

ΕΠΛ232 – Προγραμματιστικές Τεχνικές και Εργαλεία

Δομές Δεδομένων II (Ταξινομημένες Λίστες)

Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου

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



Συμβόλαιο Μαθήματος

• Αξιολόγηση

- 55% Τελική Εξέταση (1)
- 20% Ενδιάμεση Εξέταση (1)
 - Ημερομηνία: 23/10/2019
- **25% Ασκήσεις**
 - Προγραμματιστικές Ασκήσεις (3) – 5%
 - Ομαδική Εργασία (1) – 10%

Περιεχόμενο Διάλεξης 13

- Στοιβά, Ουρά και Ταξινόμηση
 - Επανάληψη Στοιβάς και Ουράς
 - Αδυναμία Υποστήριξης Ταξινόμησης
- **Ταξινομημένες Λίστες**
 - Δηλώσεις και Αρχικοποίηση (στατικά, δυναμικά, σε συνάρτηση, δείκτη-δείκτη)
 - Διαδικασίες `printList()`, `printListRecursive()`
 - Διαδικασίες `insert()`, `insertRecursive()`
 - Διαδικασίες `delete()`, `deleteRecursive()`

Επόμενη
Διάλεξη



Διασυνδεδεμένες δομές δεδομένων

- Η μνήμη ενός πίνακα δεσμεύεται συνεχόμενα.
 - Η πρόσβαση στο i-οστό στοιχείο είναι **άμεση** καθώς η διεύθυνση του είναι γνωστή **ΕΚ ΤΩΝ ΠΡΟΤΕΡΩΝ** και μπορεί να υπολογιστεί από τον μεταφραστή.
- Μπορούμε να κατασκευάσουμε δομές τα στοιχεία των οποίων βρίσκονται σε διαφορετικές θέσεις στη μνήμη, και τα οποία συνδέονται μεταξύ τους μέσω **δεικτών**.
 - Η θέση του «i-οστού» στοιχείου δεν είναι γνωστή και πρέπει να εντοπιστεί, διανύοντας την δομή κατά μήκος των συνδέσεων μεταξύ των κόμβων της.
 - **Σημείωση**: ακριβώς για αυτό το λόγο, σε αυτές τις δομές δεν υφίσταται η έννοια «i-οστό» στοιχείο.



Διασυνδεδεμένες δομές δεδομένων

- Οι διασυνδεδεμένες δομές δεν κατασκευάζονται μονομιάς αλλά **σταδιακά**, κόμβο προς κόμβο.
 - Κάθε κόμβος που προστίθεται, πρέπει να **συνδεθεί** κατάλληλα με τους ήδη υπάρχοντες κόμβους, ώστε να είναι δυνατή η προσπέλαση όλων των κόμβων.
 - Για κάθε κόμβο που απομακρύνεται, πρέπει να **προσαρμοστούν** οι διασυνδέσεις των υπολοίπων.
- Κάθε διασυνδεδεμένη δομή έχει και ένα συγκεκριμένο (διαφορετικό) τρόπο προσπέλασης, σύμφωνα με τον οποίο:
 - Ορίζονται οι δείκτες διασύνδεσης των κόμβων και
 - Υλοποιούνται οι διάφορες λειτουργίες αναζήτησης, προσθήκης και απομάκρυνσης.

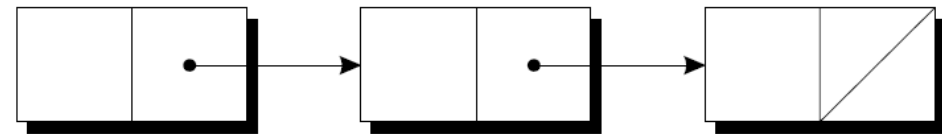


Αυτοαναφορικές δομές δεδομένων

- Μια από τις κλασικές μορφές διασυνδεδεμένων δομών είναι οι **αυτοαναφορικές** δομές (ή αναδρομικές δομές).
 - Μια δομή ονομάζεται αυτοαναφορική όταν ένα από τα πεδία της είναι ένας δείκτης σε δομή ιδίου τύπου.
 - Οι αυτοαναφορικές δομές κατασκευάζονται μέσα από την επανάληψη του ίδιου **δομικού στοιχείου (κόμβου)**.

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

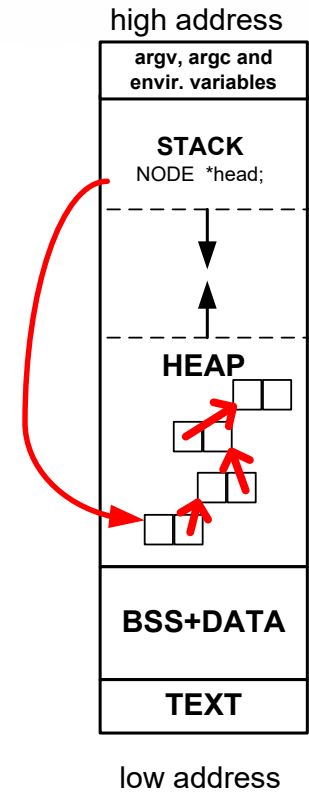
ΣΥΝΙΣΤΟΜΕΝΟΣ ΟΡΙΣΜΟΣ



Στοίβα, Ουρά και Ταξινόμηση

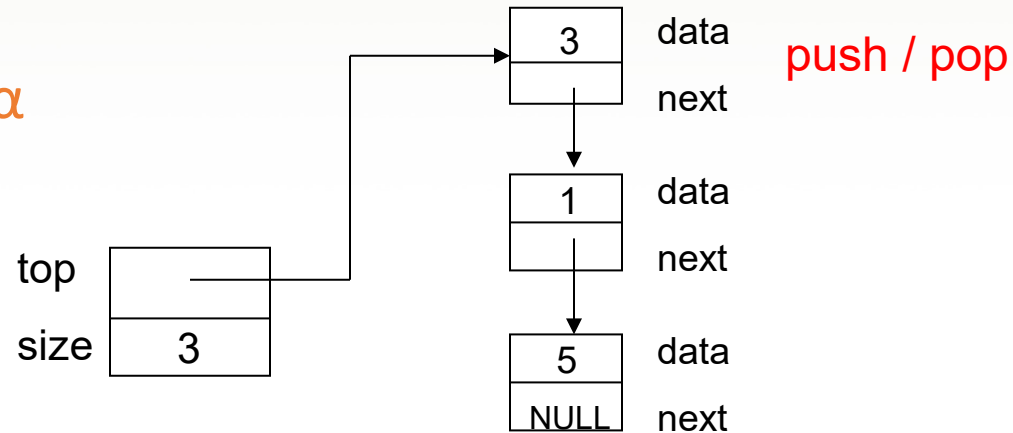
- Στις δομές δεδομένων **ουρά** και **στοίβα** είδαμε πως η θέση των στοιχείων εξαρτάται από την σειρά εισδοχής τους.
 - σε μια **Στοίβα** το στοιχείο **στην κορυφή** είναι πάντα αυτό που **εισήχθη τελευταίο**
 - σε μια **Ουρά**, αυτό που **εισήχθη πρώτο**.
- Έτσι αν εφαρμόζαμε διαδοχικά τις εντολές:
 - εισαγωγή του **5**,
 - εισαγωγή του **1**
 - εισαγωγή του **3**
 - θα είχαμε . . .

ΔΙΕΡΓΑΣΙΑ

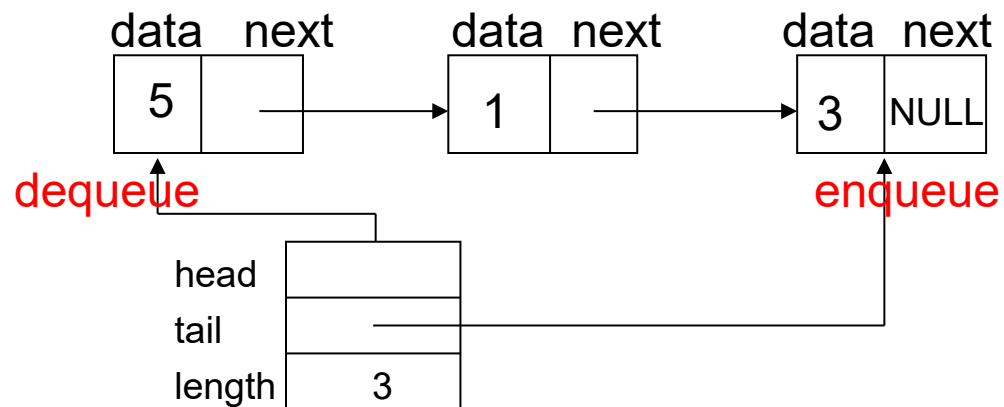


Στοίβα, Ουρά και Ταξινόμηση

- Σε Στοίβα



- Σε Ουρά



Εφαρμόζοντας **τρεις** φορές τις διαδικασίες **pop** και **dequeue** στη **στοίβα** και την **ουρά**, αντίστοιχα, θα επιστρέφονταν

- από τη **Στοίβα** τα στοιχεία 3, 1, 5 (με αυτή τη σειρά), και
- από την **Ουρά** τα στοιχεία 5, 1, 3 (με αυτή τη σειρά).



Η Δομή Δεδομένων Στοίβα

(με Δυναμική Χορήγηση Μνήμης)

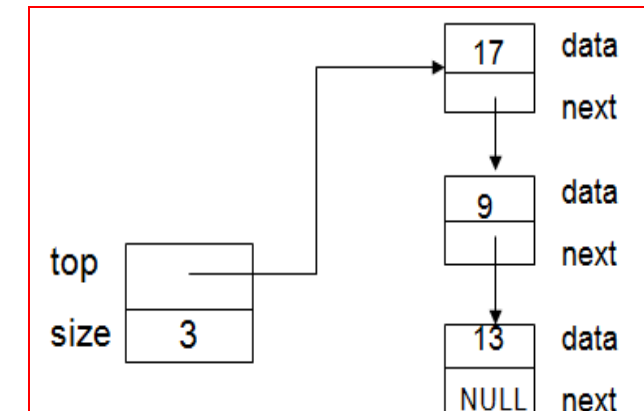
```

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

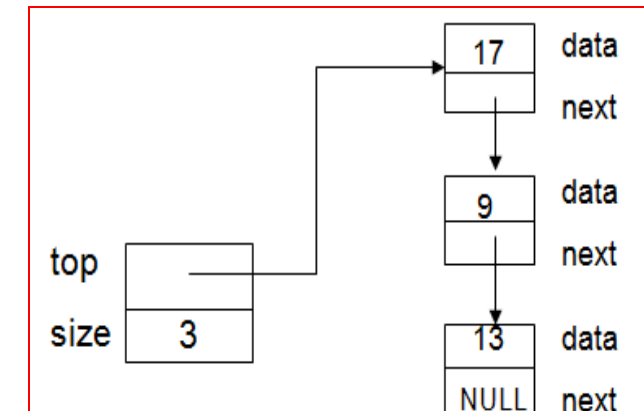
(με Δυναμική Χορήγηση Μνήμης)

```

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;
}

```



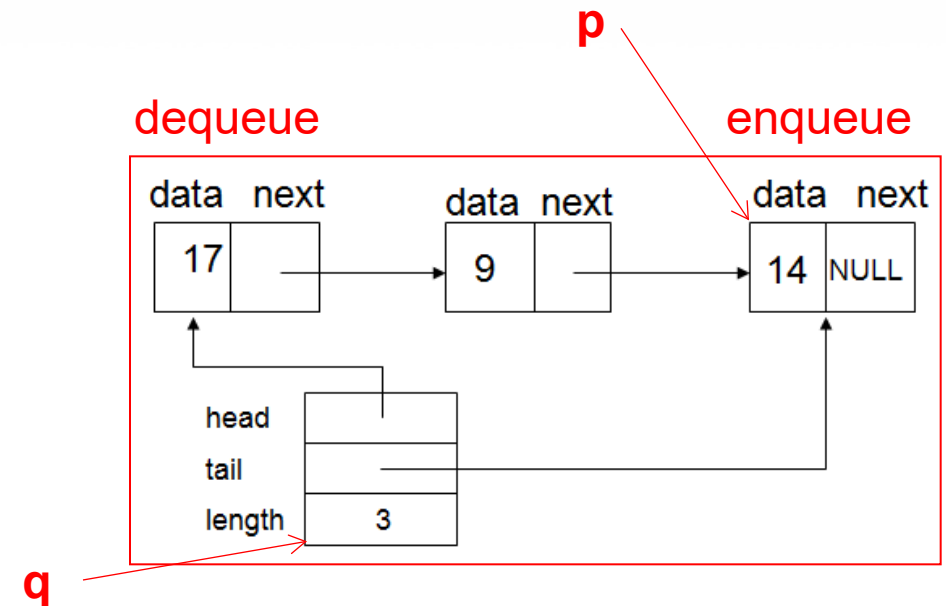
Άσκηση

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

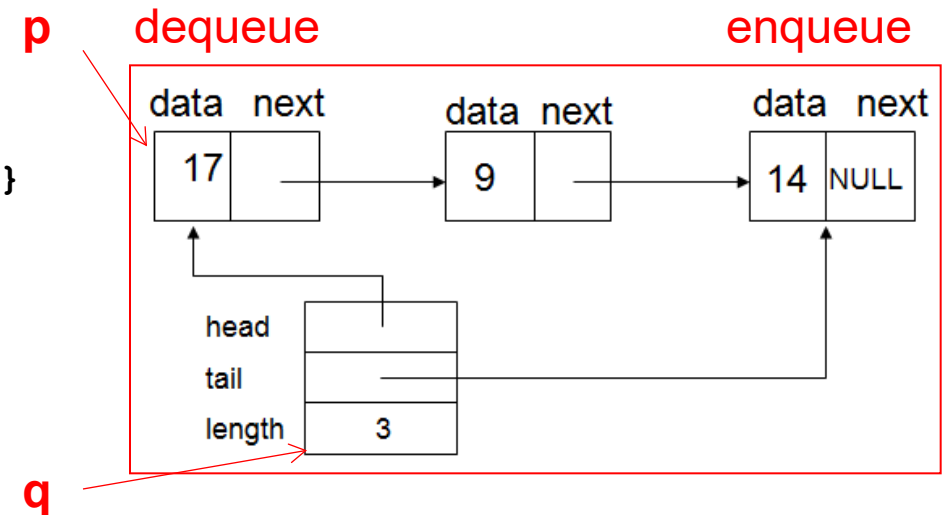
```

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);

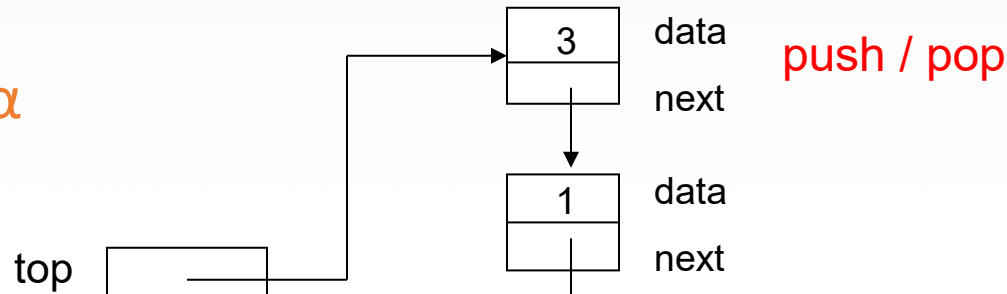
    if (q->length == 0) {
        q->tail = NULL;
    }
    return EXIT_SUCCESS;
}

```



Στοίβα, Ουρά και Ταξινόμηση

- Σε Στοίβα

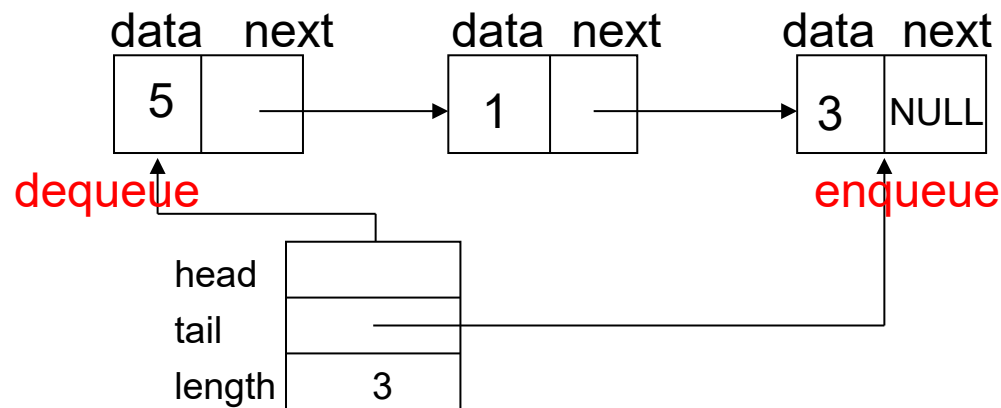


Εφαρμόζοντας **τρεις** φορές τις διαδικασίες **pop** και **dequeue** στη **στοίβα** και την **ουρά**, αντίστοιχα, θα επιστρέφονταν

Συνεπώς, οι δομές αυτές δεν μπορούν να χρησιμοποιηθούν για να έχουμε πρόσβαση σε διατεταγμένα, βάσει κάποιας συνθήκης, δεδομένα. ☹️

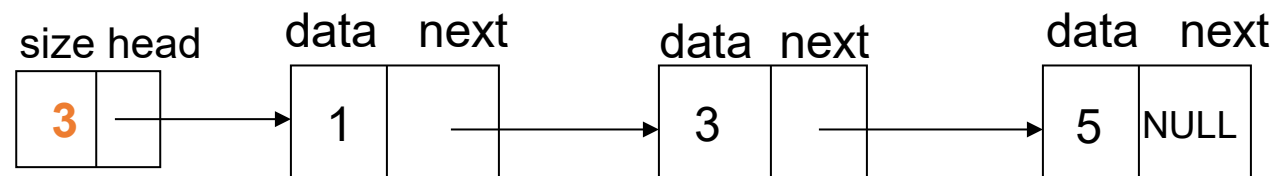
α 3, 1, 5

α 5, 1, 3



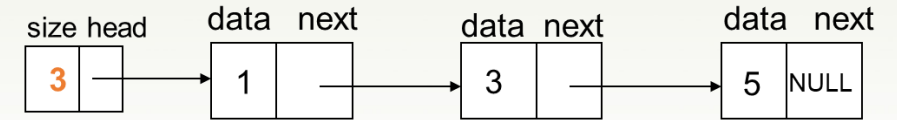
Ταξινομημένες Λίστες

- **Ταξινομημένα τα Δεδομένα σε κάποια σειρά** (π.χ., αύξουσα στα ακόλουθα παραδείγματα) και από την οποία να μπορούμε εύκολα να προσθέτουμε και να αφαιρούμε στοιχεία.
- Έτσι, για παράδειγμα αν εφαρμόζαμε διαδοχικά τις εντολές: **εισαγωγή του 5, εισαγωγή του 1 και εισαγωγή του 3** θα θέλαμε η λίστα μας να είχε τη μορφή:



Ταξινομημένες Λίστες

Δηλώσεις και Αρχικοποίηση



- Για την υλοποίηση μιας δομής τύπου **Ταξινομημένη Λίστα** (όπως και στις στοίβες και ουρές) απαιτείται η παρακάτω δήλωση κόμβων:

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

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

A) Στατικά στο main()

```
LIST list;  
list.head = NULL;  
list.size = 0;  
foo(&list);  
  
void foo(LIST *list);
```

B) Δυναμικά στο main()

```
LIST *list = NULL;  
list = (LIST *)  
        malloc(sizeof(LIST));  
list->head = NULL;  
list->size = 0;  
foo(list);
```



Ταξινομημένες Λίστες

Δηλώσεις και Αρχικοποίηση

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

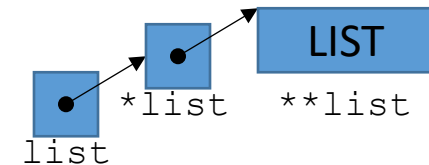
Γ) Δυναμικά σε Συνάρτηση (με `return pointer`):

```
LIST *initList() {  
    LIST *list = (LIST *) malloc(sizeof(LIST));  
    if (list == NULL) return NULL;  
    list->head = NULL;  
    list->size = 0;  
    return list;  
}
```

```
int main() {  
    LIST *list = initList();  
    ...  
}
```

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

```
int initList2(LIST **list) {  
    *list = (LIST *) malloc(sizeof(LIST));  
    if ((*list) == NULL) return EXIT_FAILURE;  
    (*list)->head = NULL;  
    (*list)->size = 0;  
    return EXIT_SUCCESS;  
}
```



```
int main() {  
    LIST *list = NULL;  
    initList2(&list);  
    ...  
}
```



Ταξινομημένες Λίστες

Λειτουργίες / Διαδικασίες

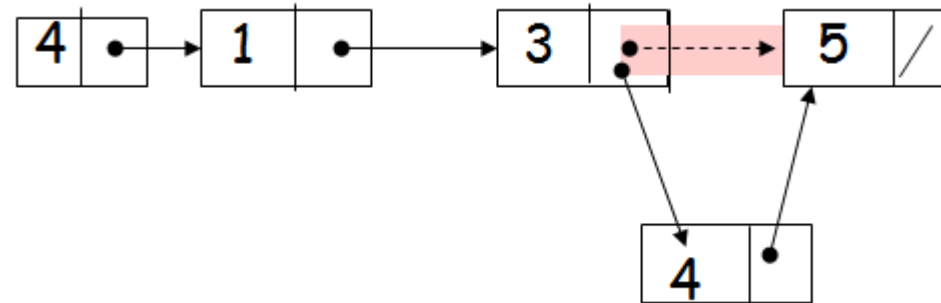
- Τώρα απομένει να ορίσουμε τρεις διαδικασίες:
 - **printlist**: τυπώνει όλα τα στοιχεία της λίστας l.
`void printlist (LIST *l);`
 - **insert**: εισαγάγει ένα στοιχείο x μέσα στη λίστα l.
`void insert (LIST *l, int x);`
 - **delete**: αφαιρεί κάποιο στοιχείο x (αν υπάρχει) από τη λίστα l.
`void delete (LIST *l, int x);`
- Οι πιο πάνω λειτουργίες μαζί με τις συνοδευτικές λειτουργίες δημιουργίας, καταστροφής της λίστας πρέπει να τοποθετηθούν σε αρχεία `list.c` και `list.h`
 - Μαζί με τον σχετικό οδηγό δοκιμής **#ifdef DEBUG ... #endif**.
 - Το `main` θα περιέχει ένα `switch` που μας επιτρέπει να καλέσουμε τις διαδικασίες ταξινομημένης λίστας.
 - **Δεν θα μας απασχολήσει στο παρόν στάδιο το return code των συναρτήσεων αυτών.**



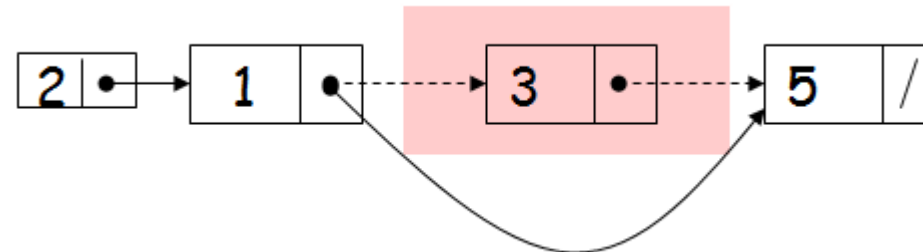
Ταξινομημένες Λίστες

Διαγραμματική Απεικόνιση

- **Εισαγωγή** του 4 στη λίστα:



- **Εξαγωγή** του 3 από τη λίστα:



Τι οριακές περιπτώσεις εντοπίζετε;



Διαγραφή κόμβου σε ταξινομημένα συνδεδεμένη λίστα

- Η διαγραφή ενός κόμβου περιλαμβάνει τρία βήματα:
 1. Εντοπισμός του κόμβου που θα διαγραφεί.
 2. Τροποποίηση του προηγούμενου κόμβου, ώστε να "παρακάμπτει" τον διαγραμμένο κόμβο.
 3. Κλήση της `free` για να ανακτήσετε το χώρο που καταλαμβάνει ο διαγραμμένος κόμβος.
- Λύση παρόμοια με εκείνη της απλά συνδεδεμένης λίστας
 - Τροποποίηση όσο αφορά τον τερματισμό!



Διαγραφή κόμβου σε ταξινομημένα συνδεδεμένη λίστα

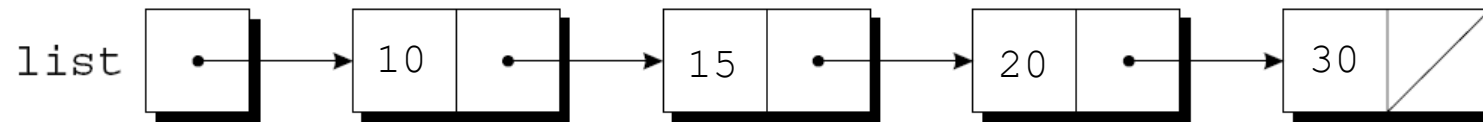
- Η τεχνική "τελικού δείκτη" περιλαμβάνει τη διατήρηση ενός δείκτη στον προηγούμενο κόμβο (`prev`) καθώς και ένα δείκτη στον τρέχοντα κόμβο (`cur`).
- Ας υποθέσουμε ότι η `list` δείχνει στη λίστα προς αναζήτηση και `n` είναι ο ακέραιος που θα διαγραφεί (**η λίστα είναι ταξινομημένη σε αύξουσα σειρά**).
- Ένας βρόγχος που υλοποιεί το βήμα 1:

```
for (cur = list, prev = NULL;  
    cur != NULL && cur->value < n;  
    prev = cur, cur = cur->next) ;
```
- Όταν ο βρόγχος τερματίζεται, η `cur` δείχνει στον κόμβο που θα διαγραφεί και η `prev` δείχνει στον προηγούμενο κόμβο.

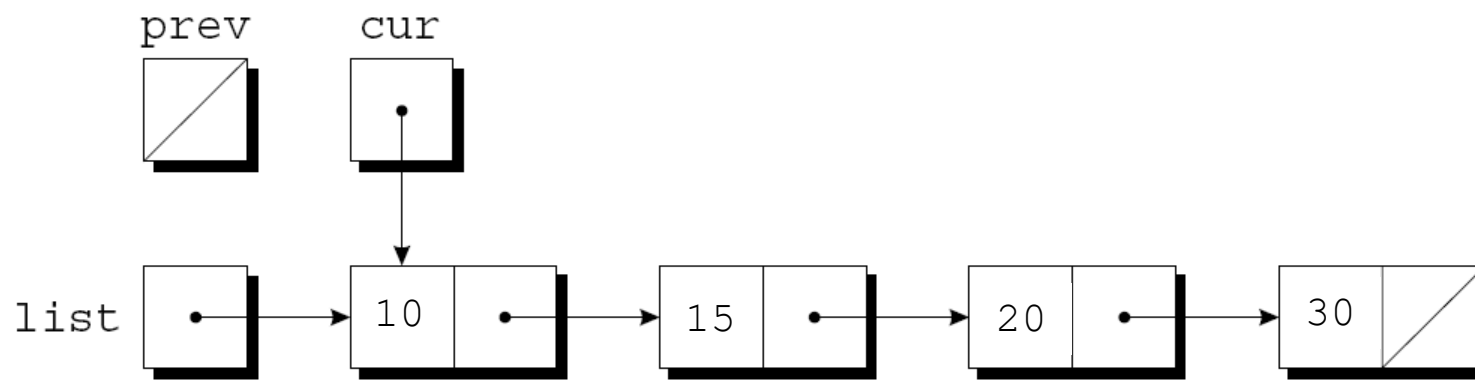


Διαγραφή κόμβου σε ταξινομημένα συνδεδεμένη λίστα

- Ας υποθέσουμε ότι η `list` έχει την ακόλουθη εμφάνιση και `n` είναι 20:

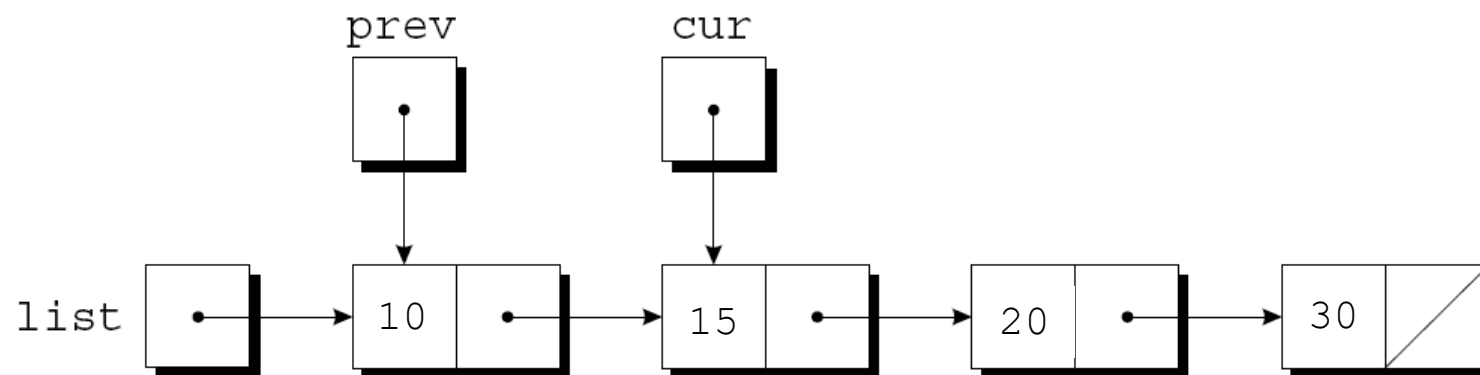


- Ακολούθως, η `cur = list`, και `prev = NULL` έχουν εκτελεστεί:



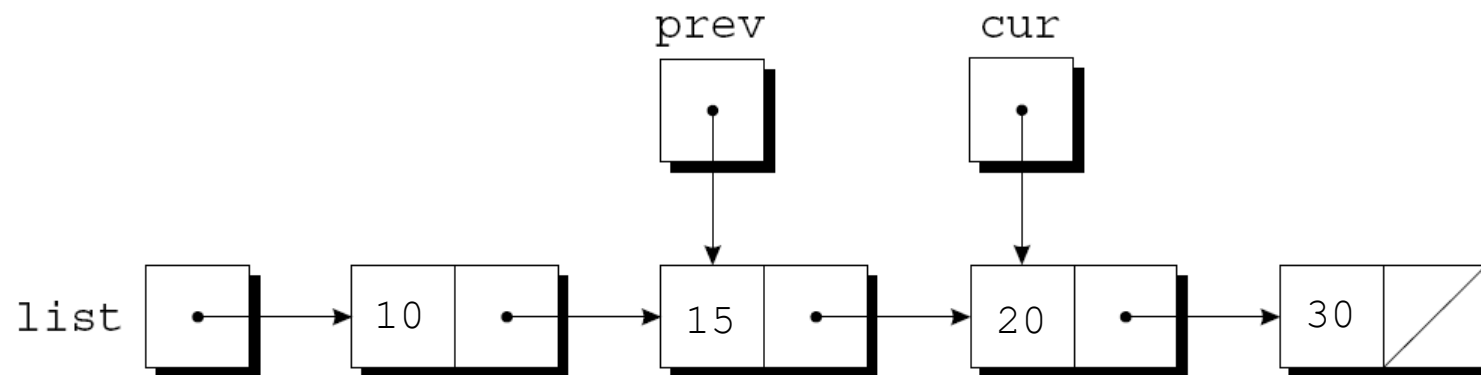
Διαγραφή κόμβου σε ταξινομημένα συνδεδεμένη λίστα

- Ο έλεγχος `cur != NULL && cur->value < n` είναι ορθός, αφού η `cur` δείχνει σε έναν κόμβο, και ο κόμβος είναι μικρότερος από το 20.
- Ακολούθως, η `prev = cur`, `cur = cur->next` έχουν εκτελεστεί:



Διαγραφή κόμβου σε ταξινομημένα συνδεδεμένη λίστα

- Ο έλεγχος `cur != NULL && cur->value < n` είναι πάλι σωστός, οπότε η `prev = cur, cur = cur->next` εκτελείται ξανά:



- Αφού το `cur` τώρα δείχνει στον κόμβο ο έλεγχος `cur->value < n` είναι λάθος και ο βρόγχος τερματίζει. Ακολούθως θα υπάρξει ένας επιπλέον έλεγχος `cur->value != n`, αν ο έλεγχος είναι λάθος (άρα το `value` του node είναι ίσο με 20) προχωράμε στην παράκαμψη του node (επόμενη σελίδα), αλλιώς έχουμε έξοδο χωρίς διαγραφή.

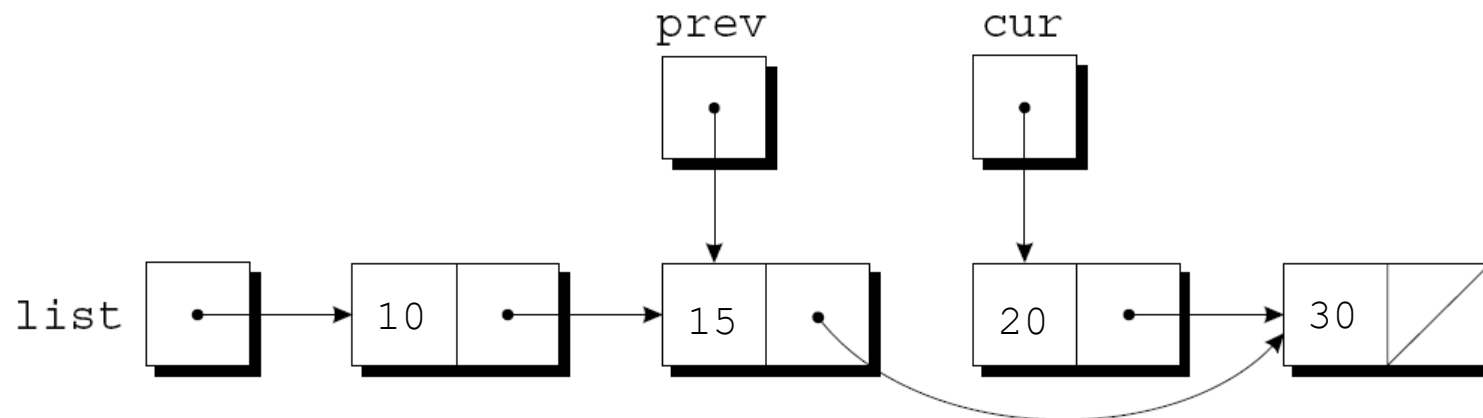


Διαγραφή κόμβου σε ταξινομημένα συνδεδεμένη λίστα

- Η παράκαμψη που απαιτείται από το βήμα 2.
- Η δήλωση

```
prev->next = cur->next;
```

κάνει το δείκτη στο προηγούμενο κόμβο να δείχνει στον κόμβο που δείχνει η τρέχουσα τιμή του κόμβου:



Διαγραφή κόμβου σε ταξινομημένα συνδεδεμένη λίστα

- Το Βήμα 3 είναι να απελευθερώσει τη μνήμη που καταλαμβάνει ο τρέχων κόμβος:

```
free (cur) ;
```

Ταξινομημένες Λίστες

Συνάρτηση Οδηγού Χρήσης

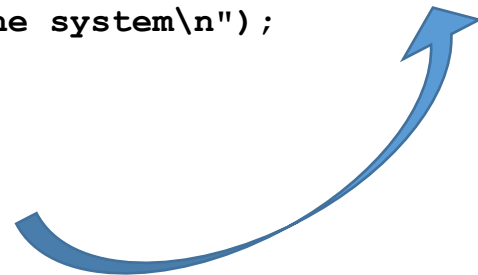
```
#ifdef DEBUG
int main(){
    int x = 0;          /* User Function Choice*/
    int y; /* Value to be inserted/deleted */
    LIST *list = initList();

    while (x != 4){

        printf("You have the following options:\n");
        printf("\t 1\t: inserting an item in the list\n");
        printf("\t 2\t: deleting an item from the list\n");
        printf("\t 3\t: printing the list elements\n");
        printf("\t 4\t: exiting from the system\n");
        scanf("%d", &x);

        // συνέχεια
```

```
switch (x){
    case 1: printf("Enter number to be inserted: ");
            scanf("%d", &y);
            insert(list, y);
            break;
    case 2: printf("Give me the number to be deleted: ");
            scanf("%d", &y);
            delete(list, y);
            break;
    case 3: printlist(list);
            break;
    case 4: printf("Goodbye!\n");
            break;
    default: printf("Wrong entry - try again!\n");
            break;
}
}
return 0;
}
#endif
```



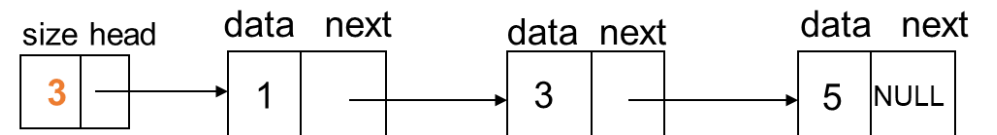
Ταξινομημένες Λίστες

Συνάρτηση `printlist`

```
void printlist(LIST *l) {
    NODE *p = NULL; // copy pointer

    if ((l == NULL) || (l->size == 0)) {
        printf("The list is empty\n");
        return;
    }

    p = l->head;
    while (p != NULL) {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}
```

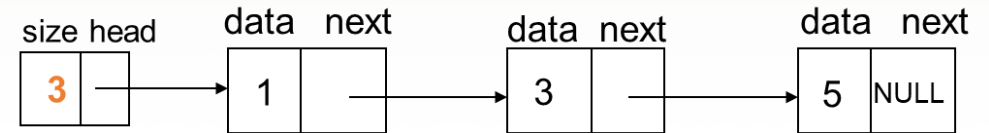


Ταξινομημένες Λίστες

Αναδρομική Συνάρτηση `printlist`

```
void printlist(LIST *l){  
    if ((l == NULL) || (l->size == 0))  
        printf("The list is empty\n");  
    else  
        printnode(l->head);  
}
```

```
static void printnode(NODE *p){  
    if (p != NULL){  
        printf("%d", p->data);  
        printnode(p->next);  
    }  
}
```



→ "Εσωτερική συνάρτηση" με εμβέλεια αρχείου

Αναδρομή Χωρίς return
στην οπισθοχώρηση



Πρόβλημα 1: `count`

- Γράψετε συνάρτηση στη γλώσσα C η οποία παίρνει ως όρισμα ένα δείκτη σε λίστα και η οποία επιστρέφει το μέγεθος της λίστας.

- **Πρότυπο Συνάρτησης:**

```
int count(LIST *list);
```

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

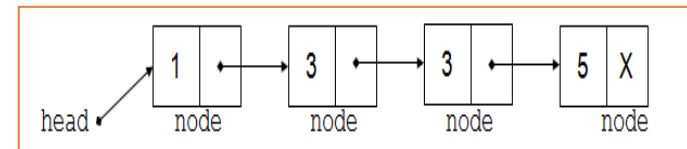
```
typedef struct {  
    NODE *head;  
} LIST;
```

- **Κλήση Συνάρτησης**

```
LIST *list;
```

```
...
```

```
printf("%d", count(list)); ➔ τυπώνει 4
```

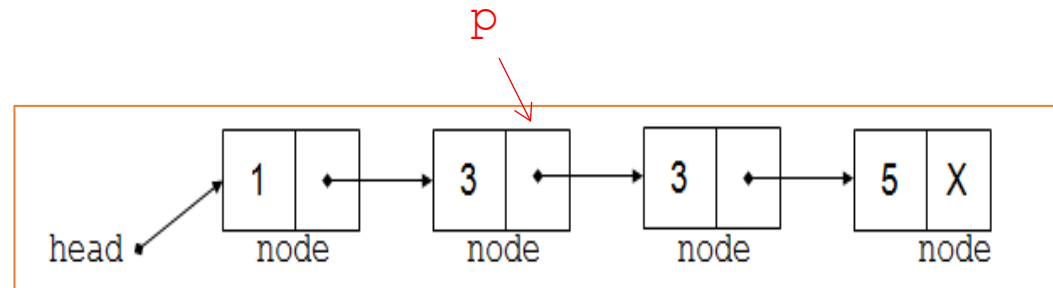


Πρόβλημα 1: `count`

Επαναληπτική Λύση

```
int count(LIST *l) {  
    if (l == NULL) return 0;  
    return countList(l->head);  
}
```

```
static int countList(NODE *p) {  
    int count = 0;  
    while (p != NULL) {  
        count++;  
        p = p->next;  
    }  
    return count;  
}
```

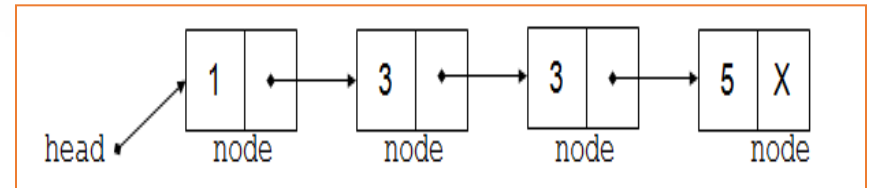


Πρόβλημα 1: `count`

Αναδρομική Λύση 1

```
int count(LIST *l) {  
    if (l == NULL) return 0;  
    int count = 0;  
    countListRecursive(l->head, &count);  
    return count;  
}
```

```
static void countListRecursive(NODE *p, int *count) {  
    if (p == NULL) {  
        return 0;  
    }  
    (*count)++;  
    countListRecursive(p->next, count);  
}
```



**Αναδρομή ΧΩΡΙΣ return
στην οπισθοχώρηση και
τιμή δια αναφοράς**

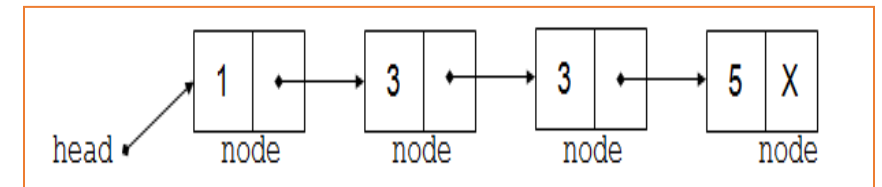


Πρόβλημα 1: `count`

Αναδρομική Λύση 2

```
static int countListRecursive (NODE *p) {  
    if (p == NULL) {  
        return 0;  
    }  
    return 1 + countListRecursive (p->next);  
}
```

```
int count (LIST *l) {  
    if (l == NULL) return 0;  
    return countListRecursive (l->head);  
}
```



Αναδρομή ME return
στην οπισθοχώρηση

