3.2 Υλοποίηση ΑΤΔ Ουρά με πίνακα

Επειδή οι ουρές μοιάζουν πολύ με τις στοίβες, μπορούμε να υλοποιήσουμε μια ουρά χρησιμοποιώντας πάλι πίνακες. Για την ουρά, λοιπόν, χρειαζόμαστε έναν πίνακα, Queue, για την αποθήκευση των στοιχείων της ουράς, και δύο μεταβλητές:

- την μεταβλητή Front, όπου θα αποθηκεύεται η θέση του πίνακα στην οποία βρίσκεται το στοιχείο που μπορεί να διαγραφεί, δηλαδή το πρώτο στοιχείο της ουράς, και
- την Rear, όπου θα αποθηκεύεται η θέση του πίνακα στην οποία μπορεί να εισαχθεί ένα νέο στοιχείο, δηλαδή η θέση μετά το τελευταίο στοιχείο της ουράς.

Ένα στοιχείο, λοιπόν, μπορεί να διαγραφεί από την ουρά ανακτώντας το στοιχείο που βρίσκεται στην θέση Front του πίνακα Queue, δηλαδή Queue[Front], και αυξάνοντας την μεταβλητή Front κατά 1. Ένα στοιχείο εισάγεται στην ουρά με αποθήκευσή του στην θέση Rear του πίνακα Queue, δηλαδή Queue[Rear], με την προϋπόθεση, βέβαια ότι η μεταβλητή Rear είναι μικρότερη του ορίου του πίνακα, και αυξάνοντας την μεταβλητή Rear κατά 1.

Η δυσκολία εδώ είναι ότι τα στοιχεία μετατοπίζονται προς τα δεξιά μέσα στον πίνακα, πράγμα το οποίο σημαίνει ότι ίσως να χρειαστεί όλα τα στοιχεία του πίνακα να μετατοπιστούν πίσω στις αρχικές θέσεις. Για να γίνει κατανοητή η λειτουργία της ουράς, ας θεωρήσουμε μια ουρά 5 θέσεων στην οποία εισάγονται με τη σειρά οι αριθμοί 30, 17 και 25, εν συνεχεία διαγράφονται οι 30 και 17 και, τέλος, εισάγονται οι 7 και 53, όπως δείχνουν τα ακόλουθα σχήματα.

Queue

θέση	**1**	**2**	**3**	**4**	**5**
αριθμός	30				

Εισαγωγή του 30

Queue

θέση	**1**	**2**	**3**	**4**	**5**
αριθμός	30	17			

Εισαγωγή του 17

Queue

θέση	**1**	**2**	**3**	**4**	**5**
αριθμός	30	17	25		

Εισαγωγή του 25

Queue

θέση	1	2	3	4	5
αριθμός		17	25		

Διαγραφή του 30

Queue

θέση	1	2	3	4	5
αριθμός			25		

Διαγραφή του 17

Queue

θέση	1	2	3	4	5
αριθμός			25	7	

Εισαγωγή του 7

Queue

θέση	1	2	3	4	5
αριθμός			25	7	53

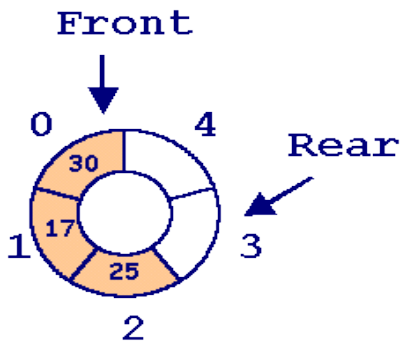
Εισαγωγή του 53

Στο σημείο αυτό, για να εισάγουμε έναν νέο αριθμό στην ουρά θα πρέπει πρώτα να μετατοπίσουμε τους 3 αριθμούς στις πρώτες θέσεις του

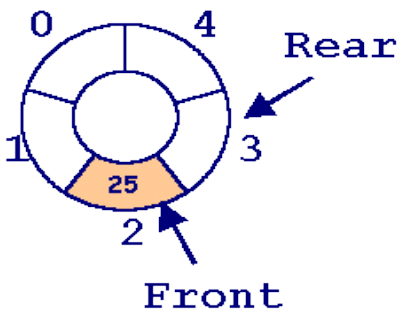
Queue

θέση	1	2	3	4	5
αριθμός	25	7	53		

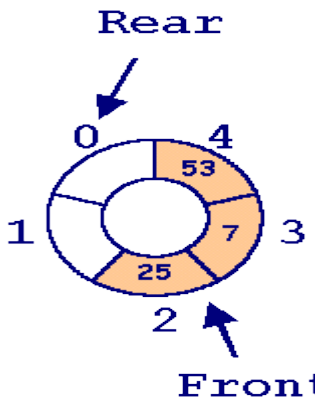
Για να αποφύγουμε τις μετατοπίσεις, μπορούμε να θεωρήσουμε έναν κυκλικό πίνακα, όπου το πρώτο στοιχείο ακολουθεί το τελευταίο. Αυτό μπορεί να γίνει αν δεικτοδοτήσουμε τον πίνακα, ξεκινώντας από το 0, και αυξάνουμε τις μεταβλητές Front και Rear, ώστε να παίρνουν τιμές από 0 μέχρι το όριο της ουράς. Οι παραπάνω πράξεις εισαγωγής και διαγραφής γίνονται όπως δείχνουν τα ακόλουθα σχήματα.



Εισαγωγή των 30, 17 και 25



Διαγραφή των 30 και 17

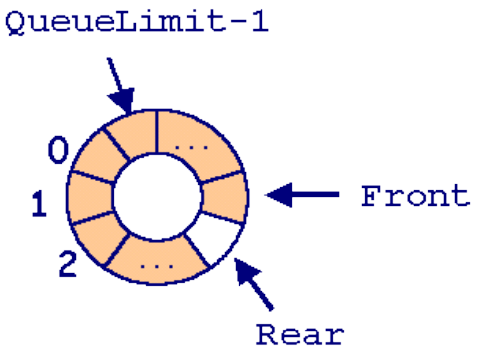


Εισαγωγή των 7 και 53

Τώρα μπορούμε να εισαγάγουμε νέο στοιχείο στη θέση 0 χωρίς να χρειάζεται να μετακινήσουμε τα υπόλοιπα.

Ας εξετάσουμε τη βασική διαδικασία EmptyQ που καθορίζει αν η ουρά είναι κενή. Αν η ουρά περιέχει ένα μόνο στοιχείο, τότε αυτό βρίσκεται στη θέση Front του πίνακα και η Rear είναι η κενή θέση που ακολουθεί. Αν αυτό το στοιχείο διαγραφεί, η μεταβλητή Front αυξάνεται κατά 1 και η Rear έχει την ίδια τιμή. Επομένως, για να καθορίσουμε αν η ουρά είναι κενή, το μόνο που χρειάζεται να κάνουμε είναι να ελέγξουμε τη συνθήκη Front == Rear. Αρχικά, η CreateQ θέτει τις Front και Rear ίσες με 0.

Όπως στην υλοποίηση της στοίβας με πίνακα υπήρχε η πιθανότητα γεμάτης στοίβας, έτσι και στην υλοποίηση της ουράς προκύπτει η πιθανότητα γεμάτης ουράς. Για να δούμε πώς μπορεί να εξεταστεί η συνθήκη γεμάτης ουράς, υποθέτουμε ότι ο πίνακας είναι σχεδόν γεμάτος, με μόνο μία κενή θέση:



Αν ένα στοιχείο αποθηκευόταν σ' αυτήν τη θέση, τότε η Rear θα αυξανόταν κατά 1 και θα είχε την ίδια τιμή με την Front. Όμως, η συνθήκη Front == Rear δηλώνει ότι η ουρά είναι κενή. Έτσι, λοιπόν, δεν θα μπορούσαμε να ξεχωρίσουμε μια κενή ουρά από μια γεμάτη αν αυτή η θέση χρησιμοποιούνταν για την αποθήκευση ενός στοιχείου. Κάτι τέτοιο μπορεί να αποφευχθεί αν διατηρούμε μια κενή θέση στον πίνακα. Επομένως, η συνθήκη που δείχνει αν η ουρά είναι γεμάτη είναι τώρα η (Rear+1) **%** QueueLimit == Front, όπου QueueLimit είναι το μέγιστο μέγεθος της ουράς.

Για να υλοποιήσουμε, λοιπόν, μια ουρά, μπορούμε να χρησιμοποιήσουμε ως αποθηκευτική δομή μια εγγραφή αποτελούμενη από έναν κυκλικό πίνακα, τον Element, για την αποθήκευση των στοιχείων της ουράς, και από τα πεδία Front και Rear, όπου αποθηκεύουμε τη θέση της εμπρός άκρης της ουράς και τη θέση που ακολουθεί αμέσως μετά την πίσω άκρη της ουράς.

```
#define QueueLimit 20  /*μέγιστο μέγεθος της ουράς*/

typedef int QueueElementType;   /* ο τύπος των στοιχείων της ουράς ενδεικτικά
τύπου int */

typedef struct {
        int Front, Rear;
        QueueElementType Element[QueueLimit];
} QueueType;
```

Η λειτουργία δημιουργίας μιας κενής ουράς συνίσταται απλά στο να τεθούν οι μεταβλητές Front και Rear της ουράς ίσες με 0, και μια ουρά είναι κενή όταν η boolean έκφραση Queue.Front == Queue.Rear είναι αληθής.

Παρακάτω φαίνεται ένας **αλγόριθμος για τη λειτουργία πρόσθεσης ενός στοιχείου στην ουρά** (AddQ):

AddQ

/*Αλγόριθμος εισαγωγής ενός στοιχείου *Item* στην πίσω άκρη της ουράς *Queue*, εφόσον ο πίνακας *Element* δεν είναι γεμάτος*/

1. *NewRear* ← (*Queue*.*Rear* + 1) **%** *QueueLimit*

   /*Υπολογισμός της πρώτης κενής θέσης στην πίσω άκρη της ουράς, μετά από την εισαγωγή του στοιχείου *Item**/

2. **Αν** *NewRear != Queue*.*Front* **τότε** /*Αν η ουρά δεν είναι γεμάτη*/

   α. *Queue*.*Element*[*Queue*.*Rear*] ← *Item*

   /*Θέσε στην πίσω άκρη της ουράς, δηλαδή στη θέση *Queue.Rear* του πίνακα *Element* την τιμή *Item**/

   β. *Queue*.*Rear* ← *NewRear* /*Ενημέρωσε την τιμή της *Queue.Rear* ώστε να δείχνει στην 1η κενή θέση στην πίσω άκρη της ουράς*/

Αλλιώς

Γράψε 'Προέκυψε σφάλμα ουράς: γεμάτη ουρά'

Τέλος_ αν

Παρακάτω φαίνεται ένας **αλγόριθμος για τη λειτουργία αφαίρεσης ενός στοιχείου από την ουρά** (RemoveQ):

RemoveQ

/*Αλγόριθμος ανάκτησης και διαγραφής του στοιχείου *Item* από την εμπρός άκρη της ουράς *Queue*, εφόσον ο πίνακας *Element* δεν είναι κενός*/

Αν η ουρά δεν είναι κενή, **τότε**

α. *Item* ← *Queue*.*Element*[*Queue*.*Front*]

/*Θέσε την Item ίση με το πρώτο στοιχείο της ουράς*/

β. *Queue*.*Front* ← (*Queue*.*Front*+1) **%** *QueueLimit*

/*Ενημέρωσε την τιμή της *Front* ώστε να δείχνει στη νέα εμπρός άκρη της ουράς*/

Αλλιώς

Γράψε 'Προέκυψε σφάλμα ουράς: κενή ουρά'

Τέλος_ αν

Επειδή και οι ουρές, όσο και οι στοίβες, είναι χρήσιμες στην επίλυση προβλημάτων, θα ήταν ωφέλιμο να είχαμε έναν τύπο δεδομένων Ουρά, όπως είχαμε και για τη στοίβα. Έτσι κατασκευάστηκε ένα πακέτο σε C γι' αυτόν τον τύπο δεδομένων, το οποίο περιλαμβάνει τις απαιτούμενες δηλώσεις καθώς και τις διαδικασίες και συναρτήσεις που υλοποιούν τις βασικές λειτουργίες και σχέσεις της ουράς. Παρακάτω φαίνεται μια διασύνδεση (αρχείο κεφαλίδας) (QueueADT.h) και η υλοποίηση του (QueueADT.c) για τον τύπο δεδομένων Ουρά, που μπορεί να χρησιμοποιηθεί σε προγράμματα C.

```
                    /*Πακέτο για τον ΑΤΔ Ουρά*/

/*Filename: QueueADT.h */

#define QueueLimit 20                  /*μέγιστο μέγεθος της ουράς*/

typedef int QueueElementType;          /* ο τύπος των στοιχείων της ουράς
                                          ενδεικτικά τύπου int */

typedef struct {
   int Front, Rear;
   QueueElementType Element[QueueLimit];
} QueueType;

typedef enum {FALSE, TRUE} boolean;

void CreateQ(QueueType *Queue);
boolean EmptyQ(QueueType Queue);
boolean FullQ(QueueType Queue);
void RemoveQ(QueueType *Queue, QueueElementType *Item);
void AddQ(QueueType *Queue, QueueElementType Item);

/*Filename: QueueADT.c */

#include <stdio.h>
#include "QueueADT.h"

void CreateQ(QueueType *Queue)
/*  Λειτουργία:  Δημιουργεί μια κενή ουρά.
    Επιστρέφει:  Κενή ουρά.*/
{
     Queue->Front = 0;
     Queue->Rear = 0;
}

boolean EmptyQ(QueueType Queue)
/*Δέχεται:          Μια ουρά.
  Λειτουργία:       Ελέγχει αν η ουρά είναι κενή.
  Επιστρέφει:       TRUE αν η ουρά είναι κενή, FALSE διαφορετικά.*/
{
        return (Queue.Front == Queue.Rear);
}

boolean FullQ(QueueType Queue)
/*Δέχεται:          Μια ουρά.
  Λειτουργία:       Ελέγχει αν η ουρά είναι γεμάτη.
  Επιστρέφει:       TRUE αν η ουρά είναι γεμάτη, FALSE διαφορετικά.*/
{
        return ((Queue.Front) == ((Queue.Rear +1) % QueueLimit));
}
```

```
 void RemoveQ(QueueType *Queue, QueueElementType *Item)
/*Δέχεται:          Μια ουρά.
Λειτουργία:        Αφαιρεί το στοιχείο Item από την εμπρός άκρη της ουράς αν η ουρά
                   δεν είναι κενή.
Επιστρέφει:        Το στοιχείο Item και την τροποποιημένη ουρά.
Έξοδος:            Μήνυμα κενής ουρά αν η ουρά είναι κενή.*/
{
        if(!EmptyQ(*Queue))
        {
                *Item = Queue ->Element[Queue -> Front];
                Queue ->Front  = (Queue ->Front + 1) % QueueLimit;
        }
        else
                printf("Empty Queue\n");
}

void AddQ(QueueType *Queue, QueueElementType Item)
/*Δέχεται:          Μια ουρά Queue και ένα στοιχείο Item.
  Λειτουργία:      Προσθέτει το στοιχείο Item στην ουρά Queue αν η ουρά δεν είναι
                   γεμάτη.
  Επιστρέφει:      Την τροποποιημένη ουρά.
  Έξοδος:          Μήνυμα γεμάτης ουράς αν η ουρά είναι γεμάτη.*/
{
    int NewRear;

        if(!FullQ (*Queue))
        {
                NewRear = (Queue ->Rear + 1) % QueueLimit;
                Queue ->Element[Queue ->Rear] = Item;
                Queue ->Rear = NewRear;
        }
        else
                printf("Full Queue\n");
}
```

Εφόσον έχουμε έτοιμο το πακέτο για τον ΑΤΔ Ουρά, μπορούμε να το χρησιμοποιήσουμε σε ένα πρόγραμμα-πελάτη που θα υλοποιεί το πρόβλημα της δημιουργίας προβλημάτων άσκησης-εξάσκησης για στοιχειώδη αριθμητική, που περιγράφτηκε στην ενότητα 2.1. Ένα τέτοιο πρόγραμμα είναι το πρόγραμμα-πελάτης DrillAndPractice.c, το οποίο χρησιμοποιεί το QProbADT.h (υλοποίηση QProbADT.c) για τον ΑΤΔ Ουρά. Η μόνη διαφορά του από το QueueADT.c είναι ότι τα στοιχεία της ουράς είναι τύπου δομής struct {int Addend1, Addend2;} και όχι int.