Δομές

- Πολλές φορές, μία σύνθετη οντότητα μπορεί να καθορισθεί από ένα σύνολο δεδομένων, πιθανώς διαφορετικών τύπων, οπότε θα ήταν χρήσιμο να ομαδοποιούσαμε τα δεδομένα αυτά κάτω από ένα γενικό όνομα, με σκοπό να αναφερόμαστε στην οντότητα αυτή με το όνομα αυτό.
- Στην C, η δυνατότητα ομαδοποίησης δεδομένων, που αποτελούν κατά κάποιο τρόπο τα χαρακτηριστικά μίας προγραμματιστικής οντότητας, παρέχεται από τις δομές.
- Μία δομή ορίζεται με τον εξής τρόπο:
  `struct` ⟨ετικέτα δομής⟩ {
  ⟨τύπος⟩$_1$ ⟨μέλος⟩$_1$;
  ⟨τύπος⟩$_2$ ⟨μέλος⟩$_2$;
  .....
  ⟨τύπος⟩$_n$ ⟨μέλος⟩$_n$;
  };
  Ο ορισμός μίας δομής αποτελείται από τους ορισμούς των μελών της, που περιγράφονται σαν μεταβλητές συγκεκριμένων τύπων.
- Φυσικά, πίνακες, δείκτες ή ακόμα και δομές μπορούν να είναι μέλη μίας δομής.
- Για πιο αναλυτική συζήτηση περί δομών, παραπέμπεστε στις παραγράφους §6.1 έως §6.6 του [KR]. Για δημιουργία νέων ονομάτων τύπων (εντολή `typedef`), ενώσεις και πεδία bit, που θα συζητηθούν στη συνέχεια, οι παράγραφοι από το [KR] είναι οι §6.7, §6.8 και §6.9, αντίστοιχα.

- Παραδείγματα:

```
struct employee {
  char firstname[10];
  char lastname[18];
  int id_number;
  float salary;
};
struct wordinfo {
  char *word;
  int nlines;
};
```

- Στα παραδείγματα αυτά ορίζουμε μία δομή `struct employee` για την αναπαράσταση πληροφοριών για ένα υπάλληλο (μικρό όνομα, επώνυμο, αριθμός μητρώου και μισθός) και μία δομή `struct wordinfo` για την αναπαράσταση της πληροφορίας σχετικά με την ύπαρξη μίας λέξης σε κάποιο κείμενο, καθώς και το πλήθος των γραμμών στις οποίες εμφανίζεται.

- Ο ορισμός μίας δομής είναι μία δήλωση για την οποία δεν γίνεται κάποια δέσμευση μνήμης. Μπορούμε όμως να ορίσουμε συγκεκριμένες μεταβλητές που έχουν σαν τύπο μία δομή, ως εξής:
`struct` ⟨ετικέτα δομής⟩ ⟨μεταβλητή⟩$_1$, ..., ⟨μεταβλητή⟩$_k$;
Τότε, δεσμεύεται η απαιτούμενη μνήμη για τις μεταβλητές, ανάλογα με τον χώρο που χρειάζεται για να φυλαχθούν τα μέλη της δομής.

- Παραδείγματα:

```
struct employee jim, jane, the_employee;
struct wordinfo first_word, next_word, last_word;
```

- Άλλα παραδείγματα δηλώσεων δομών:

```
struct point {
  double x;
  double y;
};
struct upright_rectangle {
  struct point p1;
  struct point p2;
};
```

- Εδώ ορίζουμε τη δομή `struct point` για την αναπαράσταση ενός σημείου στο επίπεδο (μέσω των συντεταγμένων του), καθώς και τη δομή `struct upright_rectangle` για την αναπαράσταση ενός ορθογωνίου παραλληλογράμμου στο επίπεδο, με τις πλευρές παράλληλες στους άξονες (μέσω δύο σημείων που είναι απέναντι κορυφές σε μία διαγώνιο). [α΄]

[α΄] Μπορείτε να ορίσετε δομές και για άλλες γεωμετρικές οντότητες στο επίπεδο, για παράδειγμα, ευθύγραμμα τμήματα, τρίγωνα, κύκλους, ή, ανεξαρτήτως προσανατολισμού, τετράγωνα, ρόμβους, ορθογώνια παραλληλόγραμμα, (πλάγια) παραλληλόγραμμα και τυχαία τετράπλευρα; Μπορείτε να ορίσετε και γεωμετρικές οντότητες στον τρισδιάστατο χώρο;

- Ο ορισμός μεταβλητών με τύπο κάποια συγκεκριμένη δομή μπορεί να γίνει ταυτόχρονα με τη δήλωση της δομής. Παράδειγμα:

```
struct point {
  double x;
  double y;
} pa, pb, pc;
```

  Στην περίπτωση αυτή, δεν είναι απαραίτητο να δώσουμε ετικέτα στη δομή. Για παράδειγμα, ο εξής ορισμός των μεταβλητών `pa`, `pb` και `pc` είναι αποδεκτός:

```
struct {
  double x;
  double y;
} pa, pb, pc;
```

  Φυσικά, τότε, αν θέλουμε στη συνέχεια να ορίσουμε και άλλες μεταβλητές του τύπου της δομής, πρέπει να επαναλάβουμε τη δήλωσή της, αφού δεν της είχαμε δώσει κάποια ετικέτα.

- Μεταβλητές με τύπο δομή μπορούν να ανατίθενται σαν τιμές σε άλλες μεταβλητές του ίδιου τύπου, μπορούν να δίνονται σαν ορίσματα κατά την κλήση συναρτήσεων και μπορούν να επιστρέφονται από συναρτήσεις στο όνομά τους.

- Με τον ορισμό μίας μεταβλητής τύπου δομής, μπορούμε να κάνουμε και αρχικοποίηση των μελών της. Παράδειγμα:

```
struct point my_point = {22.4, -38.9};
```

  Για αυτόματες μεταβλητές τύπου δομής, αλλά και προκαθορισμένων τύπων, αρχικοποίηση μπορεί να γίνει και με εντολή αντικατάστασης ή μέσω επιστροφής συνάρτησης.

- Η αναφορά σ΄ ένα μέλος μίας μεταβλητής τύπου δομής γίνεται με την παράσταση
  ⟨μεταβλητή⟩ . ⟨μέλος⟩

- Παραδείγματα:

```
struct point vert1, vert2;
struct upright_rectangle my_rect;
vert1.x = 2.4;
vert2.y = 7.8;
my_rect.p1.x = -8.3;
```

  Η τελευταία εντολή είναι ισοδύναμη με την

```
(my_rect.p1).x = -8.3;
```

  επειδή ο τελεστής . είναι αριστερά προσεταιριστικός.

- Όπως ορίζουμε μεταβλητές με τύπο κάποια δομή, έτσι μπορούμε να ορίσουμε και δείκτες σε δομές. Παράδειγμα:

  ```
  struct point *ppa, *ppb;
  ```

  Φυσικά, είναι δική μας ευθύνη να κάνουμε τους δείκτες `ppa` και `ppb` να δείξουν σε διευθύνσεις στις οποίες φυλάσσονται ή θα φυλαχθούν δεδομένα τύπου δομής. Αυτό μπορεί να γίνει είτε με την ανάθεση της διεύθυνσης κάποιας ήδη ορισμένης μεταβλητής τύπου δομής, είτε με δυναμική δέσμευση. Παραδείγματα:

  ```
  struct point my_point;
  ppa = &my_point;
  ppb = malloc(sizeof(struct point));
  ```

- Έχοντας ορίσει ένα δείκτη σε δομή, όπως τον `ppa` στο προηγούμενο παράδειγμα, μπορούμε να αναφερθούμε σε συγκεκριμένο μέλος κατά τα γνωστά, για παράδειγμα `(*ppa).x`.

- Στην παράσταση `(*ppa).x` οι παρενθέσεις είναι απαραίτητες. Αν γράφαμε `*ppa.x`, αυτό, λόγω των σχετικών προτεραιοτήτων των τελεστών `.` και `*` θα ήταν ισοδύναμο με το `*(ppa.x)`, το οποίο όμως, στην προκείμενη περίπτωση δεν είναι συντακτικά σωστό, αφού το `ppa.x` δεν είναι δείκτης.

- Αν ένας ⟨δείκτης⟩ δείχνει σε κάποια δομή και θέλουμε να αναφερθούμε σ΄ ένα ⟨μέλος⟩ της, αντί να γράφουμε
  (*⟨δείκτης⟩) .⟨μέλος⟩
  η C μας δίνει τη δυνατότητα να γράψουμε πιο απλά
  ⟨δείκτης⟩->⟨μέλος⟩

- Όπως ο τελεστής ., έτσι και ο -> είναι αριστερά προσεταιριστικός. Για παράδειγμα, αν έχουμε ορίσει

  ```
  struct upright_rectangle *rp;
  ```

  τότε η παράσταση `rp->p1.x` είναι η ίδια με την `(rp->p1).x`.

- Οι τελεστές . και -> μαζί με τις () για κλήσεις συναρτήσεων και τις [] για δείκτες πινάκων βρίσκονται στην κορυφή της ιεραρχίας προτεραιότητας των τελεστών και επομένως συνδέονται ισχυρά με τους τελεστέους.

- Παράδειγμα:

  ```
  struct wordinfo *pw;
  ```

  Με την έκφραση `++pw->nlines` αυξάνει το `nlines` γιατί είναι ισοδύναμη με την `++(pw->nlines)`. Αν θέλαμε να αυξήσουμε τον δείκτη `pw` θα έπρεπε να γράψουμε `(++pw)->nlines` ή `(pw++)->nlines`[α΄], ανάλογα με το πότε θα επιθυμούσαμε να γίνει η αύξηση του δείκτη, πριν ή μετά την προσπέλαση του μέλους `nlines`.

- Έχοντας πλέον αναφερθεί σε όλους τους τελεστές της C, ο Πίνακας 2–1 του [KR] (σελ. 83), είναι μία πολύ χρήσιμη πηγή για αναφορά, σχετικά με τις προτεραιότητές τους.

[α΄] Εδώ δεν είναι απαραίτητες οι παρενθέσεις, αφού δεν θα υπήρχε αμφιβολία για το τι θα σήμαινε το `pw++->nlines`.

- Όπως συμβαίνει και με όλους τους τύπους στην C, μπορούμε να ορίσουμε και πίνακες δομών.
- Παράδειγμα:

```
int main(void)
{ int i, n;
  struct worddata {
    char *word;
    int numb;
    char let;
  } wordarray[] = {
      {"ABCD", 1, 'a'}, {"EF", 2, 'b'},
      {"GHIJ", 3, 'c'}, {"KL", 4, 'd'},
      {"M", 5, 'e'}};
  struct worddata *p = wordarray;
  n = sizeof(wordarray)/sizeof(struct worddata);
  printf("%d ", n);
  for (i=0 ; i < 2 ; i++) {
    printf("%c ", *p->word++);
    printf("%s ", ++p->word);
    printf("%c ", p->let++);
    printf("%c ", (++p)->let);
    printf("%d ", p++->numb);
    printf("%d ", ++p->numb); }
  for (i=0 ; i < n ; i++)
    printf("%s %d %c ", wordarray[i].word,
           wordarray[i].numb, wordarray[i].let);
  printf("\n"); return 0; }
```

Τι θα εκτυπωθεί από το πρόγραμμα αυτό; [α΄]

[α΄] `5 A CD a b 2 4 G IJ c d 4 6 CD 1 b EF 2 b IJ 4 d KL 4 d M 6 e`

Εύρεση τριγώνων μεγίστου και ελαχίστου εμβαδού

```
/* File: triangles.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

struct point {
  double x;                  /* A point in 2 dimensions is defined */
  double y;                        /* by its x- and y- coordinates */
} *points;

struct triangle {
  struct point a;   /* A triangle is defined by its three vertices */
  struct point b;
  struct point c;
};

struct triangle get_triangle(int, int, int);
double triangle_area(struct triangle *);
double get_side(struct point, struct point);

int main(int argc, char *argv[])
{ int i, j, k, n = 10;
  long seed;
  struct triangle tr, max_tr, min_tr;
  double area, max_area, min_area;
  seed = time(NULL);
  if (argc > 1)
    n = atoi(argv[1]);
  if (n < 3) {           /* In order to have at least one triangle */
    printf("At least three points are needed\n");
    return 1;
  }
  if ((points = malloc(n * sizeof(struct point))) == NULL) {
                              /* Allocate memory to store n points */
    printf("Sorry, not enough memory!\n");
    return 1;
  }
```

```
                              /* Initialize random number generator */
  srand((unsigned int) seed);
  for (i=0 ; i < n ; i++) {
         /* Generate points with coordinates between 0.0 and 100.0 */
     /* RAND_MAX is max number that rand() returns - usually 32767 */
    (points+i)->x = (100.0 * rand())/(RAND_MAX+1.0);
    (points+i)->y = (100.0 * rand())/(RAND_MAX+1.0);
  }
  for (i=0 ; i < n ; i++)              /* Printout points generated */
    printf("P%-2d: (%4.1f,%4.1f)\n", i, (points+i)->x, (points+i)->y);
  tr = get_triangle(0, 1, 2);                  /* Get first triangle */
  area = triangle_area(&tr);     /* Compute area of first triangle */
           /* Initialize max and min triangles with first triangle */
  max_tr = min_tr = tr;
  max_area = min_area = area;    /* Areas of max and min triangles */
  for (i=0 ; i < n-2 ; i++)    /* Iterate through all combinations */
    for (j=i+1 ; j < n-1 ; j++)  /* of points in order to form all */
      for (k=j+1 ; k < n ; k++) {             /* possible triangles */
        tr = get_triangle(i, j, k);         /* Get current triangle */
        area = triangle_area(&tr);      /* Area of current triangle */
        if (area > max_area) {       /* Is current larger than max? */
          max_tr = tr;
          max_area = area;
        }
        if (area < min_area) {      /* Is current smaller than min? */
          min_tr = tr;
          min_area = area;
        }
      }
  printf("\n");
  printf("Max triangle: ");                /* Printout max triangle */
  printf("(%4.1f,%4.1f) ", max_tr.a.x, max_tr.a.y);
  printf("(%4.1f,%4.1f) ", max_tr.b.x, max_tr.b.y);
  printf("(%4.1f,%4.1f) ", max_tr.c.x, max_tr.c.y);
  printf(" Area: %10.5f\n", max_area);
  printf("Min triangle: ");                /* Printout min triangle */
  printf("(%4.1f,%4.1f) ", min_tr.a.x, min_tr.a.y);
  printf("(%4.1f,%4.1f) ", min_tr.b.x, min_tr.b.y);
  printf("(%4.1f,%4.1f) ", min_tr.c.x, min_tr.c.y);
  printf(" Area: %10.5f\n", min_area);
  return 0;
}
```

```
struct triangle get_triangle(int i, int j, int k)
{ struct triangle tr;
  tr.a = *(points+i);
  tr.b = *(points+j);
  tr.c = *(points+k);
  return tr;              /* Return triangle with vertices i, j, k */
}

double triangle_area(struct triangle *tr)
{ double s1, s2, s3, t;
  s1 = get_side(tr->a, tr->b);            /* Get length of side ab */
  s2 = get_side(tr->b, tr->c);            /* Get length of side bc */
  s3 = get_side(tr->c, tr->a);            /* Get length of side ca */
  t = (s1+s2+s3)/2;               /* Compute half of the perimeter */
  return sqrt(t*(t-s1)*(t-s2)*(t-s3));             /* Return area */
}

double get_side(struct point p1, struct point p2)
{            /* Return Euclidean distance between points p1 and p2 */
  return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
```

```
% gcc -o triangles triangles.c -lm
% ./triangles 17
P0 : (37.5,51.6)
P1 : (31.4, 0.8)
P2 : (10.3,32.8)
P3 : ( 4.5,38.4)
P4 : (27.4,96.3)
P5 : (28.4,43.0)
P6 : (69.8,55.7)
P7 : (13.3,71.4)
P8 : (17.8,76.2)
P9 : ( 3.9, 8.8)
P10: (29.6,69.6)
P11: (20.6,43.3)
P12: ( 9.1,76.1)
P13: (33.9,25.6)
P14: (34.1,66.4)
P15: (15.1,22.9)
P16: (67.5,49.7)

Max triangle: (27.4,96.3) (69.8,55.7) ( 3.9, 8.8)  Area: 2331.69411
Min triangle: (28.4,43.0) (17.8,76.2) (33.9,25.6)  Area:    0.09948
%
```

Αυτο-αναφορικές δομές

- Τα μέλη μίας δομής μπορεί να είναι οποιουδήποτε τύπου, ακόμα και δείκτες σε δομές του ίδιου τύπου. Χρησιμοποιώντας τέτοιου είδους δομές, που ονομάζονται αυτο-αναφορικές και είναι πολύ χρήσιμες στον προγραμματισμό, μπορούμε να οργανώσουμε δεδομένα με τρόπους που διευκολύνουν εξαιρετικά τη διαχείριση και επεξεργασία τους.

- Οι πλέον συνήθεις οργανώσεις δεδομένων μέσω αυτο-αναφορικών δομών είναι οι συνδεδεμένες λίστες (ή, απλά, λίστες) και τα δυαδικά δέντρα. Μπορεί κανείς να ορίσει και άλλες οργανώσεις δεδομένων, όπως διπλά συνδεδεμένες λίστες, ουρές, δέντρα όχι κατ΄ ανάγκη δυαδικά, κλπ.

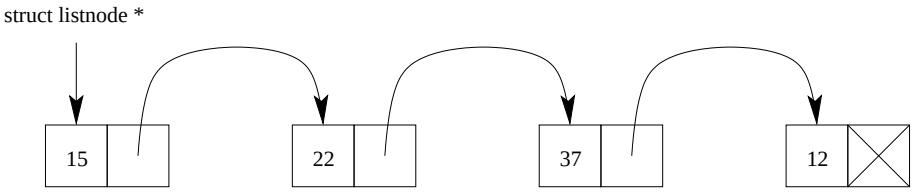
- Έστω η δήλωση:

```
struct listnode {
  int value;
  struct listnode *next;
};
```

  Αυτή η δομή ορίζει έναν κόμβο λίστας στον οποίο φυλάσσεται ένας ακέραιος και ένας δείκτης σε κόμβο λίστας.

- Αν θέλουμε να αποθηκεύσουμε μία ακολουθία από ακεραίους, απροσδιορίστου πλήθους, αλλά και μεταβαλλόμενου κατά τη διάρκεια της εκτέλεσης ενός προγράμματος, μπορούμε να το κάνουμε βάζοντας κάθε ακέραιο σ΄ ένα κόμβο λίστας και δείχνοντας από κάθε κόμβο στον επόμενό του.

- Η αναφορά στη λίστα γίνεται με ένα δείκτη στον πρώτο κόμβο της. Ο τελευταίος κόμβος της λίστας έχει σαν δείκτη σε επόμενο κόμβο το `NULL`.

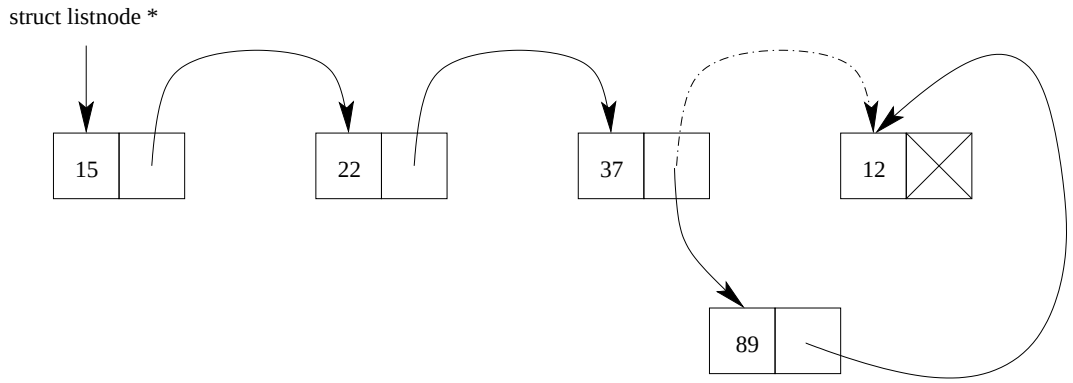


  Αυτή είναι μία λίστα με στοιχεία, κατά σειρά, τα 15, 22, 37 και 12.

- Για να υλοποιήσουμε τη δυναμική φύση της λίστας, πρέπει πάντα να γίνεται, για τη φύλαξη ενός κόμβου της, η κατάλληλη δυναμική δέσμευση μνήμης (μέσω `malloc`) και, φυσικά, η αποδέσμευση (μέσω `free`), όταν δεν χρειαζόμαστε πλέον τον κόμβο.

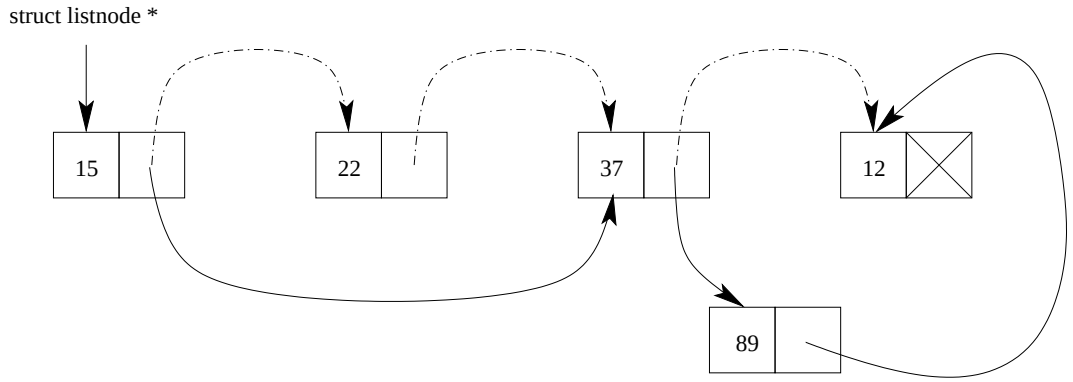
- Οι κόμβοι μίας λίστας δεν φυλάσσονται σε διαδοχικές θέσεις μνήμης, αφού η μνήμη γι΄ αυτούς δεσμεύεται με διαφορετικές `malloc`. Αυτό σημαίνει ότι δεν μπορούμε να έχουμε άμεση πρόσβαση στο N-οστό στοιχείο μίας λίστας, όπως στους πίνακες, αλλά μόνο ακολουθώντας την αλυσίδα των στοιχείων από το πρώτο έως το N-οστό.

- Η παρεμβολή ενός κόμβου σε συγκεκριμένη θέση σε μία λίστα δεν απαιτεί την μετακίνηση κανενός άλλου κόμβου. Απλώς πρέπει να τροποποιηθεί ο δείκτης για επόμενο του κόμβου που προηγείται της θέσης που εισάγεται ο νέος κόμβος, ώστε να δείξει πλέον στον νέο κόμβο. Επίσης, ο δείκτης για επόμενο του νέου κόμβου θα πρέπει να δείξει στον επόμενο κόμβο από τη θέση που έγινε η εισαγωγή.



Εδώ, έγινε εισαγωγή, στη λίστα της προηγούμενης σελίδας, του στοιχείου 89 μεταξύ των στοιχείων 37 και 12.

- Αντίστοιχα ισχύουν και για τη διαγραφή ενός κόμβου από μία λίστα. Δεν απαιτείται καμία μετακίνηση.



Εδώ, έγινε διαγραφή του στοιχείου 22 από την προηγούμενη λίστα.

- Φυσικά, μπορούμε να ορίσουμε κόμβους λίστας στους οποίους η πληροφορία που φυλάσσουμε (πλην του δείκτη στον επόμενο κόμβο) να είναι οποιουδήποτε τύπου, όχι μόνο ακέραιος, αλλά και κινητής υποδιαστολής, δείκτης σε κάποιο τύπο (π.χ. συμβολοσειρά), γενικά οτιδήποτε.

- Επίσης, μπορούμε σ΄ ένα κόμβο λίστας να ορίσουμε περισσότερα από ένα μέλη για την πληροφορία του κόμβου, για παράδειγμα, μία συμβολοσειρά, έναν ακέραιο και ένα δείκτη σε λίστα. [α΄]

- Μία άλλη, πολύ χρήσιμη, οργάνωση δεδομένων μέσω αυτο-αναφορικών δομών είναι τα δυαδικά δέντρα.

- Ένα δυαδικό δέντρο είναι μία συλλογή κόμβων, οργανωμένων σε μία δενδρική διάταξη, καθένας από τους οποίους μπορεί να έχει μέχρι δύο κόμβους-παιδιά. Κάθε κόμβος έχει έναν κόμβο-γονέα, εκτός από τον πρώτο κόμβο του δέντρου, που είναι ο κόμβος-ρίζα.

- Τα παιδιά ενός κόμβου αναφέρονται σαν το αριστερό και το δεξί παιδί του κόμβου. Για την αναπαράσταση ενός κόμβου χρησιμοποιούμε μία αυτο-αναφορική δομή, όπως η εξής:

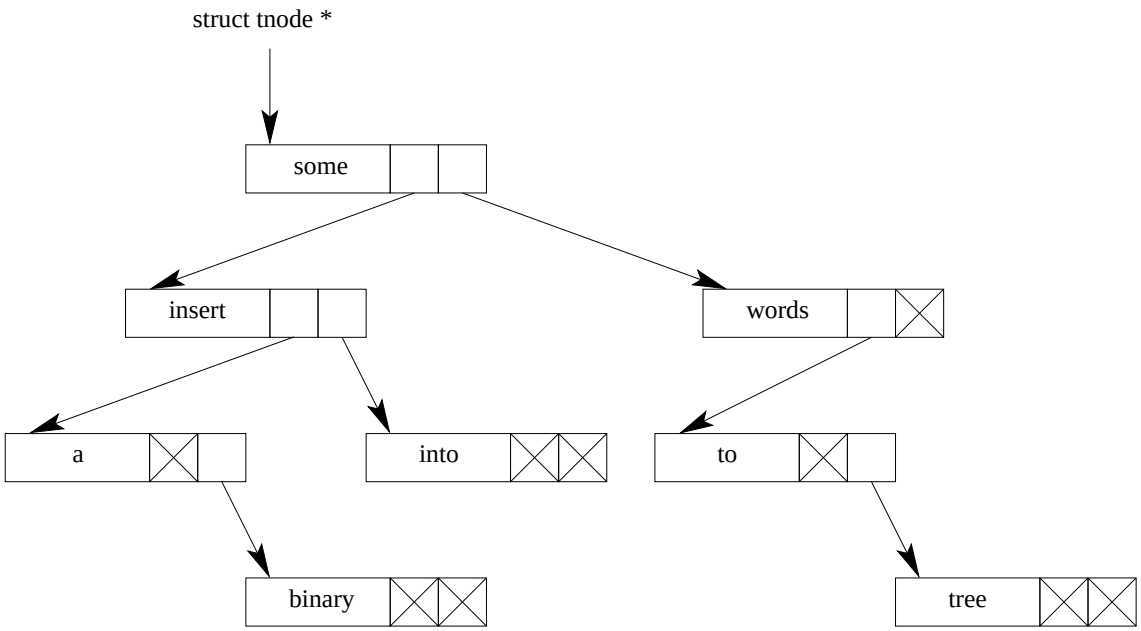
```
struct tnode {
  int value;
  struct tnode *left;
  struct tnode *right;
};
```

[α΄] Δεν είναι ενδιαφέρον να μπορούμε να έχουμε και λίστες από λίστες;

- Με τη χρήση της δομής `struct tnode`, σε κάθε κόμβο του δέντρου φυλάσσεται ένας ακέραιος, αλλά θα μπορούσαμε σαν πληροφορία προς φύλαξη να έχουμε δεδομένα οποιουδήποτε τύπου, όπως και στις λίστες, και, φυσικά, και περισσότερα του ενός δεδομένα. Οι δείκτες `left` και `right` είναι οι διευθύνσεις των κόμβων που είναι αριστερό και δεξί παιδί, αντίστοιχα, του κόμβου.
- Όπως και στις λίστες, αν κάποιο παιδί δεν υπάρχει, ο αντίστοιχος δείκτης είναι `NULL`.
- Η αναφορά σ΄ ένα δυαδικό δέντρο γίνεται μέσω ενός δείκτη στη ρίζα του.
- Κατά τη δημιουργία ενός δυαδικού δέντρου, πρέπει να δεσμεύεται δυναμικά μνήμη (μέσω `malloc`) για να φυλαχθεί κάθε κόμβος, ενώ όταν ένας κόμβος δεν χρειάζεται πλέον, πρέπει η μνήμη που καταλαμβάνει να αποδεσμεύεται (μέσω `free`).
- Τα δυαδικά δέντρα είναι πραγματικά χρήσιμα ως οργανώσεις δεδομένων όταν είναι ταξινομημένα. Ένα δυαδικό δέντρο είναι ταξινομημένο όταν, με βάση κάποιο κριτήριο διάταξης στην πληροφορία που αποθηκεύουμε στους κόμβους του, ο κόμβος-ρίζα έπεται όλων των κόμβων στο αριστερό παιδί του και προηγείται όλων των κόμβων στο δεξί παιδί του και, επίσης, τόσο το αριστερό όσο και το δεξί παιδί είναι ταξινομημένα με την ίδια λογική.

- Σαν κριτήρια διάταξης της πληροφορίας των κόμβων ενός δέντρου, μπορούμε να έχουμε την αριθμητική διάταξη, αν η πληροφορία αυτή είναι ένας ακέραιος ή ένας πραγματικός αριθμός, ή την αλφαβητική διάταξη, αν η πληροφορία είναι μία συμβολοσειρά. Αν έχουμε περισσότερες της μίας πληροφορίες στον κόμβο, τότε κάποια, δηλαδή το αντίστοιχο μέλος της δομής, παίζει τον ρόλο του λεγόμενου κλειδιού, που με βάση αυτό ορίζεται η διάταξη μεταξύ δύο κόμβων.

- Έστω η πρόταση "some words to insert into a binary tree". Στο σχήμα, φαίνεται ένα ταξινομημένο δυαδικό δέντρο που περιέχει τις λέξεις της πρότασης αυτής στους κόμβους του.



- Το συγκεκριμένο δέντρο κατασκευάστηκε εισάγοντας σταδιακά τις λέξεις της πρότασης με τη σειρά που εμφανίζονται στην πρόταση. Αν είχαν εισαχθεί με διαφορετική σειρά, το δέντρο θα ήταν τελικά διαφορετικό. [α΄]

[α΄] Ποιο θα ήταν το δέντρο αν η εισαγωγή των λέξεων γινόταν από την τελευταία λέξη της πρότασης προς την πρώτη;

Δημιουργία νέων ονομάτων τύπων

- Στην C παρέχεται η δυνατότητα, μέσω της εντολής `typedef`, να δώσουμε δικά μας ονόματα σε τύπους που χρησιμοποιούμε συχνά, και μετά να ορίζουμε μεταβλητές αυτών των τύπων με βάση το νέο όνομα.

- Παραδείγματα:

```
typedef int Length;
typedef char *String;

typedef struct listnode *Listptr;

struct listnode {
  int value;
  Listptr next;
};

typedef struct tnode *Treeptr;

typedef struct tnode {
  int value;
  Treeptr left;
  Treeptr right;
} Treenode;

Length len, maxlen;
String name, line[10];
Listptr a_list;
Treenode a_tree_node;
Treeptr a_tree;
```

Ενώσεις και πεδία bit

- Όταν η μνήμη ήταν ακριβή, είχε νόημα στα προγράμματά μας να κάνουμε οικονομία μνήμης.
- Ένας τρόπος για εξοικονόμηση μνήμης είναι οι ενώσεις, που ορίζονται όπως οι δομές (με τη λέξη-κλειδί `union`). Δεν δεσμεύεται χώρος για όλα τα μέλη της όταν ορίζεται μία μεταβλητή τύπου ένωσης, αλλά αυτός που απαιτείται για τη φύλαξη του μέλους με τις μεγαλύτερες απαιτήσεις σε μνήμη. Όλα τα μέλη φυλάσσονται σ΄ αυτόν τον χώρο. Παράδειγμα:

```
union alternative_data {
  int selection;
  int ivalue;
  float fvalue;
  char *svalue;
} id_numb;
```

- Οι ενώσεις είναι χρήσιμες και για το πακετάρισμα τυπικών παραμέτρων συναρτήσεων, όταν αυτές δεν χρειάζονται όλες ταυτόχρονα κατά την κλήση της συνάρτησης.
- Άλλος τρόπος για οικονομία μνήμης είναι τα πεδία bit. Ορίζοντας μία δομή, μπορούμε να δηλώσουμε μέλη τύπου ακεραίου, αλλά με συγκεκριμένο πλήθος bits το καθένα.
- Παράδειγμα:

```
struct {
  unsigned int mode  : 2;
  unsigned int bool  : 1;
  unsigned int octal : 3;
} flags;
```

Διαχείριση συνδεδεμένων λιστών

```
/* File: listmanagement.c */
#include <stdio.h>
#include <stdlib.h>

typedef struct listnode *Listptr;

struct listnode {
  int value;
  Listptr next;
};

int empty(Listptr);
int in(Listptr, int);
int n_th(Listptr, int, int *);
void insert_at_start(Listptr *, int);
void insert_at_end(Listptr *, int);
int delete(Listptr *, int);
void print(Listptr);

int main(void)
{ Listptr alist;
  int v;
  alist = NULL;                                        /* List is NULL */
                                              /* Check if list is empty */
  printf("List is%s empty\n", empty(alist) ? "" : " not");
  insert_at_start(&alist, 44);                 /* List is 44--> NULL */
  printf("List is "); print(alist);
  insert_at_end(&alist, 55);             /* List is 44--> 55--> NULL */
  printf("List is "); print(alist);
  insert_at_start(&alist, 33);     /* List is 33--> 44-> 55--> NULL */
  printf("List is "); print(alist);
  insert_at_end(&alist, 66); /* List is 33--> 44-> 55--> 66--> NULL */
  printf("List is "); print(alist);
                                              /* Check if list is empty */
  printf("List is%s empty\n", empty(alist) ? "" : " not");
                                                  /* Check membership */
  printf("55 is%s in list\n", in(alist, 55) ? "" : " not");
  printf("77 is%s in list\n", in(alist, 77) ? "" : " not");
```

```
  if (n_th(alist, 2, &v))                      /* Return 2nd element */
    printf("Item no 2 is %d\n", v);
  else
    printf("Item no 2 does not exist\n");
  if (n_th(alist, 6, &v))                      /* Return 6th element */
    printf("Item no 6 is %d\n", v);
  else
    printf("Item no 6 does not exist\n");
  printf("Deleting 55. %s\n", delete(&alist, 55) ? "OK!" : "Failed!");
                                  /* List is 33--> 44--> 66--> NULL */
  printf("List is "); print(alist);
  printf("Deleting 22. %s\n", delete(&alist, 22) ? "OK!" : "Failed!");
                                  /* List is 33--> 44--> 66--> NULL */
  printf("List is "); print(alist);
  printf("Deleting 33. %s\n", delete(&alist, 33) ? "OK!" : "Failed!");
                                        /* List is 44--> 66--> NULL */
  printf("List is "); print(alist);
  return 0;
}

int empty(Listptr list)                    /* Check if list is empty */
{ if (list == NULL)                           /* Is it really empty? */
    return 1;                                          /* Yes, it is */
  else
    return 0;                                        /* No, it isn't */
}

int in(Listptr list, int v)          /* Check if v is member of list */
{ while (list != NULL)          /* Visit list elements up to the end */
    if (list->value == v)   /* Did we find what we are looking for? */
      return 1;                                       /* Yes, we did */
    else
      list = list->next;                  /* No, go to next element */
  return 0;                             /* Finally, v is not in list */
}
```

```
int n_th(Listptr list, int n, int *vaddr)
           /* Return n-th element of list, if it exists, into vaddr */
{ while (list != NULL)    /* Maybe search up to the end of the list */
    if (n-- == 1) {              /* Did we reach the right element? */
      *vaddr = list->value;                        /* Yes, return it */
      return 1;                                       /* We found it */
    }
    else
      list = list->next;                  /* No, go to next element */
  return 0;                             /* Sorry, list is too short */
}

void insert_at_start(Listptr *ptraddr, int v)
                      /* Insert v as first element of list *ptraddr */
{ Listptr templist;
  templist = *ptraddr;                /* Save current start of list */
  *ptraddr = malloc(sizeof(struct listnode)); /* Space for new node */
  (*ptraddr)->value = v;                                 /* Put value */
  (*ptraddr)->next = templist;      /* Next element is former first */
}

void insert_at_end(Listptr *ptraddr, int v)
                       /* Insert v as last element of list *ptraddr */
{ while (*ptraddr != NULL)                      /* Go to end of list */
    ptraddr = &((*ptraddr)->next);/* Prepare what we need to change */
  *ptraddr = malloc(sizeof(struct listnode)); /* Space for new node */
  (*ptraddr)->value = v;                                 /* Put value */
  (*ptraddr)->next = NULL;              /* There is no next element */
}
```

```
int delete(Listptr *ptraddr, int v)
               /* Delete element v from list *ptraddr, if it exists */
{ Listptr templist;
  while ((*ptraddr) != NULL)   /* Visit list elements up to the end */
    if ((*ptraddr)->value == v) {    /* Did we find what to delete? */
      templist = *ptraddr;         /* Yes, save address of its node */
      *ptraddr = (*ptraddr)->next;        /* Bypass deleted element */
      free(templist);     /* Free memory for the corresponding node */
      return 1;                           /* We deleted the element */
    }
    else
      ptraddr = &((*ptraddr)->next);/* Prepare what we might change */
  return 0;        /* We did't find the element we were looking for */
}

void print(Listptr list)                  /* Print elements of list */
{ while (list != NULL) {       /* Visit list elements up to the end */
    printf("%d--> ", list->value);         /* Print current element */
    list = list->next;                        /* Go to next element */
  }
  printf("NULL\n");                            /* Print end of list */
}
```

```
% gcc -o listmanagement listmanagement.c
% ./listmanagement
List is empty
List is 44--> NULL
List is 44--> 55--> NULL
List is 33--> 44--> 55--> NULL
List is 33--> 44--> 55--> 66--> NULL
List is not empty
55 is in list
77 is not in list
Item no 2 is 44
Item no 6 does not exist
Deleting 55. OK!
List is 33--> 44--> 66--> NULL
Deleting 22. Failed!
List is 33--> 44--> 66--> NULL
Deleting 33. OK!
List is 44--> 66--> NULL
%
```

Διαχείριση δυαδικών δέντρων

```
/* File: treemanagement.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct tnode *Treeptr;

typedef struct tnode {
  char *word;
  Treeptr left;
  Treeptr right;
} Treenode;

Treeptr addtree(Treeptr, char *);
void treeprint(Treeptr, int);
void nodesprint(Treeptr);
int treedepth(Treeptr);
int treesearch(Treeptr, char *);

int main(int argc, char *argv[])
{ Treeptr p;
  char buf[80];
  p = NULL;                                /* Initialize binary tree */
  while (scanf("%s", buf) != EOF)          /* Read words from input */
    p = addtree(p, buf);          /* and insert them into the tree */
  printf("Tree is:\n"),
  treeprint(p, 0);                  /* Kind of tree pretty printing */
  printf("\nNodes are:\n");
  nodesprint(p);         /* Print tree nodes in alphabetical order */
  printf("\n\nTree depth is %d\n", treedepth(p));
                                /* Compute and print depth of tree */
  printf("\n");
  while (--argc) {                            /* For each argument */
    argv++;       /* check whether it coincides with any tree node */
    printf("%s found %s\n",
           (treesearch(p, *argv)) ? "   " : "not", *argv);
  }
  return 0;
}
```

```
Treeptr addtree(Treeptr p, char *w)   /* Insert word w into tree p */
{ int cond;
  if (p == NULL) {                              /* If tree is empty */
    p = malloc(sizeof(Treenode));   /* Allocate space for new node */
                                    /* Allocate space to copy word */
    p->word = malloc((strlen(w)+1) * sizeof(char));
    strcpy(p->word, w);                /* Copy word w to tree node */
    p->left = NULL;           /* Left subtree of new node is empty */
    p->right = NULL;         /* Right subtree of new node is empty */
  }
  else if ((cond = strcmp(w, p->word)) < 0)
                       /* Does word w precede word of current node? */
    p->left = addtree(p->left, w);
                            /* If yes, insert it into left subtree */
  else if (cond > 0)       /* Does it follow word of current node? */
    p->right = addtree(p->right, w);
                           /* If yes, insert it into right subtree */
  /* If it is the same with word of current node, do not insert it */
  return p;                                          /* Return tree */
}

void treeprint(Treeptr p, int indent)         /* Pretty print tree */
{ int i;
  if (p != NULL) {                          /* If tree is not empty */
    treeprint(p->right, indent+4);
                /* Print right subtree 4 places right of root node */
    for (i=0 ; i < indent ; i++)
      printf(" ");                      /* Take care for indentation */
    printf("%s\n", p->word);                     /* Print root node */
    treeprint(p->left, indent+4);
                 /* Print left subtree 4 places right of root node */
  }
}

void nodesprint(Treeptr p)                      /* Print tree nodes */
{ if (p != NULL) {                           /* If tree is not empty */
    nodesprint(p->left);                       /* Print left subtree */
    printf("%s ", p->word);                       /* Print root node */
    nodesprint(p->right);                    /* Print right subtree */
  }
}
```

```
int treedepth(Treeptr p)                /* Compute depth of tree p */
{ int n1, n2;
  if (p == NULL)                       /* Depth of empty tree is 0 */
    return 0;
  n1 = treedepth(p->left);        /* Compute depth of left subtree */
  n2 = treedepth(p->right);      /* Compute depth of right subtree */
  return (n1 > n2) ? n1+1 : n2+1;
     /* Return maximun of depths of left and right subtrees plus 1 */
}

int treesearch(Treeptr p, char *w)
                              /* Check whether word w is in tree p */
{ int cond;
  if (p == NULL)                                /* If tree is empty */
    return 0;                                 /* We didn't find word */
  if ((cond = strcmp(w, p->word)) == 0)
                  /* Word w is the same with word of current node */
    return 1;
  else if (cond < 0)         /* If w precedes word of current node */
    return treesearch(p->left, w);          /* Search left suftree */
  else                                                /* Otherwise */
    return treesearch(p->right, w);        /* search right subtree */
}
```

```
% gcc -o treemanagement treemanagement.c
% cat words.txt
some
words
to
insert
into
a
binary
tree
% ./treemanagement into found words < words.txt
Tree is:
    words
             tree
         to
some
         into
    insert
             binary
         a

Nodes are:
a binary insert into some to tree words

Tree depth is 4

    found into
not found found
    found words
%
```