Τμήμα Πληροφορικής
Πανεπιστήμιο Κύπρου

ΕΠΛ232 – Προγραμματιστικές Τεχνικές και Εργαλεία

Διάλεξη 14: Δομές Δεδομένων III (Λίστες και Παραδείγματα)

Δημήτρης Ζεϊναλιπούρ

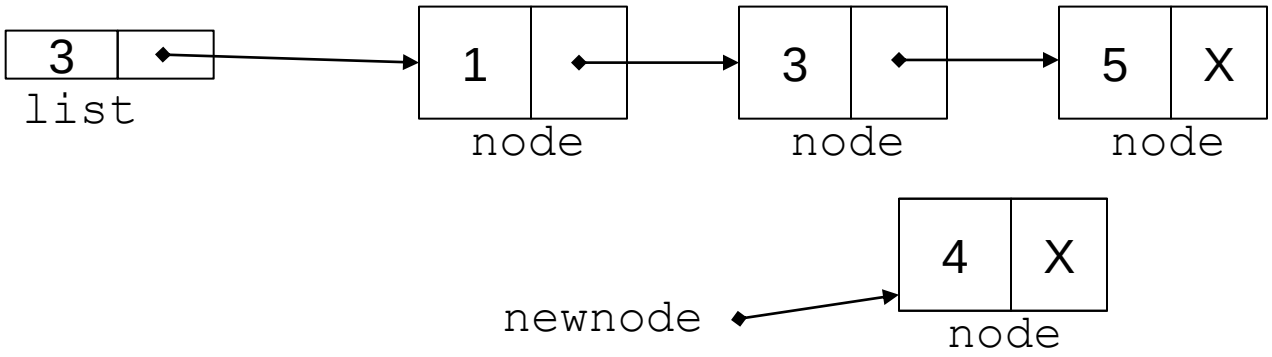
http://www.cs.ucy.ac.cy/courses/EPL232

Περιεχόμενο Διάλεξης 14

- **Υλοποίηση Insert/Delete σε Λίστα**
  - Επαναληπτική & Αναδρομική Λύση
- **Προβλήματα Λιστών**
  1. Απάλειψη Διπλοτύπων από Λίστα – `eliminateDuplicates(list)`
     - Επαναληπτική & Αναδρομική Λύση
  2. Συγχώνευση Ταξινομημένων Ακολουθιών – `mergeLists(list1, list2, list3)`
     - Επαναληπτική & Αναδρομική Λύση

Ταξινομημένες Λίστες (Συνάρτηση `insert`)

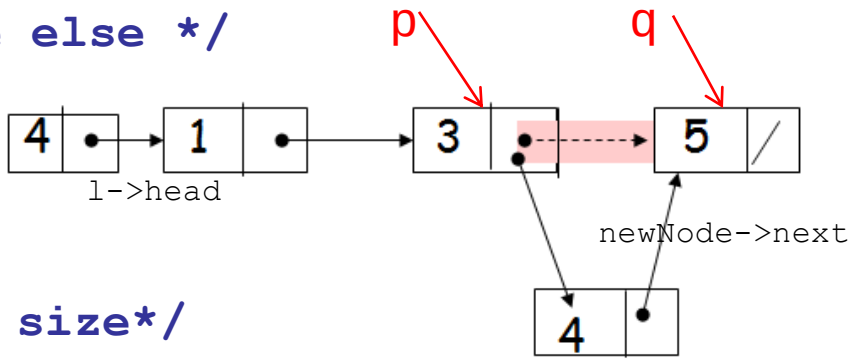
```
void insert(LIST *l, int x){
  NODE  *p = NULL, *q = NULL; /* prev, curr copy pointers*/

  if (l == NULL) { printf("Unable to Enter"); return; }
  /* Create and Fill New Node */
  NODE * newNode = (NODE *)malloc(sizeof(NODE));
   if (newNode == NULL) {
        printf("Error: Unable to Allocate Memory!");
       return;
  }
   newNode->data = x;
   newNode->next = NULL;
```



συνέχεια...

Ταξινομημένες Λίστες (Συνάρτηση `insert`)

```
/* 2) Connect newNode to List */
if ( l->head == NULL ) /* If Empty add to l->head */
  l->head = newNode;
else {
  p = q = l->head; /* Fast forward p and q until data<x*/
  while ( ( q != NULL ) && ( q->data < x ) ) {
       p = q;
       q = q->next;
  }
  if ( p == q ) {   /* i.e., Insert to first position */
       newNode->next = l->head; // e.g., insert "0"
       l->head = newNode; // Branch required for this!
  } else { /* Insert anywhere else */
       newNode->next = q;
       p->next = newNode;
  }
 } // else()
(l->size)++; /* Increase List size*/
} // insert()
```

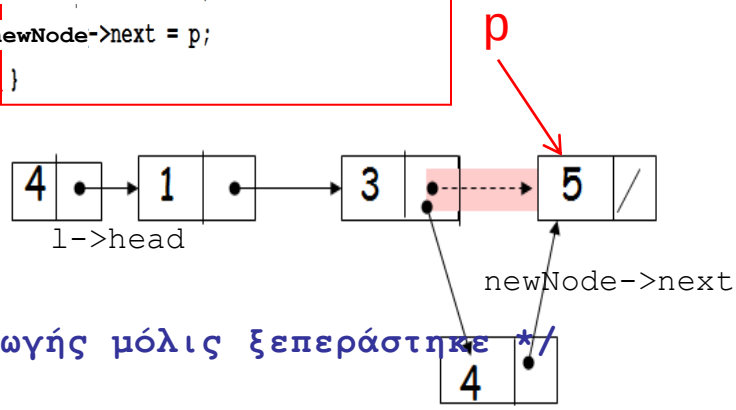


Ταξινομημένες Λίστες (Αναδρομική Συνάρτηση insert)

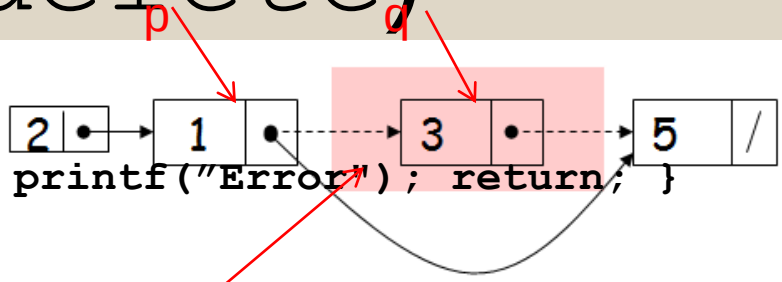
Αναδρομή ΜΕ return στην οπισθοχώρηση

```
void insert(LIST *l, int x){
      ++(l->size); // η προκαταβολική αύξηση θα βελτιωθεί σε λίγο
      l->head = insertnode(l->head, x);
}

static NODE *insertnode(NODE *p, int x){
  NODE   *newNode = NULL;
  if (p == NULL){ /* Η Λίστα είναι άδεια */
        newNode = (NODE *)malloc(sizeof(NODE));
        newNode->data = x;
        newNode->next = NULL;
  }
  else if ( x <= p->data ) { /* Το Σημείο Εισαγωγής μόλις ξεπεράστηκε */
        newNode = (NODE *)malloc(sizeof(NODE));
        newNode->data = x;
        newNode->next = p;
  }
  else { /* Το σημείο εισαγωγής είναι πιο κάτω … συνεχίζει η αναδρομή */
        p->next = insertnode(p->next, x); /* δέσιμο τιμής επιστροφής */
        newNode = p; // ανάθεση p στο newNode για το return.
  }
  return newNode;
}
```

```
if ((p == NULL)|| ( p->data >= x )) {
newNode= (NODE *)malloc(sizeof(NODE));
newNode->data = x;
newNode->next = p;
}
```



Ταξινομημένες Λίστες (Συνάρτηση delete)



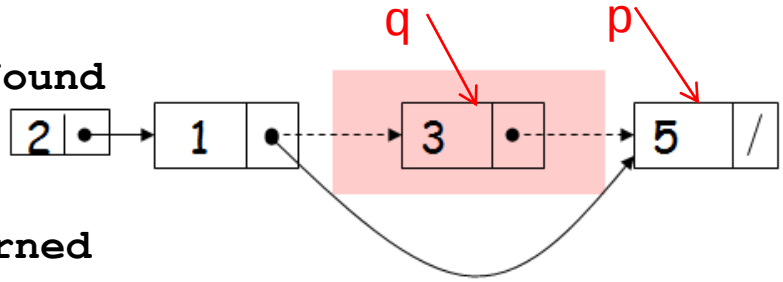

```
void delete(LIST *l, int x){
  NODE *p = NULL, *q = NULL;
  if ((l==NULL) || (l->head==NULL)) { printf("Error"); return; }
  p = q = l->head;
  // Fast Forward (ffwd) list
   while ( ( q != NULL ) && ( q->data < x ) ) {
        p = q;          // p: prev
        q = q->next; // q: next
   }  // end of list      // just passed the possible position for x
   if ( ( q == NULL ) || ( q->data > x ) )
        printf("Item %d not found \n", x);
        return;
  }
   if (p == q) // Item to be deleted is in 1st position
      l->head = l->head->next; // initially p, q point to 1st
   else // ffwd toke us somewhere else in the list.
      p->next = q->next; // fix pointer of prev
   free( q );                  // deallocate curr pointer
   (l->size)--;              // adjust list size
}
```

Ταξινομημένες Λίστες (Αναδρομική Συνάρτηση `delete`)

```
void delete(LIST *l, int x){
    if ((l==NULL) || (l->head==NULL)) { printf("Error"); return; }
    l->head = deletenode(l->head, x, &(l->size));
}
static NODE *deletenode(NODE *p, int x, int *size){
  if ( ( p == NULL) || (p->data > x) ) {
        printf("Item %d not found \n", x);
  }
  else if ( p->data == x ){//item found
        NODE *q = NULL;
        q = p;
        p = p->next; // to be returned
        free(q);
        (*size)--;
  } else {
        p->next = deletenode(p->next, x, size);
  }
  return p;
}
```

Καλύτερο από προσέγγιση με το `insert`

Surpassed possible position of x or reached end of list



Αναδρομή ΜΕ return στην οπισθοχώρηση και τιμή δια αναφοράς

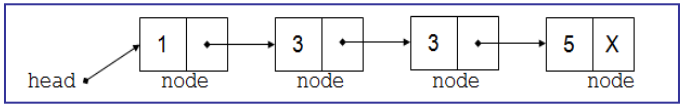
Πρόβλημα 1: `eliminateDuplicates`

- Γράψετε συνάρτηση στη γλώσσα C η οποία παίρνει ως όρισμα ένα **δείκτη σε ταξινομημένη λίστα** και η οποία **απαλείφει** τα όποια **διπλότυπα** ενδέχεται να υπάρχουν.
- **Πρότυπο Συνάρτησης:**

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

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

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



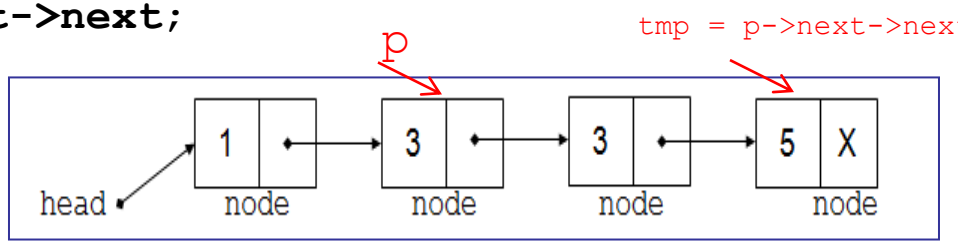
- **Κλήση Συνάρτησης**

```
LIST *list; initList2(&list); fillList(list);
eliminateDuplicates(list); printlist(list);
```

➔ **τυπώνει 1, 3, 5**

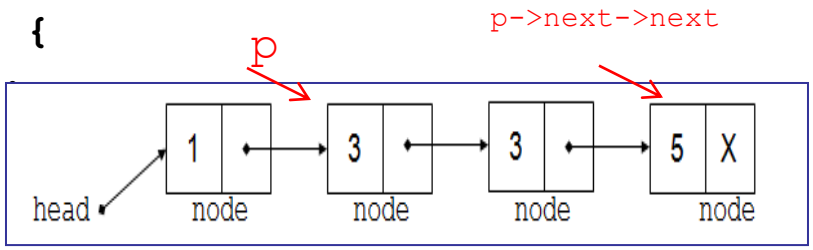
Πρόβλημα 1: `eliminateDuplicates` (Επαναληπτική Λυση)

```
static void eliminateDuplicateList(NODE *p, int *size) {
    if (p == NULL)
        return;         // do nothing if the list is empty

    while(p->next!=NULL) {
        if (p->data == p->next->data) { // next is a duplicate
            NODE *tmp = p->next->next;
            free(p->next);
            p->next = tmp;
            (*size)--;
        }
        else
            p = p->next;
    }
}
int eliminateDuplicates(LIST *l) {
    if (l == NULL) return EXIT_FAILURE;
    eliminateDuplicateList(l->head, &(l->size));
    return EXIT_SUCCESS;
}
```



Πρόβλημα 2: `eliminateDuplicates` (Αναδρομική Λύση)

```
static void eliminateDuplicateRecursive(NODE *p, int *size) {
    if (p == NULL) return;    // do nothing if the list is empty
    if (p->next!=NULL) {
        if (p->data == p->next->data) {
            NODE *tmp = p->next->next;
            free(p->next);
            p->next = tmp;
            (*size)--;
        }
        else
            p = p->next;
        eliminateDuplicateRecursive(p, size);
    }
}
int eliminateDuplicates(LIST *l) {
    if (l == NULL) return EXIT_FAILURE;
    eliminateDuplicateList(l->head, &(l->size));
    return EXIT_SUCCESS;
}
```



Κατά τα άλλα πολύ όμοια με την επαναληπτική έκδοση

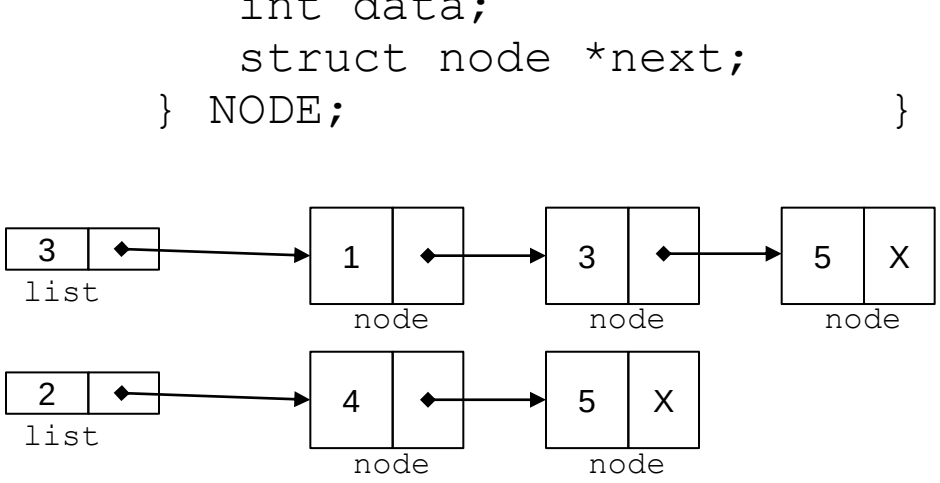
Πρόβλημα 2: `mergeLists`

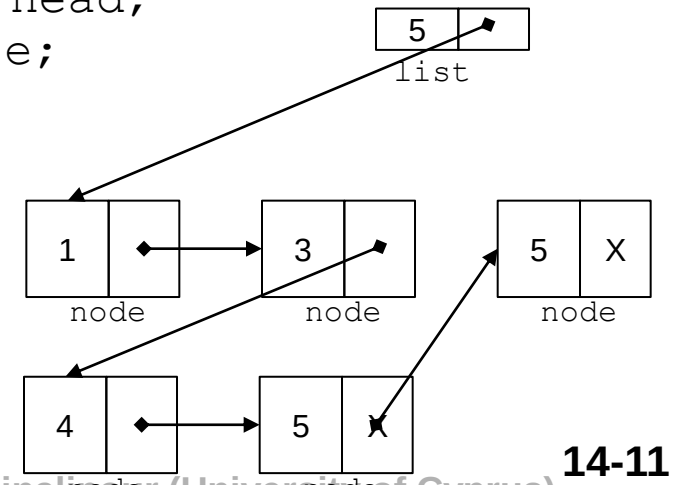
- Γράψετε συνάρτηση στη C η οποία παίρνει ως όρισμα δυο δείκτες σε **ταξινομημένες λίστες** και η οποία **συγχωνεύει (ταξινομημένα)** τις **δυο λίστες** χωρίς δημιουργία ενδιάμεσης λίστας.
- **Πρότυπο Συνάρτησης:**

```
int mergeLists(LIST *l1, LIST *l2, LIST *newl);
```

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

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





Πρόβλημα 2: `mergeLists` (Διάγραμμα Λύσης)

- Η Συνάρτηση `mergeLists()`

```
int mergeLists(LIST *l1, LIST *l2, LIST *newl) {
    if ((l1 == NULL) || (l2 == NULL) || (newl == NULL)) return EXIT_FAILURE;
    newl->head = mergeLoop(l1->head, l2->head);
    newl->size = l1->size + l2->size;
    return EXIT_SUCCESS;
}
```

- Η Κλήση `mergeLists()` από το `main()`

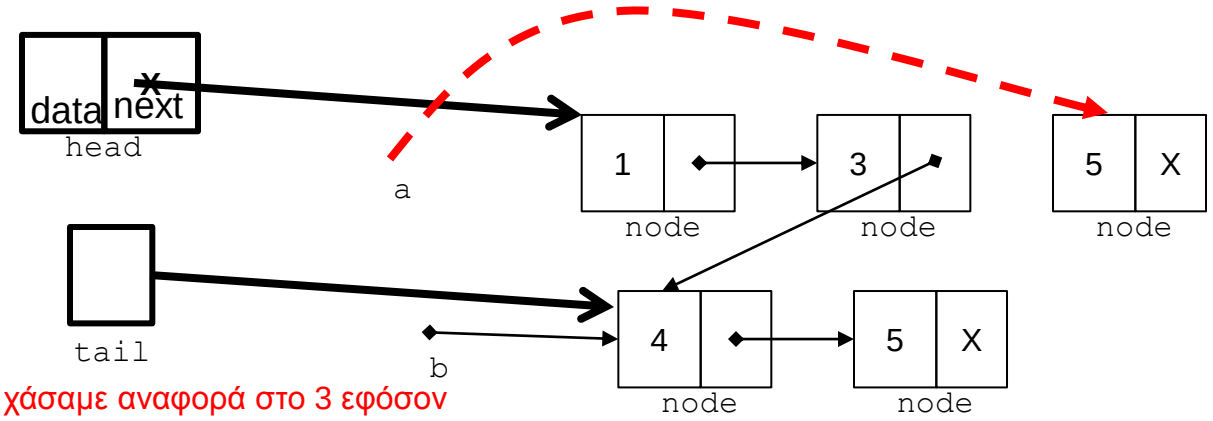
```
int main() {
    LIST *l1, *l2, *l3;
    l1 = l2 = l3 = NULL;
    if (initList(&l1)==EXIT_FAILURE || initList(&l2)==EXIT_FAILURE
    ||  initList(&l3) == EXIT_FAILURE) {
        printf("Error: Unable to initialize list");
        return EXIT_FAILURE;
    } // fill l1, l2 with some values at this point ...
    if (mergeLists(l1,l2,l3)==EXIT_FAILURE) { printf("…"); }
}
```

Πρόβλημα 2: `mergeLists` (Διάγραμμα Λύσης)

1. Έχουμε δυο μέτωπα λιστών a και b.
2. Θα δημιουργήσουμε **βοηθητικό κόμβο** `head` ο οποίος θα δείχνει στην κεφαλή της συγχωνευμένης λίστας.
3. Με **δείκτη** θα θυμόμαστε και το **τέλος** της συγχ. λίστας
   - έτσι ώστε να προσθέτουμε τον επόμενο στο `tail->next`:

```
NODE *tail = &head; // Αρχικοποίηση tail
```

4. Απεικόνιση Κατάστασης μετά την συγχώνευση 1, 3, 4:

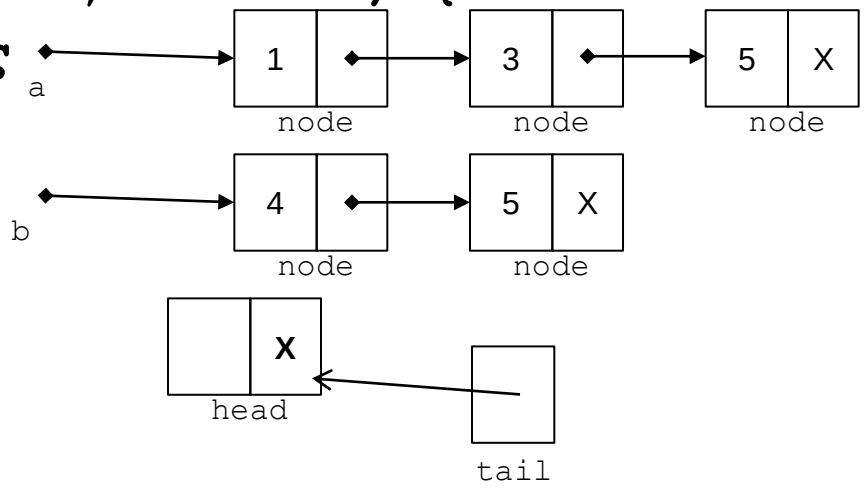


(Δεν χάσαμε αναφορά στο 3 εφόσον έδειχνε το tail πάνω του)
Το tail επιτελεί το ρόλο του prev δείκτη

Το **a** και **b** θα υποδεικνύουν τον επόμενο κόμβο κάθε λίστας

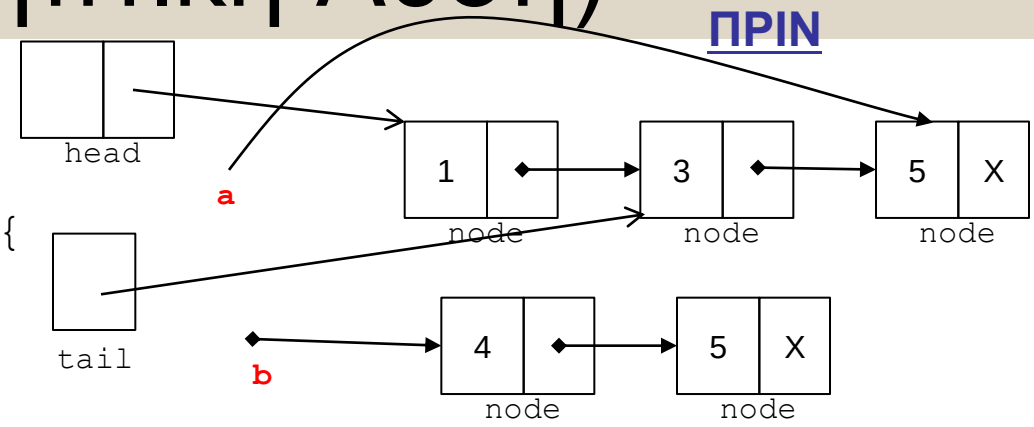
Πρόβλημα 2: `mergeLists` (Επαναληπτική Λύση)

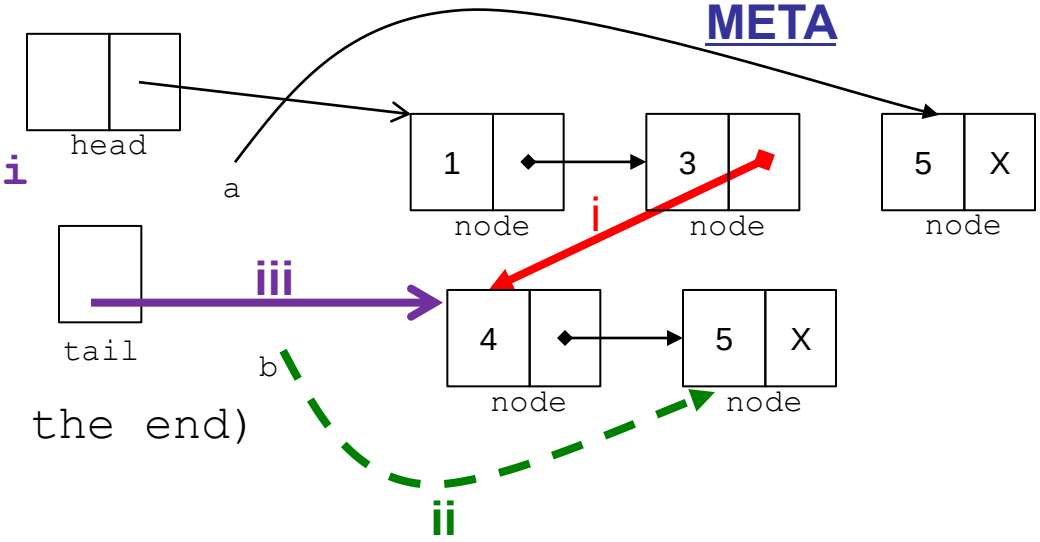
```
static NODE *MergeLoop(NODE *a, NODE *b) {
//Βοηθητικοί για αρχή, τέλος
    NODE head;
    NODE *tail = &head;

    head.next = NULL;
    // or tail->next = NULL
    while (true) {
        if (a == NULL) { // Φτάσαμε στο τέλος της a
            tail->next = b; // ένωσε υπόλοιπη λίστα b
            break;
        }
        else if (b == NULL) {// Φτάσαμε στο τέλος της b
            tail->next = a; // ένωσε υπόλοιπη λίστα a
            break;
        }
```



Πρόβλημα 2: `mergeLists` (Επαναληπτική Λύση)

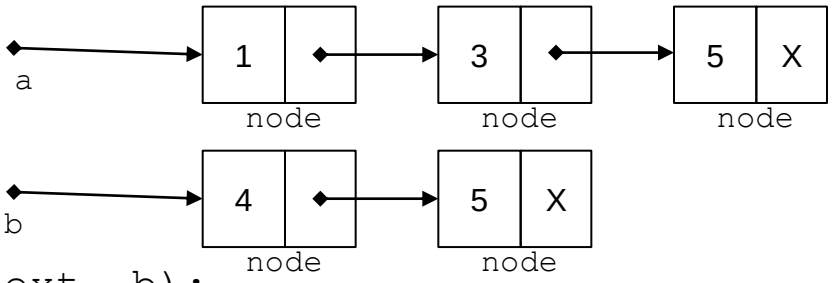

```
// Συνέχεια
//while (true) {    …
    if (a->data <= b->data) {
        tail->next = a;
        a = a->next;
    } else {
       tail->next = b; // i
       b = b->next;    // ii
    }

   tail = tail->next; // iii
  }
  // head.next now points
  // to the beginning of the
  // merged list (a,b are at the end)
  return head.next;
}
```





Πρόβλημα 3: `mergeLists` (Αναδρομική Λύση)

```
static NODE *mergeRecursive(NODE *a, NODE *b) {
    // Base cases: either terminated.
    if (a == NULL) return b;
    else if (b == NULL) return a;

    // Pick either a or b, and recur
    if (a->data <= b->data) {
        a->next = mergeRecursive(a->next, b);
        return a;
    }
    else {
        b->next = mergeRecursive(a, b->next);
        return b;
    }
}
```




```
int mergeLists(const LIST *l1, const LIST *l2, LIST *newl) {
    if ((l1 == NULL) || (l2 == NULL) || (l3 == NULL)) return EXIT_FAILURE;
    newl->head = mergeRecursive(l1->head, l2->head);
    newl->size = l1->size + l2->size;
    return EXIT_SUCCESS;
}
```