Κεφάλαιο 6
ΓΡΑΜΜΙΚΕΣ ΛΙΣΤΕΣ

6.1 ΕΙΣΑΓΩΓΗ

Γραμμική λίστα (***linear list***) είναι ένα πεπερασμένο σύνολο από κόμβους x_1, x_2,....., x_n όπου το στοιχείο x_k προηγείται του στοιχείου x_{k+1} και έπεται του x_{k-1}.

Κατατάσσονται συνήθως σε δύο κατηγορίες:

- Σειριακές γραμμικές λίστες (*sequential linear lists*)
- Συνδεδεμένες γραμμικές λίστες (*linked linear lists*)

Στην πρώτη κατηγορία καταλαμβάνονται συνεχόμενες θέσεις μνήμης του Η/Υ για την αποθήκευση των κόμβων.

Στην δεύτερη κατηγορία οι κόμβοι των λιστών βρίσκονται σε απομακρυσμένες θέσεις που είναι μεταξύ τους συνδεδεμένες.

Επίσης, οι γραμμικές λίστες χαρακτηρίζονται σαν:

- Στατικές δομές δεδομένων (*static data structures*)
- Δυναμικές δομές δεδομένων (*dynamic data structures*)

Στην πρώτη κατηγορία, κατά τον προγραμματισμό των λειτουργιών των λιστών, έχει προκαθορισθεί το μέγεθος της μνήμης που απαιτείται για την αποθήκευση των λιστών.

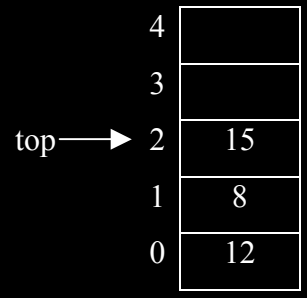
Στην δεύτερη κατηγορία, μία λίστα μπορεί να αυξομειωθεί κατά την διάρκεια εκτέλεσης του προγράμματος.

6.2 ΣΕΙΡΙΑΚΕΣ ΛΙΣΤΕΣ

6.2.1 *Στοίβα (stack)*

Μπορούμε να την παραλληλίσουμε σαν μία στοίβα από πιάτα. Κάθε νέο στοιχείο τοποθετείται στην κορυφή (top). Το στοιχείο που βρίσκεται στην κορυφή της στοίβας εξέρχεται πρώτο. Αυτή η μέθοδος επεξεργασίας ονομάζεται

LIFO (**L**ast **I**n **F**irst **O**ut)



Μία στατική στοίβα υλοποιείται με τη χρήση ενός μονοδιάστατου πίνακα και ενός δείκτη. 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*).

Υλοποίηση στοίβας σε C:

```
#define  N  100
int stack[N], top = -1;

void push(int stack[],int *t,int obj)
{
   if (*t = = N-1)
     {
         printf("Stack overflow...\n");
         getch();
         abort();
     }
  else
         stack[++(*t)] = obj;
}

int  pop(int  stack[],int *t)
{
  int  r ;

  if (*t < 0)
     {
        printf("Stack empty...\n");
        getch();
        abort();
     }
  else
        r = stack[(*t)--];
  return  r;
}
```

6.2.2 Ουρά (queue)

Την έννοια της ουράς την 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 καταφθάνει, τοποθετείται στο τέλος της ουράς. Η μέθοδος αυτή επεξεργασίας ονομάζεται

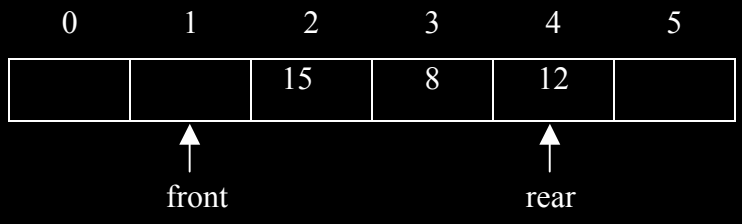
FIFO (First In First Out)

Δύο βασικές λειτουργίες:

- **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).

Επειδή βολεύει – προγραμματιστικά – 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.



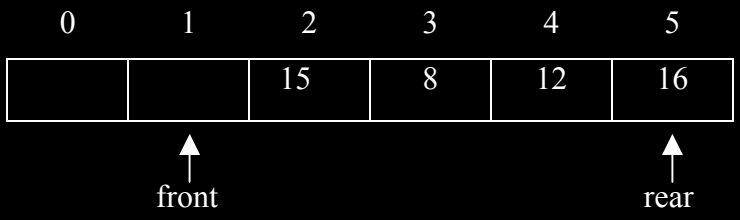
<u>Υλοποίηση ουράς σε C:</u>

```
#define N 100

int  q[N], front = -1, rear = -1;

void enqueue(int q[], int *r, int obj)
{
   if (*r = = N-1)
     {
        printf("Queue is full...");
        getch();
     }
   else
       q[++(*r)] = obj;
 }
```

```
void  dequeue(int q[], int *f, int  r)
{
  int  x;

  if (*f = = r)
       printf("Queue is empty...\n");
  else
      {
         x = q[++(*f)];
         printf("%d has been deleted...",x);
      }
}
```

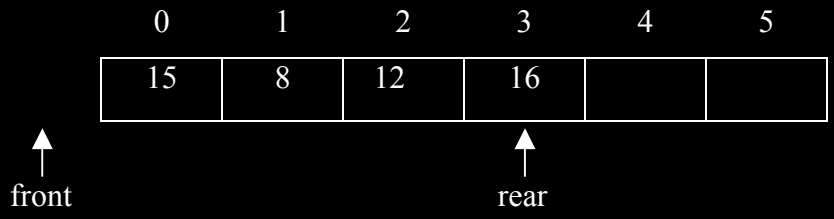
Η υλοποίηση της ουράς με πίνακα έχει ένα μειονέκτημα. Υπάρχει περίπτωση ο rear να φθάσει στο πάνω όριο του πίνακα, αλλά στην ουσία να μην υπάρχει υπερχείλιση, επειδή ο front θα έχει αυξηθεί (*εικονική υπερχείλιση*).

π.χ.



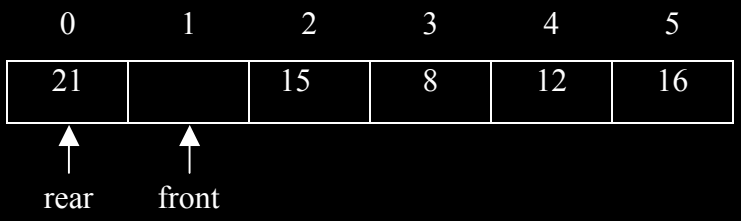
Στην πραγματικότητα υπάρχει ελεύθερος χώρος για την εισαγωγή και νέων στοιχείων. Μία λύση θα ήταν να μεταφερθούν τα στοιχεία στο αριστερό άκρο του πίνακα.

π.χ.



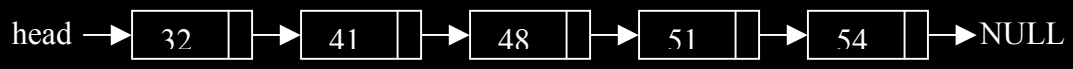

```
elements = front – rear;
first = front + 1;
for (i=0; i<elements; i++)
      q[i] = q[first++];
front = -1;
rear = elements – 1;
```

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)*.



6.3 ΣΥΝΔΕΔΕΜΕΝΕΣ ΛΙΣΤΕΣ

Οι στατικές δομές που μελετήθηκαν, παρουσιάζουν προβλήματα στην εισαγωγή και διαγραφή κόμβων και στην αποδοτική εκμετάλλευση της διαθέσιμης μνήμης. Κύριο χαρακτηριστικό των συνδεδεμένων λιστών είναι ότι οι κόμβοι τους βρίσκονται σε απομακρυσμένες θέσεις μνήμης και η σύνδεσή τους γίνεται με δείκτες. Κατ' αυτό τον τρόπο, η εισαγωγή και διαγραφή κόμβων γίνεται πολύ πιο απλά. Ένα άλλο θετικό χαρακτηριστικό είναι ότι, δεν απαιτείται εκ των προτέρων καθορισμός του μέγιστου αριθμού κόμβων της λίστας και μπορεί η λίστα να επεκταθεί ή να συρρικνωθεί κατά την εκτέλεση του προγράμματος (*δυναμικές δομές*).



Κάθε κόμβος της λίστας υλοποιείται με μία δομή (structure) με δύο στοιχεία. Το ένα μέλος (data) περιέχει τα δεδομένα (οποιουδήποτε τύπου) του κόμβου και το άλλο μέλος (next) είναι δείκτης προς τον επόμενο κόμβο. Τοποθετείται ένας δείκτης (head) στον πρώτο κόμβο για να προσπελάζεται η λίστα, ενώ ο δείκτης του τελευταίου κόμβου δείχνει στο NULL για να εντοπίζεται το τέλος της λίστας.

6.3.1 Απλή Συνδεδεμένη Λίστα

Παράδειγμα υλοποίησης λίστας ακεραίων στην C:

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

α) Δημιουργία λίστας

```
PTR  list_create(PTR  head)
{
  PTR   current;
  int  x;

  printf("Give an integer, 0 to stop:");
  scanf("%i",&x);
  if (x = = 0)
     return  NULL;
  else
    {
      head = malloc(sizeof(struct node));
      head->data = x;
      current = head;
      printf("Give an integer, 0 to stop:");
      scanf("%i",&x);
      while (x!=0)
            {
               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;
    }
  return head;
}
```

β) Εισαγωγή στοιχείου στη σωστή θέση
(*υποτίθεται η λίστα είναι ταξινομημένη*)

```
PTR insert_to_list(PTR head, int  x)
{
  PTR current, previous, newnode;
  int found;

  newnode = malloc(sizeof(struct node));
  newnode->data = x;
  newnode->next = NULL;
  if (head = = NULL)
      head = newnode;
  else
     if (newnode->data < head->data)
       {
         newnode->next = head;
         head = newnode;
       }
```

```
    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 από λίστα

```
PTR  delete_from_list(PTR head, int  x)
{
  PTR current, previous;
  int found;

  current = head;
  if (current = = NULL)
    {
      printf("Empty list...nothing to delete.\n");
      getch();
    }
  else
      if (x = = head->data)
      {
          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
```

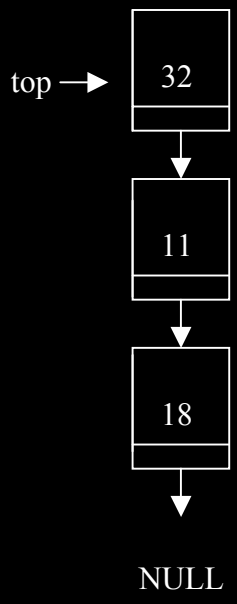
```
                {
                  previous = current;
                  current = current->next;
                }
             }
          if  (found = = 1)
             {
                previous->next = current->next;
                free(current);
             }
          else
             {
                printf("\nThe character is not in the list.");
                getch();
             }
       }
  return head;
}
```

δ) Εκτύπωση λίστας

```
void  print_list(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;
           }
}
```

6.3.3 Στοίβα ως Συνδεδεμένη Λίστα

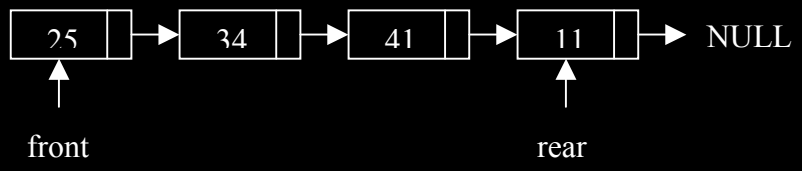
Η στοίβα υλοποιείται με έναν δείκτη top που αρχικοποιείται στο NULL, και δείχνει ότι η στοίβα είναι άδεια. Η λειτουργία push δεν ελέγχει για υπερχείλιση γιατί θεωρητικά η στοίβα μπορεί να έχει όσους κόμβους θέλουμε (αφού δημιουργείται δυναμικά). Επίσης, η λειτουργία pop, με τη βοήθεια της εντολής free επιστρέφει στη διαθέσιμη μνήμη του Η/Υ το χώρο που καταλαμβάνονταν από τον κόμβο που διαγράφηκε.




```
PTR top = NULL;

PTR push(int obj, PTR  t)
{
  PTR newnode;

  newnode = malloc(sizeof(struct node));
  newnode->data = obj;
  newnode->next = t;
  t = newnode;
  return t;
}

PTR pop(PTR t, int *obj)
{
  PTR p;

  if (t = = NULL)
    {
      printf("Stack empty.\n");
      getch();
    }
  else
    {
      p = t;
      t = t->next;
     *obj = p->data;
      free(p);
    }
   return t;
}
```

6.3.3 Ουρά ως Συνδεδεμένη Λίστα

Η ουρά υλοποιείται με τη χρήση δύο δεικτών front και rear που 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.




```
PTR front = NULL, rear = NULL;
void enqueue(int obj, PTR *pf, PTR *pr)
{
  PTR newnode;
  newnode = malloc(sizeof(struct node));
  newnode->data = obj;
  newnode->next = NULL;
  if ((*pf) = = NULL)
    {
     *pf = newnode;
     *pr = newnode;
    }
  else
    {
     (*pr)->next = newnode;
      *pr = newnode;
    }
}

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