

# ΕΠΛ232 – Προγραμματιστικές Τεχνικές και Εργαλεία

Δομές Δεδομένων Ι (Στοιίβες & Ουρές)  
Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου

<http://www.cs.ucy.ac.cy/courses/EPL232>

# Περιεχόμενο Διάλεξης 10

- **Δομές Δεδομένων**
  - Εισαγωγή και Παραδείγματα Δομών
  - Αυτοαναφορικές Δομές Δεδομένων
  - Δείκτες-σε-Δείκτες & Δυναμικοί Πολυδιάστατοι Πίνακες (μη-συνεχόμενοι)
  - Δημιουργία Απλής Μονά-Συνδεδεμένης Λίστας
- **Η Δομή Δεδομένων Στοίβα**
  - Λογική και Δηλώσεις
  - Υλοποίηση με Δυναμική Δέσμευση Μνήμης
- **Η Δομή Δεδομένων Ουρά**
  - Λογική και Δηλώσεις
  - Υλοποίηση με Δυναμική Δέσμευση Μνήμης

# Δομές Δεδομένων (Data Structures)

- Η **δυναμική δέσμευση** μνήμης, που είδαμε σε προηγούμενη διάλεξη χρησιμοποιείται κατά κύριο λόγο για **αποθήκευση αντικειμένων** τύπου **structure**.
  - Δεν συνηθίζεται για απλούστερους τύπους (π.χ., float, int, char κλπ.)
- Αυτό γιατί μπορούμε μέσω αντικειμένων τύπου structure να **φτιάξουμε κόμβους**, να τους **συνδέσουμε** μεταξύ τους και να δημιουργήσουμε:
  - Συνδεδεμένη λίστα
  - Στοίβες, Ουρές και λίστες αναμονής,
  - Δέντρα, Γράφοι, κλπ.



# Δομές Δεδομένων (Data Structures)

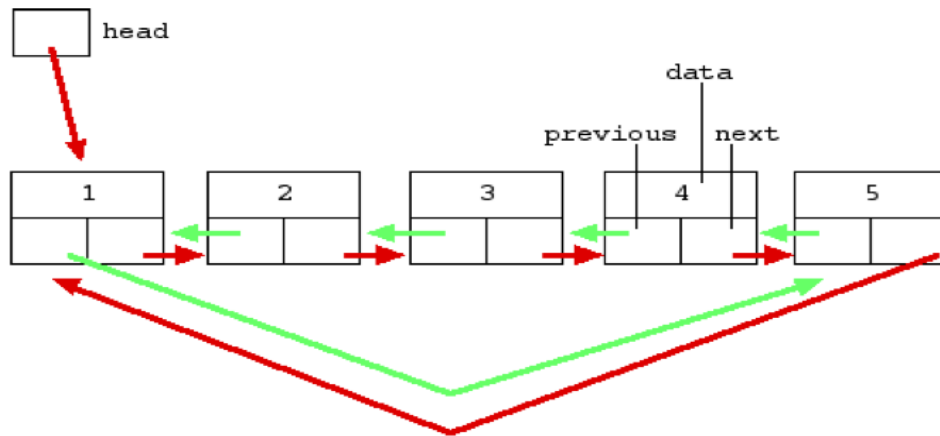
- Οι δομές δεδομένων είναι ένα δομικός τρόπος για την **οργάνωση δεδομένων** στη μνήμη ενός υπολογιστή.
- Μερικές από τις πιο συχνά χρησιμοποιούμενες δομές δεδομένων περιλαμβάνουν **λίστες, πίνακες, στοίβες, ουρές, σωρούς, δέντρα** και **γραφήματα**.
- Ο τρόπος με τον οποίο οργανώνονται τα δεδομένα επηρεάζει την απόδοση ενός προγράμματος (ειδικό για διαφορετικές εργασίες). Για παράδειγμα,
  - Μια ουρά είναι μια καλή δομή δεδομένων για την αποθήκευση δεδομένων που πρέπει να διατηρηθούν σε σειρά, π.χ. ένα σύνολο εγγράφων που περιμένουν να εκτυπωθούν σε έναν εκτυπωτή δικτύου.
  - Ένα δυαδικό δέντρο είναι μια καλή δομή δεδομένων για την αναζήτηση ταξινομημένων δεδομένων. Το μεσαίο στοιχείο από τη λίστα αποθηκεύεται στον ριζικό κόμβο, με μικρότερα στοιχεία στα αριστερά και τα μεγαλύτερα στοιχεία στα δεξιά.



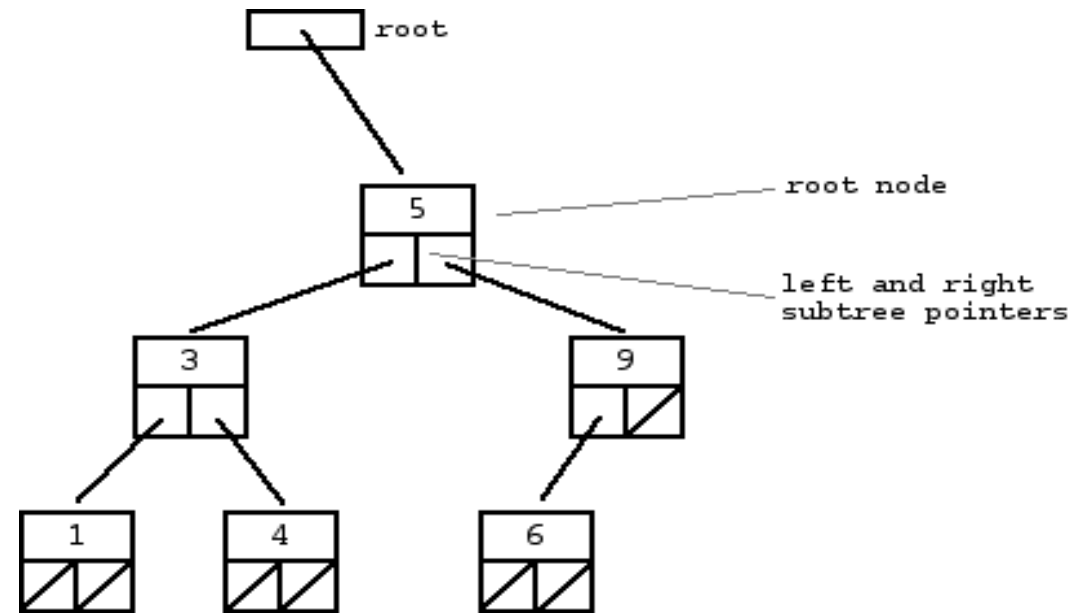
# Δομές Δεδομένων (Data Structures)

## Παραδείγματα Δομών Δεδομένων στην Μνήμη

### Κυκλική Διπλά Συνδεδεμένη Λίστα

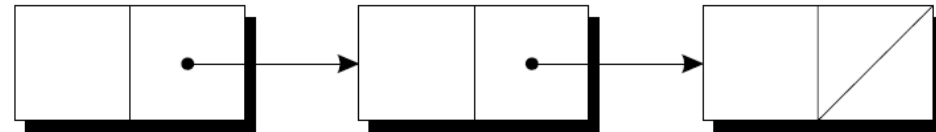


### Δυαδικό Δένδρο (Αναζήτησης)



# Δημιουργία (Απλής) Μονά-Συνδεδεμένης Λίστας

- Μια **συνδεδεμένη λίστα** αποτελείται από μια αλυσίδα δομών (που ονομάζονται κόμβοι), με κάθε κόμβο να περιέχει ένα δείκτη στον επόμενο κόμβο της αλυσίδας:



- Ο τελευταίος κόμβος στη λίστα περιέχει έναν μηδενικό δείκτη (NULL).
- Μια συνδεδεμένη λίστα είναι πιο ευέλικτη από έναν πίνακα: μπορούμε εύκολα να εισάγουμε και να διαγράψουμε κόμβους, επιτρέποντας στη λίστα να μεγαλώνει και να συρρικνώνεται ανάλογα με τις ανάγκες μας.
- Για να ορίσουμε μια συνδεδεμένη λίστα, θα χρειαστούμε μια δομή που να αντιπροσωπεύει έναν μεμονωμένο κόμβο.
  - Μια δομή κόμβου θα περιέχει δεδομένα (ένας ακέραιος στο παράδειγμα που ακολουθεί) συν ένα δείκτη στον επόμενο κόμβο της λίστας (**Αυτοαναφορικές Δομές Δεδομένων**).



# Αυτοαναφορικές Δομές Δεδομένων

- **Αυτοαναφορική Δομή Δεδομένων (Self-referential Data Structure):** Μια δομή η οποία στη δήλωση της **φέρει πεδίο** το οποίο αναφέρεται στον **ίδιο τύπο δομής** με αυτό της δήλωσης.
- **Ορισμός Κόμβου Λίστας Ακέραιων:**

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

OK αλλά χωρίς typedef

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

**ΣΥΝΙΣΤΟΜΕΝΟΣ ΟΡΙΣΜΟΣ**

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

Λάθος Μεταγλώττισης

- Όπως όλα τα typedef, να τοποθετούνται στην **αρχή του προγράμματος** και αργότερα σε **αρχεία κεφαλίδας .h**
- Ακολουθούν **παραδείγματα χρήσης** στην επόμενη διαφάνεια.



# Δημιουργία (Απλής) Μονά-Συνδεδεμένης Λίστας

- Στη συνέχεια, θα χρειαστούμε μια μεταβλητή που πάντα οδηγεί στον πρώτο κόμβο της λίστας:

```
struct node *first = NULL;
```

- Ορίζοντας το `first` σαν `NULL` δηλώνει ότι η λίστα είναι αρχικά κενή.
- Καθώς κατασκευάζουμε μια συνδεδεμένη λίστα, θα δημιουργήσουμε κόμβους, προσθέτοντας ένα προς ένα στη λίστα.
- Τα βήματα για τη δημιουργία μιας λίστας:
  1. Εκχώρηση μνήμης για τον κόμβο.
  2. Αποθήκευση δεδομένων στον κόμβο.
  3. Εισαγωγή του κόμβου στη λίστα.





# Δημιουργία (Απλής) Μονά-Συνδεδεμένης Λίστας

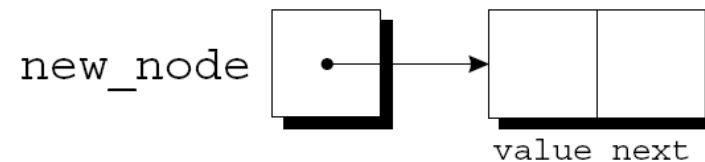
- Όταν δημιουργούμε έναν κόμβο, θα χρειαστούμε μια μεταβλητή που μπορεί να δείχνει προσωρινά τον κόμβο:

```
struct node *new_node;
```

- Θα χρησιμοποιήσουμε τη `malloc` για να εκχωρήσετε μνήμη για τον νέο κόμβο, επιστρέφοντας την τιμή στο `new_node`:

```
new_node = malloc(sizeof(struct node));
```

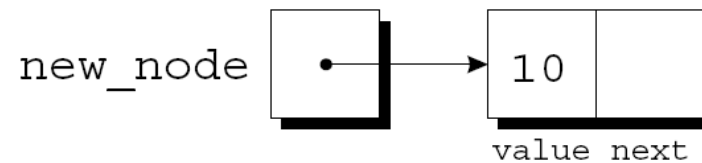
- Το `new_node` δείχνει σε ένα μπλοκ μνήμης, τόσο μεγάλο όσο να κρατήσει μια δομή κόμβων:



# Δημιουργία (Απλής) Μονά-Συνδεδεμένης Λίστας

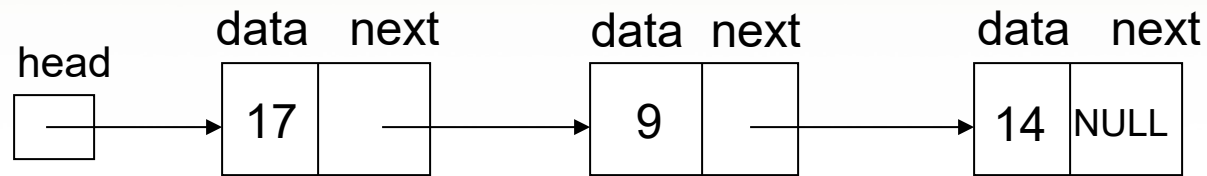
- Στη συνέχεια, θα αποθηκεύουμε δεδομένα στο `value` το οποίο είναι μέλος του νέου κόμβου:

```
(*new_node).value = 10;
```



- Οι παρενθέσεις γύρω από το `*new_node` είναι υποχρεωτικές επειδή ο τελεστής `.` θα είχε προτεραιότητα έναντι του τελεστή `*`.
- Ακολουθεί παράδειγμα...

# Δημιουργία (Απλής) Μονά-Συνδεδεμένης Λίστας

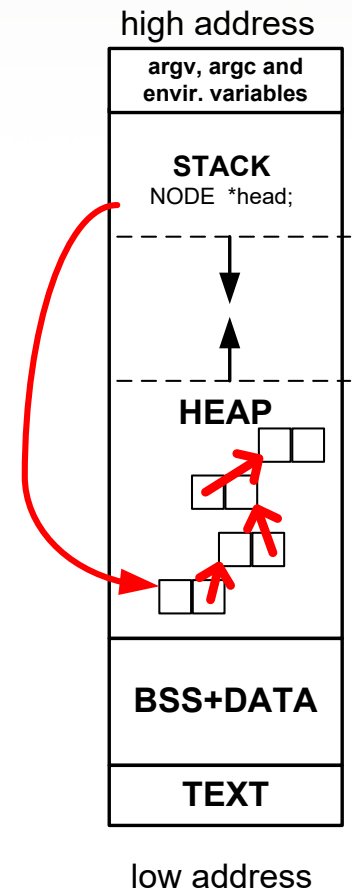


- Την παραπάνω συνδεδεμένη λίστα θα μπορούσαμε να την υλοποιήσουμε ως εξής:

```
NODE *head, *p1, *p2; // ή struct node *head, *p1, *p2
head = (NODE *) malloc (sizeof(NODE));
p1 = (NODE *) malloc (sizeof(NODE));
p2 = (NODE *) malloc (sizeof(NODE));
head->data = 17;
p1 -> data = 9;
p2 -> data = 14;
head->next = p1;
p1->next = p2;
p2->next = NULL;
```

- Σημείωση: οι δείκτες p1 και p2 είναι στο παρόν στάδιο βοηθητικοί για την υλοποίηση της λίστας εάν και δε χρειάζονται απαραίτητα

## ΔΙΕΡΓΑΣΙΑ



# Συνδεδεμένη Λίστα έναντι Πίνακα

- **Πλεονεκτήματα Λίστας**

- Εύκολες Εισαγωγές και Εξαγωγές κόμβων με προσαρμογή δεικτών.
- Δεν χρειάζεται προσδιορισμένο μέγεθος η λίστα.

- **Μειονεκτήματα Λίστας**

- Χάνουμε τη δυνατότητα για **τυχαία προσπέλαση**, π.χ.,  $a[i]$ .
- Είναι **προγραμματιστικά** πιο **δύσκολο** να υλοποιηθεί.

# Συνδεδεμένη Λίστα

- Σε επόμενες διαλέξεις θα δούμε πως:
  - Εισαγωγή νέου κόμβου στην αρχή μιας συνδεδεμένης λίστας
  - Αναζήτηση μέσα σε μια συνδεδεμένη λίστα
  - Διαγραφή ενός κόμβου από μια συνδεδεμένη λίστα
- **Ταξινομημένη λίστα:** Όταν οι κόμβοι μιας λίστας διατηρούνται σε σειρά — βάση των δεδομένων που είναι αποθηκευμένα μέσα στους κόμβους — λέμε ότι η λίστα είναι ταξινομημένη.
  - Γρήγορη Αναζήτηση – Δύσκολη νέα καταχώρηση

# Δείκτες σε Δείκτες (Pointers to Pointers)

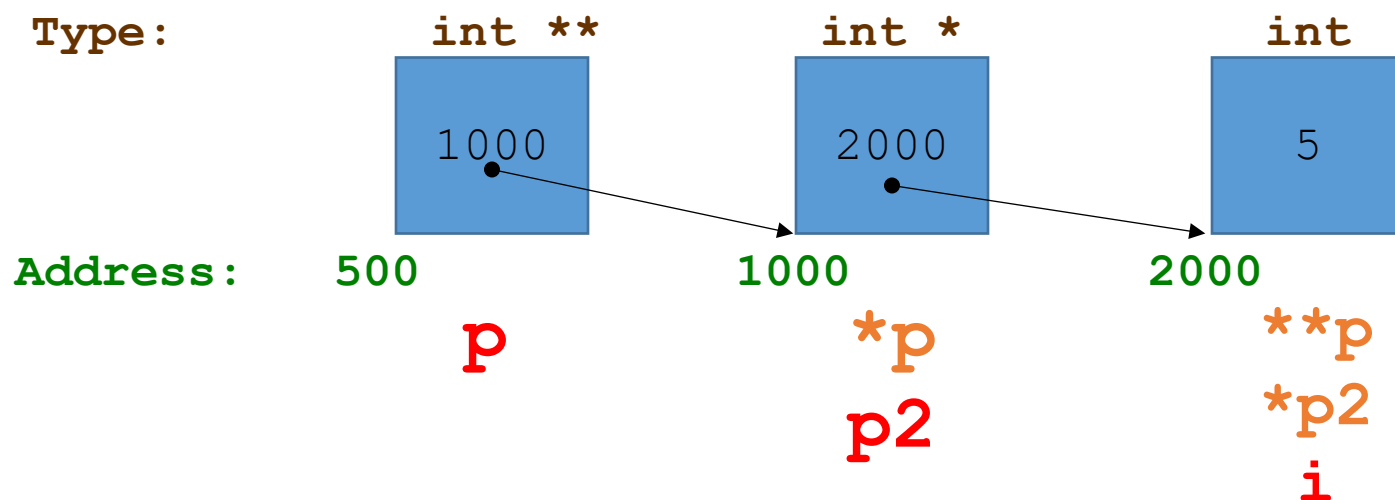
- Η έννοια των "δεικτών σε δείκτες" χρησιμοποιείται συχνά στο πλαίσιο των συνδεδεμένων δομών δεδομένων.
- Συγκεκριμένα, όταν ένα όρισμα σε μια συνάρτηση είναι μια μεταβλητή δείκτη, πολλές φορές είναι επιθυμητό να θέλουμε να τροποποιηθεί η μεταβλητή (**τοπική εμβέλεια**).
- Αυτό απαιτεί τη χρήση ενός δείκτη που δείχνει σε ένα άλλο δείκτη.

# ΔΕΙΚΤΕΣ ΣΕ ΔΕΙΚΤΕΣ (Pointers to Pointers)

```
int **p = NULL;
int *p2 = NULL;
int i = 5;
p2 = &i;
p = &p2;
```

```
typedef unsigned long int ADDR;
```

```
printf("%ld", (ADDR) &p); // 500
printf("%ld", (ADDR) p); // 1000
printf("%ld", (ADDR) &p2); // 1000
printf("%ld", (ADDR) *p); // 2000
printf("%ld", (ADDR) p2); // 2000
printf("%d", **p); // 5
printf("%d", *p2); // 5
printf("%d", i); // 5
```



# Δείκτες σε Δείκτες (Pointers to Pointers)

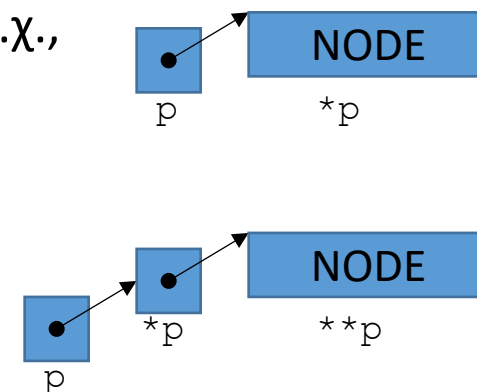
- Όταν ένα όρισμα μιας συνάρτησης είναι δείκτης (π.χ., `*p`) τότε το αντικείμενο περνά δια-διεύθυνσης, αλλά η ίδια η αναφορά (`p`) έχει τοπική εμβέλεια.

- Συνεπώς τυχούσα αλλαγή της αναφοράς `p` δε θα είχε αντίκτυπο εξωτερικά, π.χ.,

- ```
void initNode(NODE *p) {  
    p->data = 4; // OK  
    p = (NODE *) malloc(sizeof(NODE)); // ERROR }  
Με δείκτη-σε-δείκτη διορθώνεται το πρόβλημα:
```

- Με δείκτη-σε-δείκτη διορθώνεται το πρόβλημα:

```
void initNode(NODE **p) {  
    *p = (NODE *) malloc(sizeof(NODE)); // OK  
    (*p)->data = 4; // OK  
    (*p).data = 4 // ERR ίδιο με p->data = 4 }  
  
Main: NODE *p = NULL; initNode(&p)  
      ή      NODE **p = NULL; initNode(p)
```





# Συνεχόμενος Πολυδιάστατος Πίνακας

## Δυναμικά Δεσμευμένοι Πίνακες

- Μια δεύτερη εφαρμογή του δείκτη-σε-δείκτη είναι και η δημιουργία **δυναμικών πολυδιάστατων** πινάκων
- Στη **διάλεξη 9** είδαμε την ακόλουθη λύση:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *a = NULL;
    int n;
    printf("Enter Matrix (nxn) Size:");
    scanf("%d", &n);
    int k = 0;
    a = (int *) malloc(n * n * sizeof(int));
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            a[i*n + j] = ++k;

    free(a);
    return 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, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
```

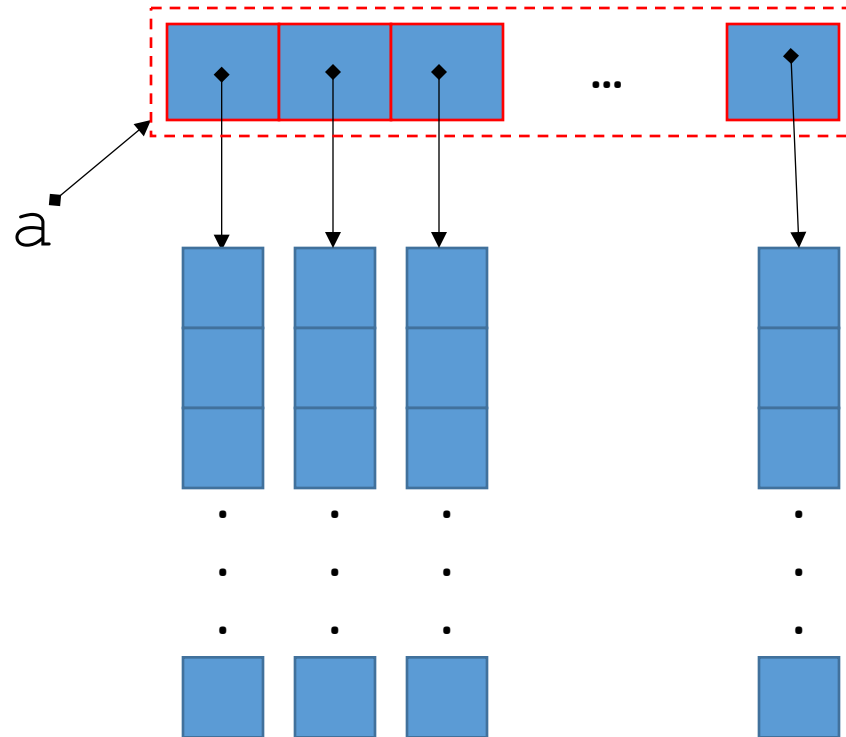
Πρόβλημα: Θα ήταν  
ευκολότερη η αναφορά σε  
 $a[i][j]$  παρά  $a[i*n+j]$ !



# Μη-Συνεχόμενος Πολυδιάστατος Πίνακας

## Δυναμικά Δεσμευμένοι Πίνακες

- Μια καλύτερη λύση είναι με τη χρήση **μη-συνεχόμενου δυναμικού πολυδιάστατου πίνακα**.



```
int **a = NULL; int N = 10;
...
// Δέσμευση Πίνακα Δεικτών
a = (int **)
    malloc(sizeof(int *) * N);

...
// Δέσμευση Πίνακα για a[i]
for(i=0; i<N; i++) {
    a[i] = (int *)
        malloc(sizeof(int) * N);
    ...
}
// Χρήση Στοιχείων Πίνακα
a[0][0] = 10;
```



# Μη-Συνεχόμενος Πολυδιάστατος Πίνακας

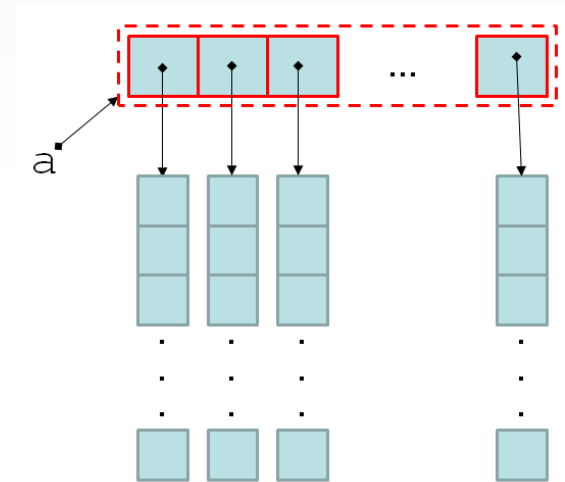
## Δυναμικά Δεσμευμένοι Πίνακες

### Ολοκληρωμένο Παράδειγμα:

```
#include <stdlib.h>
int main() {
    int **a = NULL, N;
    printf("Enter Matrix (NxN) Size:");
    scanf("%d", &N);

    /* create list of pointers */
    a = (int **) malloc(sizeof(int *) * N);
    if (a == NULL) {
        printf("Error: Unable to allocate enough memory!");
        return(EXIT_FAILURE);
    }

    /* allocate vector for each a[i] */
    for(int i=0; i<N; i++) {
        a[i] = (int *) malloc(sizeof(int) * N);
        if (a[i] == NULL) {
            printf("Error: Unable to allocate enough memory!");
            return(EXIT_FAILURE);
        }
    }
}
```



# Μη-Συνεχόμενος Πολυδιάστατος Πίνακας

## Δυναμικά Δεσμευμένοι Πίνακες

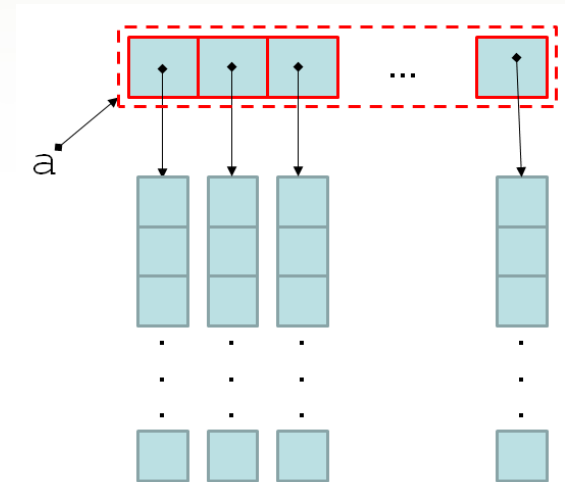
... Συνέχεια

```
int k = 0;
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        a[i][j] = ++k;

for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
        printf("%3d,", a[i][j]);
    }
    printf("\n");
}

for(int i=0; i<N; i++)
    free(a[i]);
free(a);

return 0;
}
```



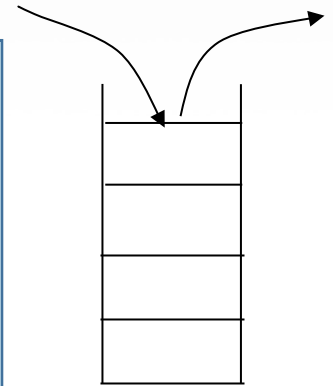
|     |     |     |     |     |     |     |     |     |      |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 1,  | 2,  | 3,  | 4,  | 5,  | 6,  | 7,  | 8,  | 9,  | 10,  |
| 11, | 12, | 13, | 14, | 15, | 16, | 17, | 18, | 19, | 20,  |
| 21, | 22, | 23, | 24, | 25, | 26, | 27, | 28, | 29, | 30,  |
| 31, | 32, | 33, | 34, | 35, | 36, | 37, | 38, | 39, | 40,  |
| 41, | 42, | 43, | 44, | 45, | 46, | 47, | 48, | 49, | 50,  |
| 51, | 52, | 53, | 54, | 55, | 56, | 57, | 58, | 59, | 60,  |
| 61, | 62, | 63, | 64, | 65, | 66, | 67, | 68, | 69, | 70,  |
| 71, | 72, | 73, | 74, | 75, | 76, | 77, | 78, | 79, | 80,  |
| 81, | 82, | 83, | 84, | 85, | 86, | 87, | 88, | 89, | 90,  |
| 91, | 92, | 93, | 94, | 95, | 96, | 97, | 98, | 99, | 100, |



# Η Δομή Δεδομένων Στοίβα

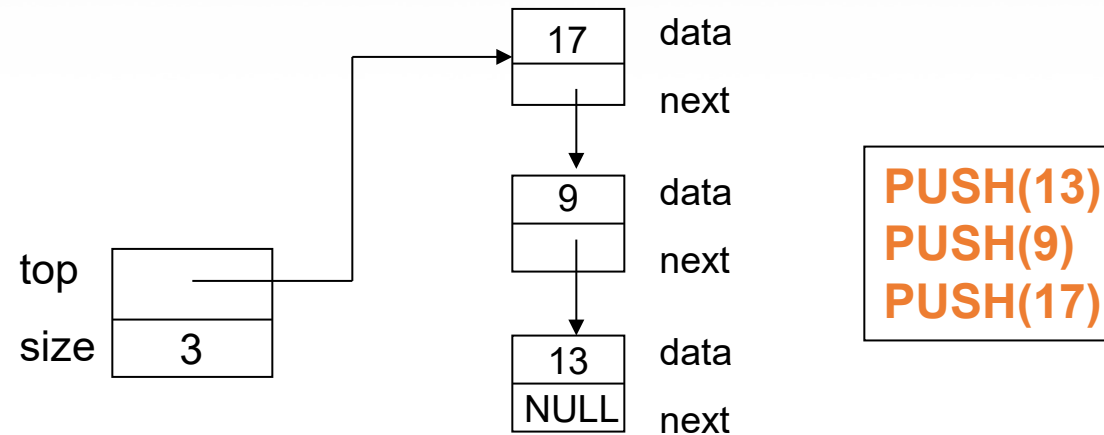
(με Δυναμική Χορήγηση Μνήμης)

- Μια στοίβα είναι μια λίστα στοιχείων που συνοδεύεται από τις διαδικασίες
  - **push**, για εισαγωγή στοιχείου στη λίστα, και
  - **pop**, για εξαγωγή του τελευταία εισαγμένου στοιχείου της λίστας.
- Για ευκολία στην διεκπεραίωση των **δύο διαδικασιών** θα ήταν χρήσιμο, ανά πάσα στιγμή να γνωρίζουμε που βρίσκεται η **κεφαλή της στοίβας** (και το μέγεθός της).
- Η **νέα εικόνα** της στοίβας που θα θέλαμε να έχουμε βασισμένη σε **συνδεδεμένη λίστα** είναι αυτή που δίνεται στην **επόμενη διαφάνεια**.



# Η Δομή Δεδομένων Στοίβα

(με Δυναμική Χορήγηση Μνήμης)



- Προσέξτε ότι για την παραπάνω υλοποίηση απαιτείται να καθορίσουμε δύο πράγματα:
  1. Τη **μορφή του κόμβου** της **συνδεδεμένης λίστας** και
  2. Τη **μορφή του κόμβου** που κρατά πληροφορίες για τη **στοίβα γενικά** όπως την κορυφή της στοίβας και το μέγεθός της.



# Η Δομή Δεδομένων Στοίβα

## (Δηλώσεις και Αρχικοποίηση)

- Συνεπώς, ως πρώτο βήμα για την υλοποίηση της στοίβας απαιτούνται οι παρακάτω δηλώσεις κόμβων:

```
typedef struct node {  
    int      data;  
    struct node *next;  
} NODE;  
  
typedef struct {  
    NODE *top;  
    int  size;  
} STACK;
```

- Οι ορισμοί λοιπόν και οι κλήσεις (4 εναλλακτικοί τρόποι):

### A) Στατικά στο main()

```
STACK stack;  
stack.top = NULL;  
stack.size = 0;  
foo (&stack) ;
```

```
void foo(STACK *stack) ;
```

### B) Δυναμικά στο main()

```
STACK *stack = NULL;  
stack = (STACK *)  
        malloc(sizeof(STACK)) ;  
stack->top = NULL;  
stack->size = 0;  
foo (stack) ;
```



# Η Δομή Δεδομένων Στοίβα

## (Δηλώσεις και Αρχικοποίηση)

→ Κατά την έξοδο, η αναφορά `stack` χάνεται όχι όμως ο χώρος που δεσμεύτηκε με την `malloc` (η διεύθυνση του οποίου επιστρέφεται)

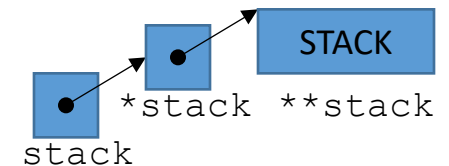
### Γ) Δυναμικά σε Συνάρτηση (με return pointer):

```
(STACK *) initStack() {  
    STACK *stack = (STACK *) malloc(sizeof(STACK));  
    if (stack == NULL) return NULL;  
    stack->top = NULL;  
    stack->size = 0;  
    return stack;  
}
```

```
int main() {  
    STACK *stack = NULL;  
    stack = initStack();  
    ...  
}
```

### Δ) Δυναμικά σε Συνάρτηση (με δείκτη-σε-δείκτη):

```
int initStack2(STACK **stack) {  
    *stack = (STACK *) malloc(sizeof(STACK));  
    if ((*stack) == NULL) return EXIT_FAILURE;  
    (*stack)->top = NULL;  
    (*stack)->size = 0;  
    return EXIT_SUCCESS;  
}
```



```
int main() {  
    STACK *stack;  
    initStack2(&stack);  
    ...  
}
```





# Η Δομή Δεδομένων Στοίβα

(με Δυναμική Χορήγηση Μνήμης)

- Μια **στοίβα** μπορεί να έχει **διάφορες συναρτήσεις**, όπως:

- `STACK *initStack();`
- `int initStack2(STACK **stack);`
- `bool IsEmptyStack(STACK *stack);`
- `void top(STACK *stack);`
- `int push(int value, STACK *stack);`
- `int pop(STACK *stack, int *retval);`

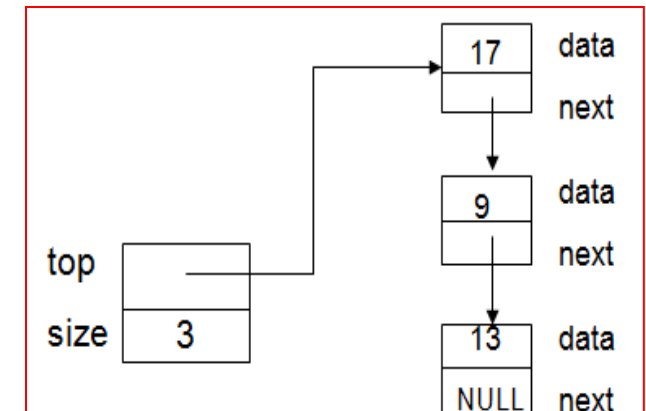
- Αυτές οι συναρτήσεις θα πρέπει για λόγους καλύτερης **οργάνωσης του κώδικα** να τοποθετηθούν σε ξεχωριστό αρχείο
  - Η οργάνωση του κώδικα σε πολλαπλά αρχεία θα μελετηθεί στις επόμενες διαλέξεις (11 και 12).



# Η Δομή Δεδομένων Στοίβα

(με Δυναμική Χορήγηση Μνήμης)

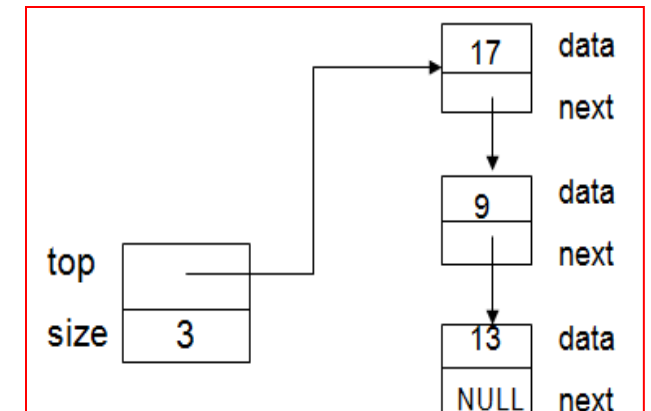
```
int push(int value, STACK *s) {  
  
    NODE *p = NULL;  
    if (s == NULL){ return EXIT_FAILURE;}  
  
    p = (NODE *) malloc(sizeof(NODE));  
    if ( p == NULL ) {  
        printf("System out of memory!\n");  
        return EXIT_FAILURE;  
    }  
    p->data = value;        // assign to new node  
    p->next = s->top;      // adjust next pointer  
    s->top = p;           // point head to this node  
    (s->size)++;         // increase stack size  
    return EXIT_SUCCESS;  
}
```



# Η Δομή Δεδομένων Στοίβα

(με Δυναμική Χορήγηση Μνήμης)

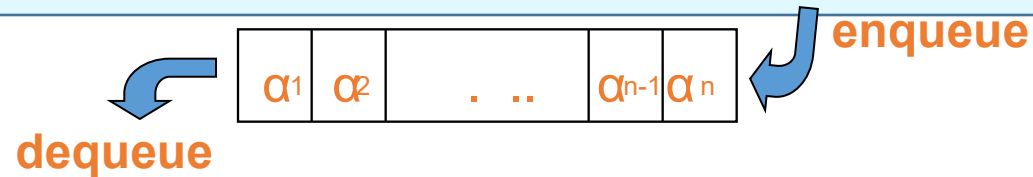
```
int pop(STACK *s, int *retval) {  
  
    NODE *p = NULL;  
    if ( s == NULL || s->size == 0 ) {  
        printf("Sorry, stack is empty...\n");  
        return EXIT_FAILURE;  
    }  
    if (retval == NULL) {  
        printf("Retval is null"); return EXIT_FAILURE; }  
    *retval = (s->top)->data; // top of stack  
    p = s->top; // remember for free()  
    s->top = p->next; // top should point to next  
    (s->size)--; // decrease stack size  
    free(p); // free allocated space  
    return EXIT_SUCCESS;  
}
```



# Η Δομή Δεδομένων Ουρά

(με Δυναμική Χορήγηση Μνήμης)

- Μια **ουρά (queue)** είναι μια λίστα στοιχείων που συνοδεύεται από τις διαδικασίες
  - **enqueue**, για εισαγωγή στοιχείου στη λίστα, και
  - **dequeue**, για εξαγωγή του στοιχείου που εισάχθηκε πρώτο στη λίστα.

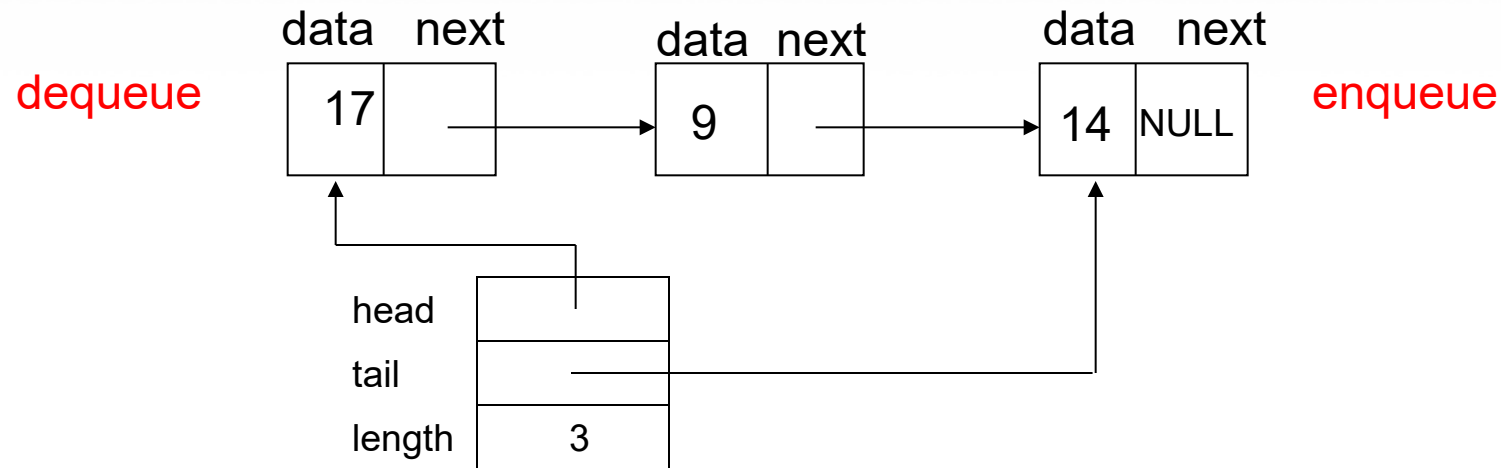


- Για ευκολία στην διεκπεραίωση των δύο διαδικασιών θα ήταν χρήσιμο, **ανά πάσα στιγμή** να γνωρίζουμε που βρίσκεται η **αρχή** και το **τέλος** της λίστας (και το μέγεθός της).
- Η νέα **εικόνα** της **ουράς** που θα θέλαμε να έχουμε βασισμένη σε συνδεδεμένη λίστα είναι η παρακάτω:



# Η Δομή Δεδομένων Ουρά

## (με Δυναμική Χορήγηση Μνήμης)



- **Παρατήρηση:** η μορφή του κόμβου είναι η ίδια και σ' αυτό το παράδειγμα με αυτή της στοίβας.
  - Άρα η δήλωση της δομής **NODE** παραμένει η ίδια.
  - Η **μορφή**, όμως του **κόμβου (QUEUE)** που κρατά πληροφορίες για την ουρά αλλάζει και παρουσιάζεται παρακάτω.



# Η Δομή Δεδομένων Ουρά

## (με Δυναμική Χορήγηση Μνήμης)

- Η ουρά λοιπόν θα αναπαρίσταται βάσει της δομής:

```
typedef struct {  
    NODE *head;  
    NODE *tail;  
    int length;  
} QUEUE;
```

### A. Στατική Δήλωση Ουράς στο main():

```
QUEUE queue;  
queue.head = queue.tail = NULL;  
queue.length = 0;
```

### B. Δυναμική Δήλωση Ουράς στο main():

```
QUEUE *queue = NULL;  
queue = (QUEUE *) malloc(sizeof(QUEUE));  
queue->head = queue->tail = NULL;  
queue->length = 0;
```

**Ισχύουν αυτά που  
αναφέραμε ήδη για  
την στοίβα!**



# Η Δομή Δεδομένων Ουρά

## (με Δυναμική Χορήγηση Μνήμης)

→ Κατά την έξοδο, η αναφορά `queue` χάνεται όχι όμως ο χώρος που δεσμεύτηκε με την `malloc` (η διεύθυνση του οποίου επιστρέφεται)

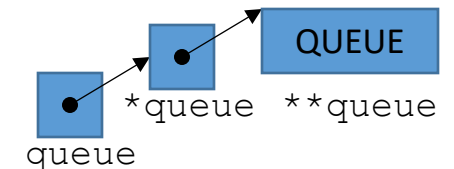
### Γ) Δυναμικά σε Συνάρτηση (με `return pointer`):

```
QUEUE *initQueue() {
    QUEUE *queue = (QUEUE *) malloc(sizeof(QUEUE));
    if (queue == NULL) return NULL;
    queue->head = queue->tail = NULL;
    queue->size = 0;
    return queue;
}
```

```
int main() {
    QUEUE *queue = NULL;
    queue = initQueue();
    ...
}
```

### Δ) Δυναμικά σε Συνάρτηση (με δείκτη-σε-δείκτη):

```
int initQueue2(QUEUE **queue) {
    *queue = (QUEUE *) malloc(sizeof(QUEUE));
    if ((*queue) == NULL) return EXIT_FAILURE;
    (*queue)->head = (*queue)->tail = NULL;
    (*queue)->size = 0;
    return EXIT_SUCCESS;
}
```



```
int main() {
    QUEUE *queue;
    initQueue2(&queue);
    ...
}
```



# Άσκηση

## Υλοποίηση Συναρτήσεων Ουράς

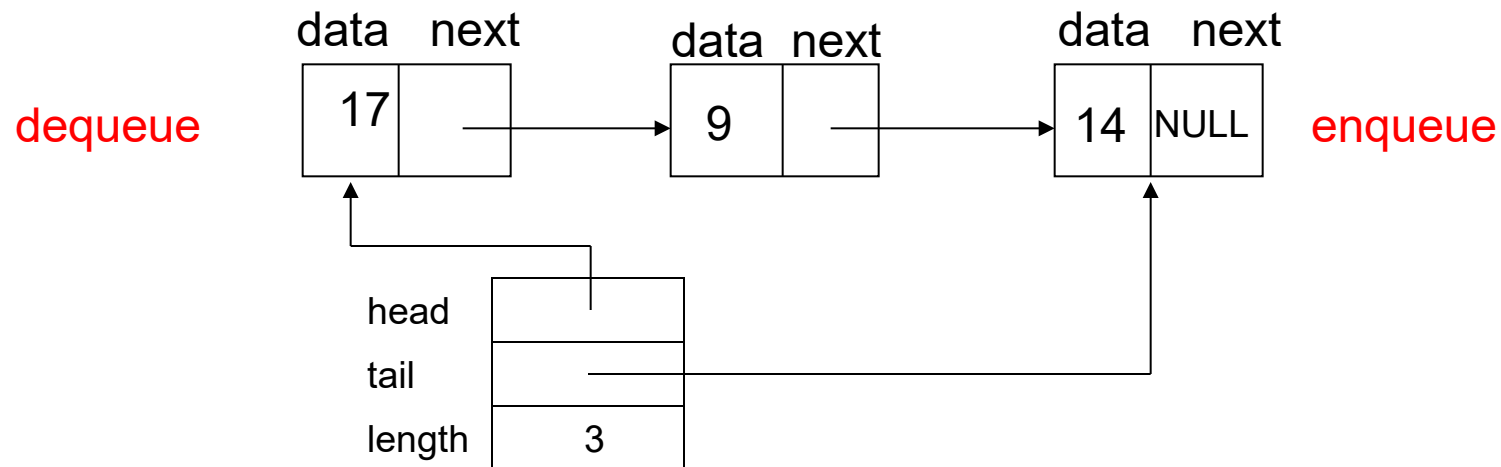
Να υλοποιήσετε τις πράξεις ουράς

`int enqueue(int value, QUEUE *q)` ,

για εισαγωγή κόμβου στο τέλος της ουράς, και

`int dequeue(QUEUE *q, int *retval)`

για διαγραφή κόμβου από την αρχή της ουράς αντίστοιχα.

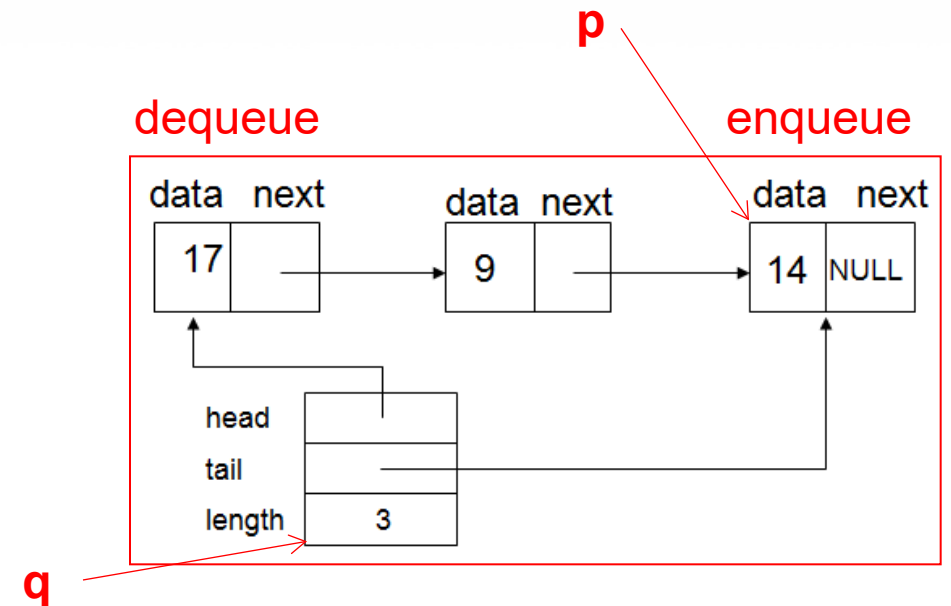




# Άσκηση

## Υλοποίηση Συναρτήσεων Ουράς

```
int enqueue (int value, QUEUE *q)  {
    NODE *p = NULL;
    if (q == NULL){ return EXIT_FAILURE; }
    p = (NODE *)malloc(sizeof(NODE));
    if ( p == NULL ) {
        printf("System out of memory...\n");
        return EXIT_FAILURE;
    }
    p->data = value;
    p->next = NULL;
    if (q->length == 0)
        q->head = q->tail = p;
    else { // append on end
        q->tail->next = p;
        q->tail = p;
    }
    (q->length)++;
    return EXIT_SUCCESS;
}
```



# Άσκηση

## Υλοποίηση Συναρτήσεων Ουράς

```
int dequeue(Queue *q, int *retval) {  
    NODE *p = NULL;          // copy pointer used for free()  
    if ((q == NULL) || (q->head == NULL)) {  
        printf("Sorry, queue is empty \n");  
        return EXIT_FAILURE;  
    }  
    if (retval == NULL) {  
        printf("RetVal is null"); return EXIT_FAILURE; }  
    p = q->head;  
    *retval = q->head->data;  
    q->head = q->head->next;  
    free(p);  
    --(q->length);  
    if (q->length == 0) {  
        q->tail = NULL;  
    }  
    return EXIT_SUCCESS;  
}
```

