10. Γραμμικές δομές δεδομένων

Σύνοψη
Στο κεφάλαιο αυτό μελετώνται διάφορες δομές δεδομένων. Αρχικά παρουσιάζονται οι κατηγορίες και τα χαρακτηριστικά των δομών δεδομένων και ακολούθως μελετώνται οι σειριακές δομές της στοίβας και της ουράς, καθώς και εφαρμογές τους. Στη συνέχεια, εισάγεται η έννοια της συνδεδεμένης λίστας και μελετώνται η απλά και η διπλά συνδεδεμένη λίστα, όπως και η κυκλική λίστα. Στην επόμενη ενότητα αναπτύσσονται οι υλοποιήσεις της στοίβας και της ουράς ως απλά συνδεδεμένες λίστες. Το κεφάλαιο ολοκληρώνεται με ένα εκτενές παράδειγμα ανάπτυξης προγράμματος.

Λέξεις κλειδιά
γραμμικές/στατικές/δυναμικές δομές δεδομένων – κόμβος – στοίβα – ώθηση – εξώθηση – ουρά – εισαγωγή – εξαγωγή – απλά/διπλά συνδεδεμένη λίστα – κυκλική λίστα – δείκτης κεφαλής/ουράς.

Προαπαιτούμενη γνώση
Λεξιλόγιο της γλώσσας C – μεταβλητές – εκφράσεις – τελεστές – έλεγχος ροής προγράμματος – συναρτήσεις – πίνακες – δείκτες – δυναμική διαχείριση μνήμης – δομές – αρχεία

10.1 Γενικά

Στα προηγούμενα κεφάλαια χρησιμοποιήθηκε ο πίνακας ως συλλογή ομοειδών μεταβλητών. Ο πίνακας αποτελεί μία **δομή δεδομένων** (data structure) και μάλιστα **γραμμική** (linear), καθώς του *i*-στου στοιχείου του πίνακα προηγείται ένα μόνο στοιχείο (το στοιχείο *i*-1) και έπεται ένα μόνο στοιχείο (το στοιχείο *i*+1). Εάν σε κάποια δομή δεδομένων τα στοιχεία της συνδέονται με περισσότερα του ενός επόμενα ή/και με περισσότερα του ενός προηγούμενα, τότε η δομή ονομάζεται **μη γραμμική** (nonlinear). Γενικά, μία δομή δεδομένων αναπαριστά οντότητες του φυσικού κόσμου μέσω ενός αφηρημένου μοντέλου, στο οποίο προσδιορίζονται λειτουργίες (πράξεις). Το μοντέλο αυτό καλείται **αφηρημένος τύπος δεδομένων** (abstract data type) (ΑΤΔ).

Σε ό,τι αφορά τον τρόπο δημιουργίας, οι δομές δεδομένων διακρίνονται στις: *(α)* **στατικές**, όπως ο πίνακας, στις οποίες το μέγεθος της μνήμης καθορίζεται από την αρχή στο πρόγραμμα και όχι κατά τη διάρκεια της εκτέλεσής του και *(β)* στις **δυναμικές**, στις οποίες το μέγεθός τους μεταβάλλεται ανάλογα με τα δεδομένα που εισάγονται ή διαγράφονται κατά τον χρόνο εκτέλεσης του προγράμματος.

Ο προσδιορισμός της θέσης ενός στοιχείου γίνεται είτε με χρήση αριθμοδείκτη (index), όπως στην περίπτωση των πινάκων, όπου τα δεδομένα είναι σειριακά αποθηκευμένα, είτε με χρήση δείκτη (pointer), ο οποίος χρησιμοποιείται για να συνδέει δύο διαδοχικά στοιχεία σε μία δομή δεδομένων, όταν αυτά δεν είναι σειριακά αποθηκευμένα. Η δομή της πρώτης περίπτωσης ονομάζεται **σειριακή γραμμική λίστα** (sequential linear list) και η δομή της δεύτερης περίπτωσης καλείται **συνδεδεμένη γραμμική λίστα** (linked linear list).

Η βασική μονάδα των δομών δεδομένων – το «στοιχείο» της δομής δεδομένων – είναι ο **κόμβος** (node). Έως τώρα στους πίνακες ο κόμβος ταυτιζόταν με την τιμή ενός δεδομένου, καθώς το όνομα του πίνακα παρείχε το πρώτο byte του χώρου αποθήκευσης του πίνακα και ο αριθμοδείκτης του στοιχείου, που είχε ως τιμή το συγκεκριμένο δεδομένο, πληροφορούσε για το προηγούμενο και το επόμενο στοιχείο του πίνακα. Ωστόσο, στη γενική περίπτωση ένας κόμβος αποτελείται από δύο τμήματα: *(α)* το τμήμα της πληροφορίας ή δεδομένου (data), στο οποίο φιλοξενείται η τιμή του δεδομένου και *(β)* το τμήμα της διεύθυνσης (next), στο οποίο φιλοξενείται η διεύθυνση του επόμενου κόμβου με τον οποίο συνδέεται ο παρών. Η μορφή του κόμβου απεικονίζεται στο **Σχήμα 10.1**:

Σχήμα 10.1 *Απεικόνιση του κόμβου*

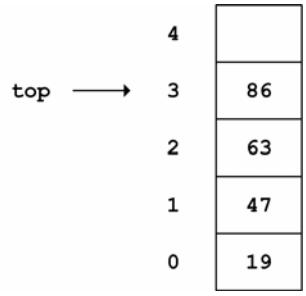
Οι βασικές λειτουργίες που επιτελούνται σε μία δομή δεδομένων, είναι οι ακόλουθες:

- *Προσπέλαση κόμβου* (access): Πρόσβαση σε κόμβο της δομής με σκοπό τη χρήση ή την τροποποίηση του περιεχομένου του.
- *Εισαγωγή κόμβου* (insertion): Προσθήκη νέου κόμβου σε υφιστάμενη δομή ή δημιουργία νέας δομής με την εισαγωγή του πρώτου κόμβου.
- *Διαγραφή κόμβου* (deletion): Αφαίρεση κόμβου από δομή.
- *Αναζήτηση* (searching): Εύρεση ενός ή περισσότερων κόμβων που περιέχουν συγκεκριμένη πληροφορία, η οποία αποτελεί το κλειδί της αναζήτησης.
- *Μεταβολή /τροποποίηση* (modification) του περιεχομένου ενός κόμβου.
- *Προσάρτηση* (append): Εισαγωγή κόμβου στο τέλος της δομής. Ο νέος κόμβος είναι ίδιας μορφής με εκείνους της δομής.
- *Ταξινόμηση* (sorting): Διάταξη των κόμβων μίας δομής κατά αύξουσα ή φθίνουσα σειρά.
- *Συγχώνευση* (merging): Συνένωση δύο ή περισσότερων δομών σε μία δομή, βάσει κανόνων τοποθέτησης των κόμβων στη νέα δομή.
- *Διαχωρισμός* (separation): Διάσπαση μίας δομής σε δύο ή περισσότερες.

10.2 Στοίβα

Η στοίβα (stack) είναι μία μορφή οργάνωσης των δεδομένων, στην οποία το δεδομένο που τοποθετείται τελευταίο στη στοίβα, ανασύρεται πρώτο. Μπορούμε να την παραλληλίσουμε με μία στοίβα από πιάτα, όπου κάθε νέο πιάτο τοποθετείται στην κορυφή (top) και χρησιμοποιείται πρώτο. Αυτή η μέθοδος επεξεργασίας ονομάζεται **LIFO** (**L**ast **I**n **F**irst **O**ut).

Εάν η στοίβα υλοποιηθεί με πίνακα (στατική στοίβα), απεικονίζεται ως εξής (αριστερά των κελιών υπάρχει η αρίθμησή τους):



Σχήμα 10.2 *Απεικόνιση στοίβας*

Ο **top** είναι ένας δείκτης που δείχνει στην κορυφή της στοίβας. Από την απεικόνιση της στοίβας στο **Σχήμα 10.2** προκύπτει ότι βρίσκονται αποθηκευμένα δεδομένα στις πρώτες **4** θέσεις, άρα ο **top** δείχνει στην τέταρτη θέση, δηλαδή θα αποκτήσει την τιμή **3** (η αρίθμηση των θέσεων ξεκινά από το **0**). Για να προσπελαστεί το δεδομένο **47**, πρέπει πρώτα να αφαιρεθούν τα δεδομένα **86** και **63**, ώστε το **47** να βρεθεί στην κορυφή της στοίβας. Αντίστοιχα, εάν στα υπάρχοντα τέσσερα δεδομένα προστεθεί ένα νέο δεδομένο, αυτό θα αποθηκευτεί στην $5^{η}$ θέση, δηλαδή ο **top** θα μετακινηθεί κατά μία θέση προς τα πάνω.

Θα πρέπει να σημειωθεί ότι στην υλοποίηση της στοίβας με πίνακα, ο πίνακας αποτελεί μόνο το μέσο αποθήκευσης. Η λειτουργία της στοίβας διαφέρει από εκείνη ενός κλασικού πίνακα, καθώς αφενός μεν δεν έχει σταθερό πλήθος στοιχείων, αφετέρου δε μόνο το δεδομένο που βρίσκεται στην κορυφή της στοίβας μπορεί να προσπελαστεί.

Δύο είναι οι κύριες λειτουργίες στη στοίβα:

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

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

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

Η υλοποίηση των δύο λειτουργιών δίνεται ακολούθως, όπου χάριν ευκολίας θεωρείται ότι τα δεδομένα είναι τύπου ακεραίου:

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

#define N 100          /* Το μέγεθος του πίνακα για την αποθήκευση της
                          στοίβας */

void push(int stack[], int *t, int obj);
int pop(int stack[], int *t);
/*-------------------------------------*/
int main()
{
  int stack[N];
  int getStackValue;      /* Μεταβλητή υποδοχής των δεδομένων μετά
                             από pop */
  int top=-1;             /* Αρχικοποίηση του δείκτη, ώστε με την
                             έναρξη εισαγωγής δεδομένων ο δείκτης να
                             λάβει την τιμή 0  */
  ............
  push(stack,&top,<κάποιος ακέραιος>);
  ............
  push(stack,&top,<κάποιος ακέραιος>);
  ............
  getStackValue==pop(stack,&top);
  ............
  return 0;
}
/*-------------------------------------*/
void push(int stack[], int *t, int obj)
{
  if ((*t)==(N-1))
  {
     printf( "ERROR! Stack overflow...\n" );
     getchar();
     abort();
  }
  else  stack[++(*t)] = obj;
}
/*-------------------------------------*/
int pop(int stack[], int *t)
{
  int r;
  if ((*t)<0)
  {
     printf( "ERROR! Empty stack, unable to pop...\n" );
     getchar();
```

```
            abort();
        }
        else
            r=stack[(*t)--];
        return(r);
    }
```

10.2.1 Παράδειγμα

Το πρόγραμμα που ακολουθεί, υλοποιεί τον έλεγχο σωστής χρήσης των παρενθέσεων στις αριθμητικές εκφράσεις. Για παράδειγμα, η παράσταση **(a*(b+c)+d)** χρησιμοποιεί σωστά τις παρενθέσεις, ενώ οι παραστάσεις **(a*b+c)+d)**, **(a-b))*b+c)** δεν χρησιμοποιούν σωστά τις παρενθέσεις, γιατί πλεονάζουν δεξιές παρενθέσεις. Αντίστοιχα, η παράσταση **(a*(b+c+d)** δεν τις χρησιμοποιεί σωστά, γιατί πλεονάζει μία αριστερή παρένθεση.

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

#define N 10

void push(char stack[], int *t, char obj);
/*Εφόσον τα στοιχεία που αποσύρονται από τη στοίβα δεν χρησιμοποιούνται, η συνάρτηση pop δεν χρειάζεται να έχει επιστρεφόμενη τιμή */
void pop(char stack[], int *t);

int main()
{
  char stack[N],expr[30];
  int i,error,top=-1;

  printf( "Input expression, ENTER to quit:\n" );
  gets(expr);
  error=0;
  i=0;
  while ((expr[i]!='\0') && (error==0))
  {
     switch (expr[i])
     {
        case '(':
          push(stack,&top,expr[i]);
        break;
        case ')':
          if (top==-1)
             error=1;
          else
             pop(stack,&top);
        break;
     }
     i++;
  }
  if (top==-1)
     if (error==0)
        printf( "Correct expression." );
     else
```

```
        printf( "ERROR! Too many right parentheses." );
    else
      printf( "ERROR. Too many left parentheses." );

    return 0;
}
/*-----------------------------------*/
void push(char stack[], int *t, char obj)
{
  if ((*t)==(N-1))
  {
    printf( "ERROR! Stack overflow...\n" );
    getchar();
    abort();
  }
  else stack[++(*t)] = obj;
}
/*-----------------------------------*/
void pop(char stack[], int *t)
{
  char r;
  if ((*t)<0)
  {
    printf( "ERROR! Empty stack, unable to pop...\n" );
    getchar();
    abort();
  }
  else
    r=stack[(*t)--];
}
```

(α) *(β)* *(γ)*

Εικόνα 10.1 *Οι τρεις περιπτώσεις αποτελεσμάτων του προγράμματος του παραδείγματος 10.2.1*

10.2.2 Παράδειγμα

Μία κλασική εφαρμογή που χρησιμοποιεί τη δομή της στοίβας, είναι η επεξεργασία από τις γλώσσες προγραμματισμού αριθμητικών εκφράσεων, όπως **2+3 ή 2*(3+4)** ή ακόμα **((2+4)*7)+3*(9-5)**. Αυτή η μορφή παράστασης ονομάζεται *ένθετη* (infix) και έχει το χαρακτηριστικό ότι οι τελεστές τοποθετούνται μεταξύ των τελεσταίων (operands).

Κάθε ένθετη παράσταση χρησιμοποιεί προτεραιότητες στη σειρά εκτέλεσης των πράξεων. Π.χ. στην παράσταση **3+4*5** έχει προτεραιότητα η πράξη του πολλαπλασιασμού. Βέβαια, οι προτεραιότητες τροποποιούνται με τη χρήση παρενθέσεων, π.χ. **2*(3+4)** προτεραιότητα έχει η πράξη μέσα στις παρενθέσεις, δηλαδή η πρόσθεση. Οι προτεραιότητες των τελεστών παρατίθενται στον **Πίνακα 2.7**.

Εάν, όμως, ο μεταφραστής προσπαθήσει να εκτελέσει μία παράσταση στη ένθετη μορφή της, όπως π.χ. την παράσταση **3+4*5**, όταν φτάσει στον τελεστή της πρόσθεσης, δεν γνωρίζει εάν θα πρέπει να εκτελέσει την πρόσθεση ή να την αναβάλει για αργότερα, γιατί πιθανόν να προηγείται κάποια άλλη πράξη. Για τον λόγο αυτόν ο μεταφραστής ακολουθεί τις ακόλουθες δύο διαδικασίες για τον σωστό υπολογισμό των εκφράσεων:

- Η ένθετη μορφή της αριθμητικής έκφρασης μεταφράζεται σε επιθεματική (postfix) μορφή, όπου οι τελεστές εμφανίζονται μετά τους τελεσταίους. Π.χ. η παράσταση **2+3** γίνεται **23+** και η παράσταση **3+4*5** μετατρέπεται σε **345*+**.
- Η επιθεματική μορφή χρησιμοποιείται, κατόπιν, για τον υπολογισμό της έκφρασης. Η επιθεματική μορφή ονομάζεται και *Αντίστροφη Πολωνική Σημειογραφία* (Reverse Polish Notation – RPN).

Για τη μετατροπή μιας ένθετης μορφής σε επιθεματική ακολουθείται μία διαδικασία σάρωσης της αριθμητικής έκφρασης και εφαρμογής των παρακάτω κανόνων:

1. Αν το στοιχείο είναι τελεσταίος, τότε τοποθετείται στα δεξιά της επιθεματικής μορφής.
2. Αν το στοιχείο είναι τελεστής, τότε κατευθύνεται σε μία στοίβα. Εκεί συγκρίνεται με τον τελεστή της κορυφής της στοίβας.
3. Αν ο εισερχόμενος τελεστής έχει μεγαλύτερη προτεραιότητα από τον τελεστή της κορυφής της στοίβας, τότε ο τελεστής τοποθετείται στη στοίβα (*push*).
4. Αν η προτεραιότητα του εισερχόμενου τελεστή είναι μικρότερη ή ίση από την προτεραιότητα του τελεστή της κορυφής της στοίβας, τότε:
   - Εξάγονται όλοι οι τελεστές της στοίβας με προτεραιότητα μεγαλύτερη ή ίση από την προτεραιότητα του νέου τελεστή (*pop*) και τοποθετούνται στα δεξιά της επιθεματικής μορφής.
   - Ο νέος τελεστής τοποθετείται στη στοίβα (*push*).

Όταν η αριθμητική έκφραση έχει παρενθέσεις, τότε ακολουθούνται οι παρακάτω κανόνες:

1. Η αριστερή παρένθεση τοποθετείται αμέσως στη στοίβα (*push*).
2. Η δεξιά παρένθεση προκαλεί την εξαγωγή (*pop*) όλων των τελεστών, μέχρι να συναντηθεί η αριστερή παρένθεση στη στοίβα. Ύστερα, οι δύο παρενθέσεις αγνοούνται.

Για παράδειγμα, η παράσταση **(2*(3/2+4)-3)/2** μετατρέπεται σε **232/4+*3-2/** σύμφωνα με τα ακόλουθα βήματα:

Βήμα	**Σάρωση ενθεματικής παράστασης**	**Στοίβα**	**Επιθεματική μορφή**
1	**(**	**(**	
2	**2**	**(**	**2**
3	*****	**(***	**2**
4	**(**	**(*(**	**2**
5	**3**	**(*(**	**23**
6	**/**	**(*(/**	**23**
7	**2**	**(*(/**	**232**
8	**+**	**(*(/+**	**232/**
9	**4**	**(*(/+**	**232/4**
10	**)**	**(***	**232/4+**
11	**-**	**(-**	**232/4+***
12	**3**	**(-**	**232/4+*3**
13	**)**		**232/4+*3-**
14	**/**	**/**	**232/4+*3-**
15	**2**	**/**	**232/4+*3-2**
16	**τέλος**		**232/4+*3-2/**

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

#define N 100

void push(char stack[],int *t, char obj);
char pop(char stack[],int *t);
void gotoper(char stack[], int *t, char charVar, int intVar, char
postfix[], int *k);
void gotparen(char stack[], int *t, char postfix[], int *k);
```

```
int main()
{
  char stack[N];
  int i,j,top=-1;
  char infix[100],postfix[100];

  printf( "Input expression, ENTER to quit:\n" );
  gets(infix);
  i=0;
  j=0;
  while (infix[i]!='\0')
  {
     switch (infix[i])
     {
        case '+':
        case '-':
          gotoper(stack,&top,infix[i],3,postfix,&j);
        break;
        case '*':
        case '/':
          gotoper(stack,&top,infix[i],4,postfix,&j);
        break;
        case '(':
          push(stack,&top,infix[i]);
        break;
        case ')':
          gotparen(stack,&top,postfix,&j);
        break;
        default:
          postfix[j++]=infix[i];
        break;
     }
     i++;
  }
  postfix[j]='\0';
  j=0;
  while (postfix[j]!='\0')
  {
     printf( "%c",postfix[j] );
     j++;
  }

  return 0;
}
/*-------------------------------------*/
/*Για λόγους εξοικονόμησης χώρου, τα σώματα των συναρτήσεων push/pop
παραλείπονται, καθώς ταυτίζονται με εκείνα του Παραδείγματος 10.2.1   */
/*-------------------------------------*/
void gotoper(char stack[], int *t, char charVar, int intVar, char
postfix[], int *k)
{
  int top,prectop,done=0;
  char optop;
  top=*t;
  while((top!=-1) && (done==0))
```

```
   {
      if (stack[top]=='(')
      {
         prectop=1;
         done=1;
      }
      else
      {
         if ((stack[top]=='+') || (stack[top]=='-'))
            prectop=3;
         else
            prectop=4;
         if (prectop<intVar)
            done=1;
         else
            postfix[(*k)++]=pop(stack,&top);
      }
   }
   push(stack,&top,charVar);
   *t=top;
}
/*-------------------------------------*/
void gotparen(char stack[], int *t, char postfix[], int *k)
{
   int top,done=0;
   char cs;
   top=*t;
   while ((top!=-1) && (done==0))
   {
      cs=pop(stack,&top);
      if (cs=='(')
         done=1;
      else
         postfix[(*k)++]=cs;
   }
   *t=top;
}
```

10.3 Ουρά

Την έννοια της ουράς τη συναντάμε συχνά στην καθημερινή μας ζωή, π.χ. ουρά αναμονής με ανθρώπους. Το άτομο που είναι πρώτο στην ουρά, εξυπηρετείται και εξέρχεται. Το άτομο που μόλις καταφτάνει, τοποθετείται στο τέλος της ουράς. Αυτή η μέθοδος αυτή επεξεργασίας ονομάζεται **FIFO** (**F**irst **I**n **F**irst **O**ut).

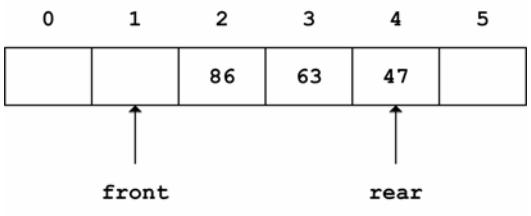
Δύο είναι οι κύριες λειτουργίες στην ουρά:

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

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

Για λόγους προγραμματιστικής διευκόλυνσης ο δείκτης **rear** δείχνει πάντα στο τελευταίο στοιχείο, ενώ ο δείκτης **front** δείχνει μία θέση πριν το πρώτο στοιχείο και, κατά συνέπεια, η ισότητα των δύο δεικτών αποδεικνύει ότι η ουρά είναι άδεια.

Εάν η ουρά υλοποιηθεί με πίνακα (στατική ουρά), απεικονίζεται στο **Σχήμα 10.2**. Θα πρέπει να τονιστεί ότι, κατ' αντιστοιχία με τη δομή της στοίβας, στην υλοποίηση της ουράς με πίνακα ο πίνακας αποτελεί μόνο το μέσο αποθήκευσης.



Σχήμα 10.3 *Απεικόνιση ουράς*

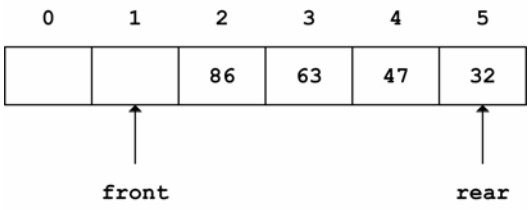
Όταν δημιουργείται η ουρά και πριν δεχτεί κάποιο δεδομένο, οι δύο δείκτες έχουν τιμή **-1**. Στο στιγμιότυπο της ουράς, που παρουσιάζεται στην **Εικόνα 10.3**, ο πίνακας διαθέτει **6** θέσεις (η πρώτη θέση έχει δείκτη θέσης **0**). Έχουν γίνει **5** εισαγωγές δεδομένων και **2** εξαγωγές.

Η υλοποίηση των δύο λειτουργιών δίνεται ακολούθως, όπου χάριν ευκολίας θεωρείται ότι τα δεδομένα είναι τύπου ακεραίου:

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

#define N 100        /* Το μέγεθος του πίνακα για την αποθήκευση της
                        ουράς*/

void push(int stack[], int *t, int obj);
int pop(int stack[], int *t);
/*-------------------------------------*/
int main()
{
  int q[N];
  int getStackValue;      /* Μεταβλητή υποδοχής των δεδομένων μετά
                             από pop */
  int front=-1,rear=-1;   /* Αρχικοποίηση των δεικτών, ώστε με την
                             έναρξη εισαγωγής και εξαγωγή δεδομένων
                             οι δείκτες να λάβουν τιμές συμβατές με
                             αριθμοδείκτες πίνακα */
  ............
  enqueue(q,&rear,<κάποιος ακέραιος>);
  ............
  enqueue(q,&rear,<κάποιος ακέραιος>);
  ............
  dequeue(q, int *front, int  rear);
  ............
  return 0;
}
/*-------------------------------------*/
void enqueue(int q[], int *r, int obj)
{
  if (*r==N-1)
  {
    printf( "ERROR! Queue is full..." );
    getchar();
  }
  else
```

```
    q[++(*r)]=obj;
}
/*-------------------------------------*/
void dequeue(int q[], int *f, int r)
{
  int x;
  if (*f==r)
    printf( "ERROR! Queue is empty...\n" );
  else
  {
    x=q[++(*f)];
    printf("%d has been deleted...",x);
  }
}
```

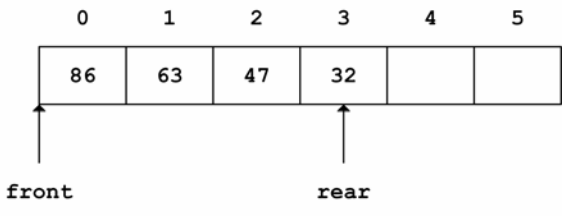
Η υλοποίηση της ουράς με πίνακα έχει ένα μειονέκτημα. Υπάρχει περίπτωση ο **rear** να φθάσει στο πάνω όριο του πίνακα, αλλά στην ουσία να μην υπάρχει υπερχείλιση (επειδή ο **front** θα έχει αυξηθεί), όπως φαίνεται στο **Σχήμα 10.4**.



Σχήμα 10.4 *Φαινομενική υπερχείλιση ουράς*

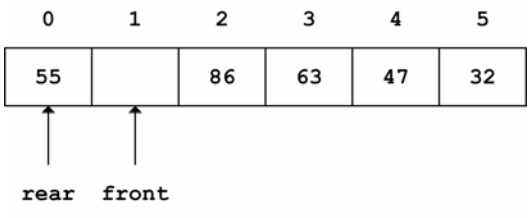
Στην πραγματικότητα, η υπερχείλιση είναι φαινομενική, καθώς στις αρχικές θέσεις του πίνακα, από τις οποίες έχουν γίνει εξαγωγές, υπάρχει ελεύθερος χώρος για την εισαγωγή νέων στοιχείων. Μία λύση θα ήταν να μεταφερθούν τα στοιχεία στο αριστερό άκρο του πίνακα (**Σχήμα 10.5**):

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



Σχήμα 10.5 *Άρση της φαινομενικής υπερχείλισης της ουράς με μετακίνηση στοιχείων στην αρχή της ουράς*

Μία πιο αποτελεσματική υλοποίηση θα ήταν η ουρά να αναδιπλώνεται, δηλαδή όταν ο **rear** = **N-1**, να επανατοποθετείται στο **0**. Η δομή αυτή ονομάζεται **κυκλική ουρά** *(circular queue)*. Στην ουρά του **Σχήματος 10.4** γίνεται εισαγωγή του δεδομένου **55**, το οποίο τοποθετείται πλέον στην αρχή του πίνακα που φιλοξενεί την ουρά, όπως φαίνεται στο **Σχήμα 10.6**.



Σχήμα 10.6 *Κυκλική ουρά*

10.3.1 Παράδειγμα

Το πρόγραμμα που ακολουθεί, αφορά στην εξυπηρέτηση αυτοκινήτων σε διόδια με χρήση ουράς. Συγκεκριμένα:

1. Η συνάρτηση `int display_menu()` υλοποιεί το παρακάτω μενού:

MENU	
======	
1.Car arrival	*(επιλογή για νέα άφιξη αυτοκινήτου)*
2.Car departure	*(επιλογή για αναχώριση αυτοκινήτου)*
3.Queue state	*(επιλογή για την εμφάνιση της τρέχουσας κατάστασης της ουράς)*
0.Exit	*(επιλογή για έξοδο)*

3. Η επιλογή θα επιστρέφεται στη `main()` και θα επιτελούνται οι ακόλουθες λειτουργίες:

(i) **Επιλογή 1:** Θα πληκτρολογούνται τα στοιχεία της πινακίδας του αυτοκινήτου σε αλφαριθμητικό `plate`, το οποίο θα τοποθετείται στο τέλος της ουράς, με χρήση της συνάρτησης

```
void enqueue(char **q, int *rear, char *obj)
```

(ii) **Επιλογή 2:** Το αυτοκίνητο που βρίσκεται πρώτο στην ουρά, θα διαγράφεται μαζί με ένα ανάλογο μήνυμα επιβεβαίωσης, με χρήση της συνάρτησης

```
void dequeue(char **q, int *front, int rear, int length)
```

όπου `length` το μήκος του αλφαριθμητικού που φιλοξενεί την πινακίδα (**7** αριθμοί και γράμματα + πιθανό κενό + μηδενικός χαρακτήρας = **9**)

(iii) **Επιλογή 3:** Θα εμφανίζονται, με τη σειρά, οι πινακίδες των αυτοκινήτων που παραμένουν στην ουρά, για να εξυπηρετηθούν, με χρήση της συνάρτησης

```
void printqueue(char **q, int front, int rear)
```

(iv) **Επιλογή 4:** Το πρόγραμμα θα τερματίζεται.

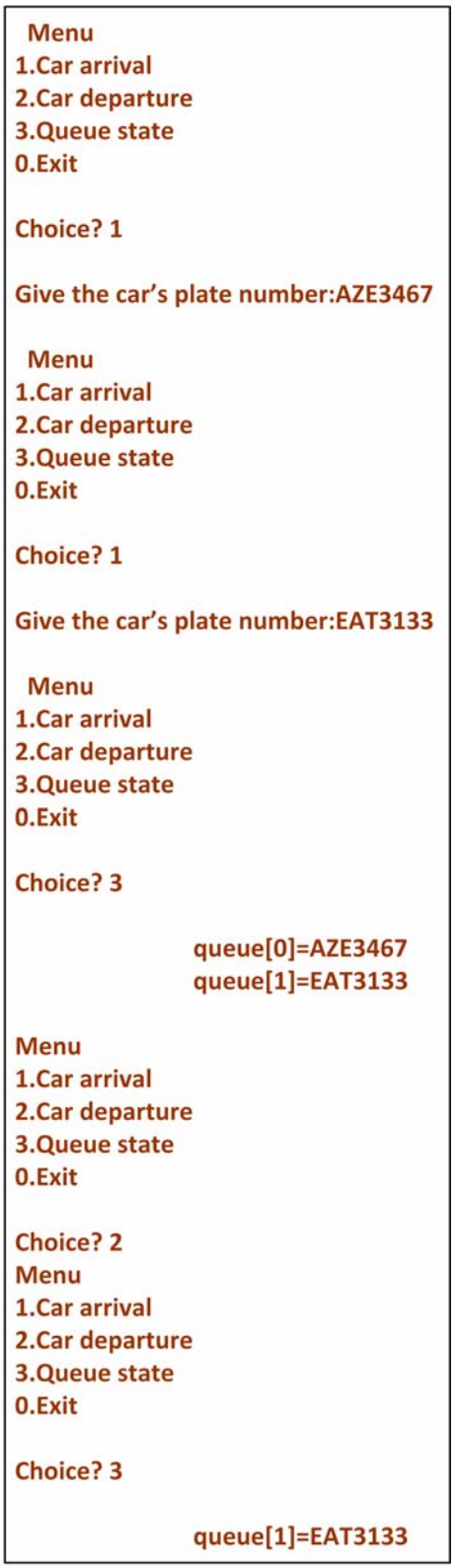
4. Οι πίνακες χαρακτήρων θα υλοποιηθούν με δυναμική δέσμευση μνήμης.

Εφόσον τα δεδομένα είναι αλφαριθμητικά, η ουρά θα αποθηκευτεί σε δισδιάστατο δυναμικό πίνακα χαρακτήρων και το δεδομένο «πινακίδα» σε μονοδιάστατο δυναμικό πίνακα. Το συνολικό πρόγραμμα είναι το ακόλουθο, ενώ η **Εικόνα 10.2** φιλοξενεί ένα στιγμιότυπο των αποτελεσμάτων:

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

#define PLATE_LENGTH 9
#define N 100

void enqueue(char **q, int *rear, char *obj);
```

```
void dequeue(char **q, int *front, int rear);
void printqueue(char **q, int front, int rear);
int display_menu();
/*-------------------------------------*/
int main()
{
  int i,front=-1,rear=-1,choice;
  char **q,*plate;

  plate=(char *)malloc(PLATE_LENGTH*sizeof(char));
  assert(plate!=NULL);
  q=(char **)malloc(N*sizeof(char *));
  assert(q!=NULL);
  for (i=0;i<N;i++)
  {
     q[i]=(char *)malloc(PLATE_LENGTH*sizeof(char));
     assert(q[i]!=NULL);
  }

  do
  {
     choice=display_menu();
     switch (choice)
     {
        case 1:
          printf( "\nGive the car's plate number:" );
          scanf( "%s",plate );
          enqueue(q,&rear,plate);
        break;
        case 2:
          dequeue(q,&front,rear);
        break;
        case 3:
          printqueue(q,front,rear);
        break;
        case 0:
          printf( "\nExiting ...\n" );
        break;
     }
  } while (choice!=0);

  for (i=(N-1);i>=0;i--)
     free(q[i]);
  free(q);
  free(plate);

  return 0;
}
/*-------------------------------------*/
int display_menu()
{
  int choice;
  do
  {
     printf( "\n\n  Menu:\n" );
```

```
            printf( "1.Car arrival\n" );
            printf( "2.Car departure\n" );
            printf( "3.Queue state\n" );
            printf( "0.Exit\n" );
            printf( "\nChoice? " );
            scanf( "%d",&choice );
          }  while  ((choice!=0)  &&  (choice!=1)  &&  (choice!=2)  &&
(choice!=3));
          return(choice);
        }
        /*-------------------------------------*/
        void enqueue(char **q, int *rear, char *obj)
        {
          if ((*rear)==(N-1))
          {
            printf( "ERROR! Queue is full..." );
            getchar();
          }
          else strcpy(q[++(*rear)],obj);
        }
        /*-------------------------------------*/
        void dequeue(char **q, int *front, int rear)
        {
          if ((*front)==rear)
            printf( "ERROR! Queue is empty...\n" );
          else
            printf( "%s has been served and deleted from the queue...",
q[++(*front)] );
        }
        /*-------------------------------------*/
        void printqueue(char **q, int front, int rear)
        {
          int i;
          for (i=(front+1);i<=rear;i++)
            printf( "\n\tqueue[%d]=%s",i,q[i] );
        }
```



Εικόνα 10.2 *Η έξοδος του προγράμματος του παραδείγματος 10.3.1*

10.4 Συνδεδεμένες λίστες

Οι στατικές δομές που μελετήθηκαν, παρουσιάζουν προβλήματα στην εισαγωγή και διαγραφή κόμβων, όταν αυτοί δε βρίσκονται στην αρχή ή το τέλος τους, καθώς και στην αποδοτική εκμετάλλευση της διαθέσιμης μνήμης. Κύριο χαρακτηριστικό των συνδεδεμένων λιστών (linked lists) είναι ότι αποτελούν ακολουθίες κόμβων, οι οποίοι πιθανότατα βρίσκονται σε απομακρυσμένες θέσεις μνήμης και η σύνδεσή τους γίνεται με δείκτες. Κατ' αυτόν τον τρόπο, η εισαγωγή και διαγραφή κόμβων σε οποιαδήποτε θέση της λίστας γίνεται πολύ πιο απλά. Ένα άλλο θετικό χαρακτηριστικό είναι ότι δεν απαιτείται εκ των προτέρων καθορισμός του μέγιστου αριθμού κόμβων της λίστας και η λίστα μπορεί να επεκταθεί ή να συρρικνωθεί κατά την εκτέλεση του προγράμματος (δυναμική δομή δεδομένων).

Οι κυριότερες κατηγορίες λιστών είναι:

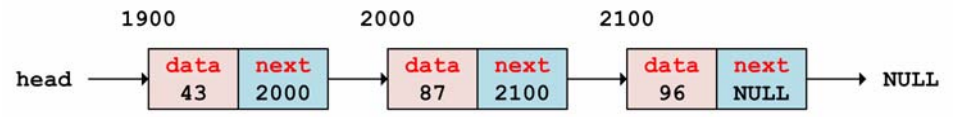
- Η απλά συνδεδεμένη λίστα (simply linked list).
- Η διπλά συνδεδεμένη λίστα (doubly linked list).
- Η κυκλική λίστα (circular list).

10.4.1 Απλά συνδεδεμένη λίστα

Η απλά συνδεδεμένη λίστα αποτελείται από μία αλυσίδα κόμβων, καθένας εκ των οποίων συνδέεται με τον επόμενο μέσω του τμήματος διεύθυνσης (next), δηλαδή το περιεχόμενο του τμήματος διεύθυνσης είναι η διεύθυνση του επόμενου κόμβου. Τοποθετείται ένας **δείκτης κεφαλής** (head) στον πρώτο κόμβο για να προσπελαύνεται η λίστα, ενώ ο δείκτης του τελευταίου κόμβου δείχνει στο **NULL**, για να εντοπίζεται το τέλος της λίστας. Ο κομβος υλοποιείται με την ακόλουθη δομή:

```
struct node
{
  <τύπος δεδομένου> <όνομα μεταβλητής>;
   struct node *next;
};
typedef struct node *PTR;
```

Ένα παράδειγμα απλά συνδεδεμένης λίστας με τρεις κόμβους που περιέχουν ακεραίους, δίνεται στο **Σχήμα 10.7**:



Σχήμα 10.7 *Απλά συνδεδεμένη λίστα τριών κόμβων*

Ο πρώτος κόμβος αποθηκεύεται στη διεύθυνση **1900**, στην οποία δείχνει ο **head**, και το μέλος **data** έχει περιεχόμενο το **43**. Ο πρώτος κόμβος συνδέεται με τον δεύτερο μέσω του μέλους **next**, το οποίο έχει ως περιεχόμενο τη διεύθυνση του δεύτερου κόμβου, **2000**. Με τον ίδιο τρόπο γίνεται η σύνδεση με τον τρίτο κόμβο, ο οποίος είναι τερματικός, καθώς το μέλος **next** έχει τιμή το **NULL**. Συνεπώς, η λίστα έρχεται σε επικοινωνία με το πρόγραμμα μόνο μέσω του δείκτη **head**.

Οι βασικές λειτουργίες της απλά συνδεδεμένης λίστας υλοποιούνται ακολούθως, όπου χάριν ευκολίας τα δεδομένα θεωρούνται ακέραιοι αριθμοί:

(α) **Δημιουργία λίστας με την προσθήκη ενός ή περισσότερων κόμβων**

```
PTR createList(PTR head)
{
  PTR current;
  int x;
  printf( "Give an integer, 0 to stop: " );
  scanf( "%i",&x );
```

```
  if (x==0)
     return NULL;
  else
  {
     /* Δημιουργία του πρώτου κόμβου*/
     head=malloc(sizeof(struct node));
     head->data=x;
     current=head;
     printf("Give an integer, 0 to stop:" );
     scanf( "%i",&x );
     while (x!=0) /* Εισαγωγή περισσότερων κόμβων. Κάθε νέος κόμβος
                     συνδέεται με το μέλος next του προηγούμενου */
     {
        current->next=malloc(sizeof(struct node));
        current=current->next;
        current->data=x;
        printf( "Give an integer, 0 to stop:" );
        scanf( "%i",&x );
     }
     current->next=NULL;  /* Σύνδεση του τερματικού κόμβου με το
                             NULL, είτε η λίστα αποτελείται από έναν
                             μόνο κόμβο είτε έχουν εισαχθεί
                             περισσότεροι. */
  }
  return head; /* Επιστροφή στην καλούσα συνάρτηση της διεύθυνσης
                  του πρώτου κόμβου της λίστας*/
}
```

(β) **Εισαγωγή στοιχείου σε διατεταγμένη λίστα (οι κόμβοι είναι τοποθετημένοι κατά αύξουσα τιμή των δεδομένων τους)**

```
PTR insertToList(PTR head, int x)
{
  PTR current,previous,newnode;
  int found;
  /* Δημιουργία νέου κόμβου */
  newnode=malloc(sizeof(struct node));
  newnode->data=x;
  newnode->next=NULL;
  /* Εάν η λίστα είναι άδεια, ο νέος κόμβος είναι η κεφαλή της. */
  if (head==NULL)
     head=newnode;
  /* Εάν η λίστα δεν είναι άδεια, αναζητείται το σημείο στο οποίο
     θα παρεμβληθεί ο νέος κόμβος, ώστε η λίστα να παραμείνει
     διατεταγμένη.  */
  else
  /* Περίπτωση τοποθέτησης του νέου κόμβου στην κεφαλή  */
     if (newnode->data < head->data)
     {
        newnode->next=head;
        head=newnode;
     }
  /* Περίπτωση τοποθέτησης του νέου κόμβου ανάμεσα σε δύο
     υφιστάμενους ή στο τέλος της λίστας */
     else
     {
```

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

(γ) **Διαγραφή στοιχείου**

```
PTR deleteFromList(PTR head, int x)
{
   PTR current,previous;
   int found;
   current=head;
   if (current==NULL)
   {
      printf( "Empty list...nothing to delete.\n" );
      getchar();
   }
   else
      if (x==head->data)   /* Διαγραφή του πρώτου κόμβου   */
      {
         /* Ο δεύτερος κόμβος γίνεται πλέον πρώτος */
         head=head->next;
         free(current);
      }
      else
      {
         previous=current;
         current=head->next;
         found=0;
         while ((current!=NULL) && (found==0))
         {
            if (x==current->data)
               found=1;
            else
            {
               previous=current;
               current=current->next;
            }
         }
         if (found==1)
         {
            previous->next=current->next;
```

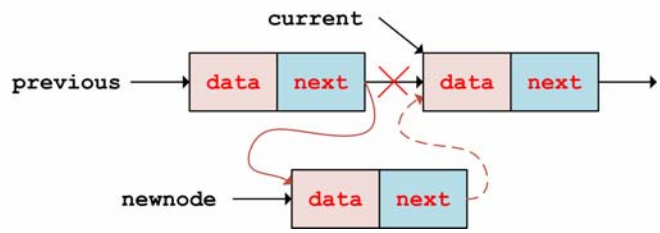
```
            free(current);
         }
         else
         {
            printf( "\nThe datum is not in the list." );
            getchar();
         }
      }
   return head;
}
```

(δ) **Εκτύπωση λίστας**

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

Παρατηρήσεις:

1. Από τους ανωτέρω κώδικες προκύπτει ότι σημαντικό ρόλο στην εισαγωγή και διαγραφή κόμβου διαδραματίζει η σωστή σύνδεση, ώστε να μη χαθεί η συνέχεια της λίστας. Προς αυτήν την κατεύθυνση χρησιμοποιείται το ζεύγος δεικτών σε κόμβο `current` και `previous`, οι οποίοι κρατούν τις διευθύνσεις δύο διαδοχικών κόμβων και επιτρέπουν τη «γεφύρωσή» τους με τον νέο κόμβο κατά τη λειτουργία της εισαγωγής:



Σχήμα 10.8 *Εισαγωγή κόμβου*

Στο **Σχήμα 10.8** απεικονίζεται η διαδικασία εισαγωγής κόμβου ανάμεσα σε δύο υφιστάμενους. Με την πρόταση

```
previous->next=newnode;
```

συνδέεται ο προηγούμενος κόμβος με τον νέο κόμβο, διακόπτοντας τη σύνδεση με τον επόμενο κόμβο της λίστας, τη διεύθυνση του οποίου κρατά ο δείκτης `previous`. Με την πρόταση

```
newnode->next=current;
```

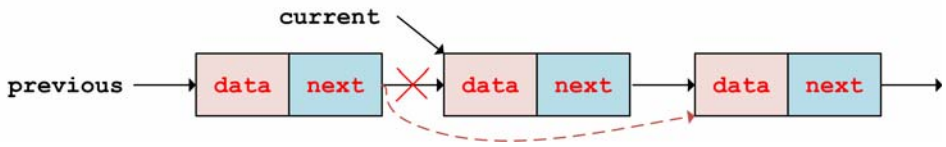
ο νέος κόμβος συνδέεται με τον επόμενο της λίστας, οπότε η σύνδεση διατηρείται.

Κατ' αντιστοιχία, όταν διαγράφεται ένας κόμβος, όπως απεικονίζεται στο **Σχήμα 10.9**, με την πρόταση

```
previous->next=current->next;
```

συνδέεται ο πρώτος με τον τρίτο κόμβο, ενώ ο δεύτερος δεν υπάγεται πλέον στη λίστα. Αυτός μέσω του δείκτη current διαγράφεται. Η μνήμη που καταλαμβάνει, απελευθερώνεται με την πρόταση

```
free(current);
```



Σχήμα 10.9 *Διαγραφή κόμβου*

2. Η διάσχιση της λίστας, π.χ. για την εκτύπωση των περιεχομένων της, γίνεται με χρήση του δείκτη σε κόμβο `current`, ο οποίος προωθείται από κόμβο σε κόμβο.

10.4.1.1 Παράδειγμα

Ο παρακάτω κώδικας δημιουργεί λίστα οκτώ κόμβων, η οποία θα τροφοδοτηθεί με τη σειρά δεδομένων **24, 31, 37, 1, 21, 25, 33, 30** (θεωρείται ότι η σειρά αριθμών βρίσκεται αποθηκευμένη σε προϋπάρχον αρχείο). Η συνάρτηση `PTR deleteEven(PTR head)` δέχεται τη διεύθυνση του πρώτου κόμβου της λίστας, διαγράφει τους κόμβους της λίστας που έχουν δεδομένα άρτιους αριθμούς και έχει επιστρεφόμενη τιμή τη διεύθυνση του πρώτου κόμβου της προκύψασας λίστας. Ακολούθως, εκτυπώνεται η νέα μορφή της λίστας.
Σημείωση: Ο αριθμός των κόμβων είναι δεδομένος (οκτώ), ωστόσο η συνάρτηση `PTR deleteEven()` είναι γενική, δηλαδή δεν θεωρούνται γνωστά ο αριθμός των υπαρχόντων κόμβων και τα περιεχόμενά τους.

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

#define N 8

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

PTR deleteEven(PTR head);
/*-----------------------------------*/
int main()
{
  FILE *fin;
  PTR current,newnode,head;
  int i;
  fin=fopen( "data.dat","r" );
  printf( "\n\n\nINITIAL LIST\n" );
  /* Δημιουργία λίστας 8 κόμβων με περιεχόμενα από το αρχείο  */
  for (i=0;i<N;i++)
  {
    newnode=malloc(sizeof(struct node));
    assert(newnode!=NULL);
    fscanf( fin,"%d",&newnode->data );
```

```
            newnode->next=NULL;
            if (i==0)  /* Πρώτος κόμβος  */
               head=newnode;
            else /* Υπόλοιποι κόμβοι  */
               current->next=newnode;
            current=newnode;
            printf( "\nnode %2d:\taddress=%d\tdata=%d",i+1,current,current-
>data );
         }
         fclose(fin);
         printf( "\n\nDelete nodes with even data:\n" );
         head=deleteEven(head);
         /* Εκτύπωση της λίστας μετά τις διαγραφές */
         printf("\n\n\nFINAL LIST\n");
         i=1;
         current=head;
         while (current!=NULL)
         {
            printf(  "\nnode  %2d:\taddress=%d\tdata=%d",i,current,current-
>data );
            i++;
            current=current->next;
         }

         return 0;
      }
      /*-----------------------------------*/
      PTR deleteEven(PTR head)
      {
         int i;
         PTR current,previous;
         i=1;
         current=head;
         while (current!=NULL)
         {
            if ((current->data%2)==0) /* Συνθήκη διαγραφής */
            {
               printf( "\n\t\tNode no %d, with data value %d, will be de-
leted",i,current->data );
               if (current==head)   /* Διαγραφή του πρώτου κόμβου  */
               {
                  head=head->next;
                  free(current);
                  current=head;
               }
               else /* Διαγραφή κόμβου σε θέση διαφορετική της πρώτης */
               {
                  previous->next=current->next;
                  free(current);
                  current=previous->next;
               }
               previous=current;
            }
            else /* Μετάβαση στον επόμενο κόμβο για έλεγχο */
            {
```

```
            previous=current;
            current=current->next;
        }
        i++;
    }
    return(head);
}
```

```
INITIAL LIST

node 1:        address=7542792  data=24
node 2:        address=7542688  data=31
node 3:        address=7554256  data=37
node 4:        address=7554272  data=1
node 5:        address=7554288  data=21
node 6:        address=7554304  data=25
node 7:        address=7554320  data=33
node 8:        address=7554336  data=30

Delete nodes with even data:

            Node no 1, with data value 24, will be deleted
            Node no 1, with data value 30, will be deleted


FINAL LIST

node 1:        address=7542688  data=31
node 2:        address=7554256  data=37
node 3:        address=7554272  data=1
node 4:        address=7554288  data=21
node 5:        address=7554304  data=25
node 6:        address=7554320  data=33
```

Εικόνα 10.3 *Η έξοδος του προγράμματος του παραδείγματος 10.4.1.1*

Από τα αποτελέσματα συνάγεται ότι οι πρώτοι δύο κόμβοι τοποθετήθηκαν σε θέσεις μνήμης που απέχουν τόσο μεταξύ τους όσο και από τους υπόλοιπους έξι κόμβους, επιβεβαιώνοντας τη λογική κατανομής χώρου αποθήκευσης των δυναμικών δομών δεδομένων.

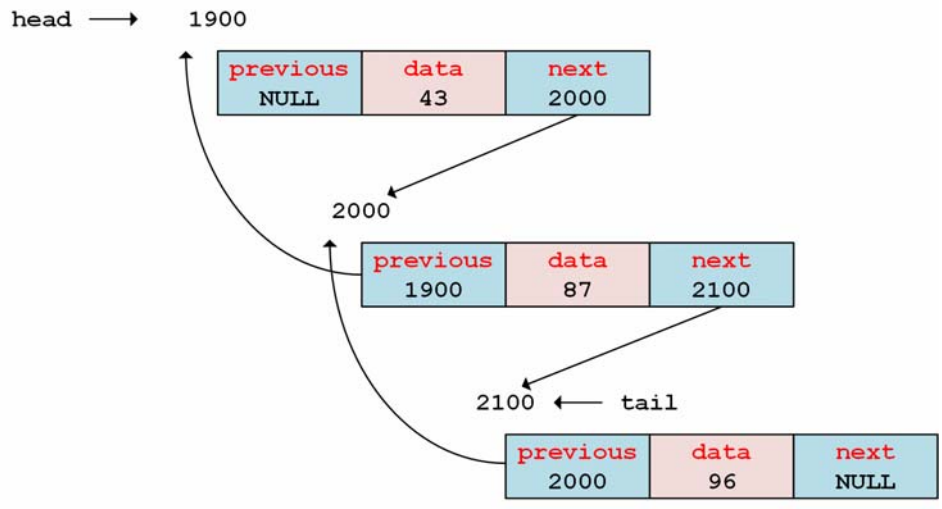
10.4.2 Διπλά συνδεδεμένη λίστα

Από την ανάλυση της απλά συνδεδεμένης λίστας διαπιστώνεται ότι, λόγω της χρήσης ενός μόνο δείκτη σε κάθε κόμβο, για να καταστεί προσβάσιμος ένας κόμβος της λίστας απαιτείται διάσχιση ολόκληρης της λίστας από την αρχή έως τον ζητούμενο κόμβο. Η διπλά συνδεδεμένη λίστα αποτελεί τροποποίηση της απλής, εισάγοντας ένα δεύτερο δείκτη σε κάθε κόμβο, ο οποίος δείχνει στον προηγούμενο κόμβο. Η διαχείριση της λίστας γίνεται πλέον με δύο δείκτες: τον **δείκτη κεφαλής** (head), ο οποίος δείχνει στον πρώτο κόμβο, και τον **δείκτη ουράς** (tail), ο οποίος δείχνει στον τελευταίο κόμβο. Ο κάθε κόμβος έχει ένα τμήμα δεδομένων και δύο τμήματα διεύθυνσης (**previous** και **next**). Ο κομβος υλοποιείται με την ακόλουθη δομή:

```
struct node
{
```

```
    <τύπος δεδομένου> <όνομα μεταβλητής>;
    struct node *next,*prev;
};
typedef struct node *PTR;
```

Ένα παράδειγμα διπλά συνδεδεμένης λίστας με τρεις κόμβους που περιέχουν ακεραίους, δίνεται στο **Σχήμα 10.10**:



Σχήμα 10.10 *Διπλά συνδεδεμένη λίστα τριών κόμβων*

Οι δύο βασικές λειτουργίες της εισαγωγής και της διαγραφής στοιχείου περιγράφονται ακολούθως:

(α) **Εισαγωγή στοιχείου**

Η εισαγωγή στοιχείου (τον κόμβο τον διαχειρίζεται ο **newnode**) στην αρχή της λίστας γίνεται ως εξής:

```
head->prev=newnode;
newnode->next=head;
head=newnode;
newnode->prev=NULL;
```

Αντίστοιχα γίνεται και η εισαγωγή κόμβου στο τέλος της λίστας:

```
tail->next=newnode;
newnode->prev=tail;
tail=newnode;
newnode->next=NULL;
```

Για την εισαγωγή κόμβου σε ενδιάμεση θέση θεωρείται ότι ο δείκτης **current** δείχνει στον κόμβο μετά τον οποίο θα γίνει η εισαγωγή, οπότε ο **current->next** δείχνει στον κόμβο που θα ακολουθεί τον εισαγόμενο:

```
newnode->next=current->next;
(current->next)->prev=newnode;
current->next=newnode;
newnode->prev=current->next;
```

(β) **Διαγραφή στοιχείου**

Η διαγραφή του κόμβου της κεφαλής γίνεται ως εξής:

```
current=head;          /* τοποθετείται σε δείκτη, ο οποίος
```

```
                                        θα επιτελέσει την απελευθέρωση μνήμης  */
head=head->next;
head->prev=NULL;
free(current);
```

Κατ' αντίστοιχο τρόπο διαγράφεται ο τελευταίος κόμβος της λίστας:

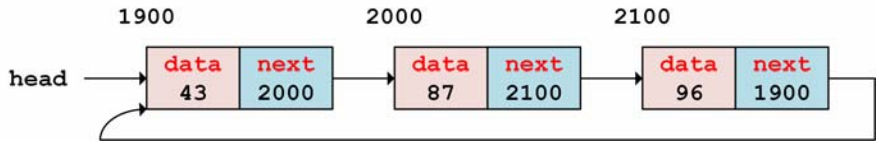
```
newnode=tail;
tail=tail->prev;
tail->next=NULL;
free(newnode);
```

Η διαγραφή κόμβου, ο οποίος βρίσκεται σε ενδιάμεση θέση, αποτελεί την Άσκηση 5 του κεφαλαίου.

Η διπλά συνδεδεμένη λίστα καθιστά ευκολότερη τη διαχείριση των κόμβων (εισαγωγή – διαγραφή), γεγονός ιδιαίτερα σημαντικό σε προβλήματα ταξινόμησης και αναζήτησης. Επίσης, τα περιεχόμενα των κόμβων προσπελαύνονται και από τις δύο κατευθύνσεις. Παρουσιάζει το μειονέκτημα της πολυπλοκότητας στη διαχείριση των κόμβων και της επιπρόσθετης απαιτούμενης μνήμης για κάθε κόμβο.

10.4.3 Κυκλική λίστα

Η κυκλική λίστα είναι μία απλά συνδεδεμένη λίστα, με τη διαφορά ότι το μέλος **next** του τελευταίου κόμβου δεν έχει τιμή **NULL** αλλά τη διεύθυνση του πρώτου κόμβου. Η δομή του κόμβου είναι ίδια με αυτή της απλά συνδεδεμένης λίστας. Ένα παράδειγμα κυκλικής λίστας με τρεις κόμβους που περιέχουν ακεραίους δίνεται στο **Σχήμα 10.11**:



Σχήμα 10.11 *Κυκλική λίστα τριών κόμβων*

Η διπλά συνδεδεμένη λίστα παρέχει ευελιξία λόγω του σχήματός της και βρίσκει εφαρμογή σε διαδικασίες, όπως ο διαμοιρασμός χρόνου από το λειτουργικό σύστημα. Ο διαμοιρασμός του χρόνου που οι εφαρμογές που τρέχουν σε έναν υπολογιστή και θα έχουν πρόσβαση στους πόρους του συστήματος, μπορεί να υλοποιηθεί με χρήση κυκλικής λίστας, καθώς δεν υπάρχουν δείκτες NULL. Έτσι, οι εφαρμογές θα διαμοιράζονται τον χρόνο με κυκλική εναλλαγή.

10.5 Εφαρμογές των συνδεδεμένων λιστών

Οι συνδεδεμένες λίστες εφαρμόζονται στην υλοποίηση των δομών δεδομένων της στοίβας και της ουράς, παρέχοντας τα πλεονεκτήματα της επεκτασιμότητας χωρίς πρακτικό περιορισμό υπερχείλισης και της εξοικονόμησης μνήμης, καθώς οι διαγραφές στοιχείων οδηγούν σε απελευθέρωση μνήμης.

10.5.1 Η στοίβα ως συνδεδεμένη λίστα

Η στοίβα υλοποιείται με έναν δείκτη **top**, ο οποίος αρχικοποιείται στο **NULL** και δείχνει ότι η στοίβα είναι άδεια. Ο δείκτης **top** διαδραματίζει τον ρόλο του δείκτη κεφαλής στην αρχή της λίστας. Όπως αναφέρθηκε ανωτέρω, η λειτουργία **push** δεν ελέγχει για υπερχείλιση, γιατί θεωρητικά η στοίβα μπορεί να έχει όσους κόμβους θέλουμε (αφού δημιουργείται δυναμικά). Επίσης, η λειτουργία **pop**, με τη βοήθεια της συνάρτησης `free()`, απελευθερώνει τον χώρο μνήμης που καταλαμβανόταν από τον κόμβο που διαγράφηκε.

Η εισαγωγή στοιχείου στην κορυφή της στοίβας υλοποιείται με την εισαγωγή νέου κόμβου στην αρχή της συνδεδεμένης λίστας. Αντίστοιχα, η διαγραφή στοιχείου από τη στοίβα υλοποιείται με τη διαγραφή του πρώτου κόμβου της συνδεδεμένης λίστας. Στο **Σχήμα 10.7** απεικονίζεται μία στοίβα ακέραιων αριθμών **3** θέσεων, με στοιχείο κορυφής το **43**.

(α) **Εισαγωγή στοιχείου στην κορυφή της στοίβας**

Η εισαγωγή στοιχείου (τον κόμβο τον διαχειρίζεται ο **newnode**) στην κορυφή της στοίβας γίνεται ως εξής:

```
PTR push(PTR top, int x)
{
  PTR newnode;
  int found;
  /* Δημιουργία νέου κόμβου */
  newnode=malloc(sizeof(struct node));
  newnode->data=x;
  /* Εάν η στοίβα είναι άδεια, ο νέος κόμβος είναι η κορυφή της.*/
  if (top==NULL)
  {
    top=newnode;
    newnode->next=NULL;
  }
  /* Εάν η στοίβα δεν είναι άδεια, τοποθετείται ο νέος κόμβος στην
κεφαλή */
  else
  {
    newnode->next=top;
    top=newnode;
  }
  return top;
}
```

(β) **Διαγραφή στοιχείου από τη στοίβα**

Ο κώδικας της διαγραφής στοιχείου από τη στοίβα είναι ο ακόλουθος:

```
PTR pop(PTR top, int *obj)
{
  PTR current;
  if (top==NULL)
  {
    printf( "Empty stack...nothing to delete.\n" );
    getchar();
  }
  else
  {
    current=top;
    top=top->next;
    *obj=current->data;
    free(current);
    return top;
  }
}
```

10.5.2 Η ουρά ως συνδεδεμένη λίστα

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

(α) **Εισαγωγή στοιχείου στο τέλος της ουράς**

```
void enqueue(int obj, PTR *pf, PTR *pr)
{
  PTR newnode;
  newnode=malloc(sizeof(struct node));
  newnode->data=obj;
  newnode->next=NULL;
  /* Εάν η ουρά είναι άδεια, ο νέος κόμβος είναι η αρχή και το τέλος
     της. */
  if ((*pf)==NULL)
  {
     *pf=newnode;
     *pr=newnode;
  }
  /* Εάν η ουρά δεν είναι άδεια, τοποθετείται ο νέος κόμβος στο
     τέλος*/
  else
  {
     (*pr)->next=newnode;
     *pr=newnode;
  }
}
```

(α) **Διαγραφή στοιχείου από την αρχή της ουράς**

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

10.6 Παράδειγμα ανάπτυξης προγράμματος

Θα υλοποιηθεί μία εφαρμογή διαχείρισης πελατών τράπεζας, οι οποίοι περιμένουν στην ουρά, για να εξυπηρετηθούν από το ταμείο. Η τράπεζα διατηρεί ένα αρχείο με εγγραφές πελατών με τις ακόλουθες πληροφορίες:

- Αριθμός λογαριασμού.
- Ονοματεπώνυμο πελάτη.

- Ποσό λογαριασμού.

Το πρόγραμμα είναι καθοδηγούμενο από ένα μενού, με δύο βασικές επιλογές:

1. Άφιξη πελάτη στην τράπεζα.

2. Αναχώρηση πελάτη (εφόσον έχει εξυπηρετηθεί).

Κάθε φορά που προσέρχεται ένας πελάτης (επιλογή **1**), τοποθετείται στο τέλος μίας ουράς με τη λειτουργία **enqueue** (στην ουσία τοποθετείται ο αριθμός λογαριασμού του πελάτη, αν και θα μπορούσε να είναι κάποιος αριθμός προτεραιότητας).

Όταν ο χρήστης επιλέξει τη λειτουργία **2**, τότε αυτομάτως ο πελάτης που βρίσκεται στην αρχή της ουράς, προωθείται για να εξυπηρετηθεί, αφού πρώτα γίνει εξαγωγή του αριθμού λογαριασμού από την ουρά με τη λειτουργία **dequeue**. Βάσει του καταχωρημένου στην ουρά αριθμού λογαριασμού διεξάγεται αναζήτηση στο αρχείο των πελατών. Εάν βρεθεί η εγγραφή τότε, αφού εμφανιστούν τα περιεχόμενα της εγγραφής στην οθόνη, ο πελάτης ζητείται να επιλέξει μία εκ των δύο συναλλαγών:

1. Ανάληψη.

2. Κατάθεση.

Μετά την καταχώρηση του είδους της συναλλαγής ζητείται το ποσό της συναλλαγής, το οποίο και προστίθεται ή αφαιρείται από το τρέχον ποσό του λογαριασμού. Στο τέλος, η ενημερωμένη εγγραφή καταχωρείται πίσω στο αρχείο.

Εάν η αναζήτηση στο αρχείο είναι ανεπιτυχής, τότε ο αριθμός λογαριασμού αγνοείται και η ροή του προγράμματος επανέρχεται στο αρχικό μενού.

Τέλος, εάν ο χρήστης επιλέξει την έξοδο από το μενού, τότε το πρόγραμμα τερματίζει, αφού πρώτα εμφανίσει το πλήθος των πελατών που περιμένουν στην ουρά, για να εξυπηρετηθούν.

Με βάση την ανωτέρω περιγραφή της λειτουργίας της εφαρμογής, αρχικά ορίζεται το δεδομένο της εγγραφής πελάτη ως μία δομή:

```
struct accountT
{
   int numb;
   char name[15];
   float amount;
};
typedef struct account bankAccount;
```

Η αποθήκευση των εγγραφών των πελατών θα γίνει με χρήση ουράς. Πέραν της συνάρτησης **main()**, οι λειτουργίες του προγράμματος θα μεριστούν στις ακόλουθες συναρτήσεις:

- **`enqueue()`**: Εισάγει στοιχείο στο τέλος της ουράς.
- **`dequeue()`**: Εξάγει στοιχείο από την αρχή της ουράς.
- **`searchFile()`**: Αναζητά μία εγγραφή στο αρχείο βάσει του αριθμού λογαριασμού. Σε περίπτωση επιτυχίας, διαβάζει τα περιεχόμενα της εγγραφής.
- **`getSelection()`**: Επιλέγει το είδος της συναλλαγής (ανάληψη ή κατάθεση).
- **`updateAmount()`**: Ανάλογα με την επιλογή της συναλλαγής, ενημερώνει το ποσό που διαθέτει ο λογαριασμός.
- **`displayFile()`**: Εμφανίζει στην οθόνη τα περιεχόμενα του αρχείου.
- **`fillBlanks()`**: Διευθετεί την εμφάνιση των αποτελεσμάτων στην οθόνη, προσθέτοντας τα απαραίτητα κενά.

Το τελικό πρόγραμμα έχει την ακόλουθη μορφή:

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

```
#define N 100 /*Μέγεθος του πίνακα για την αποθήκευση της ουράς*/

struct accountT{
  int numb;
  char name[20];
  float amount;
};
typedef struct accountT bankAccount;
/*-----------------------------------*/
int getSelection(bankAccount account);
void updateAmount(bankAccount *p, int sel);
int searchFile(FILE *fp, int x, bankAccount *p);
void fillBlanks(char *s, int x);
void displayFile(FILE *fp);
void enqueue(int q[N], int *r, int obj);
int dequeue(int q[N], int *f, int rear);
/*-----------------------------------*/
int main()
{
   int i,choice,found,custCode,sel,reclen,ncust=0,endFlag=0;
   int q[N],front=-1,rear=-1;
   FILE *fp;
   bankAccount account;
   fp=fopen("data.dat","rb+");   assert(fp!=NULL);
   displayFile(fp);
   rewind(fp);
   do
   {
     printf( "\n\n  Menu\n" );
     printf( "1.Customer arrival\n" );
     printf( "2.Customer departure\n" );
     printf( "3.Exit\n" );
     printf( "\nChoice: " );
     scanf( "%d",&choice );
     switch (choice)
     {
        case 1:
           printf( "\nGive the account number:" );
           scanf( "%d",&custCode );
           enqueue(q,&rear,custCode);
       break;
       case 2:
           custCode=dequeue(q,&front,rear);
           if (custCode!=0)  {
              found=0;
              found=searchFile(fp,custCode,&account);
              if (found==1)   {
                 sel=getSelection(account);
                 updateAmount(&account,sel);
                 reclen=sizeof(bankAccount);
                 fseek(fp,-reclen,1);
                 fwrite(&account,sizeof(bankAccount),1,fp);
              }
              else   {
                printf( "\nERROR! Invalid account number!\n");
```

```
                getchar();
              }
              rewind(fp);
           }
           break;
           case 3:
              endFlag=1;
             break;
      }
    } while (endFlag==0);
    if (front==rear) printf( "No customers waiting in queue..." );
    else
    {
      for (i=front+1;i<=rear;i++)    ncust++;
      printf( "Number of customers waiting in queue: %d ",ncust );
    }
    getchar();
    fclose(fp);
    return 0;
}
/*-------------------------------------*/
void enqueue(int q[N], int *r, int obj)
{
    if ((*r)==N-1)
    {
      printf( "Queue is full..." );
      getchar();
    }
    else
      q[++(*r)]=obj;  }
/*-------------------------------------*/
int dequeue(int q[N], int *f, int rear)
{
    int temp;
    if ((*f)==rear)
    {
      printf( "Empty Queue...\n" );
      temp=0;
    }
    else
    {
      temp=q[++(*f)];
      printf( "Customer %d is served...",temp );
    }
    getchar();
    return temp;
}
/*-------------------------------------*/
int getSelection(bankAccount account)
{
    int sel;
    printf( "\nAccount no.: %i\n", account.numb );
    printf( "Name       : %s\n", account.name );
    printf( "Amount    : %6.2f\n", account.amount );
    printf( "\n1. Deposit\n" );
```

```
   printf( "2. Withdraw\n" );
   printf( "\n\nSelect transaction:" );
   do
   {
     scanf( "%d",&sel );
   } while ((sel<1) || (sel>2));
   return sel;
}
/*-------------------------------------*/
void updateAmount(bankAccount *p, int sel)
{
  float amt;
  printf( "Give the amount:" );
  scanf( "%f",&amt );
  if (sel==1)  p->amount+=amt;
  else
     if (amt<=p->amount)  p->amount-=amt;
     else
     {
       printf( "The amount is too large to withdraw!" );
       getchar();
     }
}
/*-------------------------------------*/
int searchFile(FILE *fp, int x, bankAccount *p)
{
   bankAccount account;
   int found=0;
   fread(&account,sizeof(bankAccount),1,fp);
   while ((!feof(fp)) && (found==0))
   {
     if (x==account.numb)    found=1;
     else    fread(&account,sizeof(bankAccount),1,fp);
   }
   *p=account;
   return found;
}
/*-------------------------------------*/
void fillBlanks(char *s, int x)
{
  int i;
  for (i=1;i<=(x-strlen(s));i++)
     strcat(s," ");
}
/*-------------------------------------*/
void displayFile(FILE *fp)
{
  bankAccount account;
  char temp[20]="NAME";
  fillBlanks(temp,20);
  printf( "\nThe contents of the file are:\n\n" );
  printf( "ACC.NO.     %s     AMOUNT\n",temp );
  fread(&account,sizeof(bankAccount),1,fp);
  while (!feof(fp))
  {
```

```
            fillBlanks(account.name,20);
            printf("%4d      %s      %7.2f\n",account.numb, account.name, ac-
count.amount);
            fread(&account,sizeof(bankAccount),1,fp);
          }
      }
```

```
The contents of the file are:

ACC.No.    NAME          AMOUNT
  101      Johnson         310.00
  102      Jameson        1310.00
  103      Taylor            0.00

  Menu
1.Customer arrival
2.Customer departure
3.Exit

Choice: 1

Give the account number:102

  Menu
1.Customer arrival
2.Customer departure
3.Exit

Choice: 2
Customer 102 is served...
Account no.      : 102
Name             : Jameson
Amount           : 1310.00

1.Deposit
2.Withdraw

Select transaction:1
Give the amount:150

  Menu
1.Customer arrival
2.Customer departure
3.Exit

Choice: 2
Empty Queue...
```

Εικόνα 10.4 *Αποτελέσματα του προγράμματος*

Στην **Εικόνα 10.4** παρουσιάζεται ένα στιγμιότυπο από την εκτέλεση του προγράμματος. Αρχικά, εμφανίζονται στην οθόνη οι εγγραφές που βρίσκονται στο αρχείο. Η πρώτη επιλογή είναι η άφιξη του πελάτη

με λογαριασμό **102** και η εισαγωγή του στην ουρά. Η επόμενη επιλογή είναι η αναχώρηση πελάτη, οπότε ο πελάτης **102** προχωρά στο ταμείο για συναλλαγή. Επιλέγεται η κατάθεση **150** ευρώ, οπότε εμφανίζονται τα στοιχεία του λογαριασμού του πελάτη (το ποσό είναι αυτό που υπάρχει πριν τη συναλλαγή). Επιλέγοντας εκ νέου αναχώρηση πελάτη, ενώ η ουρά έχει αδειάσει, εμφανίζεται σχετικό μήνυμα.

Ερωτήσεις αυτοαξιολόγησης - ασκήσεις

Ερωτήσεις αυτοαξιολόγησης

Στις παρακάτω ερωτήσεις επιλέξτε μία από τις τέσσερις απαντήσεις.

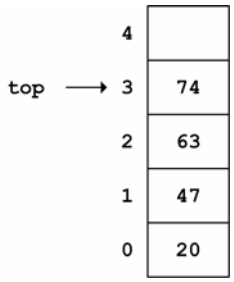
(1) Η λειτουργία pop(απώθηση) μίας στοίβας:
(α) ελέγχει αν η στοίβα είναι γεμάτη και τοποθετεί ένα νέο στοιχείο στην κορυφή της στοίβας.
(β) ελέγχει αν η στοίβα είναι άδεια και τοποθετεί ένα νέο στοιχείο στην κορυφή της στοίβας.
(γ) ελέγχει αν η στοίβα είναι γεμάτη και εξάγει το στοιχείο που βρίσκεται στην κορυφή της στοίβας.
(δ) ελέγχει αν η στοίβα είναι άδεια και εξάγει το στοιχείο που βρίσκεται στην κορυφή της στοίβας.

(2) Έστω το παρακάτω τμήμα προγράμματος, που δημιουργεί μία συνδεδεμένη λίστα.

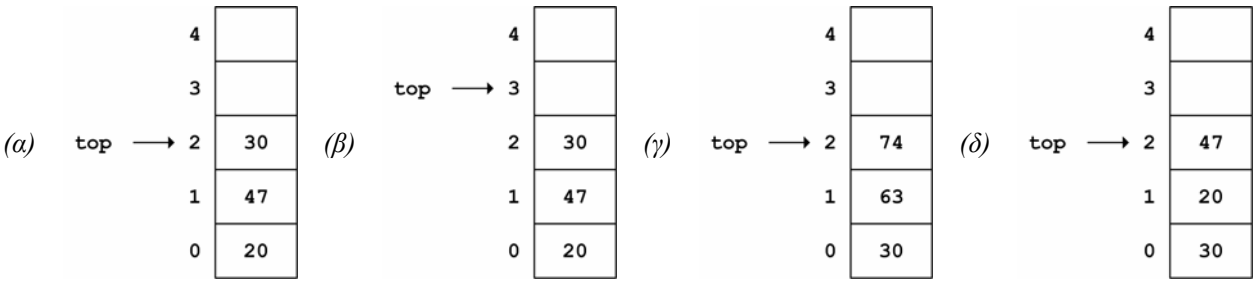
```
typedef struct node
{
   float data;
   struct node *next;
} *ptr;

ptr createList(ptr  p, float pin[]);

int main()
{
   float a[5]={9.1,7.2,3.8,3.1,6.9};
   ptr p=NULL;
   p=createList(p,a);
}

ptr createList(ptr  p, float pin[])
{
   int i;
   ptr current;
   for (i=0; i<5; i++)
   {
      current=malloc(sizeof(struct node));
      current->data=pin[i];
      current->next=p;
      p=current;
   }
   return p;
}
```

Με ποια σειρά αποθηκεύονται οι τιμές στους κόμβους, ξεκινώντας απο την αρχή της λίστας;
(α) 9.1, 7.2, 3.8, 3.1, 6.9
(β) 6.9, 3.1, 3.8, 7.2, 9.1
(γ) 3.1, 3.8, 6.9, 7.2, 9.1
(δ) 9.1, 7.2, 6.9, 3.8, 3.1

(3) Έστω η παρακάτω στοίβα:



Εκτελείται δύο φορές η λειτουργία της απώθησης (pop) και μία φορά η λειτουργία της ώθησης (push) του αριθμού **30**. Ποια είναι η νέα μορφή της στοίβας;



(4) Χρησιμοποιώντας τη δομή μίας στοίβας, υπολογίζεται η τιμή της παράστασης:

4 5 7 + 2 * +

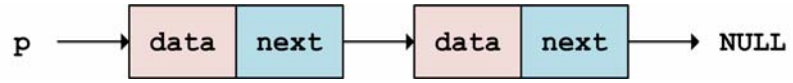
εφαρμόζοντας τους εξής κανόνες:

1. Αν το στοιχείο είναι τελεσταίος, τότε τοποθετείται σε μία στοίβα (push).
2. Αν το στοιχείο είναι τελεστής, τότε:
   - *i.* Εξάγονται διαδοχικά δύο τελεσταίοι από την κορυφή της στοίβας (pop).
   - *ii.* Εκτελείται η πράξη που δηλώνει ο τελεστής.
   - *iii.* Το αποτέλεσμα τοποθετείται στη στοίβα (push).

Ποια θα είναι η τιμή της παράστασης;

(α) 36
(β) 28
(γ) 24
(δ) 52

(5) Έστω η ακόλουθη συνδεδεμένη λίστα:



Θέλουμε να εισαχθεί ο παρακάτω νέος κόμβος ανάμεσα στους δύο κόμβους της λίστας.

Ποιος είναι ο κώδικας παρεμβολής του νέου κόμβου;

(α)
```
p=r;
r=p->next;
```
(α)
```
p->next=r;
r->next=p->next;
```
(α)
```
p->next=r->data;
r->next=p->next;
```
(α)
```
r->next=p->next;
p->next=r;
```

Ασκήσεις

Άσκηση 1

Να υλοποιηθεί η εφαρμογή της ενότητας 10.6 με χρήση δυναμικής ουράς.

Άσκηση 2

Να γραφεί κώδικας για τη δημιουργία λίστας έξι κόμβων, η οποία θα περιέχει ως δεδομένα τις δυνάμεις του 3 από το 2 έως το 7. Ακολούθως, να συγγραφεί αναδρομική συνάρτηση `void printListBackwards(PTR head)`, η οποία θα δέχεται τη διεύθυνση του πρώτου κόμβου της λίστας και θα εκτυπώνει τα περιεχόμενα της λίστας ανάποδα (από τον τελευταίο κόμβο προς τον πρώτο).

Άσκηση 3

Έστω μία υπάρχουσα και μη κενή συνδεδεμένη λίστα, η οποία περιέχει ως δεδομένα ακέραιους αριθμούς. Να γραφεί πρόγραμμα, το οποίο θα αναζητά τους κόμβους εκείνους που έχουν ως δεδομένο περιττό ακέραιο και όποτε βρει τέτοιον κόμβο, θα τον μεταφέρει από τη θέση, στην οποία βρίσκεται στην αρχή της λίστας (δηλαδή ο κόμβος αυτός θα γίνει σε εκείνο το στάδιο εκτέλεσης του προγράμματος ο πρώτος κόμβος της λίστας).

Άσκηση 4

Έστω μία υπάρχουσα και μη κενή συνδεδεμένη λίστα, η οποία περιέχει ως δεδομένα ακέραιους αριθμούς. Να γραφεί πρόγραμμα, το οποίο θα αναζητά τον κόμβο που έχει ως δεδομένο έναν αριθμό που διαβάζεται από το πληκτρολόγιο. Εάν βρεθεί τέτοιος κόμβος, θα διαγράφονται όλοι οι κόμβοι της λίστας από την αρχή της έως και αυτόν τον κόμβο.

Άσκηση 5

Να γραφεί πρόγραμμα για τη διαγραφή κόμβου από διπλά συνδεδεμένη λίστα, ο οποίος δεν βρίσκεται στην αρχή ή το τέλος της λίστας.

Άσκηση 6

Να γραφεί πρόγραμμα, το οποίο θα χρησιμοποιεί διπλά συνδεδεμένη λίστα για την αφαίρεση των σημείων στίξης από τα περιεχόμενα ενός υφιστάμενου αρχείου κειμένου.

Άσκηση 7

Να γραφεί πρόγραμμα, το οποίο θα διαχωρίζει μία υφιστάμενη απλά συνδεδεμένη λίστα με δεδομένα ακέραιους αριθμούς, σε δύο άλλες. Οι κόμβοι της αρχικής λίστας με δεδομένα θετικούς αριθμούς θα εντάσσονται στη μία εκ των δύο νέων λιστών, ενώ οι κόμβοι με στοιχεία αρνητικούς αριθμούς και το μηδέν θα εντάσσονται στην έτερη νέα λίστα.

Βιβλιογραφία κεφαλαίου

Καρολίδης, Δ. (2013), *Μαθαίνετε Εύκολα C*, αυτοέκδοση.

Κοίλιας, Χ. (2004), *Δομές Δεδομένων και Οργανώσεις Αρχείων*, Εκδόσεις Νέων Τεχνολογιών.

Μποζάνης, Π. (2006), *Δομές Δεδομένων*, Εκδόσεις Τζιόλα.

Παπουτσής, Ι. (2010), *Εισαγωγή στις Δομές Δεδομένων και στους Αλγόριθμους – Υλοποίηση σε C*, Εκδόσεις Σταμούλης.

Χατζηγιαννάκης, Ν. (2012), *Η Γλώσσα C σε Βάθος*, 4η Έκδοση, Εκδόσεις Κλειδάριθμος.

Kalicharan, N. (2013), *Advanced Topics in C – Core Topics in Data Structures*, Apress.

Sahni, S. (2004), *Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++*, Εκδόσεις Τζιόλα.

Sedgewick, R. (2005), *Αλγόριθμοι σε C*, Εκδόσεις Κλειδάριθμος.

Wirth, N. (2004), *Αλγόριθμοι και Δομές Δεδομένων*, Εκδόσεις Κλειδάριθμος.