Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Κατασκευή Βιβλιοθήκης Συναρτήσεων σε Γλώσσα C για Συνδεδεμένες Λίστες** **A Library of Functions in C for Linked Lists**
Ονοματεπώνυμο Φοιτητή	**Αλέξανδρος Γεωργίου**
Πατρώνυμο	**Σεραφείμ**
Αριθμός Μητρώου	**ΜΠΠΛ/ ...13008.............................**
Επιβλέπων	**Θέμης Παναγιωτόπουλος, Καθηγητής**

Σεπτέμβρης 2017

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Όνομα Επώνυμο Βαθμίδα	Όνομα Επώνυμο Βαθμίδα	Όνομα Επώνυμο Βαθμίδα

Περίληψη . . . .

Abstract . . . .

Εισαγωγή 0

Εισαγωγή στις Συνδεδεμένες Λίστες 1

Αποθήκευση της λίστας στη μνήμη 2

2. Ενέργειες στις Συνδεδεμένες Λίστες 5

3. Εισαγωγή ενός νέου κόμβου 16

4. Κυκλικές συνδεδεμένες λίστες 20

5. Κυκλικές Διπλά συνδεδεμένες λίστες 22

6. Επίδειξη του κώδικα 22

7. Συμπεράσματα 48

8. Βιβλιογραφία 50

Περίληψη

Οι συνδεδεμένες λίστες είναι βασικές δομές δεδομένων στη C. Η γνώση τους είναι απαραίτητη στους προγραμματιστές της γλώσσας αυτής. Η εργασία αυτή εξηγεί τις βασικές αρχές των συνδεδεμένων λιστών, παραθέτοντας μια βιβλιοθήκη συναρτήσεων σε γλώσσα C. Οι συνδεδεμένες λίστες είναι δυναμικές δομές δεδομένων, με μέγεθος που μπορεί να αυξομειωθεί κατά τον χρόνο εκτέλεσης. Σε μια συνδεδεμένη λίστα τα στοιχεία (οι κόμβοι) μπορούν να βρίσκονται σε απομακρυσμένες θέσεις. Οι συνδεδεμένες λίστες προτιμώνται όταν δε γνωρίζουμε το μέγεθος των δεδομένων που θα αποθηκευτούν. Για παράδειγμα, σε ένα σύστημα διαχείρισης υπαλλήλων, δε μπορούν να χρησιμοποιηθούν πίνακες, καθώς αυτοί έχουν σταθερό μέγεθος, ενώ μπορεί να προστεθεί οποιοσδήποτε αριθμός υπαλλήλων. Σε περιπτώσεις σαν αυτή, οι συνδεδεμένες λίστες μπορούν να χρησιμοποιηθούν καθώς η χωρητικότητά τους μπορεί να αυξηθεί (ή να μειωθεί) κατά τον χρόνο εκτέλεσης. Στην αρχιτεκτονική των συνδεδεμένων λιστών βασίστηκαν αλγόριθμοι όπως η σωρός του Fibonacci. Ο αλγόριθμος περιείχε όλες τις βασικές λειτουργίες των συνδεδεμένων λιστών όπως την αναζήτηση, εισαγωγή , αναζήτηση ελάχιστου κ.α. Οι αλγόριθμοι που αναπτύχθηκαν με βάση τη σειρά αριθμών και τις λειτουργίες των συνδεδεμένων λιστών βρήκαν χρησιμότητα σε διάφορους τομείς της πληροφορικής και του προγραμματισμού.

Abstract

Linked list is one of the fundamental data structures in C. Knowledge of linked lists is must for C programmers. This work explains the fundamentals of linked lists with a library of C functions. Linked lists are a data structure which you may want to use in real programs. The strengths and weaknesses of linked lists give an appreciation of the some of the time, space, and code issues which are useful to thinking about any data structures in general. Linked list is a dynamic data structure whose length can be increased or decreased at run time. In a linked list the elements (or nodes) may be kept at any location. Linked lists are preferred mostly when you don't know the volume of data to be stored. Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including lists (the abstract data type), stacks, queues, associative arrays, and S-expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation .For example, in an employee management system, one cannot use arrays as they are of fixed length while any number of new employees can join. In scenarios like these, linked lists are used as their capacity can be increased (or decreased) at run time.

Εισαγωγή

Οι συνδεδεμένες λίστες αναπτύχθηκαν το 1955 από τους Allen Newell, Cliff Shaw και Herbert A. Simon στο RAND Corporation ως κύρια δομή δεδομένων για τη γλώσσα

επεξεργασίας των πληροφοριών τους. Το IPL χρησιμοποιήθηκε για την ανάπτυξη αρκετών προγραμμάτων πρόωρης τεχνητής νοημοσύνης, συμπεριλαμβανομένης της Λογικής Μηχανής Θεωρίας, του Γενικού Επίλυσης Προβλημάτων Οι Newell και Simon αναγνωρίστηκαν με το βραβείο ACM Turing το 1975 επειδή "έχουν κάνει βασικές συνεισφορές στη τεχνητή νοημοσύνη, την ψυχολογία της ανθρώπινης γνώσης και την επεξεργασία των καταλόγων". Το πρόβλημα της μηχανικής μετάφρασης για την επεξεργασία φυσικής γλώσσας δημιούργησε ανάγκες με αποτέλεσμα στο Ινστιτούτο Τεχνολογίας της Μασαχουσέτης (MIT) να χρησιμοποιηθεί συνδεδεμένες λίστες ως δομές δεδομένων στη γλώσσα προγραμματισμού του COMIT για την έρευνα στον τομέα της γλωσσολογίας.

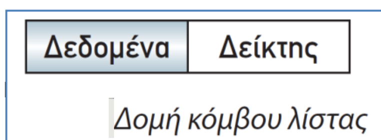
Η LISP ήταν υπεύθυνη για τον επεξεργαστή λιστών. Μία από τις σημαντικότερες δομές δεδομένων της LISP είναι ο συνδεδεμένη λίστα.

Η χρησιμότητα και των δύο συνδεδεμένων καταλόγων και των γλωσσών που χρησιμοποιούν αυτές τις δομές ως πρωταρχική εκπροσώπηση των δεδομένων τους φάνηκε καθώς αρκετά λειτουργικά συστήματα που αναπτύχθηκαν από τους Technical Systems Consultants (αρχικά της West Lafayette Indiana, και αργότερα Chapel Hill, Βόρεια Καρολίνα) χρησιμοποίησαν απλά συνδεδεμένους καταλόγους ως δομές αρχείων. Μια καταχώρηση καταλόγου επεσήμανε τον πρώτο τομέα ενός αρχείου και εντοπίστηκαν διαδοχικά τμήματα του αρχείου, μεταφέροντας τους δείκτες. Συστήματα που χρησιμοποιούν αυτή την τεχνική συμπεριλάμβαναν Flex (για τον επεξεργαστή Motorola 6800 CPU), mini-Flex (ίδια CPU) και Flex9 (για την CPU Motorola 6809). Μια παραλλαγή που αναπτύχθηκε από την TSC και κυκλοφόρησε στην αγορά από την Smoke Signal Broadcasting στην Καλιφόρνια, χρησιμοποίησε διπλά συνδεδεμένες λίστες με τον ίδιο τρόπο.

Το λειτουργικό σύστημα TSS / 360, που αναπτύχθηκε από την IBM για τις μηχανές System 360/370, χρησιμοποίησε μια λίστα διπλών συνδέσμων για τον κατάλογο συστημάτων αρχείων τους. Η δομή καταλόγου ήταν παρόμοια με το Unix, όπου ένας κατάλογος θα μπορούσε να περιέχει αρχεία και άλλους καταλόγους και να επεκταθεί σε οποιοδήποτε βάθος.

Εισαγωγή στις Συνδεδεμένες Λίστες

Στις λίστες το κύριο χαρακτηριστικό είναι ότι οι κόμβοι τους συνήθως βρίσκονται σε απομακρυσμένες θέσεις μνήμης και η σύνδεσή τους γίνεται με δείκτες. Ο δείκτης (pointer) είναι ένας ιδιαίτερος τύπος που προσφέρεται από τις περισσότερες σύγχρονες γλώσσες προγραμματισμού. Ο δείκτης δεν λαμβάνει αριθμητικές τιμές όπως ακέραιες, πραγματικές κ.ά., αλλά οι τιμές του είναι διευθύνσεις στην κύρια μνήμη και



Δομή κόμβου λίστας

χρησιμοποιείται ακριβώς για τη σύνδεση των διαφόρων στοιχείων μιας δομής, που είναι αποθηκευμένα σε μη συνεχόμενες θέσεις μνήμης. Συνήθως ο δείκτης είναι ένα πεδίο κάθε κόμβου της δομής, όπως φαίνεται στο σχήμα. Το πεδίο Δεδομένα μπορεί να περιέχει μία ή περισσότερες αλφαριθμητικές ή αριθμητικές πληροφορίες. Στο σχήμα παρουσιάζεται μια λίστα με τέσσερις κόμβους, όπου οι δείκτες έχουν τη μορφή βέλους, προκειμένου να φαίνεται ο κόμβος που παραπέμπουν.

Μία λίστα με τέσσερις κόμβους

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

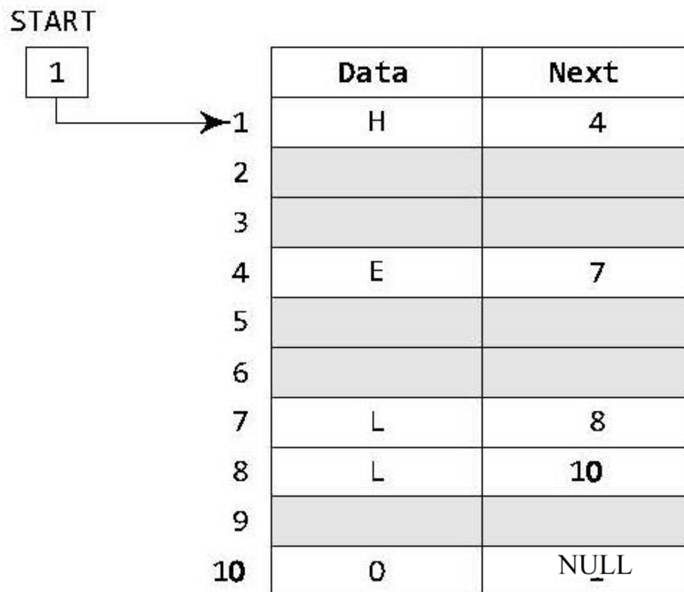
Με την συνδεδεμένη αναπαράσταση (linked representation), τα στοιχεία της λίστας αποθηκεύονται σε μη διαδοχικές θέσεις στη μνήμη του Η/Υ. Τώρα, η μετάβαση από το ένα στοιχείο στο επόμενο, πραγματοποιείται με τη χρήση ενός συνδέσμου (link) μεταξύ των στοιχείων. Τα «στοιχεία» μιας συνδεδεμένης λίστας αποκαλούνται κόμβοι (nodes). Ένας κόμβος αποτελείται από δύο τμήματα. Το αριστερό τμήμα περιέχει την πληροφορία (τιμή) του κόμβου και συνιστά το στοιχείο του κόμβου. Το δεξιό τμήμα περιέχει τη διεύθυνση της (πρώτης) θέσης μνήμης του επόμενου κόμβου, είναι δηλαδή ένας δείκτης στον επόμενο κόμβο της λίστας. Μια λίστα με κόμβους τέτοιας δομής ονομάζεται απλά συνδεμένη λίστα (one-way or singly linked list). Υπάρχουν και άλλοι τύποι συνδεδεμένης λίστας, που παρουσιάζονται στη συνέχεια.

Αποθήκευση της λίστας στη μνήμη

Η υλοποίηση μιας απλά συνδεδεμένης λίστας μπορεί να γίνει κατά δύο τρόπους. Ο ένας τρόπος είναι με τη χρήση πινάκων. Ο τρόπος αυτός χρησιμοποιείται για την υλοποίηση συνδεδεμένων λιστών στις παλαιότερες γλώσσες υψηλού επιπέδου, όπως η FORTRAN, η BASIC και η COBOL, που δε διαθέτουν μηχανισμό δημιουργίας δεικτών (συνδέσμων).

Ο δεύτερος τρόπος χρησιμοποιεί μεταβλητές δείκτη (pointer variables). Ο δείκτης (pointer) είναι ένας τύπος δεδομένων που έχει πεδίο τιμών το σύνολο των διευθύνσεων της μνήμης του Η/Υ. Μια μεταβλητή δείκτη παίρνει σαν τιμές διευθύνσεις θέσεων μνήμης που αντιστοιχούν σε κόμβους, δηλαδή σε σύνθετες τιμές. Έτσι, για κάθε κόμβο χρησιμοποιείται και μια μεταβλητή δείκτη (ή ένας

δείκτης). Ο τρόπος αυτός χρησιμοποιείται για υλοποίηση με γλώσσες υψηλού επιπέδου που παρέχουν κάποιο μηχανισμό δημιουργίας δεικτών, όπως η PASCAL και η C. Στο κεφάλαιο αυτό, ασχολούμαστε με αυτό το δεύτερο τρόπο υλοποίησης συνδεδεμένων λιστών. Συνήθως, σε μια τέτοια γλώσσα, ο κάθε κόμβος ορίζεται σαν μια εγγραφή με δύο πεδία, το ένα περιέχει το στοιχείο και το άλλο τον δείκτη..



1.1. Δημιουργία άδειας λίστας

Κώδικας δημιουργίας άδειας λίστας

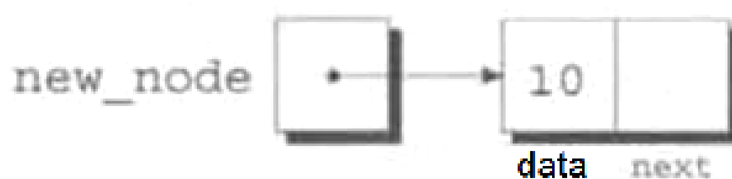
```
struct node
{
      int data;
      struct node* next;
};
```

```
struct node * head = NULL ;
```

1.2. Δημιουργία ενός νέου κόμβου

α) struct node* new_node;
β) new_node = (struct node*)malloc(sizeof(struct node));
γ) new_node->data = 10;

α) Δημιουργούμε έναν δείκτη προς το νέο στοιχείο
β) Δεσμεύουμε μνήμη για το νέο στοιχείο και τοποθετούμε τον δείκτη να δείχνει αυτόν τον χώρο
γ) Δίνουμε τιμή στο πεδίο data του κόμβου.



2. Ενέργειες στις Συνδεδεμένες Λίστες

2.1. Δημιουργία λίστας με Ν κόμβους

```
struct node * create_list(struct node *start, int num)
{
  struct node *new_node, *ptr;
  int i ;
  for (i = 0 ; i < num ; i++)   // Ο βρόγχος τρέχει n φορές, κάθε φορά που ένας
κόμβος τοποθετείται στο τέλος της λίστας
  {
      if (start == NULL)   // new node is the first node
      {
        new_node = (struct node*)malloc(sizeof(struct node));  // Δέσμευσε μνήμη
για το νέο κόμβο
        start = new_node;
        new_node->data = i;
        new_node->next = NULL ;   // Στη κυκλική λίστα γίνεται :  new_node->next
= start ;
      }
      else    // ο νέος κόμβος δεν είναι ο πρώτος κόμβος,     nodes no 1 to num-1,
να εισαχθεί ο νέος κόμβος στο τέλος της λίστας
      {
        new_node = (struct node*)malloc(sizeof(struct node)); // Δέσμευσε μνήμη
για το νέο κόμβο
        new_node->data = i;
        ptr = start;
        while(ptr->next != NULL)    // Ο δείκτης μετατίθεται στο τελευταίο κόμβο

             { ptr = ptr->next; }
        ptr->next = new_node;     //  Ο προηγούμενος τελευταίος κόμβος ισούται με
το καινούργιο κόμβο
        new_node->next = NULL;    // και ο νέος κόμβος γίνεται τελευταίος
                    }
  }
  printf("\n Linked List Created \n\n");
```

```
  return start;
}
```

Καλούμε αυτή τη συνάρτηση για να φτιάξουμε λίστα με 100 κόμβους ως εξής:

```
struct node * head ;
head  =  create_list(NULL, 100) ;
```

Η δημιουργία των κόμβων της λίστας γίνεται με την κλήση της malloc( ). Ο δείκτης head τοποθετείται στην κεφαλή της λίστας. Αυτός αποτελεί τον δείκτη ο οποίος δεν θα μετακινείται ποτέ.

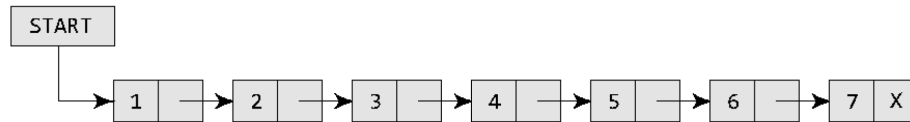
Η γλώσσα προγραμματισμού C διαχειρίζεται τη μνήμη στατικά, αυτόματα ή δυναμικά. Οι μεταβλητές στατικής διάρκειας κατανέμονται στην κύρια μνήμη, συνήθως μαζί με τον εκτελέσιμο κώδικα του προγράμματος και παραμένουν για όλη τη διάρκεια του προγράμματος. Οι μεταβλητές αυτόματης διάρκειας κατανέμονται στη στοίβα και έρχονται και μεταβαίνουν καθώς καλούνται οι λειτουργίες και επιστρέφουν. Για τις μεταβλητές στατικής διάρκειας και αυτόματης διάρκειας, το μέγεθος της κατανομής πρέπει να είναι σε σταθερό χρόνο . Εάν το απαιτούμενο μέγεθος δεν είναι γνωστό μέχρι την εκτέλεση (για παράδειγμα, αν τα δεδομένα αυθαίρετου μεγέθους διαβάζονται από το χρήστη ή από ένα αρχείο δίσκου), τότε η χρήση αντικειμένων δεδομένων σταθερού μεγέθους είναι ανεπαρκής.

Ο χρόνος ζωής της μνήμης που διατίθεται μπορεί επίσης να προκαλέσει ανησυχία. Η μνήμη στατικής και αυτόματης διάρκειας δεν είναι επαρκής για όλες τις καταστάσεις. Τα αυτόματα κατανεμημένα δεδομένα δεν μπορούν να παραμείνουν σε πολλαπλές κλήσεις λειτουργιών, ενώ τα στατικά δεδομένα παραμένουν για τη διάρκεια του προγράμματος, ανεξάρτητα από το αν είναι απαραίτητα ή όχι. Σε πολλές περιπτώσεις ο προγραμματιστής απαιτεί μεγαλύτερη ευελιξία στη διαχείριση της διάρκειας ζωής της μνήμης.

Οι περιορισμοί αυτοί αποφεύγονται με τη χρήση της δυναμικής κατανομής μνήμης, στην οποία η μνήμη διαχειρίζεται πιο ρητά (αλλά πιο ευέλικτα), τυπικά με την κατανομή της από το ελεύθερο κατάστημα (ανεπίσημα αποκαλούμενο "σωρός"), μια περιοχή μνήμης δομημένη για το σκοπό αυτό. Στο C, η λειτουργία malloc της βιβλιοθήκης χρησιμοποιείται για την κατανομή ενός μπλοκ μνήμης στον σωρό. Το πρόγραμμα αποκτά πρόσβαση σε αυτό το μπλοκ μνήμης μέσω ενός δείκτη που επιστρέφει το malloc. Όταν η μνήμη δεν είναι πλέον απαραίτητη, ο δείκτης μεταβιβάζεται σε ελεύθερο, ο οποίος μεταθέτει τη μνήμη έτσι ώστε να μπορεί να χρησιμοποιηθεί για άλλους σκοπούς.

Ορισμένες πλατφόρμες παρέχουν κλήσεις βιβλιοθήκης οι οποίες επιτρέπουν τη δυναμική κατανομή χρόνου εκτέλεσης από τη στοίβα C αντί του σωρού (π.χ. alloca ()). Αυτή η μνήμη απελευθερώνεται αυτόματα όταν λήξει η λειτουργία κλήσης.

2.1.1. Διάσχιση συνδεδεμένης λίστας



Η διάσχιση (traversal) είναι μια από τις συνήθεις πράξεις στις δομές δεδομένων. Διάσχιση σε μία λίστα (ή έναν πίνακα) σημαίνει να «περάσουμε» από όλα τα στοιχεία της, το ένα μετά το άλλο, και να εφαρμόσουμε κάποιο είδος επεξεργασίας σε αυτά. Αυτό, συχνά ονομάζεται και επίσκεψη (visiting). Το είδος της επεξεργασίας εξαρτάται από το συγκεκριμένο πρόβλημα. Π.χ., αύξηση της τιμής των στοιχείων κατά ένα.

```
void  traverse_list(struct node * start)
{
   struct node *ptr;

   if ( start == NULL )          // Η λίστα δεν υπάρχει
      { printf("\n There is no list to work with \n\n"); }

   ptr = start ;
   while (ptr != NULL)      // Στη κυκλικά συνδεδεμένη λίστα γίνεται:  while (ptr->next != start)
   {
      printf("\n value = %d", ptr->data);        // Γενικά  use  ptr->data ;
      ptr = ptr->next ;
   }
}
```

Καλούμε αυτή τη συνάρτηση ως εξής:

```
traverse_list(head) ;
```

2.1.2. Βρόχοι Διάσχισης

Αν έχουμε τον ορισμό:

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

και δύο δείκτες

```
struct node *start, *tmp ;
```

Ο πρώτος δείκτης δείχνει τώρα στην αρχή της λίστας:

```
start =  create_list(start, 100) ;
```

Μπορούμε να διασχίσουμε τη λίστα με ένα **while loop**:

```
ptr = start ;
while (tmp != NULL)
    {
        use tmp->data ;
        tmp = tmp->next;
    }
```

Με τον παραπάνω βρόχο, ο δείκτης tmp δείχνει τώρα σε NULL.

Σε **κυκλική λίστα** ο κώδικας γίνεται:

```
ptr = start ;
while (tmp->next != start)

{
```

```
                use tmp->data ;
                tmp = tmp->next;}
```

Με τον παραπάνω βρόχο, ο δείκτης tmp δείχνει τώρα στην ουρά της λίστας.

Μπορούμε να διασχίσουμε τη λίστα με ένα **for loop**:

```
        for(tmp = start;   tmp->next != NULL ;   tmp = tmp->next);
                { use tmp->data ; }
```

Με τον παραπάνω βρόχο, ο δείκτης tmp δείχνει τώρα στην ουρά της λίστας.

Σε **κυκλική λίστα** ο κώδικας γίνεται:

```
        for(tmp = start;   tmp->next != start;   tmp = tmp->next);
                { use tmp->data; }
```

Με τον παραπάνω βρόχο, ο δείκτης tmp δείχνει τώρα στην ουρά της λίστας.

2.1.3. Υπολογισμός κόμβων

```
int  count_nodes(struct node * start)
{
    struct node *ptr;
    int count = 0 ;
    ptr = start ;
    while (ptr != NULL)      // in circular list, it becomes: while(ptr->next != start)
    {
        count++ ;
        ptr = ptr->next ;
    }
    return count;
}
```

Καλούμε αυτή τη συνάρτηση για να μετρήσουμε τους κόμβους ως εξής:

```
int howmany ;
howmany  =  count_nodes(head) ;              // head πηγαίνει στη αρχή της λίστας.
printf("The list has %d nodes", howmany);
```

Με ένα βήμα:

```
printf("The list has %d nodes", count_nodes(head) );
```

2.2. Αναζήτηση στοιχείου με δεδομένη τιμή

Για την αναζήτηση ενός στοιχείου σε μια συνδεδεμένη λίστα μπορεί να εφαρμοστεί μόνο η μέθοδος της γραμμικής αναζήτησης. Η δυαδική αναζήτηση δεν μπορεί να εφαρμοστεί, διότι δεν υπάρχει απ' ευθείας τρόπος να υπολογιστεί ο δείκτης του μεσαίου' στοιχείου. Επομένως, είτε η λίστα είναι διατεταγμένη είτε όχι, χρησιμοποιούμε τη γραμμική αναζήτηση, ψάχνουμε δηλαδή τα στοιχεία της λίστας ένα-ένα. Βέβαια, όταν η λίστα είναι διατεταγμένη, έχουμε διαφορετική συνθήκη τερματισμού του αλγορίθμου.

```
struct node *find_element(struct node * start, int val)
{
    struct node *ptr, *pos;
    ptr = start ;
    while (ptr != NULL)    // in circular list, it becomes:
while (ptr->next != start)
    {
        if (val == ptr->data)       // Βρέθηκε το στοιχείο, επιστρέφουμε  τον  δείκτη
δείχνοντας το κόμβο που περιέχεται
        {
            pos = ptr ;
            return pos ;
        }
        else   { ptr = ptr->next ; }
    }
    return NULL;   // element is not found, we return NULL
}
```

Καλούμε αυτή τη συνάρτηση για να βρούμε το στοιχείο με τιμή 4 ως εξής:

```
struct node *elist;
elist  =  find_element(head, 4) ;
if (elist != NULL)  { printf("The element %d found in our list",
elist->data); }
else                  { printf("The element has not been found in
our list"); }
```

2.3. Ψάχνοντας τη Μέγιστη ή Ελάχιστη τιμή

```
int  find_max(struct node * start)     // Για την ελάχιστη τιμή:    int  find_min(struct
node * start)
{
   int max ;          // in case of min, it is:    int min ;
   struct node *ptr ;
   ptr = start ;
   max = start->data ;      //  Για την ελάχιστη τιμή:    min = start->data ;
   while (ptr != NULL)
   {
      if (ptr->data > max)  { max = ptr->data ; }
      ptr = ptr->next ;
   }
   // Στη κυκλικά συνδεδεμένη λίστα, προσθέτουμε τη γραμμή, για να ελέγξουμε το
τελευταίο κόμβο      if (ptr->data  >  max)   { max = ptr->data ; }
   return max;      /7
}
```

Καλούμε αυτή τη συνάρτηση για να βρούμε το μέγιστο στοιχείο ως εξής:

```
int meg ;
meg  =  find_max(head) ;
printf("The maximum element is %d ", meg);
```

Με ένα βήμα:

```
printf("The maximum element is %d ", find_max(head));
```

Στους πίνακες , χρησιμοποιούμαι την ίδια τεχνική:

```
int size = 8 ;       int arr[size] = {2, 5, 9, -5, 6, 8, 3, 6} ;
int meg ;            meg  =  array_max(arr, size) ;

int array_max(int * pin, int n)
{
    int max ;
    for (i = 0; i < n ; i++)   { if (pin[i] > max)  {max = pin[i] ;}
}
                                        // in case of min, it is: {   if (pin[i]  <  min)  { min = pin[i] ; }   }
    return max;     }
```

2.4. Άθροισμα και Μέση Τιμή

```
float  list_average(struct node * start)   // Σε συνάρτηση  list_sum,   it is:  int
list_sum(struct node * start)
{
   int athr, counter = 0 ;
   float ave ;
   struct node *ptr ;
   ptr = start ;
   while (ptr != NULL)
   {
      athr = athr + ptr->data ;
      counter++ ;
      ptr = ptr->next ;
   }
   ave = athr / counter ;
   return ave;
}
```

Καλουμε αυτή τη συνάρτηση για να βρούμε τη μέση τιμή ως εξής:

```
int sum ;
float average ;
sum      =  list_sum(head) ;          // head points at the beginning of the list
average  =  list_average(head) ;         // head points at the beginning of the list
```

Σε πίνακες , χρησιμοποιούμε την ίδια τεχνική , αλλά έχουμε τον αριθμό κόμβων:

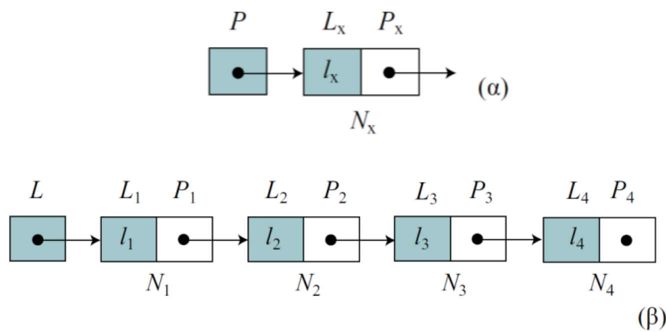
```
int size = 8 ;      int arr[size] = {2, 5, 9, -5, 6, 8, 3, 6} ;
float average ;      average  =  array_average(arr, size) ;

float  array_average(int * pin, int n)
{
   int i, athr ;
   float ave ;
   for (i = 0; i < n ; i++)    { athr = athr + pin[i] ; }
```

```
   ave = athr / counter ;
   return ave;
}
```

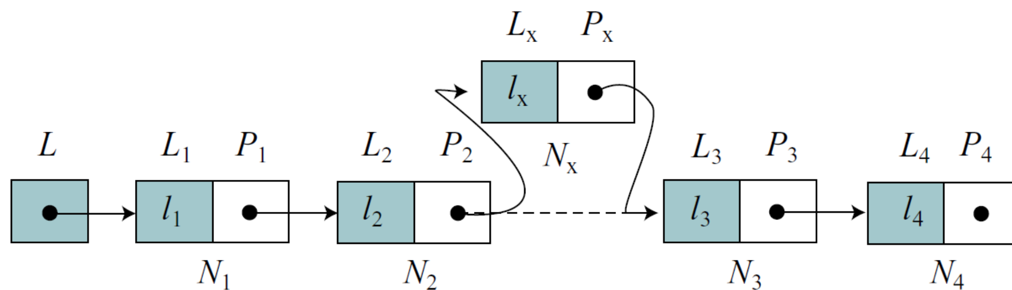
3. Εισαγωγή ενός νέου κόμβου

Με τη χρήση δεικτών διευκολύνονται οι λειτουργίες της εισαγωγής και της διαγραφής δεδομένων στις λίστες. Στο σχήμα φαίνεται η εισαγωγή ενός νέου κόμβου μεταξύ του δεύτερου και τρίτου κόμβου της προηγούμενης λίστας. Όπως φαίνεται και στο σχήμα, οι απαιτούμενες ενέργειες για την εισαγωγή (παρεμβολή) του νέου κόμβου είναι ο δείκτης του δεύτερου κόμβου να δείχνει το νέο κόμβο και ο δείκτης του νέου κόμβου να δείχνει τον τρίτο κόμβο (δηλαδή να πάρει την τιμή που είχε πριν την εισαγωγή ο δείκτης του δεύτερου κόμβου). Έτσι οι κόμβοι της λίστας διατηρούν τη λογική τους σειρά, αλλά οι φυσικές θέσεις στη μνήμη μπορεί να είναι τελείως διαφορετικές.



Κατάσταση πριν την εισαγωγή κόμβου σε συνδεδεμένη λίστα:

(α) προς εισαγωγή κόμβος, **(β) αρχική συνδεδεμένη λίστα**



Κατάσταση μετά την εισαγωγή κόμβου σε συνδεδεμένη λίστα

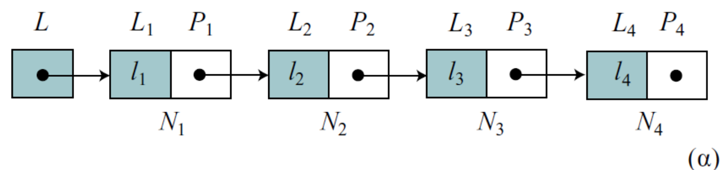
Θα εξετάσουμε τις ακόλουθες περιπτώσεις:

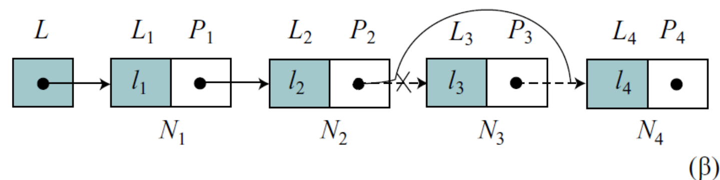
1. Εισαγωγή στην αρχή
2. Εισαγωγή στο τέλος
3. Εισαγωγή μετά από κάποιον συγκεκριμένο κόμβο
4. Εισαγωγή πριν από κάποιον συγκεκριμένο κόμβο

Ο κώδικας παρατίθεται στο επόμενο κεφάλαιο.

3.1. Διαγραφή ενός κόμβου

Αντίστοιχα για τη διαγραφή ενός κόμβου αρκεί ν' αλλάξει τιμή ο δείκτης του προηγούμενου κόμβου και να δείχνει πλέον τον επόμενο αυτού που διαγράφεται, όπως φαίνεται στο σχήμα. Ο κόμβος που διαγράφηκε (ο τρίτος) αποτελεί "άχρηστο δεδομένο" και ο χώρος μνήμης που καταλάμβανε, παραχωρείται για άλλη χρήση.





(α) Κατάσταση πριν, (β) κατάσταση μετά τη διαγραφή του 3ου κόμβου

Θα εξετάσουμε τις ακόλουθες περιπτώσεις:

1. Διαγραφή του πρώτου κόμβου
2. Διαγραφή του τελευταίου κόμβου
3. Διαγραφή κάποιου συγκεκριμένου κόμβου
4. Διαγραφή του κόμβου που βρίσκεται μετά από κάποιον δεδομένο κόμβο

Ο κώδικας παρατίθεται στο επόμενο κεφάλαιο.

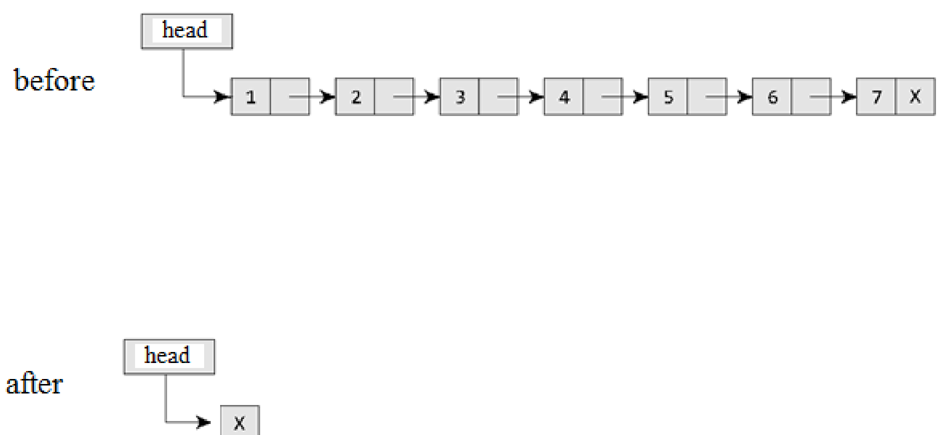
3.2. Διαγραφή μιας συνδεδεμένης λίστας

```
struct node *delete_list(struct node *start)       // Διαγραφή ολόκληρης της λίστας
{
    struct node *ptr;                 // Νέος δείκτης
    if ( start == NULL )
        { printf("\n There is no list to delete \n\n"); return start ; }
    ptr = start;                      // Ο δείκτης μετατίθεται στην αρχή της λίστας
    while(ptr->next != NULL)
      {start = delete_node_at_the_end(start);}  // διαγραφή όλων των κόμβων από το τέλος έως την αρχή αντίστροφα
    free(start);                      // Αφαίρεση πρώτου κόμβου
    start = NULL ;                    // αρχικοποίηση πρώτου κόμβου

    printf("\n Linked List Deleted \n\n");
    return start;
}
```

Καλούμε αυτή τη συνάρτηση για να διαγράψουμε τη λίστα ως εξής:

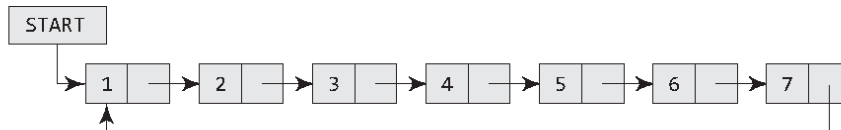
head = delete_list(head) ;
// δείκτης head στην αρχή της λίστας, όταν τελειώσει η λίστα ο δείκτης γίνεται NULL



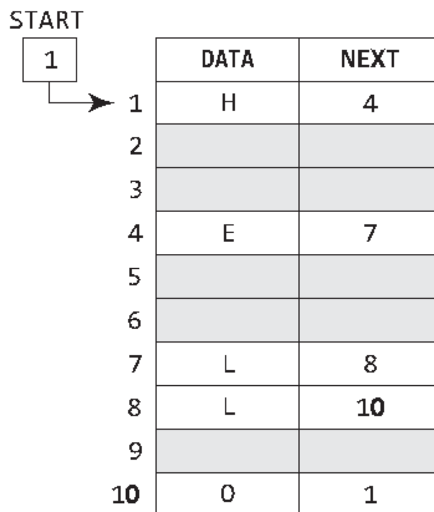
4. Κυκλικές συνδεδεμένες λίστες

Σε μια κυκλική συνδεδεμένη λίστα (circular linked list) ο δείκτης του τελευταίου κόμβου «δείχνει» πίσω στον πρώτο κόμβο της λίστας, όπως απεικονίζεται στο Σχήμα. Η κυκλική λίστα έχει το πλεονέκτημα, όπως και η διπλά συνδεδεμένη, ότι μπορούμε να προσπελάσουμε από έναν κόμβο οποιοδήποτε άλλο κόμβο.

Σχηματικά η λίστα μοιάζει ως εξής:



Η εικόνα της μνήμης είναι τώρα:



	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

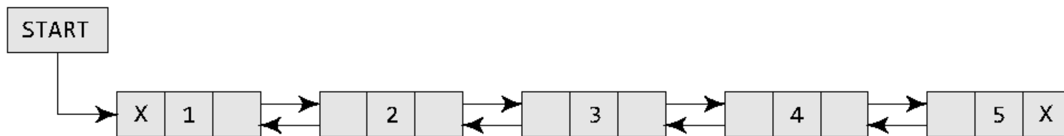
Ο κώδικας για τις κυκλικές λίστες συμπεριλαμβάνεται στα σχόλια του κώδικα για τις απλές λίστες.

4.1. Διπλά συνδεδεμένες λίστες

Στη διπλά συνδεδεμένη λίστα (two-way or doubly linked list), οι κόμβοι έχουν δύο δείκτες, έναν που «δείχνει» στον επόμενο κόμβο και ονομάζεται δεξιός δείκτης (right pointer), και έναν που «δείχνει» στον προηγούμενο κόμβο και ονομάζεται αριστερός δείκτης (left pointer). Στο Σχήμα, απεικονίζεται η σχηματική παράσταση μιας διπλά συνδεδεμένης λίστας. Όπως παρατηρείτε, ο κάθε κόμβος αποτελείται από τρία τμήματα, δύο ακραία για την αποθήκευση των δύο δεικτών και το μεσαίο για την αποθήκευση του στοιχείου (πληροφορίας) του κόβου.

Δοθέντος ενός δείκτη P μπορούμε να προσπελάσουμε όλους τους υπόλοιπους κόμβους της λίστας, κινούμενοι είτε προς τη μια είτε προς την άλλη κατεύθυνση, πράγμα που δεν μπορεί να γίνει σε μια απλά συνδεδεμένη λίστα. Γι᾽ αυτό, μπορούμε να εισάγουμε και να διαγράψουμε έναν κόμβο είτε πριν είτε μετά από τον δοθέντα.

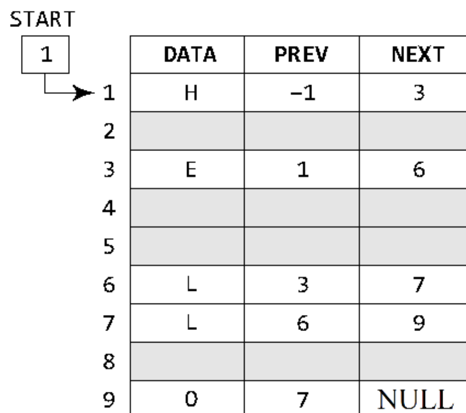
Σχηματικά η λίστα μοιάζει ως εξής:



Ο ορισμός του κόμβου της λίστας γίνεται:

```
struct dblnode
{
	int data;
	struct dblnode* next;
	struct dblnode* prev;
};
```

Η εικόνα της μνήμης είναι **τώρα:**



	DATA	PREV	NEXT
1	H	-1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	NULL

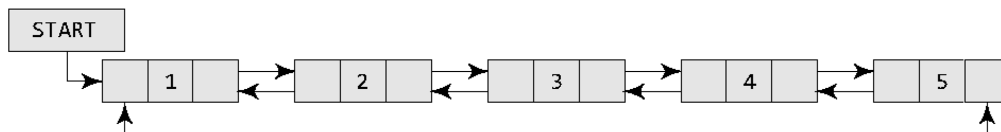
5. Κυκλικές Διπλά συνδεδεμένες λίστες

α) Σε μια κυκλική συνδεδεμένη λίστα ο δείκτης του τελευταίου κόμβου «δείχνει» πίσω στον πρώτο κόμβο της λίστας, όπως απεικονίζεται στο Σχήμα.

β) Στη διπλά συνδεδεμένη λίστα, οι κόμβοι έχουν δύο δείκτες, έναν που «δείχνει» στον επόμενο κόμβο και ονομάζεται δεξιός δείκτης (right pointer), και έναν που «δείχνει» στον προηγούμενο κόμβο και ονομάζεται αριστερός δείκτης (left pointer).

Σε μια κυκλική διπλά συνδεδεμένη λίστα συνδυάζουμε τις ιδιότητες α και β.

Σχηματικά η λίστα μοιάζει ως εξής:



Η εικόνα της μνήμης είναι τώρα:

START: 1

	DATA	PREV	Next
1	H	9	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	1

Ο κώδικας για τις κυκλικές λίστες συμπεριλαμβάνεται στα σχόλια του κώδικα για τις διπλές λίστες.

6. Επίδειξη του κώδικα

6.1. Ο κώδικας των συναρτήσεων

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

struct node
{
	int data;
	struct node* next;
};

struct dblnode
{
	int data;
	struct dblnode* next;
	struct dblnode* prev;
};


void print_list_iterative(struct node* head)
{
	struct node* ptr = head;
	if(!head)
	{
		printf("empty list!");
		return;
	}
	while(ptr)
	{
		printf("%d -> ", ptr->data);
		ptr = ptr->next;
	}
	printf("null");
}void print_list_iterative_double(struct dblnode* head)
{
	struct dblnode* ptr = head;
```

```
        if(!head)
        {
                printf("empty list!");
                return;
        }
        while(ptr)
        {
                printf("%d <-> ", ptr->data);
                ptr = ptr->next;
        }
        printf("null");
}
struct node * create_list(struct node *start, int num)
{
  struct node *new_node, *ptr;
  int i ;
  for (i = 0 ; i < num ; i++)   // loop runs num times, every time one node is put at
the end of the list
  {
     if (start == NULL)   // new node is the first node
     {
       new_node = (struct node*)malloc(sizeof(struct node));  // get memory for
the new node
       start = new_node;
       new_node->data = i;
       new_node->next = NULL ;   // in circular list, it becomes:  new_node->next
= start ;
     }
     else    // new node is not the first node,     nodes no 1 to num-1,    insert the
new node at the end of the list
     {
       new_node = (struct node*)malloc(sizeof(struct node)); // get memory for the
new node
       new_node->data = i;              // put the node's id as date
       ptr = start;
       while(ptr->next != NULL)    // ptr goes to the last node
```

```
                                // in circular: while(ptr->next != start)
          { ptr = ptr->next; }

      ptr->next = new_node;     //  the ex-last node is connected with the new
node
      new_node->next = NULL;    //  and the new node becomes last
                 // in circular list, it becomes:  new_node->next = start ;
     }
  }
  printf("\n Linked List Created \n\n");
  return start;
}

void  traverse_list(struct node * start)
{
   struct node *ptr;

   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to work with \n\n"); }

   ptr = start ;
   while (ptr != NULL)      // in circular list, it becomes:  while (ptr->next != start)
   {
      printf("\n value = %d", ptr->data);        // in general it is    use  ptr->data ;
      ptr = ptr->next ;
   }
   // in circular list,  add this line, to process the last node   printf("\n value = %d",
ptr->data);
   // in general it is    use  ptr->data ;
}

int  count_nodes(struct node * start)
{
   struct node *ptr;
   int count = 0 ;
   ptr = start ;
```

```
   while (ptr != NULL)    // in circular list, it becomes:  while (ptr->next != start)
   {
      count++ ;
      ptr = ptr->next ;
   }
   return count;
}

struct node *find_element(struct node * start, int val)
{
   struct node *ptr, *pos;
   ptr = start ;
   while (ptr != NULL)    // in circular list, it becomes:  while (ptr->next != start)
   {
      if (val == ptr->data)       // element is found, we return a pointer showing
the node containing it
      {
         pos = ptr ;
         return pos ;
      }
      else   { ptr = ptr->next ; }
   }
   return NULL;   // element is not found, we return NULL
}

int  find_max(struct node * start)     // in case of min, it is:    int  find_min(struct
node * start)
{
   int max ;          // in case of min, it is:    int min ;
   struct node *ptr ;
   ptr = start ;
   max = start->data ;       // in case of min, it is:    min = start->data ;
   while (ptr != NULL)    // in circular list, it becomes:  while (ptr->next != start)
   {
      if (ptr->data > max)  { max = ptr->data ; }
              // in case of min, it is:    if (ptr->data < min)  { min = ptr->data ; }
```

```
        ptr = ptr->next ;
    }
    // in circular list,  add this line, to check the last node      if (ptr->data  >  max)
{ max = ptr->data ; }
    return max;      // in case of min, it is:    return min ;
}


float  list_average(struct node * start)   // in case of function  list_sum,   it is:  int
list_sum(struct node * start)
{
    int athr = 0, counter = 0 ;
    float ave ;
    struct node *ptr ;
    ptr = start ;
    while (ptr != NULL)             // in circular list,  it becomes:      while (ptr->next
!=  start)
    {
        athr = athr + ptr->data ;
        counter++ ;
        ptr = ptr->next ;
    }
    // in circular list,  add this line, to process the last node  athr = athr + ptr->data
;    counter++ ;

    ave = (float)athr / counter ;
    return ave;      // in case of function  list_sum,   it is:            return  athr ;
}

struct node * insert_at_beginning(struct node * start, int num)
{
    struct node *new_node, *ptr ;
    new_node = (struct node *) malloc(sizeof(struct node));
    new_node->data = num;
    new_node->next = start;

                    //  ptr = start;
```

```
    // On circular       //  while(ptr->next != start)  { ptr = ptr->next; }
     // add these commands   //  ptr->next = new_node;

  start = new_node;
  return start;
}
struct node * insert_at_end(struct node * start, int num)
{
  struct node *ptr, *new_node;
  new_node = (struct node *) malloc(sizeof(struct node));
  new_node->data = num;
  new_node->next = NULL;    // in circular list, it becomes:  new_node->next =
start;
  ptr = start;
  while(ptr->next != NULL)
    { ptr = ptr->next; }
  ptr->next = new_node;
  return start;
}


struct node *insert_after(struct node * start, int num, int val)
{
  struct node *new_node, *ptr, *preptr;
  new_node = (struct node *) malloc(sizeof(struct node));
  new_node->data = num;
  ptr = start;
  preptr = ptr;
  while(preptr->data != val)
  {
    preptr = ptr;
    ptr = ptr->next;
  }
  preptr->next = new_node;
  new_node->next = ptr;
  return start;
```

}


```
}

struct node *insert_before(struct node * start, int num, int val)
{
   struct node *new_node, *ptr, *preptr;
   new_node = (struct node *) malloc(sizeof(struct node));
   new_node->data = num;
   ptr = start;
   while(ptr->data != val)
   {
      preptr = ptr;
      ptr = ptr->next;
   }
   preptr->next = new_node;
   new_node->next = ptr;
   return start;
}


struct node *delete_beg(struct node *start)
{
   struct node *ptr;

   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to delete a node \n\n"); return start ; }

   ptr = start;



   start = start->next;  // On circular       while(ptr->next != start)  { ptr = ptr->next; }
   free(ptr);                          // use these commands  ptr->next = start->next;
                          //                 free(start);  start = ptr->next;
   return start;
}
struct node * delete_end(struct node *start)
```

```
{
   struct node *ptr, *preptr;

   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to delete a node \n\n"); return start ; }

   ptr = start;
   while(ptr->next != NULL)     // On circular it is   while(ptr->next != start)
   {
      preptr = ptr;
      ptr = ptr->next;
   }
   preptr->next = NULL;     // On circular it is   preptr->next = ptr->next;
   free(ptr);
   return start;
}

struct node * delete_node(struct node *start, int val)
{
   struct node *ptr, *preptr;

   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to delete a node \n\n"); return start ; }

   ptr = start;
   if (ptr->data == val)
   {
      start = start->next;
      free(ptr);
      return start;
   }
   else
   {
      while(ptr->data != val && ptr->next != NULL  )  { preptr = ptr;    ptr = ptr-
>next; }
            // On circular   while(ptr->data != val && ptr->next != start)
```

```
        if (ptr->data == val)
        {
        preptr->next = ptr->next;
        free(ptr);
        return start;
        }
        else  {  printf("\n The element does not exist \n\n"); return start; }
   }
}


struct node * delete_after(struct node *start, int val)
{
   struct node *ptr, *preptr;
   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to delete a node \n\n"); return start ; }

   ptr = start;
   preptr = ptr;
   while(ptr->data != val && ptr->next != NULL  )  { preptr = ptr;    ptr = ptr->next; }
            // On circular   while(ptr->data != val && ptr->next != start)
        if (ptr->data == val)
        {
          preptr->next = ptr->next;
          free(ptr);
          return start;
        }
        else  {  printf("\n The element does not exist \n\n"); return start; }
}

struct node *delete_list(struct node *start)     // Delete the entire list
{
   struct node *ptr;             // a new pointer
   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to delete \n\n"); return start ; }
```

```
   ptr = start;                 // ptr points at the start of the list
   while(ptr->next != NULL)    // On circular it is     while(ptr->next != start)
     {start = delete_end(start);}  // delete all nodes (except first), from end to
start, going backwards

   free(start);                 // remove the first node
   start = NULL ;                 // initialize the start pointer
   printf("\n Linked List Deleted \n\n");
   return start;
}




struct dblnode * create_list_double(struct dblnode *start, int num)

{
  struct dblnode *new_node, *ptr;        int i ;
  for (i = 0 ; i < num ; i++)   // loop runs num times, every time one node is put at
the end of the list
  {
     if(start == NULL)   // new node is the first node
     {
        new_node = (struct dblnode*)malloc(sizeof(struct dblnode));  // get
memory for the new node
        start = new_node;
        new_node->data = i;
        new_node->next = NULL ;   // in circular list, it becomes:  new_node->next
= start ;
        new_node->prev = NULL ;   // in circular list, it becomes:  new_node->prev
= start ;
     }
     else    // new node is not the first node,     nodes no 1 to num-1,    insert the
new node at the end of the list
     {
       new_node = (struct dblnode*)malloc(sizeof(struct dblnode)); // get memory
for the new node
```

```
        new_node->data = i;               // put the node's id as date
        ptr = start;
        while(ptr->next != NULL)  { ptr = ptr->next; } // ptr goes to the last node

        ptr->next = new_node; //  the ex-last node is connected with the new node
        new_node->next = NULL;    //  and the new node becomes last
                    // in circular list, it becomes:  new_node->next = start ;
        new_node->prev = ptr;    //  and the new node becomes last
        // in circular list, add this line:    start->prev = new_node;
       }
   }
   printf("\n Doubly Linked List Created \n\n");
   return start;
}
void  traverse_list_double(struct dblnode * start)
{
    struct dblnode *ptr;

    if ( start == NULL )          // the list doesn't exist
        { printf("\n There is no list to work with \n\n"); }


 ptr = start ;
    while (ptr != NULL)      // in circular list, it becomes:  while (ptr->next != start)
    {
        printf("\n value = %d", ptr->data);        // in general it is    use  ptr->data ;
        ptr = ptr->next ;
    }
    // in circular list,  add this line, to process the last node          printf("\n value =
", ptr->data);        // in general it is    use  ptr->data ;
}



int  count_nodes_double(struct dblnode * start)
{
```

```
   struct dblnode *ptr;
   int count = 0 ;
   ptr = start ;
   while (ptr != NULL)    // in circular list, it becomes:  while (ptr->next != start)
   {
      count++ ;
      ptr = ptr->next ;
   }
   // in circular list,  add this line, to count the last node        count++ ;
   return count;
}
struct dblnode * find_element_double(struct dblnode * start, int val)
{
   struct dblnode *ptr, *pos;
   ptr = start ;
   while (ptr != NULL)    // in circular list, it becomes:  while (ptr->next != start)
   {
      if (val == ptr->data)        // element is found, we return a pointer showing
the node containing it
      {
         pos = ptr ;
         return pos ;
      }

else   { ptr = ptr->next ; }
   }
   // in circular list,  add this line, to check the last node         if (val == ptr->data)
{  pos = ptr ;   return pos ;   }
   return NULL;   // element is not found, we return NULL
}

int  find_max_double(struct dblnode * start)     // in case of min, it is:    int
find_min(struct dblnode * start)
{
   int max ;          // in case of min, it is:    int min ;
   struct dblnode *ptr ;
```

```
    ptr = start ;
    max = start->data ;       // in case of min, it is:    min = start->data ;
    while (ptr != NULL) // in circular list, it becomes:  while (ptr->next != start)
    {
        if (ptr->data > max)  // in case of min, it is:    if (ptr->data  <  min)
            { max = ptr->data ; }
        ptr = ptr->next ;
    }
    // in circular list,  add this line, to check the last node      if (ptr->data  >  max)
{ max = ptr->data ; }
    return max;      // in case of min, it is:    return min ;
}
float  list_average_double(struct dblnode * start)
  // in case of function  list_sum,   it is:  int  list_sum_double (struct dblnode *
start)
{
    int athr = 0, counter = 0 ;
    float ave ;
    struct dblnode *ptr ;
    ptr = start ;
    while (ptr != NULL)             // in circular list,  it becomes:      while (ptr->next
!=  start)
    {
        athr = athr + ptr->data ;
        counter++ ;
        ptr = ptr->next ;
    }

    // in circular list,  add this line, to process the last node                              athr
= athr + ptr->data ;     counter++ ;

    ave = (float)athr / counter ;
    return ave;      // in case of function  list_sum,   it is:            return  athr ;
}

struct dblnode * insert_at_beginning_double(struct dblnode * start, int num)
```

```
{
   struct dblnode *new_node, *ptr ;
   new_node = (struct dblnode *) malloc(sizeof(struct dblnode));
   new_node->data = num;
   new_node->prev = NULL;
   new_node->next = start;

                    //  ptr = start;
     // On circular         //  while(ptr->next != start)  { ptr = ptr->next; }
      // add these commands   //  ptr->next = new_node;
                            //  new_node->prev = ptr;

   start->prev = new_node;
   start = new_node;
   return start;
}



struct dblnode * insert_at_end_double(struct dblnode * start, int num)
{
   struct dblnode *ptr, *new_node;
   new_node = (struct dblnode *) malloc(sizeof(struct dblnode));
   new_node->data = num;
   new_node->next = NULL;    // in circular list, it becomes:  new_node->next =
start;
   ptr = start;
   while(ptr->next != NULL)
      { ptr = ptr->next; }
   ptr->next = new_node;
   new_node->prev = ptr;
   // in circular list, add this line:      start->prev = new_node;
   return start;
}
struct dblnode *insert_after_double(struct dblnode * start, int num, int val)
{
```

```
    struct dblnode *new_node, *ptr ;
    new_node = (struct dblnode *) malloc(sizeof(struct dblnode));
    new_node->data = num;
    ptr = start;
    while(ptr->data != val)   { ptr = ptr->next; }
    new_node->prev = ptr;
    new_node->next = ptr->next;
    ptr->next->prev = new_node;
    ptr->next = new_node;
    return start;
}



struct dblnode *insert_before_double(struct dblnode * start, int num, int val)
{
    struct dblnode *new_node, *ptr, *preptr;
    new_node = (struct dblnode *) malloc(sizeof(struct dblnode));
    new_node->data = num;
    ptr = start;
    while(ptr->data != val)   { ptr = ptr->next; }
    new_node->next = ptr;
    new_node->prev = ptr->prev;
    ptr->prev->next = new_node;
    ptr->prev = new_node;
    return start;
}


struct dblnode *delete_beg_double(struct dblnode *start)
{
    struct dblnode *ptr, *komvos;

    if ( start == NULL )          // the list doesn't exist
        { printf("\n There is no list to delete a node \n\n"); return start ; }
```

```
   ptr = start;


   start = start->next;  // On circular       while(ptr->next != start)  { ptr = ptr-
>next; }
   start->prev = NULL;     // use these commands  ptr->next = start->next;
   free(ptr);             //  komvos = start;  start = start–>next;  start–>prev = ptr;
free(komvos);


   return start;
}

struct dblnode * delete_end_double(struct dblnode *start)
{
   struct dblnode *ptr ;

   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to delete a node \n\n"); return start ; }

   ptr = start;
   while(ptr->next != NULL) { ptr = ptr->next; }
                         // On circular it is   while(ptr->next != start)  { ptr = ptr-
>next; }
   ptr->prev->next = NULL;    // On circular it is   ptr->prev->next = start;
   free(ptr);
   return start;
}
struct dblnode * delete_node_double(struct dblnode *start, int val)
{
   struct dblnode *ptr, *komvos;

   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to delete a node \n\n"); return start ; }

   ptr = start;
```

```
    if (ptr->data == val)
    {
        start = start->next;    // On circular   while(ptr->next != start)  { ptr = ptr-
>next; }
        start->prev = NULL;        // use these commands           ptr->next = start -
> next;
        free(ptr);              //                komvos = start;  start = start–>next;
        return start;           //    start–>prev = ptr;   free(komvos);   return start;
    }
    else
    {
        while(ptr->data != val && ptr->next != NULL)    { ptr = ptr->next; }
            // On circular   while(ptr->data != val && ptr->next != start)
         if (ptr->data == val)
         {
         ptr->prev->next = ptr->next;
         ptr->next->prev = ptr->prev;
         free(ptr);
         return start;
         }
         else  {  printf("\n The element does not exist \n\n"); return start; }
    }
}
struct dblnode * delete_after_double(struct dblnode *start, int val)
{
    struct dblnode *ptr, *preptr, *komvos;

    if ( start == NULL )          // the list doesn't exist
        { printf("\n There is no list to delete a node \n\n"); return start ; }

    ptr = start;
    preptr = ptr;
    while(ptr->data != val && ptr->next != NULL)    { ptr = ptr->next; }
            // On circular   while(ptr->data != val && ptr->next != start)
         if (ptr->data == val)
         {
```

```
        komvos = ptr->next;
        ptr->next = komvos->next;
        komvos->next->prev = ptr;
        free(komvos);
        return start;
       }
       else  {  printf("\n The element does not exist \n\n"); return start; }
}

struct dblnode *delete_list_double(struct dblnode *start)     // Delete the entire list
{
   struct dblnode *ptr;             // a new pointer

   if ( start == NULL )          // the list doesn't exist
      { printf("\n There is no list to delete \n\n"); return start ; }

   ptr = start;                  // ptr points at the start of the list

   while(ptr->next != NULL)    // On circular it is     while(ptr->next != start)
     {start = delete_end_double(start);}  // delete all nodes (except first), from
end to start, going backwards

   free(start);                  // remove the first node
   start = NULL ;                 // initialize the start pointer
   printf("\n Linked List Deleted \n\n");
   return start;
}
```

```
int main()
{
    struct node* list; int count ;
   struct node *elist, *alist;
    list  = create_list(NULL, 30);
    print_list_iterative(list);
    printf("\n\n");
    traverse_list(list);
    printf("\n\n");
   count  =  count_nodes(list) ;
   printf("The list has %d nodes", count);
    printf("\n\n");
   elist  =  find_element(list, 4) ;
   alist  =  find_element(list, 64) ;
   if (elist != NULL)  { printf("The element %d found in our list", elist->data); }
   else               { printf("The element 4 has not been found in our list"); }
    printf("\n\n");
   if (alist != NULL)  { printf("The element %d found in our list", alist->data); }
   else               { printf("The element 64 has not been found in our list"); }
    printf("\n\n");
   printf("The maximum element is %d ", find_max(list));
    printf("\n\n");
   printf("The average is %f ", list_average(list));
    printf("\n\n");
```

```
list = insert_at_beginning(list, 22);
print_list_iterative(list);
printf("\n\n");
list = insert_at_end(list, 22);
print_list_iterative(list);
printf("\n\n");



list = insert_after(list, 222, 15);
print_list_iterative(list);
printf("\n\n");
list = insert_before(list, 222, 15);
print_list_iterative(list);
printf("\n\n");
list = delete_beg(list);
print_list_iterative(list);
printf("\n\n");
list = delete_end(list);
print_list_iterative(list);
printf("\n\n");
list = delete_node(list, 5);
print_list_iterative(list);
printf("\n\n");
list = delete_after(list, 7777);
print_list_iterative(list);
printf("\n\n");
list = delete_list(list);
print_list_iterative(list);
printf("\n\n");
```

```
        struct dblnode* dbllist;
          struct dblnode *dblelist, *dblalist;
        dbllist  = create_list_double(NULL, 30);
        print_list_iterative_double(dbllist);
        printf("\n\n");
        traverse_list_double(dbllist);
        printf("\n\n");
      count  =  count_nodes_double(dbllist) ;
      printf("The list has %d nodes", count);
        printf("\n\n");
      dblelist  =  find_element_double(dbllist, 4) ;
      dblalist  =  find_element_double(dbllist, 64) ;
      if (dblelist != NULL)  { printf("The element %d found in our list", dblelist-
>data); }



 else              { printf("The element 4 has not been found in our list"); }
        printf("\n\n");
      if (dblalist != NULL)  { printf("The element %d found in our list", dblalist-
>data); }
      else              { printf("The element 64 has not been found in our list"); }
        printf("\n\n");
      printf("The maximum element is %d ", find_max_double(dbllist));
        printf("\n\n");
      printf("The average is %f ", list_average_double(dbllist));
        printf("\n\n");
        dbllist = insert_at_beginning_double(dbllist, 22);
        print_list_iterative_double(dbllist);
        printf("\n\n");
        dbllist = insert_at_end_double(dbllist, 22);
        print_list_iterative_double(dbllist);
        printf("\n\n");
```

```
    dbllist = insert_after_double(dbllist, 222, 15);
        print_list_iterative_double(dbllist);
        printf("\n\n");
        dbllist = insert_before_double(dbllist, 222, 15);
        print_list_iterative_double(dbllist);
        printf("\n\n");
        dbllist = delete_beg_double(dbllist);
        print_list_iterative_double(dbllist);
        printf("\n\n");
        dbllist = delete_end_double(dbllist);
        print_list_iterative_double(dbllist);
        printf("\n\n");
        dbllist = delete_node_double(dbllist, 5);
        print_list_iterative_double(dbllist);
        printf("\n\n");
        dbllist = delete_after_double(dbllist, 7777);
        print_list_iterative_double(dbllist);
        printf("\n\n");
        dbllist = delete_list_double(dbllist);
        print_list_iterative_double(dbllist);
        printf("\n\n");

        return 0;
}
```

6.2. Η έξοδος του προγράμματος

```
$ gcc -o driver--complete driver--complete.c
$ ./driver--complete


 Linked List Created


0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 ->
15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 ->
28 -> 29 -> null


 value = 0
 value = 1
 value = 2
 value = 3
 value = 4
 value = 5
 value = 6
 value = 7
 value = 8
 value = 9
 value = 10
 value = 11
 value = 12
 value = 13
 value = 14
 value = 15
 value = 16
 value = 17
 value = 18
 value = 19
 value = 20
 value = 21
 value = 22
 value = 23
 value = 24
 value = 25
 value = 26
 value = 27
 value = 28
 value = 29
The list has 30 nodes


The element 4 found in our list

The element 64 has not been found in our list

The maximum element is 29
```

The average is 14.500000

22 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> null

22 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 22 -> null

22 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 222 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 22 -> null

22 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 222 -> 15 -> 222 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 22 -> null

0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 222 -> 15 -> 222 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 22 -> null

0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 222 -> 15 -> 222 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> null

0 -> 1 -> 2 -> 3 -> 4 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 222 -> 15 -> 222 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> null

The element does not exist
0 -> 1 -> 2 -> 3 -> 4 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 222 -> 15 -> 222 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> null

Linked List Deleted
empty list!

Doubly Linked List Created

0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 15 <-> 16 <-> 17 <-> 18 <-> 19 <-> 20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> null

value = 0
value = 1
value = 2
value = 3
value = 4
value = 5
value = 6

value = 7
value = 8
value = 9
value = 10
value = 11
value = 12
value = 13
value = 14
value = 15
value = 16
value = 17
value = 18
value = 19
value = 20
value = 21
value = 22
value = 23
value = 24
value = 25
value = 26
value = 27
value = 28
value = 29
The list has 30 nodes

The element 4 found in our list

The element 64 has not been found in our list

The maximum element is 29

The average is 14.500000

22 <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 15 <-> 16 <-> 17 <-> 18 <-> 19 <-> 20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> null

22 <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 15 <-> 16 <-> 17 <-> 18 <-> 19 <-> 20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> 22 <-> null

22 <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 15 <-> 222 <-> 16 <-> 17 <-> 18 <-> 19 <-> 20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> 22 <-> null

22 <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 222 <-> 15 <-> 222 <-> 16 <-> 17 <-> 18 <-> 19 <-> 20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> 22 <-> null

0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 222 <-> 15 <-> 222 <-> 16 <-> 17 <-> 18 <-> 19 <->

20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> 22 <-> null

0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 222 <-> 15 <-> 222 <-> 16 <-> 17 <-> 18 <-> 19 <-> 20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> null
0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 222 <-> 15 <-> 222 <-> 16 <-> 17 <-> 18 <-> 19 <-> 20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> null

The element does not exist

0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 6 <-> 7 <-> 8 <-> 9 <-> 10 <-> 11 <-> 12 <-> 13 <-> 14 <-> 222 <-> 15 <-> 222 <-> 16 <-> 17 <-> 18 <-> 19 <-> 20 <-> 21 <-> 22 <-> 23 <-> 24 <-> 25 <-> 26 <-> 27 <-> 28 <-> 29 <-> null

Linked List Deleted

empty list!

7. Συμπεράσματα

Ο πίνακας στηρίζεται στις έννοιες του συνόλου και της απεικόνισης (συνάρτησης), ενώ η λίστα στηρίζεται στην έννοια της ακολουθίας. Έτσι, ένας πίνακας θεωρείται σαν μία δομή τυχαίας προσπέλασης, ενώ μία λίστα είναι στην ουσία μία δομή ακολουθίας ή σειριακής προσπέλασης (sequential access). Για να φθάσουμε, δηλαδή, σένα ένα στοιχείο μιας λίστας πρέπει να περάσουμε από όλα τα προηγούμενα ξεκινώντας από το πρώτο.

Δεύτερον, το μέγεθος ενός πίνακα παραμένει σταθερό, ενώ το μέγεθος μιας λίστας όχι, διότι συνήθως απαιτούνται διαγραφές παλαιών και εισαγωγές νέων

στοιχείων. Δηλαδή, μια λίστα είναι μια δυναμική δομή και όχι στατική, όπως ο πίνακας.

Η εξέταση μιας δομής δεδομένων με βάση τη μνήμη θα μπορούσε να αποδώσει καλύτερη εικόνα για την επιλογή μιας συγκεκριμένης δομής δεδομένων ή ακόμα και για τη σχεδίαση μιας νέας δομής δεδομένων. Επίσης δεν απαιτείται κατανάλωση μνήμης καθώς οι λειτουργίες που λαμβάνουν μέρος δεν καταλαμβάνουν επιπλέον χώρο καθιστώντας την κατανομή μνήμης πιο προβλέψιμη για μια δεδομένη είσοδο.

Επομένως, όταν απαιτείται μόνο αναζήτηση στοιχείων, ο πίνακας είναι προτιμότερος, ενώ αν γίνονται συχνά εισαγωγές ή διαγραφές απαιτείται μια δυναμική δομή όπως η συνδεδεμένη λίστα.

8. Βιβλιογραφία

[1] Μανωλόπουλος, Ι. , Δομές Δεδομένων, τόμ. Α’, ART of TEXT, 2η έκδοση, Θεσσαλονίκη 1992.

[2] Cormen, H. T., Leiserson E. C. και Rivest, L. R., Introduction to Algorithms, McGraw-Hill, Cambridge, MA 1996

[3] Main M. και Savitch W., Data Structures and Other Objects, Benjamin/Cummings, Redwood City, CA 1995.

[4] Standish, A. T. , Data Structures, Algorithm and Software Principles, Addison-Wesley, Reading, MA 1994.

[5] Weiss, A. M., Data Structures and Algorithm Analysis, Benjamin/ Cummings, 2nd Edition, Redwood City, CA 1995.

[6] Nicklaus Wirth , Algorithms & Data Structures, 1976.

[7] Summit, Steve. C Programming Notes - Chapter 11: Memory Allocation, 30 October 2011.