

Κεφάλαιο

18

**Δυναμικές
δομές
δεδομένων**



Δυναμικές δομές δεδομένων

Μέχρι στιγμής η μοναδική δομή δεδομένων που έχουμε γνωρίσει είναι ο πίνακας. Οι πίνακες ως δομές καταχώρισης δεδομένων έχουν ένα μεγάλο μειονέκτημα: πρέπει να καθορίζουμε το μέγεθος του πίνακα στην πρόταση δήλωσης του, χωρίς να έχουμε τη δυνατότητα να το αλλάξουμε (αυξήσουμε ή μειώσουμε) κατά τη διάρκεια εκτέλεσης του προγράμματος.

Το αποτέλεσμα είναι η χρήση πινάκων μεγάλου μεγέθους ώστε να είμαστε σίγουροι ότι θα μπορούμε να αποθηκεύσουμε το μεγαλύτερο πιθανό πλήθος δεδομένων που θα έχουμε. Η λύση όμως αυτή οδηγεί σε υπερκατανάλωση μνήμης, η οποία τις περισσότερες φορές δεν είναι απαραίτητη.

Επίσης, αν θέλουμε να διατηρούμε τα δεδομένα του πίνακα ταξινομημένα συνέχεια, η διαδικασία επαναταξινόμησης (ή παρεμβολής) μετά από την προσθήκη κάθε νέου στοιχείου είναι αρκετά χρονοβόρα ιδιαίτερα σε πίνακες μεγάλου μεγέθους.

Η λύση στα προηγούμενα προβλήματα είναι η χρήση δυναμικών δομών, οι οποίες δεσμεύουν μνήμη όταν τη χρειάζονται και την απελευθερώνουν όταν δεν τους είναι πια απαραίτητη.

Η χρήση δυναμικών δομών, εκτός από το ότι λύνει το πρόβλημα της κατανομής μνήμης, δίνει τη δυνατότητα να διατηρούμε τα δεδομένα ταξινομημένα συνεχώς, ώστε να μην είναι πια απαραίτητες οι χρονοβόρες διαδικασίες αναζήτησης και ταξινόμησης.

Οι δυναμικές δομές δεδομένων που θα συζητηθούν στο παρόν κεφάλαιο είναι δύο:

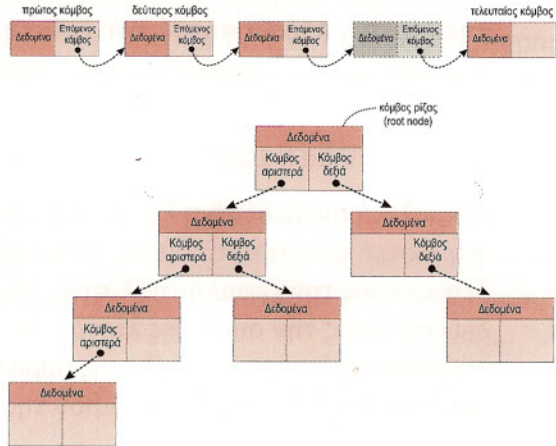
- Οι συνδεδεμένες λίστες (linked lists)
- Τα δυαδικά δένδρα

Το δομικό στοιχείο των δυναμικών δομών δεδομένων είναι ο **κόμβος**. Μια δομή δεδομένων αποτελείται από πολλούς συνδεδεμένους κόμβους. Κάθε κόμβος είναι ένα σύνολο πληροφοριών που περιέχει τόσο τα δεδομένα, όσο και πληροφορίες για τον επόμενο ή τους επόμενους κόμβους.

Η διάταξη των κόμβων διαφέρει ανάλογα με το είδος της δομής δεδομένων (συνδεδεμένη λίστα ή δυαδικό δένδρο).

Στις συνδεδεμένες λίστες, κάθε κόμβος έχει έναν προηγούμενο και έναν επόμενο κόμβο (εκτός φυσικά από τον πρώτο και τον τελευταίο).

Η δομή των δυαδικών δένδρων διαφέρει. Κάθε κόμβος του δένδρου μπορεί να συνδεθεί με έναν κόμβο προς τα αριστερά του και έναν κόμβο προς τα δεξιά του. Ο πρώτος κόμβος του δένδρου λέγεται κόμβος "ρίζας".

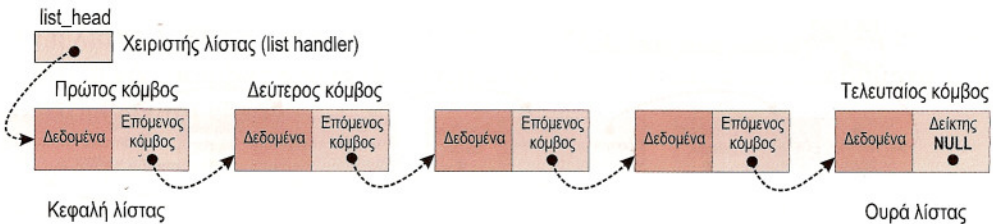


Συνδεδεμένες λίστες

Απλά συνδεδεμένη λίστα (simple linked list)

Η απλά συνδεδεμένη λίστα αποτελείται από στοιχεία (κόμβους, nodes) τα οποία περιέχουν δεδομένα. Κάθε στοιχείο της λίστας περιέχει και ένα δείκτη ο οποίος δείχνει στο επόμενο στοιχείο της λίστας. Επομένως, κάθε στοιχείο της λίστας αποτελείται από:

- Το τμήμα δεδομένων όπου καταχωρίζονται τα δεδομένα του στοιχείου και
- Το δείκτη **Επόμενος κόμβος**, που δείχνει στο επόμενο στοιχείο (κόμβο) της λίστας.

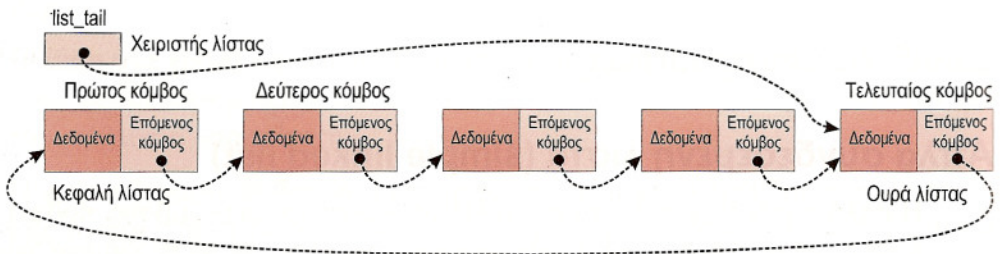


Ο τελευταίος κόμβος περιέχει ένα δείκτη NULL ο οποίος σηματοδοτεί ότι ο συγκεκριμένος κόμβος είναι το τέλος (η "ουρά") της συνδεδεμένης λίστας.

Μια θέση μνήμης τύπου δείκτη περιέχει τη διεύθυνση του κόμβου κεφαλής της λίστας. Μέσω αυτού του δείκτη, ο οποίος αποκαλείται και χειριστής της λίστας (list handler), έχουμε πρόσβαση στη λίστα.

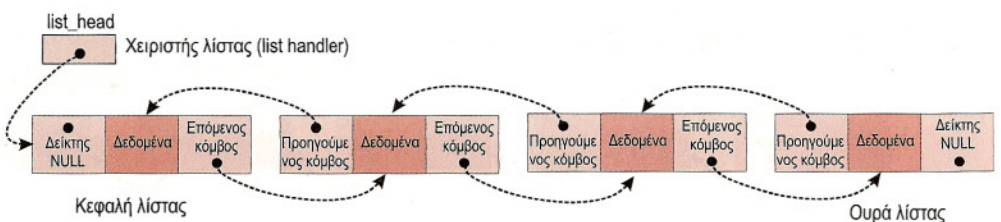
Κυκλικά συνδεδεμένη λίστα (circularly linked list)

Η κυκλικά συνδεδεμένη λίστα είναι ακριβώς ίδια με την απλή λίστα με τη διαφορά ότι ο τελευταίος κόμβος, αντί να περιέχει ένα δείκτη NULL, περιέχει ένα δείκτη προς την κεφαλή της λίστας. Ο χειριστής της λίστας είναι συνήθως ένας δείκτης προς την ουρά της λίστας. Με αυτόν τον τρόπο μπορούμε να έχουμε πρόσβαση τόσο στον κόμβο της ουράς, όσο και στον κόμβο της κεφαλής, δεδομένου ότι ο δείκτης του κόμβου της ουράς δείχνει στην κεφαλή της λίστας.



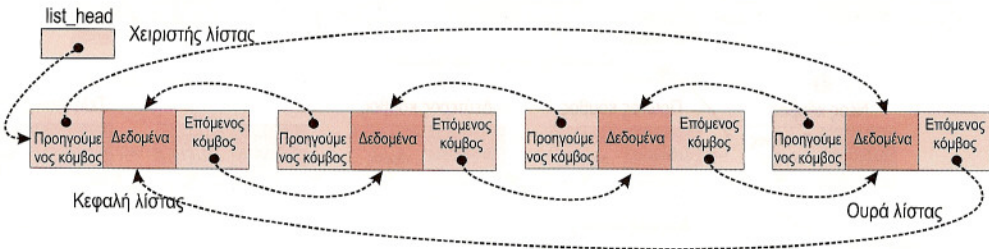
Διπλά συνδεδεμένη λίστα (double linked list)

Η διπλά συνδεδεμένη λίστα αποτελείται από κόμβους οι οποίοι, εκτός από το δείκτη για τον επόμενο κόμβο, περιέχουν και δείκτη για τον προηγούμενο κόμβο.



Κυκλικά διπλά συνδεδεμένη λίστα

Η κυκλικά διπλά συνδεδεμένη λίστα είναι ακριβώς ίδια με τη διπλά συνδεδεμένη λίστα με τη διαφορά ότι ο κόμβος της κορυφής (head node) περιέχει δείκτη προς την ουρά της λίστας για τον προηγούμενο κόμβο και ο κόμβος της ουράς (tail node) περιέχει δείκτη προς την κεφαλή της λίστας για τον επόμενο κόμβο.



Προσθήκη νέου κόμβου σε μια λίστα

Η διαδικασία που ακολουθείται όταν προσθέτουμε ένα νέο κόμβο σε μια λίστα είναι η εξής:

- Δεσμεύουμε χώρο στη μνήμη για τον νέο κόμβο (αυτό γίνεται με τις συναρτήσεις δυναμικής κατανομής μνήμης).
- Καταχωρίζουμε τα δεδομένα μέσα στο χώρο μνήμης που μόλις δεσμεύσαμε.
- Τροποποιούμε τους δείκτες της λίστας κατάλληλα ώστε να περιλαμβάνουν το νέο κόμβο που προστέθηκε.

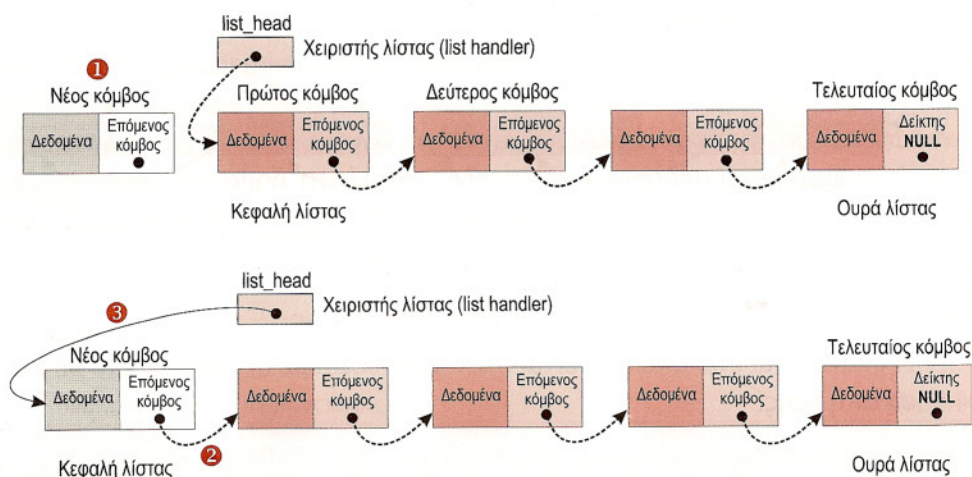
Η διαδικασία του τελευταίου σταδίου είναι διαφορετική ανάλογα με τον τύπο της συνδεδεμένης λίστας που έχουμε.

Προσθήκη νέου κόμβου σε απλά συνδεδεμένη λίστα

Αν η λίστα διαθέτει μόνο ένα χειριστή, ο οποίος δείχνει στην κεφαλή της (list head), ο νέος κόμβος προστίθεται στην αρχή της λίστας, οπότε τα βήματα που ακολουθούνται είναι:

- Δέσμευση μνήμης για το νέο κόμβο.

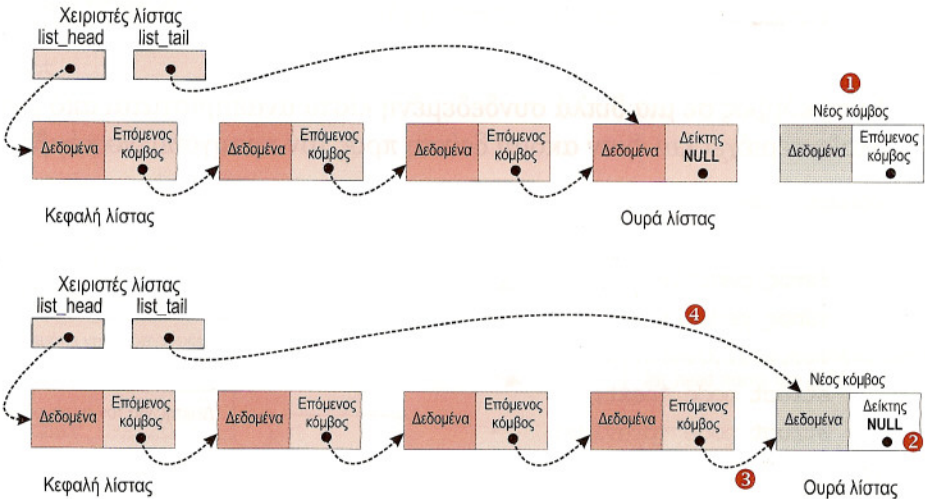
- Ο δείκτης **Επόμενος κόμβος** του νέου κόμβου πρέπει να δείχνει στην ως τώρα κεφαλή της λίστας.
- Ο χειριστής της λίστας, **list_head**, πρέπει να αλλάξει τιμή και να δείχνει στο νέο κόμβο που μόλις προστέθηκε. Αυτό επιτυγχάνεται καταχωρίζοντας στην τιμή του χειριστή της λίστας (list head) τη διεύθυνση του νέου κόμβου.



Αν η λίστα διαθέτει και χειριστή ο οποίος δείχνει στην ουρά της λίστας (list tail), ο νέος κόμβος μπορεί, με τα επόμενα βήματα, να προστεθεί στο τέλος της λίστας:

- Δέσμευση μνήμης για το νέο κόμβο
- Ο δείκτης **Επόμενος κόμβος** του νέου κόμβου πρέπει να είναι δείκτης NULL επειδή ο νέος κόμβος θα αποτελεί την ουρά της λίστας. Αυτό επιτυγχάνεται καταχωρίζοντας στην τιμή του πεδίου δείκτη του νέου κόμβου την τιμή NULL.
- Στον τελευταίο κόμβο της λίστας θα πρέπει να αλλάξει η τιμή στο πεδίο δείκτη **Επόμενος κόμβος** και να δείχνει στο νέο κόμβο που προστέθηκε. Αυτό επιτυγχάνεται καταχωρίζοντας στην τιμή του πεδίου δείκτη τη διεύθυνση του νέου κόμβου.
- Ο χειριστής της λίστας **list_tail** πρέπει να αλλάξει τιμή και να δείχνει στον νέο κόμβο που μόλις προστέθηκε. Αυτό επιτυγχάνεται καταχωρίζο-

ντας στην τιμή του χειριστή της λίστας `list_tail` τη διεύθυνση του νέου κόμβου.



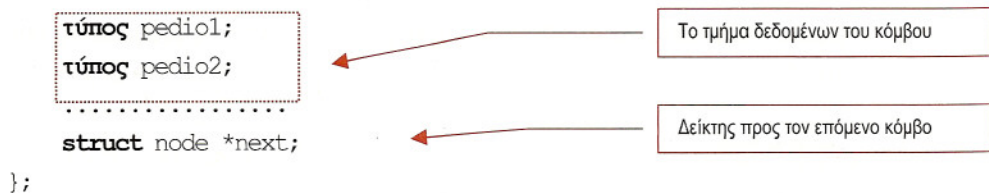
- 👉 Όταν η λίστα είναι κενή, και οι δύο χειριστές `list_head` και `list_tail` περιέχουν τιμή `NULL`.
- 👉 Ιδιαίτερος έλεγχος πρέπει να γίνει στη διαδικασία προσθήκης του πρώτου κόμβου, κατά την οποία θα πρέπει να παραλειφθεί το τρίτο από τα τέσσερα παραπάνω βήματα.
- 👉 Ανάλογα γίνεται και η προσθήκη ενός νέου κόμβου στις διπλά συνδεδεμένες λίστες αλλά και στις κυκλικά συνδεδεμένες απλές και διπλές λίστες.

Υλοποίηση συνδεδεμένης λίστας στη C

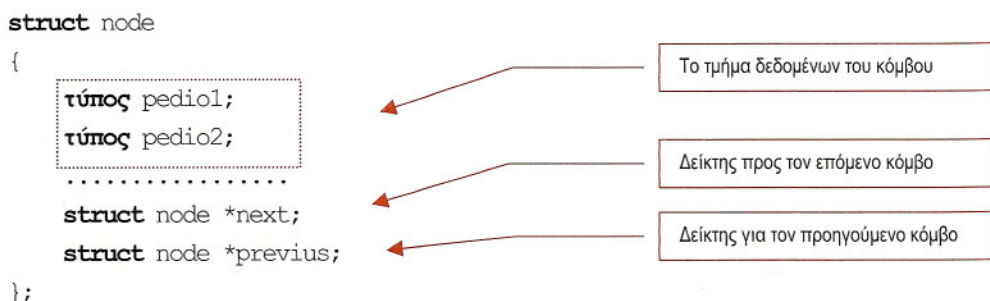
Οι συνδεδεμένες λίστες στη C υλοποιούνται με ένα συνδυασμό χρήσης δομών, μηχανισμών δυναμικής κατανομής μνήμης, και δεικτών.

Ένας κόμβος σε μια απλά συνδεδεμένη λίστα αναπαρίσταται από μία δομή στην οποία υπάρχουν πεδία για την καταχώριση των δεδομένων του κόμβου (τμήμα δεδομένων) καθώς και ένας δείκτης προς τον επόμενο κόμβο:

```
struct node
{
```

Ένας κόμβος σε μια διπλά συνδεδεμένη λίστα αναπαρίσταται από μία δομή η οποία περιέχει και έναν ακόμη δείκτη προς τον προηγούμενο κόμβο:



Ο χειριστής της λίστας είναι ένας δείκτης σε δεδομένα τύπου `node`:

```
struct node *list_handler;
```

Η δέσμευση μνήμης για ένα νέο κόμβο γίνεται με τη χρήση της `malloc()`:

```
struct node *neos;
neos = (struct node *)malloc(sizeof(struct node));
```

Η `malloc()` δεσμεύει μνήμη τόσων bytes όσο το μέγεθος του κόμβου — `sizeof(struct node)` και επιστρέφει ως τιμή ένα δείκτη στο τμήμα της μνήμης που δέσμευσε. Η μετατροπή τύπου (type casting) χρησιμοποιείται για τη μετατροπή του δείκτη που επιστρέφει η `malloc()` σε δείκτη τύπου `struct node`.

Η αποδέσμευση μνήμης την οποία κατέχει ένας κόμβος γίνεται με τη χρήση της συνάρτησης `free()`. Η

```
free(neos);
```

αποδεσμεύει το τμήμα μνήμης στο οποίο δείχνει ο δείκτης `neos` και το οποίο είχε δεσμευτεί προηγουμένως με τη συνάρτηση `malloc()`.

Το επόμενο πρόγραμμα δημιουργεί μια απλά συνδεδεμένη λίστα με δεδομένα λέξεις που δίνονται από τον χρήστη. Η διαδικασία σταματάει μόλις ο χρήστης δώσει λέξη μηδενικού μήκους (απλώς πατώντας <Enter>) και τότε εμφανίζει με τη σειρά όλα τα δεδομένα των κόμβων της λίστας.

```
struct node
{
    char data[20];
    struct node *next;
}*list_head, *neos;
void add_node_to_list();
void display_all_nodes();
```

```
main()
{
    char lex[20];
    list_head=NULL;
    while(1)
    {
        printf("Δώσε δεδομένα νέου κόμβου:");
        gets(lex);
        if(strcmp(lex,"")==0) break;
        add_node_to_list(lex);
    }
    display_all_nodes();
}
```

Η αρχική τιμή του χειριστή της λίστας `list_head` τίθεται ίση με `NULL`.

Όταν ο χρήστης δεν πληκτρολογήσει τίποτα η επαναληπτική διαδικασία σταματάει.

*/*Η παρακάτω συνάρτηση προσθέτει έναν κόμβο στην κορυφή της λίστας */*

```
void add_node_to_list(dat)
char dat[];
```

Δέσμευση ενός τμήματος μνήμης, τόσων byte όσο το μέγεθος του τύπου `struct node`.

```
{
    neos = (struct node *)malloc(sizeof(struct node));
    strcpy(neos->data, dat);
```

Καταχώριση της λέξης στο πεδίο `data` του νέου κόμβου.

```

    neos->next = list_head;
    list_head=neos;
}
/*Η παρακάτω συνάρτηση εμφανίζει όλα τα δεδομένα*/
void display_all_nodes()
{
    struct node *p;
    p=list_head;
    while (p!=NULL)
    {
        puts (p->data);
        p=p->next;
    }
}

```

Στο πεδίο next καταχωρίζεται η διεύθυνση του μέχρι στιγμής κόμβου κεφαλής της λίστας, ενώ στο χειριστή της λίστας List_head καταχωρίζεται η διεύθυνση του νέου κόμβου, ο οποίος είναι τώρα στην κορυφή της λίστας.

Στο δείκτη p καταχωρίζεται η διεύθυνση του πρώτου κόμβου της λίστας.

Εμφανίζονται τα δεδομένα του κόμβου και στο δείκτη p καταχωρίζεται η διεύθυνση του επόμενου κόμβου. Αυτό επαναλαμβάνεται μέχρι να φτάσουμε στο τέλος της λίστας (p=NULL).

Η παρακάτω συνάρτηση δέχεται ως παράμετρο ένα σύνολο χαρακτήρων και προσπαθεί να το εντοπίσει στα δεδομένα της συνδεδεμένης λίστας. Η συνάρτηση επιστρέφει ως τιμή ένα δείκτη στον κόμβο στον οποίο εντόπισε το δεδομένο, διαφορετικά (αν δεν το εντοπίσει) επιστρέφει NULL.

```

struct node *find_node(char dat[])
{
    struct node *p;
    p=list_head;
    while (p!=NULL)
    {
        if (strcmp (dat,p->data)==0) return p;
        p=p->next;
    }
    return NULL;
}

```

Στον δείκτη p καταχωρίζεται η διεύθυνση του πρώτου κόμβου της λίστας.

Ελέγχονται τα δεδομένα ενός-ενός κόμβου με τη σειρά. Μόλις εντοπιστεί το δεδομένο επιστρέφεται ως τιμή η διεύθυνση του κόμβου στον οποίο εντοπίστηκε.

Αν δεν εντοπιστεί το δεδομένο, επιστρέφεται τιμή NULL.

✋ Στα παραδείγματα στο τέλος του κεφαλαίου θα βρείτε επιπλέον κώδικα διαχείρισης συνδεδεμένων λιστών.

Η επόμενη συνάρτηση διαγράφει έναν κόμβο από την κορυφή της λίστας:


```

void delete_node()
{
    struct node *temp;
    /* αν η λίστα είναι κενή επιστρέφει αμέσως */
    if (list_head==NULL) return;
    /* αποθηκεύει προσωρινά στη μεταβλητή temp τη διεύθυνση του δεύτερου κόμβου */
    temp = list_head->next;
    free(list_head);
    /* Ο χειριστής της λίστας list_head παίρνει ως τιμή
    τη διεύθυνση του μέχρι στιγμής δεύτερου κόμβου ο οποίος
    από αυτή τη στιγμή αποτελεί την κορυφή της λίστας */
    list_head=temp;
}

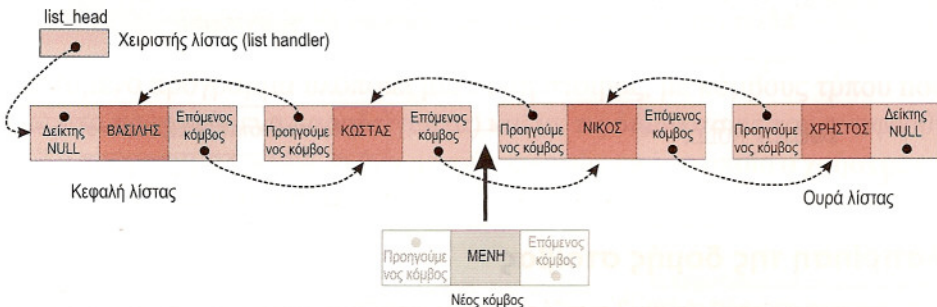
```

Απελευθερώνει το χώρο της μνήμης που καταλαμβάνει ο πρώτος κόμβος.

Διατεταγμένη συνδεδεμένη λίστα

Σε μια διατεταγμένη (ταξινομημένη) συνδεδεμένη λίστα, οι κόμβοι είναι διατεταγμένοι (ταξινομημένοι) με τη σειρά, βάσει κάποιου από τα δεδομένα της λίστας. Το δεδομένο που χρησιμοποιείται για την διάταξη των κόμβων λέγεται "κλειδί".

Η προσθήκη ενός νέου κόμβου στη λίστα δεν γίνεται πλέον στην κεφαλή ή στην ουρά της λίστας, αλλά σε συγκεκριμένο σημείο, έτσι ώστε η λίστα να διατηρηθεί ταξινομημένη. Ο πιο κατάλληλος τύπος για την υλοποίηση μιας διατεταγμένης λίστας είναι η διπλά συνδεδεμένη λίστα.



Η όλη διαδικασία υλοποίησης μιας διατεταγμένης λίστας επικεντρώνεται στον εντοπισμό της θέσης στην οποία πρέπει να παρεμβληθεί ένας νέος κόμβος.

Ας θεωρήσουμε μια διπλά συνδεδεμένη λίστα με κόμβους της παρακάτω δομής:

```
struct node
{
    char data[20];
    struct node *next;
    struct node *previous;
};
```

Το τμήμα δεδομένων αποτελείται μόνο από το πεδίο `data`, στο οποίο καταχωρίζεται η λέξη.

και με χειριστή της λίστας το δείκτη `list_head`:

```
struct node *list_head;
```

Η επόμενη συνάρτηση δέχεται ως παράμετρο το δεδομένο κλειδί του νέου κόμβου και επιστρέφει ως τιμή τη διεύθυνση του κόμβου **μετά** από τον οποίο πρέπει να παρεμβληθεί ο νέος κόμβος. Αν ο νέος κόμβος πρέπει να τοποθετηθεί στην αρχή της λίστας, η συνάρτηση επιστρέφει τιμή `NULL`.

```
struct node *find_place(char dat[])
{
    struct node *p;
    p=list_head;
    while(p!=NULL)
    {
        if(strcmp(dat,p->data)==1) return p->previous;
        p=p->next;
    }
    return NULL;
}
```

Στο δείκτη `p` καταχωρίζεται η διεύθυνση του πρώτου κόμβου της λίστας.

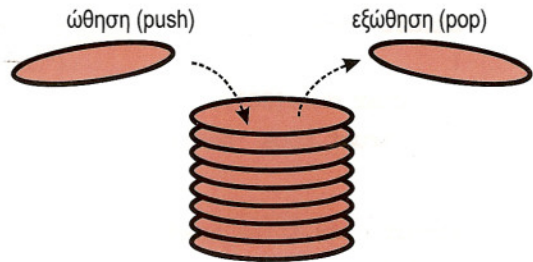
Ελέγχονται τα δεδομένα ενός-ενός κόμβου με τη σειρά. Μόλις εντοπιστεί το δεδομένο επιστρέφεται ως τιμή η διεύθυνση του κόμβου στον οποίο εντοπίστηκε.

Αν δεν εντοπιστεί το δεδομένο, επιστρέφει τιμή `NULL`.

Υλοποίηση της δομής στοίβας

Ας φανταστούμε το λαντζέρη στην κουζίνα ενός εστιατορίου, του οποίου η δουλειά είναι να πλένει τα πιάτα που του φέρνει το γκαρσόνι. Το γκαρσόνι αφήνει τα πιάτα σε μία στοίβα και κάθε φορά που φέρνει ένα νέο πιάτο το

αφήνει στην κορυφή της στοίβας. Ο λαντζέρης, για να πλύνει ένα πιάτο το παίρνει πάντα από την κορυφή της στοίβας. Η διαδικασία αυτή αποτελεί το πιο κλασικό παράδειγμα μιας δομής στοίβας.



Μια δομή στοίβας υλοποιείται με τη χρήση μιας συνδεδεμένης λίστας **LIFO** (**Last In First Out** — "Τελευταίο Μέσα Πρώτο Έξω"), στην οποία κάθε νέος κόμβος προστίθεται στην κορυφή της λίστας, αλλά και η απομάκρυνση ενός κόμβου γίνεται πάλι από την κορυφή της.

Οι βασικές διαδικασίες που χρησιμοποιούνται για το χειρισμό μιας στοίβας είναι η **ώθηση** (push) και η **εξαγωγή** ή **εξώθηση** (pop):

push() Προσθέτει ένα στοιχείο στην κορυφή της στοίβας.

pop() Εξάγει (απομακρύνει) το πιο πρόσφατο στοιχείο από την κορυφή της στοίβας.

Άλλες διαδικασίες που μπορεί επίσης να χρησιμοποιηθούν είναι οι:

gettop() Επιστρέφει το στοιχείο της κορυφής της στοίβας χωρίς όμως να το απομακρύνει.

isempty() Καθορίζει αν η στοίβα είναι κενή, δηλαδή αν δεν περιέχει κανένα στοιχείο.

Το επόμενο πρόγραμμα υλοποιεί μια δομή στοίβας, με κόμβους τύπου `node` (όπως στα προηγούμενα παραδείγματα) και με τις διαδικασίες που αναφέρθηκαν. Το πρόγραμμα εμφανίζει ένα μενού επιλογών από το οποίο ο χρήστης επιλέγει τη διαδικασία που θέλει.

```
/* stack.c */
```

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

Το τμήμα δεδομένων αποτελείται μόνο από το πεδίο `data`, στο οποίο καταχωρίζεται η λέξη.


```
struct node
{
    char data[20];
    struct node *next;
};

struct node *list_head;
void pop();
int push();
void gettop();
int isempty();
main()
{
```

```
    char ch, lex[20];
    list_head=NULL;
    while(1)
```

Ο χειριστής της στοίβας list_head παίρνει αρχική τιμή NULL, δείχνοντας έτσι σε μια κενή στοίβα.

```
    {
        printf("0->Εξοδος 1->Push 2->Pop 3->Εμφάνιση:\n");
        ch=getch();
        switch(ch)
```

Εμφανίζει το μενού επιλογών και περιμένει την επιλογή του χρήστη.

```
        case '0':
            exit(0);
```

```
        case '1':
            printf("Δωσε λέξη:");
            gets(lex);
            if(push(lex)==0)
```

Καλείται η push() και ελέγχεται η τιμή που επιστρέφει. Αν είναι 0 σημαίνει ότι δεν υπάρχει αρκετή μνήμη για δέσμευση.

```
                puts("Δεν υπάρχει διαθέσιμη μνήμη");
```

```
            break;
```

```
        case '2':
            pop();
            break;
```

```
        case '3':
            gettop();
            break;
```

default:

```
puts("Λάθος πλήκτρο");
break;
```

```
}
```

```
}
```

```
}
```

/*Η επόμενη συνάρτηση προσθέτει έναν κόμβο στην κορυφή της στοίβας */

int push(dat)

char dat[];

```
{
```

```
    struct node *neos;
```

```
    neos = (struct node *)malloc(sizeof(struct node));
```

```
    if(neos==NULL) return 0;
```

```
    strcpy(neos->data, dat);
```

```
    neos->next = list_head;
```

```
    list_head=neos;
```

```
    return 1;
```

```
}
```

Δέσμευση ενός τμήματος μνήμης, τόσων bytes όσο το μέγεθος του τύπου struct node.

Αν δεν υπάρχει αρκετή μνήμη για δέσμευση επιστρέφει τιμή 0.

Καταχώριση της λέξης στο πεδίο data του νέου κόμβου.

/*Η επόμενη συνάρτηση απομακρύνει έναν κόμβο από την κορυφή της στοίβας */

void pop()

```
{
```

```
    struct node *temp;
```

```
    if (isempty())
```

```
    {
```

```
        puts("Κενή στοίβα");
```

```
        return;
```

```
    }
```

```
    temp = list_head->next;
```

```
    free(list_head);
```

```
    list_head=temp;
```

```
}
```

Απελευθερώνει το χώρο μνήμης που καταλαμβάνει ο πρώτος κόμβος.

int isempty()

```
{
```

```
    if(list_head==NULL) return 1;
```

```

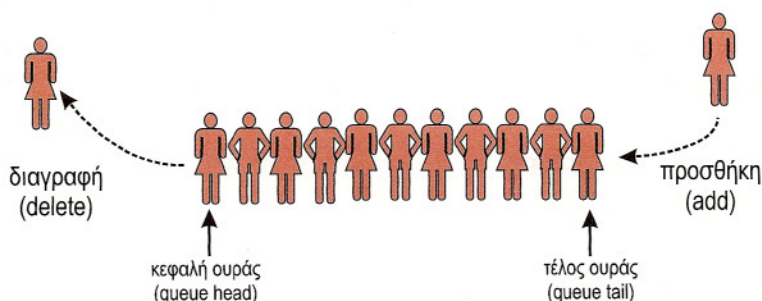
    return 0;
}

/*Η επόμενη συνάρτηση εμφανίζει τα δεδομένα του κορυφαίου στοιχείου της στοίβας */
void gettop()
{
    if (isempty())
        puts("Η στοίβα είναι κενή");
    else
        puts(list_head->data);
}

```

Υλοποίηση της δομής ουράς

Ας φανταστούμε μια σειρά πελατών μιας τράπεζας, που περιμένουν στην ουρά για να εξυπηρετηθούν από τον υπάλληλο του ταμείου. Κάθε νέος πελάτης που έρχεται προστίθεται στο τέλος της ουράς, ενώ ο υπάλληλος εξυπηρετεί πάντα τον πρώτο πελάτη της ουράς.



Η παραπάνω διαδικασία αποτελεί ένα παράδειγμα μιας δομής ουράς. Μια δομή ουράς υλοποιείται με τη χρήση μιας συνδεδεμένης λίστας **FIFO** (First In First Out —Πρώτο Μέσα Πρώτο Έξω) στην οποία κάθε νέος κόμβος προστίθεται στο τέλος της λίστας, αλλά η απομάκρυνση ενός κόμβου γίνεται από την κορυφή της λίστας.

Η υλοποίηση μιας λίστας FIFO γίνεται με τη χρήση απλής συνδεδεμένης λίστας η οποία όμως διαθέτει δύο χειριστές, έναν για την κεφαλή της λίστας

(*list_head*) και έναν για την ουρά της λίστας (*list_tail*). Περιγράψαμε τη διαδικασία προσθήκης νέων κόμβων σε μια τέτοια λίστα στην παράγραφο "Προσθήκη νέου κόμβου σε απλά συνδεδεμένη λίστα".

Οι βασικές διαδικασίες που χρησιμοποιούνται για το χειρισμό μιας ουράς είναι η **προσθήκη** (add) και η **διαγραφή** (delete):

add()	Προσθέτει ένα στοιχείο στο τέλος της ουράς.
delete()	Απομακρύνει το πιο πρόσφατο στοιχείο από την κεφαλή της ουράς.

Άλλες διαδικασίες που μπορεί επίσης να χρησιμοποιηθούν είναι οι:

gettop()	Επιστρέφει το στοιχείο της κεφαλής της ουράς χωρίς όμως να το απομακρύνει.
isempty()	Ελέγχει αν η ουρά είναι κενή, δηλαδή αν δεν περιέχει κανένα στοιχείο.

Το επόμενο πρόγραμμα υλοποιεί μία δομή ουράς, με κόμβους τύπου node (όπως στα προηγούμενα παραδείγματα), με τις διαδικασίες που αναφέρθηκαν. Το πρόγραμμα εμφανίζει ένα μενού επιλογών από το οποίο ο χρήστης επιλέγει τη διαδικασία που θέλει.

```
/* queue.c */
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <malloc.h>
```

```
struct node
{
    char data[20];
    struct node *next;
};

struct node *list_head, *list_tail;
void delete();
```

Το τμήμα δεδομένων αποτελείται μόνο από το πεδίο data, στο οποίο καταχωρίζεται η λέξη.

```

int add();
void gettop();
int isempty();

main()
{
    char ch, lex[20];
    list_head=list_tail=NULL;
    while(1)
    {
        printf("0->Εξοδος 1->Add 2->Delete 3->Εμφάνιση:\n");
        ch=getch();
        switch(ch)
        {
            case '0':
                exit(0);
            case '1':
                printf("Δωσε λέξη:");
                gets(lex);
                if(add(lex)==0)
                    puts("Δεν υπάρχει διαθέσιμη μνήμη");
                break;
            case '2':
                delete();
                break;
            case '3':
                gettop();
                break;
            default:
                puts("Λάθος πλήκτρο");
                break;
        }
    }
}

```

Οι χειριστές της ουράς list_head και list_tail παίρνουν αρχική τιμή NULL, δείχνοντας έτσι σε μια κενή ουρά.

Εμφανίζεται το μενού επιλογών και αναμένεται η επιλογή του χρήστη.

Καλείται η add() και ελέγχεται η τιμή που επιστρέφει. Αν είναι 0 σημαίνει ότι δεν υπάρχει αρκετή μνήμη για δέσμευση.

/*Η επόμενη συνάρτηση προσθέτει έναν κόμβο στο τέλος της ουράς*/

```
int add(dat)
char dat[];
{
    struct node *neos;
    neos = (struct node *)malloc(sizeof(struct node));
    if(neos==NULL) return 0;
    strcpy(neos->data, dat);
    if(list_tail!=NULL)
        list_tail->next=neos;
    neos->next = NULL;
    list_tail=neos;
    if(list_head==NULL)
        list_head=list_tail;
    return 1;
}
```

Δέσμευση ενός τμήματος μνήμης, τόσων bytes όσο το μέγεθος του τύπου struct node.

Αν δεν υπάρχει αρκετή μνήμη για δέσμευση επιστρέφεται τιμή 0.

Εάν η ουρά είναι κενή, οι χειριστές list_head και list_tail θα δείχνουν και οι δύο στο νέο κόμβο που προστίθεται.

/*Η επόμενη συνάρτηση απομακρύνει έναν κόμβο από την κεφαλή της ουράς*/

```
void delete()
{
    struct node *temp;
    if (isempty())
    {
        puts("Κενή ουρά");
        return;
    }
    temp = list_head->next;
    free(list_head);
    list_head=temp;
}

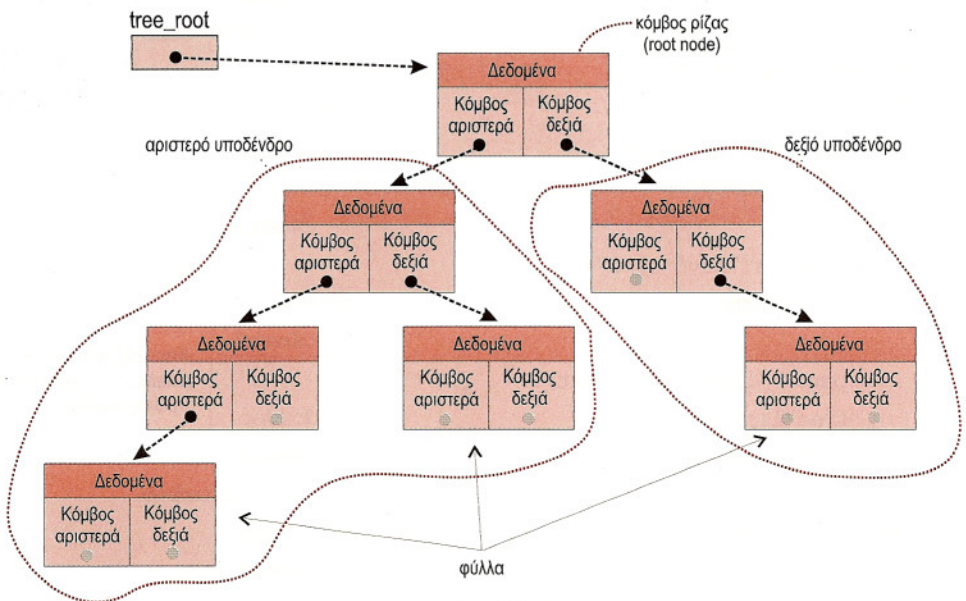
int isempty()
{
    if(list_head==NULL) return 1;
    return 0;
}
```

Απελευθερώνεται ο χώρος μνήμης που καταλαμβάνει ο πρώτος κόμβος.


```
/*Η επόμενη συνάρτηση εμφανίζει τα δεδομένα του κορυφαίου στοιχείου της ουράς*/
void gettop()
{
    if (isempty())
        puts("Η ουρά είναι κενή");
    else
        puts(list_head->data);
}
```

Δυαδικά δένδρα

Ενα δυαδικό δένδρο αποτελείται από έναν κόμβο που αποτελεί τη **ρίζα** του δένδρου (root node), ένα **υποδένδρο αριστερά** και ένα **υποδένδρο δεξιά**. Κάθε υποδένδρο είναι και αυτό ένα δυαδικό δένδρο με τον κόμβο ρίζας του, το αριστερό και το δεξιό του υποδένδρο.



Κάθε κόμβος του δυαδικού δένδρου πέρα από το τμήμα δεδομένων περιέχει ένα δείκτη προς το αριστερό υποδένδρο του και ένα δείκτη προς το δεξιό. Οι κόμβοι οι οποίοι δεν έχουν υποδένδρο ονομάζονται **φύλλα** του δυαδικού δένδρου.

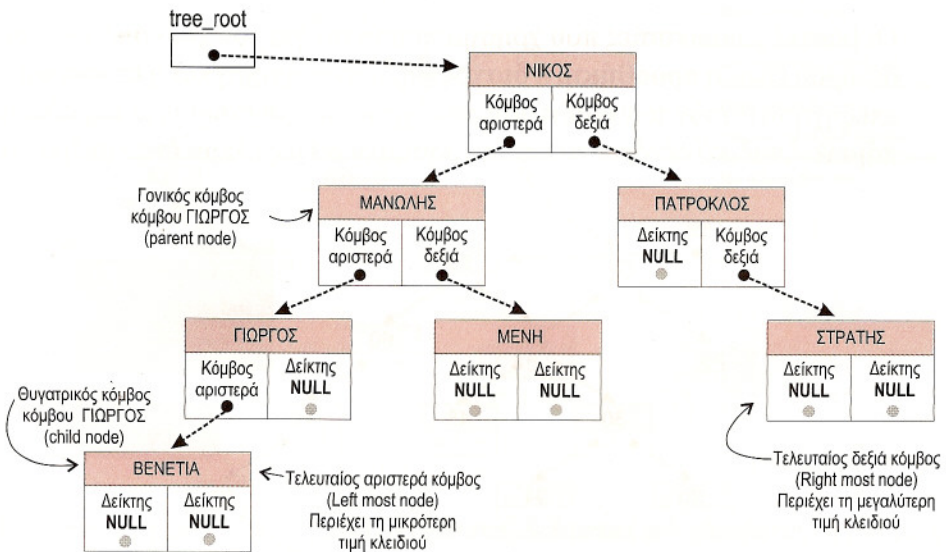
✎ Στα δυαδικά δένδρα, όπως και στις συνδεδεμένες λίστες, έχουμε πρόσβαση μέσω ενός δείκτη (στο παραπάνω σχήμα είναι ο `tree_root`), ο οποίος περιέχει την διεύθυνση του κόμβου ρίζας του δυαδικού δένδρου (root node).

Η λογική με την οποία είναι διατεταγμένοι οι κόμβοι ενός δυαδικού δένδρου είναι η εξής:

- Ένα από τα δεδομένα του κόμβου αποτελεί το **κλειδί** βάσει του οποίου θα είναι διατεταγμένο το δυαδικό δένδρο.
- Ο κόμβος στον οποίο δείχνει ο **δείκτης αριστερά** έχει τιμή κλειδιού μικρότερη (αριθμητικά ή αλφαβητικά) από την τιμή του κλειδιού του τρέχοντος κόμβου.
- Ο κόμβος στον οποίο δείχνει ο **δείκτης δεξιά** έχει τιμή κλειδιού μεγαλύτερη ή ίση από την τιμή του κλειδιού του τρέχοντος κόμβου.

Το παράδειγμα που ακολουθεί δείχνει ένα δυαδικό δένδρο στο οποίο είναι καταχωρισμένα τα στοιχεία κάποιων ατόμων. Το κλειδί που χρησιμοποιείται για τη διάταξη του συγκεκριμένο δυαδικού δένδρου είναι το πεδίο **Όνομα**.

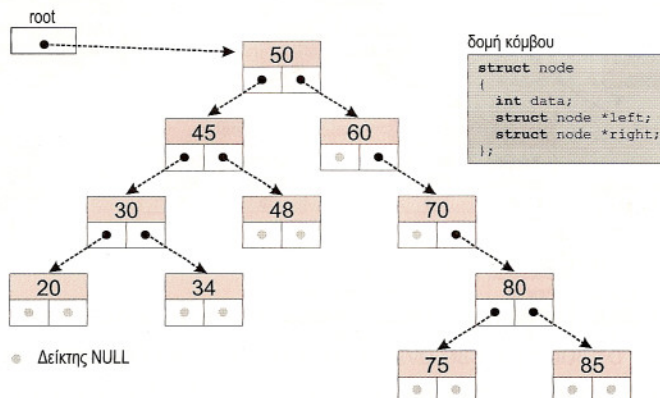
Ο πρώτος κόμβος που προστέθηκε στο δυαδικό δένδρο είναι ο κόμβος με τιμή κλειδιού ΝΙΚΟΣ, ο οποίος και αποτελεί την ρίζα του δυαδικού δένδρου.



- ☞ Κάθε κόμβος ενός δυαδικού δένδρου (εκτός από τον κόμβο ρίζας), έχει ένα **γονικό** κόμβο (parent node). Ο γονικός κόμβος είναι ο κόμβος του προηγούμενου επιπέδου ο οποίος είναι συνδεδεμένος με τον κόμβο στον οποίο αναφερόμαστε.
- ☞ Κάθε κόμβος ενός δυαδικού δένδρου μπορεί να έχει μέχρι δύο **θυγατρικούς** κόμβους (child nodes). Οι θυγατρικοί κόμβοι είναι οι κόμβοι του αμέσως χαμηλότερου επιπέδου, με τους οποίους είναι συνδεδεμένος ο κόμβος στον οποίο αναφερόμαστε.
- ☞ Κάθε κόμβος ενός δυαδικού δένδρου (εκτός από τον κόμβο ρίζας) είναι ο αριστερός ή ο δεξιός θυγατρικός κόμβος του **γονικού** κόμβου.
- ☞ Ο **τελευταίος αριστερά** κόμβος (left most node) ενός δένδρου (ή υποδένδρου) είναι ο κόμβος με τη μικρότερη τιμή κλειδιού στο συγκεκριμένο δένδρο (ή υποδένδρο).
- ☞ Ο **τελευταίος δεξιά** κόμβος (right most node) ενός δένδρου (ή υποδένδρου) είναι ο κόμβος με τη μεγαλύτερη τιμή κλειδιού στο συγκεκριμένο δένδρο (ή υποδένδρο).
- ☞ Να θυμηθούμε ότι **κάθε κόμβος** στο δεξιό υποδένδρο ενός δυαδικού δένδρου έχει τιμή μεγαλύτερη από **κάθε κόμβο** στο αριστερό του υποδένδρο.

Χειρισμός ενός δυαδικού δένδρου

Οι βασικές διαδικασίες που χρησιμοποιούνται για τον χειρισμό ενός δυαδικού δένδρου είναι η **προσθήκη**, η **διαγραφή**, η **αναζήτηση** και η **επίσκεψη**. Για την επεξήγηση αυτών των διαδικασιών θα χρησιμοποιήσουμε για παράδειγμα το επόμενο δυαδικό δένδρο, το οποίο χρησιμοποιεί ως κλειδί έναν ακέραιο αριθμό.



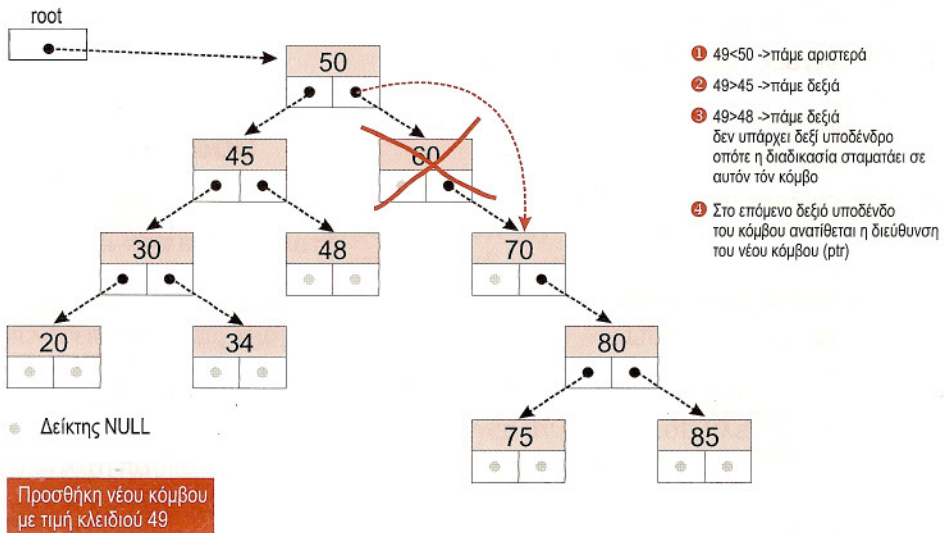
Προσθήκη νέου κόμβου

Η διαδικασία προσθήκης ενός νέου κόμβου σε ένα δυαδικό δένδρο αναλύεται στα επόμενα βήματα:

Αν το δένδρο είναι κενό ($\text{root}=\text{NULL}$), τότε πρόσθεσε το νέο κόμβο στην κορυφή και ανάθεσε στον χειριστή του δένδρου (root) τη διεύθυνση του νέου κόμβου ($\text{root}=\text{ptr}$).

Στην περίπτωση που το δένδρο δεν είναι κενό, έλεγξε την τιμή κλειδιού του νέου κόμβου με το κλειδί της ρίζας. Αν είναι μικρότερη, προχώρησε στη ρίζα του αριστερού υποδένδρου, διαφορετικά στη ρίζα του δεξιού υποδένδρου. Επανάλαβε την ίδια διαδικασία μέχρι να φτάσεις σε κόμβο που να μην οδηγεί (ούτε αριστερά ούτε δεξιά) σε υποδένδρο.

Ανάθεσε ως τιμή του αριστερού ή του δεξιού υποδένδρου του κόμβου (ανάλογα με το αποτέλεσμα του προηγούμενου σταδίου) την διεύθυνση (ptr) του νέου κόμβου.



Η συνάρτηση `newnode()`, που ακολουθεί, δεσμεύει την απαραίτητη μνήμη για ένα νέο κόμβο και καταχωρίζει την τιμή της παραμέτρου της (`num`) σαν τιμή

κλειδιού του νέου κόμβου. Επιστρέφει ως τιμή τη διεύθυνση της μνήμης που δέσμευσε.

```
struct node *newnode(int num)
{
    struct node *new;
    new=malloc(sizeof(struct node));
    new->data = num;
    new->left = NULL;
    new->right = NULL;
    return(new);
}
```

Η επόμενη συνάρτηση `insert()` χρησιμοποιεί τη `newnode()` για να δημιουργήσει ένα νέο κόμβο και να τον προσθέσει στο δυαδικό δένδρο του παραδείγματος.

```
struct node *insert(int num)
{
    /* ο δείκτης next δείχνει στον επόμενο κόμβο και ο current στον τρέχοντα */
    struct node *next, *current, *ptr;
    /* η τιμή 1 ή 0 στη μεταβλητή isleft δείχνει ότι στον τρέχοντα κόμβο έχουμε φτάσει
    από αριστερό ή δεξιό παρακλαδί του δένδρου αντίστοιχα */
    int isleft;
    current=root;
    /* η newnode() δημιουργεί ένα νέο κόμβο με τιμή
    κλειδιού num και επιστρέφει τη διεύθυνσή του */
    ptr=newnode(num);
    if (root == NULL)
    {
        root=ptr;
        return ptr;
    }
    while(1)
    {
        if(num < current->data)
        {
            isleft=1;
            current=current->left;
        }
        else
        {
            isleft=0;
            current=current->right;
        }
        if(current==NULL)
        {
            if(isleft)
                current->left=ptr;
            else
                current->right=ptr;
            return ptr;
        }
    }
}
```

Θέσε ως τιμή του τρέχοντος κόμβου τον κόμβο ρίζας του δένδρου.

Αν το δένδρο είναι κενό, ανάθεσε απλώς στο χειριστή του δένδρου (root) τη διεύθυνση του νέου κόμβου.

Αν το κλειδί του νέου κόμβου είναι μικρότερο από του τρέχοντος προχώρησε στον επόμενο κόμβο προς τα αριστερά, αλλιώς στον επόμενο κόμβο προς τα δεξιά.

```

        next = current->left;
        isleft=1;
    }
    else
    {
        next = current->right;
        isleft=0;
    }
    if (next == NULL)
    {
        if (isleft)
            current->left=ptr;
        else
            current->right=ptr;
        return ptr;
    }
    current=next;
}

```

Αν δεν υπάρχει επόμενος κόμβος, ανάθεσε στον αριστερό ή στο δεξιά δείκτη του τρέχοντος κόμβου (ανάλογα με το isleft) τη διεύθυνση του νέου κόμβου. Να επιστρέψεις ως τιμή τη διεύθυνση του νέου κόμβου.

Κάνε τρέχοντα κόμβο τον επόμενο και επανάλαβε τη διαδικασία.

Αναζήτηση κόμβου

Η διαδικασία αναζήτησης ενός κόμβου, με βάση την τιμή του κλειδιού του, αναλύεται στα εξής βήματα:

Αν το δένδρο είναι κενό (root=NULL) τότε προφανώς ο κόμβος δεν υπάρχει και να επιστρέψεις τιμή NULL.

Στην περίπτωση που το δένδρο δεν είναι κενό, έλεγξε την τιμή κλειδιού του νέου κόμβου με το κλειδί της ρίζας. Έλεγξε αν η τιμή του κλειδιού του κόμβου είναι αυτή που αναζητούμε. Αν είναι, να επιστρέψεις ως τιμή τη διεύθυνση του κόμβου που την εντοπίσαμε. Αν είναι μικρότερη, προχώρησε στη ρίζα του αριστερού υποδένδρου, διαφορετικά στη ρίζα του δεξιού υποδένδρου. Επανάλαβε την ίδια διαδικασία μέχρι να φτάσεις σε κόμβο που να μη σε οδηγή (αριστερά ή δεξιά) σε κάποιο υποδένδρο γεγονός που σημαίνει ότι η τιμή που αναζητούμε δεν εντοπίστηκε.

Η συνάρτηση `find()` αναζητά στο δυαδικό δένδρο έναν κόμβο με τιμή κλειδιού `num`. Αν τον εντοπίσει, επιστρέφει ως τιμή τη διεύθυνσή του, διαφορετικά επιστρέφει `NULL`.

```
struct node *find(int key)
{
```

```
    struct node *current;
```

```
    current=root;
```

```
    while(current->data != key)
```

```
    {
```

```
        if(key < current->data)
```

```
            current = current->left;
```

```
        else
```

```
            current = current->right;
```

```
        if(current == NULL)
```

```
            return NULL;
```

```
    }
```

```
    return current;
```

```
}
```

Όρισε ως τρέχοντα κόμβο τον κόμβο ρίζας (root).

Ενώσω ο τρέχων κόμβος δεν έχει το κλειδί που αναζητάμε, αν είναι μικρότερο κάνε τρέχοντα κόμβο τον αριστερό, διαφορετικά το δεξιό. Επανάλαβε τη διαδικασία.

Αν δεν υπάρχει επόμενος κόμβος, το κλειδί δε βρέθηκε. Να επιστρέψεις `NULL`.

Το κλειδί βρέθηκε. Να επιστρέψεις τη διεύθυνση του κόμβου στον οποίο βρέθηκε.

Επίσκεψη ενός δυαδικού δέντρου

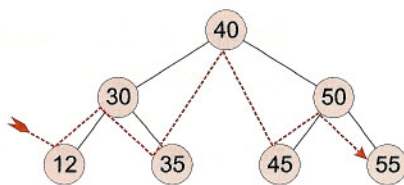
Η διαδικασία αυτή συνίσταται στην επίσκεψη κάθε κόμβου του δυαδικού δένδρου από μία φορά. Κατά την επίσκεψη του κόμβου μπορεί να γίνει επεξεργασία ή εμφάνιση των δεδομένων του. Η διαδικασία της "επίσκεψης" ονομάζεται **tree traversal**. Οι διαδικασίες επίσκεψης των κόμβων ενός δυαδικού δένδρου είναι πολύ απλές αλλά καθαρά αναδρομικές. Ανάλογα με τη σειρά με την οποία γίνεται η επίσκεψη στους κόμβους έχουμε τρεις διαφορετικές διαδικασίες.

Επίσκεψη κατά σειρά (in-order)

Η πιο συνηθής διαδικασία επίσκεψης είναι η επίσκεψη "κατά σειρά" (in-order traversal), όπου οι κόμβοι δέχονται την επίσκεψη στη διατεταγμένη τους σειρά. Πρώτα γίνεται η επίσκεψη στον κόμβο με τη μικρότερη τιμή κλειδιού και τελευταία στον κόμβο με τη μεγαλύτερη τιμή.

Η αναδρομική διαδικασία **in-order** περιλαμβάνει τα επόμενα βήματα:

- Κάλεσε αναδρομικά την ίδια διαδικασία για να επισκεφτείς το αριστερό υποδένδρο.
- Επισκέψου τον κόμβο.
- Κάλεσε αναδρομικά την ίδια διαδικασία για να επισκεφτείς το δεξιό υποδένδρο.



Όπως σε κάθε αναδρομική διαδικασία, πρέπει και εδώ να υπάρχει μία μη αναδρομική περίπτωση. Αυτό συμβαίνει στην περίπτωση που η διεύθυνση του κόμβου είναι NULL, οπότε η διαδικασία επιστρέφει αμέσως. Η επόμενη αναδρομική συνάρτηση υλοποιεί τη διαδικασία in-order για την εμφάνιση των δεδομένων όλων των κόμβων ενός δυαδικού δένδρου με αύξουσα σειρά κλειδιού.

```
void in_order_display(struct node *ptr)
```

```
{
```

```
    if (ptr == NULL) return;
```

```
    in_order_display(ptr->left);
```

```
    /*εμφάνιση των δεδομένων του κόμβου */
```

```
    printf("%d ", ptr->data);
```

```
    in_order_display(ptr->right);
```

```
}
```

Μη αναδρομική περίπτωση.

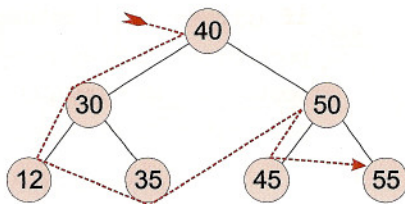
Αναδρομική κλήση για το αριστερό υποδένδρο.

Αναδρομική κλήση για το δεξιό υποδένδρο.

Επίσκεψη κατά προδιάταξη (pre-order)

Η αναδρομική διαδικασία κατά προδιάταξη (pre-order traversal) περιλαμβάνει τα εξής βήματα:

- Επισκέψου τον κόμβο.
- Κάλεσε αναδρομικά την ίδια διαδικασία για να επισκεφτείς το αριστερό υποδένδρο.
- Κάλεσε αναδρομικά την ίδια διαδικασία για να επισκεφτείς το δεξιό υποδένδρο.



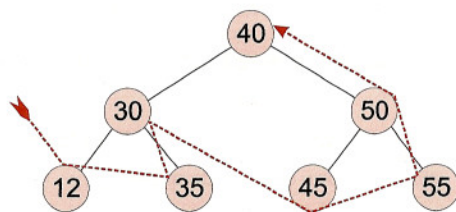
Η επόμενη συνάρτηση υλοποιεί τη διαδικασία προδιάταξης (pre-order) για την εμφάνιση των δεδομένων όλων των κόμβων ενός δυαδικού δένδρου.

```
void pre_order_display(struct node *ptr)
{
    if (ptr == NULL) return;
    printf("%d ", ptr->data);
    pre_order_display(ptr->left);
    pre_order_display(ptr->right);
}
```

Επίσκεψη κατά μεταδιάταξη (post-order)

Η αναδρομική διαδικασία κατά μεταδιάταξη (post-order traversal) περιλαμβάνει τα επόμενα βήματα:

- Κάλεσε αναδρομικά την ίδια διαδικασία για να επισκεφτείς το αριστερό υποδένδρο.
- Κάλεσε αναδρομικά την ίδια διαδικασία για να επισκεφτείς το δεξιό υποδένδρο.
- Επισκέψου τον κόμβο.



Η επόμενη συνάρτηση υλοποιεί τη διαδικασία μεταδιάταξης (post-order) για την εμφάνιση των δεδομένων όλων των κόμβων ενός δυαδικού δένδρου.

```
void post_order_display(struct node *ptr)
{
    if (ptr == NULL) return;
    post_order_display(ptr->left);
    post_order_display(ptr->right);
    printf("%d ", ptr->data);
}
```

Διαγραφή κόμβου από το δυαδικό δέντρο

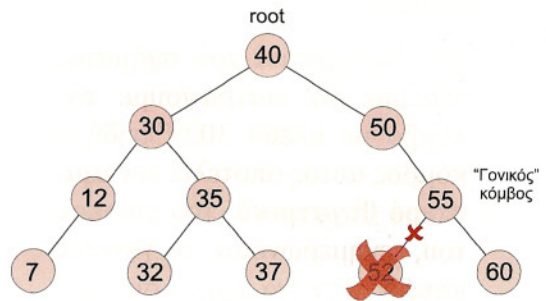
Η διαδικασία διαγραφής ενός κόμβου από ένα δυαδικό δένδρο, είναι αρκετά πολύπλοκη και μπορεί να περιλαμβάνει τέσσερις διαφορετικές περιπτώσεις:

- Ο κόμβος να είναι απλό "φύλλο" του δένδρου, δηλαδή να μην έχει κανένα θυγατρικό κόμβο.
- Ο κόμβος να έχει μόνο ένα θυγατρικό κόμβο (αριστερά ή δεξιά).
- Ο κόμβος να έχει δύο θυγατρικούς κόμβους (και αριστερά και δεξιά).
- Ο κόμβος να είναι η ρίζα (root) του δυαδικού δένδρου.

Διαγραφή κόμβου χωρίς θυγατρικούς κόμβους

Είναι η πιο απλή περίπτωση διαγραφής, κατά την οποία απλώς ενημερώνεται ο **γονικός** κόμβος ότι δεν έχει πλέον το συγκεκριμένο **θυγατρικό** κόμβο.

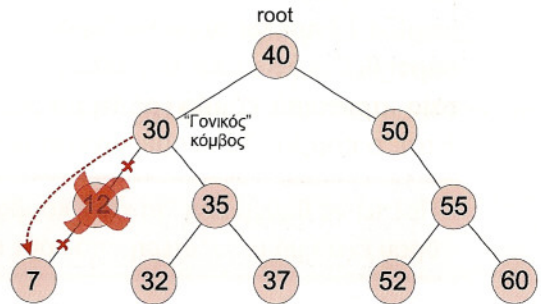
Αυτό γίνεται με την αντικατάσταση του δείκτη που έδειχνε στον συγκεκριμένο κόμβο με το NULL.



Διαγραφή κόμβου με ένα θυγατρικό κόμβο

Σε αυτή την περίπτωση, ο **γονικός** κόμβος ενημερώνεται ώστε να δείχνει κατευθείαν στο **θυγατρικό** κόμβο που διαγράφεται.

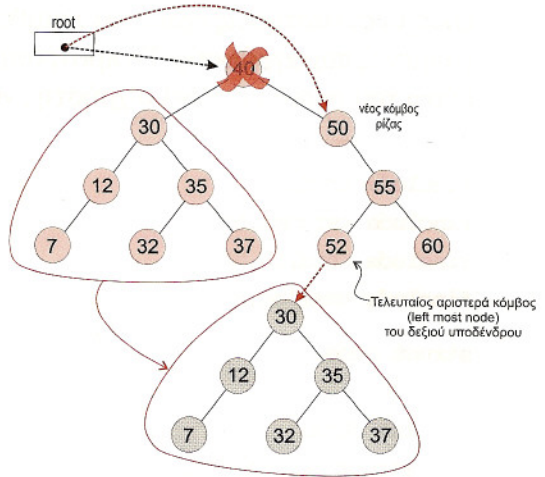
Ανάλογα αν ο κόμβος που διαγράφεται είναι ο αριστερός ή ο δεξιός θυγατρικός κόμβος του **γονικού** του, ενημερώνεται και ο κατάλληλος δείκτης ώστε να αποκτήσει τη διεύθυνση του **θυγατρικού** του κόμβου που διαγράφεται.




Διαγραφή κόμβου με δύο θυγατρικούς κόμβους

Αυτή είναι η πιο πολύπλοκη περίπτωση διαγραφής. Στην περίπτωση αυτή, ενημερώνεται ο **γονικός** κόμβος ώστε να δείχνει κατευθείαν σε έναν από τους θυγατρικούς κόμβους που διαγράφεται.

Ας θεωρήσουμε τη συνηθισμένη περίπτωση όπου ο κόμβος ρίζας έχει δύο θυγατρικούς. Αν διαγραφεί ο κόμβος ρίζας, νέα ρίζα θα πρέπει να γίνει κάποιος από τους θυγατρικούς κόμβους της υπάρχουσας. Αν επιλέξουμε ως νέα ρίζα το δεξιό θυγατρικό κόμβο, θα πρέπει ολόκληρο το αριστερό υποδένδρο να "προσαρτηθεί" κάτω από τον τελευταίο αριστερά κόμβο του δεξιού υποδένδρου. Το αντίστροφο θα πρέπει να γίνει αν επιλέξουμε ως νέα ρίζα τον αριστερό θυγατρικό κόμβο της ρίζας που διαγράφουμε.



 Και στις δύο περιπτώσεις πρέπει να ενημερωθεί και ο χειριστής του δένδρου (δείκτης root), ώστε να δείχνει στο νέο κόμβο ρίζας.

Υλοποίηση της δομής δυαδικού δένδρου

Στην παράγραφο αυτή παρατίθεται ένα ολοκληρωμένο πρόγραμμα διαχείρισης ενός δυαδικού δένδρου. Οι βασικές συναρτήσεις που χρησιμοποιούνται είναι η **προσθήκη** (insert), η **διαγραφή** (rm), η **αναζήτηση** (find), και η **εμφάνιση** (display).

insert()	Προσθέτει ένα νέο κόμβο στο δυαδικό δένδρο.
rm()	Απομακρύνει έναν κόμβο από το δυαδικό δένδρο και, αν είναι απαραίτητο, το ανασυντάσσει.
find()	Εντοπίζει έναν κόμβο στο δυαδικό δένδρο με βάση την τιμή του κλειδιού του.
display()	Επισκέπτεται όλους τους κόμβους του δυαδικού δένδρου κατά τη σειρά του κλειδιού τους (in-order traversal) και εμφανίζει την τιμή του κλειδιού.

Όλες οι συναρτήσεις έχουν υλοποιηθεί σύμφωνα με όσα αναφέραμε στις προηγούμενες παραγράφους. Το πρόγραμμα εμφανίζει αρχικά ένα μενού επιλογών μέσω του οποίου μπορεί ο χρήστης να επιλέξει την επιθυμητή λειτουργία.

```
//binary_tree.c
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <malloc.h>

struct node
{
    int data;
    struct node *left;
    struct node *right;
};

struct node *root;
struct node *newnode();
struct node *find();
struct node *insert();
void display();
struct node *find_left_most();
struct node *find_right_most();

main()
{
    char ch;
    int a;
    struct node *new;
    root=NULL;
    while(1)
    {
        printf("0->Εξοδος 1->Προσθήκη 2->Εύρεση ");
        printf("3->Διαγραφή 4->Εμφάνιση:\n");
```

```

ch=getch();
switch(ch)
{
    case '0':
        exit(0);
    case '1':
        printf("num:");
        scanf("%d",&a);
        new=insert(a);
        if (root==NULL) root=new;
        if(new==NULL)
            puts("Δεν υπάρχει αρκετή μνήμη");
        break;
    case '2':
        printf("num:");
        scanf("%d",&a);
        new=find(a);
        if(new!=NULL)
            printf("Βρέθηκε\n");
        else
            printf("Δεν υπάρχει\n");
        break;
    case '3':
        printf("num:");
        scanf("%d",&a);
        rm(a);
        break;
    case '4':
        display(root);
        puts("");
        break;
    default:
        puts("Λάθος πλήκτρο");
        break;
}

```

```
    }  
}  
  
struct node *newnode(int num)  
{  
    struct node *neos;  
    neos=malloc(sizeof(struct node));  
    neos->data = num;  
    neos->left = NULL;  
    neos->right = NULL;  
    return (neos);  
}  
  
void display(struct node *ptr)  
{  
    if (ptr == NULL) return;  
    display(ptr->left);  
    printf("%d ", ptr->data);  
    display(ptr->right);  
}  
  
struct node *find(int key)  
{  
    struct node *current;  
    current=root;  
    while(current->data != key)  
    {  
        if(key < current->data)  
            current = current->left;  
        else  
            current = current->right;  
        if(current == NULL)  
            return NULL;  
    }  
    return current;  
}
```



```

struct node *insert(int num)
{
    struct node *next,*current,*ptr;
    int isleft;
    next=current=root;
    ptr=newnode(num);
    if (root == NULL)
    {
        return ptr;
    }
    while(1)
    {
        if(num < current->data)
        {
            next = current->left;
            isleft=1;
        }
        else
        {
            next = current->right;
            isleft=0;
        }
        if(next == NULL)
        {
            if(isleft)
                current->left=ptr;
            else
                current->right=ptr;
            return ptr;
        }
        current=next;
    }
}

```

//Διαγραφή του κόμβου με κλειδί key

```
int rm(int key)
```

```
{
```

```
    struct node *current;
```

```
    struct node *parent;
```

```
    int isLeftChild = 1;
```

```
    current=parent=root;
```

```
    while(current->data != key)
```

```
    {
```

```
        parent = current;
```

```
        if(key < current->data)
```

```
        {
```

```
            isLeftChild = 1;
```

```
            current = current->left;
```

```
        }
```

```
        else
```

```
        {
```

```
            isLeftChild = 0;
```

```
            current = current->right;
```

```
        }
```

```
        if(current == NULL)
```

```
            return 0;
```

```
    }
```

```
    // Ο κόμβος με κλειδί key βρέθηκε. Η διεύθυνσή του είναι στον δείκτη current
```

```
    // και η διεύθυνση του γονικού του στον δείκτη parent
```

```
    //αν ο κόμβος δεν έχει θυγατρικούς
```

```
    if(current->left==NULL && current->right==NULL)
```

```
    {
```

```
        if(current == root)
```

```
        // αν είναι η ρίζα,
```

```
            root = NULL;
```

```
        // το δένδρο είναι κενό
```

```
        else if(isLeftChild)
```

```
            parent->left = NULL;
```

```
        // αποσύνδεση
```

```
        else
```

```
        // από το γονικό
```

```
            parent->right = NULL;
```

```
    }
```

Αναζήτηση του κόμβου με βάση το κλειδί key.

Ο κόμβος δεν εντοπίστηκε. Επιστρέφει τιμή 0 και δεν διαγράφει τίποτα.

```

//αν δεν υπάρχει αριστερός θυγατρικός κόμβος
else if(current->right==NULL)
    if(current == root)
        root = current->left;
    else if(isLeftChild)
        parent->left = current->left;
    else
        parent->right = current->left;

//αν δεν υπάρχει δεξιός θυγατρικός κόμβος
else if(current->left==NULL)
    if(current == root)
        root = current->right;
    else if(isLeftChild)
        parent->left = current->right;
    else
        parent->right = current->right;

//αν υπάρχει και αριστερός και δεξιός θυγατρικός κόμβος
else
{
    struct node *successor, *temp, *old_root;
    if(current == root)
    {
        temp=root->left;
        successor=find_left_most(root->right);
        root=root->right;
        successor->left=temp;
    }

    else if(isLeftChild)
    {
        successor=find_left_most(current->right);
        successor->left=current->left;
        parent->left = current->right;
    }
}

```



```
    }
    else
    {
        successor=find_right_most(current->left);
        successor->right=current->right;
        parent->right = current->left;
    }
}
free(current); //απελευθερώνει τη μνήμη που καταλαμβάνει ο κόμβος
return 1;
}
//εντοπίζει τον τελευταίο αριστερά κόμβο του υποδένδρου rt
struct node *find_left_most(struct node *rt)
{
    if(rt==NULL) return NULL;
    while(rt->left!=NULL)
    {
        rt=rt->left;
    }
    return rt;
}
//εντοπίζει τον τελευταίο δεξιό κόμβο του υποδένδρου rt
struct node *find_right_most(struct node *rt)
{
    if(rt==NULL) return NULL;
    while(rt->right!=NULL)
    {
        rt=rt->right;
    }
    return rt;
}
```

Παραδείγματα

Π.1 Υποθέτουμε ότι έχουμε μια απλά συνδεδεμένη λίστα με την εξής δομή κόμβου:

```
struct node
{
    float temp;
    struct node *next;
}*list_head;
```

Ο χειριστής της λίστας `list_head` δείχνει στην κεφαλή της λίστας. Οι αριθμοί που περιέχει η λίστα είναι οι θερμοκρασίες που μετρήθηκαν κατά τη διάρκεια ενός πειράματος. Η παρακάτω συνάρτηση δέχεται ως παράμετρο το χειριστή της λίστας και επιστρέφει ως τιμή τη μέγιστη θερμοκρασία που υπάρχει στη λίστα.

```
float max_temp(struct node *lh)
{
    float max;
    if (lh==NULL)
    {
        puts("Η λίστα είναι άδεια");
        return;
    }
    else
        max=lh->temp;
    while (lh!=NULL)
    {
        if (lh->temp>max) max=lh->temp;
        lh=lh->next;
    }
    return max;
}
```

Αν η λίστα είναι άδεια, εμφανίζει το κατάλληλο μήνυμα και επιστρέφει.

Καταχωρίζει ως αρχική τιμή της `max` τη θερμοκρασία του πρώτου κόμβου της λίστας.

Αν η `temp` περιέχει τιμή μεγαλύτερη από την `max`, καταχωρίζει στη `max` την τιμή της `temp`.

Στην `lh` καταχωρίζεται η διεύθυνση του επόμενου κόμβου. Αυτό θα επαναλαμβάνεται μέχρι να φτάσουμε στον τελευταίο κόμβο της λίστας. Τέλος επιστρέφει ως τιμή το περιεχόμενο της `max`.

Μια πιο σωστή προσέγγιση θα ήταν αν η συνάρτηση επέστρεφε ένα δείκτη στον κόμβο με τη μεγαλύτερη θερμοκρασία και, στην περίπτωση που η λίστα είναι άδεια, θα επιστρέφει NULL. Με λίγες μετατροπές, η συνάρτηση έχει ως εξής:

```
struct node *max_temp(struct node *lh)
{
    struct node *max_node;
    float max;
    if (lh==NULL)
    {
        return NULL;
    }
    else
    {
        max=lh->temp;
        max_node=lh;
    }
    while (lh!=NULL)
    {
        if (lh->temp>max)
        {
            max=lh->temp;
            max_node=lh;
        }
        lh=lh->next;
    }
    return max_node;
}
```

Αν η λίστα είναι άδεια επιστρέφει NULL.

Καταχωρίζει ως αρχική τιμή της max τη θερμοκρασία του πρώτου κόμβου της λίστας και στη max_node τη διεύθυνση του πρώτου κόμβου.

Αν η temp περιέχει τιμή μεγαλύτερη από την max, καταχωρίζει στη max την τιμή της temp και στη max_node τη διεύθυνση του κόμβου.

Στην lh καταχωρίζεται η διεύθυνση του επόμενου κόμβου. Αυτό θα επαναλαμβάνεται μέχρι να φτάσει στον τελευταίο κόμβο της λίστας. Τέλος επιστρέφει ως τιμή το περιεχόμενο της max_node που δείχνει στον κόμβο που περιέχει τη μέγιστη θερμοκρασία.

Π.2 Το επόμενο πρόγραμμα διαβάζει τους ακέραιους αριθμούς που υπάρχουν σε ένα αρχείο κειμένου με όνομα ARITHMOI και τους καταχωρίζει σε μια απλά συνδεδεμένη λίστα:

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

```
struct node
```

```
{
```

```
    int data;
```

```
    struct node *next;
```

```
}*list_head,*neos;
```

```
void add_node_to_list();
```

```
main()
```

```
{
```

```
    int ar;
```

```
    FILE *fp;
```

```
    fp=fopen("ARITHMOI","r");
```

```
    list_head=NULL;
```

```
    while(!feof(fp))
```

```
    {
```

```
        fscanf(fp,"%d",&ar);
```

```
        add_node_to_list(ar);
```

```
    }
```

```
    fclose(fp);
```

```
}
```

```
//Η ακόλουθη συνάρτηση προσθέτει έναν κόμβο στην κορυφή της λίστας
```

```
void add_node_to_list(int num)
```

```
{
```

```
    neos = (struct node *)malloc(sizeof(struct node));
```

```
    neos->data=num;
```

```
    neos->next = list_head;
```

```
    list_head=neos;
```

```
}
```

Η δομή των κόμβων της συνδεδεμένης λίστας.

Πρόσθαia δήλωση (forward declaration) της συνάρτησης add_node_to_list(), η οποία ορίζεται μετά από τη main().

Ανοίγει το αρχείο ARITHMOI για ανάγνωση δεδομένων.

Δημιουργεί μια κενή (προς το παρόν) λίστα.

Διαβάζει έναν-έναν τους αριθμούς από το αρχείο και τους προσθέτει στη λίστα.

Κλείνει το αρχείο.

Καταχώριση του αριθμού στο πεδίο data του νέου κόμβου.

Στο πεδίο next καταχωρίζεται η διεύθυνση του μέχρι στιγμής κόμβου κεφαλής της λίστας, ενώ στο χειριστή της λίστας list_head καταχωρίζεται η διεύθυνση του νέου κόμβου, ο οποίος είναι τώρα στην κορυφή της λίστας.

Π.3 Υποθέτουμε ότι έχουμε ένα δυαδικό δένδρο με την παρακάτω δομή κόμβου, το οποίο περιέχει τις θερμοκρασίες που παρουσιάστηκαν σε ένα πείραμα: ★★ ★

```
struct node
{
    float temp;
    struct node *left;
    struct node *right;
}*root;
```

Το δυαδικό δένδρο προσδιορίζεται από το δείκτη `root`, ο οποίος δείχνει στον κόμβο ρίζας του δένδρου. Ο επόμενος κώδικας εμφανίζει την μικρότερη θερμοκρασία από τις καταχωρισμένες στο δυαδικό δένδρο.

Η κόμβος με τη μικρότερη θερμοκρασία είναι ο τελευταίος αριστερά κόμβος του δυαδικού δένδρου. Για να εντοπίσουμε αυτόν τον κόμβο, ξεκινάμε από τον κόμβο ρίζας και ακολουθούμε όλους τους δείκτες προς τα αριστερά.

```
struct node *current;
current=root;
if (current==NULL)
{
    puts("Αδειο δένδρο");
    exit();
}
while (current->left!=NULL)
{
    current=current->left;
}
```

Το δένδρο είναι αδειο.

Ο δείκτης `current` παίρνει την τιμή του δείκτη για το αριστερό υποδένδρο. Αυτό επαναλαμβάνεται μέχρι να φτάσουμε σε κόμβο χωρίς αριστερό υποδένδρο.

```
printf("Ελάχιστη θερμοκρασία %f/n", current->temp);
```

Ανασκόπηση Κεφαλαίου 18

- Οι δυναμικές δομές δεδομένων επιτρέπουν τη διαχείριση δεδομένων δυναμικά. Δεσμεύουν μνήμη όταν τη χρειάζονται και την απελευθερώνουν όταν δεν είναι πλέον απαραίτητη.
- Οι κατάλληλες δομές δεδομένων μπορεί να επιταχύνουν εντυπωσιακά την αναζήτηση και την ταξινόμηση.
- Οι δομές δεδομένων αποτελούνται από **κόμβους**. Κάθε κόμβος περιέχει δεδομένα και δείκτες για τη σύνδεσή του με την υπόλοιπη δομή δεδομένων.
- Μια δομή επιτρέπει προσθήκη, διαγραφή, και αναζήτηση των κόμβων της.

Ασκήσεις Κεφαλαίου 18

- 18.1** Υποθέτουμε ότι έχουμε έναν πίνακα **a** με 100 τυχαίους αριθμούς. Να γραφεί κώδικας ο οποίος να δημιουργεί μια απλά συνδεδεμένη λίστα με δεδομένα τους αριθμούς του πίνακα **a**. ★★
- 18.2** Υποθέτουμε ότι έχουμε έναν πίνακα **a** με 100 τυχαίους αριθμούς. Να γραφεί κώδικας ο οποίος να δημιουργεί μια **ταξινομημένη** συνδεδεμένη λίστα με δεδομένα τους αριθμούς του πίνακα **a**. ★★★
- 18.3** Υποθέτουμε ότι έχουμε έναν πίνακα **a** με 100 τυχαίους αριθμούς. Να γραφεί κώδικας ο οποίος να δημιουργεί ένα δυαδικό δένδρο με δεδομένα τους αριθμούς του πίνακα **a**. ★★★
- 18.4** Υποθέτουμε ότι έχουμε μια απλά συνδεδεμένη λίστα με την επόμενη δομή κόμβου: ★★

```
struct node
{
    float varos;
    struct node *next;
} *list_head;
```


Ο χειριστής της λίστας `list_head` δείχνει στην κεφαλή της λίστας. Οι αριθμοί που περιέχει η λίστα είναι τα σωματικά βάρη ενός δείγματος πληθυσμού. Να γραφεί κώδικας ο οποίος να υπολογίζει το μέσο όρο του σωματικού βάρους του δείγματος.

- 18.5** Υποθέτουμε ότι έχουμε ένα δυαδικό δένδρο με την επόμενη δομή κόμβου, το οποίο περιέχει τις ηλικίες ενός δείγματος: ★ ★ ★

```
struct node
{
    float ilikia;
    struct node *left;
    struct node *right;
}*root;
```

Το δυαδικό δένδρο προσδιορίζεται από το δείκτη `root`, ο οποίος δείχνει στον κόμβο ρίζας του δένδρου. Να γραφούν τρεις συναρτήσεις οι οποίες να δέχονται ως παράμετρο το δείκτη `root` και να επιστρέφουν αντίστοιχα:

- Τη μέγιστη ηλικία
- Την ελάχιστη ηλικία
- Το μέσο όρο των ηλικιών

- 18.6** Να γραφεί πρόγραμμα το οποίο να ζητάει ονόματα και να δημιουργεί ένα δυαδικό δένδρο στο οποίο να τα καταχωρίζει. Το πρόγραμμα να σταματάει όταν δοθεί κενό ("") όνομα: ★ ★ ★

- 18.7** Υποθέτουμε ότι έχουμε μια απλά συνδεδεμένη λίστα με την επόμενη δομή κόμβου: ★ ★

```
struct node
{
    char onoma[30];
    struct node *next;
}*list_head;
```


Ο χειριστής της λίστας `list_head` δείχνει στην κεφαλή της λίστας. Η λίστα περιέχει ονόματα. Να γραφεί κώδικας ο οποίος να γράφει τα ονόματα που υπάρχουν στη λίστα σε ένα αρχείο κειμένου που ονομάζεται ΟΝΟΜΑΤΑ.

18.8 Να γραφεί κώδικας ο οποίος να διαβάζει τα ονόματα που υπάρχουν σε ένα αρχείο κειμένου που ονομάζεται ΟΝΟΜΑΤΑ (ένα σε κάθε γραμμή) και να τα καταχωρίζει σε μια **ταξινομημένη** (διατεταγμένη) απλά συνδεδεμένη λίστα: ★ ★ ★

18.9 Ποια από τα επόμενα αληθεύουν: ★

- ☐ Οι λίστες και τα δυαδικά δένδρα δεσμεύουν συγκεκριμένο μέγεθος μνήμης.
- ☐ Σε μια δομή ουράς, το πρώτο στοιχείο που προστίθεται στην ουρά είναι το πρώτο που φεύγει από την ουρά.
- ☐ Σε μια δομή στοίβας, το πρώτο στοιχείο που προστίθεται στη στοίβα είναι το πρώτο που φεύγει από τη στοίβα.
- ☐ Σε μια απλά συνδεδεμένη λίστα δεν μπορούμε να εμφανίσουμε τα στοιχεία της λίστας με τη σειρά από το τελευταίο προς το πρώτο.
- ☐ Ένα δυαδικό δένδρο διατηρεί τα δεδομένα διατεταγμένα ως προς το πεδίο-κλειδί του δυαδικού δένδρου.