένων είναι οι ακόλουθες:

- **Προσπέλαση** (access)
  πρόσβαση σε ένα κόμβο με σκοπό να εξετασθεί ή να τροποποιηθεί το περιεχόμενό του
- **Εισαγωγή** (insertion)
  η προσθήκη νέων κόμβων σε μία υπάρχουσα δομή
- **Διαγραφή** (deletion)
  η αφαίρεση ενός κόμβου από μία υπάρχουσα δομή
- **Αναζήτηση** (searching)
  γίνεται προσπέλαση των κόμβων μίας δομής, προκειμένου να εντοπισθούν ένας ή περισσότεροι που έχουν μία δεδομένη ιδιότητα

- **Ταξινόμηση** (sorting)
  οι κόμβοι μίας δομής τοποθετούνται σε αύξουσα ή φθίνουσα σειρά
- **Αντιγραφή** (copying)
  όλοι οι κόμβοι ή μερικοί από τους κόμβους μίας δομής αντιγράφονται σε μία άλλη δομή
- **Συγχώνευση** (merging)
  δύο ή περισσότερες δομές συνενώνονται σε μία ενιαία δομή
- **Διαχωρισμός** (separation)
  αποτελεί την αντίστροφη πράξη της συγχώνευσης

Στην πράξη σπάνια χρησιμοποιούνται όλες οι λειτουργίες για κάποια δομή. Παρατηρείται συχνά το φαινόμενο μία δομή δεδομένων να είναι αποδοτικότερη από μία άλλη δομή με κριτήριο κάποια λειτουργία, για παράδειγμα την αναζήτηση, αλλά λιγότερο αποδοτική για κάποια άλλη λειτουργία, για παράδειγμα την εισαγωγή. Αυτές οι παρατηρήσεις εξηγούν αφ' ενός την ύπαρξη διαφορετικών δομών, και αφ' ετέρου τη σπουδαιότητα της επιλογής της κατάλληλης δομής κάθε φορά.

2.2 ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ ΠΙΝΑΚΩΝ

Με τον όρο στατική δομή δεδομένων εννοείται ότι το ακριβές μέγεθος της απαιτούμενης κύριας μνήμης καθορίζεται κατά τη στιγμή του προγραμματισμού τους, και κατά συνέπεια κατά τη στιγμή της μετάφρασης του προγράμματος και όχι κατά τη στιγμή της εκτέλεσής του. Μία άλλη σημαντική διαφορά σε σχέση με τις δυναμικές δομές – που θα περιγραφούν παρακάτω – είναι ότι τα στοιχεία των στατικών δομών αποθηκεύονται σε συνεχόμενες θέσεις μνήμης.

Στην πράξη, οι στατικές δομές υλοποιούνται με **πίνακες** που μας είναι γνωστοί από άλλα μαθήματα και υποστηρίζονται από κάθε γλώσσα προγραμματισμού. Μπορούμε να ορίσουμε τον πίνακα ως μια δομή που περιέχει στοιχεία του ίδιου τύπου (δηλαδή ακέραιους, πραγματικούς κλπ.). Η

δήλωση των στοιχείων ενός πίνακα και η μέθοδος αναφοράς τους εξαρτάται από τη συγκεκριμένη γλώσσα υψηλού επιπέδου που χρησιμοποιείται. Όμως γενικά η αναφορά στα στοιχεία ενός πίνακα γίνεται με τη χρήση του συμβολικού ονόματος του πίνακα ακολουθούμενου από την τιμή ενός ή περισσοτέρων δεικτών (indexes) σε παρένθεση ή αγκύλη.

Ένας πίνακας μπορεί να είναι μονοδιάστατος, αλλά γενικά μπορεί να είναι δισδιάστατος, τρισδιάστατος και γενικά ν-διάστατος πίνακας.

Παράδειγμα 1

Εύρεση του ελάχιστου στοιχείου ενός μονοδιάστατου πίνακα.

Αλγόριθμος Ελάχιστο_πίνακα
Δεδομένα table[n], i, min
min = table[1]
Για i από 2 μέχρι n
Αν table[i] < n τότε
min = table[i]
Τέλος επανάληψης
Εμφάνισε min
Τέλος Ελάχιστο_Πίνακα

Υλοποίηση στην C:

```
#define N 10
main()
{
   int table[N] = {11, 23, 2, 34, 56, 65, 7, 9, 1, 25};
   int i, min;

   min = table[0];
   for (i=1; i<N; i++)
        if (table[i]<min)
```

```
            min = table[i];
   printf("Min = %d",min);
}
```

Παράδειγμα 2

Εύρεση αθροίσματος στοιχείων δισδιάστατου πίνακα.

```
Αλγόριθμος  Άθροισμα_στοιχείων_Πίνακα
Δεδομένα  table[m][n], i, j, sum, row[m], col[n]
Για i από 1 μέχρι m
    Για j από 1 μέχρι n
        Διάβασε table[i][j]
   Τέλος επανάληψης
Τέλος επανάληψης
sum = 0;
Για i από 1 μέχρι m
   row[i] = 0
Τέλος επανάληψης
Για j από 1 μέχρι n
   col[j] = 0
Τέλος επανάληψης
Για i από 1 μέχρι m
   Για j από 1 μέχρι n
        sum = sum + table[i][j]
        row[i] = row[i] + table[i][j]
        colo[j] = col[j] + table[i][j]
   Τέλος επανάληψης
Τέλος επανάληψης
Εμφάνισε sum
Για i από 1 μέχρι m
   Εμφάνισε row[i]
```

Τέλος επανάληψης

Για j από 1 μέχρι n

Εμφάνισε col[i]

Τέλος επανάληψης

Τέλος Άθροισμα_στοιχείων_Πίνακα

Υλοποίηση στην C:

```
#define M 5
#define N 5

main()
{
   int table[M][N], row[M], col[N];
   int i, j, sum;

   for (i=0; i<M; i++)
      for (j=0; i<N; i++)
          scanf("%d",&table[i][j]);
   sum = 0;
   for (i=0; i<M; i++)
        row[i] = 0;
   for (j=0; i<N; i++)
        col[j] = 0;
   for (i=0; i<M; i++)
      for (j=0; i<N; i++)
          {
           sum = sum + table[i][j];
           row[i] = row[i] + table[i][j];
           col[j] = col[j] + table[i][j];
          }
   printf("sum = %d", sum);
```

```
   for (i=0; i<M; i++)
        printf("row[%d] = %d\n", i, row[i]);
   for (j=0; i<N; i++)
        printf("col[%j] = %d\n", j, col[j]);
}
```

	table					row	
	4	16	5	21	7	53	
	28	9	38	13	51	139	
	17	67	22	40	30	176	
	20	40	10	3	13	86	
	21	34	48	29	26	158	
col	90	166	123	106	127	612	**sum**

Πίνακας 2.1 *Άθροισμα στοιχείων πίνακα*

2.3 ΑΠΟΘΗΚΕΥΣΗ ΠΙΝΑΚΩΝ

Ένας μονοδιάστατος πίνακας p[N] απαιτεί N συνεχόμενες θέσεις μνήμης για να αποθηκευτεί. Η αρχική θέση του πίνακα στη μνήμη του Η/Υ θα αναφέρεται με το συμβολισμό loc(p[lowerbound]), όπου το lowerbound είναι το κάτω όριο του πίνακα πχ. 0.

Η θέση του i-οστού στοιχείου του πίνακα p εκφράζεται ως

loc(p[i]) = loc(p[lowerbound]) + i – lowerbound

Θεωρούμε ότι κάθε στοιχείο του πίνακα καταλαμβάνει μία λέξη της κύριας μνήμης του Η/Υ.

Οι δύο πιο συνηθισμένοι τρόποι αποθήκευσης ενός δισδιάστατου πίνακα p[M][N] είναι η

- **Διάταξη κατά γραμμές** (row-major order)
- **Διάταξη κατά στήλες** (column-major order)

Στην πρώτη μέθοδο φυλάγεται πρώτα η πρώτη γραμμή, ακολουθούμενη από τη δεύτερη γραμμή και τελευταία αποθηκεύεται η m-οστή γραμμή. Στη

δεύτερη μέθοδο φυλάγεται πρώτα η πρώτη στήλη, ακολουθούμενη από τη δεύτερη στήλη και τελευταία αποθηκεύεται η n-οστή στήλη.

Αν ο πίνακας έχει φυλαχθεί κατά στήλες τότε η θέση του στοιχείου p[i][j] είναι:

loc(p[i][j]) = loc(p[0][0]) + i + M*j

2.4 ΕΙΔΙΚΕΣ ΜΟΡΦΕΣ ΠΙΝΑΚΩΝ

Αν ο πίνακας έχει κάποια ειδική μορφή τότε είναι δυνατόν να χρησιμοποιηθεί κάποια ιδιαίτερη τεχνική για την ελάττωση του χώρου φύλαξής του. Αυτό γίνεται με την ευθύνη του προγραμματιστή που πρέπει να λάβει υπ' όψη του τον τρόπο φύλαξης του πίνακα. Τέτοιες μορφές πινάκων είναι οι

- **Συμμετρικοί πίνακες** (symmetric)
  όπου τα στοιχεία είναι συμμετρικά ως προς την κύρια διαγώνιο. Εδώ απαιτούνται **n(n+1)/2** θέσεις μνήμης.
- **Τριγωνικοί πίνακες (**triangular**)**
  όπου τα στοιχεία πάνω ή κάτω της κύριας διαγωνίου είναι 0. Εδώ επίσης απαιτούνται **n(n+1)/2** θέσεις μνήμης
- **Τριδιαγώνιοι πίνακες** (tridiagonal)
  όπου όλα τα στοιχεία πλην της κύριας διαγωνίου και των δύο διπλανών διαγωνίων είναι 0. Εδώ απαιτούνται **3*n-2** θέσεις μνήμης

- **Αραιοί πίνακες** (sparse)
  όπου ένα μεγάλο ποσοστό των στοιχείων του πίνακα έχουν την τιμή 0 (συνήθως > 80%). Υπάρχουν πολλοί τρόποι για την αποδοτική από πλευράς χώρου αποθήκευση αραιών πινάκων. Π.χ. ένας τρόπος είναι να αποθηκεύεται κάθε μη μηδενικό στοιχεί του πίνακα ως μία τριάδα αριθμών. Οι δύο πρώτοι αριθμοί δείχνουν τη θέση του στοιχείου στον πίνακα και ο τρίτος αριθμός την τιμή του στοιχείου. έτσι αν υπάρχουν **n** μη μηδενικά στοιχεία απαιτούνται **3*n** θέσεις μνήμης.

ΙΙΙ. ΑΝΑΔΡΟΜΗ

3.1 Εισαγωγή…… 3-1

3.2 Υπολογισμός παραγοντικού……... 3-1

3.3 Υπολογισμός δύναμης…… 3-3

3.1 ΕΙΣΑΓΩΓΗ

Αναδρομή είναι η μέθοδος κατά την οποία, σε μία γλώσσα προγραμματισμού, μία διαδικασία ή συνάρτηση έχει την δυνατότητα να καλεί τον εαυτό της. Η υλοποίηση της αναδρομής βασίζεται στη έννοια της στοίβας. Σε κάθε κλήση μίας υπορουτίνας πρέπει να φυλάγονται οι διευθύνσεις επιστροφής. Όταν μία υπορουτίνα καλεί τον εαυτό της θα πρέπει επίσης να φυλάγονται οι προηγούμενες τιμές των μεταβλητών και να χρησιμοποιούνται όταν τελειώσει η αναδρομική κλήση.

Η χρήση της αναδρομής διευκολύνει πολύ τον προγραμματιστή στην ανάπτυξη και τον έλεγχο ενός προγράμματος. Θα πρέπει όμως να χρησιμοποιείται με μέτρο, γιατί η εκτέλεση ενός αναδρομικού προγράμματος έχει χρονικό κόστος. Γενικά, ανάμεσα σε ένα επαναληπτικό και ένα αναδρομικό πρόγραμμα θα πρέπει να προτιμάμε το πρώτο, εκτός και αν η ανάπτυξή του μας δυσκολεύει ιδιαίτερα. Η αναδρομή ενδείκνυται σε προβλήματα κάποιας σχετικής πολυπλοκότητας, που εξ' ορισμού τα εκφράζουμε αναδρομικά.

3.2 ΥΠΟΛΟΓΙΣΜΟΣ ΠΑΡΑΓΟΝΤΙΚΟΥ

- Επαναληπτικός ορισμός:
  n! = 1*2*3*……..(n-1)*n

Συνάρτηση με επανάληψη:

```
int  factorial(int n)
{
   int i,f;

   f = 1;
   for (i=2; i<=n; i++)
        f = f*i;
   return f;
```

```
}
```

- Αναδρομικός ορισμός:

$$n! = \begin{cases} 1 & \text{αν } n = 0, \\ n*(n-1)! & \text{αν } n > 0 \end{cases}$$

Συνάρτηση με αναδρομή:

```
int  factorial(int n)
{
   int f;

   if (n = = 0)
        f = 1;
   else
        f = n*factorial(n-1);
   return f;
}
```

Ας παρακολουθήσουμε τις τιμές των μεταβλητών κατά την κλήση των δύο συναρτήσεων , π.χ. για n = 4:

Επαναληπτική μέθοδος:

f = 1

i = 2

f = 1*2 = 2

i = 3

f = 2*3 = 6

i = 4

f = 6*4 = 24

Αναδρομική μέθοδος:

factorial(4) = 4*factorial(3)

factorial(3) = 3*factorial(2)

factorial(2) = 2*factorial(1)

factorial(1) = 1*factorial(0)

factorial(0) = 1

Ακολούθως, η τελευταία τιμή 1 μεταβιβάζεται στην προηγούμενη κλήση και έτσι υπολογίζεται το factorial(1) = 1. Κατά τον ίδιο τρόπο έχουμε

factorial(2) = 2*1 = 2

factorial(3) = 3*2 = 6

factorial(4) = 4*6 = 24

3.3 ΥΠΟΛΟΓΙΣΜΟΣ ΔΥΝΑΜΗΣ

- Επαναληπτικός ορισμός:
  x^n = x*x*x*……..*x, n φορές

Συνάρτηση με επανάληψη:

```
int  power(int x, int n)
{
   int i,p;

   p = 1;
   for (i=1; i<=n; i++)
        p = p*x;
   return p;
}
```

- Αναδρομικός ορισμός:

  $x^n = 1$ αν n = 0,

  $= x * x^{n-1}$ αν n > 0

Συνάρτηση με αναδρομή:

```
int  power(int x, int n)
{
   int p;

   if (n = = 0)
        p = 1;
   else
        p = x*power(x,n-1);
   return p;
}
```

IV. ΑΝΑΖΗΤΗΣΗ

4.1 Εισαγωγή..... 4-1

4.2 Σειριακή Αναζήτηση..... 4-1

4.3 Δυαδική Αναζήτηση..... 4-3

4.1 ΕΙΣΑΓΩΓΗ

Αναζήτηση (searching) είναι η διεργασία της εύρεσης κάποιας συγκεκριμένης τιμής ανάμεσα από ένα σύνολο τιμών. Το πρόβλημα της αναζήτησης είναι ένα από τα πιο ενδιαφέροντα προβλήματα της Επιστήμης των Υπολογιστών λόγω της μεγάλης πρακτικότητάς του. Το πρόβλημα γίνεται ακόμα πιο ενδιαφέρον αν ληφθεί υπόψη η μεγάλη ποικιλία των χαρακτηριστικών της δομής όπου αποθηκεύονται τα δεδομένα (π.χ. στατική ή δυναμική, τρόπος οργάνωσης, μέσο αποθήκευσης, κλπ.).

Στη συνέχεια θα εξετασθούν δύο από τις πιο γνωστές μέθοδοι αναζήτησης πινάκων:

- **Σειριακή αναζήτηση** (Sequential search)
- **Δυαδική αναζήτηση** (Binary search)

4.2 ΣΕΙΡΙΑΚΗ ΑΝΑΖΗΤΗΣΗ

Η πιο απλή μέθοδος αναζήτησης ενός στοιχείου σε ένα πίνακα είναι η **σειριακή αναζήτηση.** Γίνεται προσπέλαση των στοιχείων του πίνακα από την πρώτη θέση μέχρι την τελευταία. Κάθε φορά ελέγχεται το στοιχείο της τρέχουσας θέσης αν είναι το ζητούμενο, οπότε και τελειώνει η αναζήτηση. Αν προσπελασθεί όλος ο πίνακας και δεν βρεθεί το ζητούμενο στοιχείο, τότε η αναζήτηση θεωρείται ανεπιτυχής.

Παρακάτω δίνεται ο αλγόριθμος (σε ψευδοκώδικα) για τη σειριακή αναζήτηση, με δύο διαφορετικούς τρόπους, έναν επαναληπτικό και ένα αναδρομικό.

1) Επαναληπτική έκδοση

Αλγόριθμος Σειριακή_αναζήτηση_1
Δεδομένα p[N], i, key, found, position
found = false
position = -1
i = 0
ΟΣΟ (found = = false) ΚΑΙ (i <= N)

ΑΝ p[i] = = key ΤΟΤΕ

found = true

position = i

ΑΛΛΙΩΣ

i = i+1

ΤΕΛΟΣ ΑΝ

ΤΕΛΟΣ ΕΠΑΝΑΛΗΨΗΣ

ΑΝ found = = true ΤΟΤΕ

ΕΜΦΑΝΙΣΕ position

ΑΛΛΙΩΣ

ΕΜΦΑΝΙΣΕ "Δεν υπάρχει"

ΤΕΛΟΣ Σειριακή_αναζήτηση_1

2) Αναδρομική έκδοση

Αλγόριθμος Σειριακή_αναζήτηση_2

Δεδομένα p[N], i, key, found, position

ΑΝ i > N ΤΟΤΕ

found = false

ΑΛΛΙΩΣ

ΑΝ p[i] = = key ΤΟΤΕ

found = true

position = i

ΑΛΛΙΩΣ

i = i+1

Σειριακή_αναζήτηση_2(p,key,found,position)

ΤΕΛΟΣ ΑΝ

ΤΕΛΟΣ ΑΝ

ΤΕΛΟΣ Σειριακή_αναζήτηση_2

ΑΝ found = = true ΤΟΤΕ

ΕΜΦΑΝΙΣΕ position

ΑΛΛΙΩΣ

ΕΜΦΑΝΙΣΕ "Δεν υπάρχει"

ΤΕΛΟΣ ΑΝ

4.3 ΔΥΑΔΙΚΗ ΑΝΑΖΗΤΗΣΗ

Όταν τα στοιχεία του πίνακα είναι ταξινομημένα, τότε είναι δυνατόν να επιταχυνθεί η διαδικασία αναζήτησης, εφαρμόζοντας επιλεκτικά και όχι σειριακά την αναζήτηση. Η μέθοδος που περιγράφεται παρακάτω είναι η πιο γνωστή σε αυτή την κατηγορία. Πιο συγκεκριμένα, συγκρίνεται η τιμή του ζητούμενου στοιχείου με το μεσαίο στοιχείο του πίνακα. Αν είναι ίσα, τότε η αναζήτηση έχει τελειώσει. Αν όχι, τότε μπορεί ο μισός πίνακας να εξαιρεθεί από τις επόμενες αναζητήσεις γιατί αποκλείεται να περιέχει το ζητούμενο στοιχείο, εφόσον αυτός είναι ταξινομημένος. Η διαδικασία αυτή επαναλαμβάνεται μέχρις ότου βρεθεί το στοιχείο ή διαπιστωθεί ότι δεν υπάρχει. Η μέθοδος αυτή είναι η **δυαδική.**

Είναι φανερό ότι με τη μέθοδο αυτή μετά από κάθε σύγκριση το μέγεθος του πίνακα που θα ανιχνευθεί στη συνέχεια είναι το ½, ¼, ... του αρχικού μεγέθους. Η μέγιστη τιμή των απαιτούμενων συγκρίσεων δίνεται από τη σχέση

Μέγιστη τιμή απαιτούμενων συγκρίσεων = $\log_2 N$

Το πλεονέκτημα της μεθόδου αυτής είναι ότι είναι πολύ σταθερή, δηλαδή συνήθως η επίδοση είναι πολύ κοντά στη μέση τιμή των συγκρίσεων. Το μειονέκτημα είναι ότι ο πίνακας πρέπει να είναι ταξινομημένος.

Παρακάτω δίνεται ο αλγόριθμος (σε ψευδοκώδικα) για τη δυαδική αναζήτηση, επίσης με δύο διαφορετικούς τρόπους, έναν επαναληπτικό και ένα αναδρομικό.

1) Επαναληπτική έκδοση

Αλγόριθμος Δυαδική_αναζήτηση_1

Δεδομένα p[N], mid, left, right, key, found, position

found = false

position = -1

left = 0

right = N-1

ΟΣΟ (found = = false) ΚΑΙ (left <= right)

```
        mid = (left+right) / 2
        ΑΝ key = = p[mid] ΤΟΤΕ
                found = true
                position = mid
        ΑΛΛΙΩΣ
                ΑΝ key < p[mid] ΤΟΤΕ
                        right = mid-1
                ΑΛΛΙΩΣ
                        left = mid+1
                ΤΕΛΟΣ ΑΝ
        ΤΕΛΟΣ ΑΝ
ΤΕΛΟΣ  ΕΠΑΝΑΛΗΨΗΣ
ΑΝ found = = true ΤΟΤΕ
        ΕΜΦΑΝΙΣΕ position
ΑΛΛΙΩΣ
        ΕΜΦΑΝΙΣΕ “Δεν υπάρχει”
ΤΕΛΟΣ ΑΝ
ΤΕΛΟΣ Δυαδική_αναζήτηση_1
```

2) Αναδρομική έκδοση

```
Αλγόριθμος Δυαδική_αναζήτηση_2
Δεδομένα  p[N], mid, left, right, key, position
ΑΝ  left > right  ΤΟΤΕ
        position = -1
ΑΛΛΙΩΣ
        mid = (left + right) / 2
         ΑΝ p[mid] = = key ΤΟΤΕ
                position = mid
        ΑΛΛΙΩΣ
                ΑΝ key < p[mid]
                    position = Δυαδική_αναζήτηση_2(left, mid-1, p, key)
```

ΑΛΛΙΩΣ

position = Δυαδική_αναζήτηση_2(mid+1, right, p, x)

ΤΕΛΟΣ ΑΝ

ΤΕΛΟΣ ΑΝ

ΤΕΛΟΣ Δυαδική_αναζήτηση_2

ΑΝ position = = -1 ΤΟΤΕ

ΕΜΦΑΝΙΣΕ "Δεν υπάρχει"

ΑΛΛΙΩΣ

ΕΜΦΑΝΙΣΕ position

ΤΕΛΟΣ ΑΝ

Αριθμός συγκρίσεων με Δυαδική Αναζήτηση:

Στοιχεία Ν	Συγκρίσεις
10	4
100	7
1000	10
10000	14
100000	17
1000000	20
10000000	24
100000000	27
1000000000	30

Πίνακας 4.1 *Αριθμός συγκρίσεων με Δυαδική Αναζήτηση*

V. ΤΑΞΙΝΟΜΗΣΗ

5.1 Εισαγωγή……... 5-1

5.2 Ταξινόμηση με απευθείας επιλογή…….... 5-2

5.3 Ταξινόμηση με απευθείας εισαγωγή….... 5-3

5.4 Ταξινόμηση φυσαλίδας……. 5-4

5.5 Γρήγορη ταξινόμηση……..… 5-5

5.1 ΕΙΣΑΓΩΓΗ

Ταξινόμηση είναι η διαδικασία της τοποθέτησης ενός συνόλου στοιχείων σε μία ιδιαίτερη σειρά. Η σειρά αυτή είναι συνήθως αύξουσα (ascending) ή φθίνουσα (descending). Σκοπός της ταξινόμησης είναι η διευκόλυνση της αναζήτησης στοιχείων του ταξινομημένου συνόλου. Η χρησιμότητα της ταξινόμησης φαίνεται στην πράξη σε περιπτώσεις αναζήτησης σε τηλεφωνικούς καταλόγους, σε βιβλιοθήκες, σε λεξικά, σε διάφορες δημόσιες υπηρεσίες και οργανισμούς και γενικά παντού όπου υπάρχουν αποθηκευμένα δεδομένα και πρέπει να αναζητηθούν και να βρεθούν.

Το θέμα της ταξινόμησης παρουσιάζει μεγάλο ενδιαφέρον και έχουν αναπτυχθεί αρκετοί αλγόριθμοι ταξινόμησης. Στο κεφάλαιο αυτό θα περιγραφούν κάποιοι απ' αυτούς. Επίσης, θα πρέπει να λαμβάνεται υπόψη:

- Η επιλογή μίας συγκεκριμένης δομής δεδομένων επηρεάζει τους αλγορίθμους που εκτελούν ένα έργο.
- Η επιλογή ενός αλγορίθμου είναι μία δύσκολη διαδικασία που μπορεί να διευκολυνθεί με την ανάλυση της επίδοσης τους (performance analysis).
- Υπάρχει μία θεωρητικά βέλτιστη επίδοση που κανείς αλγόριθμος ταξινόμησης δεν μπορεί να ξεπεράσει.

5.2 ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΑΠΕΥΘΕΙΑΣ ΕΠΙΛΟΓΗ (Straight Selection)

Η μέθοδος έχει ως εξής:

- Επιλέγουμε το μικρότερο στοιχείο
- Το ανταλλάσσουμε με το πρώτο στοιχείο
- Επαναλαμβάνουμε για τα υπόλοιπα στοιχεία, μέχρι να μείνει ένα

Π.χ.

44	55	12	42	94	18	**06**	67
06	**55**	**12**	42	94	18	44	67
06	12	**55**	42	94	**18**	44	67
06	12	18	**42**	94	55	44	67
06	12	18	42	**94**	55	**44**	67
06	12	18	42	44	**55**	94	67
06	12	18	42	44	55	**94**	**67**
06	12	18	42	44	55	67	94

Ο αλγόριθμος σε C:

```
for (i=0; i<N-1; i++)
  {
      k = i;
      min = p[i];
      for (j = i+1; j<N; j++)
          {
             if (p[j] < min)
              {
                 k = j;
                 min = p[j];
              }
          }
      p[k] = p[i] ;
      p[i] = min;
  }
```

5.3 ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΑΠΕΥΘΕΙΑΣ ΕΙΣΑΓΩΓΗ (Straight Insertion)

Η μέθοδος είναι πολύ δημοφιλής στους χαρτοπαίχτες και έχει ως εξής:

- Κάθε στοιχείο, ξεκινώντας από το δεύτερο, τοποθετείται στη σωστή του θέση μετακινώντας, αν χρειαστεί, τα στοιχεία δεξιά του κατά μία θέση.

Π.χ.

44	**55**	12	42	94	18	06	67
44	55	**12**	42	94	18	06	67
12	44	55	**42**	94	18	06	67
12	42	44	55	**94**	48	06	67
12	42	44	55	94	**18**	06	67
12	18	42	44	55	94	**06**	67
06	12	18	42	44	55	94	**67**
06	12	18	42	44	55	67	94

Ο αλγόριθμος σε C:

```
for (i=2; i<=N; i++)
  {
      x = p[i];
      p[0] = x;
      j = i-1;
      while (x < p[j])
          {
             p[j+1] = p[j] ;
            j = j-1 ;
          }
      p[j+1] = x ;
}
```

5.4 ΤΑΞΙΝΟΜΗΣΗ ΦΥΣΑΛΙΔΑΣ (Bubble Sort)

Ο αλγόριθμος βασίζεται στην αρχή της σύγκρισης και ανταλλαγής ζευγών από γειτονικά στοιχεία, μέχρις ότου ταξινομηθούν όλα τα στοιχεία. Κάθε φορά μετακινείται το μικρότερο στοιχείο της ακολουθίας προς το αριστερό άκρο.

Π.χ.

44	55	12	42	94	18	06	67
06	44	55	12	42	94	18	67
06	12	44	55	18	42	94	67
06	12	18	44	55	42	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94

Ο αλγόριθμος σε C:

```
for (i=1; i<N; i++)
        for (j=N-1; j>=i; j--)
                if (p[j-1] > p[j])
                {
                        temp = p[j-1];
                        p[j-1] = p[j] ;
                        p[j] = temp ;
                }
```

5.5 ΓΡΗΓΟΡΗ ΤΑΞΙΝΟΜΗΣΗ (Quick Sort)

Ο αλγόριθμος βασίζεται στην αρχή της αντιμετάθεσης και είναι η καλύτερη γνωστή μέθοδος ταξινόμησης για τυχαία στοιχεία. Η γρήγορη ταξινόμηση στηρίζεται στην παρατήρηση ότι είναι προτιμότερο οι αντιμεταθέσεις να γίνονται μεταξύ απομακρυσμένων στοιχείων.

Στην αρχή λαμβάνεται το μεσαίο στοιχείο του πίνακα και μετακινείται στη θέση όπου τελικά θα αποθηκευτεί στο ταξινομημένο διάνυσμα. Μετά τον προσδιορισμό της τελικής αυτής θέσης, γίνεται αναδιάταξη των υπόλοιπων στοιχείων έτσι ώστε να μην υπάρχει κανένα μικρότερο στοιχείο προς τα αριστερά του και κανένα μεγαλύτερο στοιχείο προς τα δεξιά του. Έτσι, το μεσαίο αυτό στοιχείο παίζει το ρόλο του άξονα (pivot) και ο πίνακας έχει διαμεριστεί κατά τέτοιο τρόπο ώστε το αρχικό πρόβλημα έχει αναχθεί σε δύο απλούστερα προβλήματα, στην ανεξάρτητη δηλαδή ταξινόμηση των δύο υποπινάκων. Μετά τον διαμερισμό του πίνακα, η ίδια διαδικασία εφαρμόζεται στους δύο υποπίνακες, ύστερα στους υποπίνακες των υποπινάκων, κοκ., μέχρις ότου ο κάθε υποπίνακας να αποτελείται από ένα μόνο στοιχείο. Στο σημείο αυτό ο αρχικός πίνακας έχει ταξινομηθεί.

П.χ.

36 i	47	18	59	**32** x	73	41	27 j	51 j
27	47 i	18	59	**32** j	73 j	41 j	36	51
27	**32** i	18 j	59 j	47 j	73	41	36	51
27	18 i	**32** j i	59	47	73	41	36	51
[**27** i	18] j	32	[59 i	47 i	**73** i	41	36	51] j
18 i	**27** j i		59	47	51 i	41 i	36 i	**73** j i
18	27		[59 i	47	**51**	41	36] j	73
			36 i	47 i	51 i	41 j	59 j	
			36	47	41 i	**51** j i	59	
			[36 i	**47** i	41] j	51		
			36	41	**47** j i			
			[**36** i j	41] j	47			
			36	41				

Ο αλγόριθμος σε C:

```
void quicksort(int left, int right, int p[])
{
   int i, j, mid, x, temp;

   if (left < right)
     {
        i = left;
        j = right;
        mid = (left+right)/2;
        x = p[mid];
        while (i < j)
        {
           while (p[i] < x)
              i++;
           while (p[j] > x)
              j--;
           if (i < j)
           {
              temp = p[i];
              p[i] = p[j];
              p[j] = temp;
           }
        }
        quicksort(left,j-1,p);
        quicksort(j+1,right,p);
   }
}
```

VI. ΓΡΑΜΜΙΚΕΣ ΛΙΣΤΕΣ

6.1 Εισαγωγή..... 6-1

6.2 Σειριακές Λίστες..... 6-1

6.2.1 Στοίβα..... 6-1

6.2.2 Ουρά..... 6-3

6.3 Συνδεδεμένες Λίστες..... 6-6

6.3.1 Απλή συνδεδεμένη λίστα..... 6-7

6.3.2 Στοίβα ως συνδεδεμένη λίστα..... 6-11

6.3.3 Ουρά ως συνδεδεμένη λίστα..... 6-13

6.1 ΕΙΣΑΓΩΓΗ

Γραμμική λίστα (**linear list**) είναι ένα πεπερασμένο σύνολο από κόμβους $x_1, x_2, ..., x_n$ όπου το στοιχείο x_k προηγείται του στοιχείου x_{k+1} και έπεται του x_{k-1}.

Κατατάσσονται συνήθως σε δύο κατηγορίες:

- Σειριακές γραμμικές λίστες (sequential linear lists)
- Συνδεδεμένες γραμμικές λίστες (linked linear lists)

Στην πρώτη κατηγορία καταλαμβάνονται συνεχόμενες θέσεις μνήμης του Η/Υ για την αποθήκευση των κόμβων.

Στην δεύτερη κατηγορία οι κόμβοι των λιστών βρίσκονται σε απομακρυσμένες θέσεις που είναι μεταξύ τους συνδεδεμένες.

Επίσης, οι γραμμικές λίστες χαρακτηρίζονται σαν:

- Στατικές δομές δεδομένων (static data structures)
- Δυναμικές δομές δεδομένων (dynamic data structures)

Στην πρώτη κατηγορία, κατά τον προγραμματισμό των λειτουργιών των λιστών, έχει προκαθορισθεί το μέγεθος της μνήμης που απαιτείται για την αποθήκευση των λιστών.

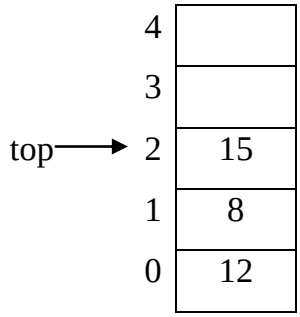
Στην δεύτερη κατηγορία, μία λίστα μπορεί να αυξομειωθεί κατά την διάρκεια εκτέλεσης του προγράμματος.

6.2 ΣΕΙΡΙΑΚΕΣ ΛΙΣΤΕΣ

6.2.1 Στοίβα (stack)

Μπορούμε να την παραλληλίσουμε σαν μία στοίβα από πιάτα. Κάθε νέο στοιχείο τοποθετείται στην κορυφή (top). Το στοιχείο που βρίσκεται στην κορυφή της στοίβας εξέρχεται πρώτο. Αυτή η μέθοδος επεξεργασίας ονομάζεται

LIFO (**L**ast **I**n **F**irst **O**ut)



Σχήμα 6.1 *LIFO (Last In First Out)*

Μία στατική στοίβα υλοποιείται με τη χρήση ενός μονοδιάστατου πίνακα και ενός δείκτη. Δύο είναι οι κύριες λειτουργίες στη στοίβα:

- **Ώθηση** (push) στοιχείου στην κορυφή της στοίβας
- **Απώθηση** (pop) στοιχείου από τη στοίβα

Η διαδικασία της **ώθησης** πρέπει οπωσδήποτε να ελέγχει μήπως η στοίβα είναι γεμάτη, οπότε έχουμε υπερχείλιση (overflow).

Αντίστοιχα, η διαδικασία της **απώθησης** πρέπει να ελέγχει αν η στοίβα έχει αδειάσει, οπότε έχουμε υποχείλιση (underflow).

Υλοποίηση στοίβας σε C:

```
#define  N  100
int stack[N], top = -1;

void push(int stack[],int *t,int obj)
{
   if (*t = = N-1)
    {
        printf("Stack overflow...\n");
        getch();
        abort();
    }
  else
```

```
        stack[++(*t)] = obj;
}

int  pop(int  stack[],int *t)
{
  int  r ;

  if (*t < 0)
    {
       printf("Stack empty...\n");
       printf("Error in expression.\n");
       getch();
       abort();
    }
  else
       r = stack[(*t)--];
  return  r;
}
```

6.2.2 **Ουρά** (queue)

Την έννοια της ουράς την συναντάμε συχνά στην καθημερινή μας ζωή, π.χ. ουρά αναμονής με ανθρώπους. Το άτομο που είναι πρώτο στην ουρά, εξυπηρετείται και εξέρχεται. Το άτομο που μόλις καταφθάνει, τοποθετείται στο τέλος της ουράς. Η μέθοδος αυτή επεξεργασίας ονομάζεται

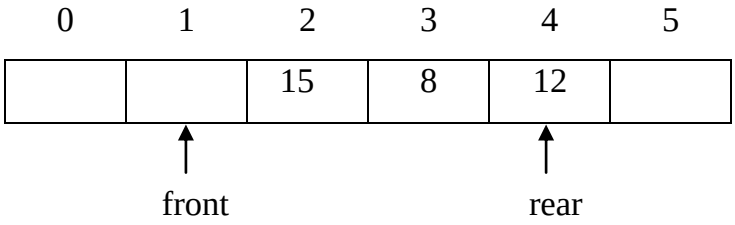
FIFO (**F**irst **I**n **F**irst **O**ut)

Δύο βασικές λειτουργίες:

- **Εισαγωγή** (enqueue) στοιχείου στο πίσω άκρο της ουράς
- **Εξαγωγή** (dequeue) στοιχείου από το εμπρός άκρο της ουράς

Επομένως, για την υλοποίηση της ουράς χρειάζονται ένας πίνακας και δύο δείκτες, ο εμπρός (front) και ο πίσω (rear).

Επειδή βολεύει – προγραμματιστικά – ο δείκτης rear δείχνει πάντα στο τελευταίο στοιχείο, ενώ ο δείκτης front δείχνει μία θέση πριν το πρώτο στοιχείο και, κατά συνέπεια, η ισότητα των δύο δεικτών αποδεικνύει ότι η ουρά είναι άδεια.



Σχήμα 6.2 *Ουρά (queue)*

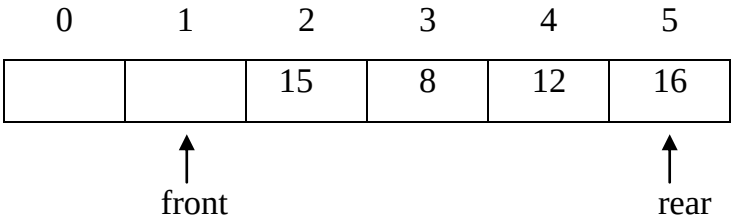
Υλοποίηση ουράς σε C:

```
#define N 100

int  q[N], front = -1, rear = -1;

void enqueue(int q[], int *r, int obj)
{
    if (*r = = N-1)
      {
         printf("Queue is full...");
         getch();
      }
    else
        q[++(*r)] = obj;
 }

void  dequeue(int q[], int *f, int  r)
```

```
{
  int  x;

  if (*f = = r)
       printf("Queue is empty...\n");
  else
     {
        x = q[++(*f)];
        printf("%d has been deleted...",x);
     }
}
```

Η υλοποίηση της ουράς με πίνακα έχει ένα μειονέκτημα. Υπάρχει περίπτωση ο rear να φθάσει στο πάνω όριο του πίνακα, αλλά στην ουσία να μην υπάρχει υπερχείλιση, επειδή ο front θα έχει αυξηθεί (εικονική υπερχείλιση).

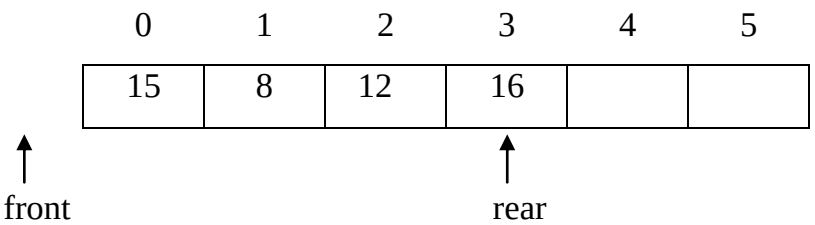
π.χ.



Σχήμα 6.3 *Εικονική υπερχείλιση*

Στην πραγματικότητα υπάρχει ελεύθερος χώρος για την εισαγωγή και νέων στοιχείων. Μία λύση θα ήταν να μεταφερθούν τα στοιχεία στο αριστερό άκρο του πίνακα.

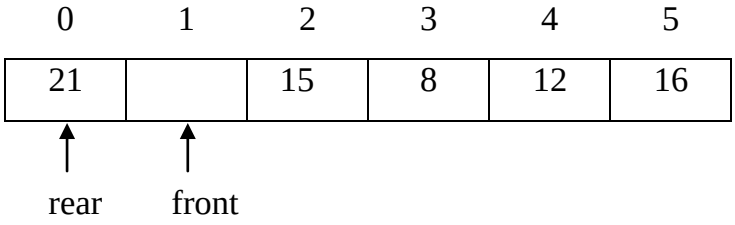
π.χ.



Σχήμα 6.4 *Λύση Εικονικής υπερχείλισης*

```
elements = front – rear;
first = front + 1;
for (i=0; i<elements; i++)
      q[i] = q[first++];
front = -1;
rear = elements – 1;
```

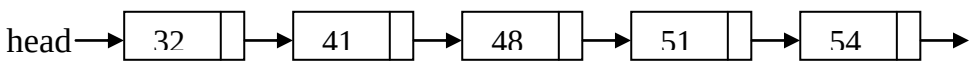
Μία πιο αποτελεσματική υλοποίηση θα ήταν η ουρά να αναδιπλώνεται, δηλαδή όταν ο rear = N-1, να επανατοποθετείται στο 0. Η δομή αυτή ονομάζεται **κυκλική ουρά** (circular queue).



Σχήμα 6.5 *Κυκλική ουρά*

6.3 ΣΥΝΔΕΔΕΜΕΝΕΣ ΛΙΣΤΕΣ

Οι στατικές δομές που μελετήθηκαν, παρουσιάζουν προβλήματα στην εισαγωγή και διαγραφή κόμβων και στην αποδοτική εκμετάλλευση της διαθέσιμης μνήμης. Κύριο χαρακτηριστικό των συνδεδεμένων λιστών είναι ότι οι κόμβοι τους βρίσκονται σε απομακρυσμένες θέσεις μνήμης και η σύνδεσή τους γίνεται με δείκτες. Κατ' αυτό τον τρόπο, η εισαγωγή και διαγραφή κόμβων γίνεται πολύ πιο απλά. Ένα άλλο θετικό χαρακτηριστικό είναι ότι, δεν απαιτείται εκ των προτέρων καθορισμός του μέγιστου αριθμού κόμβων της λίστας και μπορεί η λίστα να επεκταθεί ή να συρρικνωθεί κατά την εκτέλεση του προγράμματος (δυναμικές δομές).



Σχήμα 6.6 *Συνδεδεμένες Λίστες*

Κάθε κόμβος της λίστας υλοποιείται με μία δομή (structure) με δύο στοιχεία. Το ένα
μέλος (data) περιέχει τα δεδομένα (οποιουδήποτε τύπου) του κόμβου και το άλλο μέλος (next) είναι δείκτης προς τον επόμενο κόμβο. Τοποθετείται ένας δείκτης (head) στον πρώτο κόμβο για να προσπελάζεται η λίστα, ενώ ο δείκτης του τελευταίου κόμβου δείχνει στο NULL για να εντοπίζεται το τέλος της λίστας.

6.3.1 Απλή Συνδεδεμένη Λίστα

Παράδειγμα υλοποίησης λίστας ακεραίων στην C:

```
struct node
{
  int  data;
  struct node *next;
};
typedef struct node * PTR;

α) Δημιουργία λίστας
PTR  list_create(PTR  head)
{
  PTR   current;
  int  x;

  printf("Give an integer, 0 to stop:");
  scanf("%i",&x);
  if (x = = 0)
     return  NULL;
  else
    {
      head = malloc(sizeof(struct node));
      head->data = x;
```

```
        current = head;
        printf("Give an integer, 0 to stop:");
        scanf("%i",&x);
        while (x!=0)
            {
              current->next = malloc(sizeof(struct node));
              current = current->next;
              current->data = x;
              printf("Give an integer, 0 to stop:");
               scanf("%i",&x);
             }
        current->next = NULL;
     }
  return head;
}
```

β) Εισαγωγή στοιχείου στη σωστή θέση

(υποτίθεται η λίστα είναι ταξινομημένη **)**

```
PTR insert_to_list(PTR head, int  x)
{
  PTR current, previous, newnode;
  int found;

  newnode = malloc(sizeof(struct node));
  newnode->data = x;
  newnode->next = NULL;
  if (head = = NULL)
      head = newnode;
  else
     if (newnode->data < head->data)
```

```
        {
          newnode->next = head;
          head = newnode;
        }
      else
        {
          previous = head;
          current = head->next;
          found = 0;
          while (current != NULL && found = = 0)
              {
                 if (newnode->data < current->data)
                     found = 1;
                else
                  {
                     previous = current;
                     current = current->next;
                  }
               }
          previous->next = newnode;
          newnode->next = current;
        }
   return head;
}
```

γ) Διαγραφή στοιχείου από λίστα

```
PTR  delete_from_list(PTR head, int  x)
{
   PTR current, previous;
   int found;
```

```
current = head;
if (current = = NULL)
  {
    printf("Empty list...nothing to delete.\n");
    getch();
  }
else
    if (x = = head->data)
    {
        head = head->next;
        free(current);
    }
    else
       {
         previous = current;
         current = head->next;
         found = 0;
         while (current != NULL && found = = 0)
          {
            if (x = = current->data)
                found = 1;
            else
              {
                previous = current;
                current = current->next;
              }
           }
        if  (found = = 1)
          {
             previous->next = current->next;
```

```
                free(current);
              }
            else
              {
                printf("\nThe character is not in the list.");
                getch();
              }
          }
  return head;
}
```

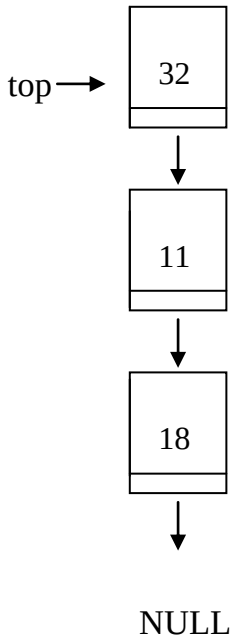
δ) Εκτύπωση λίστας

```
void  print_list(PTR head)
{
  PTR current;

  current = head;
  if (current = = NULL)
     printf("The list is empty.\n");
  else
     while (current != NULL)
          {
            printf("%i ", current->data);
            current = current->next;
          }
}
```

6.3.2 Στοίβα ως Συνδεδεμένη Λίστα

Η στοίβα υλοποιείται με έναν δείκτη top που αρχικοποιείται στο NULL, και δείχνει ότι η στοίβα είναι άδεια. Η λειτουργία push δεν ελέγχει για υπερχείλιση

γιατί θεωρητικά η στοίβα μπορεί να έχει όσους κόμβους θέλουμε (αφού δημιουργείται δυναμικά). Επίσης, η λειτουργία pop, με τη βοήθεια της εντολής free επιστρέφει στη διαθέσιμη μνήμη του Η/Υ το χώρο που καταλαμβάνονταν από τον κόμβο που διαγράφηκε.



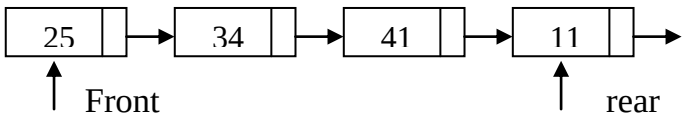
Σχήμα 6.7 *Στοίβα ως Συνδεδεμένη λίστα*

```
PTR top = NULL;

PTR push(int obj, PTR top)
{
  PTR newnode;
  newnode = malloc(sizeof(struct node));
  newnode->data = obj;
  newnode->next = top;
  top = newnode;
  return top;
}

PTR pop(PTR top,int *obj)
```

```
{
  PTR p;

  if (top = = NULL)
    {
      printf("Stack empty.\n");
      getch();
     }
  else
    {
      p = top;
      top = top->next;
     *obj = p->data;
      free(p);
      return top;
    }
}
```

6.3.3 Ουρά ως Συνδεδεμένη Λίστα

Η ουρά υλοποιείται με τη χρήση δύο δεικτών front και rear που αρχικοποιούνται στην τιμή NULL. Η άδεια ουρά εκφράζεται με front = NULL. Όπως και στην περίπτωση της στοίβας, δεν χρειάζεται να γίνεται έλεγχος υπερχείλισης από τη στιγμή που η ουρά αυξάνεται δυναμικά.



Σχήμα 6.8 *Ουρά ως Συνδεδεμένη Λίστα*

```
PTR front = NULL, rear = NULL;
void enqueue(int obj, PTR *pf, PTR *pr)
{
  PTR newnode;
```

```
  newnode = malloc(sizeof(struct node));
  newnode->data = obj;
  newnode->next = NULL;
  if ((*pf) = = NULL)
    {
     *pf = newnode;
     *pr = newnode;
    }
  else
    {
     (*pr)->next = newnode;
      *pr = newnode;
    }
}

void dequeue(PTR *pf, PTR *pr)
{
  PTR p;
  if ((*pf) = = NULL)
      printf("\nQueue empty. No elements to delete.\n");
  else
    {
      p = *pf;
      *pf = (*pf)->next;
      if ((*pf) = = NULL)
           *pr = *pf;
      printf("\n%d has been deleted...\n",p->data);
      free(p);
    }
 getch();
}
```

VII. ΔΕΝΔΡΑ

7.1 Εισαγωγή…… 7-1

7.2 Δυαδικά δένδρα……. 7-3

7.3 B-trees…….. 7-10

7.4 Tries…….. 7-13

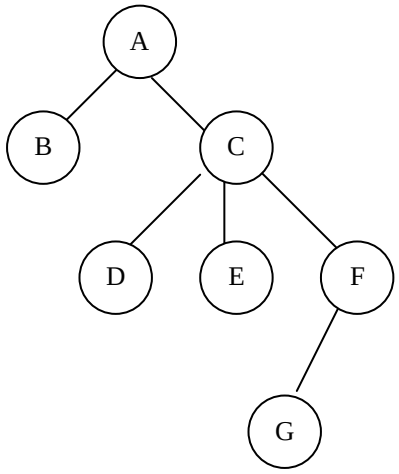
7.1 ΕΙΣΑΓΩΓΗ

Οι δομές που εξετάστηκαν στα προηγούμενα κεφάλαια, είχαν σαν κοινό χαρακτηριστικό τη γραμμικότητα μεταξύ των κόμβων τους. Στο κεφάλαιο αυτό θα εξετάσουμε την πιο ενδιαφέρουσα **μή γραμμική** δομή, το **δένδρο** (tree).

Ορισμός (αναδρομικός)

Ένα δένδρο T είναι ένα πεπερασμένο σύνολο από έναν ή περισσότερους κόμβους. Ο πρώτος κόμβος του δένδρου ονομάζεται **ρίζα** (root), ενώ οι υπόλοιποι κόμβοι απαρτίζουν άλλα υποσύνολα που με τη σειρά τους είναι δένδρα και ονομάζονται υποδένδρα (subtrees).

Ο ορισμός είναι αναδρομικός γιατί μπορεί να θεωρηθεί ότι κάθε κόμβος του δένδρου είναι ρίζα σε κάποιο υποδένδρο.



Σχήμα 7.1 *Δένδρο*

T = (A, B, C, D, E, F, G)

ρίζα A.

T1 = (B)

T2 = (C, D, E, F, G)

Βαθμός του κόμβου (node degree**)**

Είναι ο αριθμός των υποδένδρων που αρχίζουν από ένα κόμβο.

π.χ.

Ο κόμβος A έχει βαθμό 2.

Ο κόμβος C έχει βαθμό 3.

Βαθμός του δένδρου (tree degree**)**

Είναι ο μέγιστος βαθμός από όλους τους βαθμούς κόμβων.

Τα δένδρα που χρησιμοποιούνται περισσότερο στην πράξη είναι τα **δυαδικά δένδρα** (binary trees), με βαθμό 2.

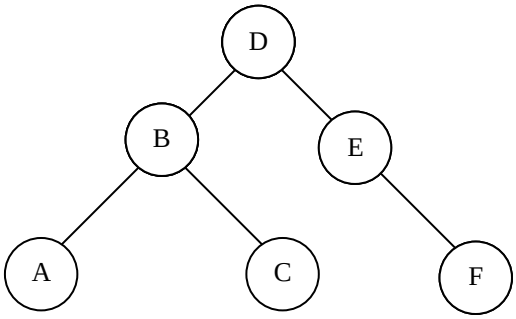
Οι κόμβοι ενός δένδρου από τους οποίους δεν αρχίζει κάποιο υποδένδρο λέγονται **φύλλα** (leaves) ή **τερματικοί** κόμβοι (terminal nodes). Όλοι οι άλλοι κόμβοι λέγονται **κλαδιά**(branches).

Η ρίζα ενός δένδρου ονομάζεται **πατέρας** (father) των ριζών των υποδένδρων και οι ρίζες των υποδένδρων ονομάζονται **παιδιά** (children) της ρίζας. Οι κόμβοι που έχουν τον ίδιο πατέρα ονομάζονται **αδελφοί** (brothers).

Όταν έχει σημασία η διάταξη των κλαδιών ενός δένδρου, τότε αυτό λέγεται **διατεταγμένο**(ordered).

Μία ενδιαφέρουσα μορφή διατεταγμένου δένδρου είναι το **δυαδικό δένδρο αναζήτησης** (binary search tree). Ένα δυαδικό δένδρο αναζήτησης είναι οργανωμένο έτσι ώστε για κάθε κόμβο **t**, όλα τα κλειδιά του αριστερού υποδένδρου να έχουν τιμή μικρότερη από την τιμή του κόμβου **t** και όλα τα κλειδιά του δεξιού υποδένδρου να έχουν τιμή μεγαλύτερη από την αντίστοιχη του κόμβου **t.**

π.χ.



Σχήμα 7.2 *Δυαδικό Δένδρο Αναζήτησης*

Ένας κόμβος **y** που βρίσκεται σε κατώτερο επίπεδο από ένα κόμβο **x** λέγεται **απόγονος** (descendant) του κόμβου **x**. Αντίστροφα, ο κόμβος **x** ονομάζεται **πρόγονος** (ancestor) του κόμβου **y.**

Η ρίζα του δένδρου βρίσκεται πάντοτε στο επίπεδο 1. Το μέγιστο επίπεδο όλων των κόμβων ενός δένδρου λέγεται **βάθος**(depth) ή **ύψος** (height) του δένδρου. **Δάσος** (forest) είναι ένα σύνολο από περισσότερα του ενός δένδρα που είναι ξένα μεταξύ τους.

7.2 ΔΥΑΔΙΚΑ ΔΕΝΔΡΑ (Binary trees)

Ορισμός

Ένα δυαδικό δένδρο αποτελείται από ένα πεπερασμένο σύνολο κόμβων. Το δένδρο είναι είτε άδειο, είτε αποτελείται από δύο άλλα δυαδικά δένδρα που ονομάζονται αριστερό και δεξιό υποδένδρο.

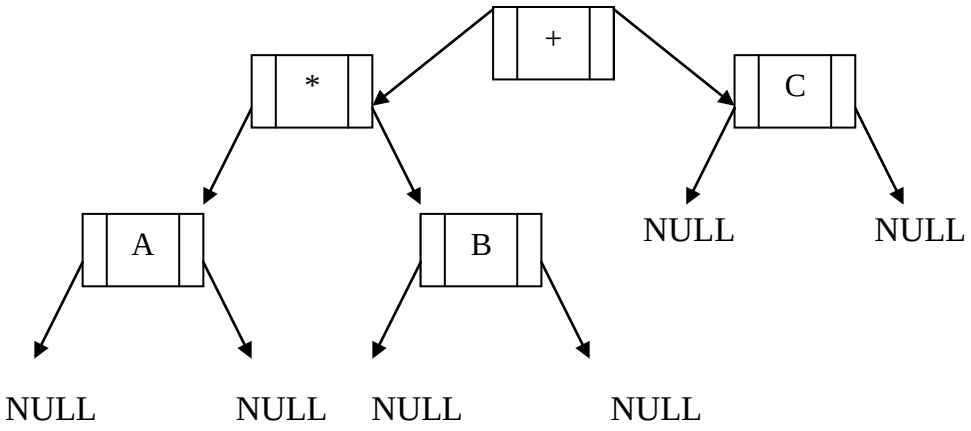
Ορισμός στην C:

```
struct  treenode
{
   char data;
   struct treenode * left;
   struct treenode * right;
};
typedef  struct treenode * PTR;
```

Μία συνηθισμένη εφαρμογή των δυαδικών δένδρων είναι η παράσταση αριθμητικών εκφράσεων στον Η/Υ.

π.χ.

Η αλγεβρική έκφραση A*B+C μπορεί να παρασταθεί ως



Σχήμα 7.3 *Η Αλγεβρική έκφραση A*B+C*

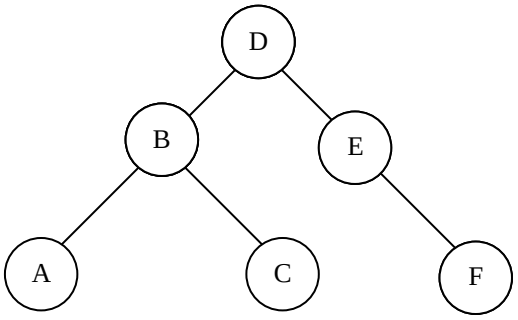
Μία σπουδαία λειτουργία σε ένα δένδρο είναι η διάσχισή του (traversal) ή διέλευση, η επίσκεψη δηλ. όλων των κόμβων του μία φορά. Υπάρχουν τρεις διαφορετικές μέθοδοι διάσχισης:

α) **Προδιατεταγμένη** μέθοδος (preorder traversal)

β) **Ενδοδιατεταγμένη** μέθοδος (inorder traversal)

γ) **Μεταδιατεταγμένη** μέθοδος (postorder traversal)

π.χ.



Σχήμα 7.4 *Διάσχιση δυαδικού δένδρου*

preorder

1. Επίσκεψη της ρίζας
2. Επίσκεψη του αριστερού υποδένδρου
3. Επίσκεψη του δεξιού υποδένδρου

(Ο αλγόριθμος είναι αναδρομικός)

Αποτελέσματα:

D, B, A, C, E, F

Λεπτομερής ανάλυση:

1. Ρίζα (**D**)[1]
2. Αριστερά (**B, A, C**) => 1. Ρίζα (**B**)[2]
3. Δεξιά (**E, F**) 2. Αριστερά (**A**) => 1. Ρίζα (**A**)[3]
   || 3. Δεξιά (**C**) 2. Αριστερά (**άδειο**)
   1. Ρίζα (**E**)[5] || 3. Δεξιά (**άδειο**)
   2. Αριστερά (**άδειο**) 1. Ρίζα (**C**)[4]

3. Δεξιά (**F**) 2. Αριστερά (**άδειο**)

|| 3. Δεξιά (**άδειο**)

1. Ρίζα (**F**)[6]

2. Αριστερά (**άδειο**)

3. Δεξιά (**άδειο**)

inorder

1. Επίσκεψη του αριστερού υποδένδρου
2. Επίσκεψη της ρίζας
3. Επίσκεψη του δεξιού υποδένδρου

(Ο αλγόριθμος είναι αναδρομικός)

Αποτελέσματα:

A, B, C, D, E, F

postorder

1. Επίσκεψη του αριστερού υποδένδρου
2. Επίσκεψη του δεξιού υποδένδρου
3. Επίσκεψη της ρίζας

(Ο αλγόριθμος είναι αναδρομικός)

Αποτελέσματα:

A, C, B, F, E, D

Υλοποίηση στην C:

```
void preorder_traversal(PTR  t)
{
  if (t!=NULL)
     {
       printf("%c ",t->data);
       preorder_traversal(t->left);
       preorder_traversal(t->right);
     }
}
```

```
void inorder_traversal(PTR  t)
{
  if (t!=NULL)
    {
      inorder_traversal(t->left);
      printf("%c ",t->data);
      inorder_traversal(t->right);
    }
}

void postorder_traversal(PTR  t)
{
  if (t!=NULL)
    {
      postorder_traversal(t->left);
      postorder_traversal(t->right);
      printf("%c “,t->data);
    }
}
```

Εισαγωγή κόμβου σε Δυαδικό Δένδρο Αναζήτησης:

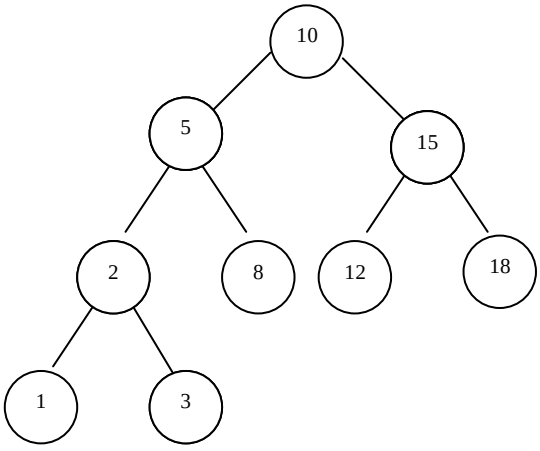
```
void insert_node(PTR *pt, char x)
{
  PTR t;

  t = *pt;
  if (t= =NULL)
    {
      t = malloc(sizeof (struct treenode));
      t->data = x;
      t->left = NULL;
```

```
        t->right = NULL;
    }
  else
    if (x<t->data)
        insert_node(&(t->left),x);
    else
        insert_node(&(t->right),x);
  *pt = t;
}
```

Διαγραφή κόμβου από Δυαδικό Δένδρο Αναζήτησης:

Η διαδικασία της διαγραφής είναι πιο σύνθετη από τη διαδικασία της εισαγωγής. Αν ο κόμβος είναι τερματικός, τότε είναι εύκολο. Επίσης εύκολη είναι η περίπτωση που ο διαγραφόμενος κόμβος έχει μόνον ένα απόγονο.

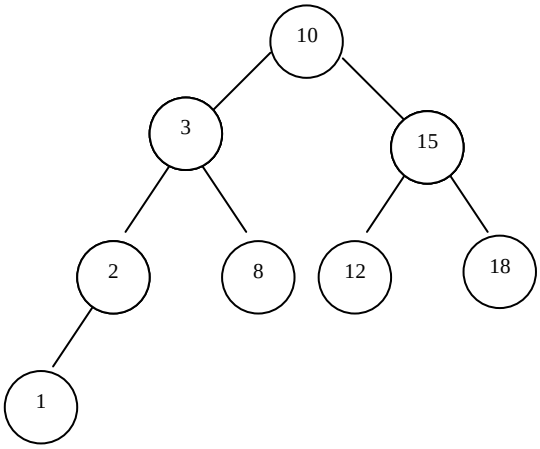
Δύσκολη είναι η περίπτωση που ο κόμβος έχει δύο απογόνους. Σ' αυτή την περίπτωση ο κόμβος πρέπει να αντικατασταθεί είτε από τον πιο δεξιό κόμβο του αριστερού υποδένδρου, ή από τον πιο αριστερό κόμβο του δεξιού υποδένδρου. Προφανώς οι δύο αυτοί υποψήφιοι έχουν το πολύ έναν απόγονο. π.χ.



Διαγραφή του κόμβου 5.

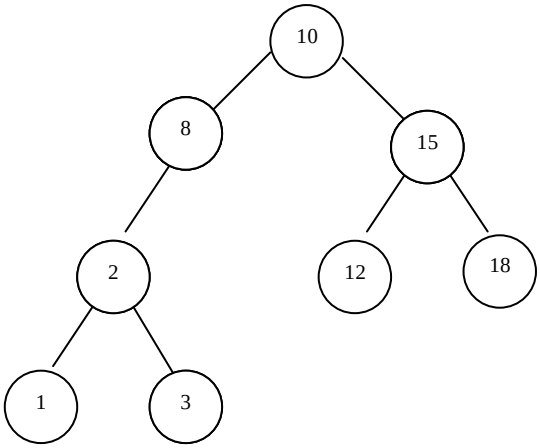
1ος τρόπος

Θα αντικατασταθεί με τον πιό δεξιό κόμβο του αριστερού υποδένδρου, δηλ. το 3.



2ος τρόπος

Θα αντικατασταθεί με τον πιό αριστερό κόμβο του δεξιού υποδένδρου, δηλ. το 8.

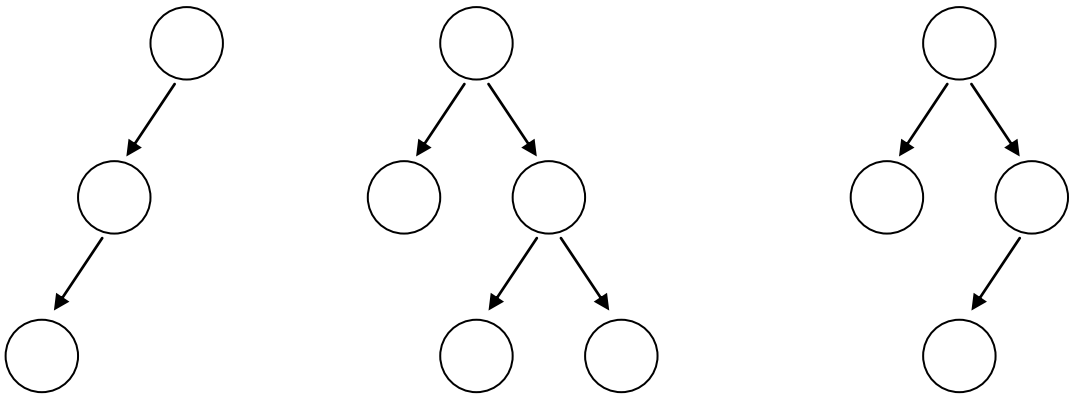


Ισορροπημένα δένδρα

Ένα δυναμικό δυαδικό δένδρο λόγω διαδοχικών εισαγωγών και διαγραφών, μπορεί, στην ακραία περίπτωση, να εκφυλισθεί σε μία γραμμική λίστα. Τότε ο χρόνος επεξεργασίας από O(logn) καταλήγει σε O(n). Συνεπώς, πρέπει το δένδρο μετά από κάθε λειτουργία να παραμένει όσο το δυνατόν πιο ισορροπημένο και μάλιστα η διαδικασία εξισορρόπησης να μην είναι δαπανηρή σε υπολογιστικούς πόρους.

Ένα δυαδικό δένδρο λέγεται **ισορροπημένο κατά ύψος** (height balanced tree) αν το ύψος του αριστερού και του δεξιού υποδένδρου κάθε κόμβου διαφέρει το πολύ κατά ένα.

Ένα δυαδικό δένδρο λέγεται **τέλεια ισορροπημένο** (perfectly balanced tree) αν το πλήθος των κόμβων του αριστερού και του δεξιού υποδένδρου κάθε κόμβου διαφέρει το πολύ κατά ένα.



α) Εκφυλισμένο δένδρο β) Ισορροπημένο κατά ύψος δένδρο γ)Τέλεια ισορροπημένο δένδρο

Σχήμα 7.5 *Ισορροπημένα Δένδρα*

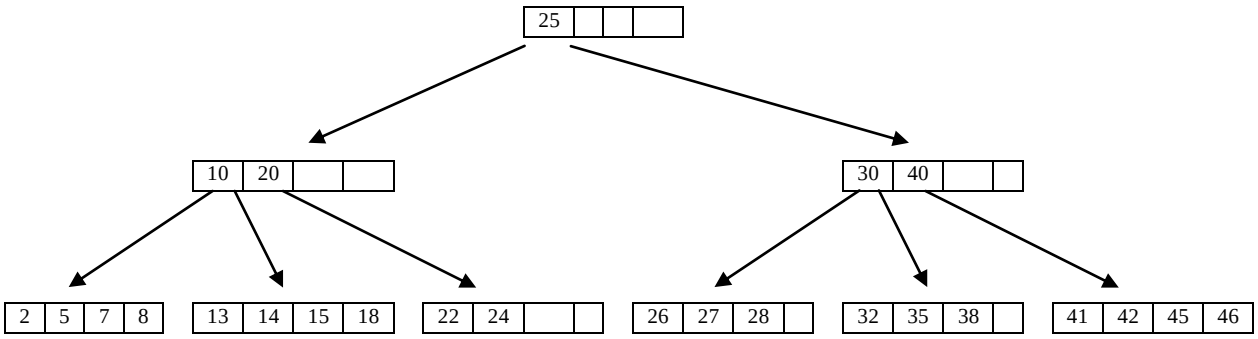
7.3 B-trees

Το όνομα μάλλον προέρχεται από τον όρο balanced tree (ισορροπημένο δένδρο) και όχι από το binary tree.

B-tree τάξης (ή βαθμού) **n** είναι το δένδρο με τα εξής χαρακτηριστικά:

- Η ρίζα έχει το λιγότερο ένα κλειδί και το περισσότερο 2n κλειδιά.
- Κάθε εσωτερικός κόμβος (εκτός της ρίζας) έχει το λιγότερο n κλειδιά και το περισσότερο 2n κλειδιά.
- Ένας κόμβος με m (1<= m <= 2n) κλειδιά έχει m+1 παιδιά.
- Όλα τα φύλλα βρίσκονται στο ίδιο επίπεδο.

π.χ.



B-tree τάξης 2 με 3 επίπεδα(n=2).

Η ρίζα έχει ένα κλειδί.

Όλα τα φύλλα βρίσκονται στο επίπεδο 3.

Κάθε εσωτερικός κόμβος έχει από 2 μέχρι 4 κλειδιά.

Οι κόμβοι (10,20) και (30,40) με 2 κλειδιά έχουν από τρία παιδιά.

Εισαγωγή σε B-tree

Γίνεται πάντα σε έναν κόμβο που είναι φύλλο.

- Αν ο κόμβος που θα δεχθεί το κλειδί έχει λιγότερα από 2n κλειδιά, η εισαγωγή είναι εύκολη.
- Αν ο κόμβος έχει 2n κλειδιά, η εισαγωγή είναι αδύνατη γιατί δημιουργείται υπερχείλιση. Τότε ο κόμβος διασπάται σε δύο άλλους κόμβους. Το μεσαίο

κλειδί ανεβαίνει στον κόμβο πρόγονο των δύο διασπασθέντων και τα υπόλοιπα κλειδιά διαμοιράζονται σ' αυτούς τους δύο κόμβους.

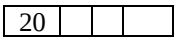
- Μία διάσπαση μπορεί να έχει ως αποτέλεσμα τη διάσπαση ενός κόμβου που βρίσκεται στο πιο πάνω επίπεδο κ.ο.κ.

  Στην πιο ακραία περίπτωση υπάρχει το ενδεχόμενο διάσπασης της ρίζας, οπότε το ύψος του δένδρου αυξάνει κατά ένα.

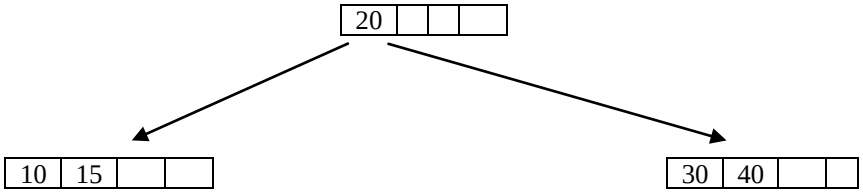
Χαρακτηριστικό των B-trees είναι ότι μεγαλώνουν από κάτω προς τα πάνω και όχι από πάνω προς τα κάτω όπως τα Δυαδικά δένδρα αναζήτησης.

Παράδειγμα

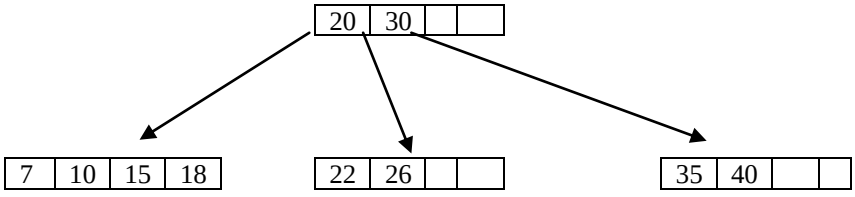
1. Εισαγωγή **20**



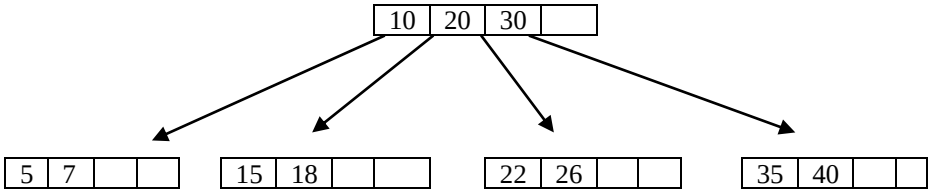
2. Εισαγωγή **40, 10, 30, 15**



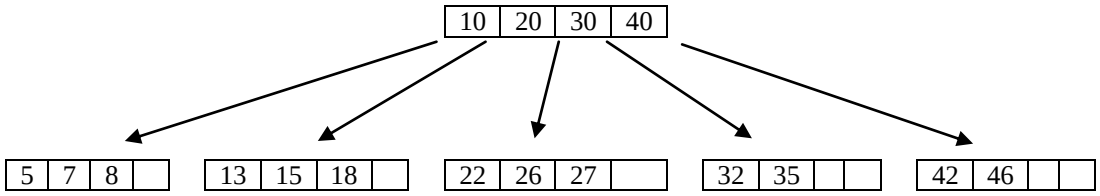
3. Εισαγωγή **35, 7, 26, 18, 22**



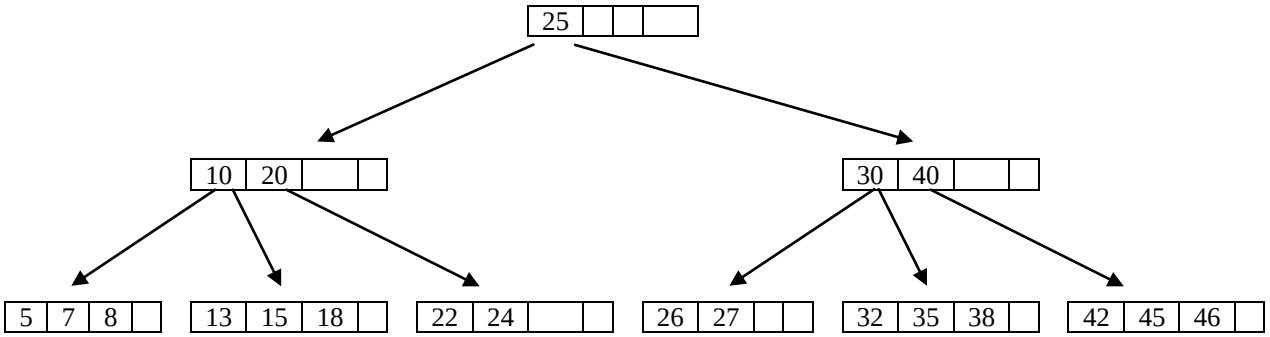
4. Εισαγωγή **5**



5. Εισαγωγή **42, 13, 46, 27, 8, 32**



6. Εισαγωγή **38, 24, 45, 25**



7.4 TRIES

Στα δένδρα που εξετάστηκαν μέχρι τώρα, κάθε κόμβος περιείχε ένα κλειδί (ή περισσότερα). Αυτή η αρχή οργάνωσης απαιτεί συνήθως ειδικά μέτρα για να προστατευθούν αυτά τα δένδρα από το να χάσουν την ισορροπία τους ή ακόμα και να εκφυλιστούν. Είναι προφανές, ότι αν θέλουμε να ψάξουμε ένα δένδρο αποτελεσματικά, θα πρέπει οι κόμβοι του να περιέχουν συγκεκριμένες τιμές, που να μας δίνουν τη δυνατότητα να αποφασίσουμε ποια διαδρομή θα ακολουθήσουμε. Μέχρι τώρα πήραμε σαν δεδομένο ότι αυτές οι τιμές πρέπει να είναι υπάρχοντα κλειδιά. Ωστόσο, αυτό δεν είναι απόλυτα απαραίτητο. Αντί να χρησιμοποιούμε σε κάθε σύγκριση ολόκληρο το κλειδί, μπορούμε να συγκρίνουμε μόνο ένα ορισμένο τμήμα του.

Αυτή η ιδέα είναι η βάση για ένα ειδικό τύπο δένδρου, που ονομάζεται trie. Η ονομασία προέρχεται από τις λέξεις tree (δένδρο) και retrieval (ανάκτηση). Για καλύτερη κατανόηση ενός trie, ας εξετάσουμε ένα παράδειγμα:

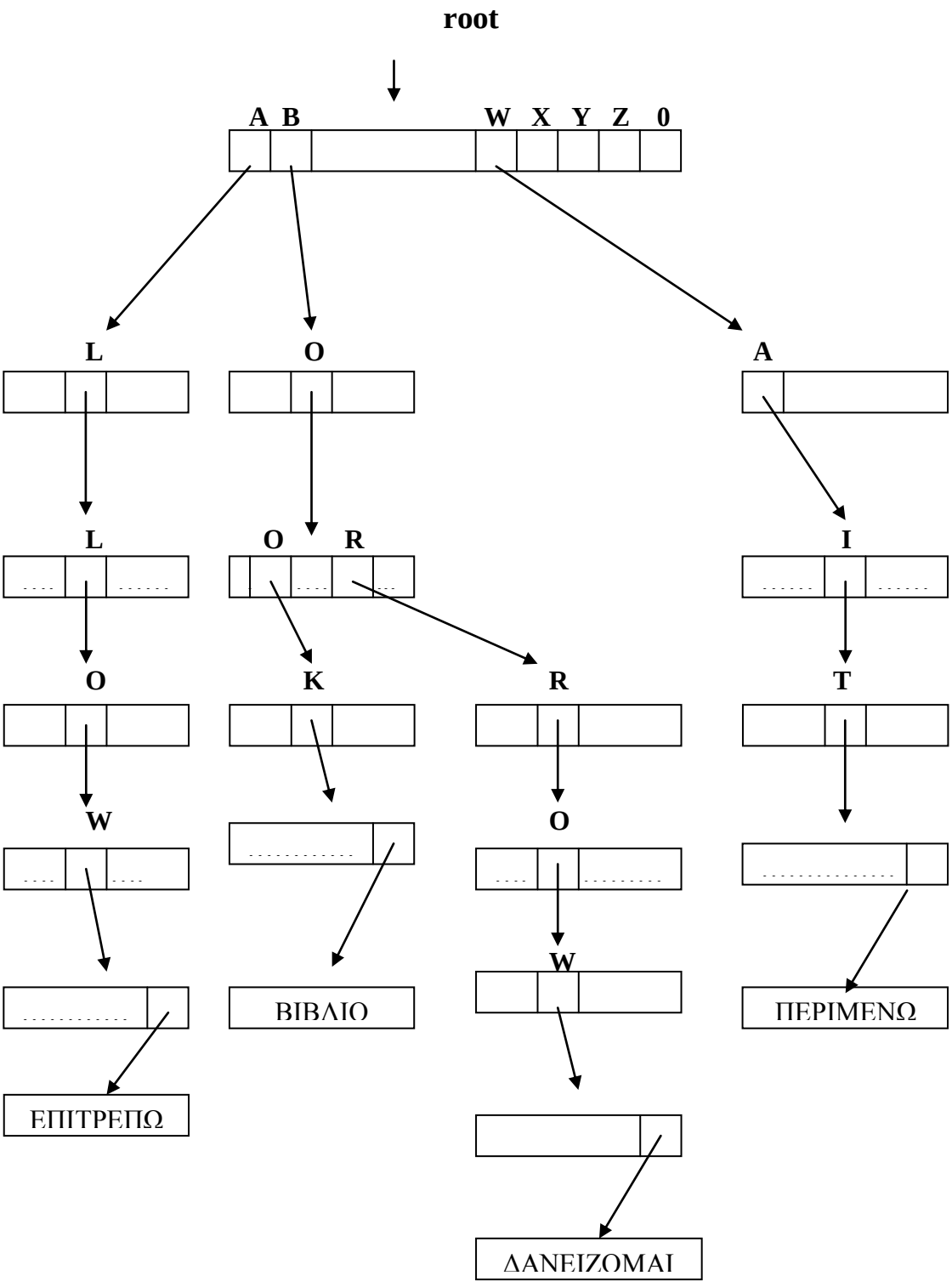
Τα κλειδιά μας θα είναι σειρές χαρακτήρων που αποτελούνται μόνο από κεφαλαία αγγλικά γράμματα.

Π.χ.

ALLOW
BOOK
BORROW
WAIT

Για να αποφασίσουμε πως θα γίνει η διακλάδωση, χρησιμοποιούμε τον κάθε μεμονωμένο χαρακτήρα της σειράς. Όλοι οι εσωτερικοί κόμβοι, που ονομάζονται κόμβοι διακλάδωσης (branch nodes), θα έχουν 27 πεδία δεικτών. Τα φύλλα, που ονομάζονται κόμβοι πληροφοριών (information nodes), θα αποτελούνται από τις πλήρεις εγγραφές πληροφοριών. Στους κόμβους διακλάδωσης υπάρχει ένα πεδίο δείκτη για κάθε ένα από τα γράμματα A,…..,Z. Επίσης, σε κάθε κόμβο διακλάδωσης διατηρούμε ένα πεδίο δείκτη που αντιστοιχεί στο μηδενικό χαρακτήρα (που συμβολίζει το τέλος μιας λέξης).

Αν για παράδειγμα ψάξουμε στο trie για τη λέξη BOOK, διαλέγουμε πρώτα το δείκτη που αντιστοιχεί στο γράμμα Β στο ριζικό κόμβο. Αυτός ο δείκτης δείχνει σε ένα νέο κόμβο διακλάδωσης. Εδώ διαλέγουμε το δείκτη που αντιστοιχεί στο γράμμα Ο. Και αυτός ο δείκτης με τη σειρά του δείχνει σε ένα νέο κόμβο διακλάδωσης. Απ' εδώ διαλέγουμε το δείκτη που αντιστοιχεί στο γράμμα Ο ξανά και κατευθυνόμαστε σε νέο κόμβο διακλάδωσης, όπου υπάρχει ο δείκτης που αντιστοιχεί στο Κ. Εδώ, μέσω του δείκτη που αντιστοιχεί στο μηδενικό χαρακτήρα προσπελάζουμε τον κόμβο πληροφοριών.



Σχήμα 7.6 *Trie*

Ένα πρόγραμμα επίδειξης

Το παρακάτω πρόγραμμα διαβάζει αγγλικές λέξεις (μαζί με τη μετάφρασή τους) από ένα αρχείο κειμένου και χτίζει ένα trie με κλειδιά τις αγγλικές λέξεις. Ύστερα τυπώνει τα περιεχόμενα του trie με μία αναδρομική συνάρτηση. Να τονίσουμε ότι, άσχετα με ποια σειρά διαβάζονται τα δεδομένα, η εμφάνισή τους θα είναι ταξινομημένη και αυτό βασίζεται στο γεγονός ότι οι δείκτες είναι διατεταγμένοι στους κόμβους διακλάδωσης σε αλφαβητική σειρά. Τέλος, υπάρχει μία συνάρτηση αναζήτησης, όπου ο χρήστης προτρέπεται να εισάγει μία αγγλική λέξη και αυτή αναζητείται στο trie, όπου αν υπάρχει τυπώνεται η μετάφρασή της.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

struct branchNode
{
  struct branchNode *p[26];
  char *infp;
};
typedef struct branchNode * brPTR;

brPTR newBrNode(void);
brPTR addWord(char *w1, char *w2, brPTR t);
void searchWord(char *w, brPTR t);
void printTrie(brPTR t);

char engWord[10][80];
char grWord[10][80];
int n = 0;
```

```
main()
{
  FILE *fp;
  brPTR tr = NULL;
  char w[80];
  int i;

  fp = fopen("lexiko.txt","r");
  for (i=0; i<10; i++)
   {
     fscanf(fp,"%s",engWord[i]);
     fscanf(fp,"%s",grWord[i]);
     tr = addWord(engWord[i], grWord[i], tr);
   }
  fclose(fp);

  printTrie(tr);
  printf("\n");
  printf("\nGive english word, TELOS to stop: ");
  scanf("%s",w);
  while (strcmp(w,"TELOS")!= 0)
     {
       searchWord(w,tr);
       printf("\nGive english word, TELOS to stop: ");
       scanf("%s",w);
     }

  return 0;
}
```

```
brPTR newBrNode()
{
  brPTR current;
  int i;

  current = malloc(sizeof(struct branchNode));
  for (i=0; i<26; i++)
      current->p[i] = NULL;
  current->infp = NULL;

 return current;
}

brPTR addWord(char *w1, char *w2, brPTR t)
{
  brPTR current;
  int i;

  if (t == NULL)
     t = newBrNode();
  current = t;
  while (isalpha(*w1))
    {
      i = *w1-'A';
      if (current->p[i] == NULL)
         current->p[i] = newBrNode();
      current = current->p[i];
      w1++;
    }
  current->infp = w2;
```

```
  return t;
}


void searchWord(char *w, brPTR t)
{
  int i, flag = 1;

  while (*w != '\0' && flag == 1)
  {     i = *w-'A';
    if (t->p[i] != NULL)
       t = t->p[i];
    else         flag = 0;
   w++;
  }
  printf("                Search result = ");
  if (t->infp != NULL)
     printf("%s\n",t->infp);
  else
     printf("Word not found!\n");
}

void printTrie(brPTR t)
{
  int i;

  if (t->infp != NULL)
    {
     printf("%s = ",engWord[n++]);
     printf("%s\n",t->infp);
    }
```

```
  for (i=0; i<26; i++)
      if (t->p[i] != NULL)
         printTrie(t->p[i]);
}
```

Θα πρέπει να τονίσουμε ότι τα tries δεν είναι οικονομικά με τη μνήμη. Εκτός από τους κόμβους πληροφοριών, υπάρχουν και οι κόμβοι διακλάδωσης που χρησιμοποιούν αρκετό χώρο μνήμης. Όμως, όσον αφορά το χρόνο, ο τύπος του trie που εξετάστηκε είναι πολύ αποτελεσματικός.

Τα tries είναι δομές δεδομένων κατάλληλες για προγράμματα ελέγχου ορθογραφίας. Σ' αυτή την περίπτωση οι κόμβοι πληροφοριών θα μπορούσαν να περιέχουν τις λέξεις ενός λεξικού.

Υπάρχει ακόμα μία πλευρά των tries που χρειάζεται την προσοχή μας και συγκεκριμένα το ότι ένα trie είναι ένα πολύ κατάλληλο μέσο για την αναζήτηση όλων των λέξεων με ένα δοσμένο πρόθεμα. Μπορεί, για παράδειγμα, να εισάγουμε τη λέξη COM και να πάρουμε τον κατάλογο:

COMMAND, COMPILER, COMPUTE, COMPUTER, COMPUTING

Σαν τελική παρατήρηση, να αναφέρουμε ότι μπορούμε να υλοποιήσουμε ένα trie και στο δίσκο αντί για την κεντρική μνήμη, όπως και με τα B-trees. Αυτό θα καθυστερήσει αρκετά τη διαδικασία της αναζήτησης αλλά, με την προϋπόθεση ότι η ταχύτητα είναι αρκετή για την εφαρμογή μας, μπορούμε με αυτή τη μέθοδο να επωφεληθούμε από τη μεγάλη χωρητικότητα των δίσκων. Άλλο ένα ευνοϊκό σημείο είναι η μονιμότητα των αρχείων σε δίσκους σε σύγκριση με την αστάθεια των tries στην κεντρική μνήμη.

VIII. ΓΡΑΦΟΙ

8.1 Εισαγωγή..... 8-1

8.2 Μέθοδοι Αναπαράστασης γράφων.. 8-4

8.3 Μέθοδοι Διάσχισης γράφων..... 8-6

8.3.1 Αναζήτηση με προτεραιότητα Βάθους 8-6

8.3.2 Αναζήτηση με προτεραιότητα Πλάτους 8-8

8.4 Το πρόβλημα του συντομότερου μονοπατιού..... 8-10

8.1 ΕΙΣΑΓΩΓΗ

Είναι η πιο γενική μορφή δομής δεδομένων. Θα μπορούσαμε να θεωρήσουμε τις δομές που εξετάστηκαν ως τώρα ως υποπεριπτώσεις των γράφων.

Η θεωρία των γράφων (graph theory) αποτελεί σπουδαίο αντικείμενο μελέτης και έρευνας και συνήθως διδάσκεται ως ανεξάρτητο μάθημα σε αρκετές επιστήμες (Μαθηματικά, Πληροφορική, Φυσική, κλπ.).

Στην Επιστήμη των Υπολογιστών οι γράφοι χρησιμοποιούνται για την μελέτη των Γλωσσών Προγραμματισμού, Λειτουργικών Συστημάτων, Βάσεων Δεδομένων, Δικτύων, κλπ.

Επίσης, οι γράφοι χρησιμοποιούνται για την επίλυση πρακτικών προβλημάτων ποικίλων επιστημονικών περιοχών όπως Ανθρωπολογία, Γεωγραφία, Οικονομικά, κλπ.

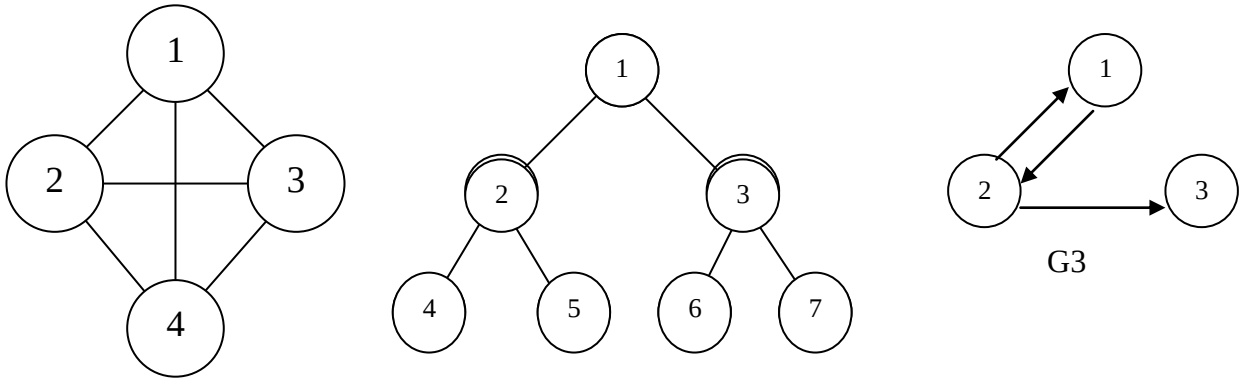
Στο κεφάλαιο αυτό θα παρουσιασθούν κάποιες ενδιαφέρουσες τεχνικές για την υλοποίηση και την επεξεργασία της δομής του γράφου.

Ορισμοί

Ένας γράφος **G** χαρακτηρίζεται από δύο σύνολα **V** και **E**. Το σύνολο V είναι ένα πεπερασμένο διάφορο του κενού σύνολο που περιέχει ως στοιχεία τις **κορυφές** (vertices) του γράφου. Το σύνολο E έχει ως στοιχεία τα ζευγάρια κορυφών του γράφου που ορίζουν τις **ακμές** (edges).

Τα σύμβολα **V(G)**, **E(G)** και **G(V,E)** χρησιμοποιούνται αντίστοιχα για την αναπαράσταση των συνόλων V, E και του γράφου G.

Π.χ.



G1 G2 G3

Σχήμα 8.1 *Γράφοι*

Ένας γράφος ονομάζεται **μη-κατευθυνόμενος** (undirected) όταν τα ζευγάρια των κορυφών που ορίζουν τις ακμές δεν είναι διατεταγμένα. Δηλαδή, τα ζευγάρια (v_1,v_2) και (v_2,v_1) ορίζουν την ίδια ακμή. Οι γράφοι G1, G2 του προηγούμενου σχήματος είναι μη-κατευθυνόμενοι.

Ένας γράφος ονομάζεται **κατευθυνόμενος** (directed graph) όταν κάθε ακμή του ορίζεται από ένα διατεταγμένο ζευγάρι κορυφών. Συμβολίζουμε το ζευγάρι αυτό $<v_1,v_2>$ και θεωρούμε την κατεύθυνση από την κορυφή v_1, που ονομάζεται ουρά (tail), προς την v_2, που ονομάζεται κεφαλή (head). Ο γράφος G_3 του σχήματος είναι κατευθυνόμενος.

Τα σύνολα V των γράφων G_1, G_2, G_3 είναι:

$V(G_1) = (1,2,3,4)$

$V(G_2) = (1,2,3,4,5,6,7)$

$V(G_3) = (1,2,3)$

Τα σύνολα E ορίζονται ως εξής:

$E(G_1) = ((1,2), (1,3), (1,4), (2,3), (2,4), (3,4))$

$E(G_2) = ((1,2), (1,3), (2,4), (2,5), (3,6), (3,7))$

$E(G_3) = (<1,2>, <2,1>, <2,3>)$

Ένας μη-κατευθυνόμενος γράφος με **n** κορυφές λέγεται **πλήρης** (complete) αν έχει ακριβώς $\frac{n(n-1)}{2}$ ακμές.

π.χ

ο γράφος G_1 του σχήματος είναι πλήρης.

Αν (v_1,v_2) είναι μία ακμή του συνόλου E(G), τότε οι κορυφές v_1 και v_2 λέγονται **διπλανές** (adjacent). Η ακμή (v_1,v_2) ονομάζεται **στιγμιότυπο** (incident) των κορυφών v_1 και v_2 και αν δύο κορυφές v_1 και v_2 δεν συνδέονται μεταξύ τους ονομάζονται **ανεξάρτητες** (independent).

Μονοπάτι (path) από μία κορυφή v_m προς μία άλλη κορυφή v_n ορίζεται η λίστα των διαδοχικών κορυφών $[v_m,.....,v_n]$ που συνδέονται με ακμές που ανήκουν στο E(G).

Για παράδειγμα, το [1,2,3,4] είναι ένα μονοπάτι από την κορυφή 1 στην κορυφή 4 στο γράφο G_1 του σχήματος.

Μήκος (length) μονοπατιού είναι ο αριθμός των ακμών που υπάρχουν στο μονοπάτι. Στο προηγούμενο παράδειγμα το μήκος του μονοπατιού είναι 3.

Ένα μονοπάτι ονομάζεται **απλό** (simple) όταν καμία κορυφή δεν εμφανίζεται παραπάνω από μία φορά σ' αυτό.

Π.χ.

Το [1,2,3,4] είναι απλό μονοπάτι, ενώ το [1,2,4,2] δεν είναι. Το [1,2,3,2] δεν είναι μονοπάτι του G_3 γιατί η ακμή <3,2> δεν ανήκει στο $E(G_3)$.

Κύκλος (cycle) είναι ένα απλό μονοπάτι όπου ταυτίζεται η πρώτη και η τελευταία κορυφή.

Π.χ.

το μονοπάτι [1,2,3,1] του G_1 είναι κύκλος.

Απ΄ αυτό συμπεραίνουμε ότι ένα δένδρο είναι ένας γράφος που δεν έχει καθόλου κύκλους.

Βαθμός (degree) κορυφής λέγεται ο αριθμός των ακμών που είναι στιγμιότυπα της κορυφής.

8.2 ΜΕΘΟΔΟΙ ΑΝΑΠΑΡΑΣΤΑΣΗΣ ΓΡΑΦΩΝ

Υπάρχουν αρκετοί τρόποι απεικόνισης ενός γράφου στην κύρια μνήμη του Η/Υ. Η αποδοτικότητα κάθε αναπαράστασης εξαρτάται από τις συγκεκριμένες λειτουργίες που εκτελούνται στο γράφο. Στη συνέχεια θα περιγραφούν δύο από τις βασικότερες μέθοδοι αναπαράστασης. Η πρώτη μέθοδος αφορά στην αναπαράσταση του γράφου με τη χρήση μιας στατικής δομής δεδομένων, ενώ η δεύτερη με τη χρήση μιας δυναμικής δομής.

Πίνακες διπλανών κορυφών (adjacency matrices)

Λίστες διπλανών κορυφών (adjacency lists)

<u>Αναπαράσταση γράφου με Πίνακα Διπλανών Κορυφών</u>

Ένας γράφος **G** με **n** κορυφές μπορεί να αναπαρασταθεί με τη βοήθεια ενός δισδιάστατου πίνακα N*N. Το στοιχείο (i, j) του πίνακα παίρνει την τιμή 1 αν η ακμή (v_i, v_j) ανήκει στο E(G), αλλιώς παίρνει την τιμή 0. Για παράδειγμα, οι πίνακες διπλανών κορυφών για τους γράφους G1, G3 του προαναφερθέντος σχήματος είναι:

0	**1**	**1**	**1**
1	**0**	**1**	**1**
1	**1**	**0**	**1**
1	**1**	**1**	**0**

G_1

0	**1**	**0**
1	**0**	**1**
0	**0**	**0**

G_3

Υλοποίηση στην C:

```
int   graph[N][N];
```

Παρατηρήσεις:

Οι n κορυφές ενός γράφου πιθανώς να αναφέρονται με γράμματα a, b, c, κλπ. ή αριθμούς 1, 2, 3, κλπ. Σ' αυτή την περίπτωση πρέπει να ορίσουμε αντιστοιχία των ονομάτων αυτών με τους δείκτες 0, 1, 2, ..., n-1 του πίνακα.

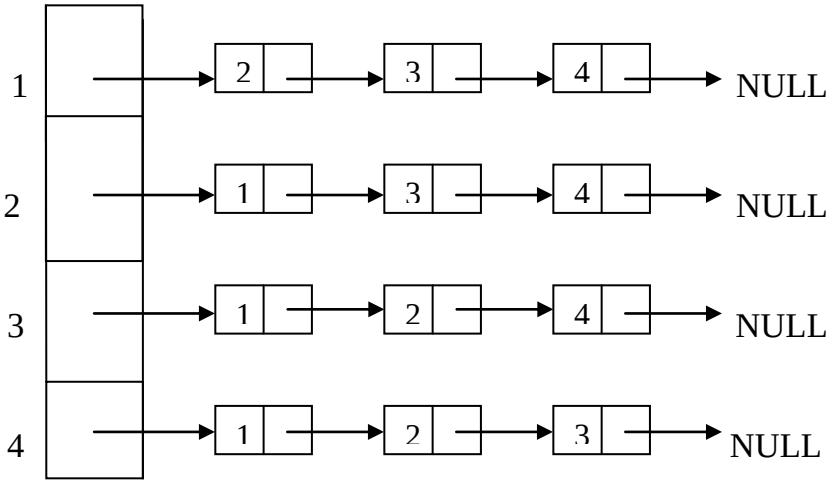
Βασικό πλεονέκτημα της αναπαράστασης ενός γράφου με τη βοήθεια πίνακα διπλανών κορυφών αποτελεί η ταχύτατη διαπίστωση αν δύο κορυφές αποτελούν ακμή ή όχι.

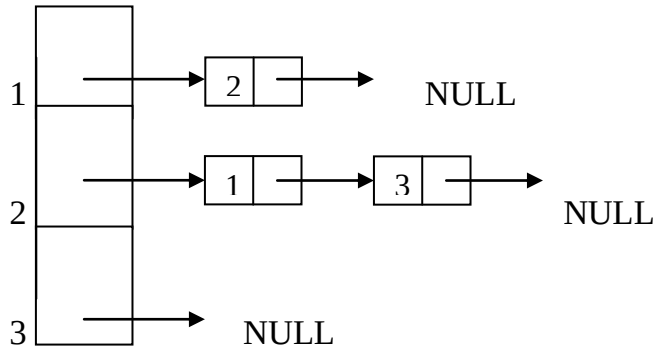
Βασικό μειονέκτημα της χρήσης του πίνακα είναι η σπατάλη μνήμης, καθώς στην ουσία χρειαζόμαστε μόνον τις θέσεις του πίνακα που δηλώνουν ακμές (με τιμή 1) και όχι αυτές που δεν δηλώνουν ακμές (τιμή 0).

Αναπαράσταση γράφου με Λίστες Διπλανών Κορυφών

Σ' αυτή την αναπαράσταση του γράφου που αποτελείται από n κορυφές, ορίζεται ένας πίνακας n θέσεων τα στοιχεία του οποίου είναι συνδεδεμένες λίστες. Κάθε κόμβος της λίστας στη θέση i του πίνακα αναπαριστά κορυφή του γράφου, η οποία συνδέεται με την κορυφή i.

π.χ.





Υλοποίηση στην C:

```
struct  graphnode
{
   int  vertex;
   struct  graphnode * next ;
}
typedef  struct graphnode * PTR;
```

```
main()
{
   PTR graph[n];
```

8.3 ΜΕΘΟΔΟΙ ΔΙΑΣΧΙΣΗΣ ΓΡΑΦΩΝ

Τα προβλήματα που συνήθως αντιμετωπίζουμε στους γράφους, καταλήγουν σε αλγορίθμους που σχετίζονται με την εύρεση ενός μονοπατιού, το οποίο να συνδέει μία αρχική κορυφή (κόμβος εκκίνησης) με μία τελική κορυφή (κόμβος άφιξης). Το μονοπάτι αυτό, ανάλογα με τη φύση του προβλήματος, απαιτούμε να έχει διάφορες ιδιότητες. Έτσι, για παράδειγμα , σ' ένα γράφο που αναπαριστά πόλεις, οι οποίες συνδέονται οδικά μεταξύ τους, μπορεί να αναζητούμε το μονοπάτι εκείνο που περνά από όσο το δυνατόν περισσότερες πόλεις ή να αναζητούμε το μονοπάτι εκείνο που συνδέει δύο πόλεις με τη μικρότερη χιλιομετρική απόσταση κλπ.

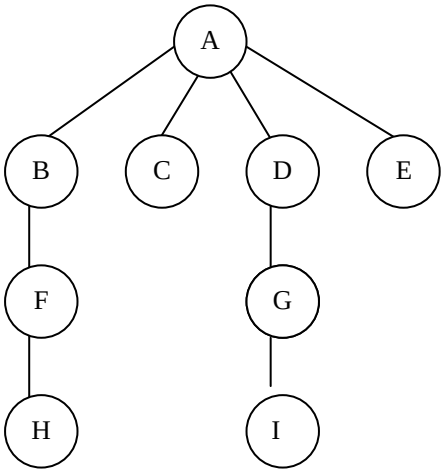
Από τους βασικότερους αλγόριθμους γράφων είναι αυτοί που μας διασφαλίζουν τρόπους επίσκεψης όλων των κορυφών ενός γράφου. Στη συνέχεια θα περιγραφούν δύο τέτοιες μέθοδοι διάσχισης γράφων.

Αναζήτηση με προτεραιότητα βάθους (depth first search)

Αναζήτηση με προτεραιότητα πλάτους (breadth first search)

8.3.1 Αναζήτηση με Προτεραιότητα Βάθους

Υλοποιείται με τη χρήση στοίβας



Σχήμα 8.2 *Αναζήτηση με προτεραιότητα βάθους*

Η μέθοδος έχει ως εξής:

Επιλέγουμε μία κορυφή εκκίνησης, π.χ. την Α, και αμέσως εφαρμόζουμε τρία πράγματα:

α) Επισκεπτόμαστε την κορυφή.

β) Τοποθετούμε (push) την κορυφή σε μία στοίβα για να τη θυμόμαστε.

γ) Τη μαρκάρουμε, ώστε να μη την επισκεφθούμε ξανά.

Ύστερα πηγαίνουμε σε οποιαδήποτε κορυφή που συνδέεται με την Α και την οποία δεν έχουμε ακόμα επισκεφθεί. Έστω ότι πηγαίνουμε στην Β. Την επισκεπτόμαστε, την μαρκάρουμε και την τοποθετούμε στην στοίβα.

Από την κορυφή Β τώρα εφαρμόζουμε την ίδια τεχνική όπως πριν. Αυτό μας οδηγεί στην κορυφή F. Μπορούμε να ονομάσουμε αυτή τη διαδικασία

Κανόνας 1

Αν είναι δυνατόν, επισκεπτόμαστε μία διπλανή κορυφή που δεν την έχουμε ξανα-επισκεφθεί, τη μαρκάρουμε και την τοποθετούμε στη στοίβα.

Εφαρμόζοντας ξανά τον Κανόνα 1, μας οδηγεί στο Η. Τώρα όμως πρέπει να κάνουμε κάτι άλλο, επειδή δεν υπάρχουν διπλανές κορυφές με το Η που δεν τις έχουμε επισκεφθεί. Έτσι εφαρμόζουμε τον Κανόνα 2:

Κανόνας 2

Αν δεν μπορούμε να ακολουθήσουμε τον Κανόνα 1, τότε, αν είναι δυνατόν, εξάγουμε (pop) μία κορυφή από τη στοίβα.

Ακολουθώντας αυτό τον κανόνα, εξάγουμε το Η από τη στοίβα που μας πηγαίνει πίσω στο F. Όμως και το F δεν έχει διπλανές κορυφές που δεν τις έχουμε επισκεφθεί και έτσι το εξάγουμε. Πηγαίνουμε στο Β και κάνουμε το ίδιο. Τελικά καταλήγουμε μόνο με το Α να υπάρχει στη στοίβα. Το Α όμως έχει διπλανές κορυφές που δεν τις έχουμε επισκεφθεί και έτσι επισκεπτόμαστε την επόμενη, που είναι η C. Από εδώ πηγαίνουμε πίσω στην Α, επισκεπτόμαστε την D, μετά την G και μετά την I και με διαδοχικές λειτουργίες εξαγωγής (pop) πηγαίνουμε πίσω στο Α. Επισκεπτόμαστε την Ε, και ξανά πίσω στην Α.

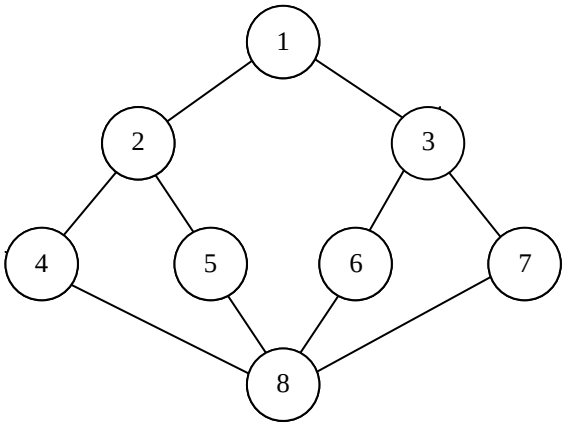
Αυτή τη φορά, όμως, η Α δεν έχει άλλες διπλανές κορυφές που δεν τις έχουμε επισκεφθεί και έτσι την εξάγουμε από τη στοίβα. Η στοίβα πλέον είναι άδεια και έτσι καταλήγουμε στον Κανόνα 3:

Κανόνας 3

Αν δεν μπορούμε να ακολουθήσουμε τον Κανόνα 1 ή τον Κανόνα 2, τότε έχουμε τελειώσει.

Άρα η σειρά είναι A,B,F,H,C,D,G,I, E.

Παράδειγμα



Μία σειρά επίσκεψης είναι 1, 2, 4, 8, 5, 7, 3, 6.

Μία άλλη είναι 1, 3,6,8,7,5,2,4.

8.3.2 Αναζήτηση με Προτεραιότητα Πλάτους

Όπως είδαμε στην Αναζήτηση με Προτεραιότητα Βάθους, ο αλγόριθμος συμπεριφέρεται σαν να θέλει να απομακρυνθεί όσο το δυνατόν περισσότερο από το σημείο εκκίνησης. Στην Αναζήτηση με Προτεραιότητα Πλάτους, απ' την άλλη, ο αλγόριθμος 'αρέσκεται' να παραμένει όσο πιο κοντά γίνεται στο σημείο εκκίνησης. Επισκέπτεται όλες τις κορυφές που είναι διπλανές στην κορυφή εκκίνησης και μόνο τότε προχωρά παρακάτω. Αυτού του είδους η

αναζήτηση υλοποιείται με τη χρήση μίας ουράς. Ας εξετάσουμε το ίδιο παράδειγμα, όπως στην Αναζήτηση με Προτεραιότητα Βάθους:

Η κορυφή Α είναι η κορυφή εκκίνησης, έτσι την επισκεπτόμαστε και την κάνουμε **τρέχουσα** κορυφή. Ύστερα, ακολουθούμε τους εξής κανόνες:

Κανόνας 1

Επίσκεψη της επόμενης (αν υπάρχει) κορυφής, που είναι διπλανή στην τρέχουσα κορυφή και δεν την έχουμε ακόμα επισκεφθεί, τη μαρκάρουμε και την εισάγουμε στην ουρά (enqueue).

Κανόνας 2

Αν δεν μπορούμε να εφαρμόσουμε τον Κανόνα 1, επειδή δεν υπάρχουν άλλες κορυφές που δεν έχουμε ακόμα επισκεφθεί, εξάγουμε (dequeue) μία κορυφή από την ουρά (αν αυτό είναι εφικτό) και την κάνουμε τρέχουσα κορυφή.

Κανόνας 3

Αν δεν μπορούμε να εφαρμόσουμε τον Κανόνα 2, επειδή η ουρά είναι άδεια, τότε έχουμε τελειώσει.

Έτσι λοιπόν, πρώτα επισκεπτόμαστε όλες τις κορυφές που είναι διπλανές στην Α και τις εισάγουμε μία προς μία στην ουρά, καθώς τις επισκεπτόμαστε. Τώρα έχουμε επισκεφθεί τις Α, Β, C, D και Ε. Στο σημείο αυτό, η ουρά (από τον δείκτη front ως τον δείκτη rear) περιέχει τις Β, C, D κα Ε.

Δεν υπάρχουν άλλες διπλανές κορυφές στην Α που δεν έχουμε επισκεφθεί, έτσι εξάγουμε την Β από την ουρά, την κάνουμε τρέχουσα κορυφή και ψάχνουμε για διπλανές κορυφές σ' αυτήν που δεν τις έχουμε ακόμα επισκεφθεί. Βρίσκουμε την F, τη μαρκάρουμε και την εισάγουμε στην ουρά.

Δεν υπάρχουν άλλες διπλανές κορυφές στην Β που δεν τις έχουμε επισκεφθεί και έτσι εξάγουμε την C από την ουρά και την κάνουμε τρέχουσα κορυφή. Δεν έχει διπλανές κορυφές που δεν τις έχουμε επισκεφθεί και έτσι εξάγουμε την D. Κατόπιν επισκεπτόμαστε την G, τη μαρκάρουμε και την εισάγουμε στην ουρά. Η D δεν έχει άλλες διπλανές κορυφές που δεν έχουμε επισκεφθεί και έτσι εξάγουμε την Ε.

Τώρα η ουρά έχει τις F, G. Εξάγουμε την F και επισκεπτόμαστε την H και μετά εξάγουμε την G και επισκεπτόμαστε την I.

Τώρα η ουρά έχει τις H, I τις οποίες εξάγουμε με τη σειρά, διαπιστώνοντας ότι δεν υπάρχουν άλλες διπλανές κορυφές που δεν έχουμε επισκεφθεί και έτσι η ουρά είναι άδεια. Στο σημείο αυτό έχουμε τελειώσει.

Άρα η σειρά επίσκεψης είναι A, B, C, D, E, F, G, H, I.

8.4 ΤΟ ΠΡΟΒΛΗΜΑ ΤΟΥ ΣΥΝΤΟΜΟΤΕΡΟΥ ΜΟΝΟΠΑΤΙΟΥ (shortest path problem)

Ίσως το πιο συνηθισμένο πρόβλημα που αντιμετωπίζουμε, και που να σχετίζεται με εφαρμογές γράφων, είναι η εύρεση του συντομότερου μονοπατιού ανάμεσα σε δύο δεδομένες κορυφές. Η λύση σ' αυτό το πρόβλημα εφαρμόζεται σε μία μεγάλη ποικιλία από πραγματικές καταστάσεις, όπως τα σχέδια κάποιου πίνακα κυκλωμάτων, το χρονοδιάγραμμα κάποιου έργου, τις χιλιομετρικές αποστάσεις μεταξύ πόλεων σε ένα οδικό δίκτυο ή ένα δίκτυο με σταθμούς ενός υπόγειου σιδηρόδρομου.

Η λύση που υλοποιείται για το πρόβλημα αυτό ονομάζεται αλγόριθμος του Dijkstra, μια και αναπτύχθηκε από τον Edsger Dijkstra, που πρώτος τον περιέγραψε το 1959. Ο αλγόριθμος βασίζεται στην αναπαράσταση ενός γράφου με τον πίνακα διπλανών κορυφών και βρίσκει το συντομότερο μονοπάτι από μία συγκεκριμένη κορυφή (πηγή – source) προς μία άλλη (προορισμός – destination). Στην ουσία ο αλγόριθμος βρίσκει το συντομότερο μονοπάτι από μία κορυφή προς όλες τις άλλες κορυφές.

ΙΧ. ΠΙΝΑΚΕΣ ΚΑΤΑΚΕΡΜΑΤΙΣΜΟΥ

9.1 Εισαγωγή..... 9-1

9.2 Συγκρούσεις..... 9-3

9.3 Ανοιχτή διευθυνσιοδότηση..... 9-3

9.4 Ξεχωριστή σύνδεση..... 9-4

9.1 ΕΙΣΑΓΩΓΗ

Ένας πίνακας κατακερματισμού (hash table) είναι μία δομή που προσφέρει πολύ γρήγορη εισαγωγή και αναζήτηση. Επιπλέον, οι πίνακες κατακερματισμού προγραμματίζονται σχετικά εύκολα. Ωστόσο έχουν και αρκετά μειονεκτήματα. Βασίζονται σε πίνακες και οι πίνακες δύσκολα διευρύνονται από τη στιγμή που θα ορισθούν. Για μερικές περιπτώσεις πινάκων κατακερματισμού, η απόδοση μπορεί να υποβαθμισθεί καταστροφικά, όταν ο πίνακας γεμίσει, κι έτσι είναι απαραίτητο για τον προγραμματιστή να έχει μία καλή ιδέα ως προς το πόσα στοιχεία χρειάζεται να αποθηκευτούν ή να είναι προετοιμασμένος να μεταφέρει κατά περιόδους δεδομένα σε ένα μεγαλύτερο πίνακα, διαδικασία οπωσδήποτε χρονοβόρα.

Επίσης, δεν είναι βολικό να επισκεπτόμαστε τα στοιχεία σε ένα πίνακα με οποιαδήποτε σειρά, όπως π.χ. από το μικρότερο προς το μεγαλύτερο. Όμως, αν δεν υπάρχει ανάγκη να επισκεφθούμε τα στοιχεία με τη σειρά και μπορούμε να γνωρίζουμε εκ των προτέρων το πλήθος των στοιχείων που θα αποθηκευτούν, οι πίνακες κατακερματισμού είναι ασυναγώνιστοι σε ταχύτητα και ευκολία.

Ένα σημαντικό θέμα είναι πώς ένα πλήθος από κλειδιά θα μετασχηματισθεί σε ένα πλήθος από θέσεις πίνακα. Σε ένα πίνακα κατακερματισμού αυτό επιτυγχάνεται με μία συνάρτηση κατακερματισμού (hashing function). Όμως, για κάποια είδη κλειδιών δεν απαιτείται η συνάρτηση κατακερματισμού. Οι τιμές των κλειδιών μπορούν κατευθείαν να χρησιμοποιηθούν ως θέσεις πίνακα. Ας εξετάσουμε αυτή την απλή περίπτωση πρώτα:

Έστω ότι γράφουμε ένα πρόγραμμα με σκοπό την προσπέλαση εγγραφών εργαζομένων για μια μικρή εταιρεία, με π.χ. 1000 εργαζόμενους. Κάθε εγγραφή εργαζόμενου απαιτεί, έστω, 1000 bytes αποθηκευτικό χώρο. Έτσι μπορούμε να αποθηκεύσουμε όλη τη βάση δεδομένων σε 1 megabyte, που άνετα χωρά στη μνήμη του υπολογιστή μας. Στον κάθε εργαζόμενο έχει δοθεί ένας αριθμός μητρώου από το 1 έως το 1000. Αυτοί οι αριθμοί μητρώου μπορούν να χρησιμοποιηθούν σαν κλειδιά για την προσπέλαση των εγγραφών. Επομένως, για την αποθήκευση των πληροφοριών, θα επιλέγαμε ένα πίνακα

1000 θέσεων, όπου η κάθε θέση του πίνακα αντιπροσωπεύει έναν αντίστοιχο αριθμό μητρώου ενός εργαζόμενου.

Άρα, όταν μας ενδιαφέρει κάποιος εργαζόμενος, μέσω του αριθμού μητρώου γίνεται άμεση προσπέλαση στον πίνακα χωρίς να χρειάζεται κάποιο είδος αναζήτησης. Επίσης, η εισαγωγή μιας νέας εγγραφής επιτυγχάνεται εύκολα και γρήγορα, π.χ. δίνεται ο αριθμός μητρώου 1001 σε ένα νέο εργαζόμενο και τοποθετείται στη θέση 1001 του πίνακα, ο οποίος βέβαια θα πρέπει πρώτα να διευρυνθεί.

Πολλές φορές, όμως, τα κλειδιά των εγγραφών δεν αποτελούν συνεχόμενη ακολουθία ακεραίων ή μπορεί να μην είναι καν αριθμητικά. Στην περίπτωση που τα κλειδιά δεν είναι αριθμητικά, χρησιμοποιούμε τις ASCII τιμές των χαρακτήρων τους –τις οποίες μετά προσθέτουμε – για να τα μετατρέψουμε σε αριθμητικά. Και πάλι όμως έχουμε να τοποθετήσουμε αριθμητικά κλειδιά με μεγάλες διαφορές μεταξύ τους σε ένα πίνακα με συνεχόμενες θέσεις.

Π.χ. έστω ότι έχουμε 10 κλειδιά στο διάστημα 0 – 199. Χρειαζόμαστε ένα πίνακα 10 θέσεων με δείκτες 0 – 9. Μία καλή λύση είναι να διαιρούμε τα κλειδιά με το 10, που είναι το μέγεθος του πίνακα και να παίρνουμε το υπόλοιπο της διαίρεσης. Για παράδειγμα, η εγγραφή με κλειδί 13 θα αποθηκευτεί στη θέση 13%10 = 3.

Μία παρόμοια έκφραση μπορεί χρησιμοποιηθεί για να βρεθεί η θέση του πίνακα για οποιοδήποτε κλειδί.

Θέση πίνακα = αρχικό κλειδί % μέγεθος πίνακα

Αυτό είναι ένα παράδειγμα μίας συνάρτησης κατακερματισμού. Μετασχηματίζει ένα μεγάλο διάστημα τιμών σε ένα μικρότερο διάστημα. Ένας πίνακας στον οποίο εισάγονται δεδομένα με μία τέτοια συνάρτηση μετασχηματισμού του κλειδιού ονομάζονται πίνακες κατακερματισμού (hash tables).

9.2 ΣΥΓΚΡΟΥΣΕΙΣ (Collisions)

Ο προηγούμενος μετασχηματισμός δεν εγγυάται πάντα ότι δύο διαφορετικά κλειδιά δεν θα οδηγηθούν στην ίδια θέση του πίνακα. Για παράδειγμα, οι εγγραφές με κλειδιά 13 και 23 προκύπτει ότι θα αποθηκευτούν στη θέση 3. Αυτή η κατάσταση ονομάζεται **σύγκρουση** (collision). Οι δύο εγγραφές που διεκδικούν την ίδια θέση στον πίνακα ονομάζονται **συνώνυμα** (synonyms).

Μία προσέγγιση για τη λύση στο πρόβλημα αυτό, είναι να εξετάσουμε τον πίνακα, να βρούμε μία άδεια θέση και να εισάγουμε εκεί το νέο στοιχείο. Αυτή η τεχνική ονομάζεται **ανοιχτή διευθυνσιοδότηση0** (open addressing).

Μία δεύτερη προσέγγιση είναι η δημιουργία ενός πίνακα που αποτελείται από συνδεδεμένες λίστες. Έτσι, όταν συμβεί σύγκρουση, το νέο στοιχείο εισάγεται στη λίστα της συγκεκριμένης θέσης του πίνακα. Αυτή η τεχνική ονομάζεται **ξεχωριστή σύνδεση** (separate chaining).

9.3 ΑΝΟΙΧΤΗ ΔΙΕΥΘΥΝΣΙΟΔΟΤΗΣΗ (Open Addressing)

Η πιο απλή μέθοδος ανοιχτής διευθυνσιοδότησης είναι η γραμμική εξέταση (linear probing). Η μέθοδος ψάχνει σειριακά στον πίνακα, ξεκινώντας από την επόμενη θέση από όπου συνέβη η σύγκρουση, μέχρι να βρει μια άδεια θέση, όπου και αποθηκεύει το νέο στοιχείο.

Παράδειγμα

Έχουμε 10 κλειδιά με τιμές 13, 28, 41, 70, 127, 131, 144, 176, 182, 199.
Όταν γίνει η αποθήκευση των προηγούμενων κλειδιών σε ένα πίνακα 10 θέσεων, θα καταλάβουν τις εξής θέσεις:

0	1	2	3	4	5	6	7	8	9
70	41	**131**	13	144	**182**	176	127	28	199

Παρατηρούμε ότι το κλειδί 131 κανονικά έπρεπε να αποθηκευτεί στη θέση 1. Εκεί όμως προηγήθηκε η αποθήκευση του κλειδιού 41. Επομένως, σύμφωνα με τη μέθοδο το κλειδί 131 αποθηκεύεται στην αμέσως επόμενη άδεια θέση, που ήταν η 2.

Η θέση 2 όμως είναι η φυσική θέση του κλειδιού 182, το οποίο δεν μπορεί να αποθηκευτεί εκεί, γιατί είναι κατειλημμένη από κάποιο συνώνυμο ενός άλλου κλειδιού. Έτσι, με τον ίδιο τρόπο, το κλειδί 182 αποθηκεύεται στην αμέσως επόμενη διαθέσιμη θέση που είναι η 5.

Αυτό είναι ένα από τα μειονεκτήματα αυτής της μεθόδου, όταν παρουσιασθούν πολλά συνώνυμα. Σε αυτή την περίπτωση ίσως είναι προτιμότερο να χρησιμοποιήσουμε κάποια άλλη μέθοδο όπως είναι ο τετραγωνικός έλεγχος (quadratic probing) ή ο διπλός κατακερματισμός (double hashing).

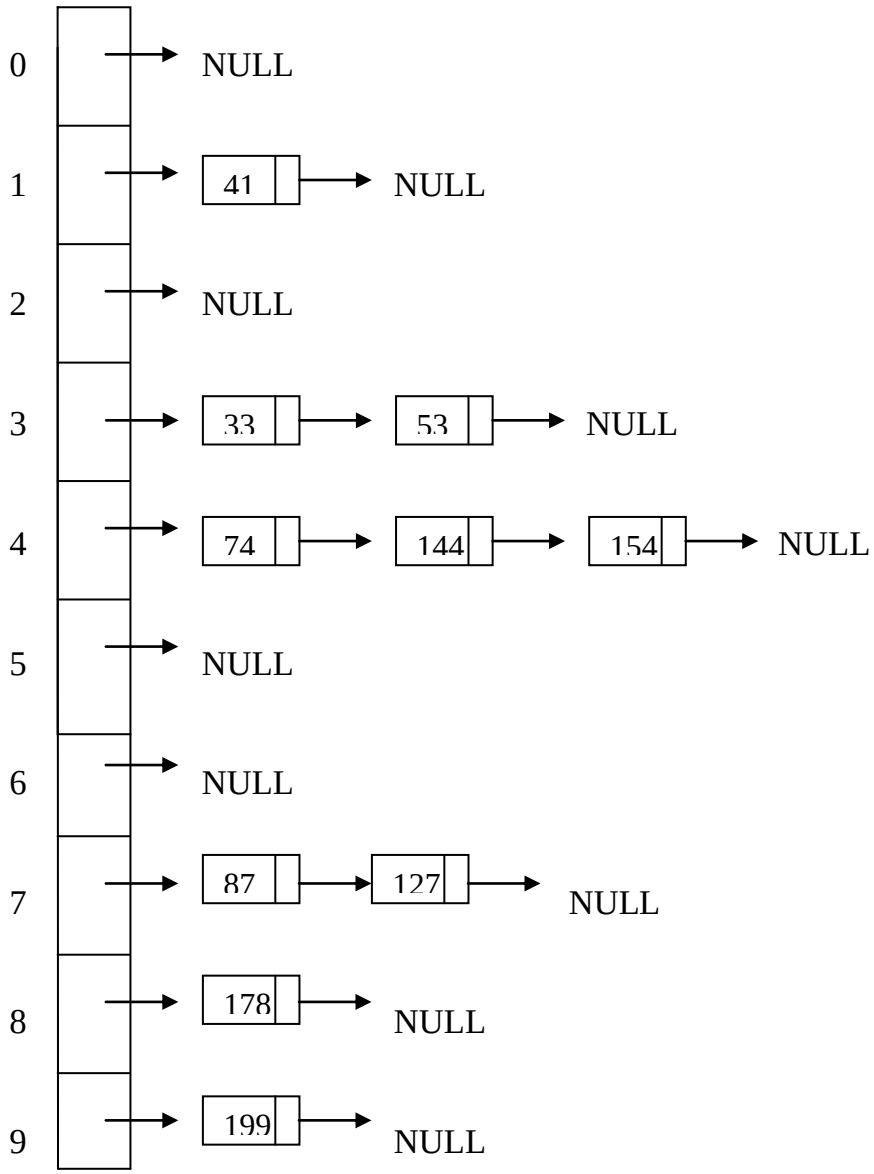
9.4 ΞΕΧΩΡΙΣΤΗ ΣΥΝΔΕΣΗ (Separate Chaining)

Όπως αναφέρθηκε και προηγουμένως, όταν υπάρχουν πολλά συνώνυμα τότε η μέθοδος γραμμικής εξέτασης δεν είναι ιδιαίτερα αποτελεσματική. Σ' αυτή την περίπτωση ίσως θα ήταν προτιμότερο να επιλέξουμε την τεχνική της **ξεχωριστής σύνδεσης** (separate chaining). Στην τεχνική αυτή ο πίνακας περιέχει ένα πλήθος από συνδεδεμένες λίστες.

Παράδειγμα

Έστω ότι έχουμε τα κλειδιά 33, 41, 53, 74, 87, 127, 144, 154, 178, 199 και θέλουμε να τα αποθηκεύσουμε σε ένα πίνακα 10 θέσεων με θέσεις 0 – 9. Χρησιμοποιώντας την μέθοδο της διαίρεσης των κλειδιών με το 10 και παίρνοντας το υπόλοιπο ως τη θέση στον πίνακα, παρατηρούμε ότι τα κλειδιά 74, 144, 154 είναι συνώνυμα . Το ίδιο και τα 33, 53 καθώς επίσης και τα 87, 127. Αν χρησιμοποιήσουμε την προηγούμενη μέθοδο θα έχουμε μια καταστροφικά αναποτελεσματική δομή αποθήκευσης. Για το λόγο αυτό θα επιλέξουμε την τεχνική της ξεχωριστής σύνδεσης.

Όταν γίνει η αποθήκευση των κλειδιών θα έχουμε την εξής κατάσταση:



ΠΑΡΑΡΤΗΜΑ Ι

Πίνακας χαρακτηριστικών Δομών Δεδομένων

Δομή Δεδομένων	Πλεονεκτήματα	Μειονεκτήματα
ΠΙΝΑΚΑΣ	Γρήγορη εισαγωγή, πολύ γρήγορη προσπέλαση αν η θέση είναι γνωστή	Αργή αναζήτηση, αργή διαγραφή, προκαθορισμένο μέγεθος
ΣΤΟΙΒΑ	LIFO λειτουργία	Αργή προσπέλαση σε άλλα στοιχεία
ΟΥΡΑ	FIFO λειτουργία	Αργή προσπέλαση σε άλλα στοιχεία
ΣΥΝΔΕΔΕΜΕΝΗ ΛΙΣΤΑ	Γρήγορη εισαγωγή και διαγραφή	Αργή αναζήτηση
ΔΥΑΔΙΚΟ ΔΕΝΔΡΟ	Γρήγορη αναζήτηση, εισαγωγή και διαγραφή	Αλγόριθμος διαγραφής πολύπλοκος
ΓΡΑΦΟΣ	Απεικονίζει καταστάσεις πραγματικού κόσμου	Μερικοί αλγόριθμοι είναι αργοί και πολύπλοκοι

ΒΙΒΛΙΟΓΡΑΦΙΑ

- *Δομές Δεδομένων, τόμος Α΄*
  Γ. Κόλλιας, Γ. Μανωλόπουλος

- *Αλγόριθμοι και τεχνικές προγραμματισμού*
  Σ. Ζαχαρόπουλος

- *Ανάπτυξη Εφαρμογών σε προγραμματιστικό περιβάλλον*
  Παιδαγωγικό Ινστιτούτο

- *Δομές Δεδομένων, Σημειώσεις*
  Δ. Σταμάτης

- *Δομές Δεδομένων, Σημειώσεις*
  Χ. Στρουθόπουλος

- *Algorithms & Data Structures*
  Nicklaus Wirth

- *Data Structures & Algorithms in JAVA*
  Robert Lafore

- *Προγραμματισμός και Δομές Δεδομένων στην C*
  Leendert Ammeraal

- *An Introduction to the Art and Science of Programming*
  Walter J. Savitch