ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

# ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Ι. ΞΕΖΩΝΑΚΗΣ

ΗΡΑΚΛΕΙΟ 2020

# ΠΕΡΙΕΧΟΜΕΝΑ

# ΚΕΦΑΛΑΙΟ 1

# ΑΛΓΟΡΙΘΜΟΙ – ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ



## 1.1. ΑΛΓΟΡΙΘΜΟΙ - ΓΕΝΙΚΑ.

Αλγόριθμος είναι η περιγραφή μιας αυστηρά καθορισμένης **σειράς βημάτων** για την λύση ενός προβλήματος. Κάθε αλγόριθμος έχει κανένα, ένα ή περισσότερα δεδομένα εισόδου και ένα τουλάχιστον αποτέλεσμα.

Οι αλγόριθμοι διατυπώνονται συνήθως σε μια γλώσσα που μπορεί να περιγράψει τη σειρά των βημάτων με σαφή και μαθηματικά ξεκάθαρο τρόπο, χωρίς ασάφειες. Τέτοιες γλώσσες είναι συνήθως οι γλώσσες προγραμματισμού, κάποια συμβολική γλώσσα, αλλά και οι φυσικές γλώσσες καμμιά φορά. Στην διατύπωση με γλώσσα προγραμματισμού χρησιμοποιείται παρακάτω, αλλά και παντού όπου απαιτείται πρόγραμμα στις σημειώσεις αυτές, η γλώσσα C.

Παρακάτω δίνεται και με τους τρεις τρόπους διατύπωσης ένας πολύ γνωστός αλγόριθμος, ο αλγόριθμος του Ευκλείδη. Αυτός χρησιμοποιείται για να βρίσκουμε τον Μέγιστο Κοινό Διαιρέτη (ΜΚΔ) δυο θετικών ακεραίων, έστω των x και y.

**α) ΔΙΑΤΥΠΩΣΗ ΜΕ ΦΥΣΙΚΗ ΓΛΩΣΣΑ:**

Διαίρεσε το x δια y με ακέραια διαίρεση και έστω z το υπόλοιπο. Αν το z είναι μηδέν, τότε ο ΜΚΔ είναι ο y. Αν το z είναι διάφορο του μηδενός, τότε επανάλαβε το παραπάνω βάζοντας στην θέση του x το y και στη θέση του y το z.

**β) ΔΙΑΤΥΠΩΣΗ ΜΕ ΣΥΜΒΟΛΙΚΗ ΓΛΩΣΣΑ:**

**Δεδομένα:** x, y ≠ 0, ακέραιοι. z ακέραιος
**Αρχή**
    z = y
**Όσο** z ≠ 0 κάνε
    z = x modulo y
    x = y
    y = z
**Τέλος επανάληψης**
**Τέλος**
**Αποτελέσματα:** ΜΚΔ = y

**γ) ΔΙΑΤΥΠΩΣΗ ΜΕ ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ( C ):**

```
int mkd (int x, int y) {
    int z ;

    z = y ;
    while (z != 0) {
        z = x % y;
        x = y;
        y = z ; }
    return y ; }
```



## 1.2. ΑΝΑΛΥΣΗ ΑΛΓΟΡΙΘΜΩΝ.

Αναφέρουμε εδώ με συντομία τον τρόπο εκτίμησης της επίδοσης ενός αλγορίθμου, το πόσο αποδοτικός είναι δηλαδή ο αλγόριθμος. Ο πιο απλός και εύκολος τρόπος είναι ο **εμπειρικός** ή αλλιώς **εκ των υστέρων** (a posteriori) τρόπος. Σύμφωνα με αυτό τον τρόπο, ο αλγόριθμος εφαρμόζεται σε ένα σύνολο δεδομένων και υπολογίζεται ο **χρόνος επεξεργασίας** και η απαιτούμενη **μνήμη**. Ο τρόπος αυτός δεν είναι πάντως ο καλύτερος για δυο κυρίως λόγους και συγκεκριμένα:

- Είναι δύσκολο να εκτιμήσουμε τη συμπεριφορά του αλγορίθμου εάν αλλάξουμε τα δεδομένα μας.
- Ο χρόνος επεξεργασίας είναι συνάρτηση του υλικού, της γλώσσας προγραμματισμού, του μεταγλωττιστή, αλλά και της εμπειρίας και ικανότητας του προγραμματιστή.

Ένας δεύτερος τρόπος εκτίμησης της επίδοσης ενός αλγορίθμου είναι ο θεωρητικός ή αλλιώς **εκ των προτέρων** (a priori) τρόπος, μας ενδιαφέρει δηλαδή πόσο αποδοτικός είναι ο αλγόριθμος. Η συνάρτηση **κεφαλαίο-Ο** (**Big-O**) είναι ένα **μέτρο της χειρότερης δυνατής περίπτωσης (πολυπλοκότητας) ενός αλγορίθμου**, καθώς το πλήθος των δεδομένων αυξάνει. Η τυπική μορφή αυτού του κατά κάποιο τρόπο «μέτρου πολυπλοκότητας», αποτελείται από μια αλγεβρική παράσταση, η οποία αντιπροσωπεύει την ταχύτητα του αλγορίθμου, ανάλογα με το πλήθος των δεδομένων στα οποία εφαρμόζεται. Για παράδειγμα, το Big-O για το σειριακό ψάξιμο μιας απλά συνδεδεμένης λίστας n κόμβων, είναι O(n). Για να το πούμε πιο απλά, εάν έχουμε n κόμβους, η χειρότερη περίπτωση είναι να χρειαστεί να ελέγξουμε όλους τους κόμβους της λίστας, άρα να έχουμε τελικά n συγκρίσεις. Υπάρχουν διάφορες περιπτώσεις Big-O, όπως για παράδειγμα $O(n^2)$, η οποία μπορεί να προκύψει από διπλούς βρόχους for, ή ακόμα και O(n!), η οποία περιγράφει αλγορίθμους που χρειάζονται τεράστιες ποσότητες χρόνου για να ολοκληρωθούν. Άλλες δυο ενδιαφέρουσες περιπτώσεις Big-O είναι οι O(logn) και O(c), όπου c είναι μια σταθερά. Στην περίπτωση του O(c), ο απαιτούμενος χρόνος για να ολοκληρωθεί ο αλγόριθμος είναι πάντα ο ίδιος, ανεξάρτητα από το πλήθος των δεδομένων.

Η συνάρτηση Big-O δεν χρησιμοποιείται μόνο για τον προσδιορισμό του πλήθους των βημάτων, της ταχύτητας ενός αλγορίθμου δηλαδή, αλλά και για την **εκτίμηση της αναγκαίας μνήμης** για την υλοποίηση του αλγορίθμου.

## 1.3. ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ - ΓΕΝΙΚΑ.

Χρησιμοποιούνται επειδή επιτρέπουν την προσπέλαση και επεξεργασία δεδομένων με εύκολο τρόπο. Η πρώτη δομή δεδομένων που γνωρίζουμε ήδη είναι οι πίνακες. Δομές, δηλαδή τρόποι οργάνωσης και επεξεργασίας δεδομένων, με τις οποίες θα ασχοληθούμε, είναι οι παρακάτω:

| **Δομή** | **Ενσωματωμένη** |
|---|---|
| Πίνακας (Array) | ΝΑΙ |
| Εγγραφή (Record) | ΝΑΙ |
| Συνδεδεμένη λίστα (Linked list) | ΟΧΙ |
| Στοίβα (Stack) | ΟΧΙ |
| Ουρά (Queue) | ΟΧΙ |
| Δέντρο (Tree) | ΟΧΙ |
| Γράφοι (Graphs) | ΟΧΙ |

Οι **ενσωματωμένες δομές** μπορούν να δηλωθούν άμεσα στη C, ενώ οι **μη ενσωματωμένες** πρέπει να δημιουργηθούν, να κατασκευαστούν δηλαδή από τον προγραμματιστή.

Ένας κύριος διαχωρισμός των δομών δεδομένων είναι σε γραμμικές και μη γραμμικές:

- **Γραμμικές** είναι οι δομές εκείνες για τις οποίες μπορεί να οριστεί **διάταξη** για δυο στοιχεία τους, δηλαδή κάποιο στοιχείο είναι πρώτο και κάποιο τελευταίο, ενώ οποιοδήποτε από τα υπόλοιπα θα έχει ένα προηγούμενο, από το οποίο έπεται και ένα επόμενο, από το οποίο προηγείται. Χαρακτηριστικά είδη γραμμικών δομών είναι οι πίνακες, οι λίστες, οι στοίβες και οι ουρές.
- Στις **μη γραμμικές δομές**, οι σχέσεις των δεδομένων είναι πιο περίπλοκες και όχι μονοδιάστατες. Κάθε στοιχείο μιας τέτοιας δομής μπορεί να έχει πολλά επόμενα στοιχεία. Χαρακτηριστικό είδος μη γραμμικής δομής είναι τα δέντρα. Στους γράφους τα πράγματα είναι ακόμη πιο περίπλοκα, αφού πολλές φορές δεν έχει νόημα η έννοια του προηγούμενου και του επόμενου στοιχείου ή αλλιώς, κάθε στοιχείο μπορεί να έχει πολλά προηγούμενα και πολλά επόμενα.

# ΚΕΦΑΛΑΙΟ 2

# ΠΙΝΑΚΕΣ

## 2.1. ΜΟΝΟΔΙΑΣΤΑΤΟΙ ΠΙΝΑΚΕΣ ΩΣ ΟΡΙΣΜΑΤΑ ΣΥΝΑΡΤΗΣΕΩΝ.

Είναι γνωστό από τα εισαγωγικά μαθήματα προγραμματισμού με C το πώς «περνάμε» διάφορα είδη μεταβλητών ως ορίσματα σε συναρτήσεις. Εάν μια συνάρτηση χρειάζεται ως πληροφορία τα στοιχεία ενός πίνακα, **δεν είναι δυνατό να θέσουμε ολόκληρο τον πίνακα ως όρισμα** στη συνάρτηση, δηλαδή η συνάρτηση δεν μπορεί να δημιουργήσει αντίγραφο του αρχικού πίνακα. Έτσι, προκειμένου να «περάσουν» οι τιμές του πίνακα στην συνάρτηση, **θέτουμε ως ορίσματα ένα δείκτη στην αρχή του πίνακα και μια μεταβλητή (int)**, η οποία δηλώνει το μέγεθος του πίνακα.

Στο παρακάτω παράδειγμα διαβάζονται ακέραιοι από το πληκτρολόγιο και καταχωρούνται στον πίνακα list. Το διάβασμα συνεχίζεται μέχρι να γεμίσει ο πίνακας ή μέχρι να δοθεί ακέραιος ίσος με μηδέν.

```
#define  SIZE 10

int main(void) {
   int  list[SIZE];
   int  k, meg, ak;

   for (k=0; k<SIZE; k++) {
      scanf ("%d", &ak);
      if (ak == 0)
          break;
      else
          list [k] = ak;
   }
   meg = max (list, k);
   printf ("Μέγιστος αριθμός του πίνακα: %d", meg);
   return 1; }
```

Όταν τελειώσει το διάβασμα, στον πίνακα θα έχουν τοποθετηθεί συνολικά k ακέραιοι διάφοροι του μηδενός. Στη συνέχεια καλείται μια συνάρτηση, η max( ), η οποία βρίσκει το μέγιστο μη μηδενικό στοιχείο (meg) του πίνακα αυτού.

Το πρόγραμμα υλοποιείται με δυο τρόπους. Η main( ) είναι και στους δυο τρόπους η ίδια. Διαφορά υπάρχει στον ορισμό και στη δήλωση της συνάρτησης max( ).

**Α' τρόπος:**

Η συνάρτηση δέχεται ως **ορίσματα ένα δείκτη σε ακέραιο και ένα ακέραιο**. Ο δείκτης σε ακέραιο γίνεται ίσος με τον δείκτη στην πρώτη θέση του πίνακα ακεραίων (του list). Ο ακέραιος γίνεται ίσος με το πλήθος των μη μηδενικών στοιχείων, τα οποία διαβάστηκαν από το πληκτρολόγιο και καταχωρήθηκαν στον πίνακα.

```
int max (int *mlist, int mnum) {
    int mmeg, mk;

    mmeg = *mlist ;
    for (mk=1 ; mk<mnum ; mk++)
        if (mmeg < *(mlist+mk))
            mmeg = *(mlist+mk);
    return mmeg;
}
```

Δήλωση της max( ):

```
int max (int *, int);
```

**Β' τρόπος:**

Είναι ισοδύναμος, χρησιμοποιείται δε συχνά με αριθμητικούς πίνακες και **κάνει χρήση αγκυλών**. Είναι εύχρηστος διότι μας υπενθυμίζει ότι κάνουμε χρήση πινάκων:

```
int max (int mlist[ ], int mnum) {
    int mmeg, mk;

    mmeg = mlist[0] ;
    for (mk=1 ; mk<mnum ; mk++)
        if (mmeg < mlist[mk])
            mmeg = mlist[mk];
    return mmeg;
}
```

Δήλωση της max( ):

```
int max (int [ ], int);
```

**Ακόμα και με τον πρώτο τρόπο (με τη χρήση δηλαδή του αστερίσκου, * ), μπορούμε να χρησιμοποιούμε αγκύλες για τους χειρισμούς των πινάκων.**

**Παράδειγμα: Ταξινόμηση ενός μονοδιάστατου πίνακα.**

Δίνουμε ένα ακόμη παράδειγμα συνάρτησης, η οποία χρειάζεται ως πληροφορία ένα πίνακα ακεραίων, άρα δέχεται ως όρισμα ένα δείκτη στο πρώτο στοιχείο του πίνακα. Η συνάρτηση χρησιμοποιείται για την ταξινόμηση του πίνακα. Η **ταξινόμηση** θα παρουσιαστεί διεξοδικότερα σε επόμενο κεφάλαιο, είναι πάντως πολύ σημαντική λειτουργία σε πολλές εφαρμογές (π.χ. σε βάσεις δεδομένων), στις οποίες ζητούμε την **αναδιάταξη μιας λίστας από στοιχεία** με αριθμητική ή αλφαβητική σειρά.

Έστω ένας πίνακας ακεραίων, ο list, με num θέσεις. Ο πίνακας έχει πάρει τιμές από το πληκτρολόγιο. Η συνάρτηση sort( ) θα ταξινομήσει τα στοιχεία του πίνακα.

Ορισμός:

```
void sort (int slist[ ], int snum) {
   int sout, sin, stemp;

   for (sout=0 ; sout<snum-1 ; sout++)
        for (sin=sout+1 ; sin<snum ; sin++)
             if (slist[sout] > slist[sin]) {
                  stemp = slist[sin];
                  slist[sin] = slist[sout];
                  slist[sout] = stemp; }
}
```

Κλήση από τη main( ):

```
sort (list, num);
```

Υπάρχουν διάφοροι αλγόριθμοι ταξινόμησης για ένα πίνακα. Αυτός που εφαρμόζεται εδώ είναι γνωστός ως **«Ταξινόμηση με επιλογή».**

Η συνάρτηση ξεκινά από το στοιχείο της πρώτης θέσης του πίνακα, το slist[0], με στόχο να τοποθετηθεί στην θέση αυτή η μικρότερη τιμή του πίνακα. Η συνάρτηση διατρέχει όλα τα υπόλοιπα στοιχεία, από το slist[1] μέχρι το slist[snum-1] και συγκρίνει καθένα με το πρώτο. Αν βρει κάποιο μικρότερο από το πρώτο, τα στοιχεία εναλλάσσονται μεταξύ τους. Τελικά το slist[0] θα έχει τη μικρότερη τιμή του πίνακα.

Αφού τελειώσουμε με το πρώτο στοιχείο επαναλαμβάνεται η ίδια διαδικασία, προσπαθώντας να βάλουμε στη θέση 1 του πίνακα τη δεύτερη σε μέγεθος τιμή. Συγκρίνονται δηλαδή όλα τα στοιχεία από το slist[2] και μετά με το slist[1]. Αν βρεθεί κάποιο στοιχείο μικρότερο από το slist[1], τα στοιχεία εναλλάσσονται μεταξύ τους κ.ο.κ.

## 2.2. ΠΙΝΑΚΕΣ ΔΥΟ ΔΙΑΣΤΑΣΕΩΝ.

Έστω για παράδειγμα ότι καταγράφουμε τη μέγιστη ημερήσια θερμοκρασία μιας πόλης για ένα έτος. Για τον σκοπό αυτό χρειαζόμαστε 366 μεταβλητές ή καλύτερα ένα πίνακα όπως ο:

```
int thermo[366];
```

Ο πίνακας είναι μια δομή, με την οποία μπορούμε να έχουμε οργανωμένα δεδομένα, όχι δηλαδή 366 διαφορετικές μεταβλητές διάσπαρτες στην μνήμη. Ο πίνακας thermo είναι πίνακας με **μια** μόνο διάσταση, δηλαδή **μια μόνο αριθμητική ετικέτα**. Ένας τέτοιος πίνακας δεν είναι πολύ βολικός αν χρειαζόμαστε π.χ. την θερμοκρασία της 27ης Μαρτίου. Αντίθετα, ο πίνακας:

```
int hottest [12] [31];
```

έχει **δυο** διαστάσεις, δηλαδή **δυο αριθμητικές ετικέτες**. Η προσπέλαση κάθε στοιχείου του γίνεται χρησιμοποιώντας και τις δυο ετικέτες. Έτσι, το στοιχείο hottest [3] [27] θα μπορούσε να αντιστοιχεί στην θερμοκρασία που αναφέραμε. Εννοείται ότι το «γέμισμα», η αρχικοποίηση του πίνακα, θα γίνει ακολουθώντας την λογική βάσει της οποίας δημιουργήθηκε ο πίνακας, θα έχουμε δηλαδή κατά νου τι εκφράζει η κάθε διάστασή του. Τον πίνακα hottest μπορούμε να τον δούμε με δυο διαφορετικές «ματιές»:

- Ως ένα πίνακα 12x31.
- Ως 12 πίνακες με 31 θέσεις ακεραίων καθένας, ένα μονοδιάστατο δηλαδή πίνακα 12 θέσεων, στην κάθε θέση του οποίου όμως δεν υπάρχει ένας ακέραιος, αλλά ένας πίνακας ακεραίων 31 θέσεων.



**Απόδοση αρχικών τιμών:**

Έστω ο:

```
int matrix[5];
```

Ο πίνακας μπορεί να πάρει αρχικές τιμές κατά την δήλωσή του ως εξής:

```
int matrix[5] = {7, 5, 1, 95, -4};
```

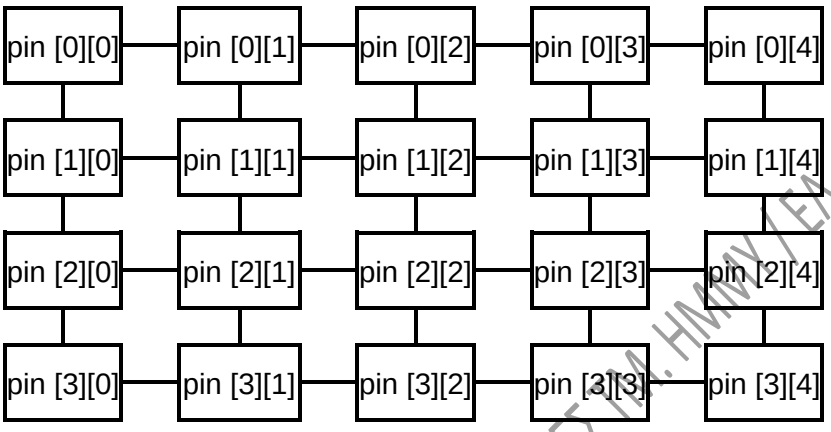
Αντίστοιχα, σε ένα πίνακα δυο διαστάσεων, η απόδοση αρχικών τιμών θα είναι:

```
int pin[4][5] = {{2, 4, 3, 5, 6},
                 {3, 6, 5, 2, 5},
                 {7, 2, 3, 0, 7},
                 {0, 6, 1, 8, 8}};
```

(οι τιμές κάθε γραμμής δηλαδή σε χωριστά άγκιστρα).

## 2.3. ΑΠΟΘΗΚΕΥΣΗ ΣΤΗ ΜΝΗΜΗ.

Τα στοιχεία ενός πολυδιάστατου πίνακα αποθηκεύονται στη μνήμη σε συνεχόμενες θέσεις, γραμμή προς γραμμή. Σχηματικά ο pin που δηλώθηκε πιο πάνω μπορεί να παρασταθεί:




Σχ. 2.1.

Αλλάζοντας τη **δεύτερη ετικέτα** κινούμαστε σε μια **γραμμή** και αλλάζοντας την **πρώτη ετικέτα** κινούμαστε κατά μήκος μιας **στήλης**.

Στη μνήμη η **αποθήκευση** των στοιχείων γίνεται ανά γραμμή, δηλαδή πρώτα το στοιχείο pin [0] [0], μετά το pin [0] [1], μετά το pin [0] [2] κλπ μέχρι το τέλος της πρώτης γραμμής. Στη συνέχεια το pin [1] [0], το pin [1] [1] κλπ, μέχρι τελικά το pin [3] [4]. Μπορούμε να επισημάνουμε την **θέση αποθήκευσης οποιουδήποτε στοιχείου του πίνακα σε σχέση με το αρχικό**. Το στοιχείο pin [2][3] για παράδειγμα είναι το 14ο στοιχείο στη σειρά σε αυτή την αποθήκευση. Το pin [j][k] στοιχείο είναι αυτό που βρίσκεται στην j*5+k+1 θέση μνήμης, αφού για να βρούμε το στοιχείο αυτό σε σχέση με την αρχή περνάμε j συμπληρωμένες πεντάδες στοιχείων και k στοιχεία ακόμα. Το +1 τίθεται λόγω της αρίθμησης των στοιχείων των πινάκων στη C με αρχή το 0.

Στη γενική περίπτωση ενός πίνακα δύο διαστάσεων κάποιου τύπου (π.χ. ακεραίου) δηλωμένου ως εξής:

**int mat[L][M];**

**το στοιχείο mat[ j ][k] βρίσκεται στη θέση:**

**j*M+k+1** (2.1)

Παρατηρείστε ως μνημονικό κανόνα, ότι το μέγεθος της πρώτης διάστασης, το L δηλαδή, δεν εμφανίζεται στην πιο πάνω παράσταση.

## 2.4. ΠΑΡΑΔΕΙΓΜΑΤΑ ΧΡΗΣΗΣ ΔΙΔΙΑΣΤΑΤΟΥ ΠΙΝΑΚΑ.

Α) Τα παρακάτω for αρχικοποιούν (γεμίζουν) ένα δισδιάστατο πίνακα από το πληκτρολόγιο γραμμή προς γραμμή:

```
…….
int pin[5][7];
int k, j;
…….
for (k=0; k<5; k++)
    for (j=0; j<7; j++)
        scanf ("%d", &pin[k][j]);
```

Είναι προφανές ότι η αρχικοποίηση του ίδιου πίνακα στήλη προς στήλη θα γινόταν με απλή εναλλαγή των μεταβλητών, δηλαδή με τα δυο for ξανά, γραμμένα ως εξής:

```
for (j=0; j<7; j++)
      for (k=0; k<5; k++)
            scanf ("%d", &pin[k][j]);
```

Β) Στο παρακάτω πρόγραμμα ο πίνακας είναι 10 γραμμών και 2 στηλών. Διαβάζουμε συνεχώς float από το πληκτρολόγιο. Το διάβασμα συνεχίζεται όσο ο αριθμός που διαβάζεται για την πρώτη στήλη του πίνακα δεν είναι μηδέν.

```
#define  ROWS  10
#define  COLS   2
int main(void) {
      float  list[ROWS][COLS];
      int    k=0, j;

      do {
            puts ("Δώστε τιμές");
            scanf ("%f %f", &list[k][0], &list[k][1]); }
      while (list[k++][0] != 0);
      for (j=0; j<k-1; j++)
            printf ("%3.0f %8.2f", list[j][0], list[j][1]);
      return 1; }
```



## 2.5. ΔΙΔΙΑΣΤΑΤΟΙ ΠΙΝΑΚΕΣ ΚΑΙ ΔΕΙΚΤΕΣ.

Έστω ο πίνακας ακεραίων zippo, δηλωμένος ως εξής:

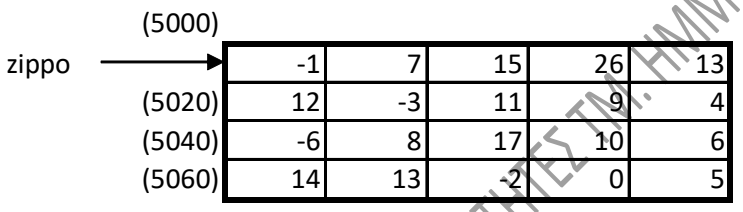
```
int zippo[4][5];
```

Ο zippo είναι πίνακας 4 γραμμών, 5 στηλών, δηλαδή **πίνακας 4 στοιχείων, κάθε ένα από τα οποία είναι πίνακας 5 στοιχείων.** Έστω ότι ο zippo έχει αποθηκευτεί στη μνήμη όπως φαίνεται στο σχ. 2.2 (ξεκινώντας από τη θέση μνήμης 5000).

Το όνομα ενός πίνακα είναι και δείκτης στο πρώτο στοιχείο του. zippo είναι όνομα πίνακα, άρα και δείκτης, άρα ο δείκτης zippo δείχνει το πρώτο στοιχείο του πίνακα, δηλαδή το στοιχείο zippo[0]. Όμως, zippo[0] είναι πίνακας 5 θέσεων ακεραίων. Ο zippo[0] είναι πίνακας ακεραίων. Το όνομά του είναι και δείκτης στο πρώτο στοιχείο του, άρα δείκτης σε ακέραιο και συγκεκριμένα στο πρώτο στοιχείο του, δηλαδή στο zippo[0][0]. Άρα:

**Ο zippo δείχνει στο zippo[0] (πίνακας ακεραίων), το δε zippo[0] δείχνει στο zippo[0][0] (ακέραιος),** άρα:

**Το zippo είναι δείκτης σε πίνακα ακεραίων, ή** μπορούμε να πούμε, **δείκτης σε δείκτη σε ακέραιο.**




Σχ. 2.2

(Οι αριθμοί σε παρενθέσεις σημαίνουν διεύθυνση μνήμης).

Θυμόμαστε ότι δείκτης σημαίνει διεύθυνση μνήμης. Ισχύει:

zippo == zippo[0] == &zippo[0][0]

Στο παράδειγμά μας όλα είναι ίσα με 5000.

Υπάρχει όμως μια **διαφορά**:

- Ο zippo[0] δείχνει σε ακέραιο (4 byte), άρα **αύξηση κατά 1 του δείκτη** σημαίνει μια μονάδα αποθήκευσης πιο κάτω στη μνήμη, του τύπου που δείχνει ο δείκτης, δηλαδή **1 ακέραιο πιο κάτω**, άρα:

  zippo[0] + 1 == 5004

- Ο zippo δείχνει σε πίνακα ακεραίων 5 θέσεων (20 byte) άρα **αύξηση κατά 1 του δείκτη** σημαίνει μια μονάδα αποθήκευσης πιο κάτω στη μνήμη, του τύπου που δείχνει ο δείκτης, δηλαδή **1 πίνακα ακεραίων πέντε θέσεων πιο κάτω**, άρα:

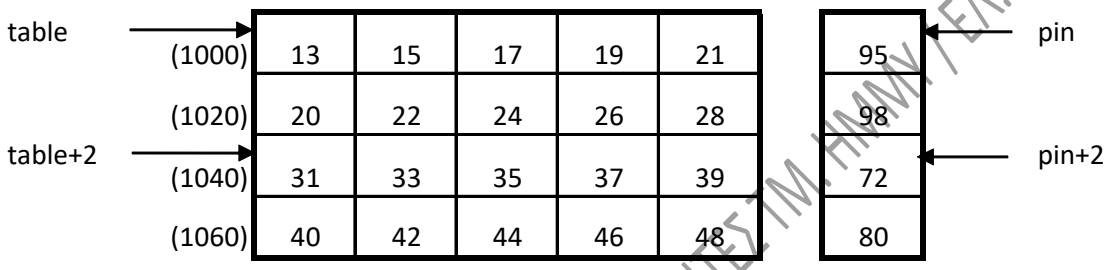
  zippo + 1 == 5020.

**Προσπέλαση των στοιχείων του πίνακα:**

Ένας τρόπος προσπέλασης των στοιχείων ενός δισδιάστατου πίνακα είναι ο κλασσικός, τον οποίο ήδη γνωρίσαμε. Δηλαδή, προκειμένου να αναφερθούμε για παράδειγμα στο στοιχείο της δεύτερης γραμμής και της πρώτης στήλης του ανωτέρω πίνακα zippo, θα γράψουμε zippo[1][0]. Τα ίδια θέματα μπορούμε να τα χειριστούμε και με άλλο τρόπο, κάνοντας χρήση και **αριθμητική δεικτών**.

Στο σχ. 2.3. παριστάνονται οι εξής πίνακες:

**int pin[4], table[4][5];**




Σχ. 2.3.

Θα μιλήσουμε για δείκτες στον πίνακα δύο διαστάσεων, κάνοντας παραλληλισμό με τον πίνακα μιας διάστασης. Έτσι:

- Ο pin+2 είναι δείκτης δυο θέσεις πιο κάτω από τον pin.
  - ◊ Ο table+2 είναι δείκτης δυο θέσεις πιο κάτω από τον table.
- Ο pin+2 είναι δείκτης σε ακέραιο.
  - ◊ Ο table+2 είναι δείκτης σε πίνακα ακεραίων.
- Τα περιεχόμενα του pin+2 είναι ένας ακέραιος.
  - ◊ Τα περιεχόμενα του table+2 είναι ένας πίνακας ακεραίων.

Αφού τα περιεχόμενα του table+2 είναι ένας πίνακας ακεραίων, είναι και δείκτης, δηλαδή διεύθυνση. Άρα, αν table == 1000, τότε:

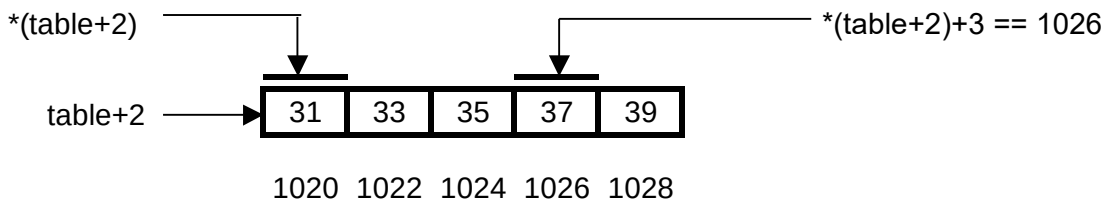
table+2 == 1040

*(table+2) == 1040

Ο *(table+2) είναι λοιπόν πίνακας ακεραίων, άρα δείκτης σε ακέραιο, άρα τα περιεχόμενά του είναι ένας ακέραιος, δηλαδή:

*(*(table+2)) == 31

Σχηματικά:




Σχ. 2.4.

Γενικά:

**table[j][k] == *(*(table+j) +k)** (2.2)

## 2.6. ΔΙΔΙΑΣΤΑΤΟΙ ΠΙΝΑΚΕΣ ΩΣ ΟΡΙΣΜΑΤΑ ΣΥΝΑΡΤΗΣΕΩΝ.

Στα προηγούμενα είδαμε πώς μπορούμε να περάσουμε ένα μονοδιάστατο πίνακα ως όρισμα σε μια συνάρτηση. Έστω τώρα ότι χρειαζόμαστε μια συνάρτηση, η οποία να χειρίζεται ένα δισδιάστατο πίνακα, άρα να δέχεται ως πληροφορία στοιχεία για τον πίνακα αυτόν. Υπάρχουν διάφοροι τρόποι για να αντιμετωπίσουμε αυτό το θέμα στην C, τρόποι δηλαδή για να «περάσουμε» τον δισδιάστατο πίνακα ή τις αναγκαίες πληροφορίες στη συνάρτηση. Δίνουμε ενδεικτικά παρακάτω τρία παραδείγματα:

Έστω ότι ένας δισδιάστατος πίνακας ακεραίων, ο matrix, έχει δηλωθεί ως εξής:

**int matrix [4][5];**

και υποτίθεται ότι έχει ήδη πάρει τιμές. Θέλουμε να γράψουμε μια συνάρτηση, την add3, η οποία θα προσθέτει τον αριθμό 3 σε όλα τα στοιχεία του πίνακα matrix.

**Α' τρόπος:**

Με διαδοχικές κλήσεις της συνάρτησης και ορίσματά της κάθε φορά τα εξής:

- Μια από τις γραμμές του δισδιάστατου πίνακα (δηλαδή ένα μονοδιάστατο πίνακα).
- Το πλήθος των στηλών του δισδιάστατου πίνακα, δηλαδή το πλήθος των στοιχείων κάθε ενός μονοδιάστατου πίνακα:

```
int main(void) {
      int matrix [4][5];
      int k;
      ……..
      for (k=0; k < 4; k++)
            add3_a (matrix[k], 5);
      ……..
}
void add3_a (int pin[ ], int num) {
      int k;
      for (k=0; k < num; k++)
            pin[k] += 3;
}
```

Η δήλωση της συνάρτησης θα είναι:

```
void add3_a (int [ ], int);
```

**Β' τρόπος:**

Θεωρώντας τον πίνακα matrix ως μονοδιάστατο πίνακα 20 ακεραίων. Στην περίπτωση αυτή θα μεταβιβάσουμε ως ορίσματα στην συνάρτηση ένα δείκτη στην αρχή του υποτιθέμενου μονοδιάστατου πίνακα και το πλήθος των στοιχείων του πίνακα, δηλαδή το πλήθος των ακεραίων του matrix:

```
int main(void) {
      int matrix [4][5];
      ……..
      add3_b (matrix[0], 4*5);
      ……..
}
void add3_b (int *pin, int num) {
      int k;
      for (k=0; k < num; k++)
            *(pin+k) += 3;
}
```

Ο κώδικας της συνάρτησης μπορεί εντελώς ισοδύναμα να είναι και εδώ αυτός του τρόπου Α'. Η δήλωση της συνάρτησης θα είναι:

```
void add3_b (int *, int);
```

**Γ' τρόπος:**

Περνώντας ως ορίσματα στην συνάρτηση τον matrix, δηλαδή ένα δείκτη στην αρχή του πίνακα και το πλήθος των γραμμών του πίνακα.

```
int main(void) {
      int matrix [4][5];
      ……..
      add3_c (matrix, 4);
      ……..
}
void add3_c (int pin[ ][5], int num) {
      int k, j;
      for (j=0; j < num; j++)
            for (k=0; k < 5; k++)
                  pin[ j ][ k ] += 3;
}
```

Παρατηρούμε ότι δεν χρειάζεται να πούμε στη συνάρτηση πόσες γραμμές έχει ο πίνακας. Η συνάρτηση θα «δούλευε» σωστά για οποιοδήποτε αριθμό γραμμών. Είναι όμως αναγκαίο να είναι γνωστός ο αριθμός των στηλών, οπότε η θέση κάθε στοιχείου μπορεί να βρεθεί ακριβώς σύμφωνα με τον τύπο (2.1). Η δήλωση της συνάρτησης θα είναι:

```
void add3_c (int [ ][5], int);
```

**Γ1' τρόπος:**

```
int main(void) {
      int matrix [4][5];
      ……..
      add3_c1 (matrix, 4);
      ……..
}
void add3_c1 (int (*pin) [5], int num) {
      int k, j;
      for (j=0; j < num; j++)
            for (k=0; k < 5; k++)
                  pin[ j ][ k ] += 3;
}
```

Παρατηρείστε τώρα την δήλωση της συνάρτησης:

```
void add3_c1 (int (*)[5], int);
```

**Άλλο παράδειγμα συνάρτησης με όρισμα διδιάστατο πινακα:**

Ο πίνακας stores που δηλώνεται παρακάτω, περιέχει στη μια στήλη τους κωδικούς και στην άλλη τις τιμές κάποιων προϊόντων (R συνολικά) μιας αποθήκης:

```
int stores [R][2];
```

Ζητούμε μια συνάρτηση, η οποία θα βρίσκει μέσα στον πίνακα τη θέση του προϊόντος με τη μέγιστη τιμή. Η συνάρτηση θα επιστρέφει αυτή τη θέση στη main( ), από όπου και θα εμφανίζεται στην οθόνη ο κωδικός του προϊόντος και η τιμή του. Έστω ότι το πλήθος των θέσεων του πίνακα όπου έχουν γίνει καταγραφές προϊόντων είναι num.

Ορίζουμε την συνάρτηση, που λέγεται **maxval( )**:

```
int maxval (int mstores[ ][2], int mnum) {
      int meg ;
      int    mk, pos=0;

      meg = mstores[0][1] ;
      pos = 0;
      for (mk=1 ; mk<mnum ; mk++)
            if (meg < mstores[mk][1]) {
                  meg = mstores[mk][1];
            pos = mk; }
      return pos;
}
```

Η κλήση της συνάρτησης και η εμφάνιση του αποτελέσματος στην οθόνη θα είναι κάτι σαν το παρακάτω:

```
val = maxval (stores, num);
printf ("Κωδικός : %d", stores[val][0]);
printf ("Τιμή : %d", stores[val][1]);
```

Η δήλωση της maxval( ) θα ήταν προφανώς:

```
int maxval (int [ ][C], int);
```



## 2.7. ΠΙΝΑΚΕΣ ΣΥΜΒΟΛΟΣΕΙΡΩΝ.

Στο πρόγραμμα που ακολουθεί γίνεται χρήση ενός πίνακα συμβολοσειρών. Η συμβολοσειρά είναι πίνακας χαρακτήρων, άρα **ένας πίνακας συμβολοσειρών είναι πίνακας πινάκων χαρακτήρων**. Όπως ξέρουμε άλλωστε, κάθε δισδιάστατος πίνακας μπορεί να θεωρηθεί ως πίνακας πινάκων.

Ο πίνακας pin παίρνει τιμές από το πληκτρολόγιο, σε κάθε γραμμή του δηλαδή διαβάζεται και αποθηκεύεται μια συμβολοσειρά. Υποθέτουμε εξ άλλου ότι ο μέγιστος αριθμός χαρακτήρων μιας συμβολοσειράς είναι 20, για τον λόγο δε αυτό ο αριθμός των στηλών του πίνακα τίθεται ίσος με 20.

Το πρόγραμμα στη συνέχεια ζητά να δοθεί από το πληκτρολόγιο ένας χαρακτήρας, ο ch, εμφανίζει δε στην οθόνη όλες τις συμβολοσειρές του πίνακα που αρχίζουν από τον ch.

```
#define R 10
#define C 20

int main(void) {
    char pin[R][C];
    int    k;

    for (k=0; k<R; k++)
          gets (pin[k]);
    puts("DWSE XARAKTHRA");
    ch = getche();
    for (k=0; k<R; k++)
          if (pin[k][0] == ch)
              puts (pin[k]);
    return 1; }
```

Έστω ότι ο πίνακας pin, μετά το διάβασμα συμβολοσειρών από το πληκτρολόγιο, έχει τα περιεχόμενα του σχ. 2.5. Όπου υπάρχει ? στον πίνακα αυτόν, σημαίνει ότι υπάρχει τυχαία τιμή:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| J | O | H | N | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| G | E | O | R | G | E | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| A | L | E | X | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| B | A | R | B | A | R | A | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| A | N | N | A | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| C | A | T | H | E | R | I | N | E | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| B | O | B | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| M | A | R | I | A | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| B | I | L | L | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| C | I | N | D | Y | \x0 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |

Σχ. 2.5.

Παρατηρούμε την **σπατάλη χώρου** για να αποθηκευτούν οι συμβολοσειρές του παραδείγματος. Άσχετα από τον αριθμό των χαρακτήρων, **ο χώρος που δεσμεύεται στη μνήμη είναι σταθερός**, δηλαδή 20 χαρακτήρες για κάθε συμβολοσειρά. Ακόμα χειρότερα, στην περίπτωση που απαιτηθεί περισσότερος χώρος για κάποια συμβολοσειρά, αυτός δεν είναι δυνατό να βρεθεί, θα δημιουργηθούν δε προβλήματα επικαλύψεων. Τέτοια προβλήματα λύνονται με τον δυναμικό χειρισμό της μνήμης, για τον οποίο θα μιλήσουμε σε επόμενη ενότητα.

## 2.8. ΕΙΔΙΚΕΣ ΜΟΡΦΕΣ ΠΙΝΑΚΩΝ.

### 2.8.1. ΣΥΜΜΕΤΡΙΚΟΙ ΠΙΝΑΚΕΣ.

**Συμμετρικός** λέγεται ένας δισδιάστατος πίνακας (π.χ. ο pin), αν είναι μεγέθους NxN και αν ισχύει:

**pin [ i ][ j ] == pin [ j ][ i ]**

για κάθε i και j. Ο παρακάτω πίνακας είναι συμμετρικός:

| | | | | |
|---|---|---|---|---|
| 6 | 3 | 7 | 5 | 10 |
| 3 | 8 | 2 | 9 | 13 |
| 7 | 2 | 4 | 1 | 11 |
| 5 | 9 | 1 | 0 | 6 |
| 10 | 13 | 11 | 6 | 7 |

Στην περίπτωση ενός πίνακα NxN, η κύρια διαγώνιος έχει μέγεθος N, το σύνολο των στοιχείων του είναι $N^2$, ενώ τόσο πάνω, όσο και κάτω από την κύρια διαγώνιο βρίσκεται ίσος αριθμός στοιχείων, έστω M. Άρα ισχύει ότι:

$$N^2 = 2 * M + N \qquad (2.3)$$

**Για την αποθήκευση στη μνήμη ενός NxN συμμετρικού πίνακα είναι αρκετή η αποθήκευση** μόνο του μισού πίνακα και της διαγωνίου, δηλαδή M+N στοιχείων, δηλαδή **$(N^2 + N) / 2$ στοιχείων**, σύμφωνα και με την πιο πάνω σχέση 2.3.

### 2.8.2. ΤΡΙΓΩΝΙΚΟΙ ΠΙΝΑΚΕΣ.

**Κάτω τριγωνικός** λέγεται ένας δισδιάστατος πίνακας (π.χ. ο mat), αν είναι μεγέθους NxN και αν ισχύει:

**mat [ i ][ j ] == 0**

για κάθε j > i (Με το i αναφερόμαστε στις γραμμές και με j στις στήλες).

Αντίστοιχα, **άνω τριγωνικός** λέγεται ένας δισδιάστατος πίνακας (π.χ. ο syn), αν είναι μεγέθους NxN και αν ισχύει:

**syn [ i ][ j ] == 0**

για κάθε i < j.

Ο παρακάτω πίνακας είναι κάτω τριγωνικός:

| | | | | |
|---|---|---|---|---|
| 6 | 0 | 0 | 0 | 0 |
| 3 | 8 | 0 | 0 | 0 |
| 7 | 2 | 4 | 0 | 0 |
| 5 | 0 | 1 | 5 | 0 |
| 11 | 5 | 8 | 4 | 2 |

ενώ ο επόμενος είναι άνω τριγωνικός:

| | | | | |
|---|---|---|---|---|
| 6 | 9 | 7 | 2 | 10 |
| 0 | 8 | 6 | 0 | 9 |
| 0 | 0 | 4 | 3 | -3 |
| 0 | 0 | 0 | 5 | 6 |
| 0 | 0 | 0 | 0 | 2 |

Για το πλήθος των στοιχείων ενός τριγωνικού πίνακα πάνω ή κάτω από την κύρια διαγώνιο ισχύουν όσα αναφέρθηκαν στην προηγούμενη παράγραφο για τους συμμετρικούς πίνακες, εφαρμόζεται δε η σχέση (2.3)



### 2.8.3. ΑΡΑΙΟΙ ΠΙΝΑΚΕΣ.

Ένας πίνακας λέγεται **αραιός**, όταν κατά ένα μεγάλο ποσοστό (συνήθως μεγαλύτερο από το 80%) τα στοιχεία του έχουν τιμή μηδέν. Τα παραπάνω ισχύουν για κάθε είδος πίνακα. Θα εξετάσουμε παρακάτω δύο τρόπους αποθήκευσης στη μνήμη ενός αραιού δισδιάστατου πίνακα:

Α’ τρόπος:

Έστω ο πίνακας:

$$\begin{matrix} 0 & 2 & 0 & 0 & 0 & 0 & 0 & 6 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 0 & -8 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \qquad (\Pi 1)$$

**Ο πίνακας αποθηκεύεται ως δυαδικός**, δηλαδή κάθε μη μηδενικό στοιχείο του έχει αντικατασταθεί με την τιμή 1, οπότε προκύπτει ο πίνακας (Π2):

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(Π2)

**Ο δυαδικός πίνακας συνοδεύεται από ένα μονοδιάστατο πίνακα**, τον:

[2 6 5 -3 7 9 -8 4] (Π3)

Ο δυαδικός πίνακας, ο Π2, αποθηκεύεται σε πολύ μικρό χώρο, αφού κάθε στοιχείο του μπορεί να αποθηκευτεί ως bit και όχι ως ακέραιος.

Έστω ότι ο αρχικός πίνακας (Π1) έχει M x N ακέραια στοιχεία. Για την αποθήκευσή του απαιτούνται M*N*4 byte (με δεδομένο ότι κάθε ακέραιος αποθηκεύεται σε 4 byte στη μνήμη). Αν ο πίνακας (Π1) έχει L μη μηδενικά στοιχεία, τότε για την αποθήκευση του (Π2) απαιτούνται M*N / 8 byte, για δε τον πίνακα (Π3) απαιτούνται 2L byte, συνολικά δηλαδή 2L + M * N / 8 byte.

Στο παράδειγμά μας, ο (Π1) χρειάζεται 5 * 8 * 4 = 160 byte στη μνήμη. Ο (Π2) χρειάζεται 5 * 8 / 8 byte, άρα 5 byte, ο δε (Π3) χρειάζεται 4 x 8 = 32 byte, δηλαδή οι (Π2) και (Π3) συνολικά 37 byte.



Β' τρόπος.

Σε αυτό τον τρόπο **κάθε μη μηδενικό στοιχείο** του πίνακα **αποθηκεύεται ως μια τριάδα αριθμών, οι οποίοι δείχνουν τη γραμμή, τη στήλη και την τιμή του στοιχείου**. Έτσι, ο (Π1) θα αποθηκευτεί όπως ο (Π4):

[0 1 2 0 7 6 1 0 5 2 1 -3 2 5 7 3 3 9 3 7 -8 4 0 4 ] (Π4)

(Θυμίζουμε ότι η πρώτη γραμμή και η πρώτη στήλη του πίνακα είναι αυτές με αύξοντα αριθμό μηδέν).

Αν ο πίνακας (Π1) έχει L μη μηδενικά στοιχεία, τότε για την αποθήκευση του (Π4) απαιτούνται 3 *4 * L byte.

## 2.9. ΠΙΝΑΚΕΣ ΠΕΡΙΣΣΟΤΕΡΩΝ ΑΠΟ ΔΥΟ ΔΙΑΣΤΑΣΕΩΝ.

Σε αντιστοιχία με τους μονοδιάστατους και δισδιάστατους πίνακες, δυο πίνακες ακεραίων τριών και τεσσάρων διαστάσεων για παράδειγμα, μπορούν να δηλωθούν αντίστοιχα ως εξής:

```
int pinax3 [3][5][7];
int pinax4 [5][6][8][3];
```

Οι πίνακες αυτοί χαρακτηρίζονται από τρεις και τέσσερεις ετικέτες αντίστοιχα. Έτσι, η προσπέλαση των στοιχείων τους γίνεται με τη χρήση των ετικετών αυτών. Αν ak και mk είναι δυο ακέραιες μεταβλητές, τότε οι παρακάτω αποδόσεις τιμών είναι συντακτικά σωστές:

```
ak = pinax3 [0][2][6];
mk = pinax4 [1][4][7][2];
```

Η αρχικοποίηση ενός πίνακα τριών διαστάσεων απαιτεί φυσικά ένα τριπλό for, του πίνακα τεσσάρων διαστάσεων απαιτεί ένα τετραπλό for κ.ο.κ. Έτσι, ο πίνακας pinax3 που δηλώθηκε πιο πάνω μπορεί να πάρει τιμές από το πληκτρολόγιο ως εξής:

```
for (k=0; k<3; k++)
    for (n=0; n<5; n++)
        for (m=0; m<7; m++)
            scanf ("%d", &pinax3[k][n][m]);
```

Η επισήμανση της θέσης αποθήκευσης στη μνήμη κάποιου στοιχείου ενός πίνακα τριών διαστάσεων, μπορεί να γίνει σε αντιστοιχία με την σχέση (2.1). Έτσι, για ένα πίνακα, τον meg, δηλωμένο ως εξής:

```
int meg[L][M][N];
```

το στοιχείο meg[x][y][z] θα βρισκόταν στη θέση:

```
x * (M * N) + y * N + z + 1
```
(2.4)

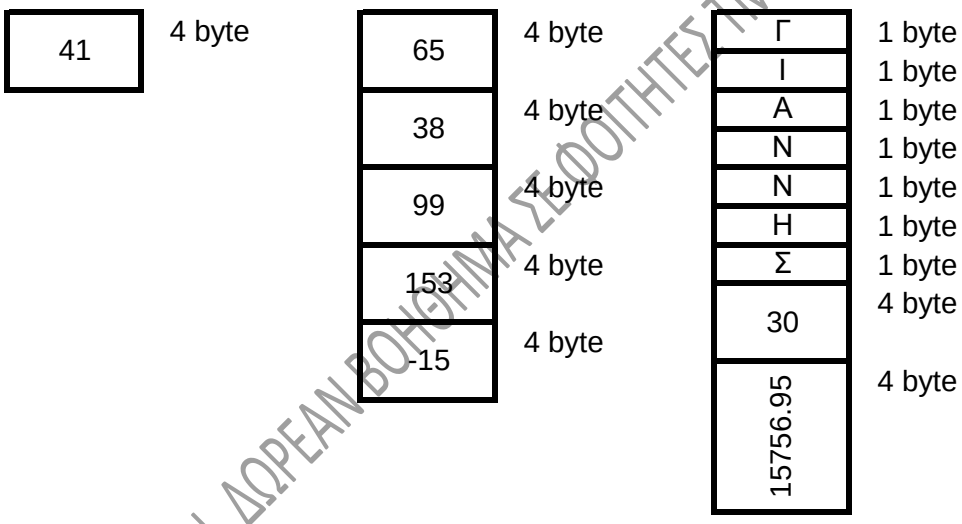
Παρατηρείστε ότι το μέγεθος της πρώτης διάστασης, το L δηλαδή, δεν εμφανίζεται και πάλι στην πιο πάνω παράσταση.

# ΚΕΦΑΛΑΙΟ 3

# ΔΟΜΕΣ και ΕΝΩΣΕΙΣ

## 3.1. ΕΙΣΑΓΩΓΗ ΣΤΙΣ ΔΟΜΕΣ - ΓΕΝΙΚΑ.

Συχνά θέλουμε να λειτουργούμε πάνω σε δεδομένα **διαφορετικού τύπου** μεταξύ τους, **τα οποία να χειριζόμαστε σαν ομάδα**. Σε αυτή την περίπτωση δεν είναι κατάλληλες οι μεμονωμένες μεταβλητές, αλλά ούτε και οι πίνακες.




Σχ. 3.1

Για παράδειγμα, θέλουμε σε ένα πρόγραμμα να κρατάμε και να αποθηκεύουμε δεδομένα, τα οποία αφορούν τα στοιχεία ενός σπουδαστή. Θέλουμε να κρατάμε για τον κάθε σπουδαστή το ονοματεπώνυμό του (συμβολοσειρά), το εξάμηνο φοίτησης (ακέραιος), το έτος γέννησης (ακέραιος) και το μέσο όρο βαθμολογίας του (float). Όλα αυτά τα στοιχεία αποτελούν μια ομάδα για τον κάθε σπουδαστή. Δεν μπορούμε να τα αποθηκεύσουμε σε ένα πίνακα, αφού ο πίνακας κρατάει στοιχεία ίδιου τύπου, πράγμα που εδώ δεν ισχύει. Θα μπορούσαμε να χρησιμοποιήσουμε βέβαια αρκετούς διαφορετικούς πίνακες (ένα πίνακα συμβολοσειρών για τα ονόματα, ένα πίνακα

ακεραίων για τα εξάμηνα, ένα άλλο πίνακα ακεραίων για τα έτη γέννησης και ένα πίνακα float για τους μέσους όρους βαθμολογίας. Αυτό είναι κάτι άβολο, αφού τα χαρακτηριστικά για τον κάθε σπουδαστή παύουν να υπάρχουν σαν ομάδα και βρίσκονται πλέον διασκορπισμένα σε διάφορους πίνακες.

Για να λύσουμε τέτοια προβλήματα χρησιμοποιούμε ένα ειδικό τύπο δεδομένων, την **δομή (structure)**. **Μια δομή αποτελείται από ένα πλήθος στοιχείων δεδομένων, τα οποία δεν χρειάζεται να είναι του ίδιου τύπου**. Στο παράδειγμα που αναφέραμε πιο πάνω, η δομή θα αποτελείται από τα στοιχεία του σπουδαστή, όπως τα αναφέραμε, αλλά και από όσα άλλα στοιχεία θέλαμε. Στο παραπάνω σχήμα 3.1 φαίνονται σχηματικά όπως θα αποθηκευόταν στη μνήμη μια απλή ακέραια μεταβλητή, ένας πίνακας ακεραίων και μια δομή.

Μια πολύ σημαντική χρήση των δομών είναι η δημιουργία **νέων μορφών δεδομένων**, όπως λίστες, δέντρα, σωροί κλπ, πολλά από τα οποία θα εξετάσουμε αναλυτικότερα στη συνέχεια. Οι περισσότερες από αυτές τις νέες μορφές σχηματίζονται από δομές **συνδεδεμένες μεταξύ τους**.



## 3.2. ΜΙΑ ΑΠΛΗ ΔΟΜΗ.

Το παρακάτω πρόγραμμα χρησιμοποιεί μια απλή δομή που περιέχει δύο στοιχεία δεδομένων: μια μεταβλητή τύπου ακεραίου, την num και μια μεταβλητή τύπου χαρακτήρα, την ch.

```
struct easy {                  // Περιγραφή των δομών του
    int num;                   // είδους easy
    char ch; };

int main(void) {
    struct easy dom;

    dom.num = 5;
    dom.ch  = 'A';
    printf("dom.num=%d, dom.ch=%c", dom.num,
                        dom.ch);
    return 1; }
```

Όταν εκτελεστεί το πρόγραμμα, στην οθόνη θα εμφανιστεί:

dom.num=5, dom.ch=A

**Σχόλια:**

1. Οι βασικοί τύποι δεδομένων της C (int, float κλπ) είναι γνωστοί στον compiler. Έτσι, χρησιμοποιώντας π.χ. έναν float, ο υπολογιστής ξέρει ότι θα δεσμευτεί γι' αυτόν χώρος 4 byte. Τέτοια «γνώση» δεν υπάρχει για τις δομές, αφού αυτές περιέχουν οποιοδήποτε πλήθος και είδος διαφορετικών στοιχείων. Έτσι, **πρέπει να πούμε στον compiler πώς είναι η δομή πριν χρησιμοποιηθούν μεταβλητές αυτού του τύπου**. Πρέπει δηλαδή να περιγράψουμε τη μορφή της δομής, να την μορφοποιήσουμε.

2. Η περιγραφή:

```
struct easy {
  int num;
  char ch; };
```

αναφέρεται σε ένα **νέο τύπο δεδομένων**, ο οποίος είναι **δομή** και είναι **τύπου easy**. Κάθε μεταβλητή αυτού του τύπου θα αποτελείται από δύο στοιχεία: μια ακέραια μεταβλητή και μια μεταβλητή τύπου χαρακτήρα. Θα λέμε σωστότερα ότι οι δομές του τύπου easy έχουν δυο **πεδία**. Σε όλες τις δομές αυτού του τύπου το ένα πεδίο θα είναι το πεδίο num της δομής, ενώ το άλλο θα είναι το πεδίο ch της δομής. Η παραπάνω περιγραφή δεν αναφέρεται σε συγκεκριμένες μεταβλητές, άρα **δεν δεσμεύει χώρο στη μνήμη**. **Λέει απλώς ποια είναι η μορφή των δεδομένων του είδους «δομή easy»**. Αυτό είναι αντίστοιχο με το εξής: λέγοντας για παράδειγμα int στις δηλώσεις, δεν δεσμεύεται χώρος. Δέσμευση γίνεται όταν αναφέρουμε και το όνομα της μεταβλητής.

3. Αφού περιγράψουμε την δομή, μπορούμε να δηλώσουμε μια ή περισσότερες μεταβλητές αυτού του τύπου. Στην δήλωση:

```
struct easy dom;
```

**δηλώνεται μια μεταβλητή**, η dom, που είναι του τύπου (του είδους) struct easy. **Εδώ δεσμεύεται χώρος** στη μνήμη. Δεσμεύεται αρκετός χώρος, ώστε να χωράει όλα τα στοιχεία της δομής, στην περίπτωσή μας δηλαδή **5 byte**, τέσσερα για τον ακέραιο και ένα για τον χαρακτήρα. Η δήλωση αυτή είναι για παράδειγμα το αντίστοιχο μιας δήλωσης, όπως η: int ak;

4. Έστω ότι θέλω να αναφερθώ στα στοιχεία της δομής και συγκεκριμένα στο πεδίο num της δομής dom. Αυτό είναι το:

```
dom.num
```

Οι εντολές:

```
dom.num = 8;
dom.ch = 'K';
```

δίνουν την τιμή 8 στο πεδίο num της dom και την τιμή 'K' στο πεδίο ch της ίδιας δομής. Δηλαδή ο **τελεστής τελείας (.) ενώνει το όνομα μεταβλητής δομής με ένα πεδίο της δομής**.

5. Όπως μπορεί να υπάρχουν περισσότερες από μια μεταβλητές τύπου int, float κλπ σε ένα πρόγραμμα, μπορεί επίσης να υπάρχει οποιοδήποτε πλήθος μεταβλητών ενός δεδομένου είδους δομής. Πιο κάτω δηλώνονται δύο τέτοιες μεταβλητές, οι dom και str:

```
struct easy dom;
struct easy str;
```

Οι εντολές:

```
dom.num = 6;
dom.ch = 'F';
str.num = 32;
str.ch = 'P';
```

δίνουν την τιμή 6 στο πεδίο num της dom, την τιμή 'F' στο πεδίο ch της dom, την τιμή 32 στο πεδίο num της str και την τιμή 'P' στο πεδίο ch της str.

6. Μπορούμε να συνδυάσουμε σε μια εντολή την περιγραφή του είδους της δομής και των μεταβλητών δομής. Οι δηλώσεις δηλαδή των dom και str της πιο πάνω παραγράφου, μπορούν να γραφούν και έτσι:

```
struct easy {
    int num;
    char ch;
} dom, str;
```



## 3.3. ΔΕΔΟΜΕΝΑ ΣΤΙΣ ΔΟΜΕΣ.

### 3.3.1. Εισαγωγή.

Στο παρακάτω πρόγραμμα οι δομές persa και persb παίρνουν τιμές από το πληκτρολόγιο, κατόπιν δε τα περιεχόμενά τους εμφανίζονται στην οθόνη:

```
struct person {
    char name[30];
    int code;
};
```

```
int main(void) {
    struct person persa, persb;
    char strcode[10];

    gets (persa.name);
    gets (strcode);
    persa.code = atoi (strcode);
    gets (persb.name);
    gets (strcode);
    persb.code = atoi (strcode);
    puts ("ΠΕΡΙΕΧΟΜΕΝΑ ΔΟΜΩΝ");
    printf ("%s\n", persa.name);
    printf ("%05d\n", persa.code);
    printf ("%s\n", persb.name);
    printf ("%05d\n", persb.code);
    return 1 ; }
```

Στο πρόγραμμα γίνεται χρήση της συνάρτησης **atoi( )**. Οι ακέραιοι διαβάζονται ως συμβολοσειρές, με την χρήση δε της συνάρτησης atoi( ) μετατρέπονται σε ακεραίους. Θυμίζουμε ότι η atoi( ) δέχεται ως **όρισμα** ένα δείκτη σε χαρακτήρα. Μετατρέπει σε ακέραιο την συμβολοσειρά που ξεκινάει από την θέση όπου δείχνει ο δείκτης και επιστρέφει αυτή την **ακέραια τιμή**. Άλλες αντίστοιχες συναρτήσεις είναι η **atof( )** που μετατρέπει μια συμβολοσειρά σε **double** και η **atol( )** που μετατρέπει μια συμβολοσειρά σε **long**. Παρατηρείστε ότι το διάβασμα των πεδίων code των persa και persb θα μπορούσε να γίνει και **με τη χρήση της scanf** ως εξής:

```
scanf ("%d\n", &persa.code);
scanf ("%d\n", &persb.code);
```

Το «μηδέν» στο **%05d** κάνει τον ακέραιο να γραφεί σε χώρο 5 διαστημάτων, ως γνωστό, με τα διαστήματα όμως που δεν χρησιμοποιούνται να εμφανίζονται στην οθόνη ως μηδέν.



### 3.3.2. Αρχικές τιμές και απόδοση τιμής.

Όπως και οι απλές μεταβλητές, οι δομές μπορούν να πάρουν **αρχικές τιμές κατά τη δήλωσή τους.** Π.χ.:

```
struct person persa = {"ΓΙΑΝΝΗΣ", 15};
struct person persb = {"ΝΙΚΟΣ", 10};
```

Εξ άλλου, η παρακάτω εντολή είναι έγκυρη, αν και δεν ίσχυε στις αρχικές εκδόσεις της C:

```
persa = persb;
```

Η δυνατότητα αυτή είναι χρησιμότατη αφού μπορούμε **να αναθέσουμε με μια εντολή τα στοιχεία μιας δομής σε μια άλλη ίδιου τύπου**, ακόμη κι αν πρόκειται για παράδειγμα για μεγάλους πίνακες.

## 3.4. ΦΩΛΙΑΣΜΕΝΕΣ ΔΟΜΕΣ.

Όπως υπάρχουν πίνακες πινάκων, έτσι μπορεί να υπάρχουν και **δομές που περιέχουν άλλες δομές**. Π.χ.:

```
struct person {
      char name[30];
      int code; };

struct couple {
      struct person chief;
      struct person memb;
      int  am; };

struct couple  first, second;
```

Η δομές του είδους couple είναι οργανωμένες έτσι, ώστε να περιέχουν τρία πεδία. Τα δύο από αυτά είναι **δομές του είδους person** και το τρίο είναι ένας ακέραιος Το πρώτο από τα πεδία της first (η οποία όπως βλέπουμε είναι μια μεταβλητή δομή τύπου couple) είναι το:

```
first.chief
```

Αυτό όμως είναι μεταβλητή δομή τύπου person, της οποίας το δεύτερο πεδίο για παράδειγμα είναι το:

```
first.chief.code
```

Μπορώ προφανώς να πω:

```
first.chief.code = 158;
```

Δηλαδή **η προσπέλαση των πεδίων των φωλιασμένων δομών γίνεται με συνεχόμενη χρήση του τελεστή τελεία (.)**.



## 3.5. ΔΟΜΕΣ ΚΑΙ ΣΥΝΑΡΤΗΣΕΙΣ.

**Μια συνάρτηση μπορεί να έχει τιμή επιστροφής ένα τύπο δομής.** Η συνάρτηση newpers( ) (βλ. παρακάτω), διαβάζει από το πληκτρολόγιο στοιχεία μιας μεταβλητής δομής και τα καταχωρεί σε μεταβλητές της main( ). Συγκεκριμένα, η συνάρτηση διαβάζει

τα στοιχεία της μεταβλητής και **επιστρέφει την τιμή που διάβασε, δηλαδή ολόκληρη τη μεταβλητή δομής**. Όπως λοιπόν μια συνάρτηση που επιστρέφει ακέραια τιμή είναι και αυτή τύπου int, έτσι και **μια συνάρτηση που έχει τιμή επιστροφής δομή, είναι και αυτή του ίδιου τύπου**.

```
#include <stdio.h>

struct person {
      char name[30];
      int code; };

struct person newpers( );      /* Δήλωση της newpers( ) */

void disppers(struct person);          /*Δήλωση της disppers( )*/

int main(void) {
      struct person persa;
      struct person persb;

      persa = newpers( );
      persb = newpers( );
      dispers (persa);
      dispers (persb);
      return 1; }

struct person newpers( ) {             /* Ορισμός newpers( ) */
      struct person temp;
      char  ch;

      gets (temp.name);
      scanf ("%d", &temp.code);
      return temp;
}

void disppers(struct person datom) /* Ορισμός disppers( ) */
{
      puts ("Στοιχεία προσώπου: ");
      printf ("Όνομα : %s\n", datom.name);
      printf ("Κωδικός : %05d\n", datom.code);
}
```

Στο παράδειγμά μας, η newpers( ) είναι τύπου struct person. Η temp είναι μια μεταβλητή δομή του τύπου person, ορισμένη ως τοπική μεταβλητή της συνάρτησης newpers( ). Μέσα στη συνάρτηση διαβάζονται τα περιεχόμενά της και τελικά η τιμή της είναι αυτή που επιστρέφεται στη main( ).

**Μια συνάρτηση μπορεί να δέχεται δομές ως ορίσματα**. Στην περίπτωση αυτή τα ορίσματα δηλώνονται με τον τρόπο που γνωρίζουμε ότι δηλώνεται οποιαδήποτε μεταβλητή. Η συνάρτηση disppers( ) στο πιο πάνω παράδειγμα δέχεται ως όρισμα μια δομή, της οποίας εμφανίζει τα στοιχεία στην οθόνη.

Και οι συναρτήσεις και το κυρίως πρόγραμμα πρέπει να ξέρουν τη μορφή της δομής person, για τον λόγο δε αυτό η περιγραφή της έχει δοθεί **εξωτερικά**, δηλαδή πριν από τη main( ), **έξω από όλες τις συναρτήσεις**.

## 3.6. ΠΙΝΑΚΕΣ ΔΟΜΩΝ.

Όπως δηλώνουμε πίνακες ακεραίων, πίνακες float κλπ, μπορούμε να δηλώσουμε και πίνακα, **κάθε στοιχείο** του οποίου **είναι μια δομή**. Η μεταβλητή pinax που δηλώνεται παρακάτω είναι ένας πίνακας δομών 30 θέσεων, κάθε μια από τις οποίες είναι του τύπου employee. Δεσμεύονται δηλαδή 30 θέσεις στη μνήμη, λέγονται και οι 30 με το όνομα pinax και **σε κάθε μια θέση θα μπει μια δομή** τύπου employee:

```
struct employee {
      char name[30];
      int    code;
      float pos; };

struct employee pinax[30];
```

Η παρακάτω εντολή διαβάζει από το πληκτρολόγιο μια συμβολοσειρά και την καταχωρεί στο στοιχείο name της τρίτης θέσης του πίνακα pinax:

```
gets (pinax[2].name);
```

## 3.7. ΔΕΙΚΤΕΣ ΚΑΙ ΔΟΜΕΣ.

Όπως ξέρουμε, κάθε δείκτης είναι στην πράξη η διεύθυνση μιας απλής μεταβλητής. Μέσω των δεικτών γνωρίζουμε λοιπόν τις διευθύνσεις των μεταβλητών στη μνήμη, καθώς και τις διευθύνσεις των πρώτων στοιχείων των πινάκων που έχουμε δηλώσει. Αντίστοιχα, οι δείκτες «κρατούν» και τις διευθύνσεις των δομών που δηλώνουμε.

Όταν έχουμε δηλώσει μια μεταβλητή δομή κάποιου είδους, άρα **όταν γνωρίζουμε το όνομά της**, η προσπέλαση των πεδίων της γίνεται με χρήση του **τελεστή τελεία (.)**. Όταν όμως θέλουμε **να προσπελάσουμε τα πεδία μιας δομής** όχι μέσω του ονόματός της (το οποίο πιθανόν και να μη γνωρίζουμε), αλλά **μέσω ενός δείκτη που δείχνει στην δομή**, τότε χρησιμοποιούμε τον **τελεστή βέλος (->)**.

Στο παράδειγμα που ακολουθεί έχει δηλωθεί μια μεταβλητή δομή, η dok, στην οποία προφανώς δείχνει ο δείκτης &dok. Στην ίδια δομή τοποθετείται να δείχνει και ο δείκτης ptr, ο οποίος έχει δηλωθεί ως δείκτης σε δομές του είδους simple.

```
struct simple {
      int    num;
      char ch; };
int main(void) {
      struct simple  dok;
      struct simple  *ptr;
      .............................
      ptr = &dok;
      ptr -> num = 303;
      ptr -> ch = 'Q';
      .............................
}
```

**Σχόλια:**

1. Στο πιο πάνω παράδειγμα, η εντολή:

   **ptr -> num = 303;**

   κάνει το πεδίο num της δομής, στην οποία δείχνει ο ptr, ίσο με 303.

2. Ο συμβολισμός:

   **ptr.num**

   προφανώς δεν είναι έγκυρος, διότι αριστερά του τελεστή τελεία (.) πρέπει να υπάρχει όνομα δομής, ενώ ο ptr είναι δείκτης σε δομή.

3. Είναι σωστός ο συμβολισμός:

   **(*ptr).num**

   Μέσα στις παρενθέσεις υπάρχουν τα περιεχόμενα του ptr, δηλαδή μια μεταβλητή δομή, άρα σωστά χρησιμοποιείται ο τελεστής τελεία. Οι παρενθέσεις εξ άλλου είναι αναγκαίες, αφού ο τελεστής τελεία (.) έχει μεγαλύτερη προτεραιότητα από τον τελεστή *.

4. Αντίστοιχα είναι σωστός και ο συμβολισμός:

   **(&ptr) -> num**

   αφού μέσα στις παρενθέσεις υπάρχει τώρα ένας δείκτης σε δομή, άρα σωστά χρησιμοποιείται ο τελεστής βέλος. Οι παρενθέσεις και εδώ είναι αναγκαίες, αφού ο τελεστής βέλος (->) έχει μεγαλύτερη προτεραιότητα από τον τελεστή &.

**Παράδειγμα:**

Στο παρακάτω πρόγραμμα γίνεται απόδοση τιμών στα διάφορα πεδία μιας δομής:

```
struct person {
      char name[30];
      int code; };

struct couple {
      struct person chief;
      struct person memb; };

int main(void) {
      struct couple frost;
      struct couple *sand;

      sand = &frost;
      frost.chief.name = "ΓΙΑΝΝΗΣ";          /* 1 */
      frost.chief.code = 1999;               /* 2 */
      sand ->chief.name[0] = 'M';            /* 3 */
      sand ->chief.code = 1958;              /* 4 */
      frost.memb->name[1] = 'K';             /* 5 */
      strcpy (sand ->memb->name, "NIK");     /* 6 */
      return 1; }
```

Στο πρόγραμμα αυτό:

- Οι γραμμές με αριθμούς 2 και 4 είναι σωστές συντακτικά.
- Η γραμμή 1 είναι λάθος, αφού η αντιγραφή συμβολοσειράς σε συμβολοσειρά γίνεται με τη χρήση της συνάρτησης strcpy( ) και όχι με τη χρήση του τελεστή απόδοσης τιμής (=). Στο αριστερό μέλος της απόδοσης τιμής ο συμβολισμός είναι έγκυρος.
- Η γραμμή 3 είναι σωστή συντακτικά
- Η γραμμή 5 είναι λάθος, διότι το frost.memb που βρίσκεται στα αριστερά του name είναι δομή και όχι δείκτης σε δομή, άρα είναι λάθος η χρήση του τελεστή βέλους.
- Η γραμμή 6 είναι λάθος, διότι το sand->memb που βρίσκεται στα αριστερά του name είναι δομή και όχι δείκτης σε δομή, άρα είναι λάθος η χρήση του τελεστή βέλους πριν το name.



## 3.8. ΕΝΩΣΕΙΣ.

Χρησιμοποιούνται όπως και οι δομές για την **ομαδοποίηση μεταβλητών διαφορετικών τύπων**. Όπως ξέρουμε, οι δομές μας επιτρέπουν να χειριζόμαστε σα μια μονάδα ένα πλήθος διαφορετικών μεταβλητών, αποθηκευμένων σε διαφορετικές θέσεις

μνήμης. Οι ενώσεις μας επιτρέπουν να χειριζόμαστε **τον ίδιο χώρο μνήμης σαν** ένα πλήθος από **διαφορετικές μεταβλητές**, δηλαδή **ένα κομμάτι μνήμης αντιμετωπίζεται ως μεταβλητή ενός τύπου σε μια περίπτωση και μεταβλητή διαφορετικού τύπου σε άλλη περίπτωση**. Έτσι, μόνο ένα από τα μέλη (τα πεδία) της ένωσης μπορεί να χρησιμοποιηθεί κάθε φορά.

Μια ένωση περιγράφεται με τον ίδιο τρόπο που περιγράφεται και μια δομή, με τη διαφορά ότι **χρησιμοποιείται η λέξη-κλειδί union**, αντί για την λέξη-κλειδί struct. Η δήλωση εξ άλλου μιας μεταβλητής ένωσης γίνεται με αντίστοιχο τρόπο με αυτόν που γίνεται η δήλωση μιας μεταβλητής δομής.

Στο παρακάτω παράδειγμα το μέγεθος των ενώσεων του είδους mixed είναι **8 byte**, παρά το ότι περιέχει ένα ακέραιο και ένα double. Αυτό γιατί, οι δυο μεταβλητές καταλαμβάνουν τον ίδιο χώρο στη μνήμη, οπότε τελικά **δεσμεύεται ο χώρος που χρειάζεται η μεγαλύτερη** από αυτές. Στο παράδειγμα αυτό, στη γραμμή 13, το πρόγραμμα θα εμφανίσει στην οθόνη «σκουπίδια», διότι θα προσπαθήσει να γράψει ένα double σαν ακέραιο.

**Παράδειγμα:**

```
union mixed {
    int   incase;
    double dcase; };

int main(void) {
    union mixed unmet;

    printf ("MEGETHOS ENOTHTAS = %5d byte\n", sizeof(union mixed));
    unmet.incase = 159;
    printf ("AKERAIA METABLHTH = %5d\n", unmet.incase);
    scanf ("%lf", &unmet.dcase);
    printf ("DOUBLE METABLHTH = %8.2lf\n", unmet.dcase);
    printf ("AKERAIA METABLHTH = %5d\n", unmet.incase);              /* 13 */
    return 1; }
```

**Με τις ενώσεις χρησιμοποιείται ο τελεστής ->** με τον ίδιο τρόπο που χρησιμοποιείται και με τις δομές.

# ΚΕΦΑΛΑΙΟ 4

# ΔΥΝΑΜΙΚΗ ΔΕΣΜΕΥΣΗ ΜΝΗΜΗΣ

## 4.1. ΓΕΝΙΚΑ.

Η C διαθέτει συναρτήσεις για την **παραχώρηση χώρου μνήμης κατά τη διάρκεια της εκτέλεσης του προγράμματος**, μια διαδικασία γνωστή ως **δυναμική δέσμευση μνήμης**. Αυτός ο τρόπος δέσμευσης μνήμης έχει σημαντικά πλεονεκτήματα σε σχέση με τη **στατική δέσμευση μνήμης**, αυτή δηλαδή που γίνεται αυτόματα με τη δήλωση μεταβλητών, δομών, πινάκων κλπ. Στη στατική δέσμευση μνήμης πρέπει να γνωρίζουμε πόση ακριβώς μνήμη χρειαζόμαστε, ενώ στη δυναμική υπάρχει αλληλεπίδραση με το πρόγραμμα κατά την εκτέλεσή του, η δε αίτηση για χώρο μνήμης «υποβάλλεται» κατά τη διάρκεια της εκτέλεσης. Στατική δέσμευση έχουμε για παράδειγμα, με την δήλωση:

```
char sym[ ] = "ΚΑΛΗΜΕΡΑ";
```

οπότε δεσμεύεται αυτόματα χώρος 9 byte.

Η C χρησιμοποιεί τρεις συναρτήσεις για την δυναμική δέσμευση μνήμης. Αυτές είναι η malloc( ), η calloc( ) και η realloc( ). Ο ορισμός των συναρτήσεων αυτών βρίσκεται στο αρχείο επικεφαλίδας stdlib.h, άρα για να τις χρησιμοποιήσουμε είναι απαραίτητη η εντολή:

```
#include <stdlib.h>
```

Στο ίδιο αρχείο βρίσκεται και ο ορισμός της συνάρτησης free( ) που θα δούμε στη συνέχεια και η οποία χρησιμοποιείται για την αποδέσμευση χώρου μνήμης.



## 4.2. Η ΣΥΝΑΡΤΗΣΗ malloc( ).

Η malloc( ) δέχεται ως **όρισμα τον αριθμό των byte των οποίων ζητούμε τη δέσμευση.** Στη συνέχεια βρίσκει ένα ελεύθερο μπλοκ μνήμης μεγέθους όσο το πλήθος των byte που ζητήσαμε και **επιστρέφει τη διεύθυνση του πρώτου byte αυτού του χώρου**. Η τιμή επιστροφής της δηλαδή είναι ένας δείκτης στην αρχή του χώρου που δεσμεύτηκε.

Όπως γνωρίζουμε, η έννοια του δείκτη είναι πάντα συνδυασμένη με το είδος των περιεχομένων της μνήμης όπου δείχνει ο δείκτης. Το ερώτημα που προκύπτει λοιπόν είναι σε τι είδος δεδομένα δείχνει ο δείκτης που επιστρέφεται από την malloc( ), αφού η ίδια η malloc( ) δεν «γνωρίζει» τι θα υπάρχει στον χώρο που δεσμεύεται, για τι είδους δεδομένα θα χρησιμοποιηθεί ο χώρος αυτός. Εισάγουμε εδώ **ένα νέο είδος δείκτη**, τον **δείκτη σε void**. Αν θέλαμε να τον περιγράψουμε, θα λέγαμε ότι αυτός είναι ένας «δείκτης σε τίποτα» ή ένας «δείκτης σε ο,τιδήποτε», ένας κατά κάποιο τρόπο «γενικός δείκτης». **Η malloc( ) λοιπόν έχει τιμή επιστροφής δείκτη σε void, ενώ αν υπάρχει αδυναμία δέσμευσης μνήμης, η malloc( ) επιστρέφει ένα μηδενικό δείκτη (NULL).**

Στο παράδειγμα που ακολουθεί γίνεται χρήση της συνάρτησης malloc( ) και σε συνδυασμό με αυτήν γίνεται χρήση και του **τελεστή sizeof**.

Παράδειγμα:

```
struct xx {
      int    num;
      char ch; };

int main( ) {
      struct xx *ptr;
      ..........................................
      ptr = (struct xx*) malloc(sizeof(struct xx));
      ..........................................
}
```

Σχόλια:

α) Στο πιο πάνω παράδειγμα περιγράφεται ένα είδος δομής που λέγεται xx. Προκειμένου να δεσμευτεί χώρος για να αποθηκευτεί μια δομή αυτού του είδους, καλείται η malloc( ), η οποία επιστρέφει ένα δείκτη σε μια περιοχή της μνήμης αρκετή, ώστε να χωράει τα στοιχεία της δομής.

β) Το μέγεθος της μνήμης που χρειαζόμαστε πρέπει να το ξέρουμε κάθε φορά που καλούμε τη malloc( ). Αυτό το βρίσκουμε με τον **τελεστή sizeof, ο οποίος δίνει το μέγεθος σε byte του τελεστέου που βρίσκεται δεξιά του**. Ο τελεστέος μπορεί να είναι **ένας προσδιοριστής τύπου σε παρενθέσεις**, όπως για παράδειγμα:

```
k = sizeof (float);
```

όπου ζητούμε από τον τελεστή sizeof το μέγεθος σε byte των δεδομένων του είδους float. Όταν όμως αναφερόμαστε σε **συγκεκριμένη μεταβλητή** (και όχι γενικά σε είδος), της οποίας ζητούμε το μέγεθος σε byte, **η μεταβλητή αυτή**

**δεν τίθεται σε παρενθέσεις**. Τέτοια μεταβλητή θα μπορούσε να είναι ακόμη και ολόκληρος πίνακας. Π.χ.:

```
int meg;
double db;
……..
meg = sizeof db;
```

Γνωρίζοντας ότι ένας float αποθηκεύεται σε 4 byte, ενώ ένας double σε 8, στις πιο πάνω περιπτώσεις, τα k και meg έχουν τιμές 4 και 8 αντίστοιχα. Εξ άλλου, στο παράδειγμα της προηγούμενης σελίδας ζητούμε από τον τελεστή sizeof το μέγεθος σε byte των δεδομένων του είδους struct xx.

γ) Παρατηρούμε ότι **δεν έχουμε δηλώσει μεταβλητή δομή. Η μεταβλητή δημιουργείται από τη συνάρτηση malloc( )**. Το πρόγραμμα δεν ξέρει πώς λέγεται η μεταβλητή, ξέρει όμως πού βρίσκεται μέσα στη μνήμη, αφού η συνάρτηση malloc( ) έχει επιστρέψει ένα δείκτη σε αυτή. Προκειμένου να μπορούμε στη συνέχεια να προσπελάσουμε τη μεταβλητή δομή που δημιουργήθηκε, **αποδίδουμε την τιμή που επέστρεψε η malloc( ) σε ένα δείκτη**, τον ptr και η προσπέλαση γίνεται πλέον μέσω του δείκτη αυτού. Η εντολή:

```
ptr = (struct xx*) malloc(sizeof(struct xx);
```

αναθέτει την τιμή του δείκτη (τη διεύθυνση δηλαδή) που επιστρέφεται από την malloc( ) στη μεταβλητή δείκτη ptr. Το όρισμα της malloc( ) είναι το μέγεθος της δομής. Ας σημειωθεί ότι αν δεν είχε γίνει η ανάθεση τιμής, ο χώρος που δεσμεύτηκε θα είχε χαθεί, αφού **νέα κλήση της malloc( ) θα είχε ως αποτέλεσμα τη δέσμευση νέου χώρου στη μνήμη** και όχι την προσπέλαση του ήδη δεσμευμένου.

δ) Θα δούμε τώρα τι είναι το (struct xx*), το οποίο βρίσκεται μπροστά από την malloc( ). Για κάθε δείκτη μέσα σε ένα πρόγραμμα πρέπει να έχουμε δυο πληροφορίες: τη διεύθυνση του στοιχείου που δείχνει ο δείκτης και τον τύπο του στοιχείου αυτού. Η malloc( ) επιστρέφει μια διεύθυνση, αλλά δεν δίνει πληροφορίες για τον τύπο. Γνωρίζουμε ότι στον χώρο που δεσμεύτηκε θα τοποθετηθεί μια δομή του είδους xx. Έτσι, για να μην υπάρξει πρόβλημα, **πρέπει να σιγουρευτούμε ότι η τιμή που επιστράφηκε από την malloc( ) είναι του τύπου "δείκτης σε struct xx"**, αφού τέτοιου είδους στοιχεία δημιουργήσαμε και τέτοια στοιχεία θα προσπελαύνουμε. Με τον τρόπο που δόθηκε η εντολή, **ο δείκτης «αναγκάζεται»** να είναι της μορφής "δείκτης σε

struct xx", αποδίδεται δε στον ptr, ο οποίος είναι ίδιου τύπου. Ας θυμηθούμε πώς «αναγκάζεται» για παράδειγμα ένας float να γίνει int και πώς αντίστροφα ένας int «αναγκάζεται» να γίνει float:

```
int   ak;
float fp;
……..
ak = (int) fp;
.....
fp = (float) ak;
```

Οφείλουμε να αναφέρουμε ότι, σε πολλές σημερινές εκδόσεις της C η προσαρμογή τύπου της malloc γίνεται αυτόματα και έτσι δεν είναι απαραίτητο να την επιβάλλουμε εμείς. Για λόγους συμβατότητας πάντως, καλό είναι να κάνουμε μόνοι μας την προσαρμογή τύπου.

Ομοίως, για να δεσμεύσουμε στην μνήμη 80 byte και στον χώρο αυτό να τοποθετήσουμε 20 ακεραίους, θα καλούσαμε την malloc( ) ως εξής:

```
int *dptr;
.....
dptr = (int *) malloc (80);
```

ενώ για την τοποθέτηση 80 χαρακτήρων σε χώρο 80 byte στην μνήμη θα είχαμε τα παρακάτω:

```
char *cptr;
.....
cdptr = (char *) malloc (80);
```



## 4.3. Η ΣΥΝΑΡΤΗΣΗ calloc( ).

Η συνάρτηση calloc() **δεσμεύει** επίσης **μνήμη** την ώρα της εκτέλεσης ενός προγράμματος. Αντί να δεσμεύσει όμως ένα πλήθος από byte όπως η malloc( ), δεσμεύει ένα πλήθος στοιχείων μνήμης κάποιου τύπου. Δέχεται ως **ορίσματα δύο ακεραίους** και συγκεκριμένα **τον αριθμό των στοιχείων μνήμης** που θα δεσμευτούν, καθώς και **το πλήθος των byte ανά στοιχείο μνήμης.** Εάν η **δέσμευση χώρου** είναι **επιτυχής, μηδενίζει τα περιεχόμενα της μνήμης** που δεσμεύει **και επιστρέφει ένα δείκτη στην αρχή** αυτού του χώρου. Όπως και στην malloc( ), η τιμή επιστροφής της είναι **δείκτης σε void.** Σε **αδυναμία δέσμευσης** μνήμης επιστρέφει ένα **μηδενικό δείκτη (NULL).**

Παράδειγμα:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *ptr, k, num;
    ……..
    scanf("%d", &num);
    ptr = (int*) calloc(num, sizeof(int));
    for (k=0; k<num; k++)
        printf("%5d\n", *(ptr+k));
    ……
}
```

Στο πιο πάνω παράδειγμα, η calloc( ) δεσμεύει χώρο για να τοποθετηθούν num ακέραιοι. Με δεδομένο ότι ο ptr δείχνει τον πρώτο ακέραιο αυτού του χώρου, η for που ακολουθεί εμφανίζει τα περιεχόμενα του χώρου που δεσμεύτηκε στην οθόνη. Σύμφωνα με όσα προαναφέρθηκαν, τα περιεχόμενα αυτά είναι ίσα με μηδέν.

Παρατηρείστε ότι, όπως η malloc( ), έτσι και **η calloc( ) χρειάζεται προσαρμογή τύπου**. Στο παράδειγμα, αυτό γίνεται με το (int *).



## 4.4. Η ΣΥΝΑΡΤΗΣΗ realloc( ).

Η συνάρτηση realloc( ) **τροποποιεί την ποσότητα μνήμης** που είχε προηγουμένως δεσμευτεί από κλήση είτε της malloc( ) είτε της calloc( ). Προσοχή! Δεν τροποποιεί χώρο που δεσμεύτηκε για παράδειγμα με την δήλωση ενός πίνακα. Η realloc( ) δέχεται **δύο ορίσματα**:

- Το πρώτο είναι ένας **δείκτης στη θέση μνήμης**, της οποιας το μέγεθος θέλουμε να τροποποιήσουμε. Αυτός ο δείκτης μπορεί να είναι η **τιμή την οποία επέστρεψε μια κλήση της malloc( ) ή της calloc( )**, δηλαδή ένας δείκτης σε void.
- Το δεύτερο είναι το νέο **πλήθος των byte που θα δεσμευτούν**. Το μέγεθος της νέας θέσης στη μνήμη πρέπει να υπολογιστεί πριν από την κλήση της realloc( ).

Ανάλογα με τη διαθεσιμότητα της μνήμης, το αποτέλεσμα της realloc( ) είναι ένα από τα επόμενα:

- Αν τα byte που θα δεσμευτούν είναι **λιγότερα** από τα ήδη δεσμευμένα, τότε η realloc( ) δεσμεύει με **επιτυχία** τον χώρο.
- Αν τα byte που θα δεσμευτούν είναι **περισσότερα** από τα ήδη δεσμευμένα, υπάρχουν δύο ενδεχόμενα:
  α) **Ο επιπλέον χώρος είναι διαθέσιμος συνεχόμενα** με τον ήδη δεσμευμένο. Τότε η realloc( ) **δεσμεύει με επιτυχία** τον χώρο.
  β) Ο **επιπλέον χώρος δεν είναι διαθέσιμος συνεχόμενα** με τον ήδη δεσμευμένο. Τότε η realloc( ) αναζητά σε άλλη θέση στη μνήμη ένα ενιαίο χώρο (μπλόκ) όσο ο συνολικά ζητούμενος από εμάς, δεσμεύει τον χώρο αυτόν, τα δε υπάρχοντα δεδομένα **αντιγράφονται** στη νέα θέση. Το παλιό μπλόκ μνήμης απελευθερώνεται και η συνάρτηση επιστρέφει ένα δείκτη στην αρχή του νέου μπλόκ.
- Σε **αδυναμία δέσμευσης** μνήμης η realloc( ) επιστρέφει ένα **μηδενικό δείκτη (NULL)**.

Η realloc( ) χρειάζεται **προσαρμογή τύπου** όπως η calloc( ) και η malloc( ).

Στο παράδειγμα που ακολουθεί γίνεται χρήση της realloc( ), προκειμένου να επεκταθεί ο χώρος που δεσμεύτηκε αρχικά για την αποθήκευση μιας συμβολοσειράς.

```
int main(void) {
    char buf[80], *mes;

    puts("Enter text");
    gets(buf);
    mes = (char *) malloc(strlen(buf)+1);
    strcpy(mes,buf);
    puts(mes);
    puts("Enter another line");
    gets(buf);
    mes = (char *) realloc(mes, (strlen(mes)+strlen(buf)+1));
    strcat(mes,buf);
    puts(mes);
    return 1; }
```

## 4.5. ΠΙΝΑΚΕΣ ΔΕΙΚΤΩΝ.

Στο παράδειγμα που ακολουθεί εισάγεται η έννοια και παρουσιάζεται η χρησιμότητα του **πίνακα δεικτών**, ενώ γίνεται επίσης χρήση της malloc( ).

Το πρόγραμμα διαβάζει συνεχώς συμβολοσειρές από το πληκτρολόγιο, 80 χαρακτήρων το πολύ κάθε μια. Το διάβασμα συνεχίζεται μέχρι να διαβαστούν συνολικά 100 συμβολοσειρές ή μέχρι να δοθεί μια μηδενική συμβολοσειρά, δηλαδή κάποια με τον χαρακτήρα '\x0' ως πρώτο στοιχείο της.

Οι συμβολοσειρές που διαβάζονται υποτίθεται ότι είναι τίτλοι βιβλίων. Δημιουργείται έτσι ένας κατάλογος βιβλίων, τα περιεχόμενα του οποίου εμφανίζονται στην οθόνη ένα προς ένα στο τέλος του προγράμματος.

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

#define LINE 81
#define MAX 100

int main(void) {
    char temp[LINE], ch;
    char *ps[MAX];
    int index=0, k;

    puts("ΔΩΣΤΕ ΤΙΤΛΟΥΣ ΒΙΒΛΙΩΝ");
    gets(temp);
    while ((index < MAX) && (temp[0] != '\x0')) {
        ps[index] = (char *) malloc(strlen(temp)+1);
        strcpy (ps[index], temp);
        index++;
        gets(temp); }
    putchar('\n');
    puts("ΛΙΣΤΑ ΒΙΒΛΙΩΝ");
    for (k=0; k<index; k++)
        puts(ps[k]);
    return 1; }
```



**Σχόλια:**

1. Διαβάζοντας την παρακάτω δήλωση από το τέλος προς την αρχή,:

   ```
   char *ps[MAX];
   ```

   καταλαβαίνουμε ότι ο ps δεν είναι πίνακας χαρακτήρων, αλλά **πίνακας δεικτών σε χαρακτήρες**. Σε κάθε θέση του δηλαδή υπάρχει ένας δείκτης, ο οποίος δείχνει σε

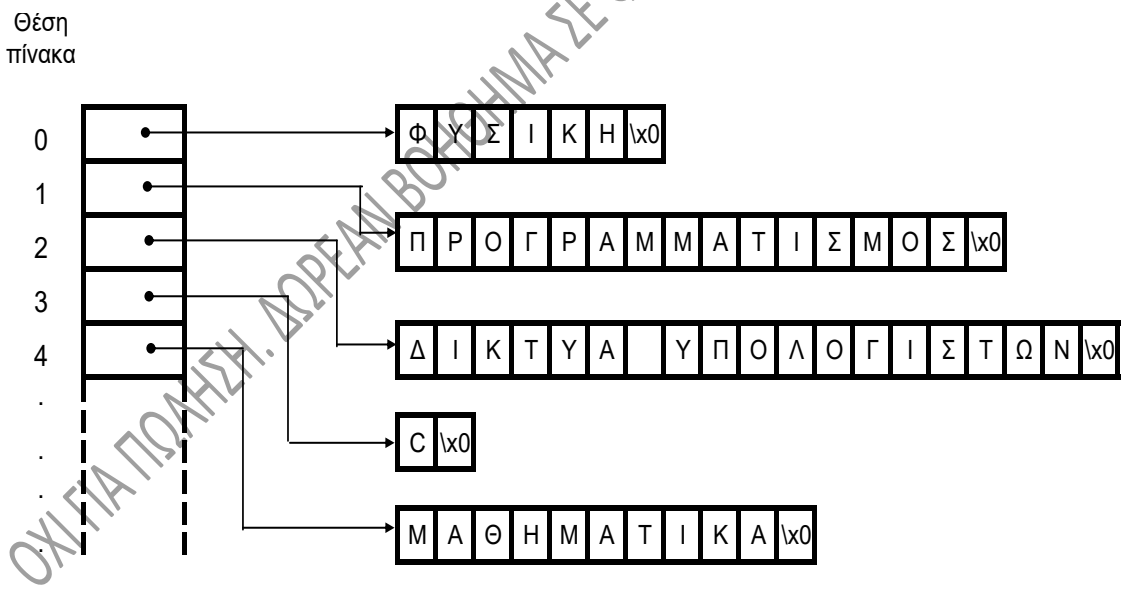
ένα χαρακτήρα (άρα μπορεί να δείχνει και στον πρώτο χαρακτήρα μιας συμβολοσειράς).

2. Κάθε συμβολοσειρά που διαβάζεται τοποθετείται αρχικά στον πίνακα χαρακτήρων temp. Στη συνέχεια η malloc( ) βρίσκει χώρο στη μνήμη για να χωρέσει η temp, καθώς και ένας χαρακτήρας ακόμη (το '\x0'). Η malloc( ) επιστρέφει ένα δείκτη σε αυτή τη θέση μνήμης, ο δε δείκτης της αντίστοιχης θέσης του πίνακα ps δείχνει κι αυτός -μετά την απόδοση τιμής- στον χώρο μνήμης που δεσμεύτηκε.

3. Είναι προφανές ότι οι δείκτες του πίνακα ps δείχνουν σε περιοχές μνήμης, στην αρχή συμβολοσειρών, οι οποίες **δεν είναι κατ' ανάγκην του ίδιου μεγέθους όλες**. Έτσι, αν για παράδειγμα οι πέντε πρώτοι τίτλοι βιβλίων που διαβάστηκαν είναι:

ΦΥΣΙΚΗ
ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ
ΔΙΚΤΥΑ ΥΠΟΛΟΓΙΣΤΩΝ
C
ΜΑΘΗΜΑΤΙΚΑ

μετά την εκτέλεση του προγράμματος στη μνήμη θα έχουμε την παρακάτω κατάσταση:




Σχ. 4.1

Συγκρίνετε την παραπάνω εικόνα με το σχ. 2.5, όπου στη μνήμη βρίσκεται αποθηκευμένος ένας πίνακας συμβολοσειρών. Η **σπατάλη** που συναντούμε εκεί, **εδώ δεν υπάρχει**, αφού για κάθε συμβολοσειρά δεσμεύεται ακριβώς ο χώρος που απαιτείται.

4. Το πρόγραμμα τερματίζεται όταν πληκτρολογήσουμε MAX τίτλους βιβλίων ή όταν δώσουμε τίτλο χωρίς περιεχόμενο, δηλαδή εάν δώσουμε το <Enter>, οπότε ο πρώτος χαρακτήρας του τίτλου αυτού είναι το '\x0'.

5. Η συνάρτηση malloc() περιγράφεται στο **αρχείο επικεφαλίδας stdlib.h**.

6. Αν χρησιμοποιήσουμε την calloc( ) αντί για την malloc( ), η εντολή:

```
ps[index] = (char *) malloc(strlen(temp)+1);
```

θα αντικατασταθεί ως εξής:

```
int mk;
……….
mk = strlen(temp)+1;
ps[index] = (char *) calloc(mk, sizeof(char));
```

## 4.6. ΔΗΜΙΟΥΡΓΙΑ ΔΥΝΑΜΙΚΩΝ ΠΙΝΑΚΩΝ.

Με την χρήση των συναρτήσεων δυναμικής δέσμευσης μνήμης, μπορούμε να δημιουργήσουμε **«δυναμικούς πίνακες»**. Με αυτό εννοούμε «πίνακες», των οποίων το μέγεθος δεν είναι γνωστό εκ των προτέρων, δεν έχει δηλαδή δηλωθεί στον χώρο δήλωσης μεταβλητών. Είναι προφανές ότι ο όρος «πίνακας» χρησιμοποιείται εδώ καταχρηστικά, σε σχέση με όσα ήδη γνωρίζουμε, αφού ο πίνακας είναι στατική δομή, δηλαδή ο χώρος που καταλαμβάνεται γι' αυτόν είναι δεδομένος από την αρχή του προγράμματος και μέχρι την λήξη του.

Στο παράδειγμα που ακολουθεί εκμεταλλευόμαστε το γεγονός ότι η δήλωση ενός πίνακα συνεπάγεται ότι έχουμε και δημιουργία ενός δείκτη, ο οποίος έχει το ίδιο όνομα με το όνομα του πίνακα. Με τη χρήση της malloc δεσμεύεται χώρος για n ακεραίους:

```
int k, n, num, *pin;
.....................
scanf("%d", &n);
pin = (int *) malloc (n*sizeof(int));
for (k=0; k<n; k++) {
      printf("pin[%d]: \n", k);
      scanf("%d", &num);
      pin[k]=num; }
```

Στο επόμενο παράδειγμα γίνεται αντίστοιχα δυναμική δημιουργία ενός διδιάστατου πίνακα ακεραίων nxm:

```
int k, j, n, m,  num, **mat;
.....................................
scanf("%d%d", &n, &m);
mat = (int **)malloc(n*sizeof(int *));
for(k=0; k<n; k++)
      mat[k] = (int *) malloc(m*sizeof(int));
for(j=0; j<n; j++) {
      for(k=0; k<m; k++) {
            printf("mat[%d][%d]: \n", j, k);
            scanf("%d", &num);
            mat[ j ][k]=num; } }
```

Η πρώτη malloc στο πρόγραμμα αυτό δεσμεύει χώρο για n δείκτες σε ακέραιο, πρακτικά δηλαδή κατασκευάζει ένα πίνακα n δεικτών. Με την πρώτη for δεσμεύεται n φορές χώρος m ακεραίων και καθένας από τους n δείκτες που δημιουργήθηκαν προηγουμένως, τοποθετείται να δείχνει σε ένα από τους n αυτούς χώρους. Προσέξτε την δήλωση του mat:

```
int **mat;
```

δηλαδή δείκτης σε δείκτη σε ακέραιο.

Η δεύτερη for γεμίζει τον «πίνακα», τον χώρο που δημιουργήσαμε δηλαδή, με ακέραιες τιμές.



## 4.7. Η ΣΥΝΑΡΤΗΣΗ free( ).

Η free( ) **ακυρώνει τη δέσμευση μνήμης** που έγινε προηγουμένως με την χρήση της calloc( ), της malloc( ) ή της realloc( ). Η free( ) δέχεται ως **όρισμα ένα δείκτη** στην περιοχή μνήμης που πρόκειται να αποδεσμευτεί.

Έτσι, στο παράδειγμα της παραπάνω παραγράφου 4.5, η εντολή:

```
free (ps[2]);
```

θα απελευθέρωνε την περιοχή μνήμης που αρχίζει στη διεύθυνση που καθορίζεται από το ps[2]. Στα περιεχόμενα του ps[2] παραμένει η ίδια διεύθυνση, όμως δεν έχουμε τη δυνατότητα να αποθηκεύσουμε εκεί πληροφορία, επειδή η μνήμη έχει απελευθερωθεί.

Το ps[2] μπορεί να χρησιμοποιηθεί ξανά στη συνέχεια αν καλέσουμε την calloc( ) ή τη malloc( ) και αποθηκεύσουμε τον δείκτη που επιστρέφεται από αυτήν στο ps[2].

Η διαχείριση της μνήμης πρέπει να γίνεται με κριτήριο την οικονομία σε ένα πρόγραμμα. Από ανεξέλεγκτη χρήση της μνήμης μπορούν εύκολα να προκύψουν προβλήματα, υπερχειλίσεις κλπ. Έτσι, μνήμη που δεν χρησιμοποιείται είναι καλό αλλά και αναγκαίο να αποδεσμεύεται.

# ΚΕΦΑΛΑΙΟ 5

# ΣΤΟΙΒΕΣ

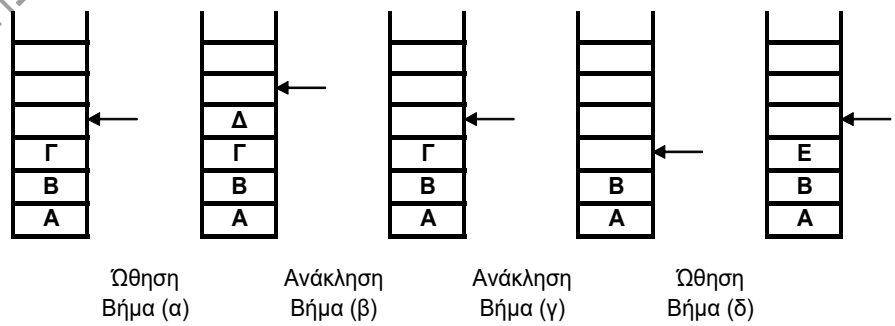
## 5.1. ΓΕΝΙΚΑ.

Στοίβα (**stack**) είναι μια **γραμμική διάταξη** στοιχείων, στην οποία τα νέα στοιχεία που συμπληρώνονται τοποθετούνται **στο ένα της άκρο** (**κεφαλή** της στοίβας, head). Τα στοιχεία που εξάγονται από τη στοίβα **εξάγονται** επίσης **μόνο από την κεφαλή**. Οι λειτουργίες λοιπόν που εκτελούνται σε μια στοίβα είναι μόνο οι εξής:

- **Ώθηση (push)** για την εισαγωγή στοιχείου.
- **Ανάκληση (pop)** για την εξαγωγή στοιχείου.

Ο τρόπος αυτός εισαγωγής και εξαγωγής στοιχείων λέγεται **LIFO** (Last In First Out), ακρωνύμιο που δηλώνει ότι **το τελευταίο στοιχείο που μπαίνει στη στοίβα είναι και το πρώτο που εξάγεται**. Η διάταξη θυμίζει μια στοίβα από πιάτα. Κάθε καινούργιο πιάτο που πλένεται τοποθετείται πάνω-πάνω στη στοίβα, ενώ το πιάτο που παίρνουμε για να το ξεπλύνουμε είναι αυτό που βρίσκεται στην κορυφή της στοίβας δηλαδή αυτό που τοποθετήθηκε πιο πρόσφατα. Δεν μπορούμε να πάρουμε κάποιο πιάτο από τη μέση της στοίβας, χωρίς να απομακρύνουμε όσα υπάρχουν πιο πάνω.

Στο παρακάτω σχήμα η στοίβα έχει ήδη τρία στοιχεία. Το βέλος δείχνει κάθε φορά την θέση της στοίβας στην οποία θα τοποθετηθεί κάποιο στοιχείο εάν γίνει ώθηση. Το στοιχείο Δ ωθείται στη στοίβα στο βήμα (α). Στα βήματα (β) και (γ) ανακαλούνται στοιχεία από τη στοίβα, ενώ στο βήμα (δ) ωθείται ξανά ένα στοιχείο στη στοίβα. Το στοιχείο που ανακαλείται από τη στοίβα -εάν δεν αποθηκευτεί κάπου αλλού- καταστρέφεται. Παρατηρείστε τα περιεχόμενα της στοίβας στα διάφορα βήματα:




Σχ. 5.1

## 5.2. ΥΛΟΠΟΙΗΣΗ ΣΤΟΙΒΑΣ.

Ο απλούστερος τρόπος για την υλοποίηση μιας στοίβας είναι με **την χρήση ενός πίνακα**, του οποίου όμως έχουν «απενεργοποιηθεί» θα λέγαμε κάποιες ιδιότητες και δυνατότητες: Γνωρίζουμε ότι σε ένα πίνακα μπορούμε να δώσουμε τιμή σε ένα οποιοδήποτε στοιχείο του ή να πάρουμε τα περιεχόμενα οποιασδήποτε θέσης του. Είναι προφανές, βάσει όσων είπαμε πιο πάνω, ότι κάτι τέτοιο δεν ισχύει στις στοίβες. **Μπορούμε λοιπόν να υλοποιήσουμε την στοίβα με την χρήση πίνακα, αλλά χωρίς τις παραπάνω δυνατότητες που μας παρέχει η χρήση των πινάκων. Τα στοιχεία θα τοποθετούνται μόνο στο ένα άκρο και θα ανακαλούνται μόνο από αυτό το άκρο**, για τον σκοπό δε αυτό θα γράψουμε ειδικές συναρτήσεις. Αν δεν διασφαλίσουμε τα παραπάνω, θα έχουμε φυσικά και πάλι πίνακα, όχι όμως στοίβα. Εάν η στοίβα του παραπάνω σχήματος 5.1 είχε υλοποιηθεί με την χρήση πίνακα, τότε μετά το βήμα (β) για παράδειγμα θα έχει ανακληθεί το στοιχείο Δ. Πρέπει να σημειώσουμε ότι το στοιχείο αυτό, μετά την ανάκλησή του, εξακολουθεί να υπάρχει στην ίδια θέση του πίνακα όπου βρισκόταν μέχρι τώρα. Όμως, επειδή ο πίνακας υλοποιεί μια στοίβα, δεν έχουμε τη δυνατότητα να πάρουμε αυτό το στοιχείο και πάλι.

Αργότερα θα αναφερθούμε στην υλοποίηση μιας στοίβας **με την χρήση συνδεδεμένης λίστας**, όπου ο απαιτούμενος χώρος δεσμεύεται με τη χρήση συναρτήσεων **δυναμικού χειρισμού μνήμης.** Και στην περίπτωση αυτή φυσικά πρέπει να τηρείται η διαδικασία LIFO που αναφέραμε ήδη, προκειμένου η λίστα να έχει τα χαρακτηριστικά της στοίβας.

Θεωρητικά, μια στοίβα **δεν έχει περιορισμό ως προς το μέγεθός** της, ως προς το πλήθος δηλαδή των στοιχείων που μπορούν να τοποθετηθούν σε αυτή. Στην πράξη όμως, ο χώρος που δεσμεύεται για τη στοίβα είναι περιορισμένος. Για τον λόγο αυτόν πριν την εισαγωγή κάποιου στοιχείου στη στοίβα απαιτείται έλεγχος για το εάν υπάρχει διαθέσιμος χώρος για την τοποθέτησή του.



## 5.3. ΩΘΗΣΗ ΚΑΙ ΑΝΑΚΛΗΣΗ

```
int stack [N];
int head;

void main(void) {
    int ak, apot;
    ……
    push (ak);
    apot = pop ( );
    …… }
```

```
void push (int elem) {
    if (head >= N)
        puts("Στοίβα πλήρης");
    else
        stack [head++] = elem; }

int pop( ) {
    head--;
    if (head < 0)
        {
            puts("Στοίβα κενή");
            return -1;
        }
    else
        return (stack[head]); }
```

Στο παραπάνω πρόγραμμα χρησιμοποιούνται δυο συναρτήσεις, οι push( ) και pop( ), οι οποίες υλοποιούν τους αλγορίθμους ώθησης και ανάκλησης στοιχείων αντίστοιχα. Στη στοίβα υποτίθεται ότι τοποθετούνται θετικές ακέραιες τιμές και για την υλοποίησή της χρησιμοποιείται ένας μονοδιάστατος πίνακας, ο stack, μεγέθους N. (Υλοποίηση με τη χρήση δυναμικού χειρισμού μνήμης θα γίνει σε επόμενη παράγραφο, αφού προηγουμένως αναφερθούμε στις συνδεδεμένες λίστες). Μια ακέραια μεταβλητή, η head έχει τιμή τέτοια, όση η τιμή της ετικέτας του πίνακα, στην οποία βρίσκεται **η κεφαλή της στοίβας**, εκεί δηλαδή όπου θα καταχωρηθεί στοιχείο εάν γίνει ώθηση. Μετά από συνεχείς ωθήσεις ή ανακλήσεις στοιχείων, η τιμή του head αλλάζει, έτσι ώστε να δηλώνει πάντα μια θέση του πίνακα πιο πάνω από τη μέγιστη κατειλημμένη:

**Σχόλια:**

1. Στο πιο πάνω πρόγραμμα, ο πίνακας που χρησιμοποιείται για τη στοίβα (stack) και ο ακέραιος που σημειώνει την κορυφή της (head) έχουν δηλωθεί ως **εξωτερικές μεταβλητές**. Από αυτό συμπεραίνουμε ότι, αφ' ενός μεν όλες οι θέσεις του πίνακα, όπως και ο head, έχουν αρχικά τιμή μηδέν, αφ' ετέρου δε ότι τις εν λόγω μεταβλητές μπορούν να «δουν» και να μεταβάλουν όλες οι συναρτήσεις του προγράμματος. Επειδή τον πίνακα stack και τον ακέραιο head πρακτικά χρησιμοποιούν μόνο οι συναρτήσεις push( ) και pop( ), **είναι κοινή πρακτική αυτά να δηλώνονται ως εξωτερικές μεταβλητές**.

2. Η push( ) τοποθετεί το όρισμά της (ένα ακέραιο εν προκειμένω) στη στοίβα. Σύμφωνα με τον αλγόριθμο λειτουργίας της, γίνεται πρώτα **έλεγχος υπερχείλισης** της στοίβας. Εάν δεν υπάρχει υπερχείλιση, τοποθετείται ο ακέραιος στη στοίβα και

μετά η μεταβλητή head αυξάνει κατά 1, σημειώνοντας έτσι τη θέση του πίνακα όπου θα γίνει νέα τοποθέτηση στοιχείου.

3. Η pop( ) πρώτα ελαττώνει την τιμή της μεταβλητής head κατά 1. Γίνεται **έλεγχος επάρκειας** της στοίβας προκειμένου να αποκλειστεί η περίπτωση της **κενής στοίβας**. Εάν η στοίβα έχει στοιχεία, άρα εάν δεν είναι κενή, η συνάρτηση επιστρέφει τιμή ίση με τα περιεχόμενα του πίνακα στη θέση head. Εάν η στοίβα είναι κενή, η συνάρτηση επιστρέφει τιμή ίση με -1 (θυμηθείτε ότι θεωρήσαμε ότι στην στοίβα αποθηκεύονται μόνο θετικοί ακέραιοι).

4. Είναι προφανές ότι η στοίβα θα μπορούσε να έχει οργανωθεί έτσι, ώστε κατά την ώθηση να έχουμε πρώτα αύξηση του head και μετά τοποθέτηση στοιχείου, ενώ κατά την ανάκληση να παίρνουμε το στοιχείο από τη θέση head του πίνακα. Η λειτουργία της στοίβας δεν θα άλλαζε, απλώς οι συναρτήσεις και οι έλεγχοι θα ήσαν λίγο διαφορετικά.

## 5.4. ΕΦΑΡΜΟΓΕΣ.

**1.** Οι στοίβες έχουν ευρύτατη εφαρμογή στους υπολογιστές. Ένα από τα πιο χαρακτηριστικά παραδείγματα είναι στην περίπτωση κλήσης υποπρογραμμάτων. **Σε κάθε κλήση υποπρογράμματος αποθηκεύεται η διεύθυνση επιστροφής σε μια στοίβα**, απ' όπου ανακαλείται όταν τελειώσει η εκτέλεση του υποπρογράμματος και πρέπει να επιστρέψει η εκτέλεση του προγράμματος στο σημείο όπου διακόπηκε. Έτσι, ακόμα και μετά από πολλές διαδοχικές κλήσεις υποπρογραμμάτων, ο έλεγχος επιστρέφει κάθε φορά στο σημείο της τελευταίας κλήσης.

**2.** **Μετατροπές μεταξύ ένθετης και μεταθεματικής μορφής:**

Δυο από τις πολύ σημαντικές εφαρμογές είναι η μετατροπή αριθμητικών παραστάσεων από την **ένθετη μορφή (infix)** στη **μεταθεματική (postfix ή reverse-Polish / αντίστροφος Πολωνικός συμβολισμός)**, καθώς και το αντίστροφο, δηλαδή ο υπολογισμός της τιμής των παραστάσεων που είναι εκφρασμένες με τον τρόπο αυτόν. **Ένθετη είναι η συνήθης μορφή των παραστάσεων, όπου οι τελεστές βρίσκονται μεταξύ των τελεστέων στις πράξεις. Στον αντίστροφο Πολωνικό συμβολισμό οι τελεστές τοποθετούνται μετά τους τελεστέους.** Έτσι, στις παρακάτω παραστάσεις:

Η ένθετη a + b γράφεται ab+ ως μεταθεματική.

| | | | | |
|---|---|---|---|---|
| Η ένθετη | (a + b) * c | γράφεται | ab+c* | ως μεταθεματική. |
| Η ένθετη | a * b + c * d | γράφεται | ab*cd*+ | ως μεταθεματική. |

Όλα τα μεταφραστικά προγράμματα υπολογίζουν την τιμή των παραστάσεων ένθετης μορφής που δέχονται στην είσοδό τους, αφού προηγουμένως τις μετατρέψουν στην ισοδύναμη μεταθεματική μορφή.

**α) Ένθετη σε μεταθεματική μορφή:**

Η μετατροπή γίνεται με τη χρήση μιας στοίβας, όπου αποθηκεύουμε τους τελεστές και ακολουθώντας τους εξής κανόνες:

- Σαρώνεται η παράσταση από τα αριστερά προς τα δεξιά.
- Αν συναντήσουμε τελεστέο (π.χ. κάποια μεταβλητή), τον τοποθετούμε στο δεξί μέρος της ήδη υπάρχουσας μεταθεματικής παράστασης.
- Αν συναντήσουμε τελεστή, τον συγκρίνουμε με τον τελεστή που υπάρχει στην κεφαλή της στοίβας. Διακρίνουμε τις εξής περιπτώσεις:
  - Αν ο τελεστής που συναντήσαμε έχει μεγαλύτερη προτεραιότητα από τον ευρισκόμενο στην κεφαλή της στοίβας, τότε ο τελεστής ωθείται στην στοίβα.
  - Αν ο τελεστής που συναντήσαμε έχει μικρότερη προτεραιότητα από τον ευρισκόμενο στην κεφαλή της στοίβας ή ίση με αυτόν, τότε ανακαλούνται από την στοίβα οι τελεστές με προτεραιότητα μεγαλύτερη ή ίση από τον τελεστή που συναντήσαμε στην παράσταση και τοποθετούνται στο δεξί μέρος της ήδη υπάρχουσας μεταθεματικής παράστασης. Μετά ο τελεστής που συναντήσαμε στην παράσταση ωθείται στην στοίβα.
  - Αν συναντήσουμε αριστερή παρένθεση, αυτή ωθείται στην στοίβα.
  - Αν συναντήσουμε δεξιά παρένθεση, τότε ανακαλούνται όλοι οι τελεστές μέχρι την αριστερή παρένθεση και τοποθετούνται στο δεξί μέρος της ήδη υπάρχουσας μεταθεματικής παράστασης. Οι δυο παρενθέσεις (αριστερή και δεξιά) αγνοούνται.

Σύμφωνα με τα παραπάνω, επιχειρούμε τη μετατροπή της παράστασης:

((A – B) + C) * ((D + E) * (F – G))

στην αντίστοιχη μεταθεματική. Στην πρώτη στήλη του πίνακα 5.1. φαίνεται ο τελεστής ή τελεστέος που συναντούμε κατά το διάβασμα, την σάρωση της παράστασης. Στην δεύτερη στήλη φαίνονται τα περιεχόμενα της στοίβας με τη σειρά

που έχουν τοποθετηθεί, στην δε τρίτη στήλη φαίνεται η μέχρι εκείνη τη στιγμή διαμορφωμένη μεταθεματική παράσταση:

| Χαρακτήρας (τελεστής ή τελεστέος) | Περιεχόμενα στοίβας | Μεταθεματική παράσταση |
|---|---|---|
| ( | ( | |
| ( | ( ( | |
| A | ( ( | A |
| - | - ( ( | A |
| B | - ( ( | A B |
| ) | ( | A B - |
| + | + ( | A B - |
| C | + ( | A B - C |
| ) | | A B – C + |
| * | * | A B – C + |
| ( | ( * | A B – C + |
| ( | ( ( * | A B – C + |
| D | ( ( * | A B – C +D |
| + | + ( ( * | A B – C +D |
| E | + ( ( * | A B – C +D E |
| ) | ( * | A B – C +D E + |
| * | * ( * | A B – C +D E + |
| ( | ( * ( * | A B – C +D E + |
| F | ( * ( * | A B – C +D E + F |
| - | - ( * ( * | A B – C +D E + F |
| G | - ( * ( * | A B – C +D E + F G |
| ) | * ( * | A B – C +D E + F G - |
| ) | * | A B – C +D E + F G - * |
| | | A B – C +D E + F G - * * |

Πίνακας 5.1



**β) Μεταθεματική σε ένθετη μορφή:**

Η αντίστροφη διαδικασία, η μετατροπή δηλαδή σε ένθετη μορφή μιας παράστασης εκφρασμένης σε μεταθεματική μορφή και συνεπώς ο υπολογισμός της, γίνεται εύκολα με τη χρήση μιας στοίβας. Για τον σκοπό αυτό:

- Σαρώνουμε την παράσταση από αριστερά προς τα δεξιά.
- Εάν συναντήσουμε τελεστέο, τον τοποθετούμε στην στοίβα.

- Εάν συναντήσουμε τελεστή, ανακαλούμε από την στοίβα τους τελεστέους στους οποίου αναφέρεται ο τελεστής, υπολογίζουμε το αποτέλεσμα και το ωθούμε στην στοίβα.
- Το τελικό αποτέλεσμα βρίσκεται στην κορυφή της στοίβας.

Στο παράδειγμα που ακολουθεί μετατρέπουμε σε ένθετη την μεταθεματική μορφή, στην οποία καταλήξαμε στην παραπάνω παράγραφο 2(α), δηλαδή την:

A B – C +D E + F G - * *

Προφανώς, εάν η μετατροπή σε μεταθεματική έχει γίνει σωστά, θα καταλήξουμε στην αρχική ένθετη παράσταση, δηλαδή στην:

((A – B) + C) * ((D + E) * (F – G))

| **Τελεστής ή τελεστέος** | **Περιεχόμενα στοίβας** |
|---|---|
| **A** | **A** |
| **B** | **B A** |
| **-** | **A - B** |
| **C** | **C A - B** |
| **+** | **A – B + C** |
| **D** | **D A – B + C** |
| **E** | **E D A – B + C** |
| **+** | **D + E A – B + C** |
| **F** | **F D + E A – B + C** |
| **G** | **G F D + E A – B + C** |
| **-** | **F - G D + E A – B + C** |
| ***** | **(D + E) * (F – G) A – B + C** |
| ***** | **(A – B + C) * ((D + E) * (F – G))** |
| | |

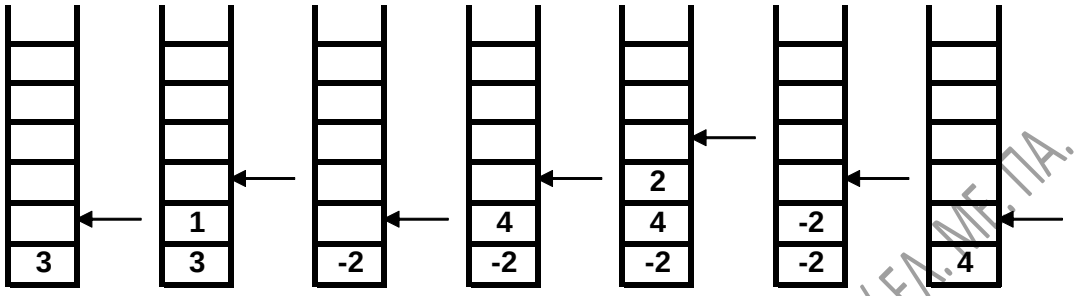
Πίνακας 5.2

Για να δώσουμε ένα αριθμητικό παράδειγμα, έστω ότι αναφερόμαστε σε μια παράσταση ακεραίων αριθμών, την:

3 1 – 4 2 – *

Αυτή μετατρέποντάς την σε μεταθεματική και βρίσκοντας το αποτέλεσμα, καταλήγει σε τιμή 4, όπως φαίνεται και από την τιμή (τελική) που υπάρχει στην κεφαλή της στοίβας του παρακάτω σχήματος 5.2.

(Το βέλος δείχνει την κεφαλή της στοίβας, σύμφωνα και με όσα έχουν προηγουμένως αναφερθεί).




Σχ. 5.2

# ΚΕΦΑΛΑΙΟ 6

# ΟΥΡΕΣ

## 6.1. ΓΕΝΙΚΑ.

Στην καθημερινή ζωή ουρές συναντώνται πολύ συχνά. Μια **ουρά (queue)** δημιουργείται όταν άνθρωποι αλλά και προγράμματα περιμένουν για να εξυπηρετηθούν, οπότε και μιλάμε για **ουρές αναμονής**. Οι ουρές χρησιμοποιούν τη μέθοδο εξυπηρέτησης **FIFO** (First In First Out), ακρωνύμιο που δηλώνει ότι **το πρώτο στοιχείο που μπαίνει στην ουρά είναι και το πρώτο που θα εξυπηρετηθεί**.

Η ουρά μπορεί να θεωρηθεί ως μια σειρά από στοιχεία με δυο άκρα, σαν ενας σωλήνας δηλαδή, στον οποίο ό,τι μπαίνει από το ένα άκρο, βγαίνει από το άλλο. Και μάλιστα **τα στοιχεία βγαίνουν με τη σειρά που μπήκαν**. Τυχαία προσπέλαση ενός συγκεκριμένου στοιχείου δεν είναι δυνατή.

Για να καταλάβουμε πώς λειτουργεί μια ουρά, ας θεωρήσουμε δυο συναρτήσεις, την qinsert( ), η οποία τοποθετεί το όρισμά της στο τέλος της ουράς και την qremove( ), η οποία εξάγει ένα στοιχείο από την ουρά. Το στοιχείο που εξάγεται, αν δεν αποθηκευτεί αλλού, καταστρέφεται. Το παρακάτω σχήμα δείχνει το αποτέλεσμα μιας ακολουθίας από τέτοιες λειτουργίες:

| Ενέργεια | Περιεχόμενα ουράς | | |
|---|---|---|---|
| qinsert(A) | A | | |
| qinsert(B) | A | B | |
| qinsert(C) | A | B | C |
| qremove( ) | B | C | |
| qinsert(D) | B | C | D |
| qremove( ) | C | D | |
| qremove( ) | D | | |
| qremove( ) | Κενή ουρά | | |

Σχ. 6.1

## 6.2. ΥΛΟΠΟΙΗΣΗ ΟΥΡΑΣ.

Όπως και στις στοίβες, ο **χώρος** που θα δεσμευτεί για την αποθήκευση της ουράς μπορεί να είναι ένας **πίνακας.** Διαφορετικά, θα γίνει δέσμευση με την χρήση συναρτήσεων **δυναμικού χειρισμού μνήμης**. Στην παράγραφο αυτή δίνουμε ένα παράδειγμα υλοποίησης ουράς με τη χρήση πίνακα, ενώ υλοποίηση με τη χρήση δυναμικού χειρισμού μνήμης θα γίνει σε επόμενη παράγραφο.

Όπως ισχύει και στις στοίβες, όταν μια ουρά υλοποιείται με την χρήση πίνακα, κάποιες ιδιότητες του πίνακα πρέπει να είναι «απενεργοποιημένες». Έτσι, δεν επιτρέπεται η τυχαία προσπέλαση οποιουδήποτε στοιχείου του. **Τα στοιχεία θα τοποθετούνται μόνο στο ένα άκρο και θα ανακαλούνται μόνο από το άλλο άκρο.** Για τον σκοπό αυτό θα γράψουμε και θα χρησιμοποιήσουμε, τις συναρτήσεις qinsert( ) και qremove( ). Αν δεν διασφαλίσουμε τα παραπάνω, θα έχουμε φυσικά και πάλι πίνακα, όχι όμως ουρά.

Μετά την ανάκληση κάποιου στοιχείου, αυτό εξακολουθεί να υπάρχει στην ίδια θέση του πίνακα όπου βρισκόταν και πριν την ανάκληση. Όμως, επειδή ο πίνακας υλοποιεί μια ουρά, δεν έχουμε τη δυνατότητα να πάρουμε αυτό το στοιχείο και πάλι.



## 6.3. ΕΙΣΑΓΩΓΗ ΣΤΟΙΧΕΙΩΝ ΣΤΗΝ ΟΥΡΑ ΚΑΙ ΑΝΑΚΛΗΣΗ.

Στο παράδειγμα που ακολουθεί υλοποιούμε μια ουρά, στην οποία υποτίθεται ότι τοποθετούνται μόνο θετικοί ακέραιοι, με την χρήση του πίνακα ακεραίων queue.

```
int queue [N];
int ip, rp;

void main(void) {
    int ak, apot;
    ……….
    qinsert (ak);
    apot = qremove ( );
    ……. }

void qinsert (int elem) {
    if (ip == N)
        puts("Ουρά πλήρης");
    else
        queue [ip++] = elem; }
```

```
int qremove ( ) {
    int data;

    if (ip == rp) {
            puts("Ουρά κενή");
            return -1; }
    else {
            data = queue[rp++];
            return data; } }
```

**Σχόλια:**

1. Στο παράδειγμα αυτό ο πίνακας που χρησιμοποιείται για την ουρά και οι ακέραιοι που δείχνουν το σημείο εισαγωγής (πίσω μέρος της ουράς, ip) και ανάκτησης στοιχείων (εμπρός μέρος της ουράς, rp) έχουν δηλωθεί ως εξωτερικές μεταβλητές, άρα όλες οι θέσεις του πίνακα και οι ip και rp έχουν τιμή μηδέν. Μια που δεχτήκαμε ότι στην ουρά αυτή αποθηκεύονται μόνο θετικοί ακέραιοι, η συνάρτηση qremove( ) θα επιστρέφει τιμή -1 μόνο σε περίπτωση «παραβίασης», εάν προσπαθήσουμε δηλαδή να εξαγάγουμε στοιχεία από κενή ουρά.

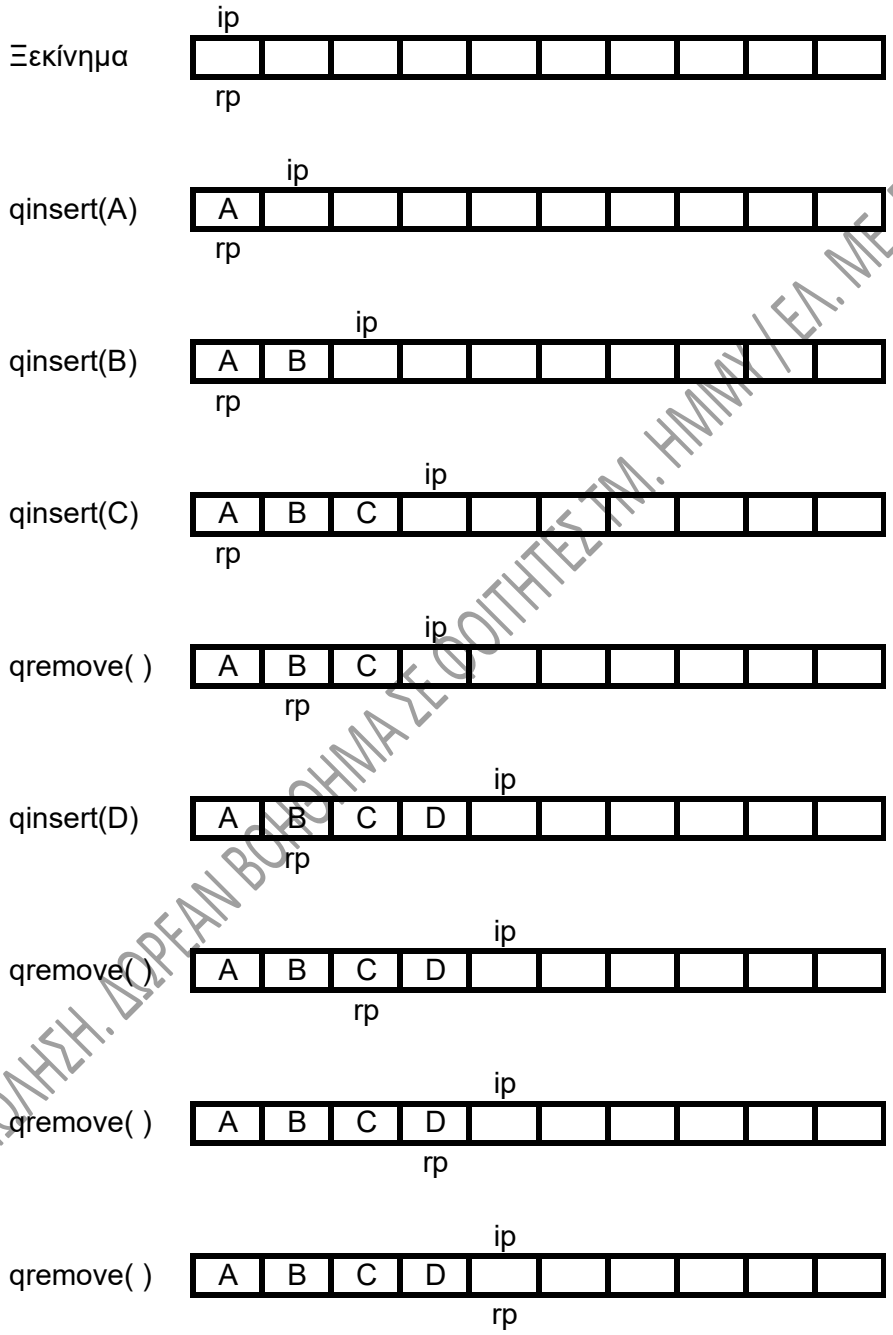
2. Η qinsert ( ) τοποθετεί το όρισμά της (εδώ ένα ακέραιο) στην ουρά. Σύμφωνα με τον αλγόριθμο λειτουργίας της, γίνεται πρώτα **έλεγχος υπερχείλισης** της ουράς. Εάν δεν ισχύει κάτι τέτοιο τοποθετείται ο ακέραιος στην ουρά και μετά η μεταβλητή ip (η οποία δείχνει το σημείο εισαγωγής) αυξάνει κατά 1, σημειώνοντας έτσι τη θέση του πίνακα όπου θα γίνει νέα τοποθέτηση στοιχείου.

3. Στην qremove() πρώτα γίνεται **έλεγχος επάρκειας** της ουράς, προκειμένου να αποκλειστεί η περίπτωση της κενής ουράς. Κενή ουρά σημαίνει ταύτιση του σημείου εισαγωγής με το σημείο ανάκτησης στοιχείων, γεγονός που συμβαίνει όταν ισχύσει ip == rp. Εάν δεν ισχύει κάτι τέτοιο, η συνάρτηση επιστρέφει τιμή ίση με τα περιεχόμενα του πίνακα σε θέση όσο το rp, το οποίο στη συνέχεια αυξάνει κατά 1, ενώ εάν η ουρά είναι κενή, η συνάρτηση, όπως είπαμε, επιστρέφει τιμή -1.

4. Στην ουσία **το rp «καταδιώκει» το ip** κατά μήκος της ουράς. Το παρακάτω σχήμα (Σχ. 6.2) δείχνει πώς φαίνεται στη μνήμη η διαδικασία του σχ. 6.1.

   Προφανώς, σιγά-σιγά, **το ip θα φτάσει να γίνει ίσο με τη μέγιστη τιμή της διάστασης του πίνακα και τότε δεν θα μπορούν να εισαχθούν νέα στοιχεία**, παρά το ότι η ουρά μπορεί να διαθέτει ελάχιστα στοιχεία (αν ο rp είναι πολύ κοντά στον ip). Τότε απαιτείται η **μετακίνηση όλων των στοιχείων στην αρχή του**

**πίνακα** από μια συνάρτηση, η κλήση της οποίας θα μπορεί να γίνεται αυτόματα, όταν δεν υπάρχει χώρος για εισαγωγή νέου στοιχείου.




Σχ. 6.2.

Αντίστοιχα, η «καταδίωξη» του ip από rp μπορεί να οδηγήσει κάποια στιγμή σε εξίσωση αυτών των δυο τιμών. Στην περίπτωση αυτή η ουρά φαίνεται κενή, αλλά

πρακτικά θα μπορούμε να τοποθετήσουμε λίγα ή και καθόλου νέα στοιχεία στην ουρά, όσο πιο μακριά από την αρχή του πίνακα γίνεται η εξίσωση των ip και rp. Όταν συμβεί η εξίσωση αυτή, μπορούμε να επαναφέρουμε τις τιμές στην αρχή, αποδίδοντας **και στις δυο μεταβλητές τιμή μηδέν**.

## 6.4. ΚΥΚΛΙΚΗ ΟΥΡΑ.

Προβλήματα όπως τα παραπάνω μπορούν να αντιμετωπιστούν με τη χρήση κυκλικών ουρών. Οι ουρές αυτές έχουν τη μορφή ενός **δακτυλίου** (ring). Στην πράξη δηλαδή είναι σα να έχουμε για μονάδα αποθήκευσης **ένα πίνακα, του οποίου το τέλος είναι ενωμένο με την αρχή.** Από αυτό το γεγονός προκύπτει και το όνομα **κυκλική ουρά (circular queue)**. Έτσι μπορούν να τοποθετούνται συνεχώς στοιχεία στην ουρά, αφού συνεχώς αφαιρούνται άλλα. Πρέπει πάντως να λαμβάνεται μέριμνα ώστε **να μην επικαλυφθούν τα πρώτα από τα τελευταία στοιχεία της ουράς**.

## 6.5. ΟΥΡΑ ΠΡΟΤΕΡΑΙΟΤΗΤΑΣ.

Στο είδος αυτό της ουράς, **ανακαλείται** κάθε φορά **το στοιχείο, το οποίο έχει τη μεγαλύτερη προτεραιότητα**. Αν η προτεραιότητα όλων των στοιχείων είναι η ίδια, τότε η λειτουργία είναι αυτή της κοινής ουράς. Είναι φανερό ότι κάθε στοιχείο της ουράς θα συνοδεύεται από κάποια πρόσθετη πληροφορία, η οποία θα καθορίζει την προτεραιότητά του. Αυτή θα μπορούσε να είναι για παράδειγμα μια ακέραια μεταβλητή. Όσο μεγαλύτερη είναι η τιμή της, τόσο μεγαλύτερη και η προτεραιότητα του συγκεκριμένου μέλους της ουράς. Ένας άλλος τρόπος υλοποίησης, αν για παράδειγμα οι προτεραιότητες των στοιχείων είναι όλες μεταξύ 1 και k, είναι η χρησιμοποίηση k πινάκων, καθένας από τους οποίους περιέχει στοιχεία ίσης μεταξύ τους προτεραιότητας.

# ΚΕΦΑΛΑΙΟ 7

# ΛΙΣΤΕΣ

## 7.1. ΓΕΝΙΚΑ.

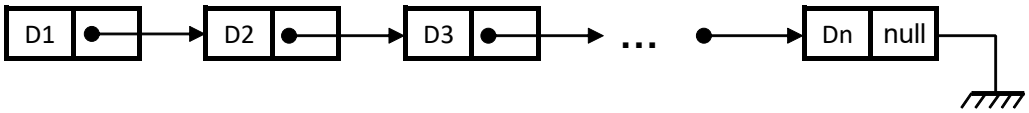
Οι πίνακες, οι στοίβες και οι ουρές που εξετάστηκαν μέχρι τώρα έχουν μερικά κοινά χαρακτηριστικά. Ένα **κοινό χαρακτηριστικό για τις στοίβες και τις ουρές** είναι ότι η ανάκτηση των στοιχείων από αυτές είναι εν γένει καταστροφική. Με αυτό εννοούμε ότι η ανάκτηση κάποιου στοιχείου σημαίνει και αφαίρεσή του από τη στοίβα ή την ουρά και, αν δεν λάβουμε μέριμνα για την αποθήκευσή του, αυτό το στοιχείο χάνεται οριστικά. Ένα **χαρακτηριστικό, κοινό για όλες τις δομές δεδομένων** που γνωρίσαμε (πίνακες, στοίβες και ουρές) είναι ότι η λογική σειρά των στοιχείων τους συμπίπτει και με τη φυσική σειρά, τις θέσεις αποθήκευσης δηλαδή στη μνήμη.

Αυτό το τελευταίο μπορεί εύκολα να πάψει να ισχύει, τα δε στοιχεία μπορούν να αποθηκεύονται σε διάφορες θέσεις μνήμης. Προϋπόθεση βέβαια είναι η διατήρηση της σύνδεσης του ενός στοιχείου με το άλλο, η ύπαρξη δηλαδή της δυνατότητας να πάμε από το ένα στοιχείο στο λογικά επόμενό του. Για να γίνει αυτό κατορθωτό χρειαζόμαστε σε κάθε θέση ένα **δείκτη** (pointer), ο οποίος **θα δείχνει τη θέση του επόμενου στοιχείου**. Ο τρόπος αυτός οργάνωσης των δεδομένων μας λέγεται **«απλά συνδεδεμένη λίστα»**. Στις συνδεδεμένες λίστες η ανάκτηση των στοιχείων δεν συνεπάγεται ούτε την απομάκρυνσή τους από την λίστα, ούτε την καταστροφή τους. Για τη διαγραφή μάλιστα ενός στοιχείου από τη λίστα πρέπει να φροντίσουμε ιδιαίτερα.



## 7.2. ΕΙΔΗ ΛΙΣΤΩΝ.

Στο σχ. 7.1 παρουσιάζεται η μορφή μιας **«απλά συνδεδεμένης λίστας»**. Κάθε στοιχείο της λίστας (δεδομένα και δείκτης) ονομάζεται **κόμβος (node)**. Με D1, D2, …Dn συμβολίζονται τα δεδομένα των κόμβων, ενώ τα βελάκια συμβολίζουν τους δείκτες από τη μια θέση της λίστας στην άλλη.

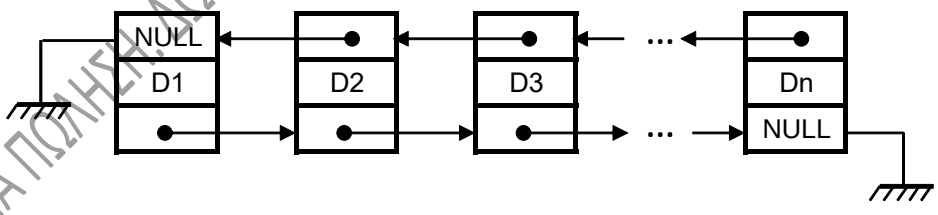



Σχ. 7.1

Η θέση του πρώτου κόμβου (**«κεφαλή»**) πρέπει να είναι συνέχεια γνωστή. Ξεκινώντας από εκεί μπορούμε να διατρέξουμε όλη τη λίστα. Για τον σκοπό αυτό θα πρέπει να υπάρχει **ένας δείκτης, ο οποίος** να τοποθετηθεί έτσι ώστε **να δείχνει στον πρώτο κόμβο της λίστας και ο οποίος δείκτης δεν πρέπει να μετακινηθεί ποτέ από εκεί**. Εάν θελήσουμε **να διατρέξουμε την λίστα** πρέπει να χρησιμοποιήσουμε **ένα άλλο δείκτη**, ο οποίος αρχικά μεν θα τοποθετηθεί στην κεφαλή και στη συνέχεια θα μετακινείται από τον ένα κόμβο στον άλλο. Με τη χρήση αυτού του δείκτη θα προσπελαύνουμε τους κόμβους της λίστας. Εξ άλλου, το **NULL, το σύμβολο της γείωσης,** δηλώνει το τέλος της λίστας. Το NULL αποτελεί συγκεκριμένη τιμή, αυτήν του **«μηδενικού δείκτη»**.

Είναι προφανές ότι **το ιδανικό μέσο για την περιγραφή των κόμβων είναι η δομή (structure)**, αφού μπορεί να ενσωματώνει περισσότερα από ένα είδη δεδομένων (άρα για παράδειγμα κείμενο, αριθμητικά στοιχεία, δείκτες κλπ).

Μια λίστα μπορεί να είναι **«διπλά συνδεδεμένη λίστα»**. Σε κάθε κόμβο μιας διπλά συνδεδεμένης λίστας υπάρχουν δεδομένα, ένας δείκτης προς τον επόμενο κόμβο, αλλά και ένας δείκτης προς τον προηγούμενο κόμβο. Σχηματικά:




Σχ. 7.2

Οι διπλά συνδεδεμένες λίστες προσφέρουν δυο κύρια πλεονεκτήματα. Το πρώτο είναι ότι ξεκινώντας από οποιοδήποτε κόμβο της, **μπορούμε να διατρέξουμε την λίστα και προς τις δυο κατευθύνσεις**. Δεν είναι δηλαδή απαραίτητο να γνωρίζουμε την κεφαλή. Αρκεί να μπορούμε να έχουμε πρόσβαση σε οποιοδήποτε κόμβο. Το δεύτερο πλεονέκτημα αποκτά ιδιαίτερη σημασία **αν** για κάποιο λόγο **καταστραφούν οι**

**σύνδεσμοι προς τη μια κατεύθυνση**. Τότε, χρησιμοποιώντας τους άλλους συνδέσμους **μπορούμε και πάλι να διατρέξουμε ολόκληρη τη λίστα**.

Οι λίστες μπορεί να είναι **στατικές** ή **δυναμικές**. Στις πρώτες, έχει οριστεί εκ των προτέρων ο μέγιστος αριθμός κόμβων. Στις δυναμικές αυτό δεν είναι απαραίτητο. Οι δυναμικές είναι πολύ πιο ευέλικτες από τις στατικές γιατί μπορούν να μεγαλώνουν ή να μικραίνουν κατά τη διάρκεια της εκτέλεσης του προγράμματος.

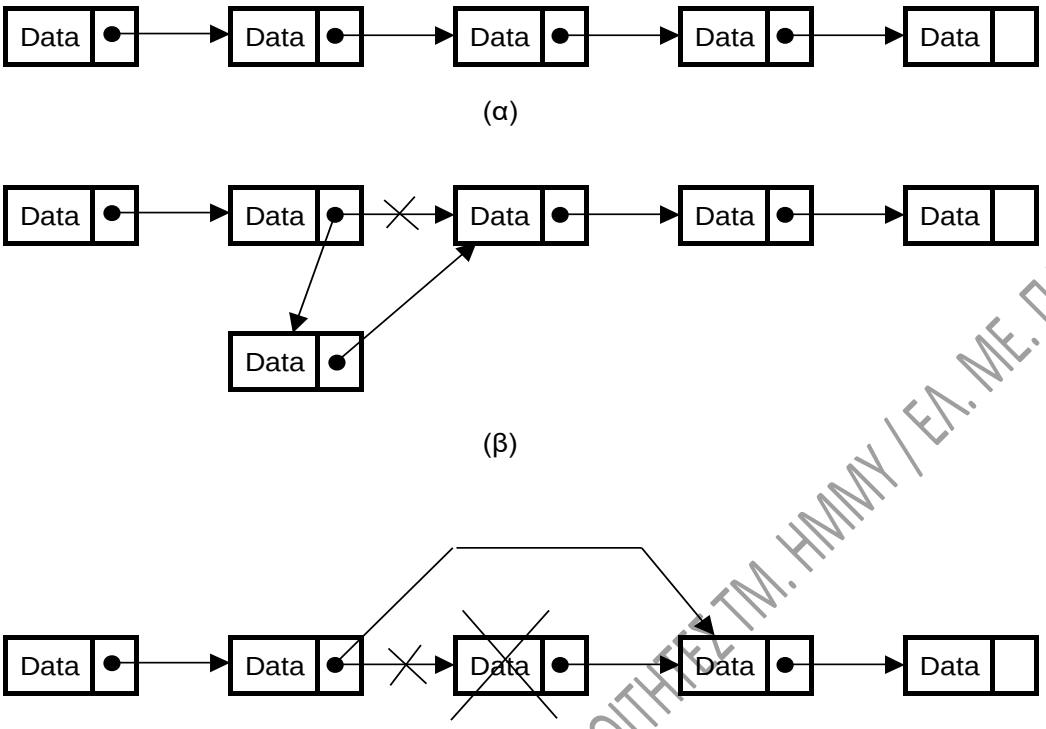
Λόγω του τρόπου οργάνωσής τους και του γεγονότος ότι η φυσική με τη λογική σειρά των στοιχείων δεν συμπίπτουν, **οι συνδεδεμένες λίστες δεν λειτουργούν με τον ίδιο τρόπο που λειτουργούν οι πίνακες**. Έτσι, για να προσπελάσουμε για παράδειγμα το 100ο στοιχείο μιας λίστας θα πρέπει να κινηθούμε από την αρχή και ακολουθώντας τους δείκτες να περάσουμε **σειριακά** από ένα-ένα κόμβο, μέχρι να φτάσουμε στον 100ο. Αντίθετα, στους πίνακες μπορούμε να προσπελάσουμε άμεσα το 100ο στοιχείο. Αυτό είναι ένα πλεονέκτημα των πινάκων σε σχέση με τις συνδεδεμένες λίστες, οι οποίες όμως πλεονεκτούν κατά πολύ στις διαδικασίες διαγραφής κάποιου κόμβου, εισαγωγής νέου ή μετακίνησής του σε άλλη θέση.



## 7.3. ΣΗΜΑΝΤΙΚΕΣ ΛΕΙΤΟΥΡΓΙΕΣ ΣΤΙΣ ΛΙΣΤΕΣ.

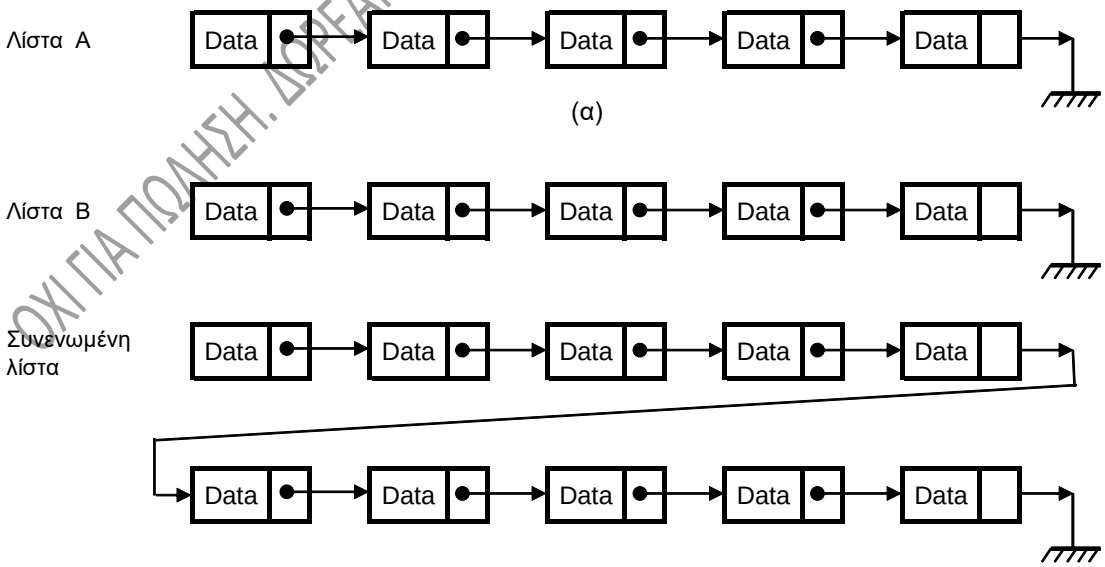
**α)** **Εισαγωγή στοιχείου.** Γίνεται αλλάζοντας τον δείκτη του στοιχείου μετά από το οποίο θα γίνει εισαγωγή, έτσι ώστε ο δείκτης αυτός να δείχνει το στοιχείο που πρόκειται να εισαχθεί. Ο δείκτης εξ άλλου του εισαγόμενου στοιχείου οδηγείται να δείχνει στο επόμενο στοιχείο της λίστας (σχ. 7.3β).

**β)** **Διαγραφή στοιχείου.** Γίνεται αλλάζοντας τον δείκτη κάποιου στοιχείου έτσι ώστε να δείχνει το μεθεπόμενο στοιχείο της λίστας, παρακάμπτοντας δηλαδή το υπό διαγραφή στοιχείο (σχ. 7.3γ).




Σχ. 7.3

**γ) Συνένωση.** Μπορούμε να συνδέσουμε δυο λίστες σε μια, κάνοντας τον τελευταίο δείκτη της πρώτης να δείξει στην κεφαλή της δεύτερης (σχ 7.4).

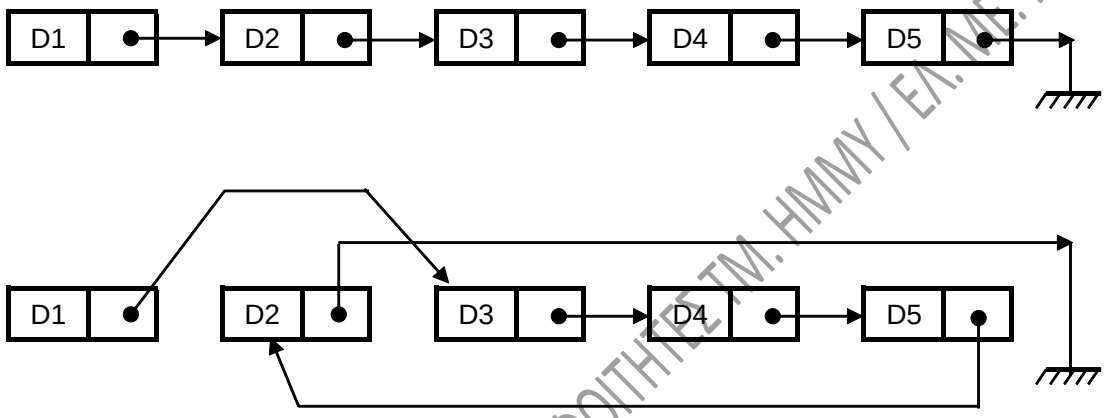



Σχ. 7.4

**δ)** **Αναζήτηση.** Αναζήτηση κάποιου στοιχείου μέσα στη λίστα με βάση κάποια κριτήρια. Απαιτείται να διασχίσουμε την λίστα με τη βοήθεια των δεικτών.

**ε)** **Αντιστροφή.** Το πρώτο στοιχείο της λίστας να γίνει τελευταίο κλπ.

**στ)** **Μετακίνηση.** Η μετακίνηση ενός κόμβου από μια θέση σε μια άλλη γίνεται απλά αλλάζοντας τις διευθύνσεις στους κόμβους. Στο σχ. 7.5, η αρχική σειρά των κόμβων D1 - D2 - D3 - D4 - D5 έχει τροποποιηθεί σε D1 – D3 – D4 – D5 – D2.




Σχ. 7.5

Σε κάθε περίπτωση **η σειρά με την οποία εκτελούνται οι κινήσεις μας** (τοποθέτηση δεικτών κλπ) **έχει ιδιαίτερη σημασία**, αφού μια «άστοχη» κίνηση μπορεί εύκολα να οδηγήσει σε απώλεια όλης της λίστας ή μέρους της, δηλαδή σε αδυναμία να προσπελάσουμε όλους ή μέρος των κόμβων της λίστας.

Ας παρατηρήσουμε τον πολύ μικρό αριθμό των ενεργειών που απαιτούνται για τη μετακίνηση του κόμβου D2 στο τέλος της λίστας στην παραπάνω παράγραφο (στ) και το σχ. 7.5. Για μια αντίστοιχη μετακίνηση στοιχείου στο τέλος ενός πίνακα θα έπρεπε να μετακινήσουμε όλα τα στοιχεία του πίνακα προς την αρχή του πίνακα και μετά να τοποθετήσουμε το εν λόγω στοιχείο στο τέλος. Ομοίως, η εισαγωγή ενός στοιχείου σε ένα πίνακα συνεπάγεται το «άνοιγμα χώρου», την μετακίνηση δηλαδή κάποιων στοιχείων προς το τέλος του πίνακα και την τοποθέτηση του νέου στοιχείου στην θέση που άδειασε. Αντίστοιχες σκέψεις μπορούμε να κάνουμε για τις διάφορες άλλες ενέργειες που αναφέρουμε πιο πάνω (διαγραφή κλπ). **Λόγω της ευκολίας** με την οποία υλοποιούνται τα παραπάνω, **προτιμούμε συχνά να χρησιμοποιούμε συνδεδεμένες λίστες αντί για πίνακες**. Από την άλλη πλευρά, **η προσπέλαση των στοιχείων ενός**

**πίνακα είναι πολύ πιο γρήγορη από αυτή των στοιχείων μιας λίστας**, αφού στον πίνακα η προσπέλαση αυτή είναι άμεση, χωρίς να χρειάζεται να ακολουθήσουμε τους δείκτες που θα μας οδηγήσουν από το ένα στο άλλο.

## 7.4. ΑΠΛΑ ΣΥΝΔΕΔΕΜΕΝΗ ΛΙΣΤΑ.

Στο πρόγραμμα που ακολουθεί δίνουμε κώδικα σε C για την δημιουργία και επεξεργασία μιας απλά συνδεδεμένης λίστας. Το πρόγραμμα υποτίθεται ότι σχετίζεται με τη γραμματεία κάποιας σχολής και δημιουργεί μια συνδεδεμένη λίστα, σε κάθε κόμβο της οποίας υπάρχουν τα εξής στοιχεία: το ονοματεπώνυμο ενός φοιτητή της σχολής (name) και ο αριθμός μητρώου του (am). Στο πρόγραμμα γίνεται εισαγωγή φοιτητών μέχρι να δοθεί αρνητικός αριθμός μητρώου. Δίδεται μια συνάρτηση, η display( ) για την εμφάνιση των στοιχείων των κόμβων της λίστας στην οθόνη, ενώ στις επόμενες παραγράφους δίνονται επίσης –ενδεικτικά- δυο ακόμη συναρτήσεις: μια συνάρτηση για την εισαγωγή ενός κόμβου σε μια τέτοια λίστα και μια συνάρτηση την διαγραφή ενός κόμβου από αυτήν.

### 7.4.1. Δημιουργία – αρχικοποίηση απλά συνδεδεμένης λίστας.

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

struct node {
      char name [30];
      int    am;
      struct node *next; };

void display (struct node *);

int main ( ) {
      struct node *head, *curr, *ptr;
      char nol[8];
      int num;

      /* Δημιουργία – Εισαγωγή στοιχείων στη λίστα */

      head = (struct node *) malloc (sizeof (struct node));        /* Γραμμή 15 */
      curr = ptr = head;                                           /* Γραμμή 16 */
      curr->next = NULL;                                           /* Γραμμή 17 */
      printf ("DWSTE ARI8MO MHTRWOY ");
      gets (nol);
      num = atoi (nol);
```

```
        while (num > 0) {
            curr->am = num;
            printf ("DWSTE ONOMA ");
            gets (curr->name);
            curr->next = (struct node *) malloc (sizeof (struct node));
            curr = curr->next;                                  /* Γραμμή 26 */
            curr->next = NULL;                                  /* Γραμμή 27 */
            printf ("DWSTE EPOMENO ARI8MO MHTRWOY ");
            gets (nol);
            num = atoi (nol); }

        while (ptr->next != curr)                   /* Γραμμή 31 */
            ptr = ptr->next;
        ptr->next = NULL;                           /* Γραμμή 33 */
        free (curr);                                /* Γραμμή 34 */
        curr = NULL; }                              /* Γραμμή 35 */
        return 1;
}

        /* Εμφάνιση των στοιχείων της λίστας στην οθόνη */

void display (struct node *head) {
        struct node *curr;

        curr = head;
        while (curr != NULL) {
            printf ("ONOMA %s. ARI8MOS MHTRWOY %5d\n", curr->name, curr->am);
            curr = curr->next; }
}
```



**Σχόλια:**

- Η περιγραφή της δομής για κάθε κόμβο της λίστας δίνεται από την:

```
struct node {
    char name [30];
    int    am;
    struct node *next; };
```

  Προφανώς η δομή μπορεί να έχει οποιοδήποτε πλήθος πεδίων, όμως **ένα από τα πεδία της πρέπει να είναι δείκτης προς δομή του ίδιου τύπου** (εδώ αυτό είναι το πεδίο next). Έτσι, με το σωστό «δέσιμο» των δεικτών θα γίνει δυνατή η δημιουργία της λίστας.

- Η δημιουργία των κόμβων της λίστας γίνεται με την κλήση της malloc( ), η οποία καλείται για πρώτη φορά στην γραμμή 15 του προγράμματος, οπότε και δημιουργείται ο πρώτος κόμβος της λίστας. **Ο δείκτης head** γίνεται ίσος με τον δείκτη που επιστρέφει η malloc( ), άρα **τοποθετείται στην κεφαλή της λίστας**.

Αυτός αποτελεί τον δείκτη ο οποίος δεν θα μετακινείται ποτέ, όπως αναφέρθηκε και στην παράγραφο 7.2., ακριβώς για να μη χαθεί η αρχή της λίστας.

- **Ο δείκτης curr** μετακινείται από κόμβο σε κόμβο και είναι αυτός που χρησιμοποιείται για να μπορέσουμε τελικά να διατρέξουμε όλη τη λίστα. Αρχικά, στην γραμμή 16, ο current τοποθετείται στην κεφαλή της λίστας (γίνεται ίσος με τον head). Η μετακίνησή του από τον ένα κόμβο στον άλλο γίνεται στη γραμμή 26 του προγράμματος με την εντολή:

```
curr = curr->next;
```

Θυμίζουμε ότι, όπως αναφέραμε ήδη στην παράγραφο 7.3., σε ένα πίνακα, τα στοιχεία του βρίσκονται αποθηκευμένα σε διαδοχικές θέσεις μνήμης, ενώ **τα μέλη μιας λίστας δεν βρίσκονται αναγκαστικά αποθηκευμένα σε διαδοχικές θέσεις μνήμης**. Ο δείκτης κάθε κόμβου δείχνει τη θέση μνήμης στην οποία είναι αποθηκευμένος ο επόμενος. Στους πίνακες, εάν γνωρίζουμε το πού βρίσκεται αποθηκευμένο ένα στοιχείο (εάν γνωρίζουμε τη διεύθυνσή του δηλαδή) μπορούμε να βρούμε και όλα τα υπόλοιπα. **Στις λίστες** αντίθετα **μπορούμε να βρούμε μόνο το επόμενο στοιχείο από το γνωστό**. Έτσι, στον πίνακα μπορούμε για παράδειγμα να πάμε από το 1ο στοιχείο στο 10ο αυξάνοντας κατά 10 τον δείκτη που δείχνει στο πρώτο στοιχείο. Αντίθετα, σε μια λίστα, για να μετακινηθούμε στο 10ο στοιχείο στη σειρά πρέπει από τον πρώτο κόμβο να οδηγηθούμε στον 2ο, μετά από τον 2ο στον 3ο κ.ο.κ.

- **Στον δείκτη της τελευταίας δομής στη σειρά δίνεται κάθε φορά η τιμή NULL**. Αυτή είναι η **μηδενική τιμή του δείκτη**, ένας δείκτης δηλαδή ο οποίος δείχνει στο «τίποτα», στο μηδέν. Αφού ο δείκτης αυτός δείχνει κανονικά σε δομές του είδους node, κανονικά θα έπρεπε να έχουμε προσαρμόσει και το NULL, έτσι ώστε να είναι της μορφής struct node, τυπικά δηλαδή στην γραμμή 27 θα έπρεπε να έχουμε γράψει:

```
curr->next = (struct node *) NULL;
```

αντίστοιχα δε και στις γραμμές 17, 33 και 35. Όμως η απόδοση τιμής με τον τρόπο που γίνεται στις γραμμές αυτές, όσον αφορά την τιμή NULL είναι αποδεκτή και σωστή.

- Όπως είναι διαρθωμένο το πρόγραμμα, δημιουργείται ένας κόμβος παραπάνω από όσους χρειαζόμαστε. Από την γραμμή 31 έως την γραμμή 37 του προγράμματος ελευθερώνουμε τον χώρο τον οποίο καταλαμβάνει ο κόμβος αυτός.

- Σε ένα πίνακα η δέσμευση χώρου στη μνήμη γίνεται για όλα τα στοιχεία του πίνακα μαζί, με τη δήλωσή του. **Στις λίστες δεσμεύεται χώρος για κάθε κόμβο χωριστά** καλώντας τη malloc( ) (ή άλλη αντίστοιχη συνάρτηση) τόσες φορές, όσες και οι κόμβοι.

- Παρατηρείστε **πώς εμφανίζονται στην οθόνη τα στοιχεία της λίστας** με την χρήση της συνάρτησης display( ). Όταν έχει τερματιστεί το διάβασμα, δεν ξέρουμε πόσους ακριβώς κόμβους περιέχει η λίστα. Αυτό που γνωρίζουμε πάντως είναι ότι **ο next δείκτης του τελευταίου κόμβου έχει τιμή NULL**. Η πρακτική που εφαρμόζουμε στο πρόγραμμα είναι **να τοποθετήσουμε αρχικά ένα δείκτη** (τον curr) **στην κεφαλή της λίστας**. Στη συνέχεια **ελέγχουμε αν ο δείκτης αυτός έχει πάρει τιμή NULL**, η οποία σηματοδοτεί το τέλος της λίστας. **Όσο αυτό δεν ισχύει προχωρούμε τον current στον επόμενο κόμβο της λίστας**. Το τμήμα του κώδικα που υλοποιεί τα παραπάνω, δηλαδή το:

```
curr=head;
while (curr != NULL) {
        ……….……
        curr = curr->next; }
```

  θα το συναντήσουμε και θα μας βοηθήσει σε πλήθος ανάλογες περιπτώσεις:

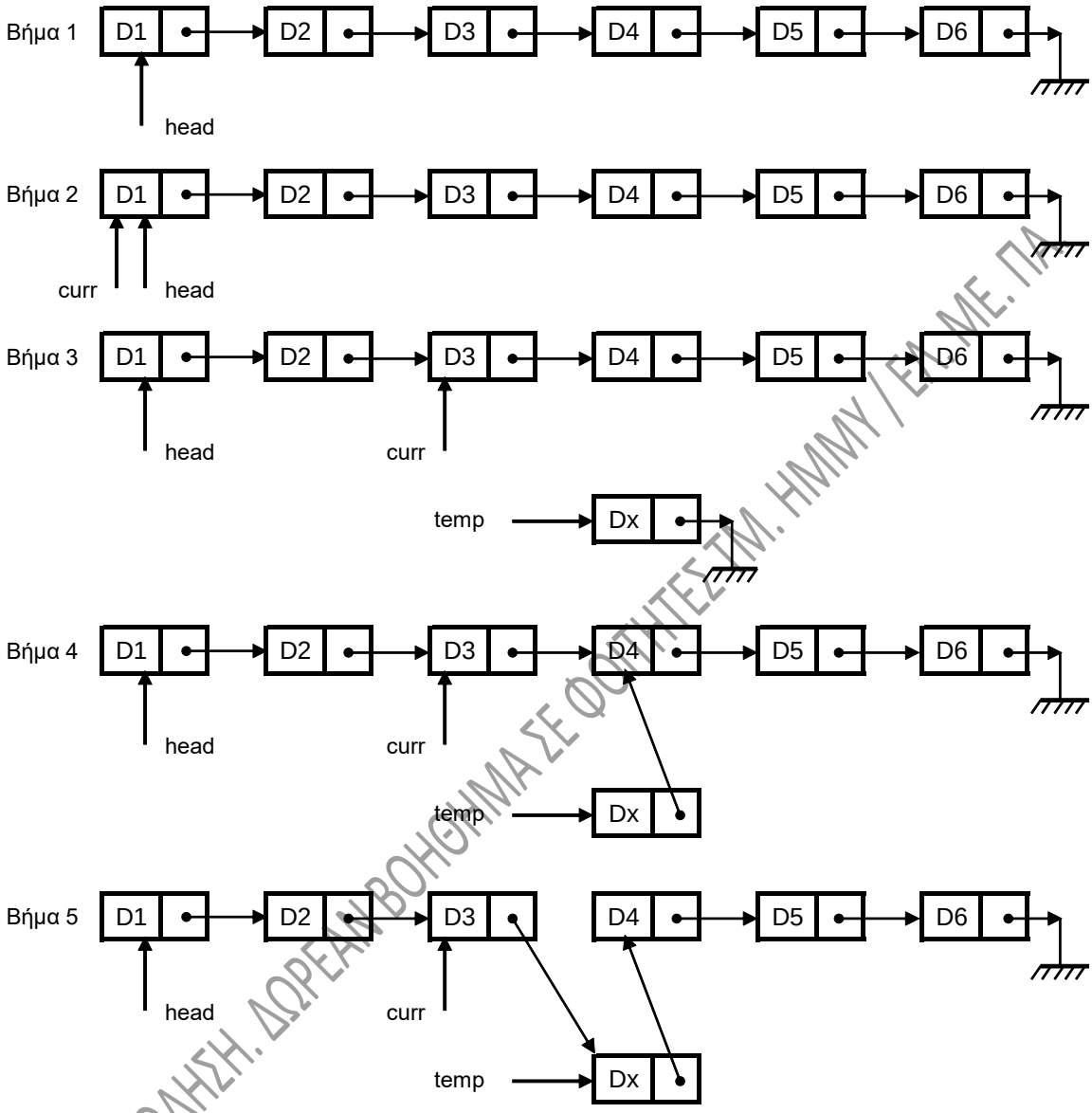
- Χρειάζεται, εκτός από το stdio.h να συμπεριληφθεί και το αρχείο κεφαλίδας **stdlib.h, για την συνάρτηση malloc( )**.



### 7.4.2. Εισαγωγή κόμβου σε απλά συνδεδεμένη λίστα.

Παρακάτω θα δούμε ένα αλγόριθμο **εισαγωγής νέου κόμβου μετά από κάποιον δεδομένο κόμβο σε μια απλά συνδεδεμένη λίστα** (η υλοποίηση μπορεί φυσικά να γίνει με διάφορους τρόπους). Σχηματικά, οι ενέργειές μας συνοψίζονται στα βήματα που περιγράφει το σχ. 7.6, θεωρώντας ότι η εισαγωγή του νέου κόμβου θα γίνει μεταξύ των κόμβων 3 και 4.

Η δημιουργία και εισαγωγή του νέου κόμβου γίνεται με τη χρήση της συνάρτησης insert_list( ) που ακολουθεί. Η συνάρτηση δέχεται ως **ορίσματα ένα δείκτη στην κεφαλή (αρχή) της λίστας και ένα ακέραιο**, ο οποίος ισούται με τον αύξοντα αριθμό του κόμβου, μετά από τον οποίο θα γίνει η εισαγωγή (έστω pos η θέση του κόμβου αυτού στην λίστα). Οι κόμβοι της λίστας είναι του τύπου node της προηγουμένης παραγράφου.




Σχ. 7.6



Παρατηρείστε ότι η συνάρτηση χρησιμοποιεί ένα τοπικό δείκτη, τον curr, ώστε μέσω αυτού να μπορούμε να έχουμε προσπέλαση στους διάφορους κόμβους της λίστας. Με τη χρήση του for, ο δείκτης curr τοποθετείται να δείχνει στον κόμβο μετά τον οποίο θα γίνει η εισαγωγή. **Ο δείκτης ihead που** χρησιμοποιεί η συνάρτηση για να **δείχνει την κεφαλή της λίστας, δεν μετακινείται**. Όπως έχουμε ήδη αναφέρει, αν ο δείκτης αυτός αλλάξει θέση, δεν θα είναι δυνατό να προσπελάσουμε στοιχεία της λίστας που βρίσκονται αποθηκευμένα σε προηγούμενες θέσεις μνήμης.

```
void insert_list (struct node *head, int pos) {
    int    k;
    struct node *temp, *curr;

    curr = ihead;
    for (k=1; k<pos; k++)
        curr = curr -> next;
    temp = (struct node *) malloc (sizeof (struct node));
    temp -> next = NULL;
    gets (temp -> student);
    scanf ("%d", &temp -> am);
    temp -> next = curr -> next;
    curr -> next = temp;
}
```

Ας θεωρήσουμε ότι η λίστα έχει δημιουργηθεί στη main( ), όπου και έχει δηλωθεί ένας δείκτης σε κόμβους του τύπου node, ο **head** και **ο οποίος δείκτης έχει τοποθετηθεί στην κεφαλή της λίστας**. Τότε η κλήση της συνάρτησης insert_list( ) για την εισαγωγή ενός νέου κόμβου μεταξύ των κόμβων 3 και 4 της αρχικής λίστας, θα ήταν:

```
int main(void) {
    struct node *head;
    ……….
    insert_list (head, 3);
    ……….
}
```

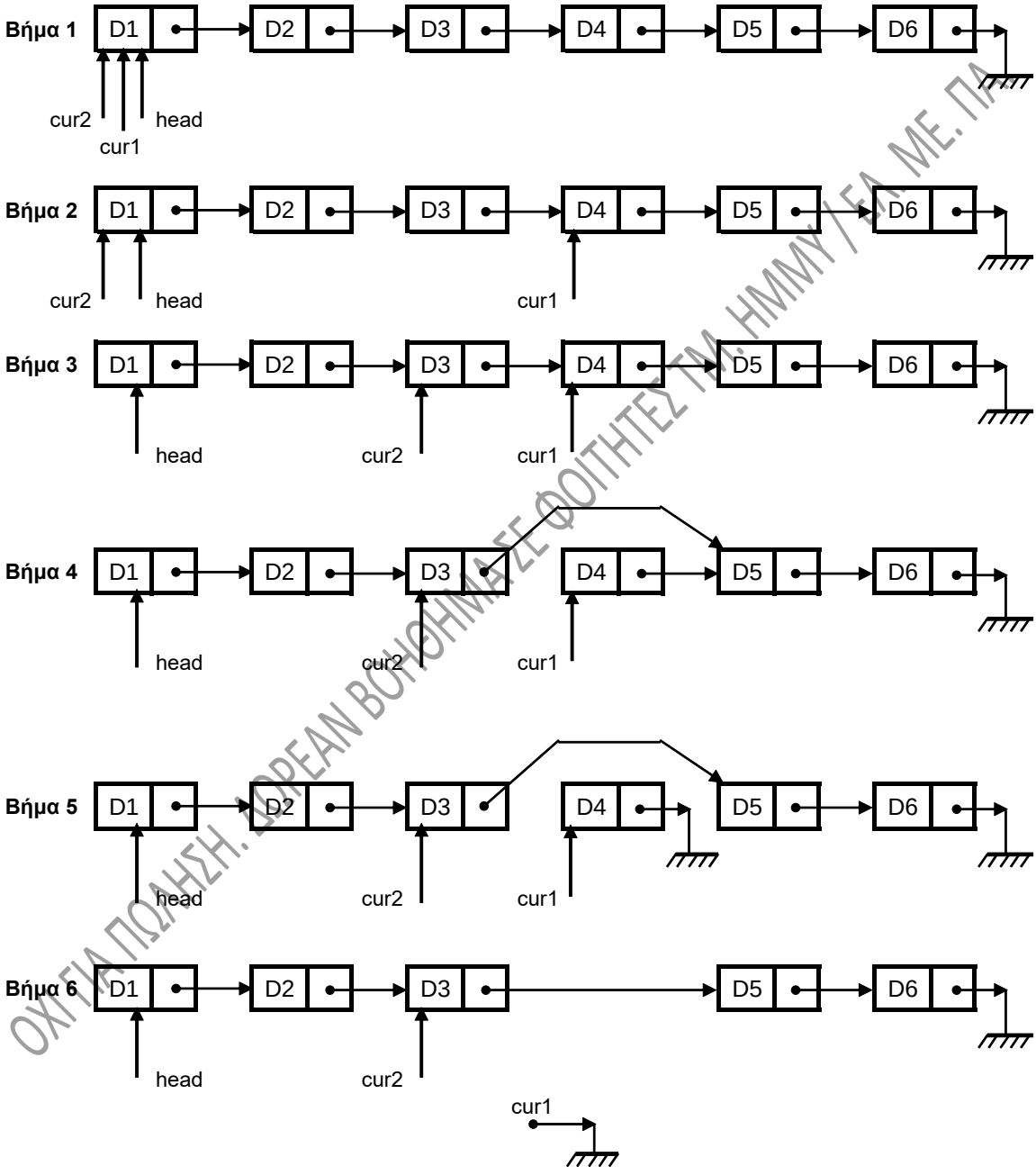
### 7.4.3. Διαγραφή κόμβου από απλά συνδεδεμένη λίστα.

Ακολουθεί ένας αλγόριθμος **διαγραφής κόμβου από μια απλά συνδεδεμένη λίστα**. Σχηματικά, οι ενέργειές μας συνοψίζονται στα βήματα που περιγράφει το σχ. 7.7, θεωρώντας ότι ο υπό διαγραφή κόμβος είναι ο 4ος κόμβος της λίστας.

Η συνάρτηση del_list( ) που ακολουθεί, διαγράφει τον κόμβο από τη λίστα. Η συνάρτηση δέχεται ως **ορίσματα ένα δείκτη στην κεφαλή (αρχή) της λίστας και ένα ακέραιο**, ο οποίος ισούται με τον αύξοντα αριθμό του κόμβου, ο οποίος θα διαγραφεί. Οι κόμβοι της λίστας είναι του τύπου node της προηγουμένης παραγράφου.

Παρατηρείστε ότι η συνάρτηση χρησιμοποιεί **δύο δείκτες**, τους cur1 και cur2. Με τη χρήση των δυο for, **ο cur1 τοποθετείται στον υπό διαγραφή κόμβο, ενώ ο cur2 στον αμέσως προηγούμενο** κόμβο της λίστας. Μετά την κατάλληλη μετακίνηση των δεικτών, **ο κόμβος D4 παύει να είναι προσπελάσιμος**. Στη συνέχεια, με τη βοήθεια του cur1 και **με τη χρήση της συνάρτησης free( ), ο χώρος μνήμης που καταλαμβάνει ο**

**κόμβος D4 μπορεί να απελευθερωθεί**, ώστε να μη γεμίζει η μνήμη με «σκουπίδια». **Σε κάθε χρήση της free( )** και για αποφυγή λαθών, **πρέπει να απομονώνεται ο χώρος που θα ελευθερωθεί**. Να «κόβονται» δηλαδή όποιοι δείκτες ξεκινούν από αυτόν ή καταλήγουν σε αυτόν.





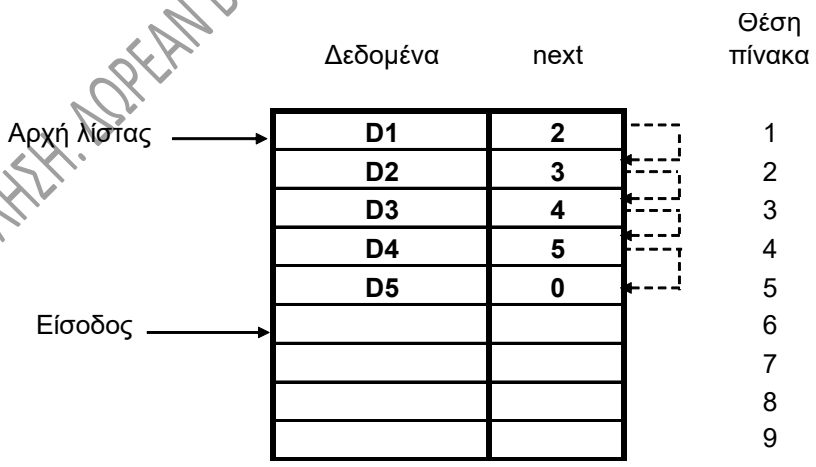

Σχ. 7.7

```
void del_list (struct node *head, int pos) {
    int    k;
    struct node *cur1, *cur2;

    cur1 = head;
    cur2 = head;
    for (k=1; k<pos; k++)
        cur1 = cur1 -> next;
    for (k=1; k<pos-1; k++)
        cur2 = cur2 -> next;
    cur2 -> next = cur1 -> next;
    cur1 -> next = NULL;
    free(cur1);
    cur1 = NULL;
}
```

## 7.5. ΥΛΟΠΟΙΗΣΗ ΑΠΛΑ ΣΥΝΔΕΔΕΜΕΝΗΣ ΛΙΣΤΑΣ ΜΕ ΤΗ ΧΡΗΣΗ ΠΙΝΑΚΑ.

Μια λίστα μπορεί να υλοποιηθεί **με τη χρήση δυο μονοδιάστατων πινάκων** (ή ίσως **με τη χρήση ενός δισδιάστατου πίνακα**). Στο σχήμα 7.8 παρουσιάζεται η υλοποίηση μιας λίστας 9 το πολύ κόμβων με τη χρήση ενός τέτοιου πίνακα. Η λίστα προς το παρόν έχει πέντε κόμβους. Τα διακεκομμένα βέλη παριστάνουν σχηματικά την αλληλουχία των θέσεων του πίνακα (κόμβων):

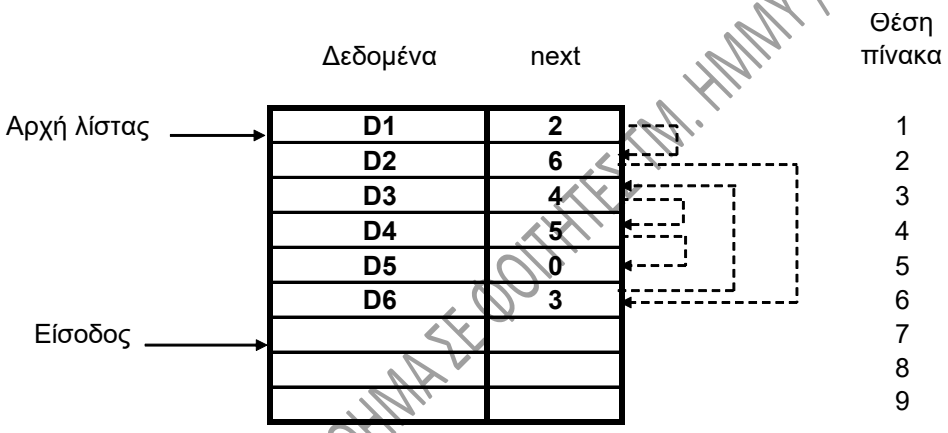



Σχ. 7.8

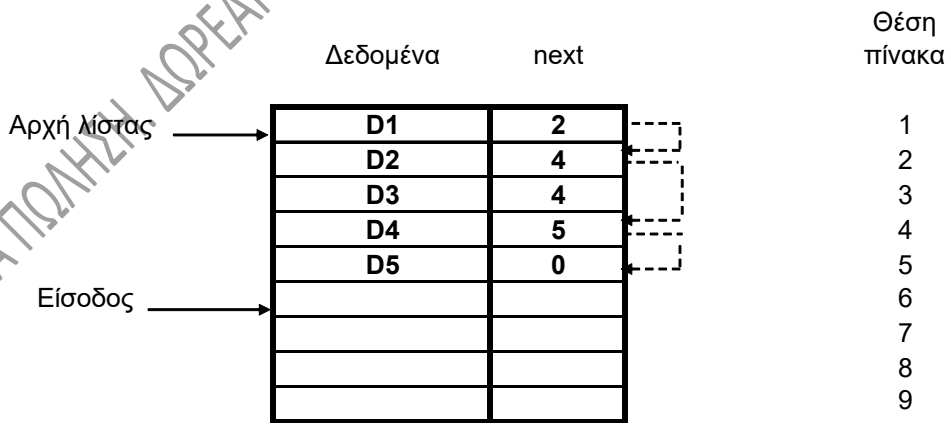
Ο αριθμός στην **«Θέση πίνακα»** στο πιο πάνω σχήμα αντιστοιχεί στον **αύξοντα αριθμό της θέσης των πινάκων** που χρησιμοποιούμε. Ο πίνακας με την ένδειξη **«Δεδομένα»** περιέχει τα **χρήσιμα δεδομένα**, τα περιεχόμενα των κόμβων. Ο πίνακας με την ένδειξη

**«next»** περιέχει το **πού βρίσκεται ο επόμενος κόμβος** για κάθε θέση του πίνακα. Ο αριθμός δηλαδή του πίνακα αυτού **αντιστοιχεί στον δείκτη** next **της δομής που περιγράφει τον κόμβο** στην προηγούμενη παράγραφο.

Η **«Αρχή λίστας»** είναι μια ένδειξη ή ένας δείκτης ή ένας ακέραιος κλπ που **δείχνει την αρχή των πινάκων**, ενώ η **«Είσοδος» δείχνει την πρώτη ελεύθερη θέση**, εκεί που θα μπει νέος κόμβος αν γίνει εισαγωγή στη λίστα, μετά την οποία η «Είσοδος» αυξάνει κατά 1. Το σχήμα 7.9α παρουσιάζει την κατάσταση των πινάκων αν εισαχθεί ένας νέος κόμβος, ο D6, μετά τον κόμβο D2, ενώ το 7.9β παρουσιάζει την κατάσταση αν διαγραφεί ο κόμβος D3 από την αρχική λίστα.




Σχ. 7.9α




Σχ. 7.9β

Οι συνεχείς προσθήκες και διαγραφές δεδομένων στην λίστα θα έχουν τελικά σαν αποτέλεσμα την **ύπαρξη ανεκμετάλλευτων κενών** στους πίνακες. Ο λόγος είναι ότι κάθε νέα προσθήκη θα γίνεται εκεί που δείχνει το «Είσοδος», το οποίο με κάθε

προσθήκη όπως είπαμε αυξάνει κατά 1. Αυτά τα ανεκμετάλλευτα κενά περιέχουν άχρηστα δεδομένα, τα οποία δεν μπορούμε πλέον να προσπελάσουμε, αφού καμμιά αναφορά και κανείς δείκτης δεν τα δείχνει.

Η απομάκρυνση των άχρηστων δεδομένων είναι αναγκαία για να μπορέσουμε να ελευθερώσουμε χώρο στους πίνακες. Για τον σκοπό αυτό πρέπει να υλοποιηθεί και να χρησιμοποιηθεί ένας **αλγόριθμος απομάκρυνσης αχρήστων**. Ένας τέτοιος αλγόριθμος μπορεί να περιλαμβάνει για παράδειγμα την **χρήση ενός δεύτερου ζεύγους πινάκων**. Στους πίνακες αυτούς και **σε διαδοχικές θέσεις μεταφέρουμε μόνο τους κόμβους που χρησιμοποιούνται και διαγράφουμε μετά τους αρχικούς πίνακες**. Ένας άλλος τρόπος περιλαμβάνει την **τοποθέτηση σε μια στοίβα της θέσης κάθε κόμβου που ακυρώνεται**. Όταν δημιουργηθεί ένας **νέος κόμβος**, αυτός **καταλαμβάνει τη θέση ενός ακυρωμένου**, αν υπάρχει, ανασύροντας από τη στοίβα τη θέση όπου θα τοποθετηθεί. **Αν δεν υπάρχει ακυρωμένος κόμβος, τότε ο νέος κόμβος τοποθετείται στην επόμενη θέση των πινάκων** (εκεί που δείχνει το «Είσοδος» στα πιο πάνω σχήματα.



## 7.6. ΔΙΠΛΑ ΣΥΝΔΕΔΕΜΕΝΗ ΛΙΣΤΑ.

Στο πρόγραμμα που ακολουθεί δίνουμε κώδικα σε C για την δημιουργία και επεξεργασία μιας διπλά συνδεδεμένης λίστας. Το πρόγραμμα, όπως και αυτό της υποτίθεται ότι σχετίζεται με τη γραμματεία κάποιας Σχολής και δημιουργεί μια διπλά συνδεδεμένη λίστα, σε κάθε κόμβο της οποίας υπάρχουν το ονοματεπώνυμο ενός φοιτητή της Σχολής και ο αριθμός των οφειλόμενων μαθημάτων για την λήψη του πτυχίου. Η εισαγωγή στοιχείων τερματίζεται εάν δοθεί ως όνομα ένα «μηδενικό» όνομα, εάν δηλαδή πληκτρολογήσουμε το Enter ως συμβολοσειρά εισόδου. Στην οθόνη εμφανίζονται τα περιεχόμενα της λίστας τόσο κατά την «ορθή», όσο και κατά την «αντίστροφη» φορά.

Δίνεται επίσης στη συνέχεια -ενδεικτικά- μια συνάρτηση για την εισαγωγή ενός κόμβου σε μια διπλά συνδεδεμένη λίστα.

**7.6.1. Δημιουργία – αρχικοποίηση διπλά συνδεδεμένης λίστας.**

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

#define N 1000

struct dnode {
      char student [30];
      int  lessons;
      struct  dnode *left;
      struct  dnode *right; };

int main(void) {
      struct dnode *head, *ptr, *tail;

         /*Δημιουργία – εισαγωγή στοιχείων στη λίστα */

      puts ("DWSE ONOMA FOITHTH");
      head = (struct dnode *) malloc (sizeof (struct dnode));
      ptr = head;                                                  /* Γραμμή 15 */
      ptr -> right = NULL;
      ptr -> left = NULL;
      gets(ptr -> student);
      while (strlen (ptr -> student) > 0) {
            puts ("DWSE ARI8MO YPOLEIPOMENWN MA8HMATWN");
            scanf ("%d", &ptr -> lessons);
            fflush (stdin);
            ptr -> right = (struct dnode *) malloc (sizeof (struct dnode));
            ptr -> right -> left = ptr;
            ptr = ptr -> right;                                    /* Γραμμή 25 */
            ptr -> right = NULL;
            puts ("DWSE ONOMA FOITHTH");
            gets(ptr -> student); }

      tail = ptr -> left;                                          /* Γραμμή 29 */
      tail -> right = NULL;
      ptr -> left = NULL;
      free (ptr);
      ptr = NULL;                                                  /* Γραμμή 33 */

         /*Εμφάνιση των στοιχείων της λίστας στην οθόνη κατά την ορθή φορά*/

      ptr =head;
      while (ptr!= NULL) {
            printf("%s %d\n", ptr->student, ptr->lessons);
            ptr = ptr -> right; }

   /*Εμφάνιση των στοιχείων της λίστας στην οθόνη κατά την αντίστροφη φορά*/

      ptr = tail;
      while (ptr!= NULL) {
            printf("%s %d\n", ptr->student, ptr->lessons);
            ptr = ptr -> left; }

      return 1; }
```

**Σχόλια:**

- Η περιγραφή της δομής για κάθε κόμβο της λίστας δίνεται από την:

```
struct dnode {
    char student [30];
    int  lessons;
    struct dnode *left;
    struct dnode *right; };
```

Η δομή μπορεί να έχει οποιοδήποτε πλήθος πεδίων, όμως **δύο από τα πεδία της πρέπει να είναι δείκτες προς δομή του ίδιου τύπου** (εδώ αυτά είναι τα πεδία left και right). Από τους δείκτες αυτούς, **ο ένας δείχνει το επόμενο στοιχείο της λίστας και ο άλλος το προηγούμενο**.

- Είναι προφανές ότι στη λίστα μπορούμε να κινηθούμε **και προς τις δύο κατευθύνσεις**. Τοποθετήσαμε ένα δείκτη στο ένα άκρο της λίστας (head) και ένα δείκτη στο άλλο άκρο (tail), ώστε να μπορούμε να ξεκινήσουμε την προσπέλαση από όποιο άκρο θέλουμε με τη βοήθεια του ptr.

- **Ο δείκτης ptr** μετακινείται από κόμβο σε κόμβο και είναι αυτός που χρησιμοποιείται για να μπορέσουμε τελικά να διατρέξουμε όλη τη λίστα. Αρχικά, στην γραμμή 15, ο ptr τοποθετείται στο «αριστερό» άκρο της λίστας και γίνεται ίσος με τον head. Η μετακίνησή του από τον ένα κόμβο στον άλλο γίνεται στη γραμμή 25 του προγράμματος με την εντολή:

```
ptr = ptr->right;
```

- **Ο δείκτης right του κόμβου στο ένα άκρο της λίστας και ο δείκτης left του κόμβου στο άλλο άκρο της λίστας** έχουν τεθεί **ίσοι με NULL**. Αυτό είναι φυσικό, αφού η λίστα τερματίζεται και προς τις δυο κατευθύνσεις. Η εμφάνιση των στοιχείων της λίστας γίνεται με τη χρήση ενός βρόχου while, πριν από τον οποίο **ο ptr έχει πάρει τιμή ίση με head ή με tail**, ανάλογα με τον κόμβο από τον οποίο ξεκινούμε. Μέσα στο while ο **ptr μετακινείται** με προς την μια ή την άλλη κατεύθυνση, **ακολουθώντας αντίστοιχα τους δείκτες right ή left**.

- Όπως είναι διαρθωμένο το πρόγραμμα, δημιουργείται ένας κόμβος παραπάνω από όσους χρειαζόμαστε. Από την γραμμή 29 έως την γραμμή 33 του προγράμματος ελευθερώνουμε τον χώρο τον οποίο καταλαμβάνει ο κόμβος αυτός.

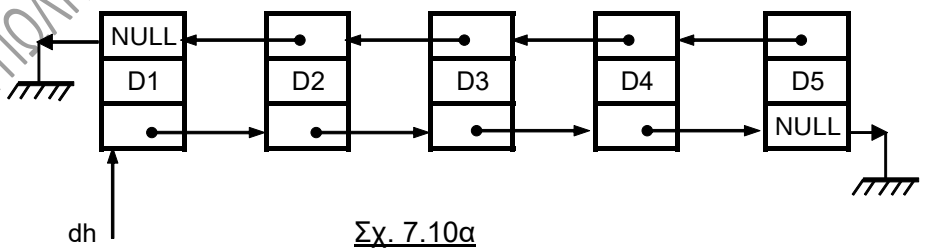
- Δεδομένου ότι η προσπέλαση όλων των κόμβων μιας διπλά συνδεδεμένης λίστας μπορεί να γίνει αν κινηθούμε από κάποιο κόμβο της προς τις δύο κατευθύνσεις, **δεν είναι απαραίτητο να γνωρίζουμε αναγκαστικά ένα δείκτη στον πρώτο**

**κόμβο της λίστας, αλλά να γνωρίζουμε ένα δείκτη σε κάποιο κόμβο της λίστας**. Άρα **δεν θεωρείται δεδομένη η ύπαρξη κεφαλής της λίστας**. Εάν θέλουμε να διατρέξουμε ολόκληρη την λίστα κινούμενοι από το ένα άκρο, πρέπει πρώτα να εξασφαλίσουμε ότι ο δείκτης μέσω του οποίου γνωρίζουμε την λίστα έχει μετακινηθεί στο άκρο αυτό.

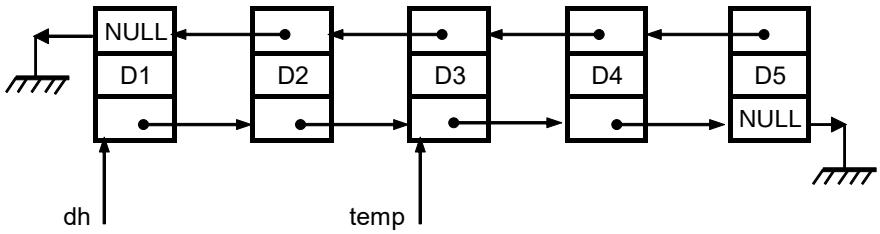
### 7.6.2. Εισαγωγή κόμβου σε διπλά συνδεδεμένη λίστα.

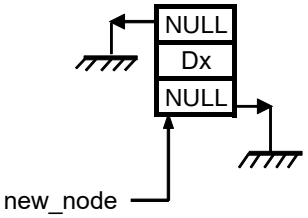
Παρακάτω θα δούμε ένα αλγόριθμο **εισαγωγής νέου κόμβου μετά από κάποιον δεδομένο κόμβο σε μια διπλά συνδεδεμένη λίστα**. Σχηματικά, οι ενέργειές μας συνοψίζονται στα βήματα που περιγράφει το σχ. 7.10, θεωρώντας ότι η εισαγωγή του νέου κόμβου θα γίνει μεταξύ των κόμβων 3 και 4.

Η δημιουργία και εισαγωγή του νέου κόμβου γίνεται με τη χρήση της συνάρτησης dinsertion( ) που ακολουθεί. Η συνάρτηση δέχεται ως **ορίσματα ένα δείκτη στο ένα άκρο της λίστας,** τον dh **και ένα ακέραιο**, ο οποίος ισούται με τον αύξοντα αριθμό του κόμβου, μετά από τον οποίο θα γίνει η εισαγωγή. Οι κόμβοι της λίστας (σχ. 7.10α) είναι του τύπου dnode της προηγουμένης παραγράφου. Η συνάρτηση χρησιμοποιεί **ένα τοπικό δείκτη,** τον temp, ώστε μέσω αυτού **να μπορούμε να έχουμε προσπέλαση στους διάφορους κόμβους της λίστας**. Με τη χρήση του for, ο δείκτης temp τοποθετείται στον κόμβο μετά τον οποίο θα γίνει η εισαγωγή (σχ. 7.10β). Στα σχήματα 7.10γ, 7.10δ, 7.10ε και 7.10στ, παριστάνονται σχηματικά οι εντολές των γραμμών 10, 11, 12 και 13 αντίστοιχα της συνάρτησης dinsertion( ).

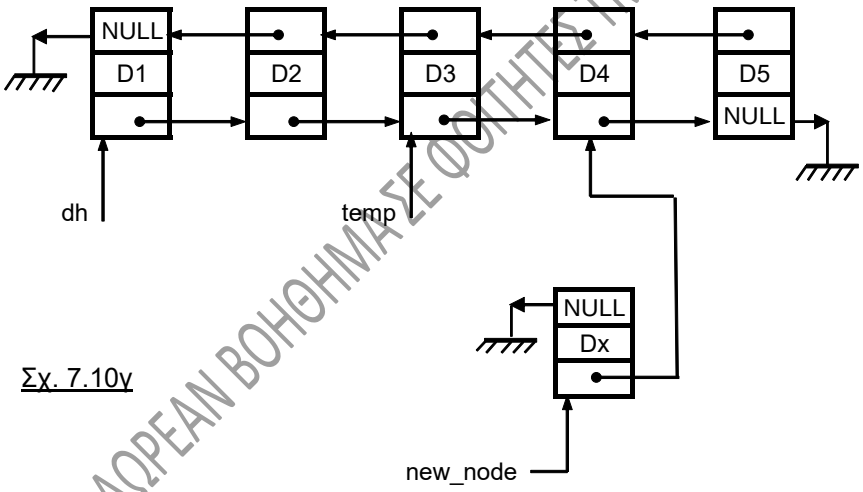



Σχ. 7.10α

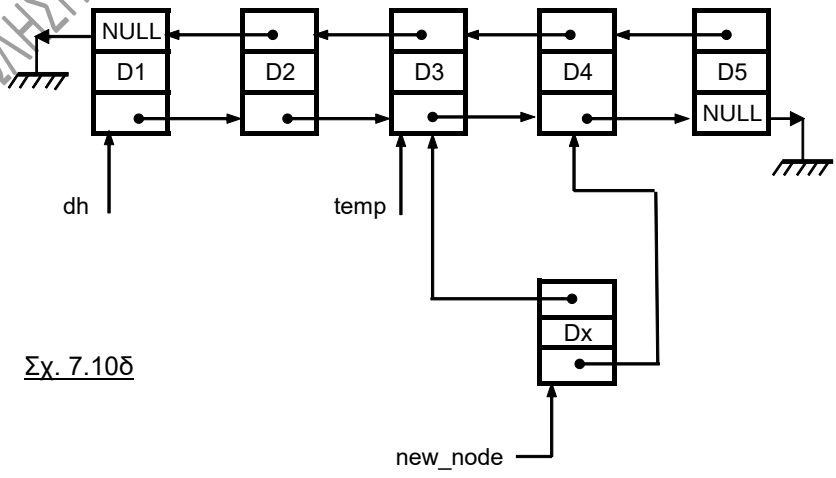


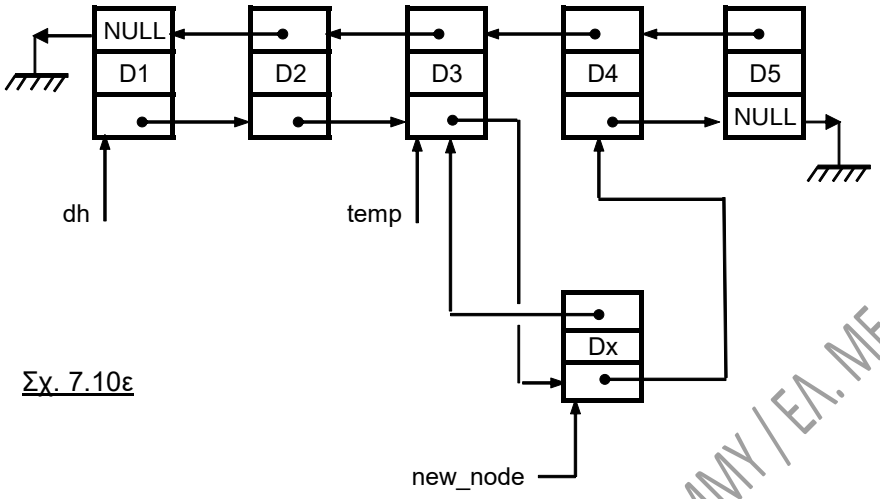

Σχ. 7.10β



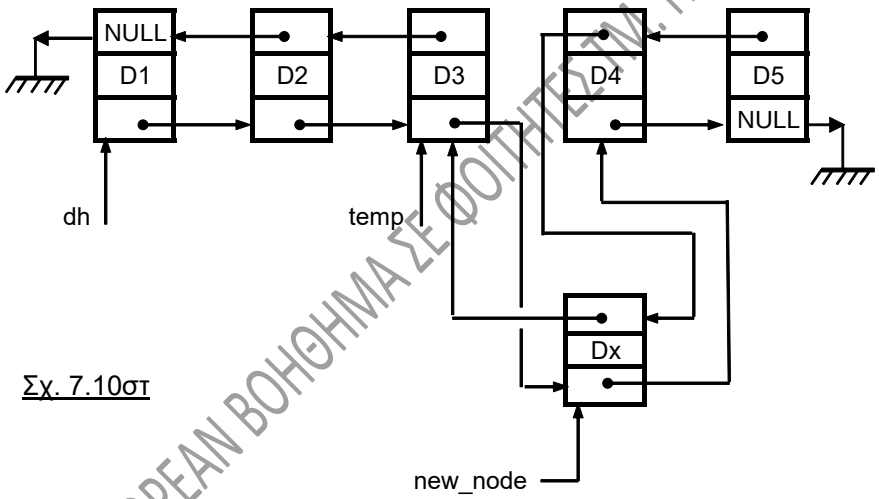

Σχ. 7.10γ




Σχ. 7.10δ




Σχ. 7.10ε




Σχ. 7.10στ



```
void dinsertion(struct dnode *dh, int dpos) {
    struct dnode *temp, *new_node;
    int k;

    temp = dh;
    for (k=1; k<dpos; k++)
        temp = temp -> right;
    new_node = (struct dnode *) malloc (sizeof (struct dnode));
    new_node -> right = temp -> right;                    /* Γραμμή  8 */
    new_node -> left = temp;                              /* Γραμμή  9 */
    temp -> right = new_node;                             /* Γραμμή 10 */
    new_node -> right -> left = new_node;                 /* Γραμμή 11 */
    scanf ("%d", & new_node -> lessons);
}
```

Αν στη main( ) ο δείκτης που βρίσκεται στο άκρο της λίστας λέγεται head, τότε η **κλήση της συνάρτησης** από τη main( ) για την εισαγωγή κόμβου στην θέση 3 της λίστας θα είναι:

```
dinsertion (head, 3);
```

η δε **δήλωση της συνάρτησης** θα είναι προφανώς:

```
void dinsertion(struct dnode *, int);
```

## 7.7. ΕΦΑΡΜΟΓΕΣ.

### 7.7.1. Υλοποίηση στοίβας με διπλά συνδεδεμένη λίστα.

Το παρακάτω πρόγραμμα υλοποιεί μια **στοίβα (stack) με τη χρήση διπλά συνδεδεμένης λίστας**. Θεωρούμε, για λόγους ευκολίας, ότι στη στοίβα θα τοποθετούνται μόνο θετικοί ακέραιοι. Ακολουθούν μερικά σχόλια για την ευκολότερη κατανόηση του προγράμματος:

- Ο δείκτης bot δείχνει τη βάση της στοίβας και δεν χρησιμοποιείται πουθενά. Αυτός που χρησιμοποιείται είναι **ο δείκτης top, ο οποίος δείχνει** κάθε φορά **τον κόμβο στον οποίο θα γίνει εισαγωγή στοιχείου**, και ο οποίος έχει δηλωθεί ως εξωτερική μεταβλητή στο πρόγραμμα, άρα γνωστός και επηρεαζόμενος από όλες τις συναρτήσεις του προγράμματος. Ο δείκτης αυτός, **εάν πρόκειται να ανακαλέσουμε κάποιο στοιχείο από την στοίβα, μετακινείται στον προηγούμενο κόμβο και μετά παίρνουμε τα περιεχόμενα** του κόμβου. Η ίδια λογική εφαρμόστηκε και στην υλοποίηση μιας στοίβας στο κεφάλαιο 5.

- Το πρόγραμμα **διαβάζει χαρακτήρες** από το πληκτρολόγιο, **μέχρι να πατηθεί ο χαρακτήρας 's'** (τερματίζεται δηλαδή με το s). **Αν ο χαρακτήρας είναι το 'u'**, τότε ζητάει ένα ακέραιο, τον οποίο τον **καταχωρεί στη στοίβα** με τη χρήση της συνάρτησης push( ). **Αν ο χαρακτήρας είναι το 'o'** τότε **ανακαλεί από την στοίβα** τον τελευταίο ακέραιο, κάνοντας χρήση της συνάρτησης pop( ). Σε κάθε άλλη περίπτωση εμφανίζεται το μήνυμα «Αδύνατη ενέργεια».

- Στο πρόγραμμά μας, η συνάρτηση pop( ) **όταν η στοίβα είναι άδεια, επιστρέφει τιμή -1**. Δεδομένου ότι η στοίβα περιέχει μόνο θετικούς ακέραιους, η τιμή αυτή είναι ενδεικτική λάθους για ανάκληση από άδεια στοίβα.

- Όπως αναφέρθηκε και πιο πάνω, σε κάθε περίπτωση και για λόγους πρόσθετης ασφάλειας, **πριν από τη χρήση της συνάρτησης free( ), καλό είναι να έχουν «γειωθεί» οι δείκτες που ξεκινούν από την περιοχή μνήμης που πρόκειται να αποδεσμευτεί**.

- Η υλοποίηση της στοίβας μπορεί να γίνει φυσικά και με τη χρήση απλά συνδεδεμένης λίστας. Διαλέξαμε τη διπλά συνδεδεμένη εδώ και την απλά συνδεδεμένη στην επόμενη εφαρμογή για υλοποίηση μιας ουράς.

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

void push (int);
int pop (void);

struct stak_node {
      int  elem;
      struct stak_node *up;
      struct stak_node *down; };

struct stak_node *top;

int main(void) {
      struct stak_node *bot;
      int  ak;
      char ch, pak[5];

      bot = (struct stak_node *) malloc (sizeof (struct stak_node));
      top = bot;
      top -> down = top -> up = NULL;
      ch = getche( );
      while (ch != 's') {
            if (ch == 'u') {
                  puts ("DWSE AKERAIO GIA EISAGWGH STHN STOIBA");
                  gets (pak);
                  ak = atoi (pak);
                  push (ak); }
            else
                  if (ch == 'o') {
                        ak = pop ( );
                        printf ("%d\n", ak);
                        if (ak == -1)
                              break; }
                  else
                        puts ("LATHOS. DWSE ALLO XARAKTHRA (u / o) ");
                        ch = getche ( ); }
      return 1;
}
```

```
void push (int uak) {
      top -> elem = uak;
      top -> up = (struct stak_node *) malloc (sizeof (struct stak_node));
      top -> up -> up = NULL;
      top -> up -> down = top;
      top = top ->up;
}

int pop( ) {
      if (top -> down != NULL) {
            top = top -> down;
            top -> up -> down = NULL;
            free (top -> up);
            top -> up = NULL;
            return (top -> elem); }
      else {
            puts ("STOIBA ADEIA");
            return -1; }
}
```



### 7.7.2. Υλοποίηση ουράς με απλά συνδεδεμένη λίστα.

Το παρακάτω πρόγραμμα υλοποιεί μια **ουρά (queue) με τη χρήση απλά συνδεδεμένης λίστας**. Στην ουρά αυτή, χάριν ευκολίας, υποθέτουμε ότι τοποθετούνται μόνο θετικοί ακέραιο. Η «φιλοσοφία» που ακολουθήθηκε είναι σκόπιμα διαφορετική από την περίπτωση της στοίβας της παραγράφου 7.7.1. Ακολουθούν μερικά σχόλια για πληρέστερη κατανόηση:

- Δεν χρησιμοποιήθηκαν εξωτερικοί δείκτες. Όλοι οι δείκτες έχουν δηλωθεί τοπικά, μέσα στις συναρτήσεις.
- Το παραπάνω έχει σαν συνέπεια την ανάγκη να «περνάνε» οι δείκτες σαν ορίσματα στις διάφορες συναρτήσεις, αφού δεν είναι γνωστοί διαφορετικά.
- Το πρόγραμμα **διαβάζει χαρακτήρες** από το πληκτρολόγιο, **μέχρι να πατηθεί ο χαρακτήρας 's'** (τερματίζεται δηλαδή με το s)**. Αν ο χαρακτήρας που εισάγεται είναι το 'u'**, τότε το πρόγραμμα περιμένει να διαβάσει ένα ακέραιο, τον οποίο τον **καταχωρεί στην ουρά** με τη χρήση της συνάρτησης qinsert( ). **Αν ο χαρακτήρας είναι το 'o'** τότε **ανακαλείται από την ουρά** ο τελευταίος ακέραιος, κάνοντας χρήση της συνάρτησης qremove( ). Σε κάθε άλλη περίπτωση εμφανίζεται το μήνυμα «Λάθος. Δώστε άλλο χαρακτήρα».

- Η συνάρτηση qinsert( ) δέχεται ως ορίσματα τον ακέραιο που πρόκειται να εισαχθεί, καθώς και ένα δείκτη που δείχνει τον κόμβο εισαγωγής.

- Η συνάρτηση qremove( ) δέχεται ως ορίσματα ένα δείκτη που δείχνει τον κόμβο στον οποίο γίνεται η εισαγωγή, ένα δείκτη στον κόμβο από όπου γίνεται η εξαγωγή στοιχείων και ένα δείκτη σε ακέραιο, ο οποίος ακέραιος θα παίρνει τιμή όση η εξαγόμενη από την ουρά τιμή. **Εάν η ουρά είναι άδεια, τότε η τιμή που θα επιστρέφεται από την ουρά θα είναι ίση με -1** (θυμηθείτε ότι θεωρήσαμε πως στην ουρά αποθηκεύονται μόνο θετικές τιμές, άρα μόνο θετικές τιμές θα επιστρέφονται εάν η ουρά δεν είναι άδεια).

- Και οι δυο συναρτήσεις έχουν τιμή επιστροφής δείκτη σε δομή qnode. Έτσι, **μετά την εισαγωγή στοιχείου** στην ουρά μέσω της qinsert( ), **ο δείκτης εισαγωγής μετακινείται στη νέα του θέση**, ενώ **μετά την ανάκτηση στοιχείου** με την χρήση της qremove( ), **ο δείκτης ανάκτησης μετακινείται στη νέα του θέση**. Αυτές οι μετακινήσεις γίνονται αντίστοιχα στις γραμμές 22 και 25 του προγράμματος. Οι εντολές:

```
ip = ip -> next;
rp = rp -> next;
```

μετακινούν τους τοπικούς δείκτες εισαγωγής και ανάκτησης αντίστοιχα μέσα στις συναρτήσεις qinsert( ) και qremove( ).



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

struct qnode {
     int elem;
          struct qnode *next; };

struct qnode *qinsert (int, struct qnode *);
struct qnode *qremove (int *, struct qnode *, struct qnode *);

int main (void) {
     struct qnode *head, *tail;
     int  ak, rak;
     char ch, pak[5];

     head=(struct qnode *)malloc (sizeof (struct qnode));
     tail=head;
     tail->next = NULL;
     ch = getche( );
```

```
        while (ch != 's') {
            if (ch == 'u') {
                puts ("DWSTE AKERAIO GIA EISAGWGH STHN OYRA");
                gets (pak);
                ak = atoi (pak);
                tail = qinsert (ak, tail); }                    /* Γραμμή 22 */
            else
                if (ch == 'o') {
                    head = qremove (&rak, head, tail);      /* Γραμμή 25 */
                    printf ("%d\n", rak);
                    if (rak == -1)
                        break; }
                else
                    puts ("LA8OS. DWSTE ALLO XARAKTHRA (u / o) ");
                    ch = getche( ); }
        return 1;
}

struct qnode *qinsert (int value, struct qnode *ip) {
        struct qnode *new_node;

        ip -> elem = value;
        new_node = (struct qnode *) malloc(sizeof(struct qnode));
        new_node -> next = NULL;
        ip -> next = new_node;
        ip = ip -> next;
        return ip;
}


struct qnode *qremove(int *value, struct qnode *rp, struct qnode *ip ) {
        struct qnode *dummy;

          if (rp == ip) {
                puts("Ουρά άδεια");
                *value = -1; }
        else {
                *value = rp -> elem;
                dummy = rp;
                rp = rp -> next;
                dummy -> next = NULL;
                free (dummy); }
        return rp;
}
```

# ΚΕΦΑΛΑΙΟ 8

# ΔΕΝΤΡΑ

## 8.1. ΓΕΝΙΚΑ.

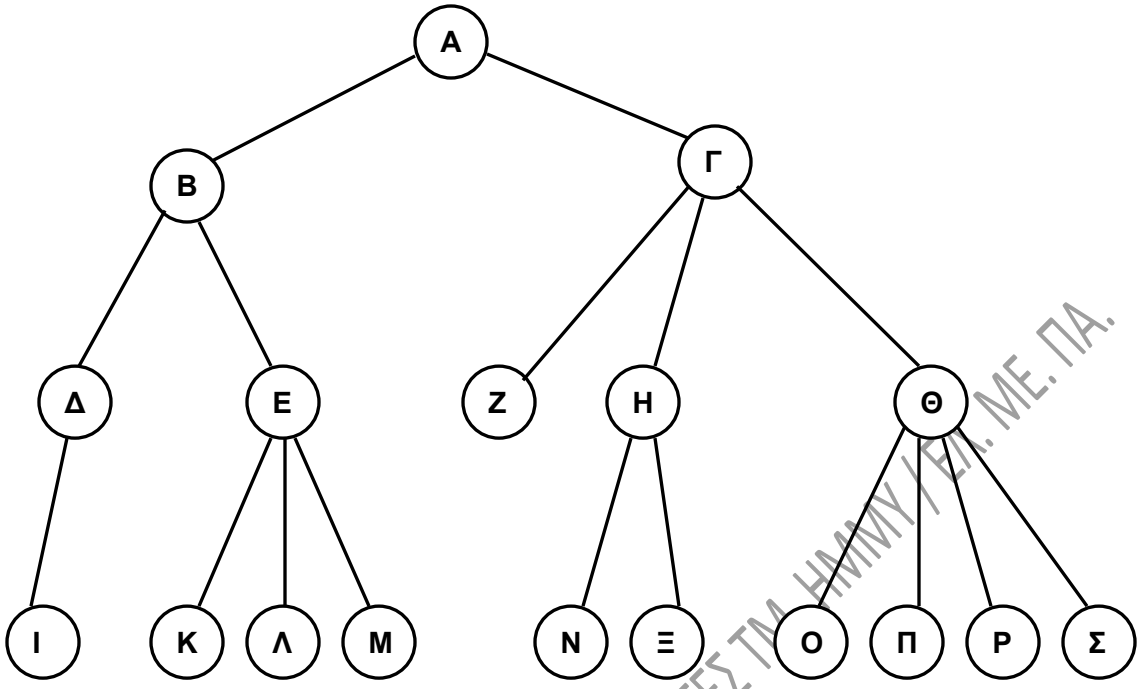
Οι δομές που εξετάσαμε μέχρι τώρα (πίνακες, στοίβες, ουρές) αποτελούν χαρακτηριστικά παραδείγματα **γραμμικών δομών**. Στις γραμμικές δομές, τα δεδομένα είναι γραμμικά διατεταγμένα, δηλαδή **κάποιο στοιχείο είναι πρώτο και κάποιο τελευταίο**, ενώ **για οποιοδήποτε υπάρχει ένα προηγούμενο και ένα επόμενο**. Στις **μη γραμμικές δομές** οι σχέσεις μεταξύ των δεδομένων είναι περισσότερο περίπλοκες. Οι δομές αυτού του είδους, με τις οποίες θα ασχοληθούμε εδώ, είναι τα δέντρα. Στα δέντρα κάθε στοιχείο έχει ένα μόνο προηγούμενο, αλλά μπορεί να έχει πολλά επόμενα στοιχεία.

Δέντρο είναι ένα **σύνολο κόμβων που συνδέονται με ακμές**. Οι ακμές, τα ευθύγραμμα τμήματα που φαίνονται στο παρακάτω σχ. 8.1, δείχνουν τη σχέση, τη σύνδεση μεταξύ των διαφόρων κόμβων, **ποιος** δηλαδή **κόμβος συνδέεται με ποιον**. Συνήθως, η ακμή είναι ένας δείκτης που οδηγεί από τον ένα κόμβο στον άλλο. Ένας από τους κόμβους λέγεται **ρίζα (root)** και αποτελεί την «αρχή» του δέντρου. Σε κάθε κόμβο του δέντρου καταλήγει μια μόνο ακμή, εκτός από τη ρίζα, στην οποία δεν καταλήγει καμμία. **Από την ρίζα μόνο ξεκινούν ακμές**. Από όλους γενικά τους κόμβους μπορούν να ξεκινούν καμμία, μια ή περισσότερες ακμές. Όπως στις απλά συνδεδεμένες λίστες πρέπει να γνωρίζουμε ανά πάσα στιγμή την κεφαλή, προκειμένου να γνωρίζουμε ολόκληρη την λίστα, έτσι και **στα δέντρα πρέπει να γνωρίζουμε ανά πάσα στιγμή την ρίζα**. Από αυτήν μπορούμε να μεταβούμε σε όλους τους κόμβους του δέντρου.

Ένα δέντρο μπορεί να παρασταθεί σχηματικά με διάφορους τρόπους. Το σχ. 8.1 δείχνει τον συνηθέστερο από αυτούς τους τρόπους:

## 8.2. ΟΡΟΛΟΓΙΑ.

Για να είναι εύκολη η αναφορά στους κόμβους του δέντρου και στις σχέσεις μεταξύ τους, είναι απαραίτητη η υιοθέτηση κάποιας ορολογίας.




Σχ. 8.1



- Ένας κόμβος x λέγεται **γονέας (parent)** των κόμβων y και z, όταν από τον x ξεκινούν ακμές, οι οποίες καταλήγουν στους y και z, οι οποίοι και λέγονται **παιδιά (children)** του x. Αν αναφερθούμε στο σχ. 8.1, ο κόμβος Α είναι ο γονέας των Β και Γ. Ομοίως ο Ε είναι ο γονέας των Κ, Λ και Μ, ενώ οι Ν και Ξ είναι παιδιά του κόμβου Η. Με την ίδια λογική, ο Π είναι **εγγονός** κόμβος του Γ, ενώ ο Β είναι **παππούς** του Κ.

- **Βαθμός (degree) ενός κόμβου** είναι **ο αριθμός των παιδιών του**. Έτσι, ο κόμβος Ε είναι βαθμού 3, ενώ ο Η είναι βαθμού 2. Βαθμός ορίζεται και για ολόκληρο το δέντρο. Ο **βαθμός ενός δέντρου** ισούται με τον **μέγιστο από τους βαθμούς των κόμβων του**. Το δέντρο του πιο πάνω σχήματος είναι ένα τετραδικό δέντρο. Πολύ συνηθισμένα δέντρα είναι τα **δυαδικά (binary)**, τα **τριαδικά (ternary)** και τα **τετραδικά (quadtree)**.

- Οι κόμβοι οι οποίοι **δεν έχουν παιδιά** λέγονται **τερματικοί** κόμβοι ή **φύλλα** (**terminal nodes** ή **leaves**) του δέντρου. Τέτοιοι κόμβοι είναι για παράδειγμα ο Z, ο Ν και ο Σ. Οι **μη τερματικοί** κόμβοι λέγονται επίσης και **εσωτερικοί (internal)** ή **κλαδιά (branches)**.

- Το **επίπεδο (level)** ενός κόμβου **ισούται με τον αριθμό των προγόνων του μέχρι τη ρίζα συν 1.** Το επίπεδο του κόμβου Ε είναι 3. Θα υιοθετήσουμε αυτή την προσέγγιση εδώ, αν και θα συναντήσετε και την λογική ότι η ρίζα είναι επιπέδου 0, άρα το επίπεδο ενός κόμβου είναι ίσο με τον αριθμό των προγόνων του.

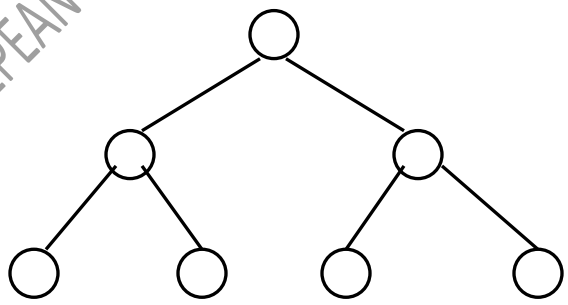
- Το **βάθος (depth)** ή **ύψος (height)** ενός δέντρου ισούται με το **μέγιστο επίπεδο των κόμβων του δέντρου.** Το ύψος του δέντρου του σχ. 8.1 είναι 4.

- Ένα δέντρο έχει πολλά **υποδέντρα (subtrees)** τα οποία προκύπτουν αν θεωρήσουμε κάθε φορά ως ρίζα ένα άλλο κόμβο πλην της πραγματικής αρχικής ρίζας.

- Ένα δέντρο λέγεται **πλήρες (complete)** όταν **περιέχει τον μέγιστο δυνατό αριθμό κόμβων**. Ο μέγιστος αυτός αριθμός εξαρτάται από το βαθμό και το βάθος του δέντρου. **Το πλήθος m των κόμβων** στη γενική περίπτωση **δέντρου βαθμού p και ύψους k δίνεται από τον τύπο:**

$$m = \frac{p^k - 1}{p - 1}$$

Έτσι, ένα πλήρες δυαδικό δέντρο βάθους k έχει $2^k - 1$ κόμβους. Το σχ. 8.2 δείχνει ένα πλήρες δυαδικό δέντρο ύψους 3, το οποίο όπως προκύπτει και από τον πιο πάνω τύπο, έχει 7 κόμβους.



Σχ. 8.2

- **Μήκος διαδρομής (path length)** για ένα κόμβο λέγεται **ο αριθμός των ακμών από τη ρίζα ως τον κόμβο.** Το μήκος διαδρομής **ισούται με το επίπεδο του κόμβου μείον 1**. Το μήκος διαδρομής για τον κόμβο Μ του σχ. 8.1 είναι 3.

Ένα δέντρο είναι μια ειδική μορφή συνδεδεμένης λίστας, σε αυτό δε εφαρμόζονται όλα όσα ξέρουμε για τις λίστες, όπως παρεμβολή, διαγραφή κλπ. Μια από τις δυσκολίες τις

οποίες καλείται κανείς να αντιμετωπίσει στα δέντρα είναι η **αναδρομικότητα** των περισσοτέρων συναρτήσεων που χρησιμοποιούνται. Αυτό είναι δικαιολογημένο, αν σκεφθούμε ότι **το δέντρο είναι από τη φύση του αναδρομική δομή δεδομένων**: κάθε υποδέντρο είναι επίσης δέντρο. Ένα δέντρο χρησιμοποιείται λοιπόν για την αναπαράσταση των δεδομένων, τα οποία βρίσκονται στους κόμβους, αλλά και των συσχετίσεων μεταξύ τους, δηλαδή των ακμών, ενώ η ολοκλήρωση των ανώτερων στοιχείων εξαρτάται από τα κατώτερα.

## 8.3. ΔΥΑΔΙΚΑ ΔΕΝΤΡΑ.

Μια ειδική μορφή δέντρων, αλλά ίσως η πιο χρήσιμη και ενδιαφέρουσα είναι τα **δυαδικά δέντρα**, αυτά δηλαδή που έχουν **βαθμό 2**, άρα δέντρα στα οποία **κανένας κόμβος τους δεν έχει περισσότερα από δύο παιδιά**. Τα δυαδικά δέντρα είναι πολύ χρήσιμα για τη δημιουργία μοντέλων συνδυασμού δύο στοιχείων, τα οποία παράγουν ένα νέο στοιχείο, το οποίο χρησιμοποιείται μαζί με κάποιο άλλο για τη δημιουργία ενός νέου στοιχείου κ.ο.κ.

### 8.3.1. Διάσχιση του δέντρου.

Μια σημαντική λειτουργία, την οποία καλούμαστε συχνά να εφαρμόσουμε σε ένα δυαδικό δέντρο είναι η **διάσχιση** του **(traverse)**. Με αυτό τον όρο εννοούμε την επίσκεψη κάθε κόμβου του δέντρου ακριβώς μια φορά. Υπάρχουν τρεις διαφορετικοί τρόποι για τη διάσχιση ενός δέντρου, σε καθένα από τους οποίους γίνονται και οι τρεις παρακάτω ενέργειες, με διαφορετική όμως σειρά σε κάθε περίπτωση:

α) Επίσκεψη της ρίζας.
β) Επίσκεψη του αριστερού υποδέντρου.
γ) Επίσκεψη του δεξιού υποδέντρου.

Ανάλογα με τη σειρά που εκτελούνται οι ενέργειες αυτές, έχουμε τους εξής τρόπους διάσχισης:

i) **Προδιατεταγμένος (Preorder):** σε αυτόν οι ενέργειες εκτελούνται με τη σειρά (α) – (β) – (γ). Η διάσχιση του δέντρου του σχ. 8.3 θα έδινε:

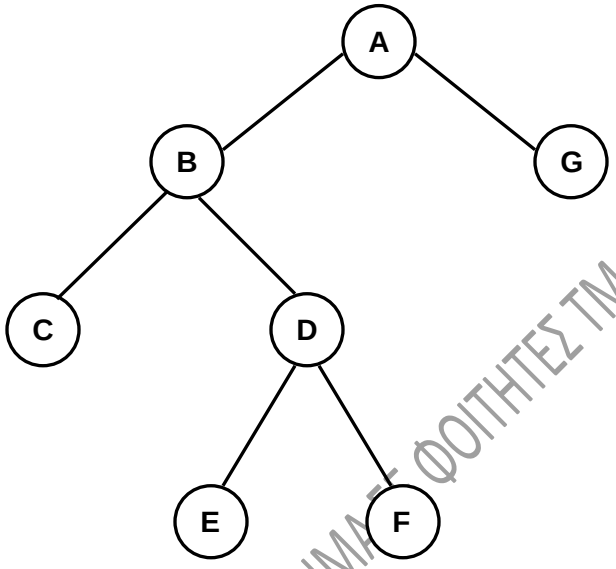
A B C D E F G

ii) **Ενδοδιατεταγμένος (Inorder):** σε αυτόν οι ενέργειες εκτελούνται με τη σειρά (β) – (α) – (γ). Η διάσχιση του δέντρου του σχ. 8.3 θα έδινε:

C B E D F A G

iii) **Μεταδιατεταγμένος (Postorder):** σε αυτόν οι ενέργειες εκτελούνται με τη σειρά (β) – (γ) – (α). Η διάσχιση του δέντρου του σχ. 8.3 θα έδινε:

C E F D B G A




Σχ. 8.3



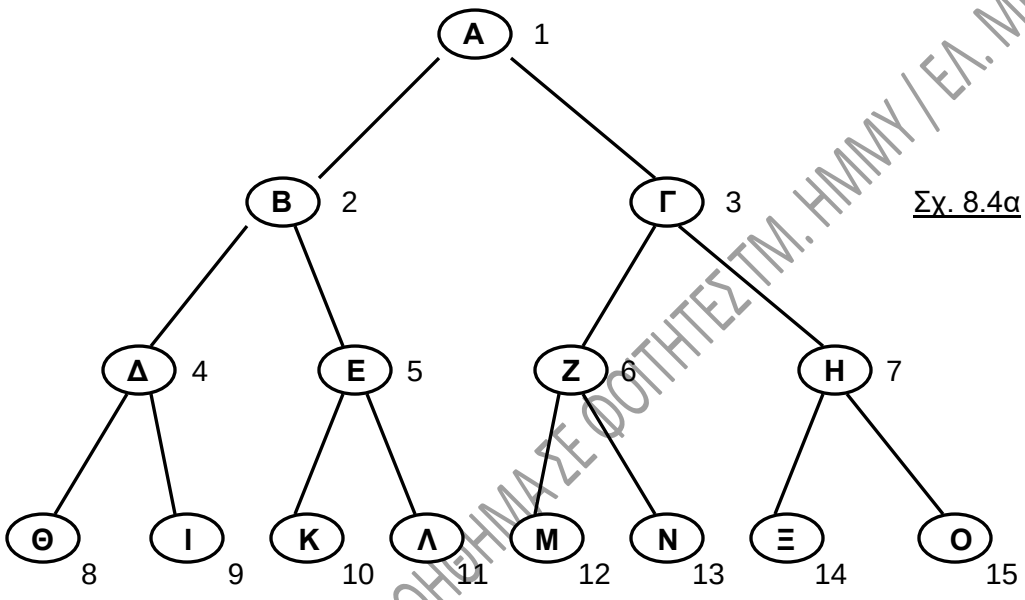
### 8.3.2. Υλοποίηση δυαδικών δέντρων με πίνακα.

Με τη χρήση των δέντρων αναπαρίστανται δεδομένα, δηλαδή τα περιεχόμενα των διαφόρων κόμβων, αλλά και οι συσχετίσεις μεταξύ τους (δηλαδή οι ακμές), τα οποία πρέπει να τηρούνται στη μνήμη του υπολογιστή. Παρουσιάζονται στη συνέχεια δυο τρόποι για την αποθήκευση αυτή.

**Α' τρόπος:**

Αυτός εφαρμόζεται κυρίως στα πλήρη δυαδικά δέντρα και κάνει **χρήση ενός μονοδιάστατου πίνακα**. Πρέπει να θυμηθούμε εδώ ότι σε ένα πλήρες δυαδικό δέντρο ύψους h, ο αριθμός των κόμβων του είναι $m = 2^h - 1$. Οι κόμβοι αριθμούνται ανά επίπεδο από αριστερά προς τα δεξιά και τοποθετούνται σε ένα πίνακα m θέσεων (τον pinax), όπως δείχνει το παράδειγμα του σχ. 8.4. Έτσι, τα παιδιά του κόμβου με αύξοντα αριθμό 5 είναι τα 10 και 11, ενώ ο κόμβος πατέρας του 5 είναι ο 2. Γενικά, για κάθε κόμβο x ισχύουν τα εξής:

- **Ο πατέρας του κόμβου x βρίσκεται στη θέση [x/2]**, αν x ≠ 1 (με το σύμβολο [ ] εννοούμε την ακέραια διαίρεση). Αν το x είναι 1, τότε ο κόμβος είναι η ρίζα.
- **Το αριστερό παιδί του κόμβου x βρίσκεται στη θέση 2x**, εάν 2x ≤ m. Εάν 2x > m, τότε ο κόμβος δεν έχει αριστερό παιδί.
- **Το δεξί παιδί του κόμβου x βρίσκεται στη θέση 2x+1**, εάν 2x+1 ≤ m. Εάν 2x+1 > m, τότε ο κόμβος δεν έχει δεξί παιδί.




Σχ. 8.4α

pinax

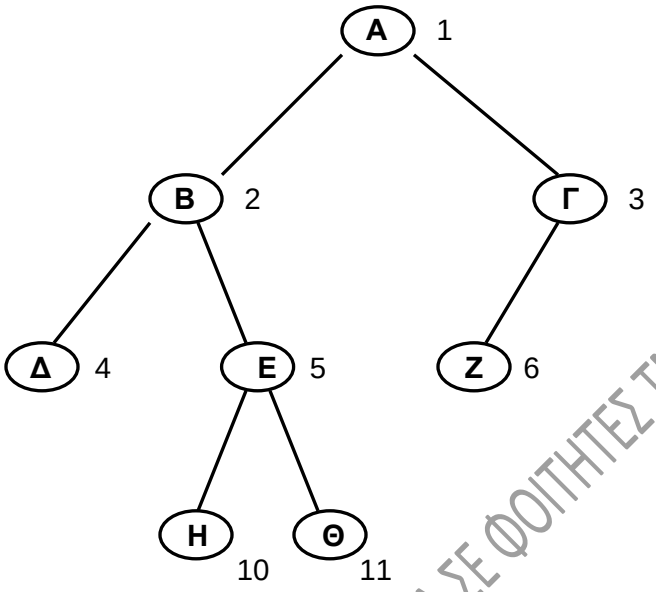
| Α | Β | Γ | Δ | Ε | Ζ | Η | Θ | Ι | Κ | Λ | Μ | Ν | Ξ | Ο |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Σχ. 8.4β

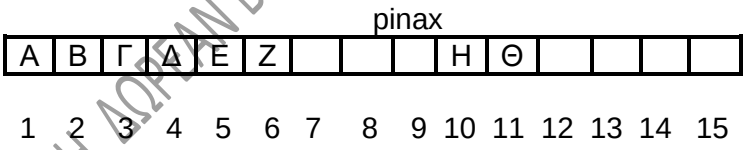


Με τον ίδιο τρόπο όπως και παραπάνω αποθηκεύονται **και μη πλήρη δέντρα**, όπως αυτό του σχ. 8.5α. Για την αρίθμηση των κόμβων του δέντρου αυτού **έχουν ληφθεί υπ' όψη και όσοι από τους κόμβους λείπουν, αλλά θα έπρεπε να υπάρχουν προκειμένου να έχουμε πλήρες δέντρο**:

Η αποθήκευση γίνεται στον πίνακα του σχ. 8.5β. Τώρα **δεν χρησιμοποιούνται όλες οι θέσεις του πίνακα**, παρά το ότι προφανώς οι θέσεις αυτές έχουν δεσμευτεί, άρα **υπάρχει σπατάλη μνήμης**. Η ποσότητα της μνήμης που σπαταλείται εξαρτάται από τη μορφή του δέντρου και είναι μεγαλύτερη όσο περισσότερο το δέντρο «απέχει» από το να είναι πλήρες.



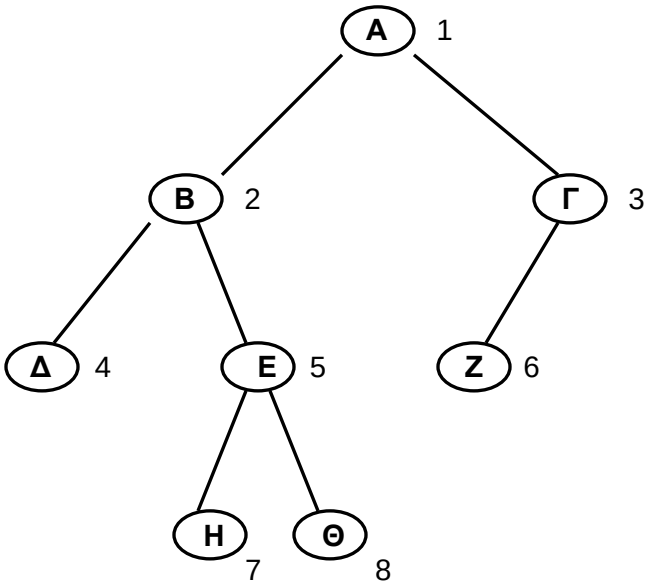

Σχ. 8.5α




Σχ. 8.5β

**Β' τρόπος:**

Για την αποφυγή του πιο πάνω προβλήματος της σπατάλης μνήμης, οι κόμβοι του δέντρου αποθηκεύονται και πάλι σε πίνακα, όμως εκτός από τα πραγματικά και «χρήσιμα» δεδομένα, αποθηκεύονται επίσης σε δύο ακόμη πίνακες **οι θέσεις του αριστερού και του δεξιού παιδιού** κάθε κόμβου. Η αρίθμηση των κόμβων του δέντρου γίνεται τώρα με τον τρόπο που φαίνεται στο σχ. 8.6α.




Σχ. 8.6α

Η **αποθήκευσή των κόμβων στη μνήμη** γίνεται με τον τρόπο που παρουσιάζεται στο σχ. 8.6β.

| data | left | right | Θέση πίνακα |
|---|---|---|---|
| A | 2 | 3 | 1 |
| B | 4 | 5 | 2 |
| Γ | 6 | 0 | 3 |
| Δ | 0 | 0 | 4 |
| E | 7 | 8 | 5 |
| Z | 0 | 0 | 6 |
| H | 0 | 0 | 7 |
| Θ | 0 | 0 | 8 |

Σχ. 8.6β

Στο σχήμα αυτό παριστάνονται τρεις πίνακες. Σε αυτόν που ονομάζεται “data” αποθηκεύονται τα χρήσιμα δεδομένα των κόμβων, στους δυο δε άλλους, τους πίνακες left και right αποθηκεύονται οι θέσεις του αριστερού και του δεξιού παιδιού αντίστοιχα κάθε κόμβου.

Αν υποθέσουμε ότι τα «χρήσιμα» δεδομένα κάθε κόμβου είναι ένας πίνακας χαρακτήρων 30 θέσεων, τότε ο κόμβος θα μπορούσε να περιγραφεί με τη χρήση μιας δομής του είδους komvos:

```
struct komvos {
    char data [30];
    unsigned a_paidi;
    unsigned d_paidi; };
```

**Ένα δυαδικό δέντρο** μπορεί να παρασταθεί ως ένας **πίνακας τέτοιων δομών**.

Το κύριο πρόβλημα σε αυτούς τους τρόπους αποθήκευσης του δέντρου είναι ότι οι πίνακες είναι σταθερών διαστάσεων, άρα **ο μέγιστος αριθμός κόμβων είναι εκ των προτέρων καθορισμένος**.

### 8.3.3. Υλοποίηση δυαδικών δέντρων με δείκτες.

**Η χρήση των δεικτών δεν περιορίζει το πλήθος των κόμβων του δέντρου**, όπως κάνει ο πίνακας της προηγούμενης παραγράφου. Για ένα δέντρο, στο οποίο τα χρήσιμα δεδομένα κάθε κόμβου είναι μια ακέραια τιμή, η περιγραφή της δομής του κόμβου θα του είδους komvos, όπως αυτή που ακολουθεί. Το δέντρο οργανώνεται έτσι ώστε ο **δείκτης lp κάθε κόμβου να δείχνει στο αριστερό παιδί του και ο δείκτης rp να δείχνει στο δεξί παιδί του**. Αν δεν υπάρχει αριστερό ή δεξί παιδί, οι δείκτες αυτοί έχουν αντίστοιχα τιμή NULL.

```
struct komvos
{
  int rec;
  struct komvos  *lp;
  struct komvos  *rp;
};
```

Ο τρόπος δημιουργίας, γεμίσματος και διάσχισης ενός δυαδικού δέντρου και οι συναρτήσεις που χρειάζονται, θα παρουσιαστούν στην επόμενη παράγραφο για μια ειδική κατηγορία δέντρων, τα δυαδικά δέντρα αναζήτησης.



## 8.4. ΔΥΑΔΙΚΑ ΔΕΝΤΡΑ ΑΝΑΖΗΤΗΣΗΣ.

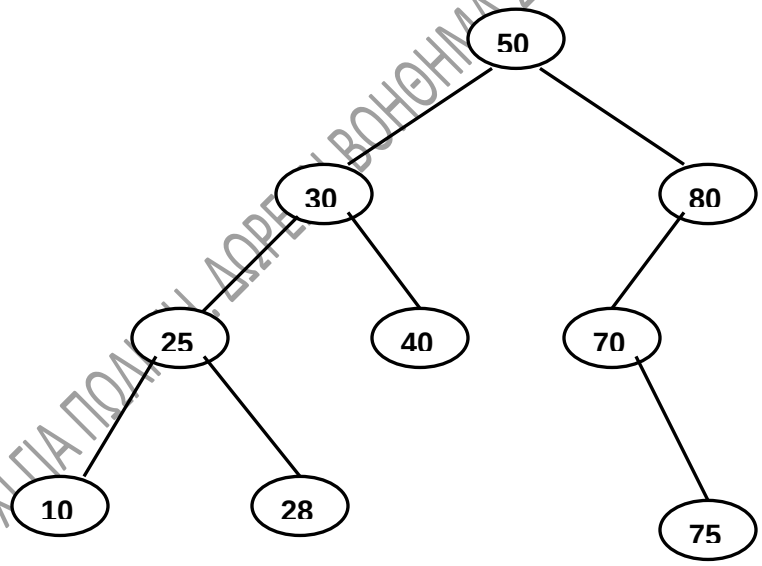
Τα δέντρα αυτά αποτελούν ειδική κατηγορία. Είναι δέντρα **διατεταγμένα**, δηλαδή δέντρα στα οποία **η διάταξη των παιδιών κάθε κόμβου,** το ποιο είναι το αριστερό και ποιο το δεξί παιδί, **έχει σημασία**. Στα δέντρα αναζήτησης **τα δεδομένα κάθε κόμβου έχουν**

**μεγαλύτερη τιμή από τα δεδομένα όλων των κόμβων που βρίσκονται στα αριστερά του** (όλων των αριστερών απογόνων του δηλαδή) **και μικρότερη από την τιμή των δεδομένων όλων των κόμβων που βρίσκονται δεξιά του** (όλων των δεξιών απογόνων του). Το δέντρο του σχ. 8.7 είναι δυαδικό δέντρο αναζήτησης.

Μπορεί κανείς εύκολα να παρατηρήσει, ότι **εάν διασχίσουμε ένα δυαδικό δέντρο αναζήτησης με τον ενδοδιατεταγμένο τρόπο**, θα προκύψει **αποτέλεσμα ταξινομημένο**. Η ταξινόμηση δεν ισχύει για τη διάσχιση του δέντρου με κάποιον από τους άλλους τρόπους.

Τέτοια δέντρα χρησιμοποιούνται όταν θέλουμε γρήγορη προσπέλαση δεδομένων, για τα οποία υπάρχει κατάταξη, τα οποία δηλαδή έχουν ήδη τοποθετηθεί στο δέντρο με βάση κάποιο **«κλειδί αναζήτησης»**. Με αυτό τον όρο εννοούμε ότι **από τα πιθανά πολλά «χρήσιμα» δεδομένα του κόμβου, κάποιο (το κλειδί) είναι εκείνο με βάση το οποίο τοποθετούνται ή αναζητούνται στοιχεία** μέσα στο δέντρο.

Αναφερόμαστε στη συνέχεια περιγραφικά σε τρεις αλγορίθμους για ένα δυαδικό δέντρο αναζήτησης και συγκεκριμένα στους αλγορίθμους αναζήτησης, εισαγωγής και διαγραφής κόμβου.




Σχ. 8.7

### 8.4.1. Αλγόριθμος αναζήτησης:

Ο αλγόριθμος με βάση τον οποίο επισημαίνουμε μια τιμή σε δυαδικό δέντρο αναζήτησης είναι ο εξής:

- **Συγκρίνουμε** την τιμή της ρίζας του δέντρου (το κλειδί της ρίζας) με την τιμή που αναζητούμε.
- **Αν η ζητούμενη τιμή είναι μικρότερη από το κλειδί της ρίζας**, τότε συνεχίζουμε στο αριστερό υποδέντρο.
- **Αν η ζητούμενη τιμή είναι μεγαλύτερη από το κλειδί της ρίζας**, τότε συνεχίζουμε στο δεξί υποδέντρο.
- Η αναζήτηση συνεχίζεται μέχρι να βρεθεί ο κόμβος με την τιμή που αναζητούμε ή μέχρι να φθάσουμε σε δείκτη με τιμή NULL, οπότε η αναζητούμενη τιμή δεν υπάρχει στο δέντρο.

### 8.4.2. Αλγόριθμος εισαγωγής:

Ο αλγόριθμος με βάση τον οποίο εισάγουμε μια τιμή σε δυαδικό δέντρο αναζήτησης είναι ο εξής:

- Αρχικά **ακολουθείται η διαδικασία αναζήτησης**, αναζητώντας ένα κόμβο με τιμή ίση με αυτή του κόμβου που πρόκειται να εισαχθεί. Τέτοια τιμή δεν υπάρχει και **καταλήγουμε σε δείκτη NULL**.
- **Δεσμεύεται μνήμη** για την τοποθέτηση της νέας τιμής και δημιουργείται ένας νέος κόμβος.
- Ο μηδενικός δείκτης στον οποίο καταλήξαμε πιο πάνω τοποθετείται έτσι ώστε να δείχνει στο νέο κόμβο.

### 8.4.3. Αλγόριθμος διαγραφής:

Ο αλγόριθμος με βάση τον οποίο διαγράφουμε ένα κόμβο από δυαδικό δέντρο αναζήτησης είναι πιο περίπλοκος. Συγκεκριμένα:

- **Αν ο υπό διαγραφή κόμβος είναι φύλλο** του δέντρου, τότε τον διαγράφουμε, μηδενίζοντας τον αντίστοιχο δείκτη ο οποίος ξεκινά από τον γονέα του.
- **Αν ο υπό διαγραφή κόμβος έχει ένα μόνο παιδί**, τότε αντικαθίσταται από το παιδί του.

- **Αν ο υπό διαγραφή κόμβος έχει δύο παιδιά**, τότε αντικαθίσταται είτε από τον πιο δεξιό κόμβο του αριστερού υποδέντρου, είτε από τον πιο αριστερό κόμβο του δεξιού υποδέντρου.

### 8.4.4. Δημιουργία, διάσχιση δέντρου. Υλοποίηση με C:

Στο πιο κάτω πρόγραμμα δημιουργείται και γεμίζει από το πληκτρολόγιο ένα δυαδικό δέντρο αναζήτησης. Σκοπός του προγράμματος αυτού είναι παράσχει ένα «εργαλείο» για άσκηση. Δεν είναι απαραίτητο να ασχοληθεί κάποιος με τον τρόπο υλοποίησης του δέντρου, άρα και με τα σχόλια που υπάρχουν στην συνέχεια. Αφού πάντως το δέντρο δημιουργηθεί, υπάρχει η δυνατότητα να γράψει κανείς τις δικές του συναρτήσεις, όπως αυτές που δίδονται έτοιμες στην συνέχεια. Κάθε συνάρτηση που γράφετε μπορεί να κληθεί στην main( ) αμέσως μετά την γραμμή 22 στο πρόγραμμα που παρουσιάζεται, όπου έχει ολοκληρωθεί η δημιουργία του δέντρου.

Το κλειδί για κάθε κόμβο υποτίθεται ότι είναι ένας θετικός ακέραιος, δηλαδή η διαδικασία γεμίσματος σταματάει με την πρώτη αρνητική τιμή που θα δοθεί.

Στο πρόγραμμα χρησιμοποιείται η **οδηγία typedef**. Αυτή επιτρέπει τη **δημιουργία ονόματος, το οποίο καθορίζει ο χρήστης, για κάποιο τύπο δεδομένων**. Η οδηγία typedef μοιάζει με τη #define, αλλά εκτελείται από το μεταγλωττιστή και όχι τον προεπεξεργαστή και δίνει συμβολικά ονόματα μόνο σε τύπους δεδομένων. Στο εξής δηλαδή οι δομές του τύπου ptr θα έχουν στο πρόγραμμα το όνομα TREE.



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

typedef struct ptr {
    int rec;
    struct ptr *lp;
    struct ptr *rp; } TREE;

TREE *treeLeaf (int);
TREE *newnode (int, TREE *, TREE *);

int main(void) {
    TREE *root = NULL;
    char akin[5];
    int  ak;
```

```
    gets (akin);
    ak=atoi (akin);
    root = newnode (ak, root, root);
    gets (akin);
    ak=atoi (akin);

    while (ak > 0) {
        newnode (ak, root, root);
        gets (akin);
        ak=atoi (akin); }                    /* Γραμμή 22 */

    return 1 ; }


TREE *treeLeaf (int num) {
    TREE *p;

    p = (TREE *) malloc (sizeof (TREE));
    p -> rec = num;
    p -> lp = NULL;
    p -> rp = NULL;
    return p; }


TREE *newnode (int num, TREE *pq, TREE *q) {
    if (q!=NULL) {
        if (num < q -> rec)
            q = newnode (num, q, q -> lp);
        else
            q = newnode (num, q, q -> rp); }
    else {
        q = treeLeaf (num);
        if (pq != NULL) {
            if (num < pq -> rec)
                pq -> lp = q;
            else
                pq -> rp = q; } }
    return q; }
```



**Σχόλια:**

- Η συνάρτηση newnode( ) είναι μια αναδρομική συνάρτηση, με την οποία επισκεπτόμαστε τους διάφορους κόμβους του δέντρου. Είναι η συνάρτηση με την οποία οδηγούμαστε προς το αριστερό ή το δεξιό υποδέντρο κάποιου κόμβου, ανάλογα με την τιμή την οποία πρόκειται να καταχωρήσουμε στο δέντρο. Εάν πρέπει να καταχωρηθεί μια νέα τιμή αλλά ο αντίστοιχος δείκτης είναι «γειωμένος», έχουμε φτάσει δηλαδή σε φύλλο του δέντρου, τότε καλείται η συνάρτηση treeLeaf( ), η οποία είναι η συνάρτηση που δεσμεύει μνήμη και

δημιουργεί ένα κόμβο. Με την πρώτη κλήση της newnode( ) και την πρώτη κλήση της treeLeaf( ) που γίνεται από αυτήν, δημιουργείται ο αρχικός κόμβος του δέντρου και τοποθετείται σε αυτόν ένας δείκτης, ο root. Η τιμή επιστροφής της πρέπει λοιπόν αναγκαστικά να είναι δείκτης σε TREE (κάτι που στις επόμενες κλήσεις δεν χρησιμοποιείται). **Ο δείκτης root πρέπει να μη μετακινηθεί ποτέ από τη ρίζα**, ώστε, όπως προαναφέρθηκε, να μπορούμε μέσω αυτού να έχουμε πρόσβαση σε όλους τους κόμβους του δέντρου

- Η treeLeaf( ) δεσμεύει χώρο στη μνήμη για τη δημιουργία του νέου κόμβου και καταχωρεί το κλειδί στο αντίστοιχο πεδίο του κόμβου. Επίσης «γειώνει» τους δείκτες προς τα παιδιά του νέου κόμβου και επιστρέφει ένα δείκτη στον νέο αυτό κόμβο που δημιουργήθηκε.



### 8.4.5. Συναρτήσεις για δυαδικά δέντρα.

#### 1) Διάσχιση με προδιατεταγμένο τρόπο (preorder):

Ορισμός:

```
void preOrder (TREE *q) {
    if (q!=NULL) {
        printf ("%d", q -> rec);
        preOrder (q -> lp);
        preOrder (q -> rp); } }
```

Δήλωση:

```
void preOrder (TREE *);
```

Κλήση:

```
preOrder (root);
```

#### 2) Διάσχιση με ενδοδιατεταγμένο τρόπο (inorder):

Ορισμός:

```
void inOrder (TREE *q) {
    if (q!=NULL) {
        inOrder (q -> lp);
        printf ("%d", q -> rec);
        inOrder (q -> rp); } }
```

Δήλωση:

```
void inOrder (TREE *);
```

Κλήση:

```
inOrder (root);
```

**3) Διάσχιση με μεταδιατεταγμένο τρόπο (postorder):**

Ορισμός:

```
void postOrder (TREE *q) {
    if (q!=NULL) {
        postOrder (q -> lp);
        postOrder (q -> rp);
        printf ("%d", q -> rec); } }
```

Δήλωση:

```
void postOrder (TREE *);
```

Κλήση:

```
postOrder (root);
```

**4) Αναζήτηση κόμβου σε δυαδικό δέντρο αναζήτησης:**

Αναζητούμε ένα κόμβο, στον οποίο υπάρχει ο ακέραιος num. Εάν βρεθεί τέτοιος κόμβος, η συνάρτηση τοποθετεί σε αυτόν ένα δείκτη, τον οποίο και επιστρέφει στη main( ). Εάν δεν υπάρχει τέτοιος κόμβος, η συνάρτηση επιστρέφει στη main( ) ένα δείκτη NULL. Η main( ) να γράφει στην οθόνη το μήνυμα “YPARXEI” ή “DEN YPARXEI”, εάν βρέθηκε ή όχι κόμβος όπως ο ζητούμενος.

Ορισμός:

```
TREE *search (TREE *root, int num) {
    TREE *ptr;

    ptr = root;
    while (ptr -> rec != snum) {
        if (ptr -> rec > num) {
            ptr = ptr -> lp;
            if (ptr == NULL)
                break; }
        if (ptr -> data < snum) {
            ptr = ptr->rp;
            if (ptr == NULL)
                break; } }
    return ptr; }
```

Δήλωση:

```
TREE *search (TREE *, int);
```

Στη main( ):

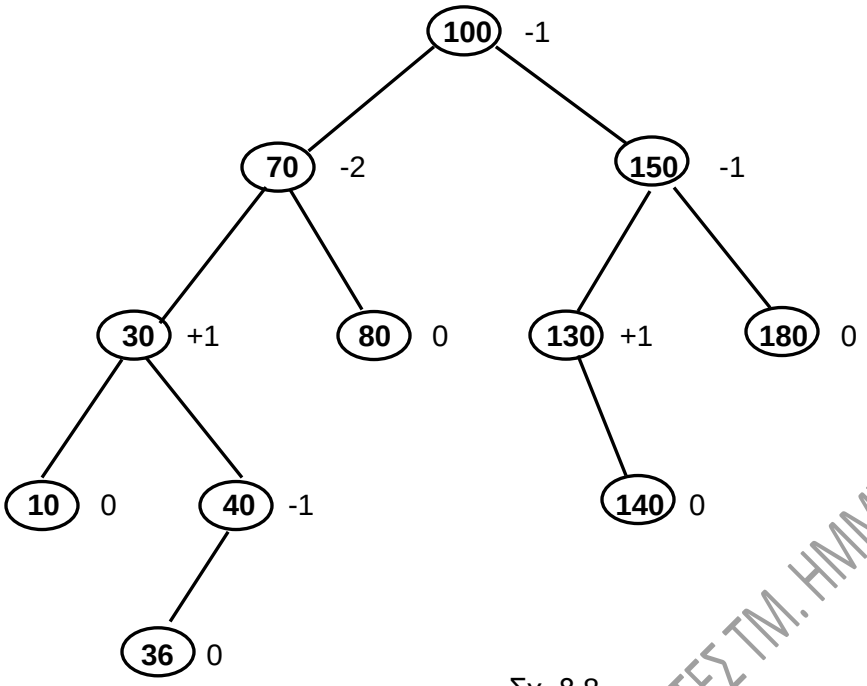
```
TREE *dkt ;
.....................................
scanf ("%d", &num);
dkt = search (root, num) ;
if (dkt != NULL)
        printf ("YPARXEI");
else
        printf ("DEN YPARXEI");
```

## 8.5. ΔΕΝΤΡΑ AVL.

### 8.5.1. Γενικά - ισοζύγιση.

Τα δέντρα αυτά αποτελούν μια **ειδική μορφή των δυαδικών δέντρων αναζήτησης**. Σε ένα AVL δέντρο **κάθε κόμβος είναι ισοζυγισμένος κατά το ύψος** (height balanced), δηλαδή **τα δυο υποδέντρα κάθε κόμβου διαφέρουν ως προς το ύψος το πολύ κατά 1**. Ο **παράγοντας ισοζύγισης** (balance factor, συντομογραφικά θα τον αναφέρουμε ως bf) ενός κόμβου είναι η διαφορά **του ύψους του δεξιού του υποδέντρου μείον το ύψος του αριστερού του υποδέντρου**. Αυτή η διαφορά είναι δηλαδή **ένας αριθμός, ο οποίος για AVL δέντρο έχει τιμή -1, 0 ή 1**.

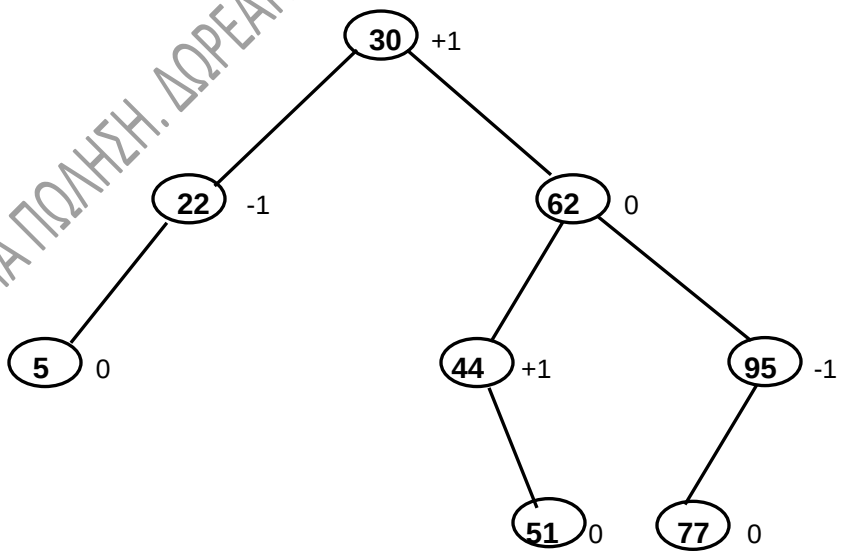
Είναι προφανές ότι για την περιγραφή κάθε κόμβου του δέντρου χρειάζεται ένα επιπλέον πεδίο, το οποίο θα «κρατάει» αυτόν τον αριθμό. Παράγοντας ισοζύγισης ίσος με -1 σε κάποιο κόμβο x σημαίνει ότι το υποδέντρο το οποίο έχει ως ρίζα τον x είναι «**αριστεροβαρές**», δηλαδή ότι «γέρνει» προς τα αριστερά, ενώ παράγοντας ισοζύγισης +1 σημαίνει ότι το υποδέντρο με ρίζα τον x είναι «**δεξιοβαρές**», δηλαδή ότι «γέρνει» προς τα δεξιά. Αυτό διαπιστώνεται και οπτικά από την εξέταση των υποδέντρων για κάθε κόμβο. Στα δέντρα των σχ. 8.8, 8.9 και 8.10 ο παράγοντας ισοζύγισης για κάθε κόμβο γράφεται δεξιά του κόμβου. Έτσι, το υποδέντρο για παράδειγμα με ρίζα 70 στο σχ. 8.8 φαίνεται και οπτικά ότι «γέρνει» προς τα αριστερά, πράγμα που είναι προφανώς συμβατό με το ότι ο παράγοντας ισοζύγισης του κόμβου αυτού είναι ίσος με -2. Λόγω του παράγοντα ισοζύγισης -2 στον κόμβο 70, το δέντρο του παραπάνω σχ. 8.8 **δεν είναι AVL δέντρο**, ενώ το επόμενο (σχ. 8.9) **είναι AVL**.



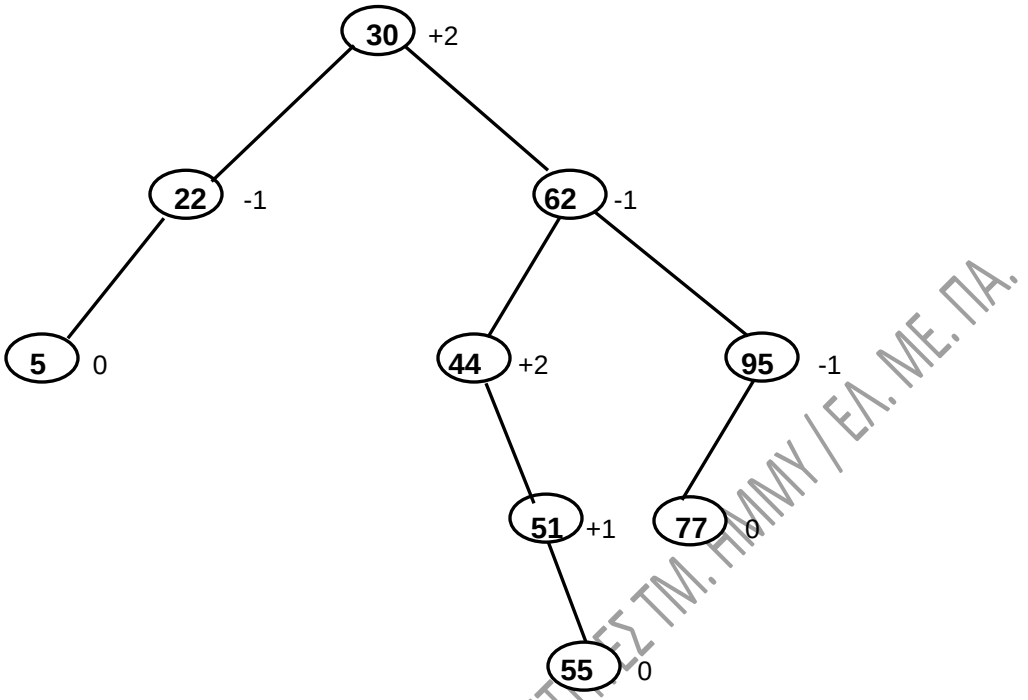

Σχ. 8.8

Η **εισαγωγή ενός νέου κόμβου σε ένα AVL δέντρο**, καθώς και η **διαγραφή κόμβου** από αυτό δεν είναι πάντα τόσο απλή όσο σε ένα κοινό δυαδικό δέντρο αναζήτησης. Για παράδειγμα, ένας νέος κόμβος με τιμή 55 στο δέντρο του σχ. 8.9 θα έπρεπε να τοποθετηθεί ως δεξί παιδί του κόμβου 51. Μια τέτοια όμως τοποθέτηση θα έκανε την τιμή του παράγοντα ισοζύγισης των κόμβων 44 και 30 ίση με +2, δηλαδή το δέντρο δεν θα ήταν πλέον AVL (σχ. 8.10).






Σχ. 8.9




Σχ. 8.10



Το θέμα αντιμετωπίζεται με κατάλληλη «**περιστροφή**» των κόμβων. Στη συνέχεια θα αναφερθούμε διεξοδικά στις περιστροφές που πρέπει να εκτελέσουμε, προκειμένου να αποκαταστήσουμε την ισορροπία του δέντρου

Είναι φανερό πάντως από τα παραπάνω **ότι καταβάλλουμε ιδιαίτερη προσπάθεια προκειμένου να χειριζόμαστε δέντρα, τα οποία να πλησιάζουν το πλήρες δέντρο όσο το δυνατόν περισσότερο**. Ο λόγος είναι ότι η αναζήτηση σε ένα πλήρες δέντρο είναι πολύ πιο εύκολη και γρήγορη από ότι σε ένα μη πλήρες δέντρο. Ο πίνακας του σχ. 8.11 που ακολουθεί δείχνει τον μέσο αριθμό συγκρίσεων για την επιτυχή αναζήτηση ενός κόμβου στην περίπτωση ενός δυαδικού δέντρου αναζήτησης τυχαίας μορφής και ενός πλήρους δυαδικού δέντρου αναζήτησης. Τα δέντρα υποτίθεται ότι έχουν n κόμβους.

| n | Πλήρες δένδρο | Τυχαίο δένδρο |
|---|---|---|
| 1000 | 9,0 | 12,0 |
| 10000 | 12,3 | 16,6 |
| 100000 | 15,6 | 21,2 |
| 1000000 | 18,9 | 25,8 |

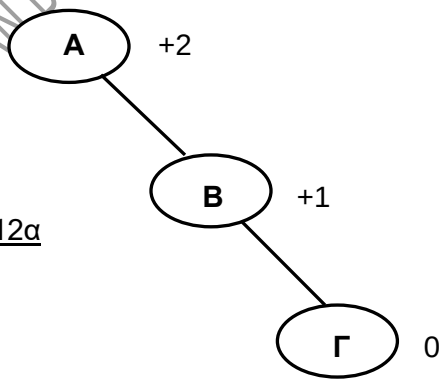
Σχ. 8.11

### 8.5.2. Περιστροφές δέντρου.

Με τις περιστροφές αποκαθιστούμε, όπως προαναφέραμε, την ισορροπία σε ένα δέντρο. Δηλαδή, **ξανακάνουμε AVL ένα δέντρο, το οποίο έπαψε να είναι τέτοιο** λόγω εισαγωγής ή διαγραφής κάποιου κόμβου.

Έχουμε αρχικά στο μυαλό μας ως γενική αρχή ότι **σε ένα δέντρο το οποίο είναι αριστεροβαρές θα πρέπει να γίνει δεξιά περιστροφή**, ενώ **σε ένα δεξιοβαρές δέντρο πρέπει να γίνει αριστερή περιστροφή. Αυτό πάντως, όπως θα δούμε στη συνέχεια, δεν είναι πάντοτε αρκετό.** Λέγοντας «δεξιά» ή «αριστερή» περιστροφή, εννοούμε την πλευρά του δέντρου πρός την οποία περιστρέφονται οι κόμβοι. Πρακτικά:

1) **Αριστερή περιστροφή (Left Rotation, LR).** Έστω ότι μετά την εισαγωγή ενός κόμβου παρουσιάζεται σε κάποιο κλώνο (υποδέντρο) του δέντρου η εξής κατάσταση:




Σχ. 8.12α

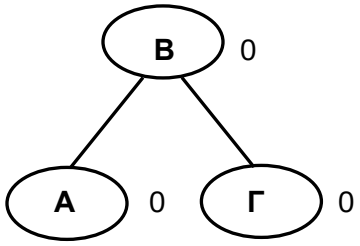
Προχωρούμε σε αριστερή περιστροφή στον κόμβο Α, ακολουθώντας τα εξής βήματα:

i) Το Β γίνεται η νέα ρίζα του υποδέντρου.

ii) Το Α παίρνει ως δεξί παιδί του το αριστερό παιδί του Β (εδώ δεν υπάρχει)

iii) Το Α γίνεται αριστερό παιδί του Β.

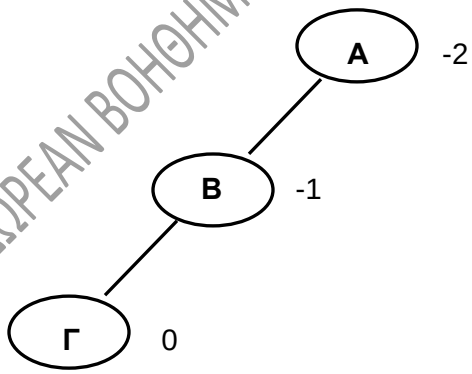
οπότε καταλήγουμε στο παρακάτω:




Σχ. 8.12β

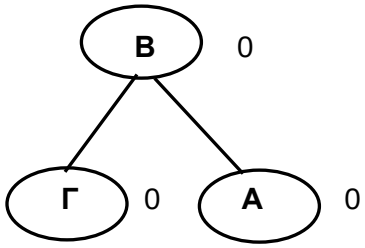
2) **Δεξιά περιστροφή (Right Rotation, RR).** Είναι η συμμετρική της προηγούμενης. Έστω ότι μετά την εισαγωγή ενός κόμβου παρουσιάζεται σε κάποιο κλώνο (υποδέντρο) του δέντρου η κατάσταση του σχ. 8.13α. Προχωρούμε σε δεξιά περιστροφή στον κόμβο Α, ακολουθώντας τα εξής βήματα:

   i) Το Β γίνεται η νέα ρίζα του υποδέντρου.
   ii) Το Α παίρνει ως αριστερό παιδί του το δεξί παιδί του Β (εδώ δεν υπάρχει)
   iii) Το Α γίνεται δεξί παιδί του Β.



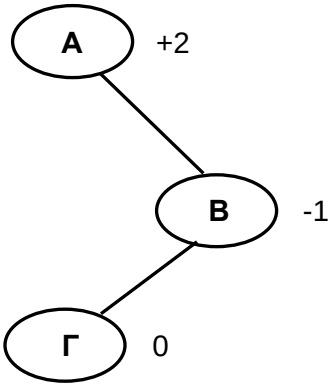

Σχ. 8.13α

οπότε καταλήγουμε στο σχ. 8.13β:



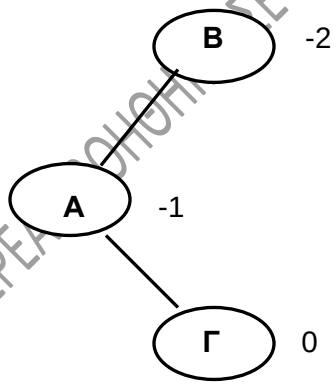

Σχ. 8.13β

3) **Αριστερή - Δεξιά περιστροφή (Left - Right Rotation, LRR).** Εφαρμόζεται όταν το δεξιό υποδέντρο είναι βαρύ αριστερά,όπως στο σχ. 8.14α:



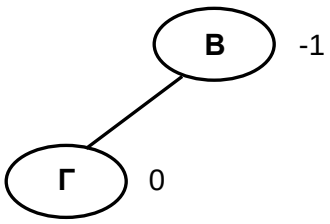

Σχ. 8.14α

Σύμφωνα με όσα αναφέρθηκαν πιο πάνω, για την LR, με απλή αριστερή περιστροφή στον κόμβο Α θα καταλήξουμε στο σχ. 8.14β, το οποίο δεν λύνει το πρόβλημα, αφού καταλήγουμε σε συμμετρική μορφή αυτής, την οποία είχαμε αρχικά.



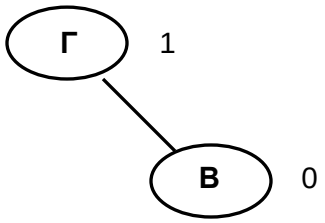

Σχ. 8.14β

**Η λύση είναι** να προχωρήσουμε σε **δεξιά περιστροφή στο δεξιό υποδέντρο, δηλαδή στον κόμβο Β** (μιλάμε για το σχ. 8.14α). Αν απομονώσουμε το δεξιό υποδέντρο από το σχήμα 8.14α, τότε έχουμε το σχ. 8.14γ:



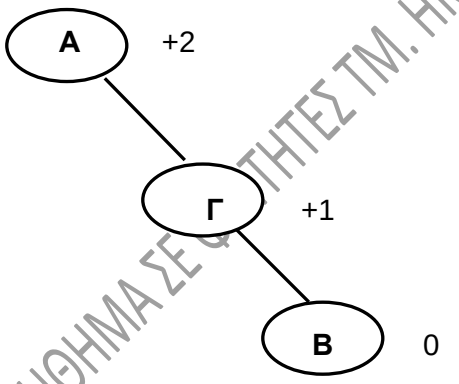

Σχ. 8.14γ

Σύμφωνα με όσα αναφέρθηκαν πιο πάνω για την RR, καταλήγουμε στο σχ. 8.14δ:



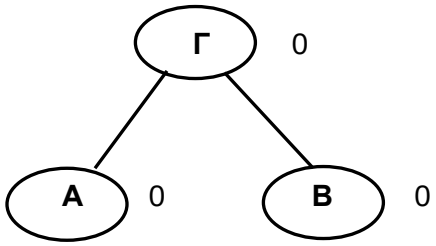

Σχ. 8.14δ

Τελικά λοιπόν το υποδέντρο μας είναι αυτό του σχ. 8.14ε:



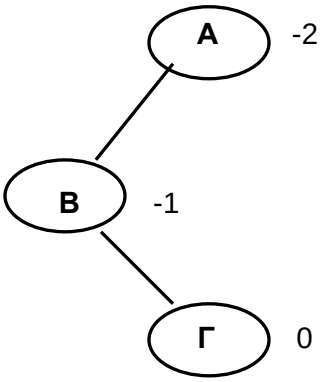

Σχ. 8.14ε

οπότε με αριστερή περιστροφή των κόμβων γύρω στον κόμβο Α, καταλήγουμε στο σχ. 8.14στ, δηλαδή σε ισοζυγισμένη μορφή:



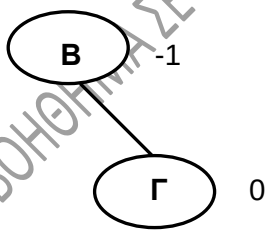

Σχ. 8.14στ

4) **Δεξιά - Αριστερή περιστροφή (Right - Left Rotation, RLR).** Εφαρμόζεται όταν το αριστερό υποδέντρο είναι βαρύ δεξιά. Δείτε το παράδειγμα του σχ. 8.15α:



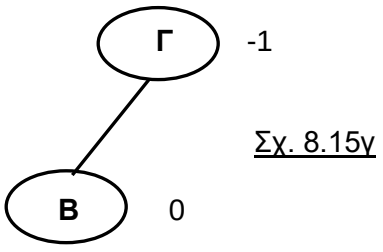

Σχ. 8.15α

Αντίστοιχα με την περίπτωση της παραπάνω παραγράφου (3), μια δεξιά περιστροφή δεν λύνει το πρόβλημα. Προχωρούμε σε αριστερή περιστροφή (LR) στον κόμβο Β για το υποδέντρο του σχ. 8.15β:

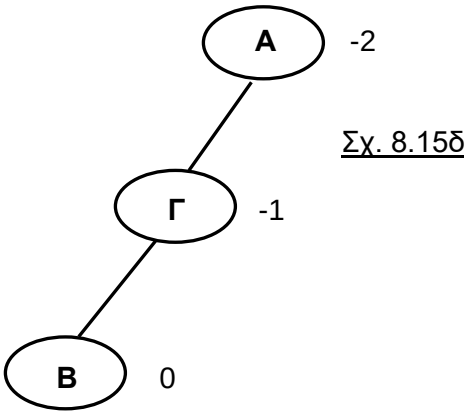



Σχ. 8.15β

και παίρνουμε το υποδέντρο του σχ. 8.15γ, το δε αρχικό υποδέντρο του σχ. 8.16α γίνεται πλέον όπως στο σχ. 8.15δ:

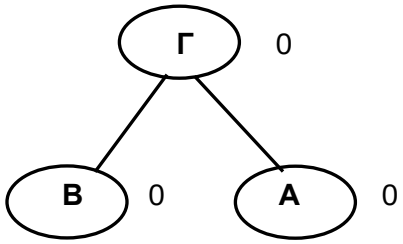



Σχ. 8.15γ



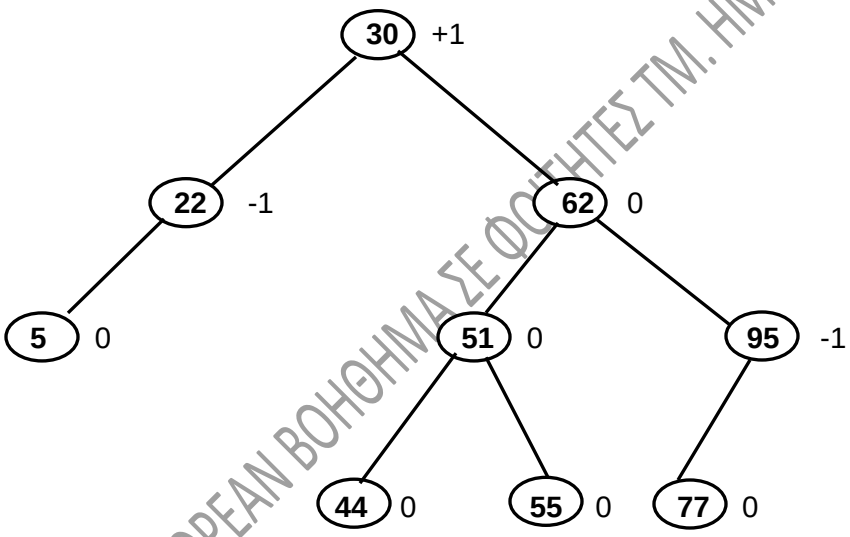

Σχ. 8.15δ

Το υποδέντρο, με απλή δεξιά περιστροφή στον κόμβο Α καταλήγει στην ισοζυγισμένη μορφή του σχ. 8.15ε:




Σχ. 8.15ε

Με βάση τα παραπάνω, δείτε στο σχ. 8.16 πώς επανέρχεται η ισορροπία στο δέντρο του σχ. 8.10, μετά από περιστροφή:




Σχ. 8.16



## 8.6. ΑΡΘΡΩΤΑ ΔΕΝΤΡΑ.

### 8.6.1. Γενικά.

Ένα **αρθρωτό δέντρο** (**splay tree**) είναι ένα δυαδικό δέντρο αναζήτησης. Η καινούργια λειτουργία, την οποία έχουμε σε αυτού του είδους τα δέντρα λέγεται **splay**. Η λειτουργία αυτή έχει ως σκοπό την **μετακίνηση ενός κόμβου προς την ρίζα** του δέντρου. Βασικό στοιχείο της μετακίνησης αυτής είναι οι περιστροφές, τις οποίες έχουμε δει και στα

προηγούμενα στα δέντρα AVL. Πριν προχωρήσουμε, να επισημάνουμε το εξής: **τα αρθρωτά δέντρα δεν είναι εντελώς ισοζυγισμένα** όπως είναι τα AVL, γι' αυτό τον λόγο άλλωστε υλοποιούνται ευκολότερα. Ο σκοπός μας στα δέντρα αυτά δεν είναι τόσο η ισοζύγιση, όσο η μετακίνηση στη ρίζα του δέντρου ενός στοιχείου. Αυτό το στοιχείο μπορεί να είναι αυτό που έχει μόλις εισαχθεί στο δέντρο ή αυτό το οποίο μόλις αναζητήσαμε στο δέντρο. Η βασική μας λοιπόν επιδίωξη είναι η μετακίνηση του στοιχείου στην ρίζα, άρα η γρήγορη προσπέλαση σε στοιχεία που έχουν προσπελαστεί πρόσφατα και δευτερευόντως η διατήρηση της ισοζύγισης του δέντρου.

Τα δέντρα αυτά είναι μια από τις κύριες δομές που χρησιμοποιούνται σε βιομηχανικές εφαρμογές. Στους υπολογιστές η λειτουργία της cache βασίζεται ακριβώς σε τέτοιου είδους δέντρα.



### 8.6.2. Λειτουργίες zig-zig, zig-zag και zig.

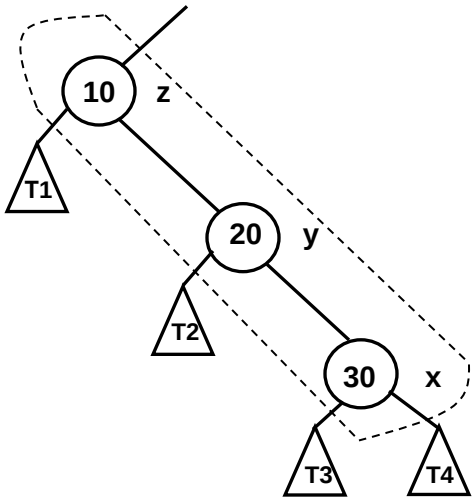
Οι περιπτώσεις τις οποίες συναντούμε **για την μετακίνηση ενός κόμβου x προς την ρίζα** (**splaying** στα αγγλικά) καλύπτονται από τις παρακάτω τρεις περιπτώσεις, κάθε μια από τις οποίες χαρακτηρίζεται τις ενέργειες τις οποίες θα κάνουμε.

#### 1η περίπτωση. Λειτουργία zig-zig:

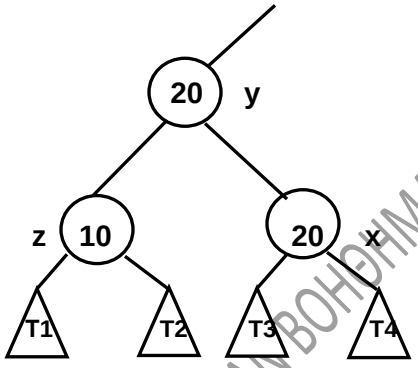
Εδώ ο κόμβος x είναι δεξί παιδί του y και ο κόμβος y δεξί παιδί του z (συμμετρικά μπορεί να έχουμε αριστερό και αριστερό παιδί). Η περίπτωση αυτή απεικονίζεται στο σχ. 8.17α. Τα τρίγωνα T1, T2, T3 και T4 συμβολίζουν τα υποδέντρα τα οποία έχουν ρίζες τους κόμβους z, y και x.

Για την μορφή αυτή, αυτή δηλαδή που φαίνεται μέσα στην διακεκομμένη γραμμή, προκειμένου να μετακινήσουμε το x προς την κορυφή, εκτελούμε **αριστερή περιστροφή του y γύρω από τον z** και μετά **αριστερή περιστροφή του x γύρω από τον y**. Μετά την πρώτη από αυτές τις περιστροφές παίρνουμε το σχ. 8.17β, ενώ μετά και από την δεύτερη περιστροφή καταλήγουμε στο σχ. 8.17γ. Οι περιστροφές γίνονται σύμφωνα με όσα έχουμε πει και για τα AVL δέντρα. Στα σχήματα φαίνονται πώς μετακινούνται σε κάθε περίπτωση και τα υποδέντρα T1, T2, T3 και T4 και πού προσκολλώνται κατά τη διαδικασία των περιστροφών.

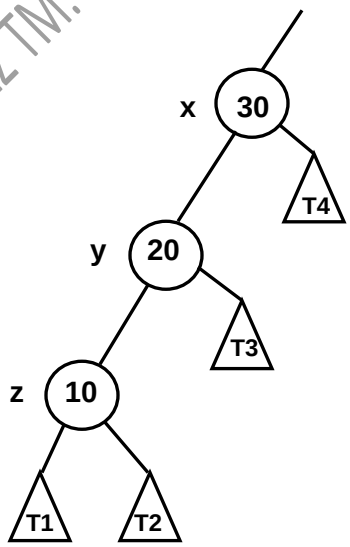
Η λειτουργία αυτού του τύπου έχει το όνομα **zig-zig**. Στην συμμετρική περίπτωση, όταν δηλαδή έχουμε αριστερό και αριστερό παιδί, εκτελούμε δυο δεξιές περιστροφές.




Σχ. 8.17α




Σχ. 8.17β



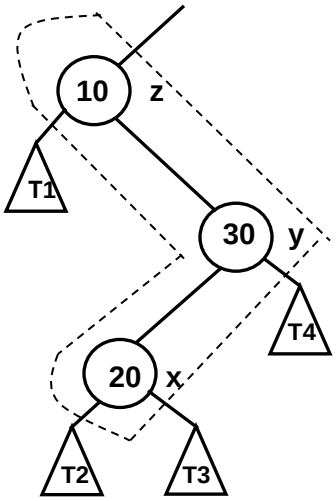

Σχ. 8.17γ



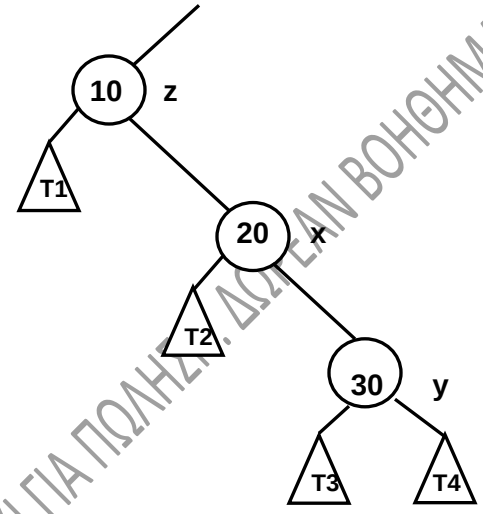
**2η περίπτωση. Λειτουργία zig-zag:**

Εδώ ένα από τα x και y είναι δεξί παιδί και το άλλο είναι αριστερό (Σχ. 8.18α). Για την μορφή αυτή, όπως φαίνεται μέσα στην διακεκομμένη γραμμή, κάνουμε τις αντικαταστάσεις όπως τις βλέπετε στο σχ. 8.18β και 8.18γ, οι οποίες και πάλι προκύπτουν από περιστροφές. Στο σχ. 8.18β έχουμε και πάλι το κομμάτι αυτό του δέντρου όπως έχει γίνει μετά από **δεξιά περιστροφή του x γύρω από τον y** και στο σχ. 8.18γ μετά από **αριστερή περιστροφή του x γύρω από τον z**. Φυσικά με αντίθετη διάταξη των x, y και z έχουμε αντίθετες περιστροφές.

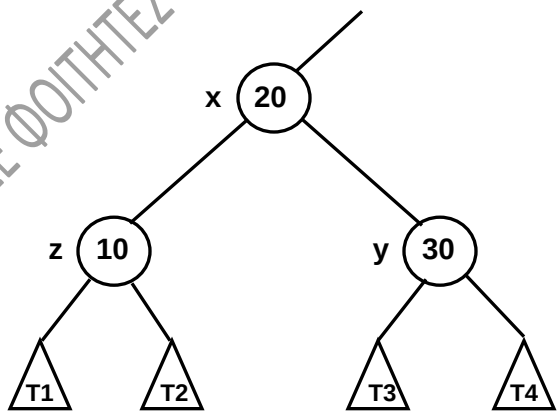
Η λειτουργία αυτού του τύπου έχει το όνομα **zig-zag**.




Σχ. 8.18α




Σχ. 8.18β




Σχ. 8.18γ

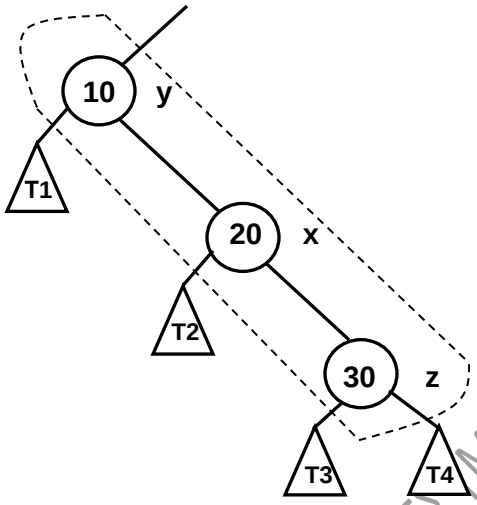


**3η περίπτωση. Λειτουργία zig:**

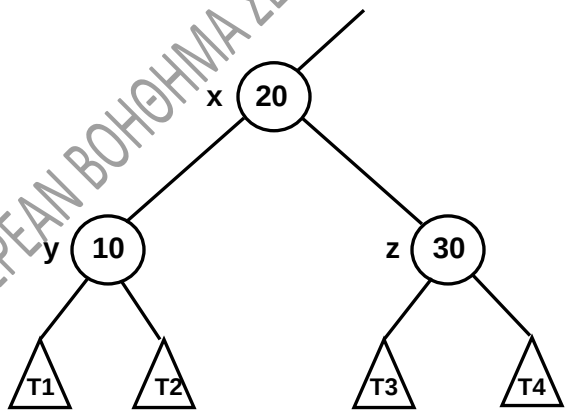
Εδώ ο x δεν έχει παππού ή αυτός ο παππούς για κάποιο λόγο δεν λαμβάνεται υπ' όψη (σχ. 8.19α), για παράδειγμα εάν ο x είναι παιδί της ρίζας. Για την μορφή αυτή, όπως φαίνεται μέσα στην διακεκομμένη γραμμή, κάνουμε τις αντικαταστάσεις όπως τις βλέπετε στο σχ. 8.19β και οι οποίες και πάλι προέκυψαν από **μια αριστερή**

**περιστροφή του x γύρω από τον y**. Και εδώ, με αντίθετη διάταξη των x, y και z έχουμε αντίθετη περιστροφή.

Η λειτουργία αυτού του τύπου έχει το όνομα **zig**.




Σχ. 8.19α



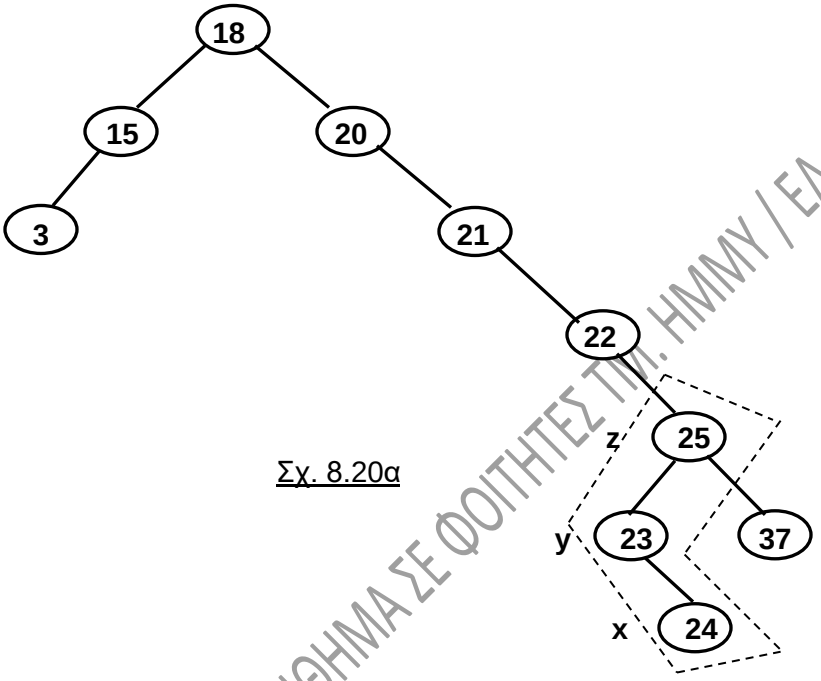

Σχ. 8.19β



Μπορούμε να παρατηρήσουμε, ότι **μετά από μια zig-zig ή μια zig-zag λειτουργία, το βάθος του x μειώνεται κατά δύο**, ενώ **μετά από μια zig λειτουργία το βάθος του x μειώνεται κατά ένα**. Από αυτό μπορούμε να συμπεράνουμε ότι **για την λειτουργία splaying ενός κόμβου x που έχει βάθος d, χρειαζόμαστε d / 2** (ακέραια διαίρεση) **λειτουργίες zig-zig ή zig-zag**. **Αν το d είναι περιττό, χρειαζόμαστε ένα επιπλέον zig**.

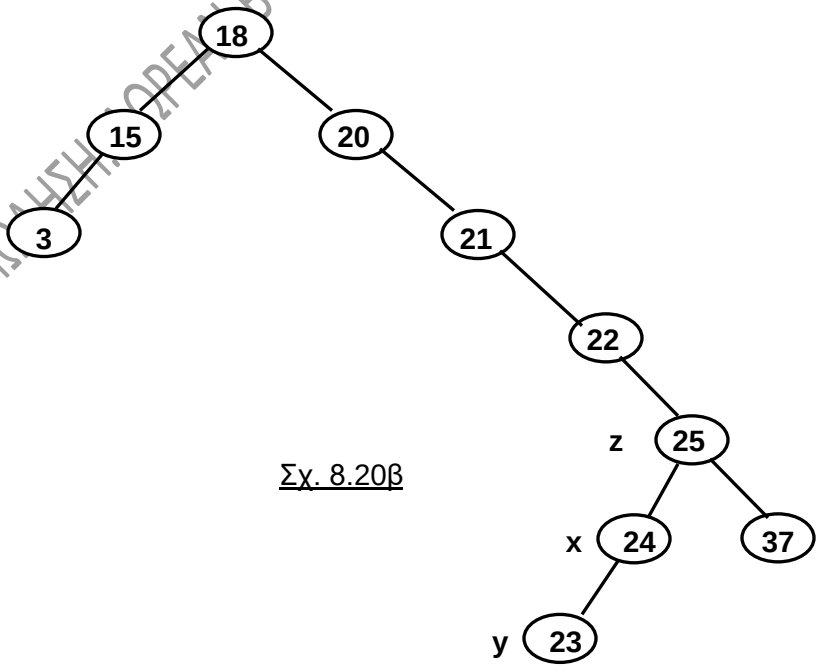
**Παράδειγμα:**

Στο δέντρο του σχ. 8.20α θέλουμε να εφαρμόσουμε λειτουργία splaying στον κόμβο με τιμή 24, δηλαδή θέλουμε να ανεβάσουμε το 24 στην ρίζα του δέντρου. Σύμφωνα με όσα προαναφέραμε, θα εφαρμόσουμε λειτουργία zig-zag στο τμήμα του δέντρου που βρίσκεται μέσα στο πλαίσιο με τις διακεκομμένες γραμμές.



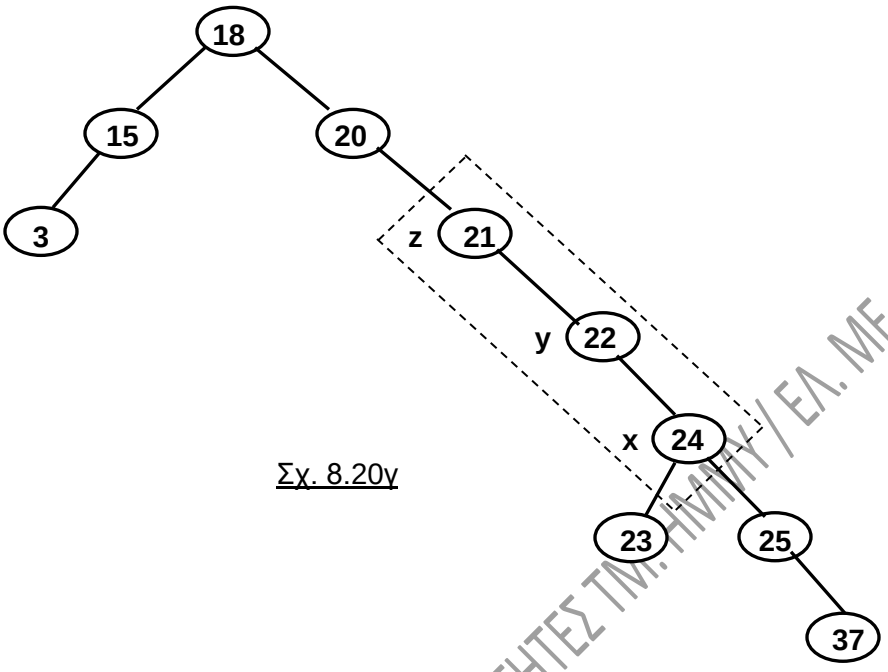

Σχ. 8.20α

Μετά από αριστερή περιστροφή του x γύρω από τον y έχουμε το δέντρο του σχ. 8.20β:



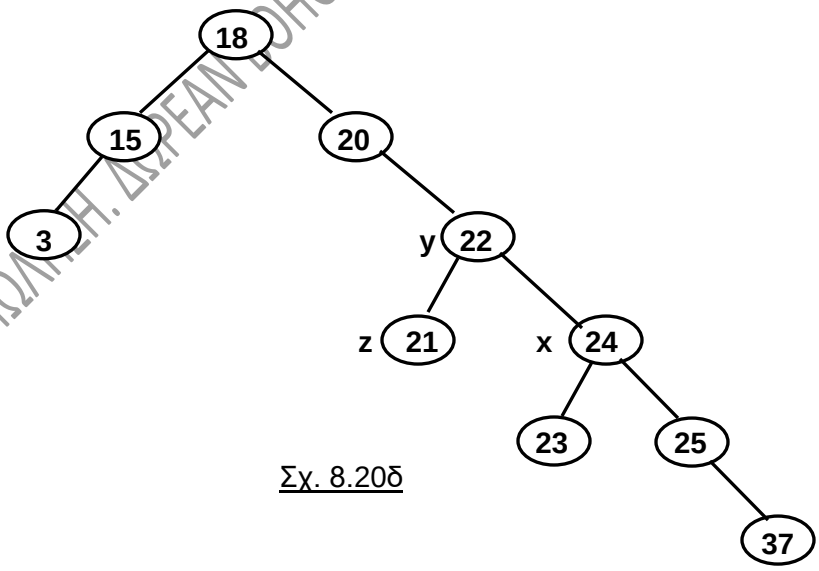

Σχ. 8.20β

Ακολουθεί δεξιά περιστροφή του x γύρω από τον z, και οδηγούμαστε στο σχ. 8.20γ:



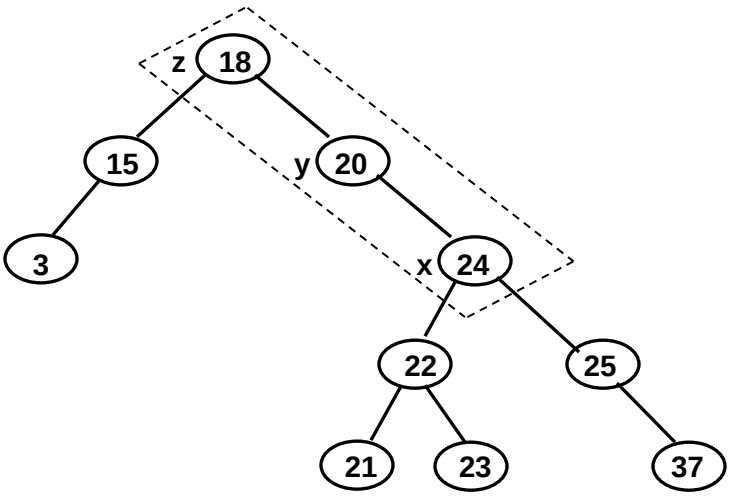

Σχ. 8.20γ

Εδώ τώρα εφαρμόζουμε λειτουργία zig-zig στο τμήμα του δέντρου που βρίσκεται μέσα στο πλαίσιο με τις διακεκομμένες γραμμές. Το x είναι το 24, δείτε όμως τώρα ότι το y είναι ο κόμβος του 22 και το z είναι ο κόμβος του 21. Εφαρμόζουμε αριστερή περιστροφή του y γύρω από τον z (σχ. 8.20δ):



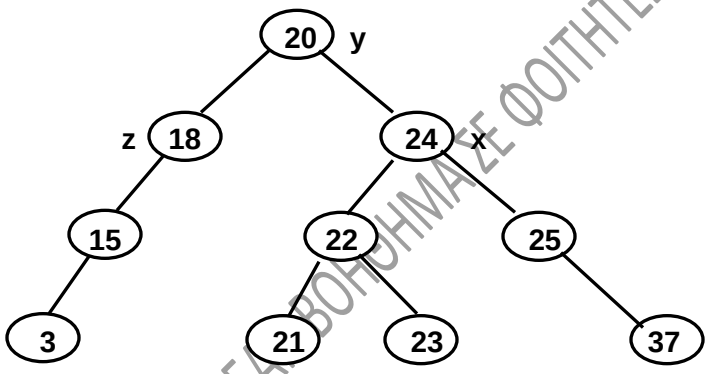

Σχ. 8.20δ

Στη συνέχεια εφαρμόζουμε αριστερή περιστροφή του x γύρω από τον y (σχ. 8.20ε):



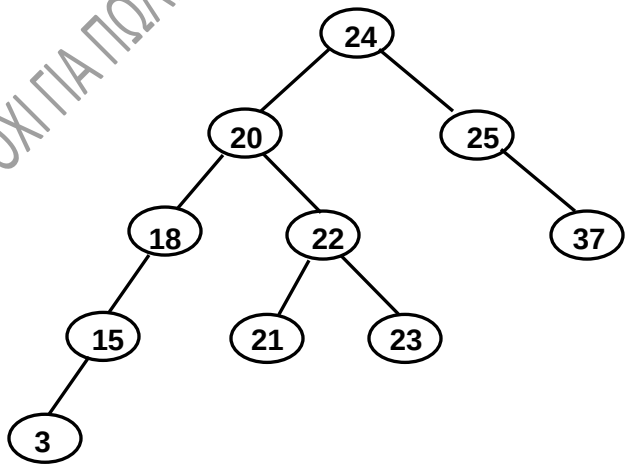

Σχ. 8.20ε

Τώρα το x είναι το 24, το y είναι το 20 και το z είναι το 18. Εφαρμόζουμε λειτουργία zig-zig μέσα στο πλαίσιο με τις διακεκομμένες γραμμές. Άρα αρχικά κάνουμε αριστερή περιστροφή του y (το 20) γύρω από το z (το 18), οπότε έχουμε το σχ. 8.20στ:




Σχ. 8.20στ

Τέλος, μετά από αριστερή περιστροφή του x γύρω από τον y καταλήγουμε στο σχ. 8.20ζ.




Σχ. 8.20ζ

## 8.7. ΚΟΚΚΙΝΟΜΑΥΡΑ ΔΕΝΤΡΑ (Red-black trees).

Τα «κοκκινόμαυρα δέντρα» (για συντομία στο εξής : ΚΜΔ) είναι ένα **είδος δυαδικών δέντρων αναζήτησης**. **Κάθε κόμβος του δέντρου χαρακτηρίζεται από ένα «χρώμα»**, με τέτοιο τρόπο, ώστε να ικανοποιούνται κάποιες συνθήκες και περιορισμοί. Κατά την εισαγωγή ή διαγραφή κόμβου, οι κόμβοι του δέντρου «ξαναχρωματίζονται», ώστε αυτές οι συνθήκες και οι περιορισμοί να εξακολουθούν να ισχύουν. Προφανώς λοιπόν, σε κάθε κόμβο κρατείται μια επιπλέον πληροφορία, από όσες ξέρουμε για τα απλά δέντρα και αυτή αφορά το χρώμα του κόμβου.

Τα ΚΜΔ χρησιμοποιούνται, όπως και τα AVL για να μειώσουν κατά το δυνατόν το ύψος ενός δέντρου αναζήτησης και να το φέρουν όσο πιο κοντά γίνεται στο πλήρες. **Τα AVL είναι περισσότερο ισορροπημένα σε σχέση με τα ΚΜΔ**, αλλά είναι πιθανό να χρειάζονται πολλές περιστροφές κατά την εισαγωγή ή τη διαγραφή κόμβου. Έτσι, **σε εφαρμογές με πολλές εισαγωγές και διαγραφές κόμβων, προτιμώνται τα ΚΜΔ**. Αντίθετα, αν αυτές οι ενέργειες δεν είναι συχνές, αλλά είναι **συχνή η αναζήτηση στοιχείων σε ένα δέντρο, προτιμώνται τα AVL**.

### 8.7.1. Ιδιότητες κοκκινόμαυρων δέντρων.

Εκτός από όσα γνωρίζουμε μέχρι τώρα για τα δυαδικά δέντρα αναζήτησης, στα ΚΜΔ ισχύουν επιπλέον τα εξής:

1. **Ένας κόμβος είναι είτε κόκκινος είτε μαύρος**.
2. **Η ρίζα είναι μαύρη**.
3. **Όλοι οι τερματικοί δείκτες NULL θεωρούνται ως φύλλα με μαύρο χρώμα** (ίδιο χρώμα με την ρίζα). Πρακτικά δηλαδή, οι δείκτες αυτοί θεωρούνται δείκτες σε εξωτερικούς κόμβους-φύλλα του δέντρου.
4. **Κάθε κόκκινος κόμβος πρέπει να έχει δύο μαύρα παιδιά**.
5. **Κάθε διαδρομή από ένα κόμβο προς οποιοδήποτε από τους NULL κόμβους απογόνους του περιέχει τον ίδιο αριθμό μαύρων κόμβων**.

Αποδεικνύεται ότι **τουλάχιστον οι μισοί κόμβοι σε κάθε μονοπάτι από τη ρίζα μέχρι κάποιο φύλλο είναι μαύροι**.

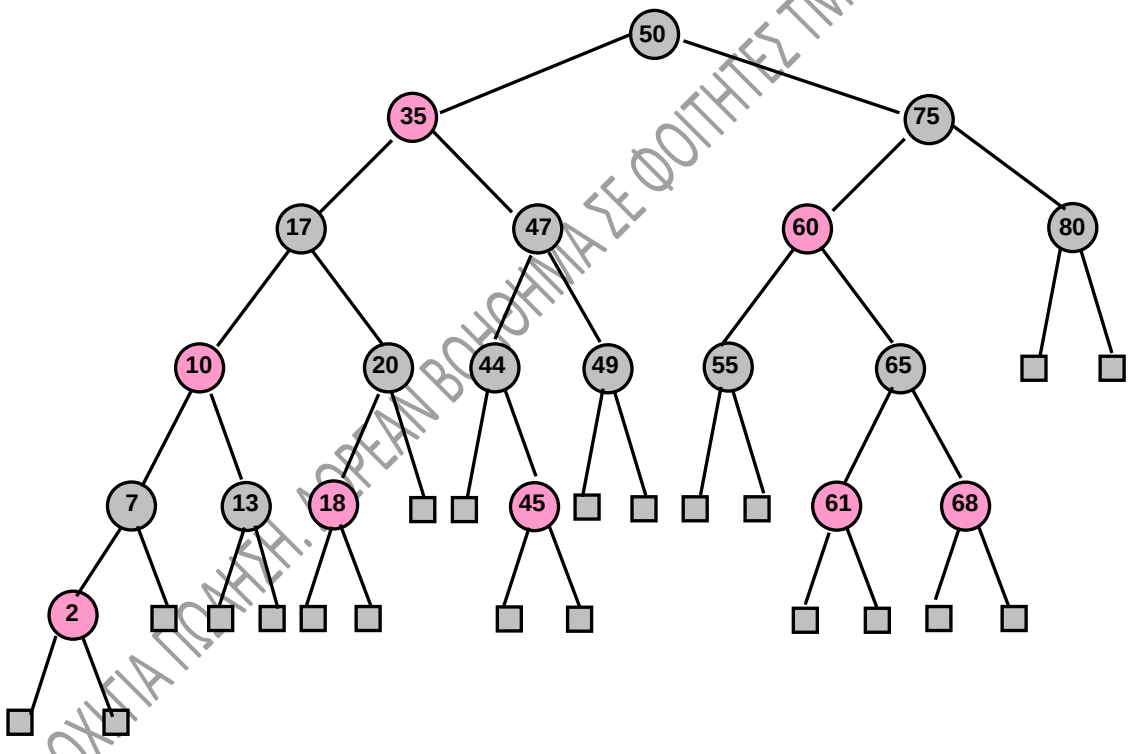
Στο σχ. 8.21 παρουσιάζεται ένα ΚΜΔ.

Εξετάζουμε στη συνέχεια πώς γίνεται η εισαγωγή κόμβου σε ΚΜΔ και πώς γίνεται η διαγραφή κόμβου από αυτό.

### 8.7.2. Εισαγωγή σε κοκκινόμαυρο δέντρο.

1. **Εισάγουμε τον νέο κόμβο** με τον τρόπο που γνωρίζουμε για τα δυαδικά δέντρα αναζήτησης.
2. **Ο νέος κόμβος** παίρνει **κόκκινο χρώμα**.
3. **Αν εξακολουθούν να ισχύουν οι ιδιότητες του ΚΜΔ** ως προς τα χρώματα των κόμβων, **η εισαγωγή έχει ολοκληρωθεί**. Σε διαφορετική περίπτωση, πρέπει να κάνουμε κάποιες ενέργειες για να αποκαταστήσουμε τις ιδιότητες του ΚΜΔ. Τις ενέργειες αυτές κωδικοποιούμε παρακάτω.

Για την ευκολότερη κατανόηση των ενεργειών που θα κάνουμε, συμβολίζουμε με x τον εισαγόμενο κόμβο αρχικά καθώς επίσης τον κόμβο, από τον οποίο προκύπτει η ανάγκη αποκατάστασης του δέντρου στη συνέχεια, με px συμβολίζουμε τον γονέα του x, με ux τον αδελφό του γονέα του x (άρα τον θείο του x) και με gpx τον παππού του x.






Σχ. 8.21

**Σύνοψη ενεργειών αποκατάστασης:**

Θεωρούμε ότι ο γονέας του x (ο px) είναι αριστερό παιδί του παππού του x (του gpx). (Αν είναι δεξί παιδί, η περίπτωση είναι απλώς συμμετρική.). **Εάν ο x και ο px είναι κόκκινοι**, πρέπει να προβούμε σε ενέργειες αποκατάστασης ως εξής:

α) **Αν ο ux είναι κόκκινος**: αλλάζουμε το χρώμα του px και του ux σε μαύρο και το χρώμα του gpx σε κόκκινο. Θεωρούμε κατόπιν ότι x γίνεται ο gpx και επαναλαμβάνουμε το ίδιο, εάν απαιτείται (Εάν δηλαδή ο καινούργιος x και ο καινούργιος px είναι κόκκινοι).

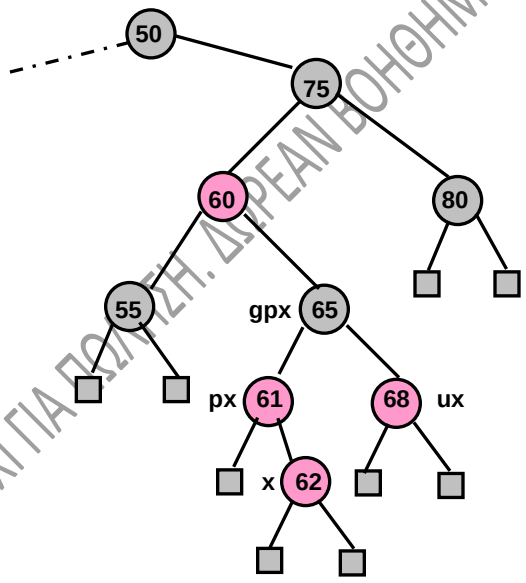
β) **Αν ο ux είναι μαύρος και:**

i) **Ο x είναι δεξί παιδί του px**: εκτελούμε αριστερή περιστροφή στον px. Με αυτήν, ο x γίνεται αριστερό παιδί του px, οπότε προχωρούμε στην επόμενη περίπτωση (όσον αφορά τις περιστροφές, συμβουλευτείτε όσα αναφέρθηκαν στα AVL δέντρα).

ii) **Ο x είναι αριστερό παιδί του px:** αλλάζουμε το χρώμα του px σε μαύρο και το χρώμα του gpx σε κόκκινο. Στη συνέχεια εκτελούμε δεξιά περιστροφή στον gpx.

**Παράδειγμα:**

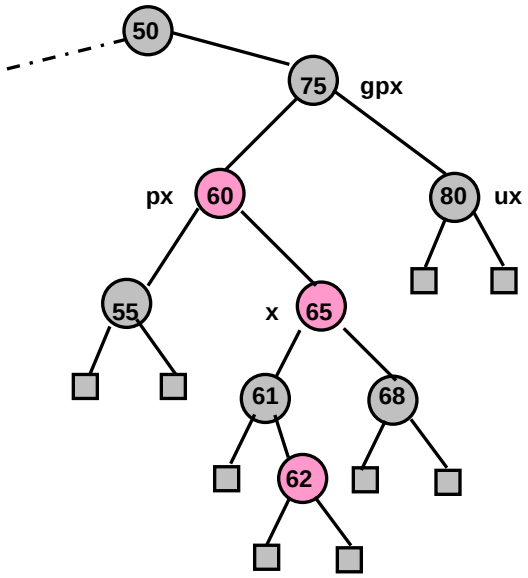
Βήμα 1: Στο ΚΜΔ του σχ. 8.21 θα κάνουμε εισαγωγή του κόμβου 62. Ο κόμβος θα είναι κόκκινος και θα είναι δεξί παιδί του 61. Οδηγούμαστε δηλαδή στο σχ. 8.22α (το αριστερό υποδέντρο του 50 προφανώς δεν αλλάζει και έτσι δεν το επανασχεδιάζουμε):




Σχ. 8.22α

Στο δέντρο αυτό, υπάρχει η ακόλουθη αντιστοιχία κόμβων με τα σύμβολα που αναφέραμε προηγουμένως: x 62, px 61, ux 68, gpx 65.

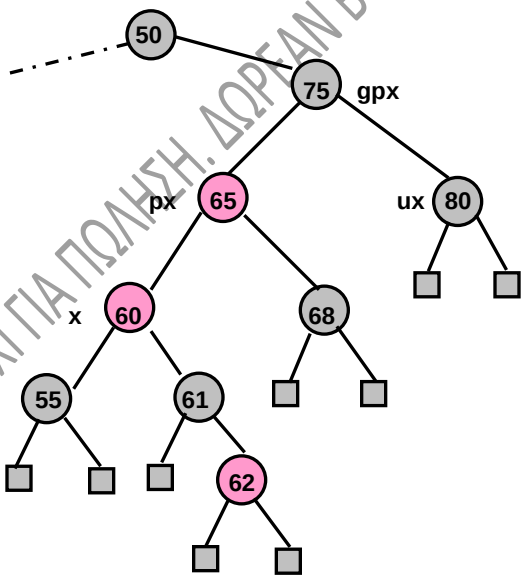
Βήμα 2: Οι x και px είναι κόκκινοι, άρα πρέπει να αποκαταστήσουμε το δέντρο. Ο ux είναι κόκκινος, άρα αλλάζουμε το χρώμα του px και του ux σε μαύρο και το χρώμα του gpx σε κόκκινο. Το δέντρο που προκύπτει είναι αυτό του σχ. 8.22β:




Σχ. 8.22β

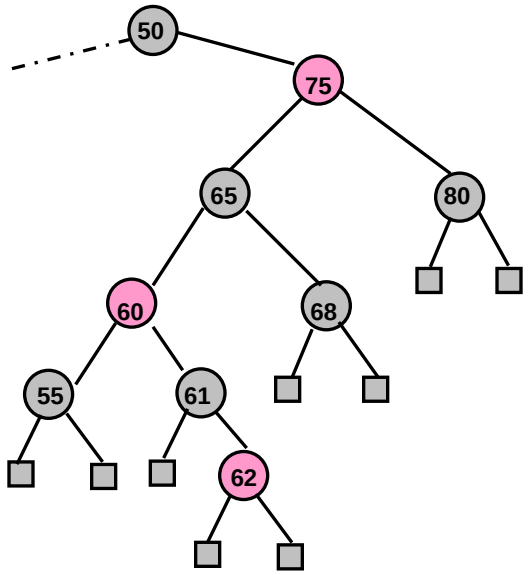
Στο δέντρο αυτό, ο 60, ως κόκκινος, θα έπρεπε να έχει μαύρα παιδιά, άρα το δέντρο χρειάζεται αποκατάσταση. Τώρα η αντιστοιχία είναι: x 65, px 60, ux 80, gpx 75.

Βήμα 3: Παρατηρούμε ότι ο ux είναι μαύρος και ο x είναι δεξί παιδί του px. Άρα εκτελούμε αριστερή περιστροφή στον 60:



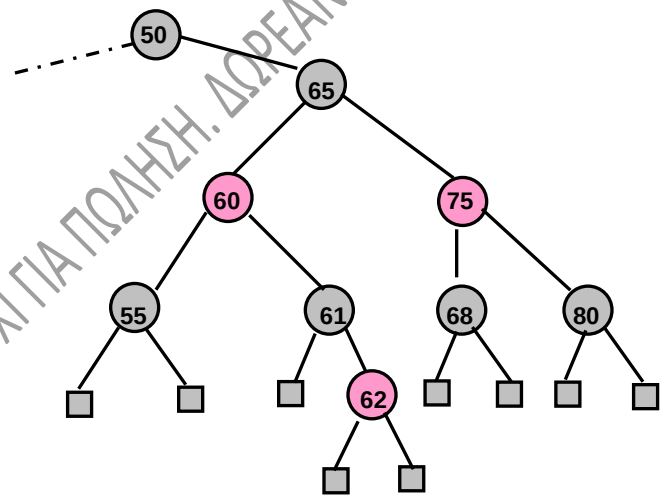

Σχ. 8.22γ

Η νέα αντιστοιχία: x 60, px 65, ux 68, gpx 75, με τον 60 να είναι αριστερό παιδί του 65, άρα αλλάζουμε το χρώμα του px σε μαύρο και το χρώμα του gpx σε κόκκινο. Καταλήγουμε στο σχ. 8.22δ:




Σχ. 8.22δ

Στη συνέχεια εκτελούμε δεξιά περιστροφή στον 75, οπότε η τελική αποκατεστημένη μορφή του ΚΜΔ είναι αυτή του σχ. 8.22ε:




Σχ. 8.22ε

### 8.7.3. Διαγραφή από κοκκινόμαυρο δέντρο:

Η **διαγραφή** ενός κόμβου x είναι αρκετά πιο περίπλοκη από την εισαγωγή ενός κόμβου. Γίνεται κατ' αρχήν σύμφωνα με **όσα γνωρίζουμε για τα δυαδικά δέντρα αναζήτησης**. Παρατηρούμε ότι όποτε γίνεται μια διαγραφή σε ένα τέτοιο δέντρο, **στην πραγματικότητα διαγράφεται ένας κόμβος, ο οποίος είναι είτε φύλλο, είτε έχει ένα μόνο παιδί,** αφού ένας τέτοιος κόμβος παίρνει την θέση του κόμβου που απαλείφεται από το δέντρο. Δείτε για παράδειγμα τι θα συμβεί αν διαγράψουμε τον κόμβο 35 από το δέντρο του σχ. 8.21. Στην θέση του μπαίνει το 44 ή το 20. Προφανώς μας ενδιαφέρουν τα δεδομένα του κόμβου που διαγράφεται, αλλά μιλώντας για ΚΜΔ **ρίχνουμε το βάρος στο χρώμα των κόμβων**, στην πιθανή διατάραξη των ιδιοτήτων του δέντρου και στην αποκατάστασή τους.

Ένα πρόβλημα που μπορεί να παρουσιαστεί είναι ότι **μια διαγραφή ενός μαύρου κόμβου μπορεί να οδηγήσει σε παραβίαση της ιδιότητας 5 των ΚΜΔ**, δηλαδή τώρα όλες οι διαδρομές από ένα κόμβο προς οποιοδήποτε από τους NULL κόμβους απογόνους του να μη περιέχουν τον ίδιο αριθμό μαύρων κόμβων. Για να αποφύγουμε αυτό το πρόβλημα, **όταν διαγράφεται ένας μαύρος κόμβος και στην θέση του παραμένει ένα μαύρο παιδί, αυτό το παιδί θεωρείται διπλά μαύρο**. Στη συνέχεια πρέπει να ασχοληθούμε για να ξανακάνουμε το διπλά μαύρο παιδί απλά μαύρο.

Αναφέρουμε συνοπτικά για λόγους πληρότητας και με απλά μόνο παραδείγματα τις περιπτώσεις που εμφανίζονται στις διαγραφές κόμβων από ΚΜΔ: Συμβολίζουμε με:

y : τον κόμβο που θέλουμε να διαγράψουμε
z : τον κόμβο που θα διαγραφεί
x : το παιδί του κόμβου που διαγράψαμε
w : τον αδελφό του x
px : τον γονέα του x. Υποθέτουμε ότι ο x είναι αριστερό παιδί του p. Εάν είναι δεξί παιδί, τότε όλα είναι συμμετρικά.
w->left: το αριστερό παιδί του w.
w->right:το δεξί παιδί του w.

Για να διευκρινίσουμε τα παραπάνω, αν στο δέντρο του σχήματος 8.21 θέλουμε για παράδειγμα να διαγράψουμε το 10, τότε στην θέση του 10 θα μπει είτε το 7 είτε το 13. Έστω ότι θα μπει το 7. Το y στο παραπάνω παράδειγμα είναι ο κόμβος 10, το z όμως είναι ο κόμβος του 7.

Διακρίνουμε τώρα τα εξής ενδεχόμενα και περιπτώσεις:

Ενδεχόμενο Α:

α) **z κόκκινος**: προχωρούμε σε διαγραφή κατά τα γνωστά χωρίς πρόβλημα.

β) **z μαύρος με κόκκινο παιδί**: προχωρούμε σε διαγραφή κατά τα γνωστά και αλλαγή του παιδιού του z σε μαύρο.

Ενδεχόμενο Β:

**z μαύρος με μαύρο παιδί**. Το παιδί x του z γίνεται διπλά μαύρο. Η επιπλέον μονάδα μαύρου χρώματος μεταφέρεται προς τα επάνω μέχρι:

- Την ρίζα, οπότε ο αλγόριθμος τερματίζεται ή
- Να βρούμε κατάλληλο κόκκινο κόμβο, τον οποίο να κάνουμε μαύρο ή
- Να λυθεί το πρόβλημα με περιστροφές και επαναχρωματισμούς

Περιπτώσεις ενδεχομένου Β:

Περίπτωση 1: **w κόκκινο**

- Το w γίνεται μαύρο
- Το px γίνεται κόκκινο
- Εκτελούμε αριστερή περιστροφή γύρω από τον px
- Μετάβαση στην περίπτωση 2.

Περίπτωση 2: **w μαύρο**

α) **Και τα δυο παιδιά του w είναι μαύρα**.

- Το w γίνεται κόκκινο
- Το x γίνεται μαύρο (από διπλά μαύρο)
- Μεταφορά του μαύρου στον px:
  - Αν px κόκκινο, τότε το px γίνεται μαύρο και τερματισμός
  - Αν px μαύρο, τότε το px γίνεται διπλά μαύρο και επανάληψη διαδικασίας με x = px

β) **Το w->left είναι κόκκινο και το w->right είναι μαύρο.**

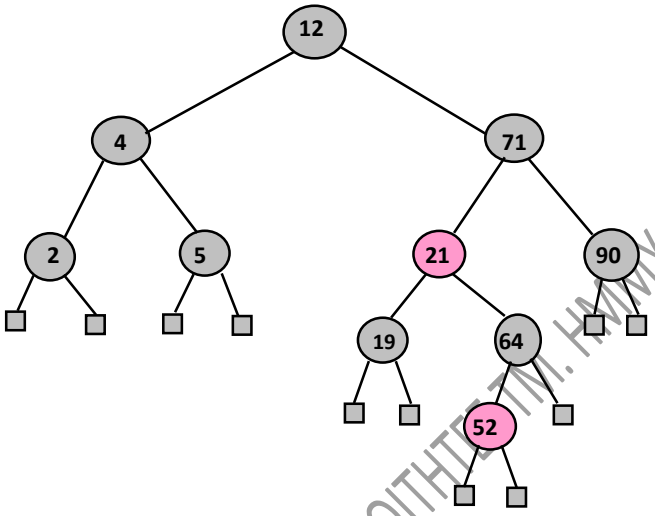
- Το w γίνεται κόκκινο
- Το w->left γίνεται μαύρο
- Εκτελούμε δεξιά περιστροφή γύρω από τον w και μεταπίπτουμε στην περίπτωση 2γ.

γ) **Το w->right είναι κόκκινο**.

- Το w->right γίνεται μαύρο
- Το w γίνεται ό,τι ήταν το χρώμα του px
- Το px γίνεται μαύρο
- Εκτελούμε αριστερή περιστροφή γύρω από τον px και τερματισμός.

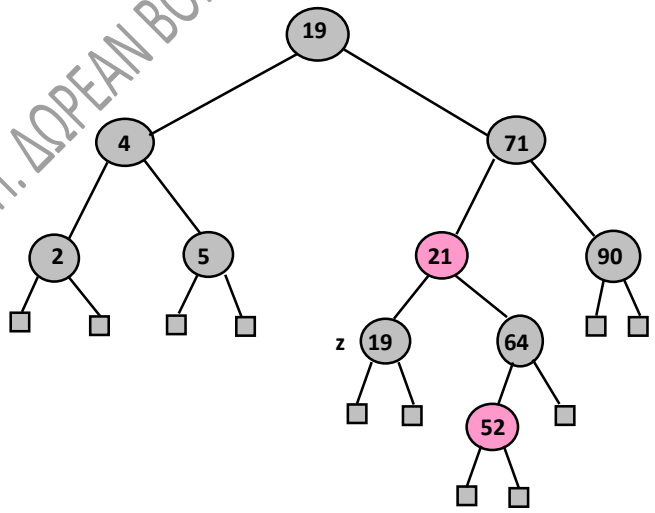
**Παράδειγμα:**

Στο παράδειγμα του σχ. 8.23α που ακολουθεί δίδεται ένα παράδειγμα διαγραφής από ένα ΚΜΔ. Στο δέντρο αυτό ζητούμε να διαγράψουμε το 12. Το διπλά μαύρο χρώμα εμφανίζεται όχι ως γκρι, αλλά ως σκούρο μαύρο.



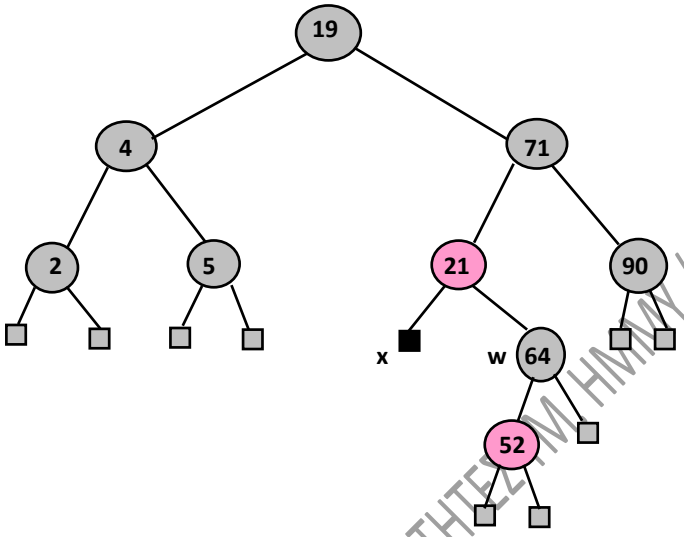

Σχ. 8.23α

Αντιγράφουμε αρχικά το 19 στην θέση του 12 (σχ. 8.23β):



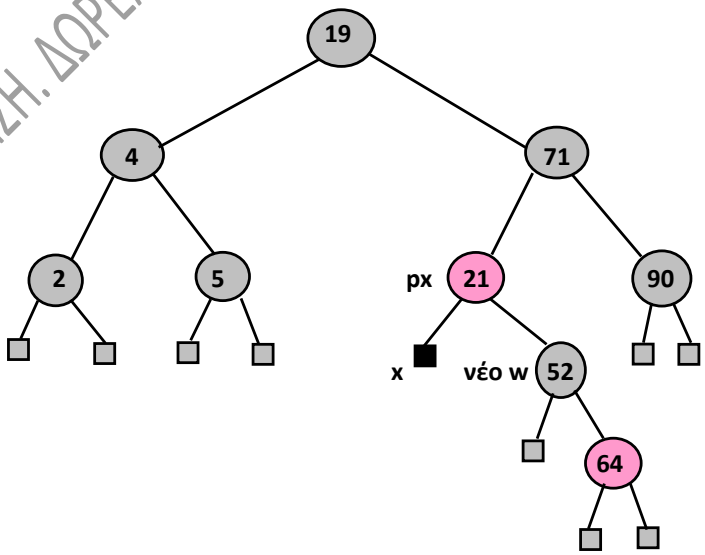

Σχ. 8.23β

Τώρα διαγράφουμε τον κόμβο του z. Έχουμε λοιπόν το ενδεχόμενο Β που αναφέραμε παραπάνω, όπου ο z είναι μαύρος με μαύρο παιδί. Το παιδί του x γίνεται διπλά μαύρο (σχ. 8.23γ):



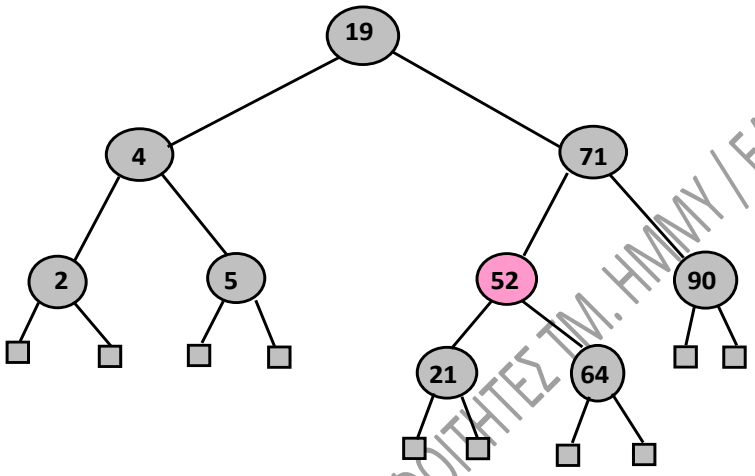

Σχ. 8.23γ

Εδώ έχουμε την περίπτωση 2β, δηλαδή το w->left κόκκινο και το w->right μαύρο, οπότε το w γίνεται κόκκινο, το w->left γίνεται μαύρο, εκτελούμε δεξιά περιστροφή στον w και καταλήγουμε στο σχ. 8.23δ:




Σχ. 8.23δ

Μεταπίπτουμε λοιπόν στην περίπτωση 2γ. Τώρα ο w είναι ο «νέος w» που φαίνεται στο σχ 8.23δ και ο w->right είναι κόκκινος. Άρα μετατρέπουμε τον w->right σε μαύρο. το w γίνεται ό,τι ήταν το px, δηλαδή κόκκινο, το px γίνεται μαύρο («ανεβαίνει» το μαύρο από τον διπλά μαύρο x) και κάνουμε αριστερή περιστροφή στον px, οπότε οι ενέργειές μας τερματίζονται (σχ. 8.23ε).




Σχ. 8.23ε

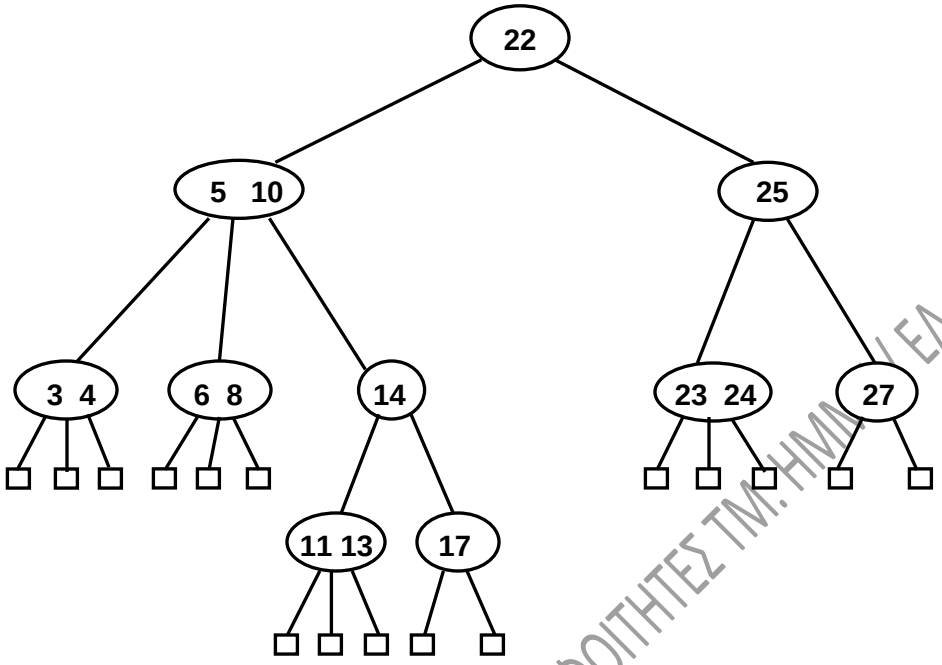


## 8.8. ΔΕΝΤΡΑ ΑΝΑΖΗΤΗΣΗΣ ΠΟΛΛΩΝ ΔΡΟΜΩΝ (Multi-Way Search Trees).

Ένα **δέντρο αναζήτησης πολλών δρόμων** είναι **διατεταγμένο δέντρο**, οι τιμές δηλαδή που αποθηκεύονται σε αυτό έχουν διάταξη. Οι ιδιότητες αυτού του δέντρου είναι οι εξής:

1. Κάθε κόμβος έχει **τουλάχιστον δύο παιδιά** και **αποθηκεύει d-1 τιμές**, όπου d ο αριθμός των παιδιών του κόμβου.
2. Για κάποιο κόμβο με παιδιά $v_1$, $v_2$,..., $v_d$, στον οποίο είναι αποθηκευμένες οι τιμές $k_1$, $k_2$,..., $k_{d-1}$, με $k_1 \le k_2 \le \ldots \le k_{d-1}$, ισχύουν τα εξής:
   i) Οι τιμές στον κόμβο $v_1$ είναι μικρότερες από το $k_1$.
   ii) Οι τιμές στον κόμβο $v_i$ είναι μεταξύ των $k_{i-1}$ και $k_i$ για i=1, 2, ..., d-1
   iii) Οι τιμές στον κόμβο $v_d$ είναι μεγαλύτερες από το $k_{d-1}$.

Στο σχ. 8.24 που ακολουθεί παρουσιάζεται ένα δέντρο πολλών δρόμων. Τα σημειωμένα με τετραγωνάκια, μπορείτε να τα θεωρήσετε ως δείκτες με τιμές NULL.




Σχ. 8.24



Παρατηρούμε ότι η ρίζα του δέντρου έχει τρία παιδιά, άρα, σύμφωνα με την πρώτη ιδιότητα που αναφέραμε παραπάνω, οι τιμές που αποθηκεύονται στην ρίζα είναι δύο. Στο πρώτο παιδί της ρίζας, οι τιμές πρέπει να είναι όλες μικρότερες από την πρώτη τιμή της ρίζας, σύμφωνα με την ιδιότητα 2(i) και πράγματι, όπως βλέπουμε, το 2, το 6 και το 8 είναι όλα μικρότερα του 11. Η τιμή του δεύτερου παιδιού της ρίζας είναι το 15 και είναι μεγαλύτερο του 11 και μικρότερο του 24 (ιδιότητα 2(ii)). Τέλος, στο τρίτο παιδί της ρίζας, οι τιμές είναι όλες μεγαλύτερες από την τελευταία τιμή της ρίζας, πράγματι, το 27 και το 32 είναι όλα μεγαλύτερα του 24 (ιδιότητα 2(iii)).

Αν σκεφτούμε με βάση τις παραπάνω ιδιότητες, μπορούμε να θεωρήσουμε ότι και τα δυαδικά δέντρα αναζήτησης, για τα οποία μιλήσαμε σε προηγούμενη ενότητα, είναι ειδικές περιπτώσεις των δέντρων πολλών δρόμων.

### 8.8.1. Διάσχιση δέντρου πολλών δρόμων.

Η διάσχιση ενός δέντρου πολλών δρόμων μπορεί να γίνει **επεκτείνοντας** κατά κάποιο τρόπο τον **ενδοδιατεταγμένο τρόπο διάσχισης** (inorder) που είχαμε δει στα δυαδικά

δέντρα. Έτσι, **επισκεπτόμαστε το στοιχείο $k_i$ του κόμβου v μεταξύ των διασχίσεων των υποδέντρων $v_i$ και $v_{i+1}$ του v**.

Στο σχ. 8.24 δηλαδή, θα επισκεφθούμε την τιμή της ρίζας, δηλαδή το 22, μεταξύ της επίσκεψης του πρώτου υποδέντρου (εδώ του πρώτου κόμβου, αυτού που έχει τις τιμές 5 και 10) και του δεύτερου υποδέντρου (εδώ του δεύτερου κόμβου, αυτού που τιμή το 25). Οι επισκέψεις και διασχίσεις είναι αναδρομικές, άρα στο πρώτο υποδέντρο, το 5 θα το επισκεφθούμε μεταξύ του κόμβου (3, 4) και του κόμβου (6,8) και το 10 θα το επισκεφθούμε μεταξύ του υποδέντρου (6, 8) και του 14. Κινούμενοι λοιπόν με αυτή την λογική, στον πίνακα του σχ. 8.25 αναφέρουμε την σειρά επίσκεψης κάθε τιμής στο δέντρο του σχ. 8.24.

| Σειρά | Τιμή |
|---|---|
| 1 | 3 |
| 2 | 4 |
| 3 | 5 |
| 4 | 6 |
| 5 | 8 |
| 6 | 10 |
| 7 | 11 |
| 8 | 13 |
| 9 | 14 |
| 10 | 17 |
| 11 | 22 |
| 12 | 23 |
| 13 | 24 |
| 14 | 25 |
| 15 | 27 |

Σχ. 8.25



### 8.7.2. Αναζήτηση σε δέντρο πολλών δρόμων.

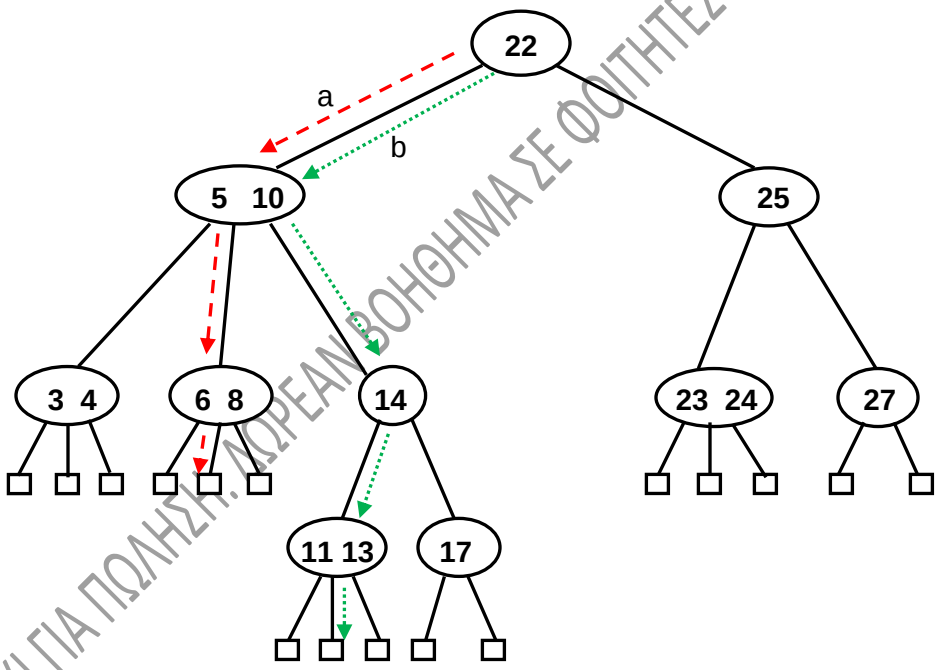
Για την αναζήτηση μιας τιμής k στο δέντρο ξεκινούμε από την ρίζα, όπως ισχύει για όλες τις αναζητήσεις σε δέντρα. **Για κάθε κόμβο του δέντρου** κάνουμε τα παρακάτω:

- **Συγκρίνουμε το k με τις τιμές του κόμβου**.

- **Εάν το k είναι ίσο με κάποια τιμή**, η αναζήτηση **τερματίζεται επιτυχώς**.
- **Εάν το k είναι μικρότερο από την πρώτη τιμή** (το $k_1$), συνεχίζουμε την **αναζήτηση στο παιδί $v_1$**.
- **Εάν το k είναι μεγαλύτερο από την i-1 τιμή και μικρότερο από την i τιμή**, συνεχίζουμε την **αναζήτηση στο παιδί $v_i$**.
- **Εάν το k είναι μεγαλύτερο από την τελευταία τιμή** (το $k_{d-1}$), συνεχίζουμε την **αναζήτηση στο παιδί $v_d$**.

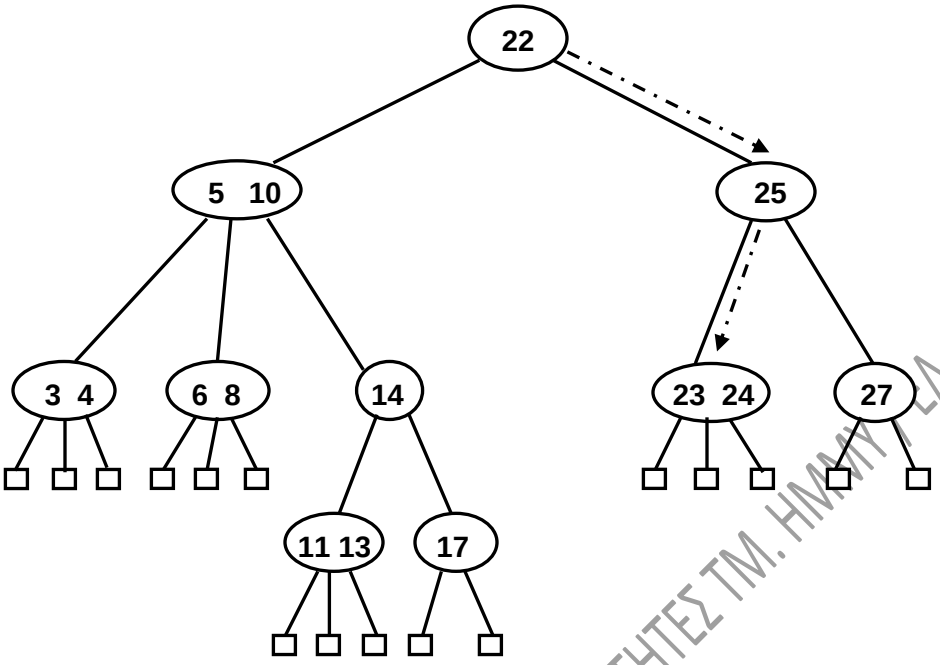
Η αναζήτηση έχει τερματιστεί ανεπιτυχώς εάν φτάσουμε σε φύλλο του δέντρου, σε NULL δείκτη δηλαδή, όπως τα έχουμε περιγράψει.

Στο σχ. 8.25 απεικονίζεται με την διακεκομμένη γραμμή a και κόκκινο χρώμα η διαδρομή που θα ακολουθήσουμε για την αναζήτηση της τιμής 7 στο δέντρο. Με την διακεκομμένη γραμμή b και πράσινο χρώμα απεικονίζεται η διαδρομή που θα ακολουθήσουμε για την αναζήτηση της τιμής 12. Και οι δύο αυτές τιμές δεν υπάρχουν στο δέντρο.




Σχ. 8.26

Ομοίως, στο σχ. 8.27, ξεκινώντας και πάλι από την ρίζα, απεικονίζεται πάλι με διακεκομμένη γραμμή η διαδρομή που ακολουθούμε όταν αναζητούμε το 24, το οποίο υπάρχει στο δέντρο.




Σχ. 8.27



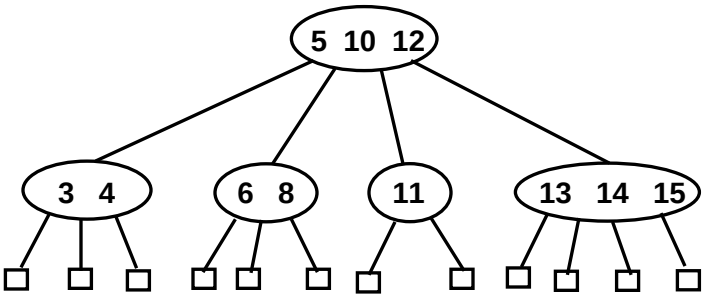
## 8.9. ΔΕΝΤΡΑ (2, 4).

Τα **δέντρα (2, 4)** είναι δέντρα πολλών δρόμων, τα οποία **παραμένουν ισοζυγισμένα** μετά από εισαγωγές και διαγραφές στοιχείων. Λέγονται και **δέντρα 2-4** ή **δέντρα 2-3-4**. Τα δέντρα αυτά έχουν τις παρακάτω δύο ιδιότητες:

1. **Κάθε κόμβος έχει το πολύ τέσσερα παιδιά** (αυτή λέγεται **ιδιότητα μεγέθους** των κόμβων).
2. **Όλοι οι δείκτες NULL βρίσκονται στο ίδιο βάθος** (αυτή λέγεται **ιδιότητα βάθους** των κόμβων).

Αν ένας κόμβος έχει 2 παιδιά, τότε λέγεται ότι είναι είδους **2-node**, αν έχει 3 παιδιά είναι είδους **3-node**, ενώ αν έχει 4 παιδιά είναι είδους **4-node**.

Στο σχ. 8.28 παρουσιάζεται ένα δέντρο (2, 4). Παρατηρείστε ότι η ρίζα του είναι κόμβος 4-node, έχει δε τέσσερα παιδιά, από τα οποία τα δύο πρώτα είναι 3-node, το τρίτο είναι 2-node και το τέταρτο είναι 4-node.



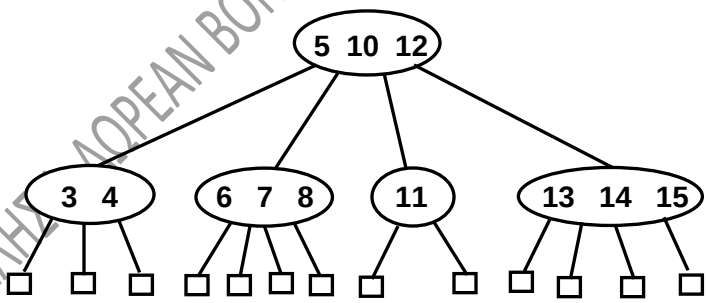

Σχ. 8.28

### 8.9.1. Εισαγωγή σε δέντρο (2, 4).

Ας δούμε πώς γίνεται η εισαγωγή μιας νέας τιμής, της k σε ένα δέντρο (2, 4).

- Αρχικά **εφαρμόζουμε αναζήτηση του k** στο δέντρο, όπως ξέρουμε για τα δέντρα αναζήτησης πολλών δρόμων. Υποθέτουμε ότι το k δεν υπάρχει ήδη, αφού θέλουμε να το εισαγάγουμε κι έτσι φτάνουμε σε NULL, με αποτυχία στη διαδικασία αναζήτησης.
- **Εισάγουμε το k στον κόμβο, στον οποίο φτάσαμε με την αναζήτηση** πριν καταλήξουμε σε NULL.

Έτσι, για την εισαγωγή του αριθμού 7 στο δέντρο του σχ. 8.28 κάνουμε αναζήτηση του 7 στο δέντρο αυτό και καταλήγουμε στο δέντρο του σχ. 8.29.

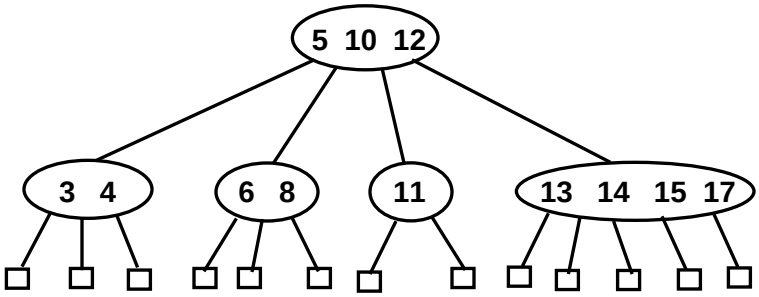



Σχ. 8.29



Εδώ πρέπει να θυμόμαστε ότι πρέπει να διατηρείται η ιδιότητα βάθους των κόμβων, δηλαδή όλα τα φύλλα να έχουν το ίδιο βάθος, όμως μπορεί να προκληθεί μια **υπερχείλιση** όπως λέμε (**overflow** στα αγγλικά) και ο κόμβος στον οποίο θα κάνουμε την εισαγωγή να γίνει 5-node κόμβος, δηλαδή κόμβος με 4 τιμές και πέντε παιδιά. Για την τελευταία περίπτωση, δείτε για παράδειγμα την εισαγωγή της τιμής 17 στο δέντρο του σχ. 8.28. Θα πρέπει να εισαχθεί στο τέταρτο παιδί της ρίζας με την αναζήτηση, όμως

τότε ο κόμβος αυτός γίνεται 5-node, αφού έχει 4 τιμές, άρα 5 παιδιά. Δείτε στο σχ. 8.30 πώς θα γίνει τότε το δέντρο:




Σχ. 8.30

Εδώ έχει παραβιαστεί η πρώτη ιδιότητα των δέντρων (2, 4) που είπαμε πιο πάνω, ότι δηλαδή κάθε κόμβος έχει το πολύ τέσσερα παιδιά.

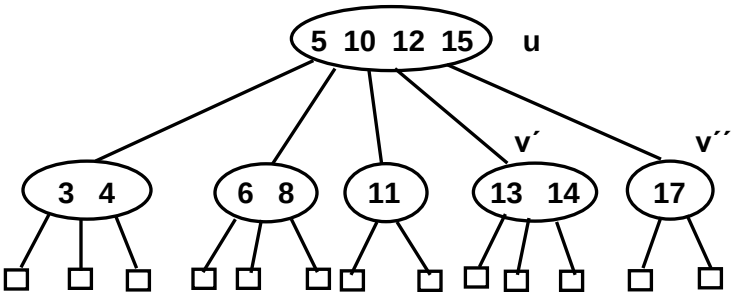
### 8.9.2. Υπερχείλιση και διάσπαση.

Η υπερχείλιση, για την οποία μιλήσαμε παραπάνω αντιμετωπίζεται με μια λειτουργία **διάσπασης** κόμβου (**split** στα αγγλικά), η οποία υλοποιείται ως εξής:

- Έστω $v_1$... $v_5$ τα παιδιά του κόμβου v και $k_1$... $k_4$ οι τιμές του v.
- **Ο κόμβος v αντικαθίσταται από δύο κόμβους**, τον v' και τον v'' ως εξής:
  - Ο κόμβος v' είναι 3-node με τιμές τις $k_1$ και $k_2$ και παιδιά τα $v_1$, $v_2$ και $v_3$.
  - Ο κόμβος v'' είναι 2-node με τιμή το $k_4$ και παιδιά τα $v_4$ και $v_5$.
- Η τιμή $k_3$ τοποθετείται στον κόμβο p, ο οποίος είναι ο γονέας του v (άρα και των v' και v'' που προέκυψαν από την διάσπασή του).

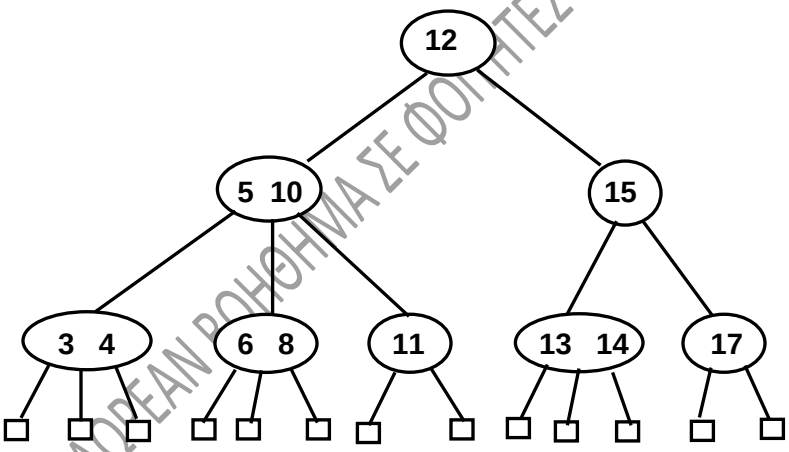
Δεδομένου ότι κατά την διάσπαση ενός κόμβου έχουμε «ανέβασμα» μιάς τιμής στον γονέα του κόμβου που διασπάται, **πρέπει να προσέξουμε μήπως το πρόβλημα της υπερχείλισης μεταφερθεί στο πιο πάνω επίπεδο**, στον κόμβο p, δηλαδή. Αν συμβεί αυτό, τότε πρέπει, ακολουθώντας τον ίδιο αλγόριθμο, να διασπάσουμε και τον κόμβο p. Αυτές **οι διαδοχικές διασπάσεις μπορεί να φτάσουν μέχρι την ρίζα** του δέντρου, όπως φαίνεται στο παράδειγμα που ακολουθεί.

Στο σχ. 8.31 προχωρούμε σε διάσπαση του 5-node που προέκυψε από την εισαγωγή του 17:




Σχ. 8.31

Με το «ανέβασμα» του 15 στον u, ο κόμβος της ρίζας έγινε τώρα 5-node, οπότε πρέπει να διασπαστεί, με το 12 να ανεβαίνει στον γονέα του. Γονέας της ρίζας δεν υπάρχει, οπότε **δημιουργείται ένας καινούργιος κόμβος** και τοποθετείται εκεί το 12 (σχ. 8.32), άρα **το δέντρο ψηλώνει κατά ένα επίπεδο**.




Σχ. 8.32



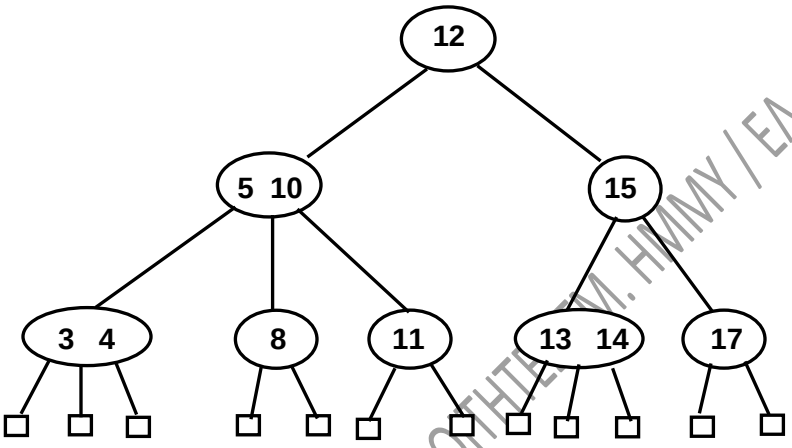
### 8.9.3. Διαγραφή από δέντρο (2, 4).

Προχωρούμε στην διαγραφή μιας τιμής k από ένα δέντρο (2, 4).

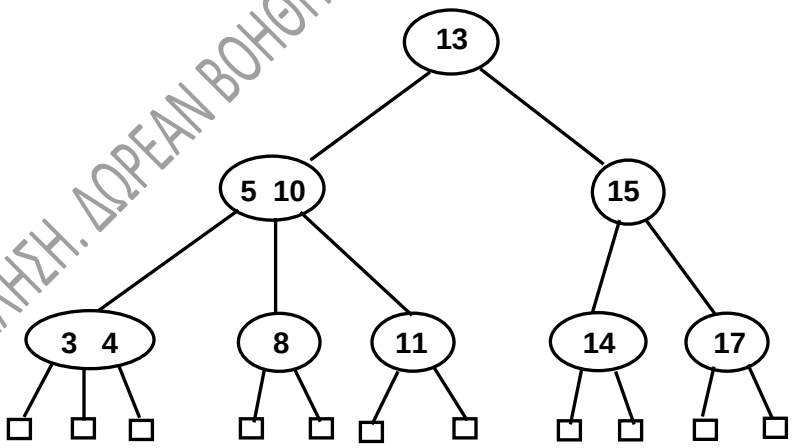
- Αρχικά **εφαρμόζουμε αναζήτηση του k** στο δέντρο, όπως ξέρουμε για τα δέντρα αναζήτησης πολλών δρόμων.
- Η διαγραφή μπορεί πάντα **να περιοριστεί** στην περίπτωση όπου **η τιμή που πρόκειται να διαγραφεί είναι αποθηκευμένη σε κόμβο, του οποίου τα παιδιά είναι NULL**. Σε διαφορετική περίπτωση, **αντικαθιστούμε το στοιχείο που πρόκειται να διαγράψουμε με το προηγούμενο ή το επόμενό του**

**σύμφωνα με τον ενδοδιατεταγμένο τρόπο διάσχισης** και μετά το διαγράφουμε.

Στο δέντρο του σχ. 8.33α διαγράφουμε το 6 από το δέντρο του σχ. 8.32, ενώ στη συνέχεια, στο σχ. 8.33β διαγράφουμε την ρίζα, το 12 δηλαδή. Παρατηρείστε ότι για την διαγραφή του 12 το αντικαταστήσαμε με το 13, το οποίο είναι η επόμενη τιμή στο δέντρο σύμφωνα με τον ενδοδιατεταγμένο τρόπο διάσχισης.




Σχ. 8.33α

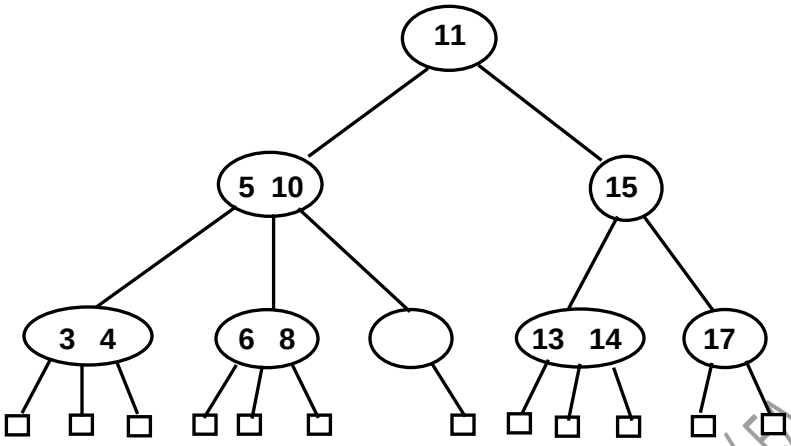



Σχ. 8.33β



### 8.9.4. Έλλειμμα, μεταφορά και συγχώνευση.

Ας θεωρήσουμε ότι θέλουμε να διαγράψουμε την ρίζα από το δέντρο του σχ. 8.32 και στην θέση της τοποθετούμε το 11, όπως είναι επιτρεπτό να κάνουμε. Τότε το δέντρο που θα έχουμε είναι αυτό του σχ. 8.34:




Σχ. 8.34

Ο κόμβος που περιείχε το 11 δεν μπορεί να απαλειφθεί, διότι ο κόμβος (5, 10) πρέπει να έχει τρία παιδιά. Επίσης ο κόμβος που περιείχε το 11 δεν μπορεί να αντικατασταθεί με τιμή NULL, γιατί τότε όλοι οι δείκτες NULL δεν θα βρίσκονται στο ίδιο βάθος. Ο κόμβος λοιπόν που περιείχε το 11 έγινε τώρα 1-node, δηλαδή κόμβος με καμμιά τιμή και με ένα παιδί, πράγμα που δεν επιτρέπεται από τις ιδιότητες των (2, 4) δέντρων. Η κατάσταση αυτή που δημιουργήθηκε λέγεται **έλλειμμα** και στα αγγλικά **underflow**.

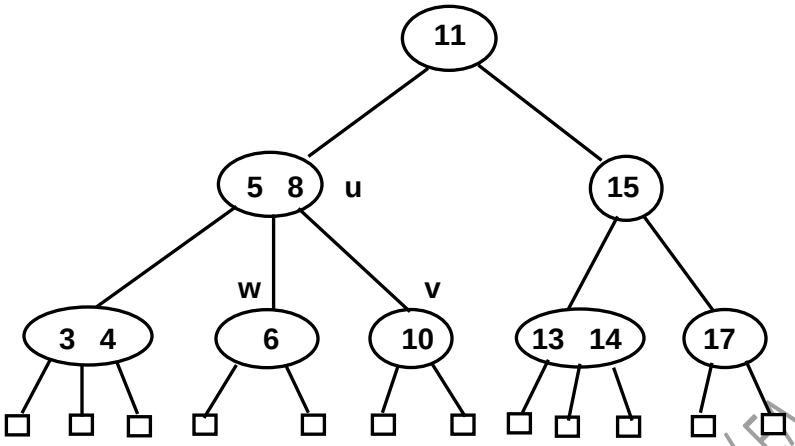
Έστω v ο κόμβος στον οποίο δημιουργείται το έλλειμμα. Για να διορθώσουμε την κατάσταση που περιγράψαμε πιο πάνω, κάνουμε τα εξής ανάλογα με την περίπτωση:

**Λειτουργία μεταφοράς (transfer operation):**

**Εάν ένας από τους πλησιέστερους αδελφούς κόμβους του v είναι κόμβος 3-node ή 4-node**, έχει δηλαδή 3 ή 4 φύλλα. Εάν υπάρχει τέτοιος κόμβος, έστω ο w, εκτελούμε μια **λειτουργία μεταφοράς** (**transfer operation**). Σε αυτή την λειτουργία κάνουμε τα εξής:

- **Μεταφέρουμε ένα παιδί** (ένα φύλλο δηλαδή) **του w στον v**.
- **Μεταφέρουμε μια τιμή του w στον u**, ο οποίος είναι ο γονέας του v και του w.
- **Μεταφέρουμε μια τιμή του u στον v** (ώστε να διατηρείται η διάταξη στο δέντρο).

Στο σχ. 8.35 παρουσιάζεται τι θα συμβεί μετά την λειτουργία μεταφοράς στο δέντρο του σχ. 8.34:



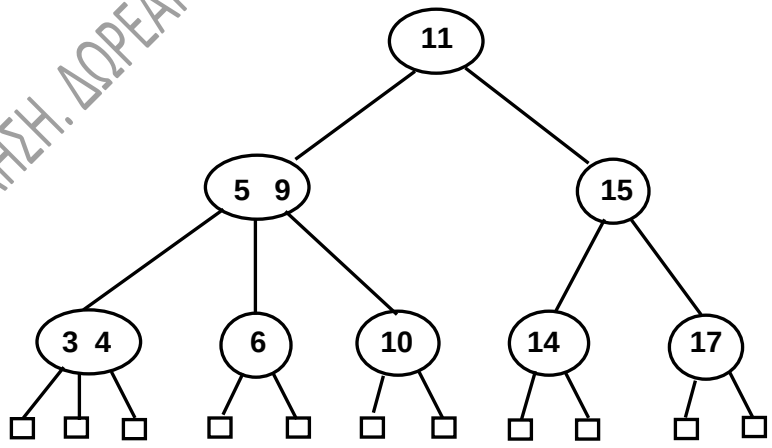

Σχ. 8.35

**Λειτουργία συγχώνευσης (fusion operation):**

**Εάν ο v έχει μόνο ένα αδελφό ή εάν και οι δύο πλησιέστεροι αδελφοί του είναι 2-node**, τότε εφαρμόζουμε μια **λειτουργία συγχώνευσης** (**fusion operation**). Σε αυτή την λειτουργία κάνουμε τα εξής:

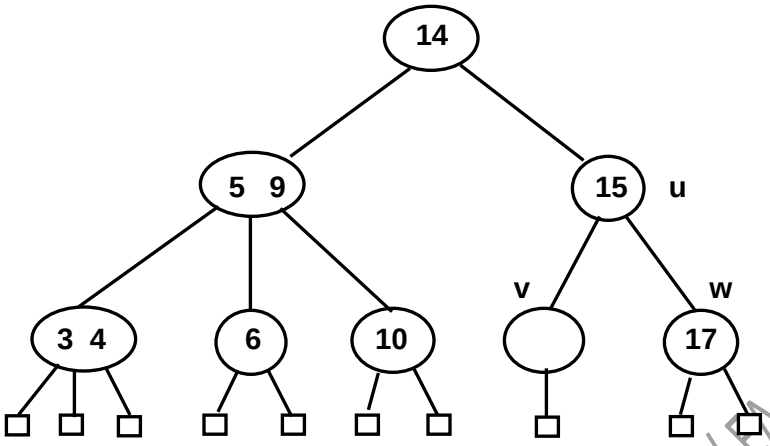
- **Ενώνουμε τον v με τον κοντινό αδελφό του** και δημιουργούμε ένα καινούργιο κόμβο, τον v΄.
- **Μεταφέρουμε μια τιμή από τον** γονέα **u στον v΄**.

Στο δέντρο του σχ. 8.36 που ακολουθεί θέλουμε να διαγράψουμε την τιμή 11.



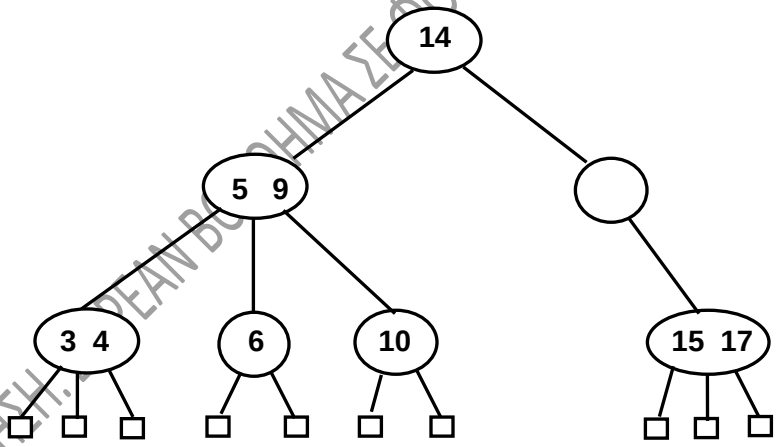

Σχ. 8.36

Η τιμή 14 μπορεί να πάρει την θέση του 11, οπότε προκύπτει έλλειμμα στο πρώτο παιδί του 15, το οποίο γίνεται κόμβος 1-node, όπως στο σχ. 8.37:



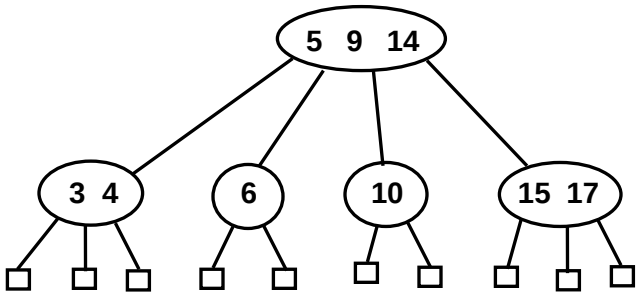

Σχ. 8.37

Ο κόμβος v συγχωνεύεται με τον αδελφό του, τον κόμβο του 17 και το 15 μεταφέρεται στον νέο κόμβο. Τότε όμως δημιουργείται έλλειμμα στο πιο πάνω επίπεδο, όπως φαίνεται στο σχ. 8.38:




Σχ. 8.38

Τώρα, αφού ο κενός κόμβος έχει ένα μόνο αδερφό, συγχωνεύεται με τον αδερφό του, το 14 κατεβαίνει στον καινούργιο αυτό κόμβο και έτσι το δέντρο χαμηλώνει κατά ένα επίπεδο, όπως βλέπουμε στο σχ. 8.39.

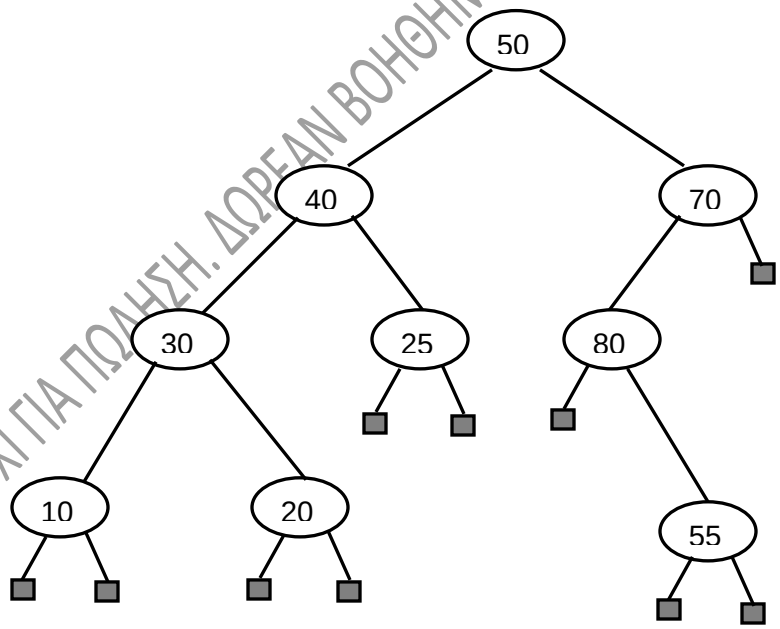



Σχ. 8.39

## 8.10. ΝΗΜΑΤΙΚΑ ΔΕΝΤΡΑ.

Σε ένα δυαδικό δέντρο (γενικά, δεν μιλάμε μόνο για δέντρο αναζήτησης) υπάρχουν δείκτες, από τους οποίους κάποιοι έχουν τιμή NULL. **Οι δείκτες αυτοί είναι δυνατό να αξιοποιηθούν** με τέτοιο τρόπο, ώστε να μην έχουν τιμή NULL, **αλλά να δείχνουν σε συγκεκριμένους κόμβους μέσα στο δέντρο**.



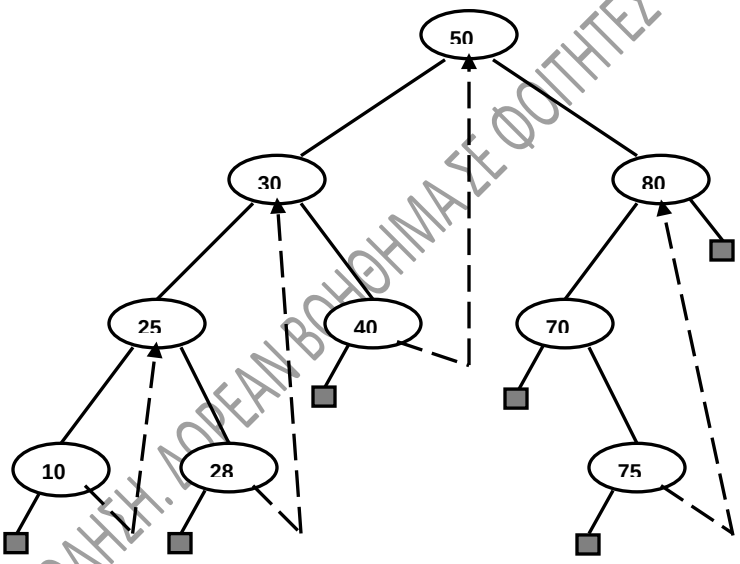



Σχ. 8.40

Αν αναφερθούμε για παράδειγμα στο δέντρο του σχ. 8.40 (τα γκρι «τετραγωνάκια» αντιστοιχούν σε κόμβους NULL), θα δούμε ότι η ενδοδιατεταγμένη διάσχιση δίνει σαν

αποτέλεσμα την επίσκεψη των κόμβων με την εξής σειρά: 10, 30, 20, 40, 25, 50, 80, 55, 70. Για να επισκεφθούμε τον κάθε κόμβο, γνωρίζοντας τον δείκτη στη ρίζα του δέντρου, πρέπει για κάθε βήμα μας να μεταβαίνουμε στην ρίζα αρχικά και από εκεί να μετακινούμαστε μέσα στο δέντρο. Εκμεταλλευόμενοι κάποιους από τους δείκτες του δέντρου, οι οποίοι έχουν τιμή NULL, μπορούμε να συντομεύσουμε και να κάνουμε ευκολότερη την μετακίνηση αυτή.

Για τον σκοπό αυτό, **ο δεξιός δείκτης κάθε κόμβου, όταν έχει τιμή NULL,** μετατρέπεται έτσι ώστε να **δείχνει στον επόμενο κόμβο του δέντρου, λαμβάνοντας υπ' όψη τον ενδοδιατεταγμένο τρόπο διάσχισης**. Έτσι, το δέντρο του σχ. 8.40 μετατρέπεται σε αυτό του σχ. 8.41. Οι δείκτες, οι οποίοι δείχνουν στις καινούργιες θέσεις λέγονται **δείκτες-νήματα** και συμβολίζονται με διακεκομμένες γραμμές. Ο δεξιός δείκτης του δεξιότερου κόμβου (του 80), εξακολουθεί και τώρα να έχει τιμή NULL. Ένα τέτοιο δέντρο λέγεται **δεξιό ενδονηματικό.**




Σχ. 8.41

Ένα πρόβλημα που εμφανίζεται σε τέτοιου είδους δέντρα είναι κατά πόσο ένας δείκτης είναι κανονικός δείκτης ή δείκτης-νήμα, αφού δεν υπάρχουν διαφορετικών ειδών δείκτες ανάλογα με το εάν είναι νήματα ή όχι. Το πρόβλημα μπορεί να αντιμετωπιστεί με την εισαγωγή ενός επιπλέον πεδίου (τύπου ακεραίου) στη δομή που περιγράφει τους κόμβους. Έστω rth το πεδίο αυτό στην παρακάτω περιγραφή δομής ptr. Αν στο rth δώσουμε τιμή αληθή (1 για παράδειγμα), τότε το rp θα το κάνουμε να «δουλεύει» ως δείκτης-νήμα, αλλιώς θα είναι δείκτης προς το δεξί παιδί του κόμβου. Σε κάθε κόμβο

δηλαδή, το rth θα παίρνει τιμή ανάλογα με το τι είναι ο δεξιός δείκτης της δομής που περιγράφει τον κόμβο.:

```
struct ptr {
    int rec;
    struct ptr *lp;
    struct ptr *rp;
    int  rth; };
```

Το **αριστερό ενδονηματικό δέντρο** υλοποιείται με παρόμοιο τρόπο, αλλά εδώ **ο αριστερός δείκτης κάθε κόμβου, όταν έχει τιμή NULL,** μετατρέπεται έτσι ώστε να **δείχνει στον προηγούμενο κόμβο του δέντρου, λαμβάνοντας υπ' όψη τον ενδοδιατεταγμένο τρόπο διάσχισης**. Με ανάλογες παραδοχές και σύμφωνα με όσα αναφέρθηκαν πιο πάνω, προκύπτει η μορφή του κόμβου, όσον αφορά τον αριστερό δείκτη του.

Το δέντρο μπορεί να υλοποιηθεί **και με τους δύο τρόπους ταυτόχρονα**, οπότε λέγεται απλά **ενδονηματικό**. Τότε απαιτούνται δύο επιπλέον πεδία (τύπου ακεραίου) στη δομή που περιγράφει τον κάθε κόμβο. Έστω lth και rth τα πεδία αυτά:

```
struct ptr {
    int rec;
    struct ptr *lp;
    struct ptr *rp;
    int  lth;
    int  rth; };
```

Αν το lth έχει τιμή αληθή, τότε το lp περιέχει δείκτη-νήμα, αλλιώς είναι δείκτης προς το αριστερό παιδί του κόμβου. Αντίστοιχα, αν το rth έχει τιμή αληθή, τότε το rp περιέχει δείκτη-νήμα, αλλιώς είναι δείκτης προς το δεξί παιδί του κόμβου.

Με ανάλογο τρόπο προκύπτουν τα **προνηματικά δέντρα** και τα **μετανηματικά δέντρα** (δεξιά, αριστερά ή απλά), αν ληφθεί υπ' όψη η **προδιατεταγμένη** ή η **μεταδιατεταγμένη διάσχιση** αντίστοιχα.

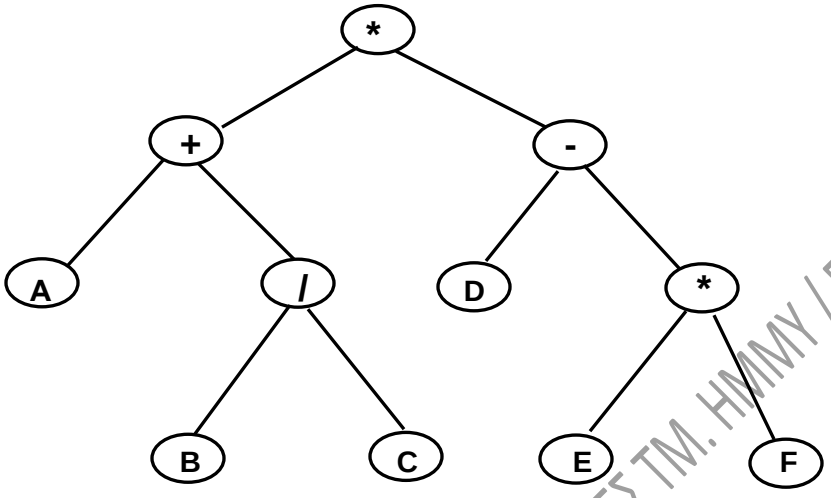


## 8.11. ΕΦΑΡΜΟΓΕΣ ΤΩΝ ΔΥΑΔΙΚΩΝ ΔΕΝΤΡΩΝ.

### 8.11.1. Αποθήκευση παραστάσεων.

Τα δυαδικά δέντρα μπορούν να χρησιμεύσουν για την **αποθήκευση παραστάσεων**. Για παράδειγμα, η παράσταση (A + B / C) * (D – E * F) αντιστοιχεί στο δέντρο του σχ. 8.42. Το δέντρο αυτό είναι **ανομοιογενές**, δηλαδή δέντρο, τα δεδομένα όλων των κόμβων του

οποίου δεν είναι ίδιου τύπου μεταξύ τους. Αποτελούνται από τελεστές (τα +, -, *, /), αλλά και τελεστέους, μεταξύ των οποίων γίνονται οι πράξεις (τα A, B, C, D, E και F). Οι δείκτες από τα φύλλα δεν σημειώνονται, αλλά υποτίθεται ότι έχουν τιμή NULL.




Σχ. 8.42

Στο δέντρο αυτό οι **τερματικοί κόμβοι είναι τελεστέοι**, ενώ **οι εσωτερικοί είναι τελεστές**. Σε όσο χαμηλότερο επίπεδο βρίσκονται κάποιοι τελεστές, τόσο μεγαλύτερης προτεραιότητας είναι η πράξη μεταξύ τους.

Για να απεικονίσουμε μια παράσταση με δυαδικό δέντρο **σαρώνουμε την παράσταση** από αριστερά προς τα δεξιά για να βρούμε τον **τελεστή με τη μικρότερη προτεραιότητα**. Αυτός **τοποθετείται στη ρίζα του δέντρου**. Έχουμε έτσι δυο υποπαραστάσεις, η μια από τις οποίες θα αποτελέσει το αριστερό παιδί του δέντρου και η άλλη το δεξί. Επαναλαμβάνεται η ίδια διαδικασία και βρίσκουμε τις ρίζες των υποδέντρων που έχουν προκύψει. Αυτό συνεχίζεται μέχρι να μπουν στο δέντρο οι τελεστέοι, στο πιο χαμηλό δηλαδή επίπεδο. Αν τώρα διασχίσουμε το πιο πάνω δέντρο με **μεταδιατεταγμένο τρόπο** θα πάρουμε την παράσταση:

A B C \ + D E F * - *

Η παράσταση αυτή, σύμφωνα και με όσα αναφέραμε στην παράγραφο 5.4, αποτελεί τη **μεταθεματική (postfix) μορφή της αρχικής** ή όπως αλλιώς λέγεται, τον **αντίστροφο Πολωνικό συμβολισμό** της. Παρατηρείστε τι θα προκύψει αν διασχίσουμε το δέντρο με **προδιατεταγμένο τρόπο**:

* + A / B C – D * E F

Η παράσταση αυτή αποτελεί την λεγόμενη **προθεματική (prefix) μορφή της αρχικής** ή όπως αλλιώς λέγεται, τον **Πολωνικό συμβολισμό** της. Η διάσχιση του δέντρου με ενδοδιατεταγμένο τρόπο δίνει την **ένθετη (infix) μορφή**, μόνο όμως στην περίπτωση που η παράσταση αυτή δεν περιέχει παρενθέσεις. Αυτό γιατί, όπως είδαμε, στο δέντρο δεν τοποθετούνται παρενθέσεις, αλλά η σειρά των πράξεων προκύπτει μόνο από την οργάνωση του δέντρου.



## 8.11.2. Αλγόριθμος Huffman.

Ο αλγόριθμος αυτός χρησιμοποιείται για την **κωδικοποίηση ενός μηνύματος**, για την μετατροπή του δηλαδή σε δυαδική μορφή. Η μετατροπή γίνεται με την **αντικατάσταση κάθε γράμματος του μηνύματος από ένα συνδυασμό bits**. Όπως γνωρίζουμε, **στον κώδικα ASCII κάθε χαρακτήρας αντιστοιχίζεται σε μια οκτάδα bits**, με αποτέλεσμα με τα 8 bits να μπορούμε να απεικονίσουμε 256 χαρακτήρες. Αντίστροφα, **εάν επιθυμούμε να διακρίνουμε μεταξύ n διαφορετικών συμβόλων, χρειαζόμαστε $\log_2 n$ bits ανά σύμβολο**.

Ας θεωρήσουμε χάριν απλότητας ότι όλα μας τα μηνύματα αποτελούνται από συνδυασμούς 8 μόνο διαφορετικών χαρακτήρων, των A, B, C, D, E, F, G, και H. Τότε μπορούμε κάθε χαρακτήρα να τον κωδικοποιήσουμε με μια τριάδα από bits, όπως φαίνεται για παράδειγμα στον πίνακα του σχ. 8.43:

| | |
|---|---|
| **A** | **000** |
| **B** | **001** |
| **C** | **010** |
| **D** | **011** |
| **E** | **100** |
| **F** | **101** |
| **G** | **110** |
| **H** | **111** |

Σχ. 8.43

Με την χρήση αυτού του κώδικα, το μήνυμα που ακολουθεί:

BACADAEAFABBAAAGAH

κωδικοποιείται με την παρακάτω ακολουθία των 54 bits:

00100001000001100010000010100000100100000000110000111

Κώδικες όπως ο ASCII ή ο ανωτέρω κώδικας των τριών bits, λέγονται **κώδικες σταθερού μήκους (fixed-length codes)**, επειδή αναπαριστούν κάθε σύμβολο, κάθε χαρακτήρα εν προκειμένω, με τον ίδιο αριθμό bits.

Μια άλλη κατηγορία κωδίκων είναι οι λεγόμενοι **κώδικες μεταβλητού μήκους (variable-length codes)**. Η χρήση αυτών των **κωδίκων** μας προσφέρει πολλές φορές πλεονεκτήματα. Σε αυτούς, **τα σύμβολα μπορεί να αναπαριστώνται με διαφορετικό πλήθος bits το καθένα**. Κλασσικό παράδειγμα τέτοιου κώδικα είναι ο κώδικας Μορς (Morse). Εκεί, το κάθε γράμμα παριστάνεται με κάποιο αριθμό από παύλες και τελείες. Παρατηρείστε τι συμβαίνει, για παράδειγμα, με το πολύ συχνά συναντώμενο στα κείμενα γράμμα Ε: επειδή ακριβώς η συχνότητά του είναι πολύ μεγάλη, το Ε έχει κωδικοποιηθεί (έχει αντιστοιχηθεί δηλαδή) με μια μόνο τελεία.

Στους υπολογιστές το αντίστοιχο της παύλας και της τελείας είναι το bit, με τιμές 0 ή 1. Γενικεύοντας την παρατήρηση που κάναμε πιο πάνω, μπορούμε να οδηγηθούμε στο να **κωδικοποιούμε με λιγότερα bits τα σύμβολα που εμφανίζονται πιο συχνά** σε ένα κείμενο **και με περισσότερα bits τα σπανιότερα**. Ας μετρήσουμε πόσο συχνά εμφανίζεται κάθε χαρακτήρας στο μήνυμα που γράψαμε πιο πάνω. Η συχνότητα εμφάνισης συνοψίζεται στο σχ. 8.44.

| Χαρακτήρας | Πλήθος χαρακτήρων στο μήνυμα |
|---|---|
| A | 9 |
| B | 3 |
| C | 1 |
| D | 1 |
| E | 1 |
| F | 1 |
| G | 1 |
| H | 1 |

Σχ. 8.44

Παρατηρείστε τώρα στον πίνακα του σχ. 8.45 μια εναλλακτική κωδικοποίηση για τους χαρακτήρες αυτούς, αντί για την κωδικοποίηση του σχ. 8.43:

| | |
|---|---|
| **A** | **0** |
| **B** | **100** |
| **C** | **1010** |
| **D** | **1011** |
| **E** | **1100** |
| **F** | **1101** |
| **G** | **1110** |
| **H** | **1111** |

Σχ. 8.45

Χρησιμοποιώντας αυτόν τον κώδικα, το μήνυμα που είχαμε προηγουμένως, δηλαδή το:

BACADAEAFABBAAAGAH

κωδικοποιείται στην παρακάτω ακολουθία bits:

10001010010110110001101010010000111001111

Το πλήθος αυτών των bits είναι 42, δηλαδή έχουμε οικονομία περισσότερη από 20% σε σχέση με την κωδικοποίηση σταθερού κώδικα που είδαμε νωρίτερα, όπου χρειαστήκαμε 54 bits.

Κατά την αποκωδικοποίηση σε ένα κώδικα σταθερού μήκους, ξέρουμε ανά πόσα bit σχηματίζεται ένας χαρακτήρας και έτσι η αποκωδικοποίηση είναι απλή και «μηχανική» θα λέγαμε. Μια από τις **δυσκολίες** που συναντά όμως κανείς χρησιμοποιώντας ένα κώδικα μεταβλητού μήκους είναι **να αποφασίσει ποιο είναι το τέλος κάθε συμβόλου** μέσα στην ακολουθία των 0 και 1. Στον κώδικα Μόρς το πρόβλημα λύνεται με την χρήση ενός «κωδικού διαχωρισμού» (μια παύση) μετά από κάθε γράμμα. Μια άλλη λύση είναι η σχεδίαση του κώδικα με τέτοιο τρόπο, ώστε **ο κωδικός για ένα σύμβολο να μη συμπίπτει με την αρχή κωδικού για οποιοδήποτε άλλο σύμβολο**. Στο παραπάνω παράδειγμα, το Α κωδικοποιείται με το 0 και το Β με το 100. Όπως παρατηρούμε, κανένας κωδικός άλλου συμβόλου δεν αρχίζει από 0 ή από 100. Ένας τέτοιος κώδικας λέγεται **«κώδικας προθέματος» (prefix code)**.

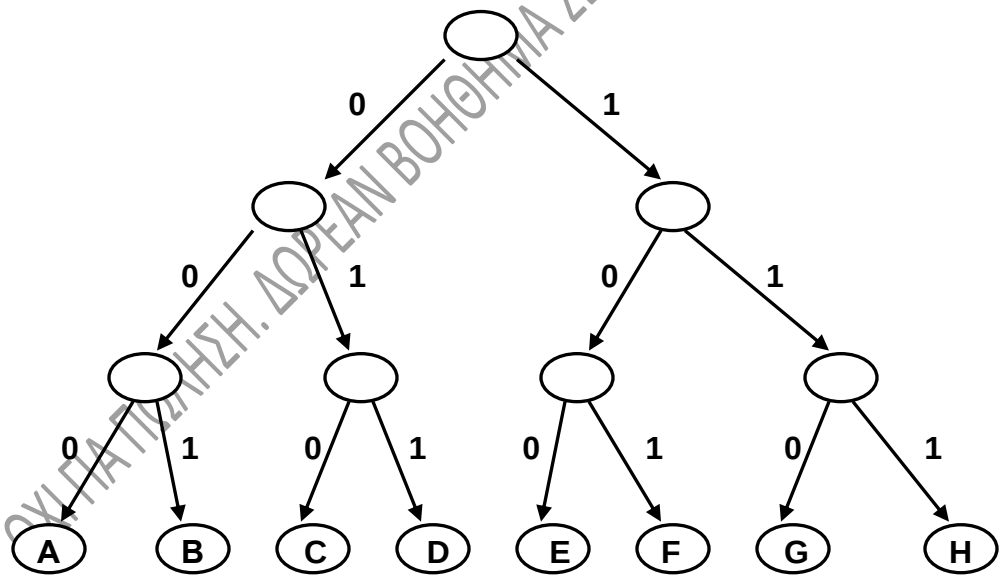
Θεωρείστε τώρα **ένα δυαδικό δέντρο, στο οποίο κάθε κόμβος «φύλλο» αντιστοιχεί σε ένα σύμβολο**. Μπορούμε να αποδώσουμε ένα **κωδικό σε κάθε σύμβολο** ως εξής: αντιστοιχούμε το **0 στη μετάβαση από την ρίζα προς το αριστερό της παιδί** και αντιστοιχούμε το **1 στη μετάβαση από την ρίζα προς το δεξί της παιδί. Επαναλαμβάνουμε την ίδια αντιστοίχιση** κατεβαίνοντας στο από κάτω επίπεδο της

ρίζας και σε όλα τα επίπεδα **μέχρι να φτάσουμε σε φύλλο. Ο κωδικός του φύλλου δημιουργείται ακολουθώντας το μονοπάτι από την ρίζα μέχρι το φύλλο, το οποίο περιέχει το σύμβολο που μας ενδιαφέρει.** Ο κώδικας που προκύπτει λέγεται **«Κώδικας Huffman»** και είναι εκ των πραγμάτων κώδικας προθέματος (prefix). Αυτό είναι προφανές αφού, λόγω της δενδροειδούς δομής που έχουμε, ένας κόμβος φύλλο (άρα σύμβολο, άρα και ο κωδικός του) δεν μπορεί να εμφανιστεί στην διαδρομή προς ένα άλλο φύλλο (δηλαδή προς ένα άλλο σύμβολο). Επίσης, **ένας κώδικας σταθερού μήκους είναι** εκ των πραγμάτων **και κώδικας προθέματος**.

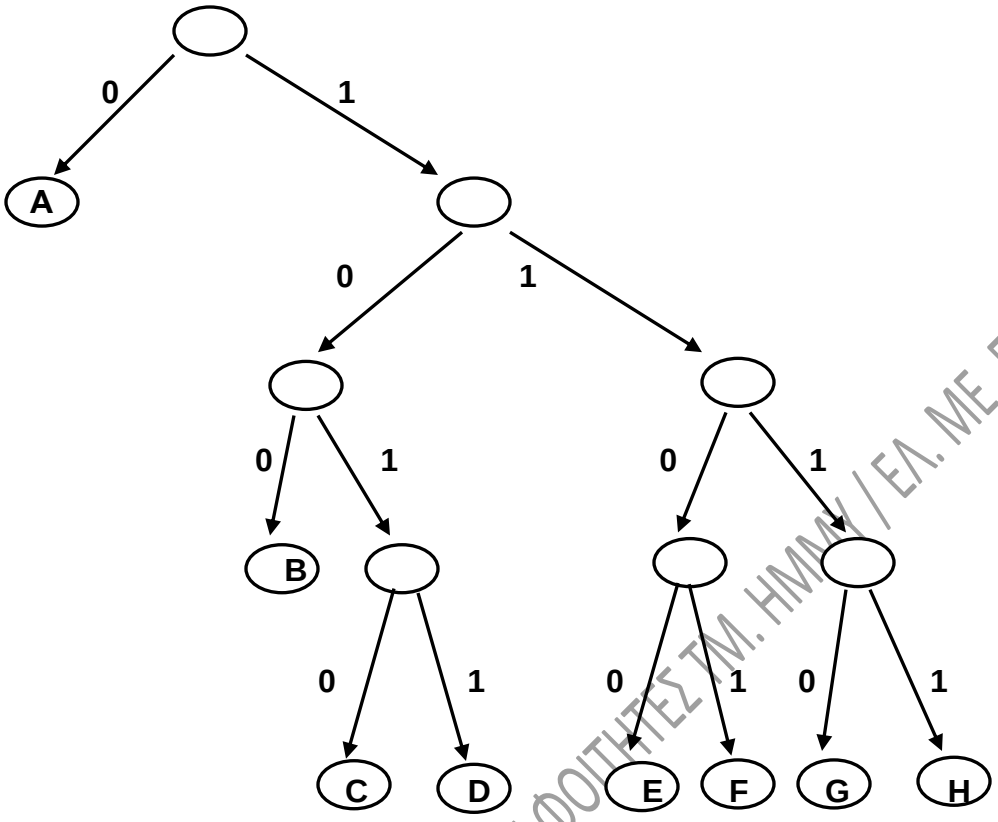
Στα σχήματα 8.46α και 8.46β που ακολουθούν, φαίνονται τα δυαδικά δέντρα για τους κώδικες των πινάκων των σχ. 8.43 και 8.45.

Μιλώντας αντίστροφα, είναι φανερός ο τρόπος με τον οποίο μπορούμε να αντιστοιχίσουμε ένα δυαδικό δέντρο σε ένα δεδομένο κώδικα προθέματος, αφού ο κώδικας, ακολουθώντας την λογική που περιγράψαμε πιο πάνω (αντιστοιχίσεις των 0 και 1) περιγράφει τελικά την μορφή του δέντρου. Το δέντρο που αντιστοιχεί στον κώδικα λέγεται **«Δυαδικό δέντρο Huffman»**.






Σχ. 8.46α




Σχ. 8.46β



**Δημιουργία του δυαδικού δέντρου Huffman:**

Θα περιγράψουμε τώρα τον τρόπο με τον οποίο, για ένα δεδομένο μήνυμα, δημιουργούμε το δυαδικό δέντρο Huffman, άρα και τον αντίστοιχο κώδικα Huffman. Για το σκοπό αυτό ακολουθούμε τα παρακάτω βήματα:

1. Αρχικά **βρίσκουμε τη συχνότητα, με την οποία εμφανίζεται κάθε σύμβολο στο μήνυμα**. Ο σκοπός μας, όπως λέχθηκε και προηγουμένως, είναι να έχουμε μικρούς κωδικούς για σύμβολα που εμφανίζονται συχνά μέσα στο μήνυμα και μεγαλύτερους κωδικούς για τα σπανιότερα. Εννοείται ότι ο κώδικας που προκύπτει πρέπει να είναι κώδικας προθέματος.
2. **Ταξινομούμε κατά τη συχνότητα εμφάνισης** την ακολουθία των συμβόλων-συχνοτήτων που προέκυψε.
3. Παίρνουμε **τα δυο σύμβολα με τη μικρότερη συχνότητα** εμφάνισης στο μήνυμα. **Κάνουμε τα σύμβολα αυτά φύλλα ενός δέντρου**, ενώ παράλληλα

**δημιουργούμε ένα κόμβο πατέρα με συχνότητα το άθροισμα των συχνοτήτων** των δυο φύλλων.

4. **Τα δυο φύλλα απομακρύνονται από την ακολουθία των συμβόλων-συχνοτήτων και εισάγεται ο κόμβος πατέρας που δημιουργήθηκε**, σε τέτοια θέση όμως, ώστε η ακολουθία των συμβόλων-συχνοτήτων να παραμένει ταξινομημένη ως προς την συχνότητα.
5. **Επαναλαμβάνουμε τα πιο πάνω βήματα 3 και 4 με τα δυο μικρότερης συχνότητας σύμβολα που υπάρχουν τώρα στην ακολουθία** των συμβόλων-συχνοτήτων, μέχρι να μείνει ένα μόνο σύμβολο.
6. **Το τελευταίο σύμβολο αποτελεί την ρίζα του δέντρου**.

Αφού ολοκληρωθεί το δέντρο, **βρίσκουμε τον κωδικό κάθε συμβόλου** αν **διατρέξουμε το δέντρο** ξεκινώντας από την ρίζα. Θεωρούμε ότι έχουμε τιμή **0 για κάθε αριστερό κλαδί** που συναντούμε στη διαδρομή μας και τιμή **1 για κάθε δεξί κλαδί**.

**Παράδειγμα:**

Έστω το μήνυμα: ΑΠΛΗυΑΚΟΛΟΥΘΙΑ

Η συχνότητα εμφάνισης των συμβόλων είναι η εξής:

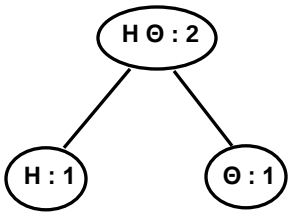
| Σύμβολο | Συχνότητα |
|---|---|
| Α | 3 |
| Π | 1 |
| Λ | 2 |
| Η | 1 |
| υ | 1 |
| Κ | 1 |
| Ο | 2 |
| Υ | 1 |
| Θ | 1 |
| Ι | 1 |

Ταξινομούμε τον παραπάνω πίνακα κατά τη συχνότητα και προκύπτει η παρακάτω ακολουθία συμβόλων-συχνοτήτων:

| Σύμβολο | Συχνότητα |
|---|---|
| Η | 1 |
| Θ | 1 |
| Ι | 1 |
| Κ | 1 |
| Π | 1 |
| Υ | 1 |
| υ | 1 |
| Λ | 2 |
| Ο | 2 |
| Α | 3 |

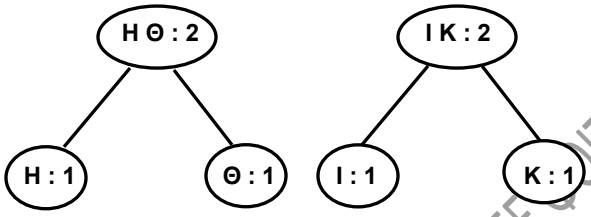
Υλοποιούμε τώρα τα βήματα που περιγράψαμε για την δημιουργία του δέντρου Huffman παρουσιάζοντας κάθε φορά το δέντρο και την ακολουθία συμβόλων-συχνοτήτων που προκύπτουν.

α)



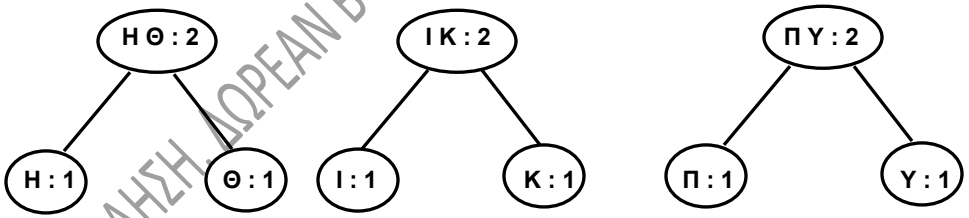

| Σύμβολο | Συχνότητα |
|---|---|
| Ι | 1 |
| Κ | 1 |
| Π | 1 |
| Υ | 1 |
| υ | 1 |
| Λ | 2 |
| Ο | 2 |
| Η Θ | 2 |
| Α | 3 |

β)



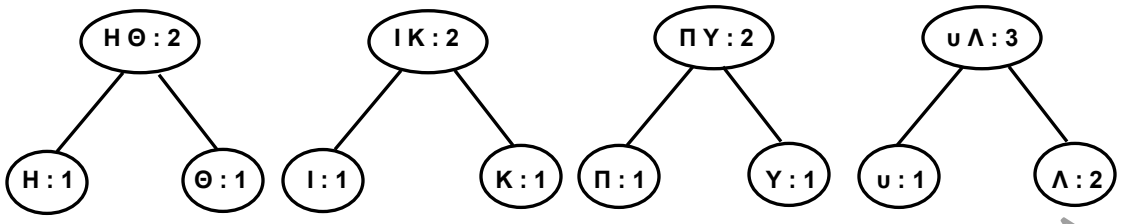

| Σύμβολο | Συχνότητα |
|---|---|
| Π | 1 |
| Υ | 1 |
| υ | 1 |
| Λ | 2 |
| Ο | 2 |
| Η Θ | 2 |
| Ι Κ | 2 |
| Α | 3 |

γ)



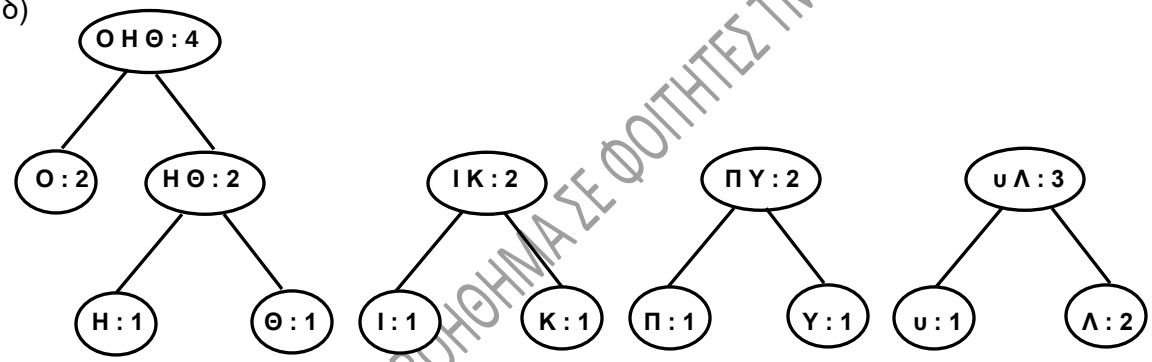

| Σύμβολο | Συχνότητα |
|---|---|
| υ | 1 |
| Λ | 2 |
| Ο | 2 |
| Η Θ | 2 |
| Ι Κ | 2 |
| Π Υ | 2 |
| Α | 3 |

δ)



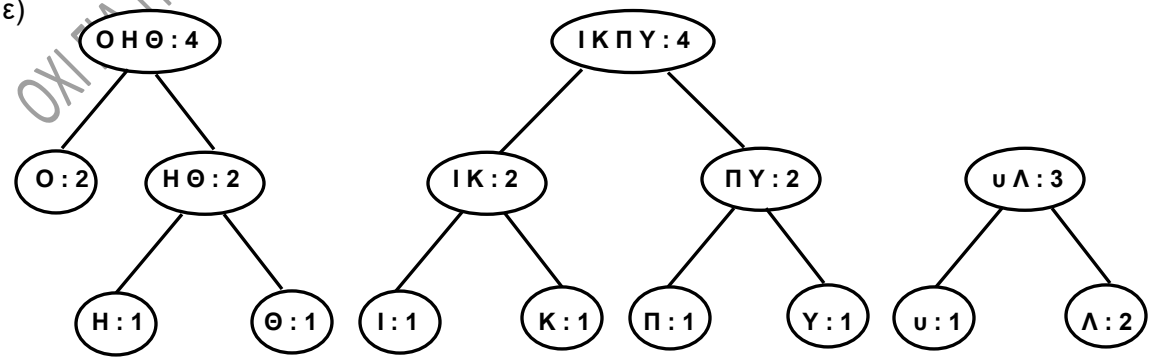

| Σύμβολο | Συχνότητα |
|---|---|
| Ο | 2 |
| Η Θ | 2 |
| Ι Κ | 2 |
| Π Υ | 2 |
| Α | 3 |
| υ Λ | 3 |

δ)



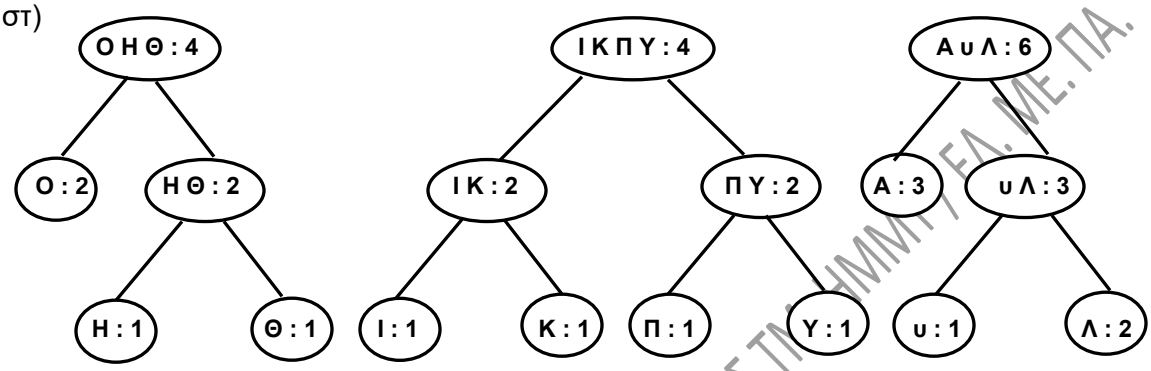

| Σύμβολο | Συχνότητα |
|---|---|
| Ι Κ | 2 |
| Π Υ | 2 |
| Α | 3 |
| υ Λ | 3 |
| Ο Η Θ | 4 |

ε)



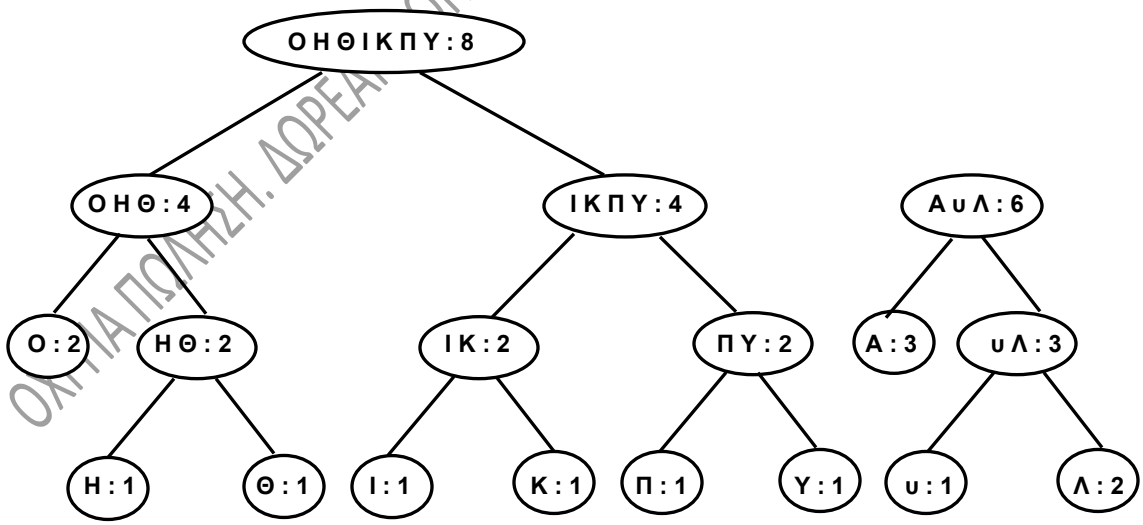
| Σύμβολο | Συχνότητα |
|---|---|
| Α | 3 |
| υ Λ | 3 |
| Ο Η Θ | 4 |
| Ι Κ Π Υ | 4 |

στ)



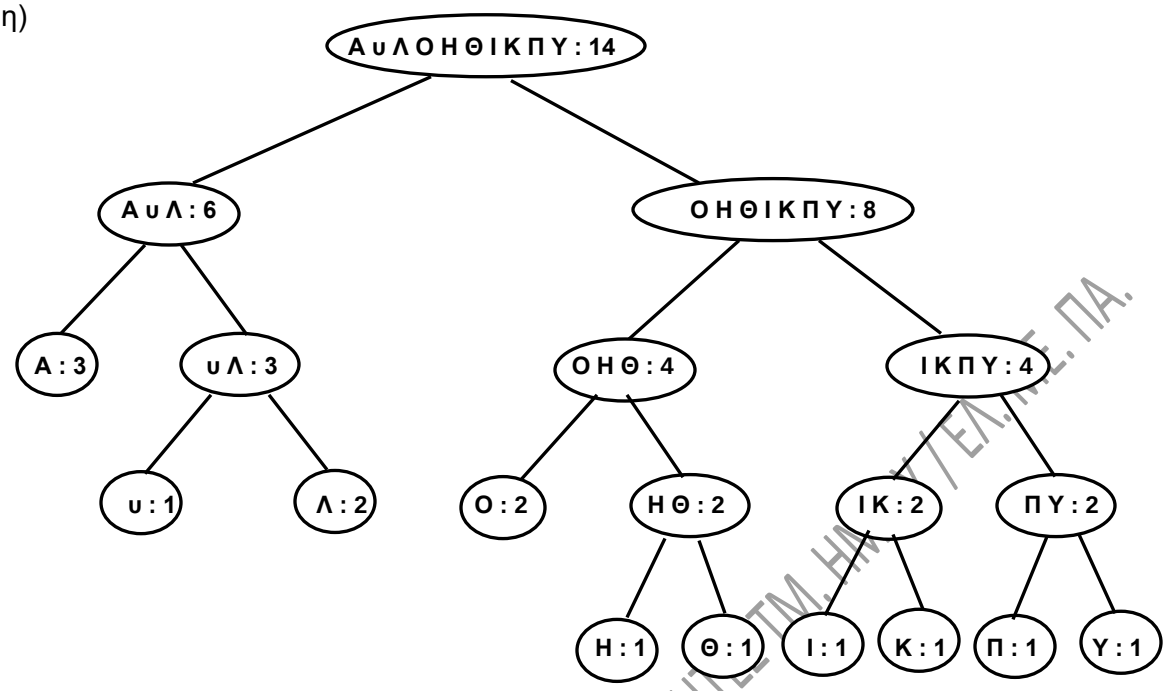

| Σύμβολο | Συχνότητα |
|---|---|
| Ο Η Θ | 4 |
| Ι Κ Π Υ | 4 |
| Α υ Λ | 6 |

ζ)




| Σύμβολο | Συχνότητα |
|---|---|
| Α υ Λ | 6 |
| Ο Η Θ Ι Κ Π Υ | 8 |

η)




| Σύμβολο | Συχνότητα |
|---|---|
| Α υ Λ Ο Η Θ Ι Κ Π Υ | 14 |

Με βάση όσα προαναφέραμε, στον πίνακα του σχ. 8.47 που ακολουθεί φαίνονται οι κωδικοί των διαφόρων συμβόλων του μηνύματός μας σε αντιστοιχία και με τη συχνότητα εμφάνισής τους στο μήνυμα:

| Σύμβολο | Κωδικός | Συχνότητα |
|---|---|---|
| Α | 00 | 3 |
| Π | 1110 | 1 |
| Λ | 011 | 2 |
| Η | 1010 | 1 |
| υ | 010 | 1 |
| Κ | 1101 | 1 |
| Ο | 100 | 2 |
| Υ | 1111 | 1 |
| Θ | 1011 | 1 |
| Ι | 1100 | 1 |

Σχ. 8.47

## 8.12. ΔΥΑΔΙΚΟΙ ΣΩΡΟΙ.

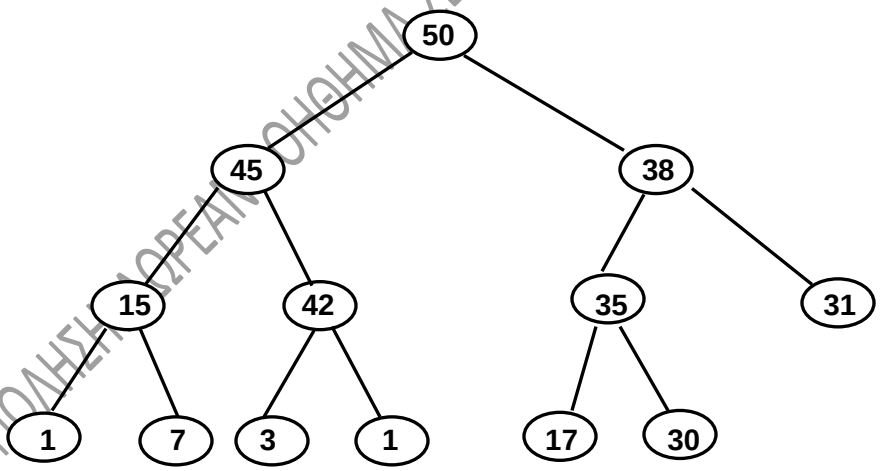
Όταν μιλάμε για **δυαδικό σωρό** (**Binary Heap**) στους υπολογιστές εννοούμε μια ειδικής μορφής **δυαδικό δέντρο** με τις εξής ιδιότητες:

Ιδιότητα 1:

**Το δέντρο είναι σχεδόν πλήρες**. Με αυτό εννοούμε ότι όλα τα επίπεδα του δέντρου είναι γεμάτα, εκτός ίσως από το τελευταίο. Το τελευταίο επίπεδο γεμίζει από αριστερά προς τα δεξιά.

Ιδιότητα 2:

**Η τιμή κάθε κόμβου** του δέντρου **είναι μικρότερη ή ίση από την αντίστοιχη τιμή του γονέα του**. Στην περίπτωση αυτή μιλούμε για **σωρό μεγίστων** (max heap). Αντίστοιχα, στον **σωρό ελαχίστων** (min heap) **η τιμή κάθε κόμβου είναι μεγαλύτερη ή ίση από την αντίστοιχη τιμή του γονέα του**. Είναι προφανές ότι σε ένα σωρό μεγίστων στην ρίζα του δέντρου βρίσκεται αποθηκευμένη η μέγιστη τιμή, ενώ σε ένα σωρό ελαχίστων στην ρίζα του δέντρου βρίσκεται η ελάχιστη τιμή. Το παρακάτω δέντρο αποτελεί ένα σωρό μεγίστων.




Σχ. 8.48

Μία από τις κύριες εφαρμογές των σωρών είναι η υλοποίηση των ουρών προτεραιότητας. Θα δείξουμε στη συνέχεια τις διαδικασίες εισαγωγής ενός κόμβου στον σωρό και διαγραφής της ρίζας ενός σωρού μεγίστων, καθώς και την δημιουργία ενός τέτοιου σωρού.

### 8.12.1. Εισαγωγή κόμβου:

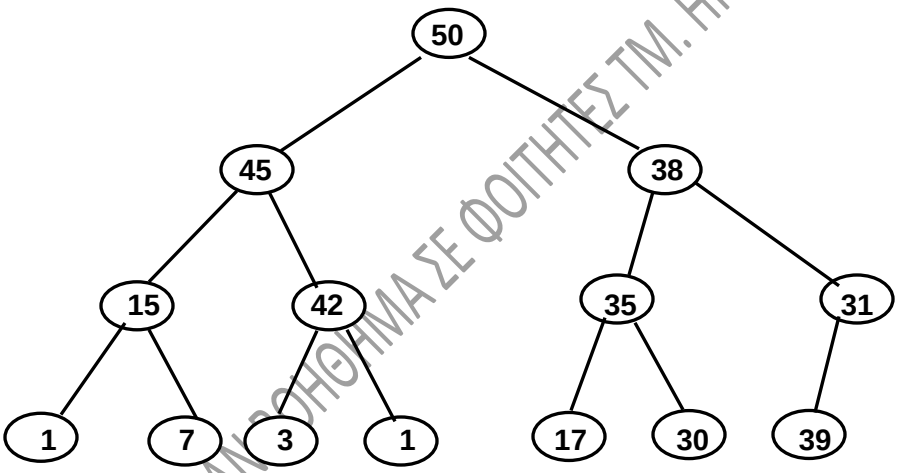
Προκειμένου να εισαγάγουμε ένα κόμβο σε ένα σωρό μεγίστων, ακολουθούμε τα εξής βήματα (αλγόριθμος εισαγωγής):

α) **Τοποθετούμε το νέο κόμβο στο χαμηλότερο επίπεδο**, στην πρώτη κενή θέση (θυμηθείτε ότι κάθε επίπεδο γεμίζει από αριστερά προς τα δεξιά).

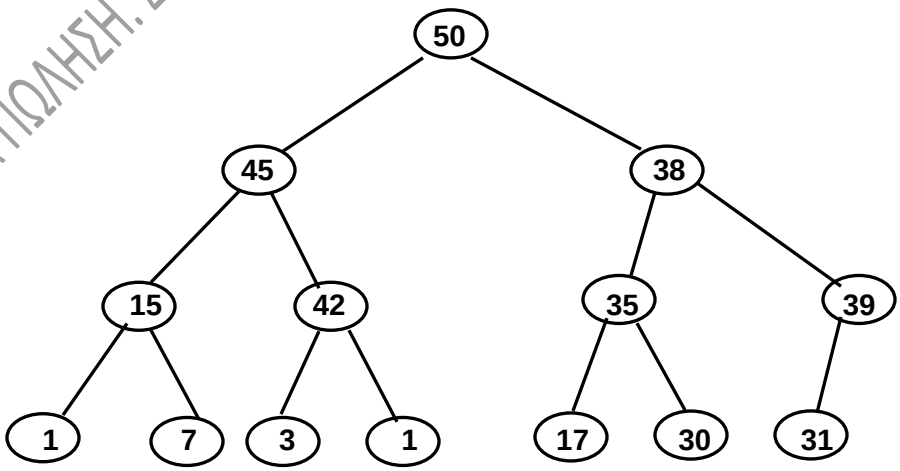
β) **Συγκρίνουμε το νεοεισαχθέν στοιχείο με τον γονέα του**. Εάν το νεοεισαχθέν βρίσκεται στην σωστή θέση, τότε η εισαγωγή τερματίζεται.

γ) **Εάν το νεοεισαχθέν στοιχείο δεν βρίσκεται στην σωστή θέση, τότε το εναλλάσσουμε με τον γονέα του** και επιστρέφουμε στο παραπάνω βήμα (β).

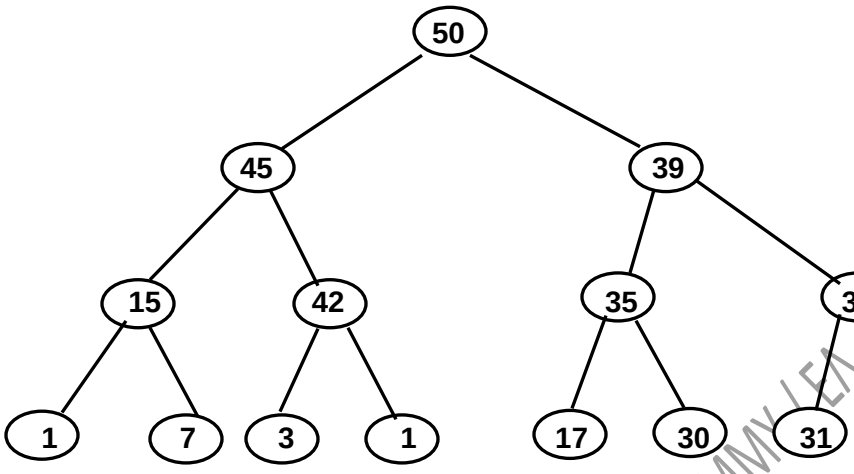
Στα σχ. 8.49α, β και γ φαίνεται η εισαγωγή ενός κόμβου με τιμή 39 για το δέντρο του σχ. 8.48.




(α)




(β)



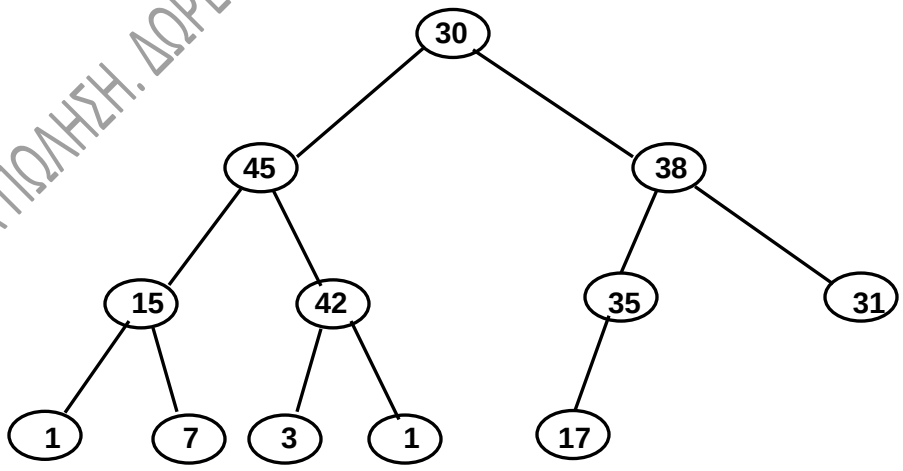

(γ)

Σχ. 8.49

**8.12.2. Διαγραφή της ρίζας του σωρού:**

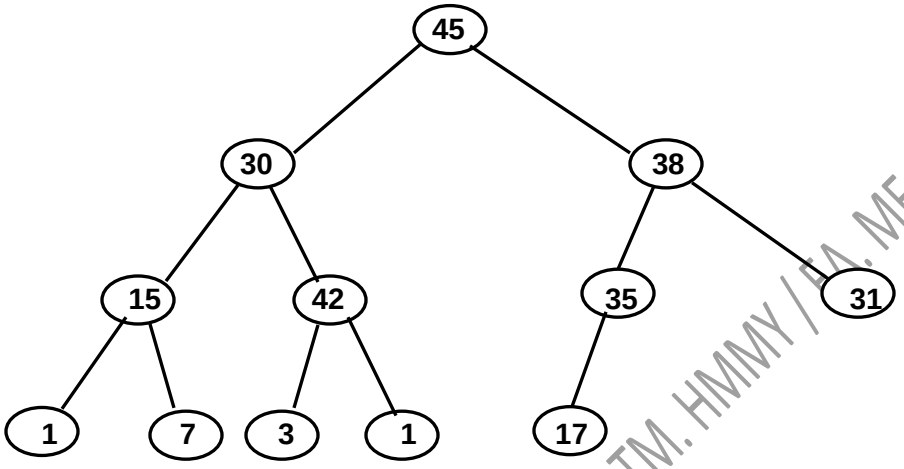
Για να γίνει σωστά η διαγραφή, πρέπει και πάλι να διατηρούνται οι ιδιότητες του σωρού, δηλαδή το δέντρο να παραμένει συμπληρωμένο προς τα αριστερά κάθε φορά. Επίσης να ισχύει η σχέση τιμών γονέα προς παιδιά σε κάθε κόμβο

**Προκειμένου να διαγράψουμε την ρίζα από τον σωρό, αντικαθιστούμε την ρίζα με τον τελευταίο κόμβο του τελευταίου επιπέδου**, για το δέντρο λοιπόν του σχ. 8.48 αντικαθιστούμε την ρίζα με το 30, οπότε θα προκύψει το παρακάτω δέντρο:



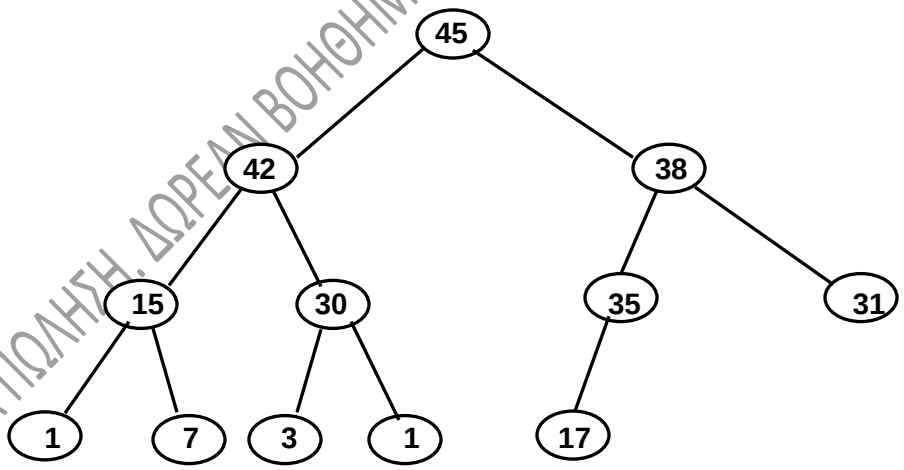

Σχ. 8.50α

Εάν το δέντρο που προέκυψε **παραβιάζει την ιδιότητα να είναι τα παιδιά μικρότερα ή ίσα με τους γονείς, αντικαθιστούμε τον κόμβο** που παραβιάζει την ιδιότητα του σωρού **με το μεγαλύτερο από τα παιδιά του.** Παίρνουμε λοιπόν το επόμενο δέντρο:




Σχ. 8.50β

Το αριστερό υποδέντρο του 45 εξακολουθεί να παραβιάζει την ιδιότητα 2 του σωρού. Έτσι, αντικαθιστούμε το 30 με το μεγαλύτερο από τα παιδιά του:




Σχ. 8.50γ

Έχουμε καταλήξει και πάλι σε σωρό, άρα η διαδικασία διαγραφής τελείωσε.

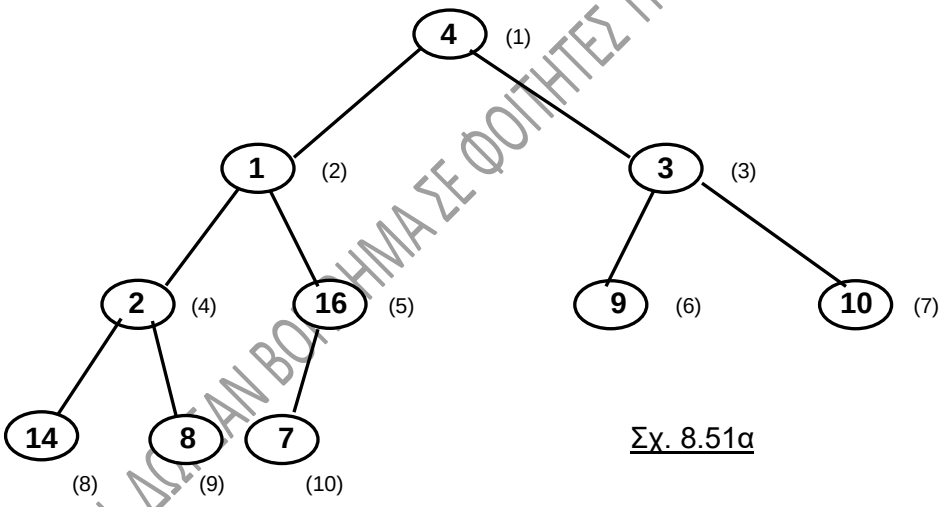
### 8.12.3. Δημιουργία σωρού μεγίστων:

Ένας σωρός μπορεί να δημιουργηθεί με συνεχείς εισαγωγές στοιχείων. Πάντως, αυτή δεν είναι η καλύτερη μέθοδος δημιουργίας του. Η καλύτερη μέθοδος αρχίζει με **αυθαίρετη τοποθέτηση των στοιχείων σε ένα δυαδικό δέντρο**, το οποίο το γεμίζουμε ανά επίπεδο από αριστερά προς τα δεξιά. Στη συνέχεια, **ξεκινώντας από το χαμηλότερο επίπεδο μετακινούμε τα στοιχεία ώστε να πάρουν τις σωστές τους θέσεις**, σύμφωνα με τις ιδιότητες του σωρού. Όταν διαπιστώνεται ότι κάποιος κόμβος πρέπει να αλλάξει θέση, αυτός **εναλλάσσεται με το μεγαλύτερο από τα παιδιά του**.

Έστω για παράδειγμα ότι θέλουμε να δημιουργήσουμε ένα σωρό για να τοποθετηθούν σε αυτόν τα στοιχεία:

4 1 3 2 16 9 10 14 8 7

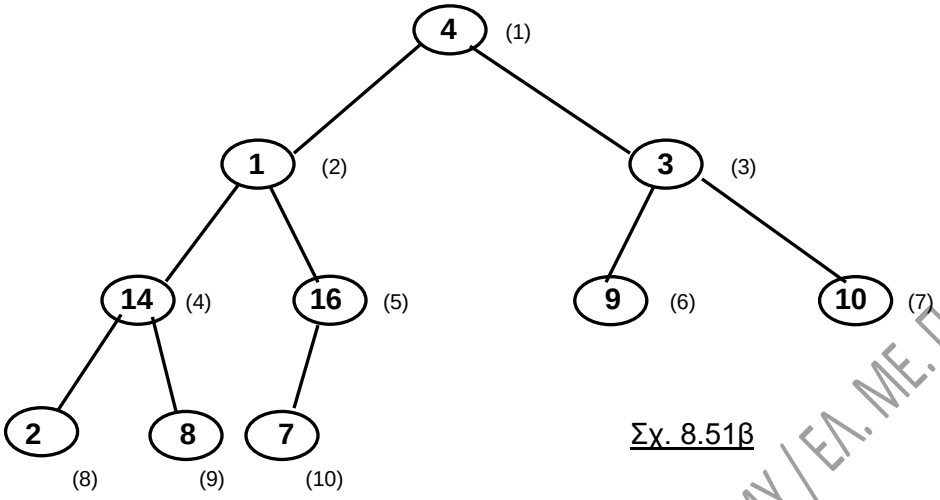
Δημιουργούμε το παρακάτω δυαδικό δέντρο:




Σχ. 8.51α

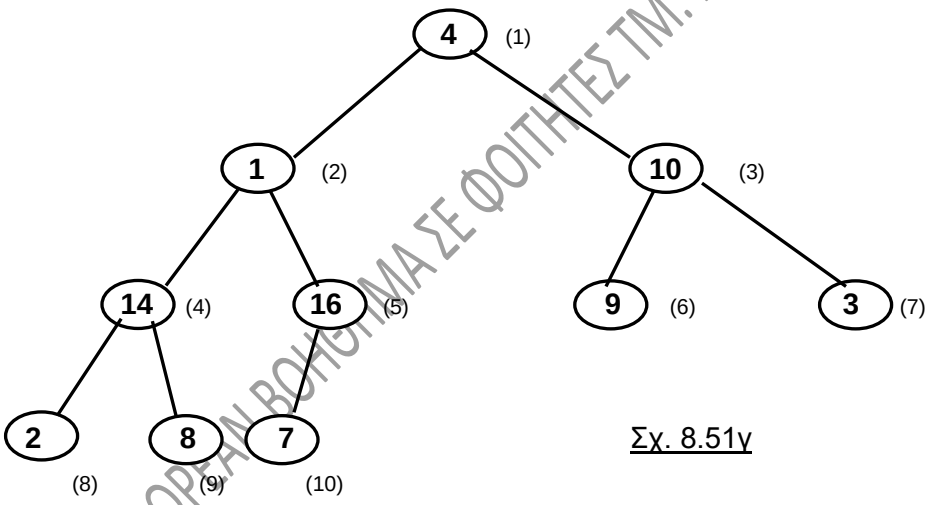
Αν αριθμήσουμε τους κόμβους ανά επίπεδο, προκύπτουν οι αριθμοί στις παρενθέσεις δίπλα σε κάθε κόμβο. Τα στοιχεία από N/2+1 έως N είναι φύλλα (N είναι το πλήθος των στοιχείων του σωρού και με N/2 εννοούμε ακέραια διαίρεση). Ξεκινούμε τον έλεγχο από το στοιχείο στην θέση N/2 (5 στο παράδειγμά μας) και συνεχίζουμε διαδοχικά τους ελέγχους κάθε κόμβου σε σχέση με τα παιδιά του για το κατά πόσον πληρούνται οι ιδιότητες του σωρού ή όχι. Συνεχίζουμε τον έλεγχο για όλους τους κόμβους μέχρι την θέση 1. Έτσι:

- **Ο κόμβος 5** δεν χρειάζεται αλλαγή.
- **Ο κόμβος 4** έχει τιμή 2 και εναλλάσσεται με το 14:



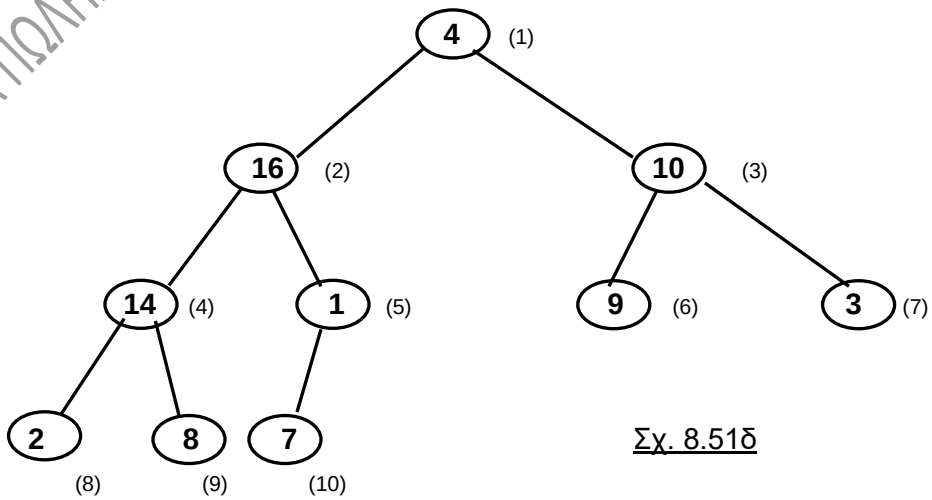

Σχ. 8.51β

- **Ο κόμβος 3** έχει τιμή 3 και εναλλάσσεται με το 10:



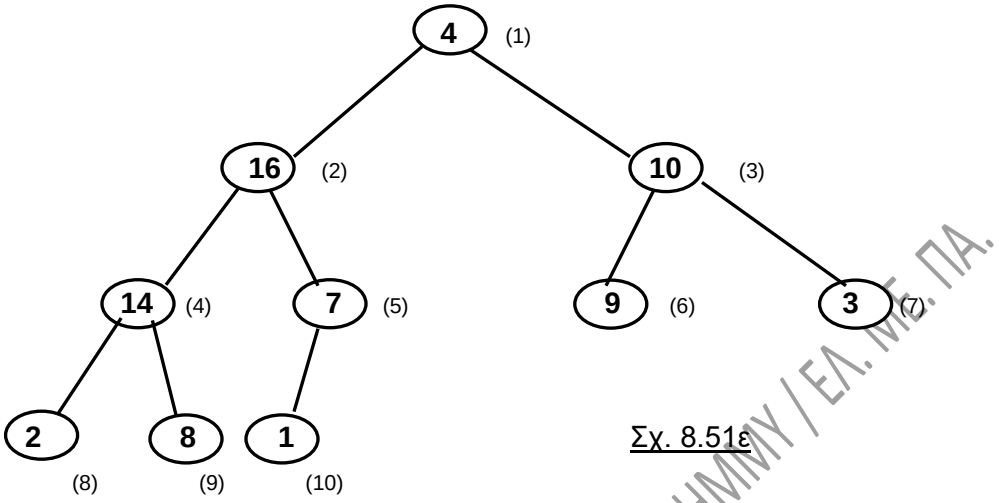

Σχ. 8.51γ

- **Ο κόμβος 2** έχει τιμή 1 και εναλλάσσεται με το 16:



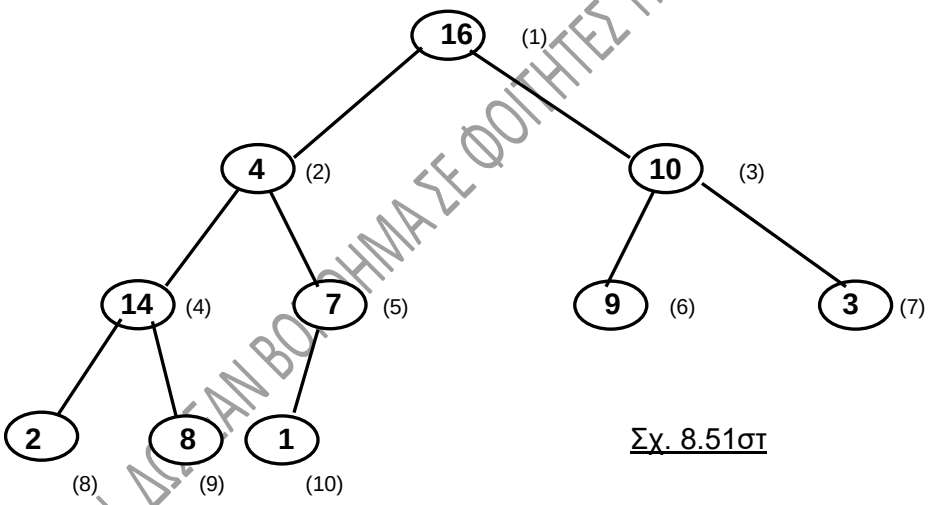

Σχ. 8.51δ

- Το 1 «κατεβαίνει» μέχρι να πάρει την θέση του. Έτσι, **ο κόμβος 5** έχει τιμή 1 και εναλλάσσεται με το 7:



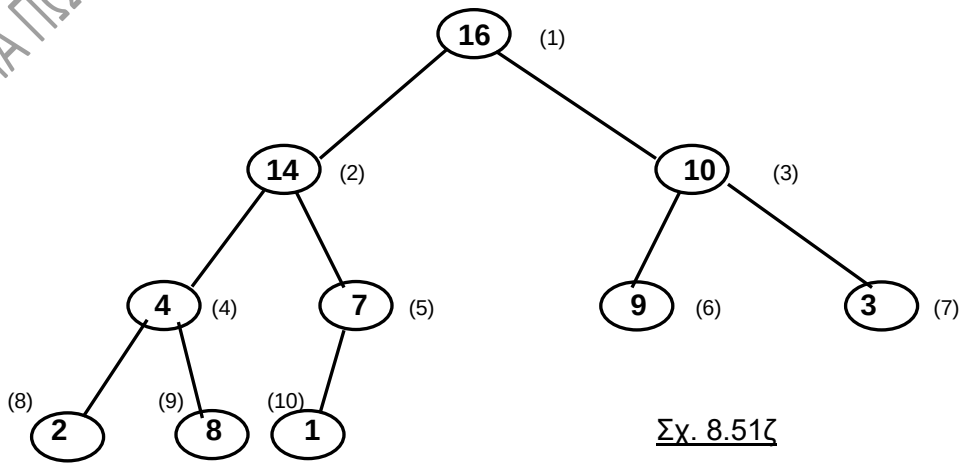

Σχ. 8.51ε

- **Ο κόμβος 1** έχει τιμή 4 και εναλλάσσεται με το 16:



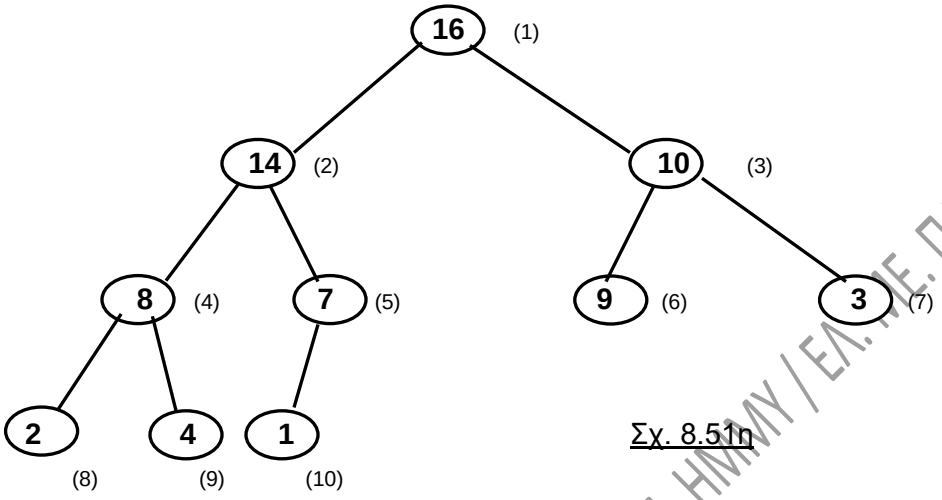

Σχ. 8.51στ

- Το 4 «κατεβαίνει» μέχρι να πάρει την θέση του. Έτσι, **ο κόμβος 4** έχει τιμή 14 και εναλλάσσεται με το 4:




Σχ. 8.51ζ

- Τέλος, η τιμή 4 του **κόμβου 4** εναλλάσσεται με το 8:




Σχ. 8.51η

Με τα βήματα αυτά έχει ολοκληρωθεί η δημιουργία του σωρού.

### 8.12.4. Άλλες λειτουργίες σε σωρό μεγίστων:

Αναφέρουμε επιγραμματικά κάποιες λειτουργίες, οι οποίες μπορούν να υλοποιηθούν με την χρήση σωρού μεγίστων (αντίστοιχα και σωρού ελαχίστων):

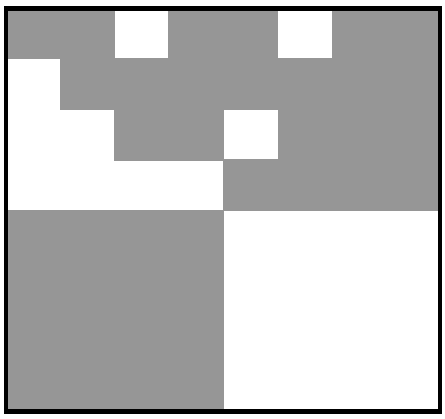
- **Ταξινόμηση:** μπορεί να γίνει με εισαγωγή στοιχείων στον σωρό και στη συνέχεια διαγραφή διαδοχικά της ρίζας του σωρού μέχρι να εξαντληθούν όλα τα στοιχεία.
- **Ένωση δύο σωρών.**
- **Υλοποίηση ουράς προτεραιότητας:** Προκύπτει άμεσα από την λογική και τον τρόπο δημιουργίας του σωρού.



## 8.13. ΤΕΤΡΑΔΙΚΑ ΔΕΝΤΡΑ.

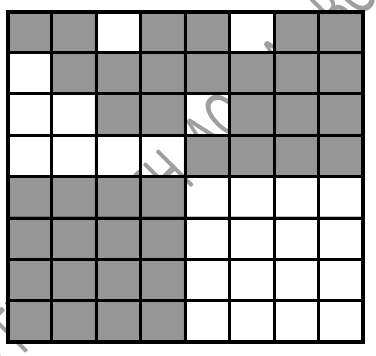
Εκτός από τα δυαδικά δέντρα υπάρχουν και **δέντρα με περισσότερα από δύο παιδιά ανά κόμβο**. Μια ειδική κατηγορία τέτοιων δέντρων είναι τα **τετραδικά**, τα οποία έχουν ευρεία χρήση σε εφαρμογές γραφικών και επεξεργασίας εικόνας. Στις εφαρμογές αυτές χρειαζόμαστε συχνά την αποθήκευση και επεξεργασία σχημάτων, εικόνων κλπ.

Ας θεωρήσουμε ότι έχουμε την εικόνα του σχ. 8.52. Η εικόνα αυτή είναι για χάρη απλότητας, ασπρόμαυρη:



Σχ. 8.52

Σε τέτοιες εφαρμογές η εικόνα χωρίζεται σε ένα μεγάλο αριθμό μικρών τετραγώνων, τα οποία ονομάζονται **εικονοστοιχεία (pixels)**, **σε καθένα από τα οποία θα αντιστοιχεί μια τιμή**, η οποία συνήθως εκφράζει τη μέση φωτεινότητα του εικονοστοιχείου. Δεδομένου ότι στο παράδειγμά μας αναφερόμαστε σε δυο μόνο χρώματα, αντιστοιχούμε το 1 στο λευκό και το 0 στο μαύρο χρώμα. Μετά τον χωρισμό σε εικονοστοιχεία, η εικόνα θα παρουσιάζει τη μορφή του σχ. 8.53α. **Η διατήρηση της πληροφορίας** αυτής της εικόνας στη μνήμη του υπολογιστή μπορεί να γίνει με τη μορφή ενός **δισδιάστατου πίνακα**, ο οποίος φαίνεται στο σχ. 8.53β.



Σχ. 8.53α

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Σχ. 8.53β

Το πρόβλημα στην απεικόνιση με πίνακα είναι ότι **απαιτείται μεγάλη ποσότητα μνήμης** για την αποθήκευση έστω και μιας μόνο εικόνας. Πράγματι, ο πίνακας του σχ. 8.53β είναι ένας πίνακας ακεραίων 8x8, δηλαδή 64 ακεραίων, πράγμα που σημαίνει ότι χρειάζονται 128 byte για αποθήκευση, σύμφωνα με τις απαιτήσεις της γλώσσας C. Για μια όχι ιδιαίτερα καλή ανάλυση εικόνας 800x600 εικονοστοιχείων ο αντίστοιχος

υπολογισμός δίνει ένα πίνακα 480000 ακεραίων, δηλαδή 960 KB. Κι αυτό για μια μόνο εικόνα.

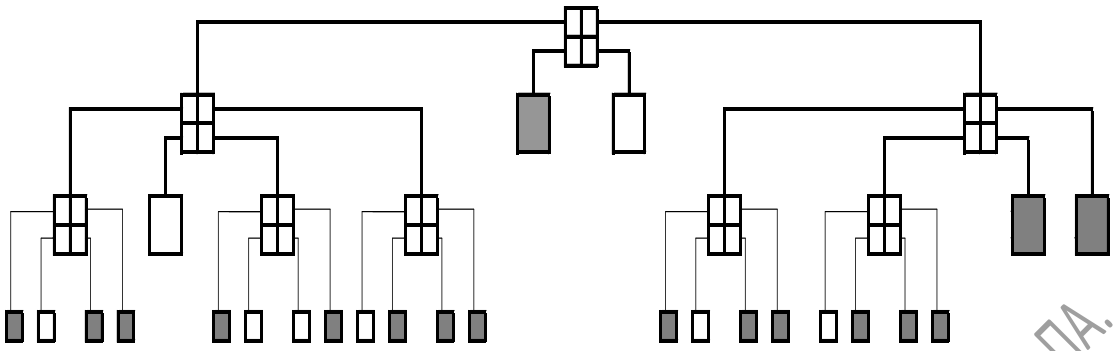
Η τεχνική που εφαρμόζεται για την αντιμετώπιση του προβλήματος είναι η εξής: **Ο πίνακας χωρίζεται σε τέσσερεις τετραγωνικούς υποπίνακες**. Όποιος από τους υποπίνακες **δεν έχει ολόκληρος το ίδιο περιεχόμενο** (εάν δηλαδή περιέχει και 0 και 1), **χωρίζεται και πάλι σε τέσσερεις υποπίνακες** (αναφέρονται συνήθως ως βόρειο-δυτικός ή **NW**, βόρειο-ανατολικός ή **NE**, νότιο-δυτικός ή **SW** και νότιο-ανατολικός ή **SE**) και η διαδικασία επαναλαμβάνεται **μέχρι να καταλήξουμε σε μεμονωμένο εικονοστοιχείο**. Οι «μονόχρωμοι», οι αμιγείς δηλαδή υποπίνακες δεν χωρίζονται παραπέρα. Οι έντονες γραμμές του σχ. 8.54 δείχνουν τον τρόπο χωρισμού του πίνακα σύμφωνα με την παραπάνω λογική:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Σχ. 8.54

Ο πίνακας του σχ. 8.54 μπορεί να παρασταθεί από ένα **τετραδικό δέντρο**. Κάθε κόμβος του δέντρου αυτού αντιστοιχεί σε ένα τετραγωνικό υποπίνακα. Αν όλα τα εικονοστοιχεία του υποπίνακα έχουν την ίδια τιμή, τότε ο κόμβος είναι φύλλο του δέντρου, αλλιώς ο πίνακας χωρίζεται σε τέσσερεις υποπίνακες, όπως στο σχ. 8.55. Τώρα είναι αρκετός **ένας ακέραιος για την αποθήκευση της πληροφορίας** όχι ενός μόνο εικονοστοιχείου, αλλά **ενός ολόκληρου ομοιόχρωμου υποπίνακα**, ενός δηλαδή φύλλου του δέντρου.

Παρατηρείστε ότι το «μοίρασμα» όλων των κόμβων γίνεται με την ίδια σειρά, γεγονός που επιβάλλεται κυρίως από την αναδρομικότητα των χρησιμοποιούμενων συναρτήσεων.

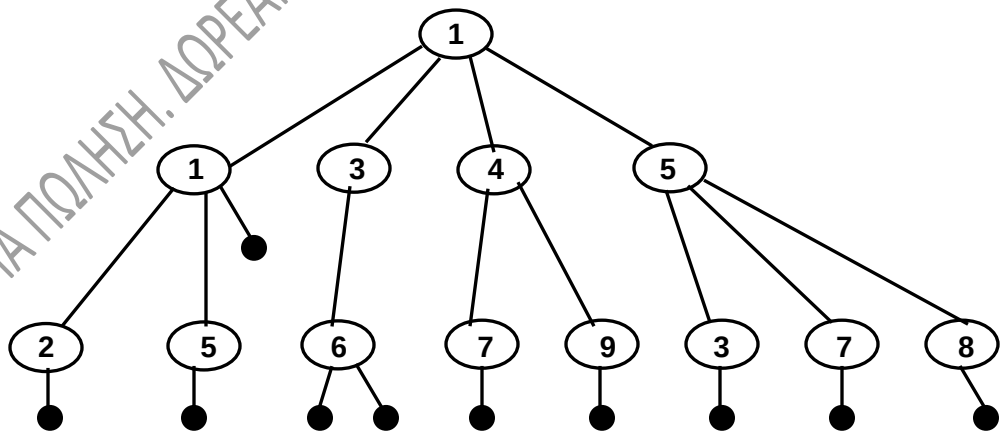


Σχ. 8.55

## 8.14. ΨΗΦΙΑΚΑ ΔΕΝΤΡΑ.

### 8.14.1. Γενικά.

Τα ψηφιακά δέντρα έχουν την ιδιαιτερότητα ότι **οι κόμβοι τους έχουν τιμές που ανήκουν σε ένα συγκεκριμένο σύνολο τιμών**, π.χ. τα γράμματα της αλφαβήτου, οι αριθμοί 0 έως 9 κλπ. **Η πληροφορία** η οποία μεταφέρεται στα δέντρα αυτά **σχηματίζεται από την διαδρομή που ακολουθούμε αν διατρέξουμε το δέντρο ξεκινώντας από την ρίζα μέχρι κάποιο φύλλο του**. Έτσι, το ψηφιακό δέντρο για τους αριθμούς 112, 115, 136, 147, 149, 153, 157 και 158 είναι αυτό του σχ. 8.56.

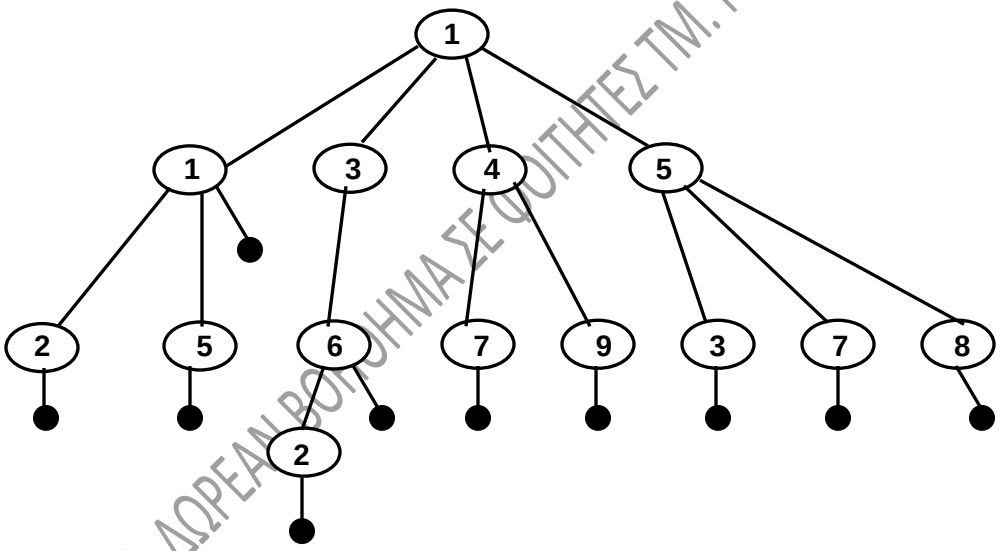



Σχ. 8.56

Η μαύρη κουκίδα που τοποθετείται στο δέντρο παίζει σε κάθε περίπτωση τον ρόλο **συμβόλου τερματισμού** της αποθηκευμένης πληροφορίας. Έτσι, ο αριθμός 112 μπορεί

να υπάρξει στο δέντρο, αφού στο 2 έχουμε σύμβολο τερματισμού. Το ίδιο συμβαίνει για το 11, καθώς και για το 115. Αντίθετα το 13 δεν είναι αποθηκευμένο στο δέντρο, αφού στο 3 δεν υπάρχει σύμβολο τερματισμού, ενώ το 136 είναι αποθηκευμένο στο δέντρο. Από τα παραπάνω προκύπτει ότι, **η δομή** που περιγράφει τον κάθε κόμβο θα **πρέπει να έχει ένα ακόμη πεδίο με την πληροφορία τερματισμού**. Στην πράξη, η μαύρη κουκίδα, στην οποία αναφερθήκαμε, είναι συχνά ένας δείκτης προς ένα αρχείο δίσκου, όπου φυλάσσονται πληροφορίες για το στοιχείο που προσπελάσαμε (για παράδειγμα ένας απλός κωδικός).

Ένα πολύ σημαντικό πλεονέκτημα των δέντρων αυτών είναι ότι **μπορούν να αποθηκεύσουν πληροφορίες μεταβλητού μήκους**. Το σχ. 8.57 δείχνει πώς έχει τροποποιηθεί το πιο πάνω δέντρο, προκειμένου να συμπεριλάβει και τον αριθμό 1362.



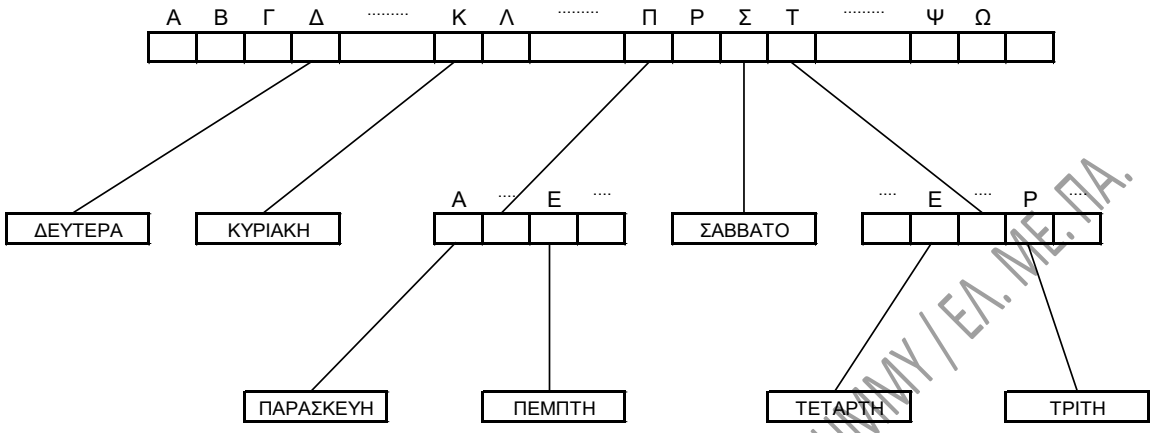

Σχ. 8.57



## 8.14.2. Δομή trie.

Η δομή trie είναι μια **υλοποίηση των ψηφιακών δέντρων για αλφαβητικά κυρίως δεδομένα** και χρησιμοποιείται πάρα πολύ σε λεξικά, ελεγκτές ορθογραφίας κλπ. Η λέξη προέρχεται από την λέξη re-**trie**-val που σημαίνει ανάκτηση, προφέρεται όμως «τράι». Στις δομές αυτές, τα φύλλα του δέντρου αποτελούν τους κόμβους πληροφορίας, ενώ οι εσωτερικοί κόμβοι είναι απλώς κλαδιά του δέντρου. **Στους κόμβους των δέντρων trie υπάρχουν μόνο δείκτες προς συγκεκριμένες θέσεις** και η πληροφορία εξάγεται έμμεσα από την θέση στην οποία δείχνει ο δείκτης.

Στο σχ. 8.58 παρουσιάζεται μια δομή trie, στην οποία οι κόμβοι πληροφορίας είναι τα ονόματα των ημερών της εβδομάδας:




Σχ. 8.58

Επειδή η καταχωρούμενη πληροφορία είναι αλφαβητική, **κάθε κόμβος κλαδί περιέχει ένα πίνακα 25 δεικτών**, όσα δηλαδή τα γράμματα του αλφαβήτου (προκειμένου για το ελληνικό αλφάβητο) και το κενό, ενώ οι δείκτες που δεν εμφανίζονται στο σχήμα αυτό (π.χ. οι δείκτες από τις θέσεις Α, Β, Γ κλπ της ρίζας του δέντρου) έχουν τιμή NULL. Παρατηρείστε πώς για παράδειγμα από τον δείκτη της θέσης Π του πρώτου κόμβου (ρίζας) οδηγούμαστε στον δείκτη της θέσης Ε του δεύτερου και από εκεί -αναγκαστικά και μόνο- στην λέξη ΠΕΜΠΤΗ.

Είναι φανερό ότι **αν οι καταχωρημένες «τιμές»** (οι μέρες της εβδομάδας εν προκειμένω) **είναι λίγες και αραιές**, τότε υπάρχει μεγάλη σπατάλη χώρου και έτσι **η δομή αυτή είναι μη συμφέρουσα**. Φυσικά, αν κάποιες τιμές είναι δεδομένο ότι δεν υπάρχουν, τότε, ανάλογα με τον αλγόριθμο υλοποίησης του δέντρου, είναι πιθανόν να μην υπάρχουν και οι αντίστοιχοι κόμβοι.

Δίνουμε στη συνέχεια ενδεικτικά κώδικα κάποιων συναρτήσεων για χειρισμό μιας δομής trie. Η περιγραφή της δομής για καθε κόμβο ενός δέντρου trie είναι:

```
struct trie {
    int leaf;
    char name[20];
    struct trie *pin[26]; }TRIE;
```

Το πεδίο leaf θα τίθεται 1 ή 0 ανάλογα με το εάν ο κόμβος είναι φύλλο του δέντρου ή εσωτερικός κόμβος. Στο πεδίο name θα φυλάσσεται η τιμή που υπάρχει στο φύλλο, αλλιώς θα περιέχει μια κενή συμβολοσειρά. Το πεδίο pin θα είναι ένας πίνακας δεικτών

σε δομές του είδους Trie. Καθένας από αυτούς τους δείκτες θα δείχνει σε ένα από τους επόμενους εσωτερικούς κόμβους του δέντρου.

Η συνάρτηση που ακολουθεί δημιουργεί ένα κόμβο του δέντρου, κάνει όλους τους δείκτες του κόμβου αυτού ίσους με NULL και επιστρέφει ένα δείκτη στον κόμβο που δημιουργήθηκε:

```
TRIE* new_node ( ) {
    TRIE* node;
    int k;
    node = (TRIE *) malloc (sizeof (TRIE));
    node -> leaf = 0;
    for (k = 0; k< SIZE; k++)
        node -> pin[k] = NULL;
    return node; }
```

Η παρακάτω συνάρτηση εισάγει μια συμβολοσειρά, την str, αποτελούμενη από μικρούς λατινικούς χαρακτήρες στο δέντρο trie. Η συμβολοσειρά δίδεται ως παράμετρος στην συνάρτηση :

```
void insert (TRIE *head, char* str) {
    TRIE * curr = head;
    while (*str) {
        if (curr -> pin[*str - 'a'] == NULL)
            curr->pin[*str - 'a'] = new_node( );
        curr = curr -> pin[*str - 'a'];
        str++; }
    curr -> leaf = 1;
    strcpy (curr -> name, str); }
```

# ΚΕΦΑΛΑΙΟ 9

# ΚΑΤΑΚΕΡΜΑΤΙΣΜΟΣ

## 9.1.ΓΕΝΙΚΑ.

Ο **κατακερματισμός** (**hashing**) είναι μια λειτουργία, η οποία χρησιμοποιεί μια ειδική συνάρτηση, την λεγόμενη **συνάρτηση κατακερματισμού** (**hash function**), έτσι ώστε να αντιστοιχίζει μια τιμή (θα την λέμε **«κλειδί»** στη συνέχεια) σε μια θέση ενός πίνακα, κάνοντας με τον τρόπο αυτό ταχύτερη την αναζήτηση στοιχείων. Η αποδοτικότητα της αντιστοίχισης εξαρτάται από την συνάρτηση που χρησιμοποιείται.

Ένας **πίνακας κατακερματισμού** αποτελείται από ένα σύνολο από θέσεις, σε κάθε μια από τις οποίες αποθηκεύεται ένα κλειδί, αφού πρώτα περάσει από την συνάρτηση κατακερματισμού. Αν το κλειδί είναι αριθμητικό, η συνάρτηση είναι συνήθως μια απλή παράσταση. Αν το κλειδί είναι αλφαριθμητικό, τότε με κάποιο τρόπο πρέπει να μετατραπεί σε αριθμητικό (π.χ. με την χρήση του κώδικα ASCII).



## 9.2. ΣΥΝΑΡΤΗΣΕΙΣ ΚΑΤΑΚΕΡΜΑΤΙΣΜΟΥ.

Δύο κύρια ερωτήματα που τίθενται και αφορούν τις συναρτήσεις κατακερματισμού είναι:

α) **Ποια είναι η καλύτερη επιλογή** για την συνάρτηση και

β) **Τι θα γίνει εάν πολλά κλειδιά αριθμοί αποθηκεύονται στην ίδια θέση του πίνακα κατακερματισμού**, κάτι που είναι πολύ πιθανό, οπότε μιλούμε για συγκρούσεις (collisions).

Μια καλή συνάρτηση κατακερματισμού θα πρέπει:

- Να χρησιμοποιεί όλο τον πίνακα κατακερματισμού
- Να κατανέμει ομοιόμορφα τις τιμές στον πίνακα και
- Να είναι απλή ως προς τον υπολογισμό της.

Αναφέρουμε τώρα κάποια παραδείγματα συχνά χρησιμοποιούμενων συναρτήσεων κατακερματισμού.

### 9.2.1. Διαίρεση με πρώτο αριθμό.

Η θέση του πίνακα όπου θα αποθηκευτεί το κλειδί προκύπτει από **το υπόλοιπο της ακέραιας διαίρεσης του κλειδιού (modulo) με το μέγεθος του πίνακα**. Προφανώς μπορεί να ληφθεί το υπόλοιπο της διαίρεσης του κλειδιού με άλλο ακέραιο, αρκεί να είναι μικρότερος από το μέγεθος του πίνακα. Είναι συνηθισμένο (και όπως προκύπτει από παρατηρήσεις, αρκετά αποδοτικό), **ο διαιρέτης να είναι ο μεγαλύτερος πρώτος αριθμός, ο οποίος είναι μικρότερος από το μέγεθος του πίνακα**.

Έστω για παράδειγμα ένα σύνολο τιμών S = {5, 14, 25, 44, 68, 99, 150, 373, 401, 509}, τις οποίες θα αποθηκεύσουμε σε ένα πίνακα κατακερματισμού Η, 7 θέσεων. Εφαρμόζουμε σε κάθε κλειδί x (σε κάθε τιμή του S δηλαδή) την συνάρτηση κατακερματισμού x % 7. Έτσι προκύπτει ο πίνακας κατακερματισμού του σχ. 9.1:

| | | | |
|---|---|---|---|
| 0 | 14 | | |
| 1 | 99 | | |
| 2 | 44 | 373 | 401 |
| 3 | 150 | | |
| 4 | 25 | | |
| 5 | 5 | 68 | 509 |
| 6 | | | |

Σχ. 9.1.

Έστω, όπως προαναφέρθηκε, ότι **τα κλειδιά δεν είναι ακέραιοι, αλλά συμβολοσειρές**. Τότε, μια συνάρτηση κατακερματισμού θα μπορούσε να είναι η πιο κάτω, η οποία θα δέχεται ως είσοδο την συμβολοσειρά arr και θα **επιστρέφει ένα ακέραιο, ο οποίος θα προκύπτει από το υπόλοιπο της ακέραιας διαίρεσης του μήκους της συμβολοσειράς με το μέγεθος του πίνακα κατακερματισμού**:

```
int hash (char *arr, int size) {
    int ak;
    ak = strlen (arr);
    return ak % size;
}
```

**Με την ίδια λογική γίνεται και η αναζήτηση** ενός στοιχείου στον πίνακα Η. Αν για παράδειγμα αναζητούμε τον αριθμό 68, πρέπει να ψάξουμε στην θέση 5, το στοιχείο δηλαδή Η[5] (το 68 % 7).

### 9.2.2. Μετατροπή ρίζας (radix conversion).

**Εάν το κλειδί δεν είναι αριθμός του δεκαδικού αριθμητικού συστήματος**, τότε **μετατρέπεται σε δεκαδικό αριθμό και το τελευταίο ψηφίο δηλώνει την θέση αποθήκευσης** στον πίνακα. Μπορεί επίσης **η θέση αποθήκευσης να προκύψει από την πράξη modulo του κλειδιού με το μέγεθος του πίνακα**.

Δίνουμε ένα σχετικά ρεαλιστικό παράδειγμα, θεωρώντας ότι τα κλειδιά ανήκουν στο δεκαεξαδικό σύστημα και η αποθήκευση γίνεται σε ένα πίνακα 101 θέσεων. Κάθε κλειδί αποθηκεύεται στην θέση που καθορίζουν τα δύο τελευταία ψηφία του αριθμού. Παρακάτω εμφανίζονται τα κλειδιά και οι θέσεις αποθήκευσης:

| Κλειδί | Δεκαδικός | Θέση |
|---|---|---|
| 13F8 | 6648 | 48 |
| 273A | 10042 | 42 |
| 5414 | 21524 | 24 |
| 3245 | 12869 | 69 |
| 105B | 4187 | 87 |
| A03C | 41020 | 20 |
| 400F | 16399 | 99 |
| 06FD | 1789 | 89 |

Σχ. 9.2.

### 9.2.3. Αναδίπλωση (folding).

**Το κλειδί χωρίζεται σε δύο μέρη, τα οποία προστίθενται μεταξύ τους**. Η θέση αποθήκευσης μπορεί για παράδειγμα να προκύψει από τα δύο τελευταία ψηφία του αριθμού, όπως επίσης μπορεί να προκύψει από την πράξη modulo του κλειδιού με το μέγεθος του πίνακα. Δείτε το παράδειγμα του σχ. 9.3 για ένα πίνακα κατακερματισμού 101 θέσεων και αναδίπλωση μετά το τρίτο στοιχείο του κλειδιού. Το πρώτο κλειδί χωρίζεται σε δυο μέρη, δηλαδή τα 563 και 49. Τα προσθέτουμε μεταξύ τους και παίρνουμε άθροισμα 612. Κάνοντας μετά την πράξη modulo μεταξύ του 612 και του μεγέθους του πίνακα (δηλαδή του 101) παίρνουμε αποτέλεσμα 12, η οποία είναι και η

θέση αποθήκευσης του πρώτου κλειδιού στον πίνακα κατακερματισμού. Η ίδια λογική εφαρμόζεται και για τα υπόλοιπα κλειδιά.

| Κλειδί | Θέση |
|---|---|
| 56349 | 12 |
| 10042 | 42 |
| 21524 | 39 |
| 12869 | 97 |
| 23566 | 01 |
| 41020 | 30 |
| 16399 | 62 |
| 36445 | 09 |

Σχ. 9.3.

Σε μια παραλλαγή της μεθόδου **προστίθεται το πρώτο τμήμα με το δεύτερο αφού αναστραφούν τα ψηφία του**. Για το πρώτο κλειδί δηλαδή του σχ. 9.3 θα προσθέσουμε το 563 με το 94. Το αποτέλεσμα δίνει 657. Θα αποθηκεύσουμε λοιπόν το κλειδί στην θέση 57%101, άρα στην θέση 57 του πίνακα κατακερματισμού.

### 9.2.4. Τετράγωνο του μέσου (mid square).

Σε αυτού του είδους την συνάρτηση κατακερματισμού **παίρνουμε τα μεσαία ψηφία του κλειδιού, τον αριθμό που προκύπτει τον υψώνουμε στο τετράγωνο και παίρνουμε τα μεσαία ψηφία αυτού του αριθμού**.

Στο παράδειγμα του σχ. 9.4 που ακολουθεί λαμβάνουμε τα τρία μεσαία ψηφία του κλειδιού και τα δύο μεσαία ψηφία του τετραγώνου. Για το πρώτο κλειδί για παράδειγμα τα τρία μεσαία ψηφία είναι 634, οπότε $634^2$ = 401956 και τα μεσαία ψηφία αυτού του αριθμού είναι το 19. Θα αποθηκεύσουμε λοιπόν το κλειδί στην θέση 19%101, άρα στην θέση 19 του πίνακα κατακερματισμού (για ένα πίνακα κατακερματισμού 101 θέσεων).

| Κλειδί | Τετράγωνο | Θέση |
|---|---|---|
| 56349 | 401956 | 19 |
| 10042 | 000016 | 00 |
| 21524 | 023104 | 31 |
| 12869 | 081796 | 17 |
| 23566 | 126736 | 67 |
| 41020 | 010404 | 04 |
| 16399 | 408321 | 83 |
| 36445 | 414736 | 47 |

Σχ. 9.4.

## 9.3. ΑΝΤΙΜΕΤΩΠΙΣΗ ΣΥΓΚΡΟΥΣΕΩΝ.

Όπως προαναφέραμε, υπάρχει περίπτωση **περισσότερα από ένα κλειδιά να αποθηκεύονται στην ίδια θέση** του πίνακα κατακερματισμού. Στην περίπτωση αυτή λέμε ότι έχει συμβεί μια **σύγκρουση** (**collision**). Η ύπαρξη συγκρούσεων κάνει δύσκολη τόσο την αποθήκευση στοιχείων, όσο και την αναζήτησή τους αργότερα. Τέτοιες καταστάσεις προσπαθούμε να τις αποφύγουμε με διάφορες μεθόδους, κάποιες, από τις οποίες παρουσιάζονται παρακάτω.

### 9.3.1. Αντιμετώπιση συγκρούσεων με αλυσσίδες.

Όπως αναφέρθηκε, περισσότερες από μια τιμές πιθανόν να έχουν το ίδιο αποτέλεσμα εάν εφαρμοστεί σε αυτές η συνάρτηση κατακερματισμού. Μπορούμε τώρα να δημιουργήσουμε τον **πίνακα κατακερματισμού Η ως πίνακα δεικτών**. **Κάθε θέση του θα περιέχει ένα δείκτη, ο οποίος θα δείχνει στην κεφαλή μιας απλά συνδεδεμένης λίστας**.

Για να βρούμε ένα κλειδί, το k, θα ψάξουμε στην λίστα, η οποία δείχνεται από την θέση H[k]. Οι δείκτες του πίνακα που δεν χρησιμοποιούνται θα δείχνουν προφανώς σε NULL. Αντίστοιχα, η εισαγωγή κλειδιού στον πίνακα και η διαγραφή κλειδιού από τον πίνακα, γίνεται με τον ίδιο τρόπο που κάνουμε εισαγωγή και διαγραφή στοιχείου στις απλά συνδεδεμένες λίστες. Προφανώς δεν υπάρχει όριο στο πόσα στοιχεία θα περιέχει κάθε λίστα.

### 9.3.2. Αντιμετώπιση συγκρούσεων με ανοικτή διεύθυνση.

Με την χρήση της συνάρτησης κατακερματισμού υπολογίζουμε την θέση k του πίνακα, όπου θα φυλαχθεί το κλειδί, όπως είπαμε και στα προηγούμενα. Τώρα εφαρμόζουμε τον εξής αλγόριθμο:

- **Αν η θέση k είναι κενή**, τότε **το κλειδί αποθηκεύεται εκεί**.
- **Αν η θέση k δεν είναι κενή**, τότε δοκιμάζουμε μια **θέση, η οποία προκύπτει από μια άλλη συνάρτηση**, η οποία δέχεται ως ορίσματα την τιμή έναρξης και ένα βήμα.

Για να γίνουν κατανοητά τα παραπάνω, δίνουμε δυο παραλλαγές της αντιμετώπισης συγκρούσεων με ανοικτή διεύθυνση, σύμφωνα με όσα είπαμε αμέσως προηγουμένως.

**α) Αντιμετώπιση συγκρούσεων με γραμμική δοκιμή.**

Στην μέθοδο αυτή χρησιμοποιούμε ως **συνάρτηση κατακερματισμού** την **h(k) % m** και ως **δεύτερη συνάρτηση** την **(h(k) + i) % m για 1 <= i <= n-1**, όπου n το μέγεθος του πίνακα και m ένας ακέραιος, μικρότερος ή ίσος προς το μέγεθος του πίνακα. **Όταν συμβεί μια σύγκρουση το i παίρνει τιμή 1, εάν συμβεί ξανά σύγκρουση, το i παίρνει τιμή 2 κ.ο.κ**. Αν σκεφθείτε, αυτό είναι ισοδύναμο με το εξής: **εάν ένα κλειδί συγκρούεται** σε κάποια θέση **με ένα άλλο**, τότε **το κλειδί αυτό τοποθετείται στην επόμενη κενή θέση του πίνακα**.

Έστω ότι θέλουμε για παράδειγμα να κάνουμε εισαγωγή στον πίνακα κατακερματισμού των κλειδιών του συνόλου S = {37, 52, 43, 63, 9, 17, 70, 33, 72} με m=11. Στο σχ. 9.5 φαίνεται πώς θα τοποθετηθούν ένα-ένα τα κλειδιά του S στον πίνακα (σε ποια θέση). Με κίτρινο χρώμα εμφανίζεται η θέση όπουθ θα έπρεπε να τοποθετηθεί το κλειδί, αλλά συμβαίνει σύγκρουση για πρώτη φορά και με πράσινο χρώμα εμφανίζεται η θέση όπου θα έπρεπε κατόπιν να τοποθετηθεί το κλειδί, αλλά συμβαίνει σύγκρουση για δεύτερη φορά:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **37** | | | | | 37 | | | | | | |
| **52** | | | | | 37 | | | | 52 | | |
| **43** | | | | | 37 | | | | 52 | | 43 |
| **63** | | | | | 37 | | | | 52 | 63 | 43 |
| **9** | 9 | | | | 37 | | | | 52 | 63 | 43 |
| **17** | 9 | | | | 37 | | 17 | | 52 | 63 | 43 |
| **70** | 9 | | | | 37 | 70 | 17 | | 52 | 63 | 43 |
| **33** | 9 | 33 | | | 37 | 70 | 17 | | 52 | 63 | 43 |
| **72** | 9 | 33 | | | 37 | 70 | 17 | 72 | 52 | 63 | 43 |

Σχ. 9.5.

**β) Αντιμετώπιση συγκρούσεων με τετραγωνική δοκιμή.**

Χρησιμοποιούμε ως **συνάρτηση κατακερματισμού** την **h(k) % m** και ως **δεύτερη συνάρτηση** την:

$$(h(k) + c_1 * i + c_2 * i^2) \% m$$

όπου m ένας ακέραιος, μικρότερος ή ίσος προς το μέγεθος του πίνακα και τα $c_1$ και $c_2$ είναι σταθερές. Το i ( i=1, 2, ...) στην πράξη δηλώνει τις προσπάθειες για την εύρεση κενής θέσης. Όπως και στην γραμμική δοκιμή, **όταν συμβεί μια σύγκρουση το i παίρνει τιμή 1, εάν συμβεί ξανά σύγκρουση, το i παίρνει τιμή 2 κ.ο.κ**. τετραγώνου του i. Έχει καλύτερη απόδοση από την γραμμική δοκιμή

Εφαρμόζουμε την μέθοδο αυτή για τα ίδια κλειδιά όπως και στο προηγούμενο παράδειγμα, δηλαδή για τα κλειδιά του συνόλου S = {37, 52, 43, 63, 9, 17, 70, 33, 72} με m=11, $c_1$=1 και $c_2$=2. Στο σχ. 9.6 φαίνεται πώς θα τοποθετηθούν ένα-ένα τα κλειδιά του S στον πίνακα. Και πάλι, με κίτρινο χρώμα εμφανίζεται η θέση όπου θα έπρεπε να τοποθετηθεί το κλειδί, αλλά συμβαίνει σύγκρουση για πρώτη φορά και με πράσινο χρώμα εμφανίζεται η θέση όπου θα έπρεπε κατόπιν να τοποθετηθεί το κλειδί, αλλά συμβαίνει σύγκρουση για δεύτερη φορά:

| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **37** | | | | | 37 | | | | | | |
| **52** | | | | | 37 | | | | 52 | | |
| **43** | | | | | 37 | | | | 52 | | 43 |
| **63** | 63 | | | | 37 | | | | 52 | | 43 |
| **9** | 63 | | | | 37 | | | | 52 | 9 | 43 |
| **17** | 63 | | | | 37 | | 17 | | 52 | 9 | 43 |
| **70** | 63 | | | | 37 | | 17 | 70 | 52 | 9 | 43 |
| **33** | 63 | | | 33 | 37 | | 17 | 70 | 52 | 9 | 43 |
| **72** | 63 | | | 33 | 37 | 72 | 17 | 70 | 52 | 9 | 43 |

Σχ. 9.6.

# ΚΕΦΑΛΑΙΟ 10

# ΓΡΑΦΟΙ

## 10.1. ΓΕΝΙΚΑ.

Ο **γράφος** (**graph**) είναι μια **μη γραμμική** δομή δεδομένων. Είναι η πιο γενική δομή, δεδομένου ότι κάθε στοιχείο μπορεί να μην έχει **κανένα** ή να έχει **ένα** ή και **πολλά** προηγούμενα και επόμενα στοιχεία. Για τον λόγο αυτό, δεν μπορεί να υπάρξει καμμιά διάταξη σε ένα γράφο. Η «γενικότητα» που προαναφέραμε, έγκειται στο ότι οι δομές που εξετάστηκαν μέχρι τώρα (λίστες, δένδρα κλπ), μπορούν να θεωρηθούν ως υποπεριπτώσεις ή ειδικές περιπτώσεις των γράφων, το αντικείμενο των οποίων είναι αρκετά μεγάλο. Η θεωρία γράφων αποτελεί στην πράξη ξεχωριστό κλάδο της Πληροφορικής με πάρα πολλές και ποικίλες εφαρμογές.

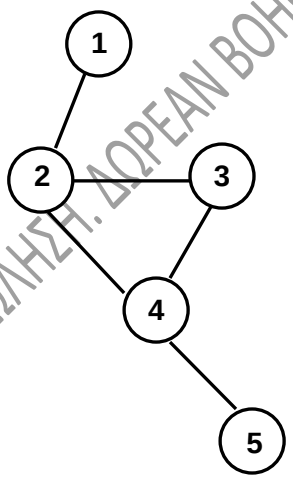
Ένας γράφος G **χαρακτηρίζεται από δυο σύνολα**, τα V και E. Το V είναι ένα πεπερασμένο σύνολο, το $V=\{v_1, v_2,..., v_n\}$, με $n>0$, τα στοιχεία του οποίου είναι οι **κορυφές** (κόμβοι, vertices) του γράφου, οι οποίες χρησιμοποιούνται για την παράσταση των δεδομένων. Αντίστοιχα, το $E=\{(x,y)$, όπου x, y ε V$\}$ είναι το σύνολο των **ακμών** (edges) του γράφου, οι οποίες παριστάνουν τις σχέσεις μεταξύ των δεδομένων. Είναι προφανές λοιπόν ότι **μια ακμή «περιγράφεται» από το ζεύγος των κορυφών τις οποίες συνδέει**.

Ο γράφος μπορεί να συμβολιστεί ως G(V,E). Πρέπει να παρατηρήσουμε ότι **ένας γράφος ορίζεται πλήρως από τα σύνολα V και E**. Αυτό σημαίνει ότι δυο γράφοι μπορεί να είναι εντελώς ταυτόσημοι, παρά το ότι η μορφή τους, η θέση των κορυφών τους και το μήκος των ακμών στη γεωμετρική τους παράσταση μπορεί να διαφέρουν.

## 10.2. ΟΡΟΛΟΓΙΑ.

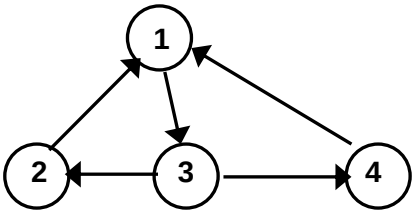
1. Ένας γράφος λέγεται **ζυγισμένος** (weighted graph), όταν **κάθε ακμή του χαρακτηρίζεται από ένα αριθμό, ο οποίος ονομάζεται βάρος** (**weight**). Το βάρος μπορεί να υποδηλώνει διάφορα μεγέθη, όπως για παράδειγμα τον απαιτούμενο χρόνο για την μετάβαση από την μια κορυφή στην άλλη ή την απόσταση δύο κορυφών (για παράδειγμα τις χιλιομετρικές αποστάσεις σε ένα γράφο με κορυφές τις πόλεις ενός νομού).

2. Ένας γράφος λέγεται **μη κατευθυνόμενος** (undirected), αν οι ακμές του δεν είναι προσανατολισμένες προς κάποια κατεύθυνση, αν δηλαδή τα ζεύγη $(v_1, v_2)$ και $(v_2, v_1)$ παριστάνουν την ίδια ακμή (προφανώς αυτό πρέπει να ισχύει για κάθε ζεύγος $(v_i, v_j)$). Αντίθετα, αν τα διάφορα ζεύγη $(v_i, v_j)$ έχουν διάταξη, τότε ο γράφος λέγεται **κατευθυνόμενος**. Το $<v_1, v_2>$ παριστάνει μια κατευθυνόμενη ακμή, στην οποία το $v_1$ αποτελεί την **ουρά** (tail) και το $v_2$ αποτελεί την **κεφαλή** (head) της ακμής. Σχηματικά παριστάνουμε την κατεύθυνση μιας ακμής με ένα βέλος. Ο γράφος $G_1$ του σχ. 10.1 είναι μη κατευθυνόμενος, ενώ κατευθυνόμενος είναι ο γράφος $G_2$. Στο σχήμα φαίνονται επίσης τα σύνολα ακμών και κορυφών για τους δυο γράφους:




**$G_1$**

$V(G_1) = \{1, 2, 3, 4, 5\}$
$E(G_1) = \{(1,2), (2,3), (2,4), (3,4), (4,5)\}$




**$G_2$**

$V(G_2) = \{1, 2, 3, 4\}$

$E(G_2) = \{<2,1>, <1,3>, <3,2>, <3,4>, <4,1>\}$

Σχ. 10.1

3. Αποδεικνύεται ότι **ο μέγιστος αριθμός ακμών** για κάθε μη κατευθυνόμενο γράφο με n κορυφές είναι **n*(n-1)/2**, ενώ ο αντίστοιχος αριθμός για ένα κατευθυνόμενο γράφο είναι **n*(n-1)**. Ένας γράφος λέγεται **πλήρης** (complete) εάν έχει τον μέγιστο αριθμό ακμών.

4. Αν ($v_1$, $v_2$) είναι μια ακμή ενός **μη κατευθυνόμενου γράφου**, τότε οι κορυφές $v_1$ και $v_2$ λέγονται **γειτονικές** (adjacent) και η ακμή ($v_1$, $v_2$) λέγεται **προσκείμενη** στις κορυφές $v_1$ και $v_2$. Στον γράφο $G_1$ του σχ. 10.1 οι κορυφές 1, 3 και 4 είναι γειτονικές της κορυφής 2. Εξ άλλου οι ακμές (1,2), (2,3) και (2,4) είναι προσκείμενες στην 2. Αν <$v_1$, $v_2$> είναι μια ακμή ενός **κατευθυνόμενου γράφου**, τότε η $v_1$ λέγεται **γειτονική προς** την $v_2$ ενώ η $v_2$ λέγεται **γειτονική από** την $v_1$. Στον Στον γράφο $G_2$ του σχ. 10.1 η κορυφή 1 είναι γειτονική προς την 3 και συγχρόνως γειτονική από την 4 και από την 2.

5. **Βαθμός κορυφής** (degree) ενός **μη κατευθυνόμενου γράφου** λέγεται ο αριθμός των ακμών που είναι προσκείμενες της κορυφής. Ο βαθμός της κορυφής 4 στον γράφο $G_1$ του σχ. 10.1 είναι 3. Σε ένα **κατευθυνόμενο γράφο** αναφέρονται ο **βαθμός εισόδου** (in-degree) και ο **βαθμός εξόδου** (out-degree) για τον αριθμό των ακμών που καταλήγουν σε μια κορυφή και τον αριθμό των ακμών που ξεκινούν αντίστοιχα από μια κορυφή. Ο βαθμός εισόδου της κορυφής 3 στον γράφο $G_2$ του σχ. 10.1 είναι 1, ενώ ο βαθμός εξόδου της ίδιας κορυφής είναι 2.

6. Ένα σύνολο κορυφών ενός γράφου λέγεται **μονοπάτι** (path) αν υπάρχουν ακμές που ενώνουν διαδοχικά αυτές τις κορυφές. **Μήκος μονοπατιού** είναι ο αριθμός των ακμών του μονοπατιού. Σε ένα **απλό μονοπάτι** όλες οι κορυφές (εκτός ίσως από την πρώτη και την τελευταία) είναι διαφορετικές. Αν η πρώτη κορυφή ενός απλού μονοπατιού συμπίπτει με την τελευταία, το μονοπάτι λέγεται **κύκλος** (cycle).

7. Ένας γράφος λέγεται **συνδεδεμένος ή συνεκτικός** (connected) όταν για κάθε δυο κορυφές του υπάρχει τουλάχιστον ένα μονοπάτι που να τις συνδέει. Με βάση αυτό τον ορισμό, **ένα δένδρο είναι ένας συνδεδεμένος γράφος που δεν περιέχει κύκλους**.

8. **Απόσταση** (distance) μεταξύ δύο κορυφών είναι το μήκος του συντομότερου μονοπατιού που τις συνδέει.

9. **Γράφος Euler** λέγεται εκείνος ο γράφος, στον οποίο υπάρχει κάποια διαδρομή που περνάει από όλες τις ακμές του ακριβώς μια φορά, ενώ **γράφος Hamilton** λέγεται εκείνος ο γράφος, στον οποίο υπάρχει κάποια διαδρομή που περνάει από όλους τους κόμβους του ακριβώς μια φορά

10. **Υπογράφος** $G_x$ ενός γράφου G είναι ένας γράφος, του οποίου οι κορυφές και οι ακμές υπάρχουν και στον G. Δηλαδή ισχύει ότι το $V(G_x)$ είναι υποσύνολο του V(G) και το $E(G_x)$ είναι υποσύνολο του E(G).

## 10.3. ΤΡΟΠΟΙ ΥΛΟΠΟΙΗΣΗΣ ΓΡΑΦΩΝ.

Οι γράφοι μπορούν να υλοποιηθούν με διάφορους τρόπους στην κύρια μνήμη του υπολογιστή, ανάλογα με τις λειτουργίες που θα εκτελεστούν σε αυτούς. Παρουσιάζουμε στην συνέχεια τους πιο συνηθισμένους τρόπους υλοποίησης.

### 10.3.1. Πίνακας γειτονικών κορυφών.

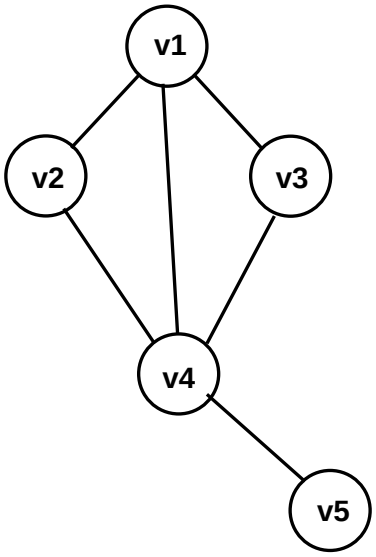
Το μεγαλύτερο **πλεονέκτημα** της παράστασης ενός γράφου με την χρήση πίνακα είναι ότι ο υπολογισμός μονοπατιών, κύκλων κλπ μπορεί να γίνει εύκολα, εφαρμόζοντας γνωστές ιδιότητες των πινάκων. **Μειονέκτημα** αυτής της παράστασης είναι ότι μας απομακρύνει από την οπτική αναπαράσταση των γράφων, η οποία πολλές φορές είναι αναγκαία, αφού διάφορες ιδιότητες που προκύπτουν εύκολα από την παράσταση ενός γράφου, δεν είναι το ίδιο εύκολο να φανούν από την παράσταση με πίνακα.

Για ένα **μη κατευθυνόμενο γράφο** G με n κορυφές ο **πίνακας γειτονικών κορυφών** (**adjacent matrix**) είναι ένας n x n τετραγωνικός πίνακας, έστω ο pin, για τον οποίο ισχύει:

**pin[ j ][k] = 1, αν η κορυφή $v_j$ είναι γειτονική με την κορυφή $v_k$** και

**pin[ j ][k] = 0, αν η κορυφή $v_j$ δεν είναι γειτονική με την κορυφή $v_k$**

Στα σχ. 10.2 (α) και (β) που ακολουθούν, παρουσιάζεται ένας μη κατευθυνόμενος γράφος και ο πίνακας γειτονικών κορυφών του, ο οποίος, όπως παρατηρούμε, είναι **συμμετρικός**. (Είναι προφανές ότι αν τον πίνακα θελήσουμε να τον χειριστούμε προγραμματιστικά με την χρήση της C, μπορούμε να κάνουμε ό,τι για όλους τους πίνακες, δηλαδή να ξεκινήσουμε την αρίθμηση γραμμών και στηλών από το μηδέν).




Σχ. 10.2α

| | v1 | v2 | v3 | v4 | v5 |
|---|---|---|---|---|---|
| **v1** | 0 | 1 | 1 | 1 | 0 |
| **v2** | 1 | 0 | 0 | 1 | 0 |
| **v3** | 1 | 0 | 0 | 1 | 0 |
| **v4** | 1 | 1 | 1 | 0 | 1 |
| **v5** | 0 | 0 | 0 | 0 | 1 |

Σχ. 10.2β

Από τον πίνακα γειτονικών κορυφών μπορεί να προκύψει **ο βαθμός κάποιας κορυφής** ενός μη κατευθυνόμενου γράφου, **αθροίζοντας τα περιεχόμενα της αντίστοιχης σειράς του πίνακα**. Έτσι, ο βαθμός $d_k$ της k κορυφής δίνεται από το άθροισμα:

$$d_k = \sum_{j=1}^{n} pin[k][j]$$

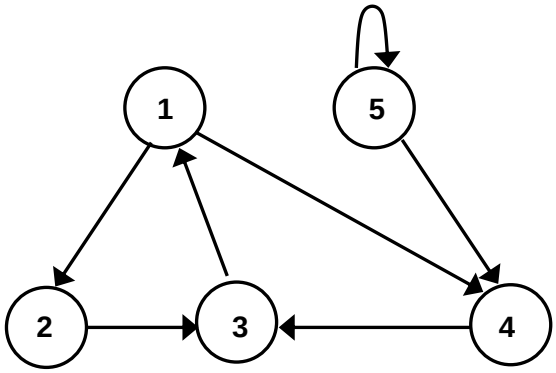
(θεωρώντας ότι η αρίθμηση γραμμών και στηλών αρχίζει από το 1). Έτσι, ο βαθμός της κορυφής v1 του πιο πάνω σχήματος 10.2 είναι 3, ο βαθμός της v4 είναι 4 κλπ.

Αν ο **γράφος** είναι **κατευθυνόμενος**, για τον πίνακα γειτονικών κορυφών ισχύει ότι:

**pin[ j ][k] = 1, αν η κορυφή $v_k$ είναι γειτονική από την κορυφή $v_j$** και

**pin[ j ][k] = 0, αν η κορυφή $v_k$ δεν είναι γειτονική από την κορυφή $v_j$**

Στα σχ. 10.3 (α) και (β) παρουσιάζονται αντίστοιχα ένας κατευθυνόμενος γράφος και ο πίνακας γειτονικών κορυφών του.




Σχ. 10.3α

| | v1 | v2 | v3 | v4 | v5 |
|---|---|---|---|---|---|
| **v1** | 0 | 1 | 0 | 1 | 0 |
| **v2** | 0 | 0 | 1 | 0 | 0 |
| **v3** | 1 | 0 | 0 | 0 | 0 |
| **v4** | 0 | 0 | 1 | 0 | 0 |
| **v5** | 0 | 0 | 0 | 1 | 1 |

Σχ. 10.3β

Σε ένα κατευθυνόμενο γράφο, ο βαθμός εισόδου της κορυφής k δίνεται από το **άθροισμα των τιμών της στήλης k** του πίνακα γειτονικών κορυφών, ενώ ο βαθμός εξόδου της κορυφής k δίνεται από το **άθροισμα των τιμών της γραμμής k** του πίνακα.

Από πίνακες, όπως αυτός του σχ. 10.3β και κάνοντας χρήση ιδιοτήτων των πινάκων και στοιχείων Γραμμικής Άλγεβρας, μπορούμε να παρατηρήσουμε τα παρακάτω: **Εάν πολλαπλασιάσουμε τον πίνακα με τον εαυτό του**, προκύπτει ένας νέος πίνακας ίδιων προφανώς διαστάσεων, ο οποίος ονομάζεται **πίνακας μονοπατιού μήκους 2**. Για τον πίνακα αυτόν (έστω ότι λέγεται A2) ισχύουν τα εξής:

- Εάν το A2[ j ][ k ] είναι ίσο με μηδέν, τότε δεν υπάρχει μονοπάτι μήκους 2, το οποίο να συνδέει την κορυφή j με την κορυφή k.

- Εάν το A2[ j ][ k ] είναι διάφορο του μηδενός, τότε υπάρχει μονοπάτι μήκους 2, το οποίο να συνδέει την κορυφή j με την κορυφή k και μάλιστα τα μονοπάτια είναι τόσα, όσος ο αριθμός A2[ j ][ k ].

Ομοίως, πολλαπλασιάζοντας τον πίνακα μονοπατιού μήκους 2 με τον αρχικό πίνακα, καταλήγουμε στον **πίνακα μονοπατιού μήκους 3**, δηλαδή ένα πίνακα που περιγράφει εάν υπάρχει μονοπάτι μήκους 3 μεταξύ δύο κορυφών κ.ο.κ.

Οι πίνακες μονοπατιού μήκους 2 και μήκους 3 για τον γράφο του σχ. 10.3 είναι οι πίνακες των παρακάτω σχ. 10.4α και 10.4β αντίστοιχα:

| | **v1** | **v2** | **v3** | **v4** | **v5** |
|---|---|---|---|---|---|
| **v1** | 0 | 0 | 2 | 0 | 0 |
| **v2** | 1 | 0 | 0 | 0 | 0 |
| **v3** | 0 | 1 | 0 | 1 | 0 |
| **v4** | 1 | 0 | 0 | 0 | 0 |
| **v5** | 0 | 0 | 1 | 1 | 1 |

(α)

| | **v1** | **v2** | **v3** | **v4** | **v5** |
|---|---|---|---|---|---|
| **v1** | 2 | 0 | 0 | 0 | 0 |
| **v2** | 0 | 1 | 0 | 1 | 0 |
| **v3** | 0 | 0 | 2 | 0 | 0 |
| **v4** | 0 | 1 | 0 | 1 | 0 |
| **v5** | 1 | 0 | 1 | 1 | 1 |

(β)

Σχ. 10.4



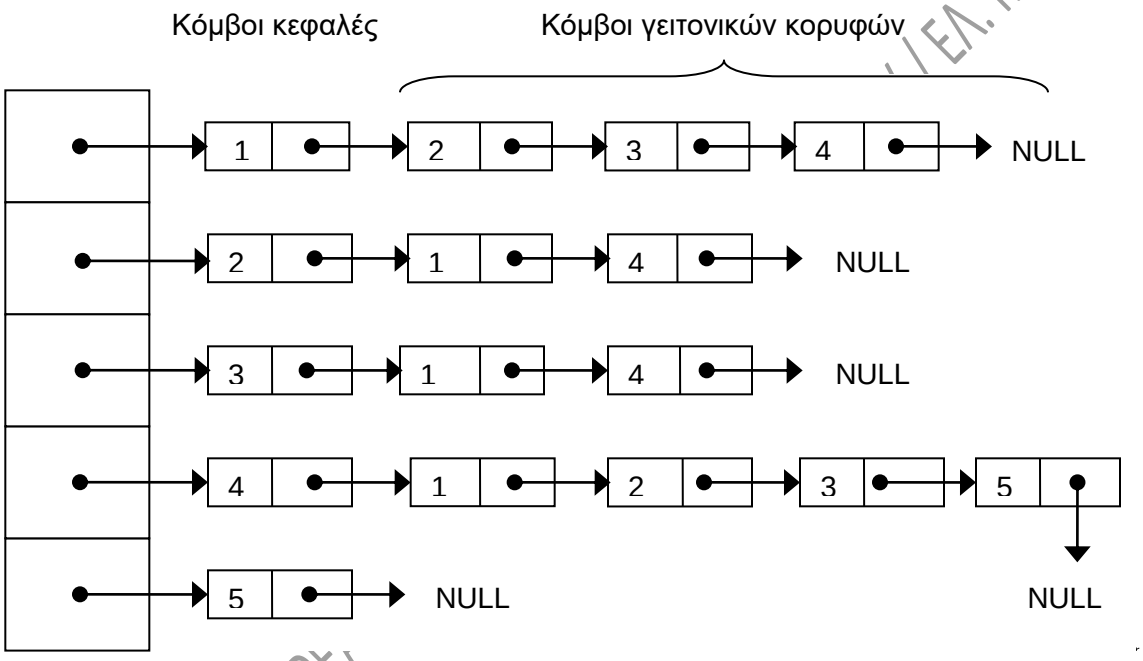
Από τα παραπάνω καταλαβαίνουμε ότι δεν μπορούμε να διαγράψουμε μια κορυφή ή να εισαγάγουμε μια νέα σε ένα γράφο, ο οποίος παριστάνεται με πίνακα γειτονικών κορυφών, αφού οι πίνακες αποτελούν **«στατικές»** δομές. Πρέπει επίσης να σημειωθεί ότι **αν ο γράφος είναι αραιός**, η παράσταση με πίνακα γειτονικών κορυφών οδηγεί σε **μεγάλη σπατάλη μνήμης**.

### 10.3.2. Λίστες γειτονικών κορυφών.

Στην παράσταση ενός γράφου με λίστες γειτονικών κορυφών, **σε κάθε κορυφή του γράφου αντιστοιχούμε μια απλά συνδεδεμένη λίστα**, η οποία περιέχει όλες τις γειτονικές κορυφές της αρχικής. Ο γράφος υλοποιείται με την **χρήση ενός πίνακα**

**δεικτών**, κάθε στοιχείο του οποίου είναι ένας δείκτης προς ένα κόμβο μιας λίστας. Η κεφαλή της λίστας περιέχει τον αριθμό της κορυφής του γράφου και ένα δείκτη προς τον επόμενο κόμβο, ο οποίος με την σειρά του περιέχει τον αριθμό μιας κορυφής γειτονικής προς την αρχική και δείκτη προς τον επόμενο κόμβο κοκ.

Σχηματικά, οι λίστες γειτονικών κορυφών για τον γράφο του σχ. 10.2α δίνονται στο παρακάτω σχ. 10.5. Στο σχήμα αυτό θεωρήσαμε ότι η αρίθμηση των κόμβων του σχ. 10.2α είναι: 1 για τον κόμβο v1, 2 για τον v2, 3 για τον v3, 4 για τον v4 και 5 για τον κόμβο v5.




Σχ. 10.5



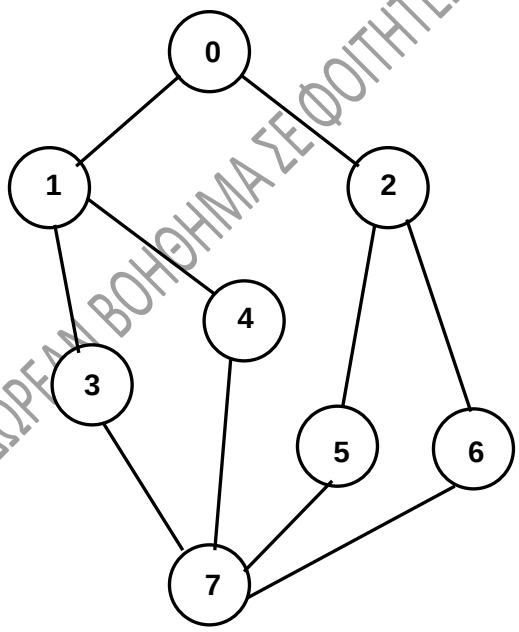
## 10.4. ΔΙΑΣΧΙΣΗ ΓΡΑΦΩΝ.

Το πρόβλημα της διάσχισης ενός γράφου είναι αντίστοιχο με το πρόβλημα διάσχισης ενός δένδρου και έγκειται στην επίσκεψη όλων των κορυφών του γράφου, ξεκινώντας από κάποια κορυφή. Οι μέθοδοι διάσχισης ενός γράφου είναι η διάσχιση με προτεραιότητα βάθους (Depth First Search, DFS) και η διάσχιση με προτεραιότητα πλάτους (Breadth First Search, BFS).

### 10.4.1. Διάσχιση με προτεραιότητα βάθους (Depth First Search, DFS).

Σύμφωνα με αυτό τον τρόπο διάσχισης ενός γράφου, **ξεκινούμε από μια τυχούσα κορυφή και επισκεπτόμαστε κάθε φορά μια νέα κορυφή, την οποία δεν έχουμε ακόμα επισκεφθεί και η οποία είναι γειτονική αυτής στην οποία βρισκόμαστε. Αν δεν υπάρχει τέτοια κορυφή, κάνουμε ένα βήμα προς τα πίσω και επαναλαμβάνουμε την ίδια διαδικασία στην προηγούμενη κορυφή από την οποία έχουμε περάσει**. Είναι προφανές ότι κάθε κορυφή πρέπει να «μαρκάρεται» με κάποιο τρόπο όταν την επισκεφθούμε, ώστε να αποφύγουμε την επίσκεψή της για δεύτερη φορά.

Στο σχ. 10.6 παρουσιάζεται ένας μη κατευθυνόμενος γράφος 8 κορυφών. Στη συνέχεια δίνονται ενδεικτικά αποτελέσματα διάσχισης του γράφου με την μέθοδο της προτεραιότητας βάθους, ανάλογα με την κορυφή από την οποία ξεκινούμε. Προφανώς υπάρχουν περισσότερα του ενός αποτελέσματα σε κάθε μια από τις περιπτώσεις.




Σχ. 10.6

Αρχή από την κορυφή 0 : 0, 1, 3, 7, 4, 5, 2, 6
Αρχή από την κορυφή 1 : 1, 0, 2, 5, 7, 3, 4, 6
Αρχή από την κορυφή 5 : 5, 2, 0, 1, 3, 7, 6, 4

### 10.4.2. Διάσχιση με προτεραιότητα πλάτους (Breadth First Search, BFS).

Σύμφωνα με αυτό τον τρόπο διάσχισης, ξεκινούμε και πάλι από μια τυχούσα κορυφή. **Για κάθε κορυφή επισκεπτόμαστε όλες τις γειτονικές της κορυφές με την σειρά, χωρίς φυσικά να λαμβάνουμε υπ' όψη όσες έχουμε ήδη επισκεφθεί**. Κάθε κορυφή, την οποία επισκεπτόμαστε για πρώτη φορά, τοποθετείται σε μια ουρά. Μόλις ολοκληρωθεί η επίσκεψη όλων των γειτονικών κορυφών κάποιας εξεταζόμενης κορυφής, ανακαλούμε αυτήν που βρίσκεται στην κεφαλή της ουράς και επαναλαμβάνουμε την ίδια διαδικασία. Περιγράφουμε στη συνέχεια λεπτομερώς τον αλγόριθμο που θα ακολουθήσουμε για την διαδικασία αυτή:

Βήμα 1: Εκτύπωση αρχικής κορυφής.

Βήμα 2: Σήμανση της αρχικής κορυφής ως «επισκεφθείσας».

Βήμα 3: Ώθηση της κορυφής στην ουρά.

Βήμα 4: Ανάκληση από την ουρά: ελεγχόμενη κορυφή.

Βήμα 5: Υπάρχει γειτονική κορυφή της ελεγχόμενης; Εάν ΝΑΙ, μετάβαση στο βήμα 6, αλλιώς μετάβαση στο βήμα 9.

Βήμα 6: Έχει σήμανση «επισκεφθείσας» η γειτονική κορυφή που επισημάναμε; Αν ΝΑΙ, μετάβαση στο βήμα 7, αλλιώς στο βήμα 8.

Βήμα 7: Υπάρχει άλλη γειτονική κορυφή της ελεγχόμενης; Αν ΝΑΙ, μετάβαση στο βήμα 6, αλλιώς στο βήμα 9.

Βήμα 8: Εάν υπάρχει γειτονική κορυφή «μη επισκεφθείσα» τότε:

Βήμα 8.1: «Εκτύπωση» κορυφής.

Βήμα 8.2: Ώθηση κορυφής στην ουρά.

Βήμα 8.3: Σήμανση της κορυφής ως «επισκεφθείσας».

Βήμα 8.4: Μετάβαση στο βήμα 7.

Βήμα 9: Εάν υπάρχει κορυφή στην ουρά, μετάβαση στο βήμα 4, αλλιώς τέλος.

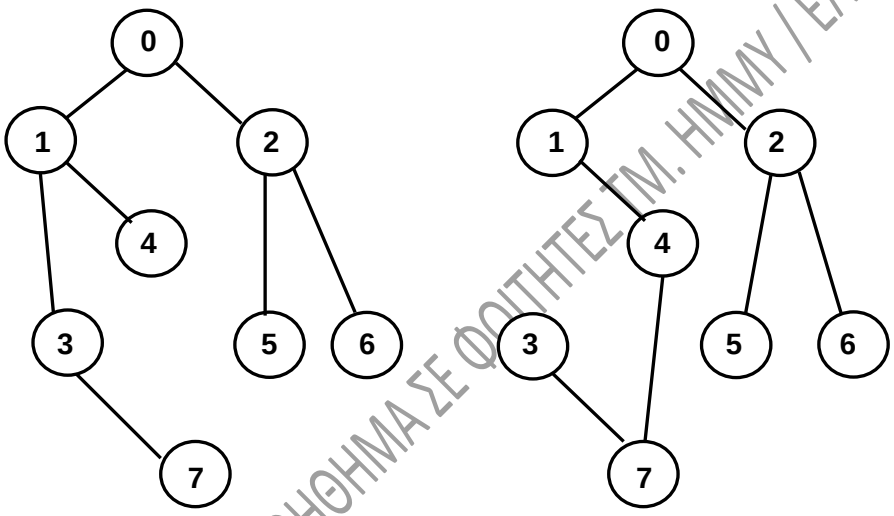
Αν διατρέξουμε τώρα τον γράφο του σχ. 10.6 με προτεραιότητα πλάτους, θα εμφανιστούν οι κορυφές με την εξής σειρά:

0, 1, 2, 3, 4, 5, 6, 7

σαν να τις είχαμε δηλαδή διατρέξει ανά επίπεδο (χρησιμοποιώντας βέβαια καταχρηστικά αυτή την έκφραση στους γράφους), οπτική που δικαιολογεί πάντως κατά κάποιο τρόπο το όνομα του αλγορίθμου.

## 10.5. ΔΕΝΤΡΑ ΕΠΙΚΑΛΥΨΗΣ.

Είναι γνωστά και ως ζευγνύοντα **δέντρα** (στα αγγλικά **spanning trees**). Εάν έχουμε ένα **μη κατευθυντικό γράφο**, ένα **δέντρο επικάλυψης του γράφου είναι ένας υπογράφος, ο οποίος είναι δέντρο** (άρα όχι κλειστός γράφος). **Το δέντρο αυτό έχει δημιουργηθεί από μερικές ακμές του αρχικού γράφου, έτσι ώστε να περιέχει όλες τις κορυφές του**. Είναι φανερό ότι για ένα δεδομένο γράφο, μπορεί να υπάρχουν περισσότερα από ένα διαφορετικά δέντρα επικάλυψης. Στο σχ. 10.7 που ακολουθεί, δίνονται δύο δέντρα επικάλυψης για τον γράφο του σχ. 10.6.

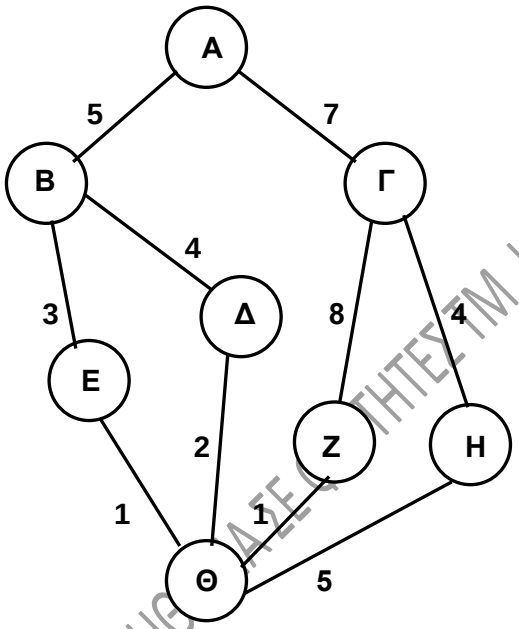



Σχ. 10.7

Όπως αναφέρθηκε στην αρχή της παραγράφου 10.2, ένας γράφος μπορεί να είναι **ζυγισμένος**, οπότε κάθε ακμή του χαρακτηρίζεται από ένα αριθμό, ο οποίος ονομάζεται βάρος και το οποίο μπορεί να υποδηλώνει διάφορα μεγέθη. Ένας ζυγισμένος γράφος φαίνεται στο σχ. 10.8. Στην περίπτωση του ζυγισμένου γράφου, μπορούμε να αναζητήσουμε τα διάφορα δέντρα επικάλυψης, έχει όμως πολλές φορές νόημα να αναζητήσουμε το **ελάχιστο δέντρο επικάλυψης (minimum spanning tree, MST)** του γράφου. **Το δέντρο αυτό είναι εκείνο το δέντρο επικάλυψης με το μικρότερο συνολικό βάρος από όλα τα δέντρα επικάλυψης του γράφου**.

Η χρησιμότητα εύρεσης του MST είναι πάρα πολύ μεγάλη και έχει εφαρμογές σε πάρα πολλούς τομείς της καθημερινότητας και της τεχνολογίας. Φανταστείτε μια εταιρεία με πολλά γραφεία σε διάφορες πόλεις, στην οποία εταιρεία χρειαζόμαστε μισθωμένες γραμμές για να ενώσουμε όλα τα γραφεία μεταξύ τους, αλλά η εταιρεία τηλεφωνίας χρεώνει διαφορετικά ποσά για την σύνδεση των πόλεων μεταξύ τους, ανάλογα με την

θέση τους. Επιθυμούμε να ενοικιάσουμε ένα σύνολο γραμμών για να συνδέσουμε τα γραφεία μεταξύ τους με το χαμηλότερο δυνατό κόστος. Η λύση είναι η σχεδίαση ενός ζυγισμένου γράφου, στον οποίο το βάρος κάθε ακμής θα παριστάνει κόστος ενοικίασης. Στην συνέχεια, θα βρούμε ένα δέντρο επικάλυψης στον γράφο και μάλιστα το ελάχιστο δέντρο επικάλυψης, το οποίο θα παριστάνει πώς θα ενώνονται όλα τα γραφεία μεταξύ τους με το ελάχιστο κόστος.




Σχ. 10.8



### 10.5.1. Εύρεση του δέντρου επικάλυψης – Αλγόριθμος PRIM.

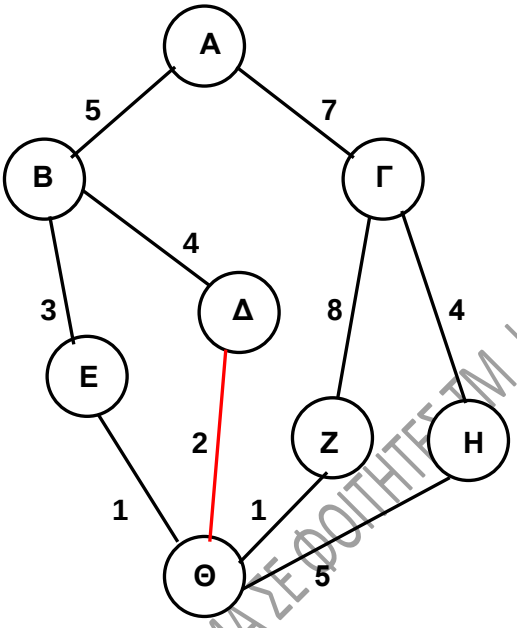
Η εύρεση του MST σύμφωνα με τον αλγόριθμο Prim υλοποιείται ως εξής:

1. **Επιλέγουμε** αυθαίρετα **μια κορυφή και την συνδέουμε με την πλησιέστερη σε αυτήν** κορυφή.
2. **Βρίσκουμε μια μη συνδεδεμένη κορυφή που είναι πλησιέστερη προς κάποιαν από τις συνδεδεμένες και συνδέουμε τις κορυφές αυτές** (την «καινούργια» με την ήδη συνδεδεμένη). Εάν υπάρχουν περισσότερες από μια πλησιέστερες, επιλέγουμε κάποιαν από αυτές αυθαίρετα.
3. **Επαναλαμβάνουμε το βήμα 2**, μέχρι να συνδεθούν όλες οι κορυφές του γράφου.

Δείτε σχηματικά την υλοποίση του αλγορίθμου αυτού χρησιμοποιώντας τον γράφο του

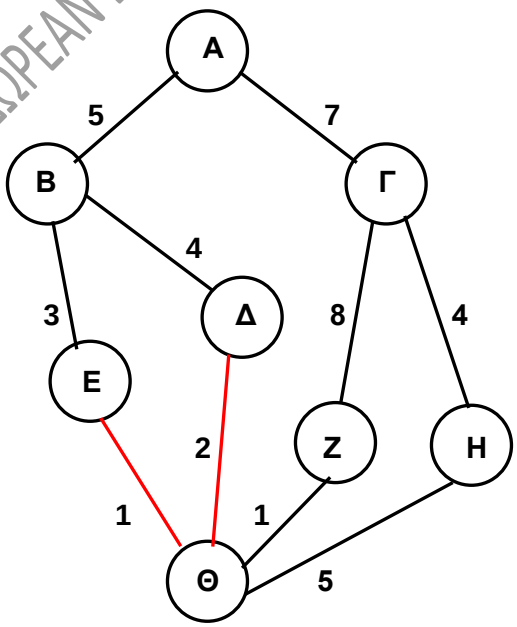
σχ. 10.8. Σε κάθε βήμα με κόκκινο χρώμα συμβολίζουμε την ακμή, η οποία συνδέει την κορυφή που ενσωματώνουμε, προκειμένου να καταλήξουμε στο MST:

Επιλέγουμε ως πρώτη κορυφή του MST την κορυφή Δ. Η πλησιέστερη σε αυτήν είναι η Θ, οπότε συνδέουμε την Δ με την Θ (σχ. 10.9α):



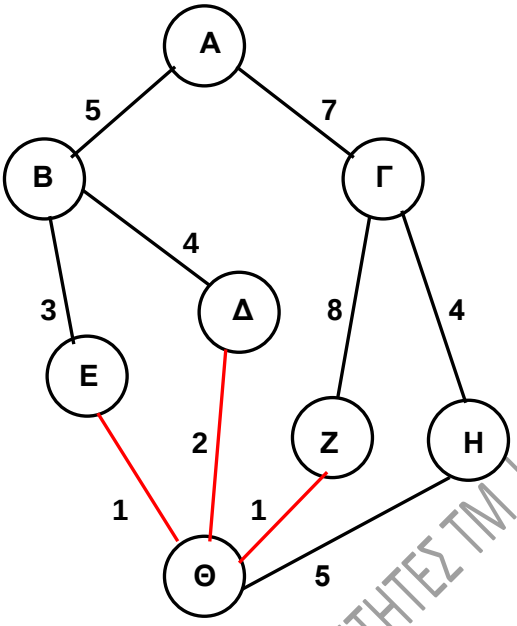

Σχ. 10.9α

Η πλησιέστερη μη συνδεδεμένη κορυφή στην Δ ή στην Θ είναι η Ε ή η Ζ. Επιλέγουμε αυθαίρετα την Ε και την συνδέουμε με την Θ (σχ. 10.9β):



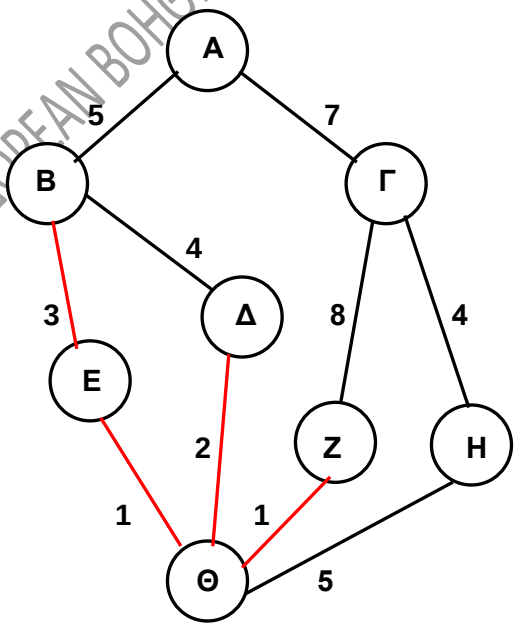

Σχ. 10.9β

Η πλησιέστερη μη συνδεδεμένη κορυφή στην Δ ή στην Θ ή στην Ε είναι η Ζ, την οποία συνδέουμε με την Θ (σχ. 10.9γ):



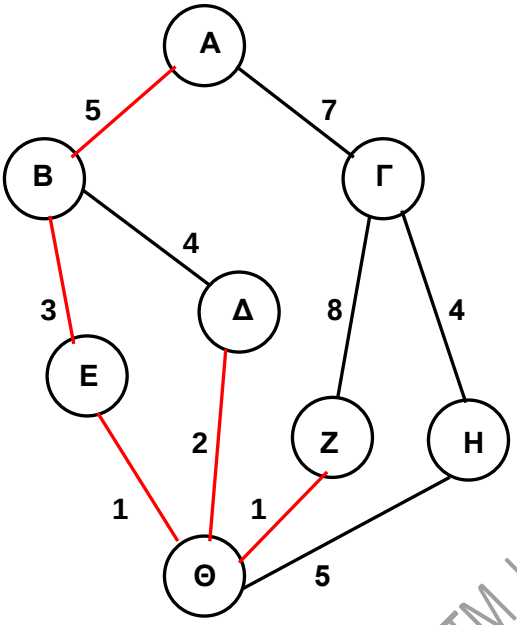

Σχ. 10.9γ

Η πλησιέστερη μη συνδεδεμένη κορυφή στην Δ ή στην Θ ή στην Ε ή στην Ζ είναι η Β, την οποία συνδέουμε με την Ε (σχ. 10.9δ):



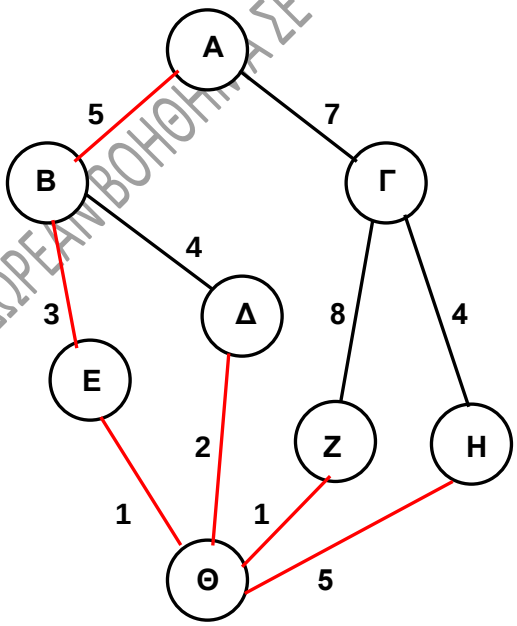

Σχ. 10.9δ

Η πλησιέστερη μη συνδεδεμένη κορυφή στην Δ ή στην Θ ή στην Ε ή στην Ζ ή στην Β, είναι η Α ή η Η. Επιλέγουμε αυθαίρετα την Α, την οποία συνδέουμε με την Β (σχ. 10.9ε):



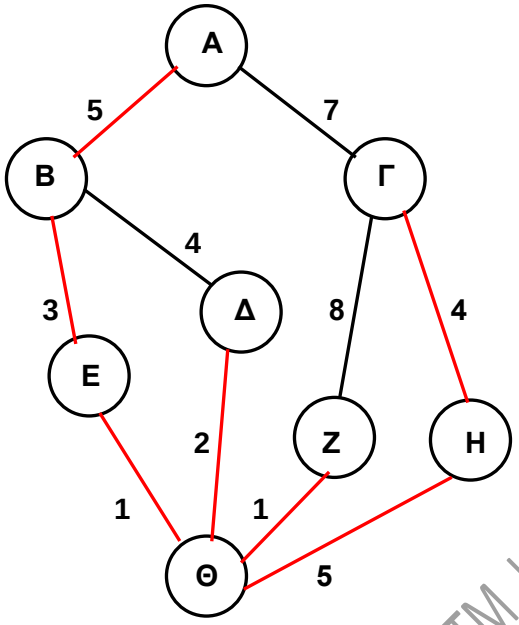

Σχ. 10.9ε

Η πλησιέστερη μη συνδεδεμένη κορυφή στην Δ ή στην Θ ή στην Ε ή στην Z ή στην Β ή στην Α είναι η Η, την οποία συνδέουμε με την Θ (σχ. 10.9στ):




Σχ. 10.9στ

Η πλησιέστερη μη συνδεδεμένη κορυφή στην Δ ή στην Θ ή στην Ε ή στην Z ή στην Β ή στην Α ή στην Η είναι η Γ, την οποία συνδέουμε με την Η, αφού από αυτήν έχει την μικρότερη απόσταση (σχ. 10.9ζ). Το δένδρο που έχει προκύψει είναι το MST, αφού έχουν πλέον συνδεθεί όλες οι κορυφές του γράφου.




Σχ. 10.9ζ



## 10.5.2. Εύρεση του ελάχιστου δέντρου επικάλυψης – Αλγόριθμος KRUSKAL.

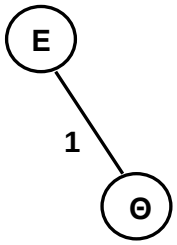
Ο αλγόριθμος του Prim που είδαμε προηγουμένως επεξεργάζεται μια-μια τις κορυφές του γράφου. **Ο αλγόριθος του Kruskal επεξεργάζεται μια-μια τις ακμές του γράφου.** Σε κάθε βήμα του αλγόριθμου του Prim οι ακμές που επιλέγονται σχηματίζουν ένα δέντρο, ενώ στην περίπτωση του αλγόριθμου του Kruskal, σχηματίζουν ένα **δάσος**, δηλαδή ένα σύνολο από δέντρα, τα οποία τελικά ενώνονται σε ένα. Έτσι:

1. **Διατάσσουμε** τις ακμές του γράφου σε αύξουσα σειρά βάρους.
2. Βρίσκουμε την **ακμή με το ελάχιστο βάρος,** από αυτές που δεν έχουμε ακόμη επεξεργαστεί και **την εισάγουμε στο υπό δημιουργία δέντρο με τις κορυφές στις οποίες είναι προσκείμενη, με την προϋπόθεση ότι η ακμή αυτή δεν δημιουργεί κύκλο,** αλλιώς την αγνοούμε.
3. **Επαναλαμβάνουμε το βήμα 2**, μέχρι να επεξεργαστούμε όλες τις ακμές του γράφου.

Δείτε την υλοποίση του αλγορίθμου αυτού χρησιμοποιώντας τον γράφο του σχ. 10.9α (ο ίδιος με αυτόν του σχ. 10.8). Οι ακμές με τα αντίστοιχα βάρη κατ' αύξουσα σειρά είναι οι εξής:

| | | | |
|---|---|---|---|
| ΘΕ | 1 | ΔΒ | 4 |
| ΘΖ | 1 | ΒΑ | 5 |
| ΘΔ | 2 | ΘΗ | 5 |
| ΕΒ | 3 | ΓΑ | 7 |
| ΗΓ | 4 | ΖΓ | 8 |

Στο σχ. 10.10α βλέπουμε το δέντρο μετά την εισαγωγή της ακμής ΘΕ (ή της ΘΖ, εισάγεται τυχαία η μία από τις δύο αφού έχουν το ίδιο βάρος) και τις ακμές που απομένουν:

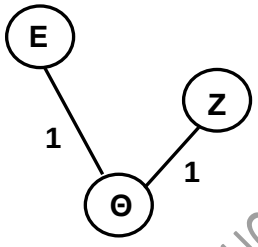



Υπόλοιπες ακμές

| | |
|---|---|
| ΘΖ | 1 |
| ΘΔ | 2 |
| ΕΒ | 3 |
| ΗΓ | 4 |
| ΔΒ | 4 |
| ΒΑ | 5 |
| ΘΗ | 5 |
| ΓΑ | 7 |
| ΖΓ | 8 |

Σχ. 10.10α

Εισάγουμε την ακμή ΘΖ:

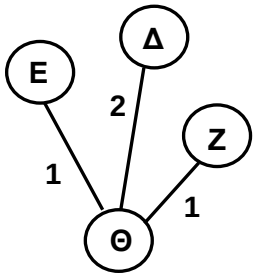



Υπόλοιπες ακμές

| | |
|---|---|
| ΘΔ | 2 |
| ΕΒ | 3 |
| ΗΓ | 4 |
| ΔΒ | 4 |
| ΒΑ | 5 |
| ΘΗ | 5 |
| ΓΑ | 7 |
| ΖΓ | 8 |

Σχ. 10.10β

Εισάγουμε την ακμή ΘΔ:

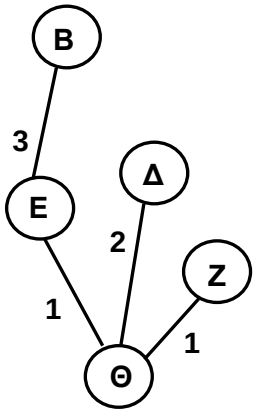



Υπόλοιπες ακμές

| | |
|---|---|
| ΕΒ | 3 |
| ΗΓ | 4 |
| ΔΒ | 4 |
| ΒΑ | 5 |
| ΘΗ | 5 |
| ΓΑ | 7 |
| ΖΓ | 8 |

Σχ. 10.10γ

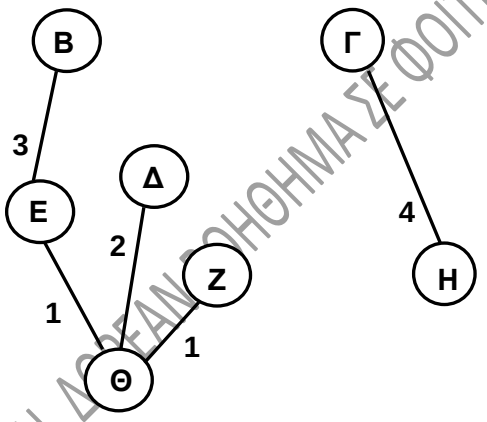
Εισάγουμε την ακμή ΕΒ:




| Υπόλοιπες ακμές | |
|---|---|
| ΗΓ | 4 |
| ΔΒ | 4 |
| ΒΑ | 5 |
| ΘΗ | 5 |
| ΓΑ | 7 |
| ΖΓ | 8 |

Σχ. 10.10δ

Εισάγουμε την ακμή ΗΓ (ή την ΔΒ, τυχαία την μία από τις δύο αφού έχουν το ίδιο βάρος):



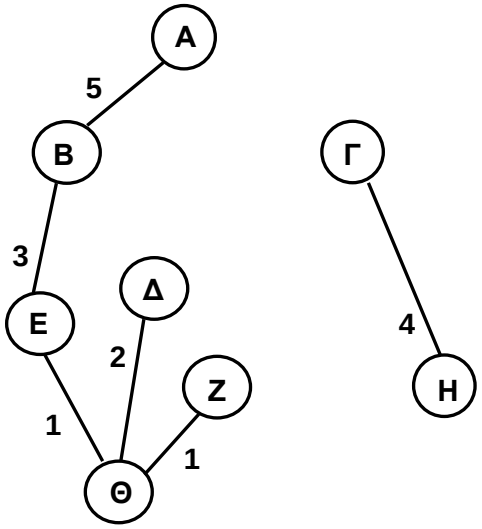

| Υπόλοιπες ακμές | |
|---|---|
| ΔΒ | 4 |
| ΒΑ | 5 |
| ΘΗ | 5 |
| ΓΑ | 7 |
| ΖΓ | 8 |

Σχ. 10.10ε

Στο σχήμα αυτό παρατηρούμε ότι έχουμε ένα δάσος από δύο δένδρα στην περίπτωσή μας, όπως αναφέρθηκε και στην αρχή αυτής της παραγράφου.
Η επόμενη ακμή για εισαγωγή είναι η ΔΒ, η οποία όμως δεν εισάγεται αφού δημιουργεί κύκλο. Απομένουν λοιπόν οι ακμές:

| | |
|---|---|
| ΒΑ | 5 |
| ΘΗ | 5 |
| ΓΑ | 7 |
| ΖΓ | 8 |

Εισάγουμε την ακμή ΒΑ:

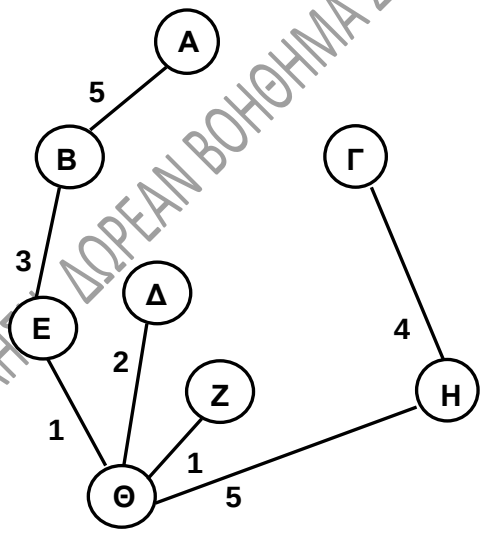



Υπόλοιπες ακμές

| | |
|---|---|
| ΘΗ | 5 |
| ΓΑ | 7 |
| ΖΓ | 8 |

Σχ. 10.10στ

Εισάγουμε την ακμή ΘΗ:




Υπόλοιπες ακμές

| | |
|---|---|
| ΓΑ | 7 |
| ΖΓ | 8 |

Σχ. 10.10ζ

Και οι δύο ακμές που απομένουν, εάν εισαχθούν δημιουργούν κύκλο, οπότε αγνοούνται.

Το δέντρο που προέκυψε είναι το MST σύμφωνα με τον αλγόριθμο Kruskal.

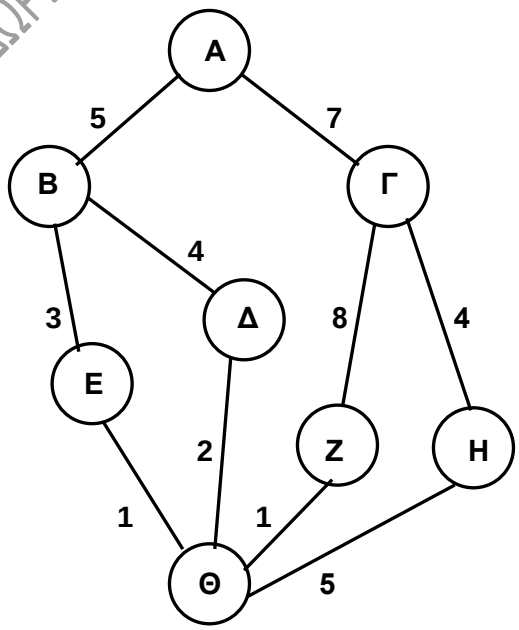
## 10.6. ΣΥΝΤΟΜΟΤΕΡΟ ΜΟΝΟΠΑΤΙ – ΑΛΓΟΡΙΘΜΟΣ DIJKSTRA.

Μια πολύ συνηθισμένη και χρήσιμη εφαρμογή σε ζυγισμένους γράφους (δηλαδή γράφους με βάρη), κατευθυνόμενους ή μη, είναι η εύρεση ενός μονοπατιού από μια κορυφή σε μια άλλη, έτσι ώστε **το άθροισμα των βαρών του μονοπατιού να είναι το ελάχιστο**. Στις περιπτώσεις αυτές μιλάμε για το **συντομότερο μονοπάτι**. Όπως έχει ήδη αναφερθεί, τα βάρη των ακμών αντιπροσωπεύουν χρόνο, απόσταση, κόστος κλπ.

Ο **αλγόριθμος Dijkstra** υπολογίζει **το συντομότερο μονοπάτι από κάποια κορυφή** (ας την ονομάσουμε «πηγή», Π) **προς όλες τις άλλες κορυφές ενός γράφου**. Για την σωστή λειτουργία του αλγορίθμου, τα **βάρη** πρέπει να είναι **μη αρνητικοί αριθμοί**.

Ο αλγόριθμος διατηρεί το σύνολο S των κορυφών για τις οποίες σε κάθε φάση έχει υπολογιστεί το μήκος του συντομότερου μονοπατιού από την Π, καθώς και ένα πίνακα, τον D, ο οποίος περιλαμβάνει σε κάθε βήμα του αλγορίθμου τις αποστάσεις dA, dB κλπ της Π από κάθε άλλη κορυφή του γράφου. Σε περίπτωση που η απόσταση αυτή δεν μπορεί να υπολογιστεί, τίθεται ίση με άπειρο (∞). Σε κάθε βήμα εισάγεται στο σύνολο S μια από τις κορυφές, της οποίας η απόσταση από την Π είναι η ελάχιστη. Όταν θα έχουν περιληφθεί όλες οι κορυφές στο S, τότε ο πίνακας D θα περιλαμβάνει την συντομότερη απόσταση από την Π προς κάθε κορυφή του γράφου.

Στο σχ. 10.11α έχουμε ένα ζυγισμένο μη κατευθυνόμενο γράφο, θεωρούμε δε ως πηγή την κορυφή Α του γράφου. Αναζητούμε δηλαδή την ελάχιστη απόσταση του Α από τις υπόλοιπες κορυφές.




Σχ. 10.11α

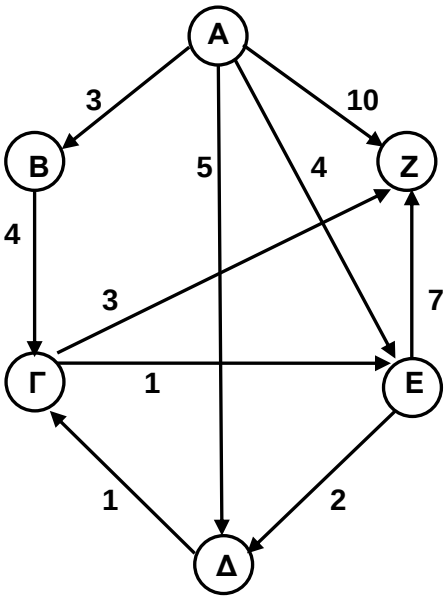
Στο σχ. 10.11β παρουσιάζονται τα διαδοχικά βήματα υλοποίησης του αλγορίθμου.

| S | dA | dB | dΓ | dΔ | dE | dZ | dH | dΘ | Επόμενη |
|---|---|---|---|---|---|---|---|---|---|
| **A** | 0 | 5 | 7 | ∞ | ∞ | ∞ | ∞ | ∞ | **B** |
| **A, B** | 0 | 5 | 7 | 9 | 8 | ∞ | ∞ | ∞ | **Γ** |
| **A, B, Γ** | 0 | 5 | 7 | 9 | 8 | 15 | 11 | ∞ | **E** |
| **A, B, Γ, E** | 0 | 5 | 7 | 9 | 8 | 15 | 11 | 9 | **Θ** |
| **A, B, Γ, E, Θ** | 0 | 5 | 7 | 9 | 8 | 10 | 11 | 9 | **Δ** |
| **A, B, Γ, E, Θ, Δ** | 0 | 5 | 7 | 9 | 8 | 10 | 11 | 9 | **Z** |
| **A, B, Γ, E, Θ, Δ, Z** | 0 | 5 | 7 | 9 | 8 | 10 | 11 | 9 | **H** |
| **A, B, Γ, E, Θ, Δ, Z, H** | 0 | 5 | 7 | 9 | 8 | 10 | 11 | 9 | |

Σχ. 10.11β

**Η τελευταία σειρά** του παραπάνω πίνακα μας δίνει **το συντομότερο μονοπάτι από την πηγή** Α **προς κάθε κόμβο του γράφου**.

Αντίστοιχα, στο σχ. 10.12α δίνεται ένας κατευθυνόμανος γράφος και στο σχ. 10.12β δίνονται τα διαδοχικά βήματα υλοποίησης του αλγορίθμου του αλγορίθμου Dijkstra για τον γράφο αυτό, θεωρώντας ως «πηγή» την κορυφή Α του γράφου.




Σχ. 10.12α

| S | dA | dB | dΓ | dΔ | dE | dZ | Επόμενη |
|---|---|---|---|---|---|---|---|
| A | 0 | 3 | ∞ | 5 | 4 | 10 | B |
| A, B | 0 | 3 | 7 | 5 | 4 | 10 | E |
| A, B, E | 0 | 3 | 7 | 5 | 4 | 10 | Δ |
| A, B, E, Δ | 0 | 3 | 6 | 5 | 4 | 10 | Γ |
| A, B, E, Δ, Γ | 0 | 3 | 6 | 5 | 4 | 9 | Z |
| A, B, E, Δ, Γ, Z | 0 | 3 | 6 | 5 | 4 | 9 | |

Σχ. 10.12β

Στην περίπτωση που τα βάρη των ακμών περιλαμβάνουν και αρνητικούς αριθμούς, τότε ο αλγόριθμος του Dijkstra δεν βγάζει σωστά αποτελέσματα, αλλά χρησιμοποιείται ο αλγόριθμος των Bellman-Ford.