Προγραμματισμός ΙΙ (ECE116)

#6 διασυνδεδεμένες δομές: λίστες

Σπύρος Λάλης
lalis@uth.gr

Διασυνδεδεμένες δομές

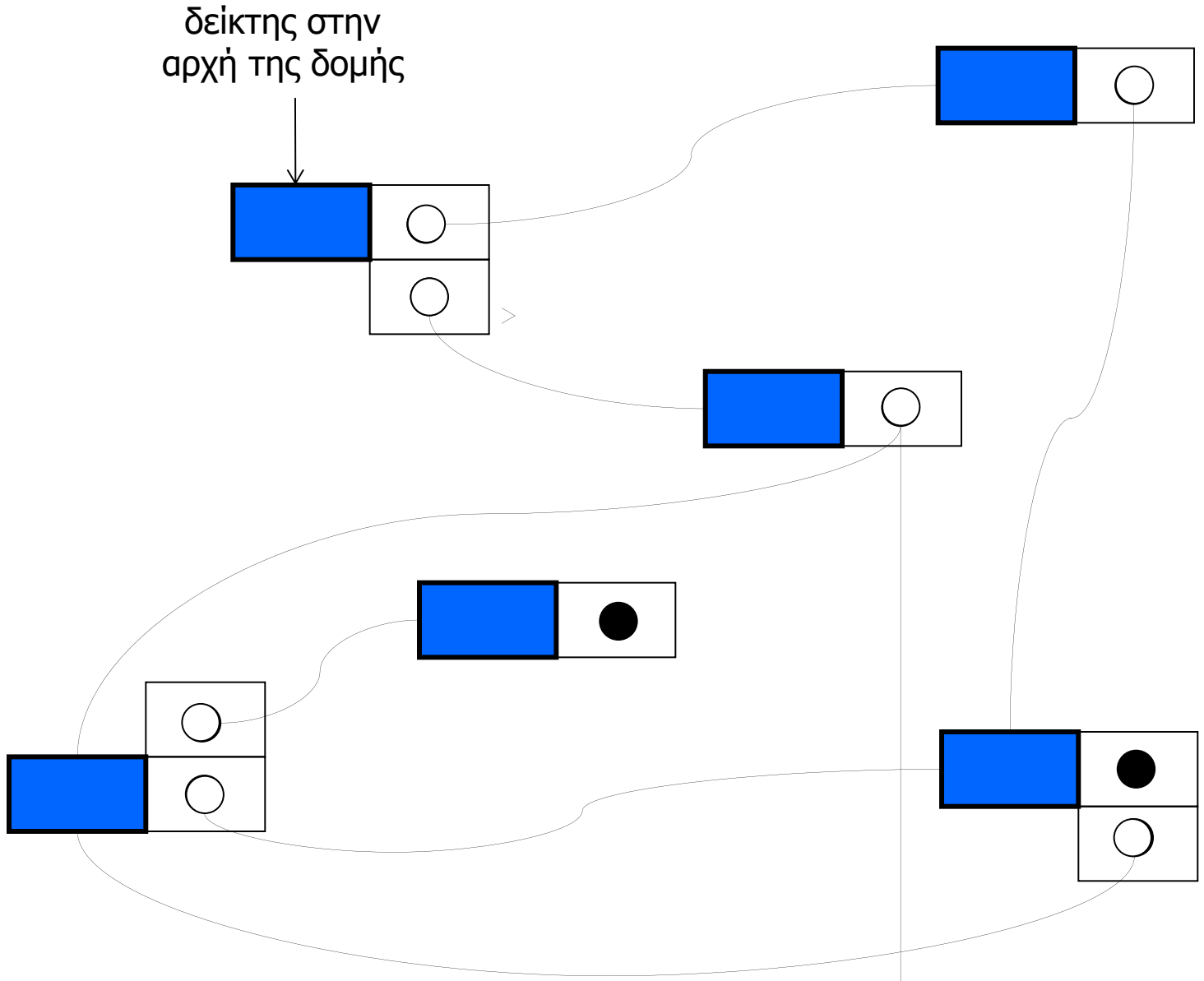
- Δομές με στοιχεία σε **μη-συνεχόμενες θέσεις** στην κυρίως μνήμη του Η/Υ

- Πως γνωρίζουμε **που** να βρούμε το «επόμενο» στοιχείο της δομής;
- Τα στοιχεία της δομής **συνδέονται** μεταξύ τους με ρητό τρόπο, μέσω **δεικτών**

- Τα στοιχεία ονομάζονται συχνά «**κόμβοι**» (nodes)

Κατασκευή διασυνδεδεμένων δομών

- Η κατασκευή γίνεται **σταδιακά**, κόμβο προς κόμβο
  - όχι μονομιάς, όπως π.χ. οι πίνακες
- Κάθε κόμβος δημιουργείται **ξεχωριστά**
  - με κατάλληλη δέσμευση δυναμικής μνήμης (ανά κόμβο)
- Και **διασυνδέεται** με τους ήδη υπάρχοντες κόμβους
- Ανάποδα, όταν ένας κόμβος απομακρύνεται, **προσαρμόζονται** οι διασυνδέσεις των υπολοίπων
  - και αποδεσμεύεται η μνήμη του κόμβου που αφαιρείται
- Κάθε δομή μπορεί να «λειτουργεί» **διαφορετικά**
  - διαφορετικοί κανόνες διαχείρισης/διασύνδεσης των κόμβων
  - διαφορετικές υλοποιήσεις των λειτουργιών αναζήτησης, προσθήκης/επέκτασης και απομάκρυνσης/συρρίκνωσης


δείκτης στην
αρχή της δομής

Διάσχιση διασυνδεδεμένων δομών

- Οι κόμβοι μιας διασυνδεδεμένης δομής **δεν** βρίσκονται σε διαδοχικές θέσεις στην μνήμη
  - σε αντίθεση με τα στοιχεία ενός (δυναμικού) πίνακα
- Η θέση του «i-οστού» κόμβου **δεν** είναι άμεσα υπολογίσιμη μέσω μιας αριθμητικής έκφρασης
  - μπορεί να μην υφίσταται καν κάποια αυστηρή σειρά

- Κάθε κόμβος πρέπει να **εντοπιστεί** με κατάλληλη **διάσχιση** της δομής (traversal)
  - ακολουθώντας τις συνδέσεις μεταξύ των κόμβων
- Αυτό μπορεί να είναι μια **πολύπλοκη** διαδικασία

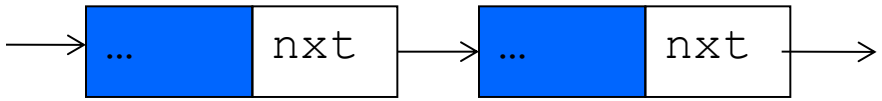
Αναδρομικές/επαναλαμβανόμενες δομές

- Όταν ένα από τα πεδία της δομής είναι δείκτης σε δομή ιδίου τύπου (**δομική αναδρομή**)

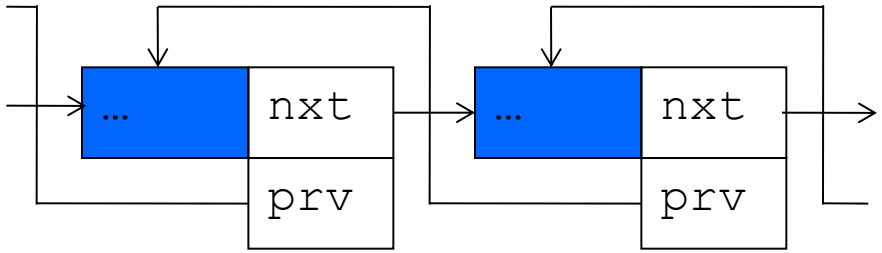
- Οι αναδρομικές δομές κατασκευάζονται με επανάληψη του ίδιου δομικού στοιχείου (κόμβου)

- Οι διασυνδέσεις μεταξύ των κόμβων (η αρχικοποίηση των δεικτών από κόμβο σε κόμβο) γίνεται σύμφωνα με τους **κανόνες** που διέπουν την δομή
  - διαφέρουν ανάλογα με την περίπτωση

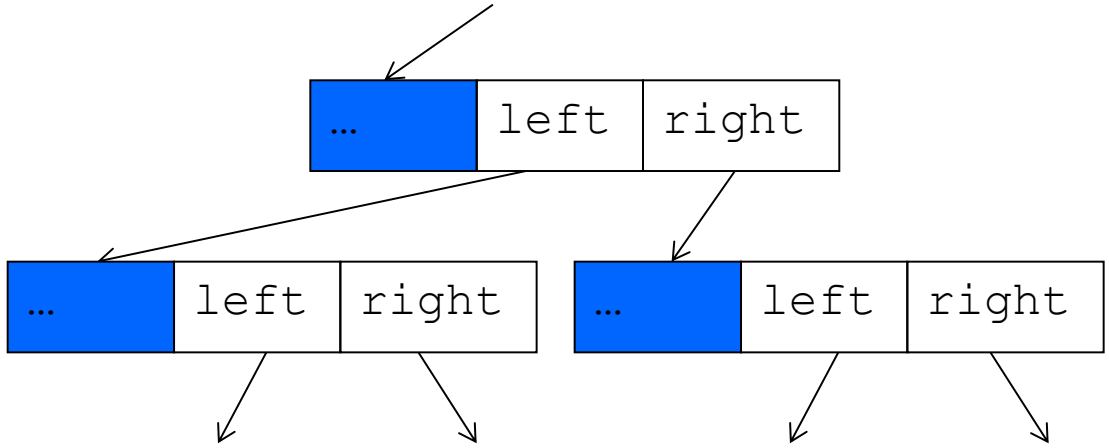
```
struct list {
  … /* data */
  struct list *nxt;
};
```



```
struct list2 {
  … /* data */
  struct list2 *nxt;
  struct list2 *prv;
};
```



```
struct btree {
  … /* data */
  struct btree *left;
  struct btree *right;
};
```



Απλή λίστα

Συμβάσεις

- Κάθε κόμβος περιέχει έναν δείκτη που δείχνει στον **επόμενο** κόμβο της λίστας
- Αν ο κόμβος είναι ο τελευταίος στην λίστα, ο δείκτης προς τον επόμενο κόμβο είναι `NULL`

- Η αρχή της λίστας είναι ένας ξεχωριστός **δείκτης** που δείχνει στον **πρώτο** κόμβο της λίστας
- Ονομάζεται **κεφαλή** της λίστας
- Μέσω της κεφαλής μπορούμε να διασχίσουμε μέρος ή ολόκληρη την λίστα (από την αρχή μέχρι το τέλος)
- Αν η λίστα είναι άδεια, η κεφαλή είναι `NULL`

```
struct list {
  int key; /* plays the role of a search key/id */
  …        /* more data, irrelevant for this example */
  struct list *nxt; /* pointer to next node */
};
```

```
void list_init(struct list **head) {
  *head = NULL;
}
```

```
struct list *list_new() {
  return(NULL);
}
```

Διαδικασία αναζήτησης

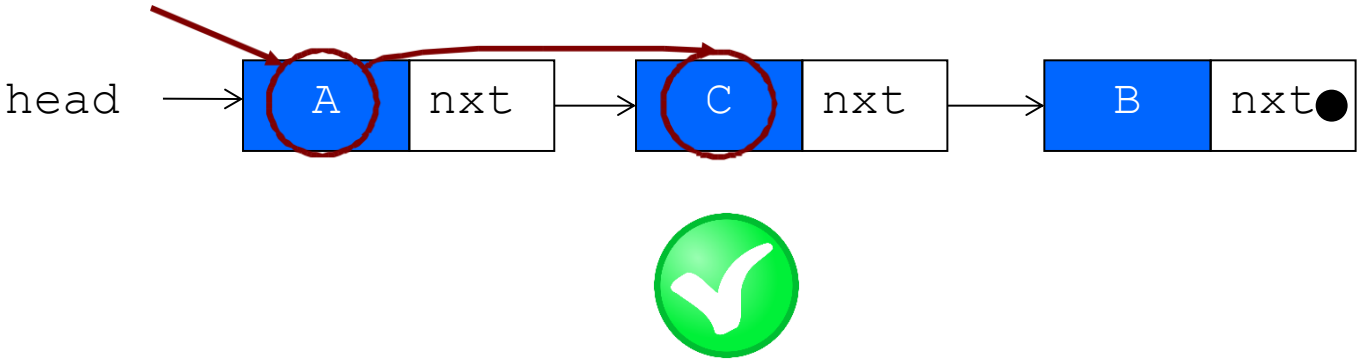
- Αρχίζουμε από την κεφαλή
- Διασχίζουμε την λίστα, **κόμβο προς κόμβο**
- Μέχρι να βρούμε τον κόμβο με την επιθυμητή τιμή
- Σε κάθε βήμα ελέγχουμε αν ο τρέχων κόμβος είναι `NULL`
  - αν ναι, φτάσαμε στο τέλος της λίστας

Ειδική(;) περίπτωση: η λίστα είναι άδεια

- **Δεν** χρειάζεται ειδική μεταχείριση σε επίπεδο κώδικα
- Αρκεί ο «συμβατικός» έλεγχος για `NULL`
- Η αναζήτηση θα σταματήσει στο πρώτο κιόλας βήμα

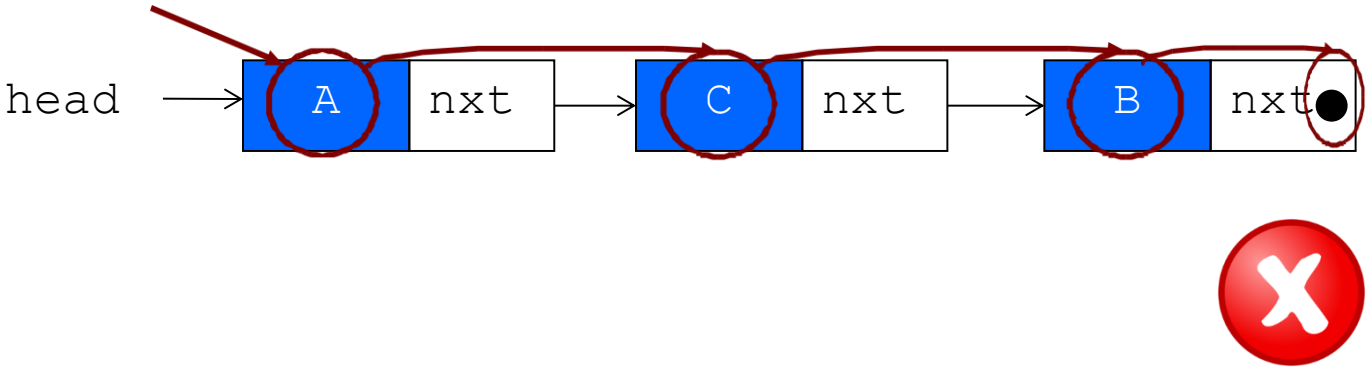
Αναζήτηση (επιτυχημένη)

```
find C
```



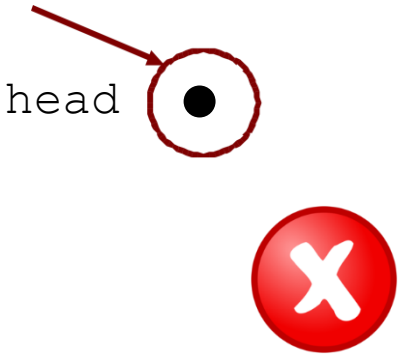
Αναζήτηση (αποτυχημένη)

```
find D
```



Αναζήτηση (αποτυχημένη)

```
find D
```



```
int list_find(struct list *head, int key) {
  struct list *curr;

  for (curr=head; (curr != NULL) && (curr->key != key);
                                        curr=curr->nxt);
  return(curr != NULL);
}
```

```
struct list *list_find(struct list *head, int key) {
  struct list *curr;

  for (curr=head; (curr != NULL) && (curr->key != key);
                                        curr=curr->nxt);
  return(curr);
}
```

Διαδικασία εισαγωγής

- Δημιουργούμε έναν καινούργιο κόμβο στον οποίο αποθηκεύουμε την επιθυμητή τιμή
- Εισάγουμε τον κόμβο **στην αρχή** της λίστας

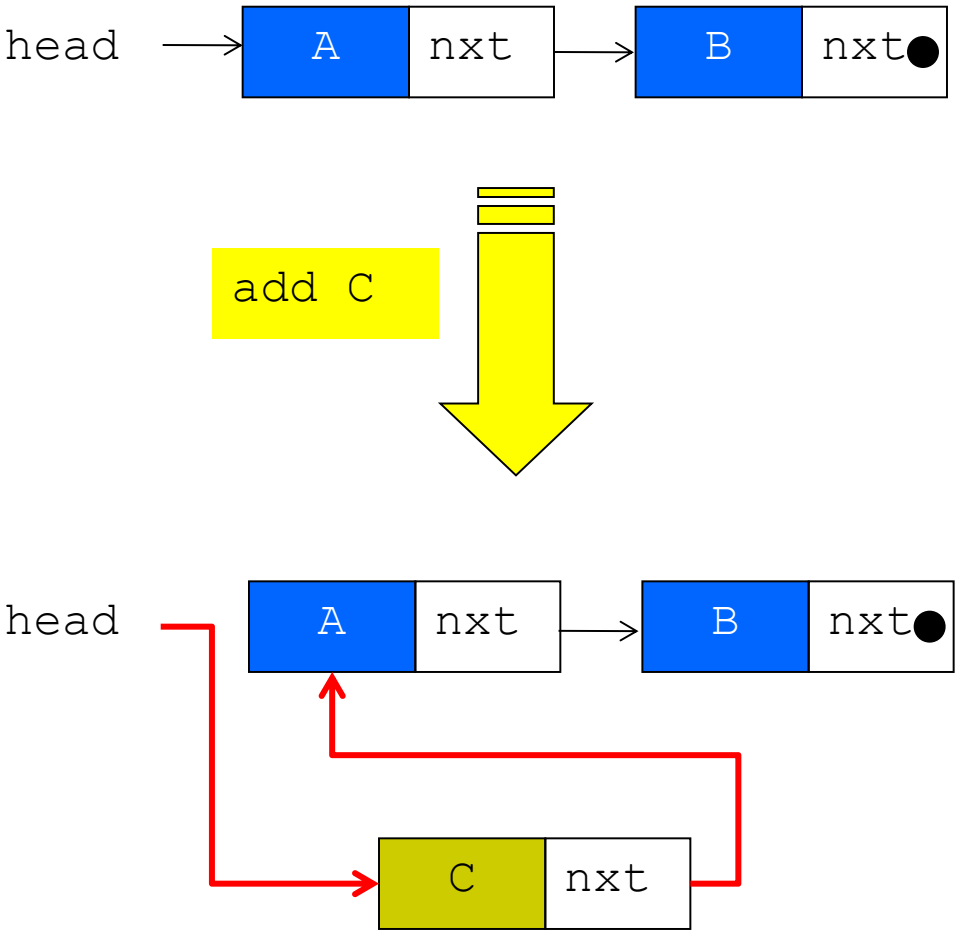
Γενική περίπτωση: η λίστα δεν είναι άδεια

Ειδική(;) περίπτωση: η λίστα είναι άδεια

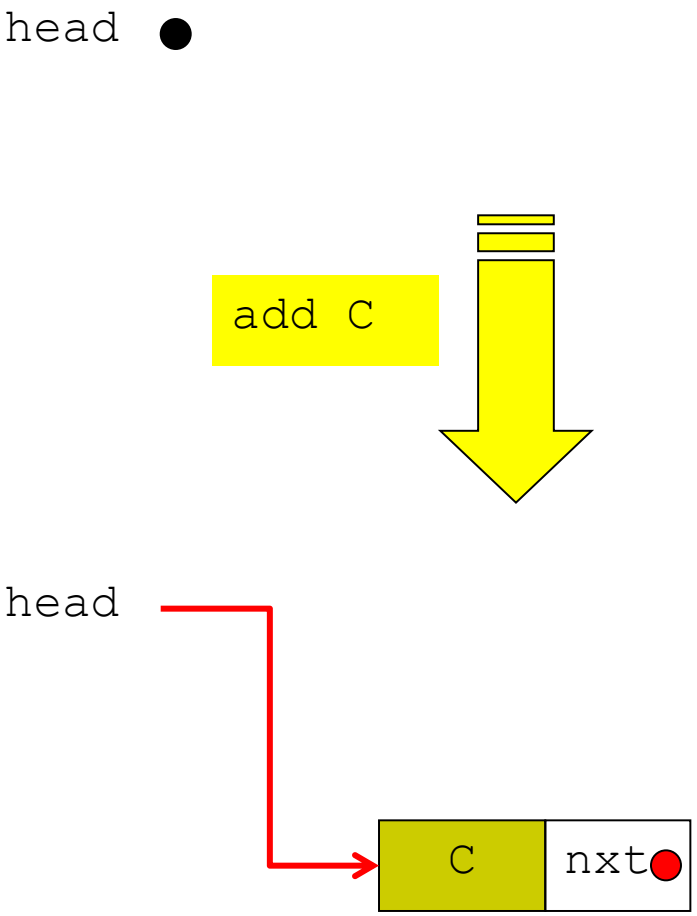
- Και στις δύο περιπτώσεις, κάνουμε τις **ίδιες** ενέργειες

1. Ο νέος κόμβος δείχνει στην παλιά κεφαλή της λίστας
2. Η κεφαλή αλλάζει ώστε να δείχνει στον νέο κόμβο

Εισαγωγή στην αρχή (σε μη κενή λίστα)



Εισαγωγή στην αρχή (σε κενή λίστα)



```
void list_insert(struct list **head, int key) {
  struct list *curr;

  curr = (struct list *)malloc(sizeof(struct list));

  curr->key = key;
  curr->nxt = *head; /* insert node at the beginning */
  *head = curr;
}
```

```
struct list *list_insert(struct list *head, int key) {
  …
  return(curr);
}
```

Διαδικασία απομάκρυνσης

- Κατά την αναζήτηση πρέπει να κρατάμε έναν δείκτη στον **προηγούμενο** κόμβο αυτού που ελέγχουμε
  - για να μπορεί να **παρακαμφθεί** ο ζητούμενος κόμβος

Εύκολη περίπτωση: ο κόμβος δεν βρέθηκε

- Δεν κάνουμε τίποτα

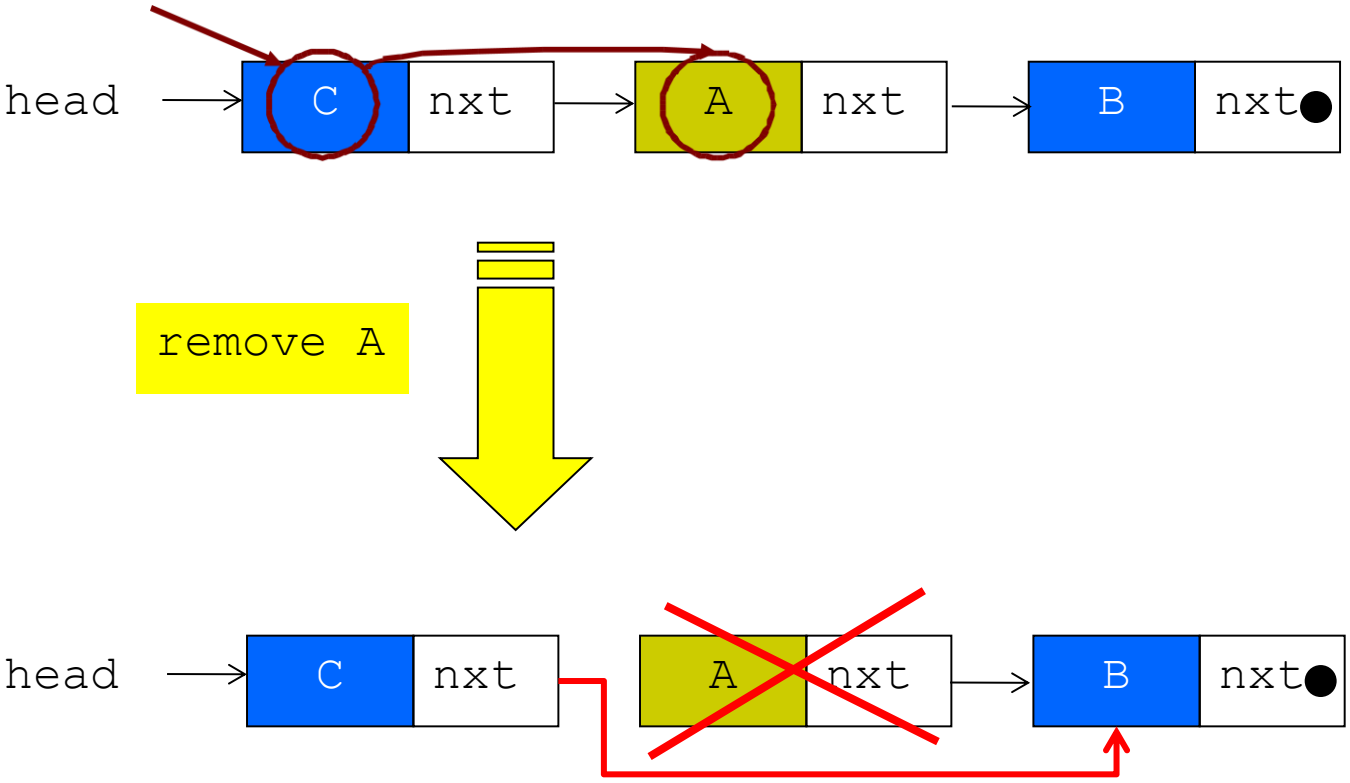
Γενική περίπτωση: ο κόμβος **δεν** είναι στην αρχή

- Ο προηγούμενος κόμβος πρέπει να **αλλάξει** έτσι ώστε να δείχνει στον **μεθεπόμενο** κόμβο

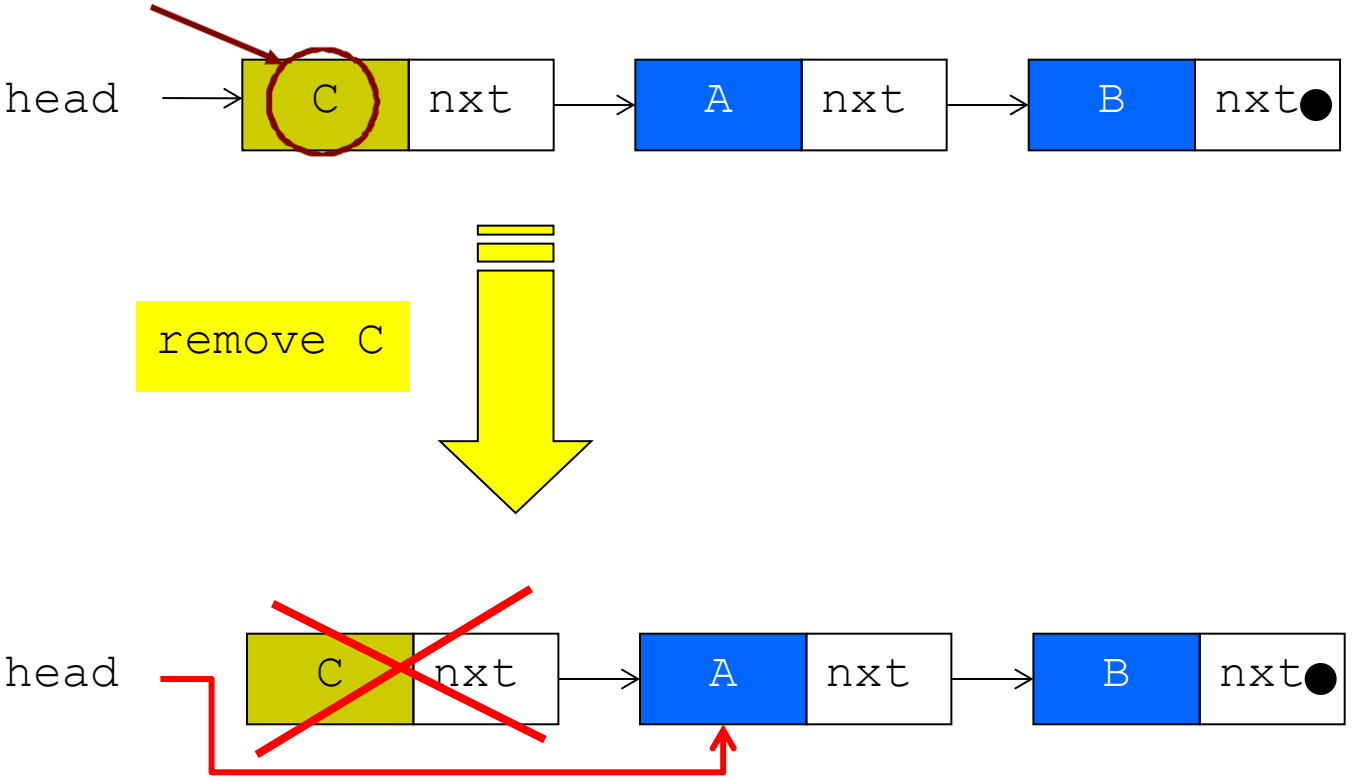
Ειδική περίπτωση: ο κόμβος **είναι** στην αρχή

- Δεν υπάρχει προηγούμενος κόμβος
- Πρέπει να αλλάξει η **κεφαλή** της λίστας για να δείχνει στον **μεθεπόμενο** κόμβο

Απομάκρυνση (με προηγούμενο κόμβο)



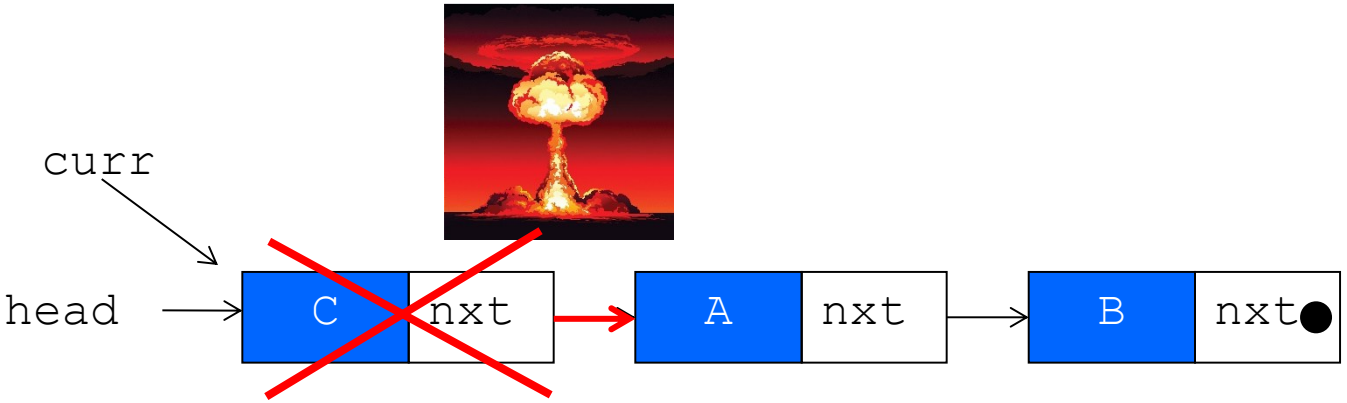
Απομάκρυνση (χωρίς προηγούμενο κόμβο)



```
void list_remove(struct list **head, int key) {
  struct list *curr,*prev;

  for (curr=*head; (curr != NULL) && (curr->key != key);
                                     prev=curr,curr=curr->nxt);

  if (curr != NULL) {
    if (curr == *head) {
      *head = curr->nxt; /* redirect head */
    }
    else {
      prev->nxt = curr->nxt; /* redirect nxt of prev node */
    }
    free(curr); /* free node memory */
  }
}
```

```
struct list *list_remove(struct list *head, int key) {
  …
  return(head);
}
```

Καταστροφή λίστας



- **Δεν** επιτρέπεται να προσπελάσουμε τα πεδία ενός κόμβου που ήδη απελευθερώσαμε!
- Πρέπει να ξέρουμε ποιος είναι ο επόμενος κόμβος ...
- ... **προτού** απελευθερώσουμε τον προηγούμενο

```
void list_destroy(struct list *head) {
  struct list *curr,*prev;

  curr = head;
  while (curr != NULL) {
    prev = curr; /* set pointer to current */
    curr = curr->nxt; /* move current to next node */
    free(prev); /* free previous node */
  }
}
```

Κυκλική λίστα
με ειδικό τερματικό κόμβο

Χρήση τερματικού κόμβου

- Στην απλή λίστα, σε κάθε βήμα της αναζήτησης γίνονται **δύο έλεγχοι**: (α) αν φτάσαμε στο τέλος της λίστας, (β) αν βρισκόμαστε στον επιθυμητό κόμβο
- Επίσης, χρειάζεται **ειδικός χειρισμός** για την περίπτωση της απομάκρυνσης του πρώτου κόμβου

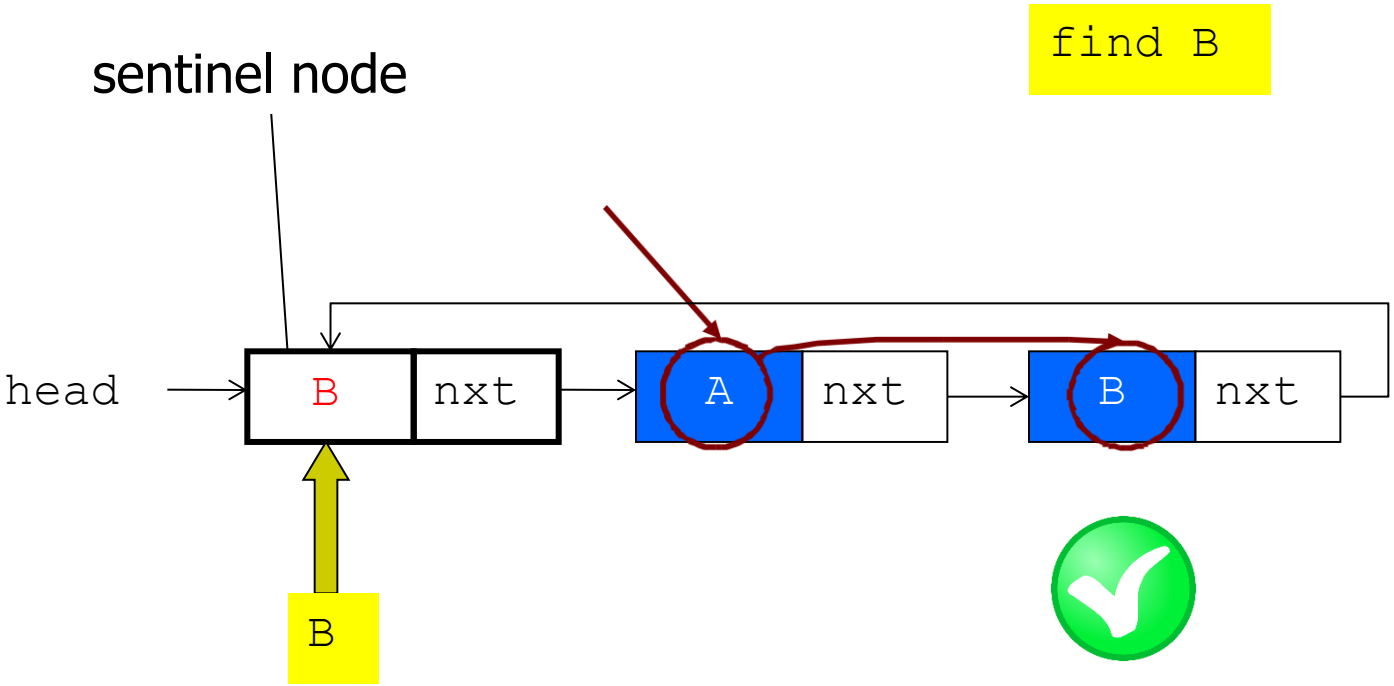
- **Βελτιστοποίηση:** υλοποιούμε τη λίστα ως **κυκλική** λίστα, και χρησιμοποιούμε έναν **ειδικό** κόμβο (που δεν έχει «πραγματικό» περιεχόμενο/δεδομένα) ως **τερματικό κόμβο / sentinel** στην αναζήτηση
- Αρχικά η (κενή) λίστα αποτελείται μόνο από τον τερματικό κόμβο που δείχνει ... στον εαυτό του

```
struct list {
  int key; /* plays the role of a search key/id */
  …        /* more data, irrelevant for this example */
  struct list *nxt; /* pointer to next node */
};
```

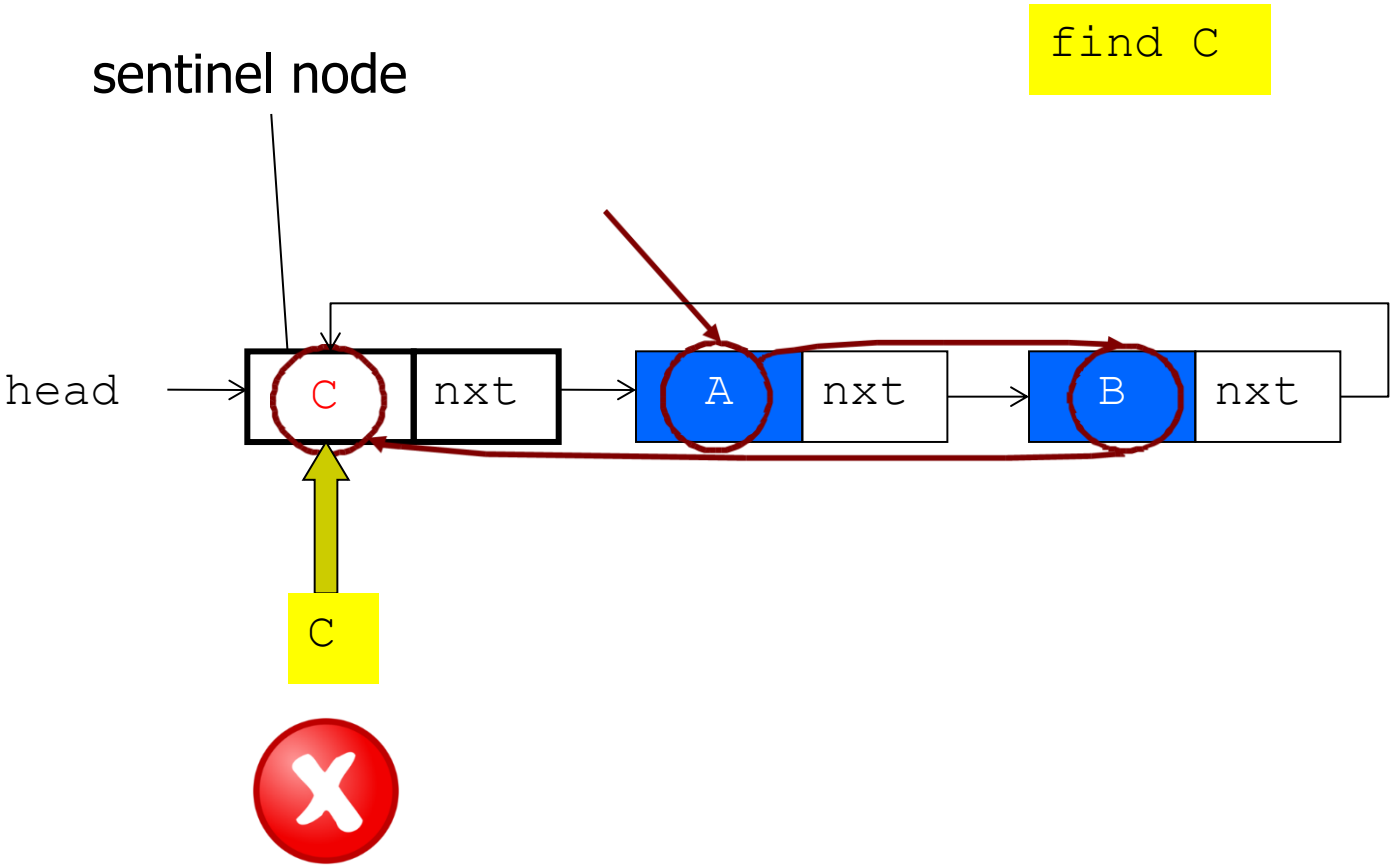
```
void list_init(struct list **head) {
  *head = (struct list *)malloc(sizeof(struct list));
  (*head)->nxt = *head;
}
```

```
struct list *list_new() {
  struct list *head;

  head = (struct list *)malloc(sizeof(struct list));
  head->nxt = head;
  return(head);
}
```

Αναζήτηση (πετυχημένη)



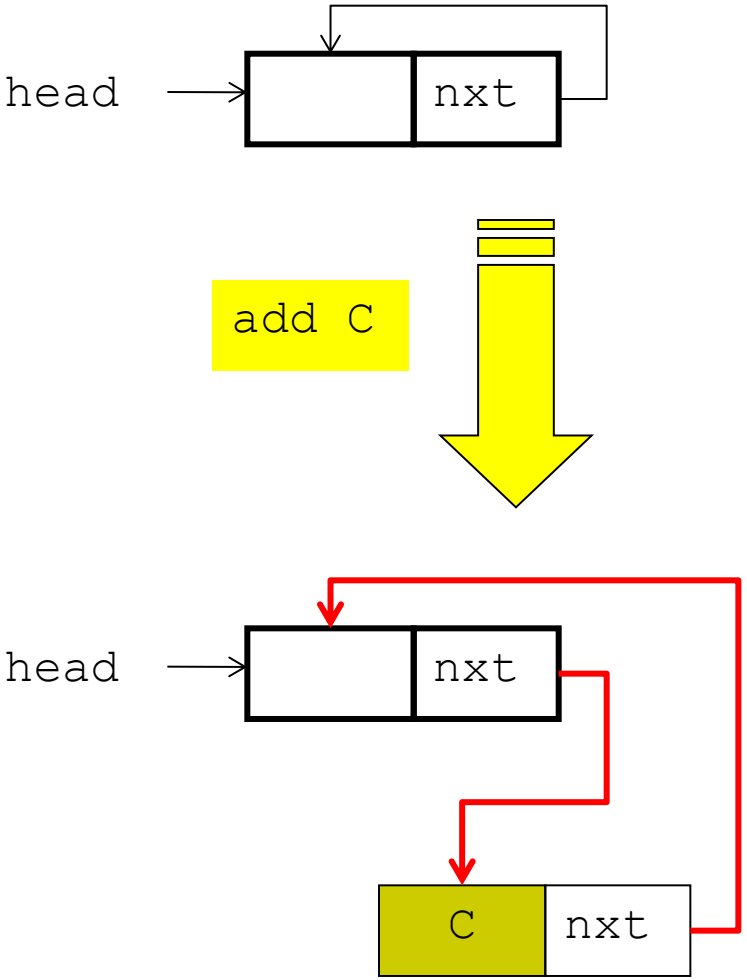
Αναζήτηση (αποτυχημένη)



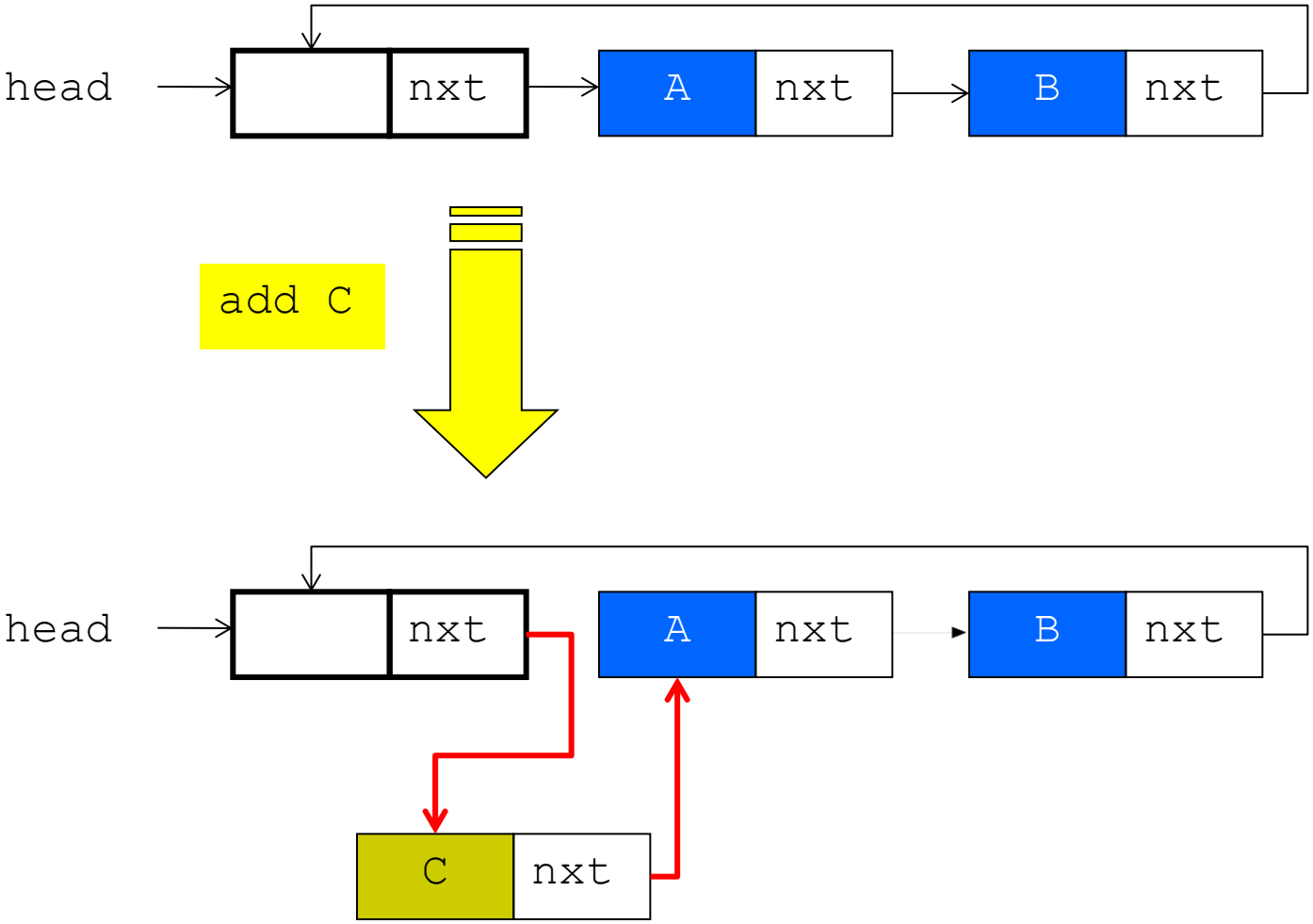
```
int list_find(struct list *head, int key) {
  struct list *curr;

  head->key = key; /* ensure termination */

  for (curr=head->nxt; curr->key!=key; curr=curr->nxt);

  return(curr != head);
}
```

```
struct list *list_find(struct list *head, int key) {
  struct list *curr;

  head->key = key; /* ensure termination */

  for (curr=head->nxt; curr->key!=key; curr=curr->nxt);

  return(curr);
}
```

Εισαγωγή (σε κενή λίστα)

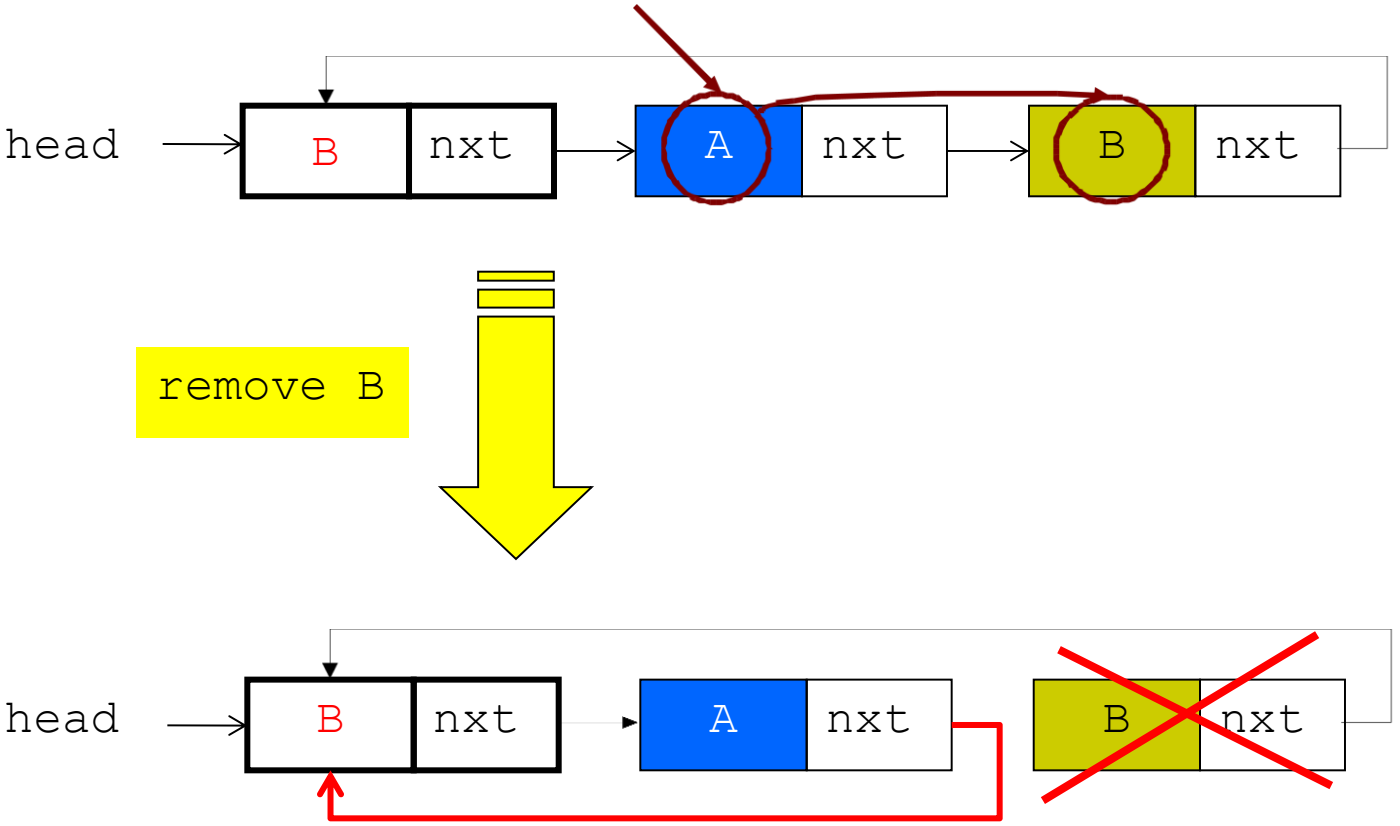


Εισαγωγή (σε μη κενή λίστα)

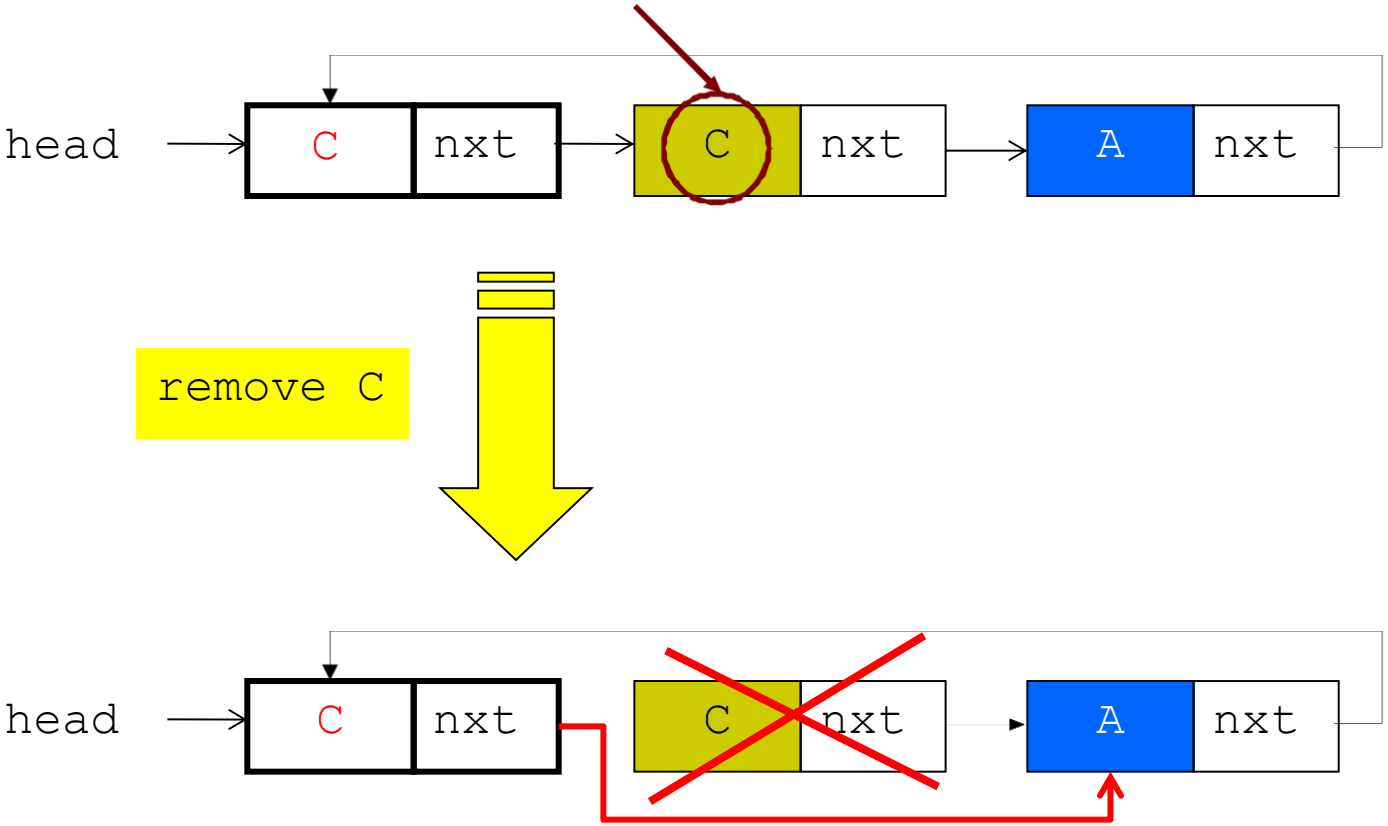


```
void list_insert(struct list *head, int key) {
  struct list *curr;

  curr = (struct list *)malloc(sizeof(struct list));

  curr->key = key;
  curr->nxt = head->nxt;
  head->nxt = curr;
}
```

Απομάκρυνση (με προηγούμενο κόμβο)

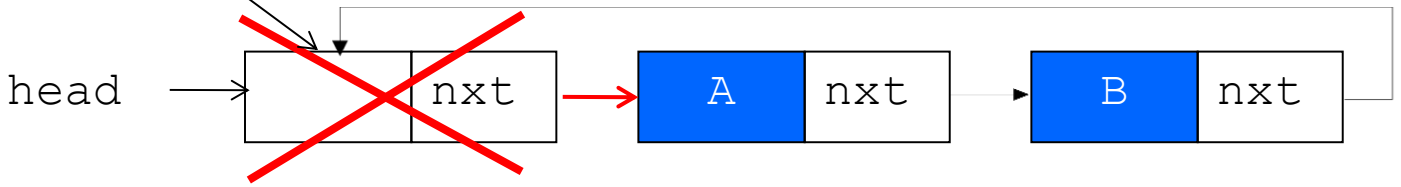


Απομάκρυνση («χωρίς» προηγούμενο κόμβο)



```
void list_remove(struct list *head, int key) {
  struct list *curr,*prev;

  head->key = key; /* ensure termination */

  for (prev=head,curr=head->nxt; curr->key!=key;
                                 prev=curr,curr=curr->nxt);

  if (curr != head) {
    prev->nxt = curr->nxt;
    free(curr);
  }
}
```

Καταστροφή λίστας



- **Δεν** επιτρέπεται να προσπελάσουμε τα πεδία ενός κόμβου που ήδη απελευθερώσαμε!
- Πρέπει να ξέρουμε ποιος είναι ο επόμενος κόμβος ... **προτού** απελευθερώσουμε τον τρέχοντα
- Παραλλαγή της λύσης για την απλή λίστα
  - ξέρουμε ότι η κεφαλή της λίστας **δεν** είναι `NULL`

```
void list_destroy(struct list *head) {
  struct list *curr,*prev;

  curr = head;
  do {
    prev = curr;
    curr = curr->nxt;
    free(prev);
  } while (curr != head);
}
```

γιατί επιτρέπεται η σύγκριση με μια διεύθυνση μνήμης που έχει ήδη **απελευθερωθεί**;

Διπλά συνδεδεμένη λίστα (με τερματικό στοιχείο)

Απλοποίηση απομάκρυνσης

- Στην απλή λίστα δεν γίνεται να πάμε προς τα πίσω
- Επίσης, στην διαδικασία της απομάκρυνσης, η λίστα πρέπει να διασχίζεται με δύο δείκτες
- Σε κάθε βήμα απαιτούνται **δύο** αλλαγές δεικτών
- **Βελτιστοποίηση:** υλοποιούμε τη λίστα ως **διπλά συνδεδεμένη** λίστα, όπου κάθε κόμβος δείχνει όχι μόνο στον επόμενο αλλά **και** στον προηγούμενο κόμβο
- Απλοποιείται η διαδικασία απομάκρυνσης
  - **δεν** χρειάζεται να έχουμε δείκτη στον προηγούμενο κόμβο
- Όμως απαιτείται σωστή διαχείριση του επιπλέον δείκτη που υπάρχει σε κάθε κόμβο της λίστας

```
struct list2 {
  int key; /* plays the role of a search key/id */
  …        /* more data, irrelevant for this example */
  struct list2 *nxt; /* pointer to next node */
  struct list2 *prv; /* pointer to previous node */
};
```

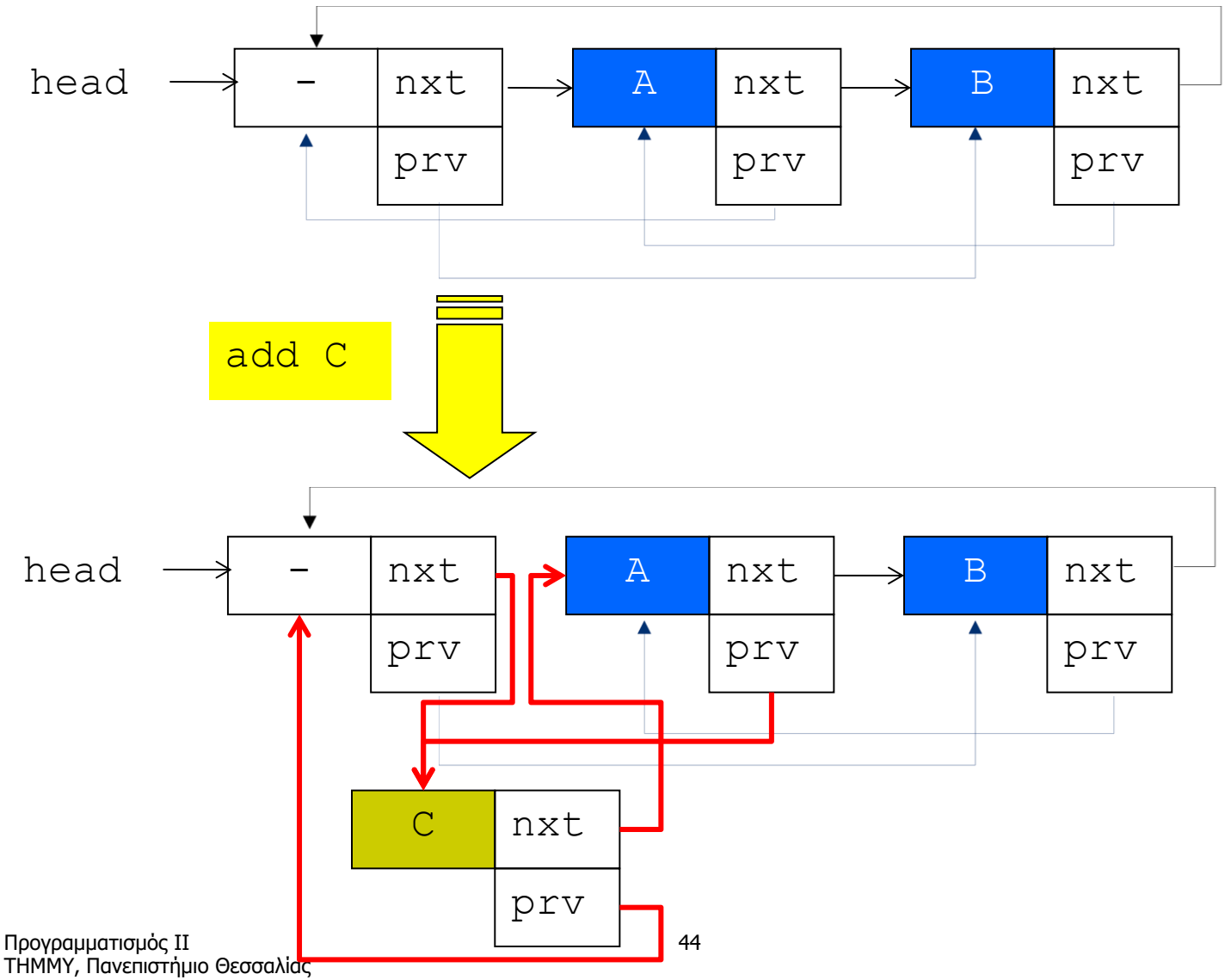
```
void list_init(struct list2 **head) {
  *head = (struct list2 *)malloc(sizeof(struct list2));
  (*head)->nxt = *head;
  (*head)->prv = *head;
}
```

```
struct list2 * list_new() {
 struct list2 *head;

 head = (struct list2 *)malloc(sizeof(struct list2));
 head->nxt = head;
 head->prv = head;
 return(head);
}
```

```
int list_find(struct list2 *head, int key) {
  struct list2 *curr;

  head->key = key;
  for (curr=head->nxt; curr->key!=key; curr=curr->nxt);
  return(curr != head);
}
```

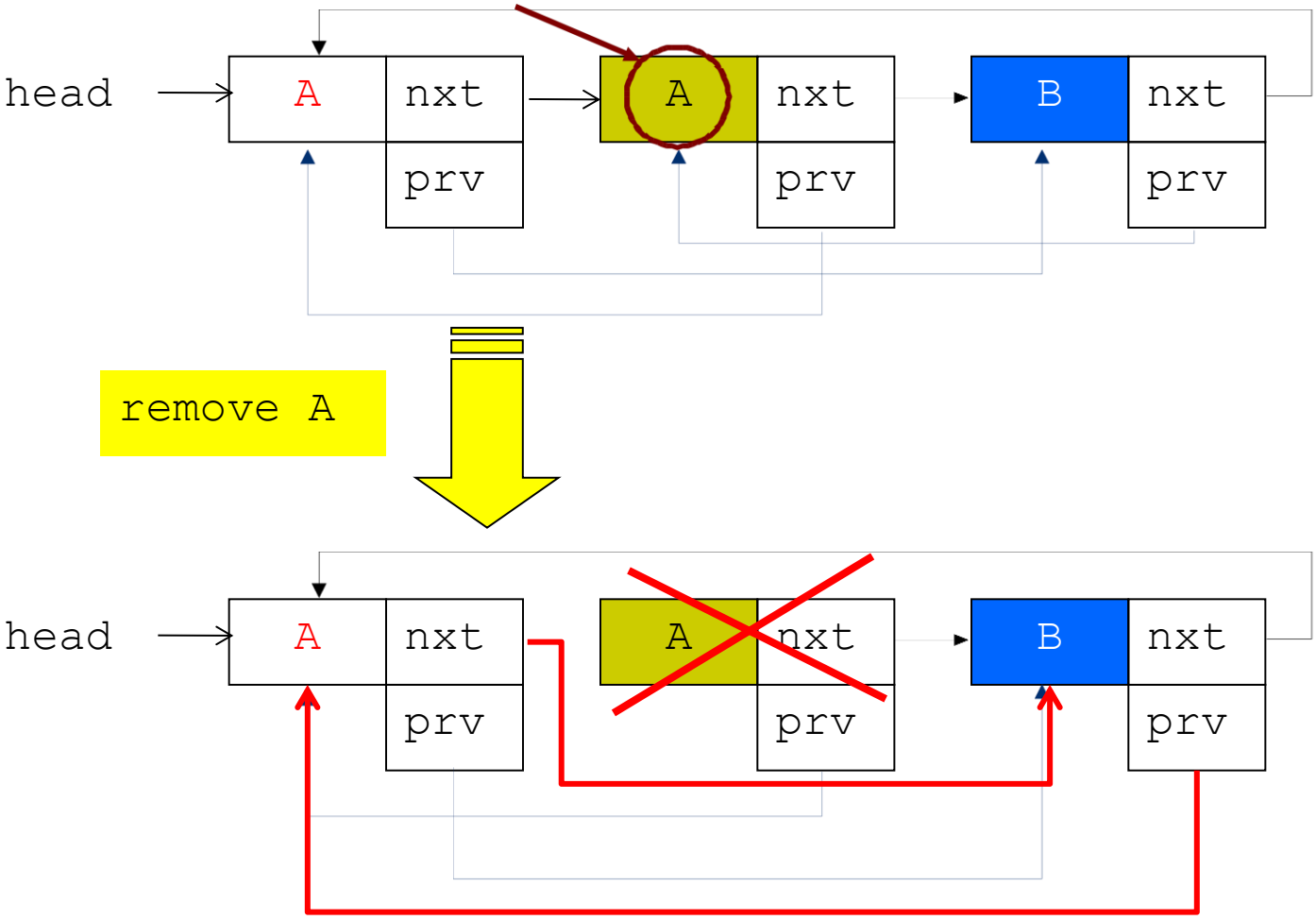
```
struct list2 *list_find(struct list2 *head, int key) {
  struct list2 *curr;

  head->key = key;
  for (curr=head->nxt; curr->key!=key; curr=curr->nxt);
  return(curr);
}
```

Εισαγωγή


head
-
nxt
prv
A
nxt
prv
B
nxt
prv
add C
head
-
nxt
prv
A
nxt
prv
B
nxt
prv
C
nxt
prv

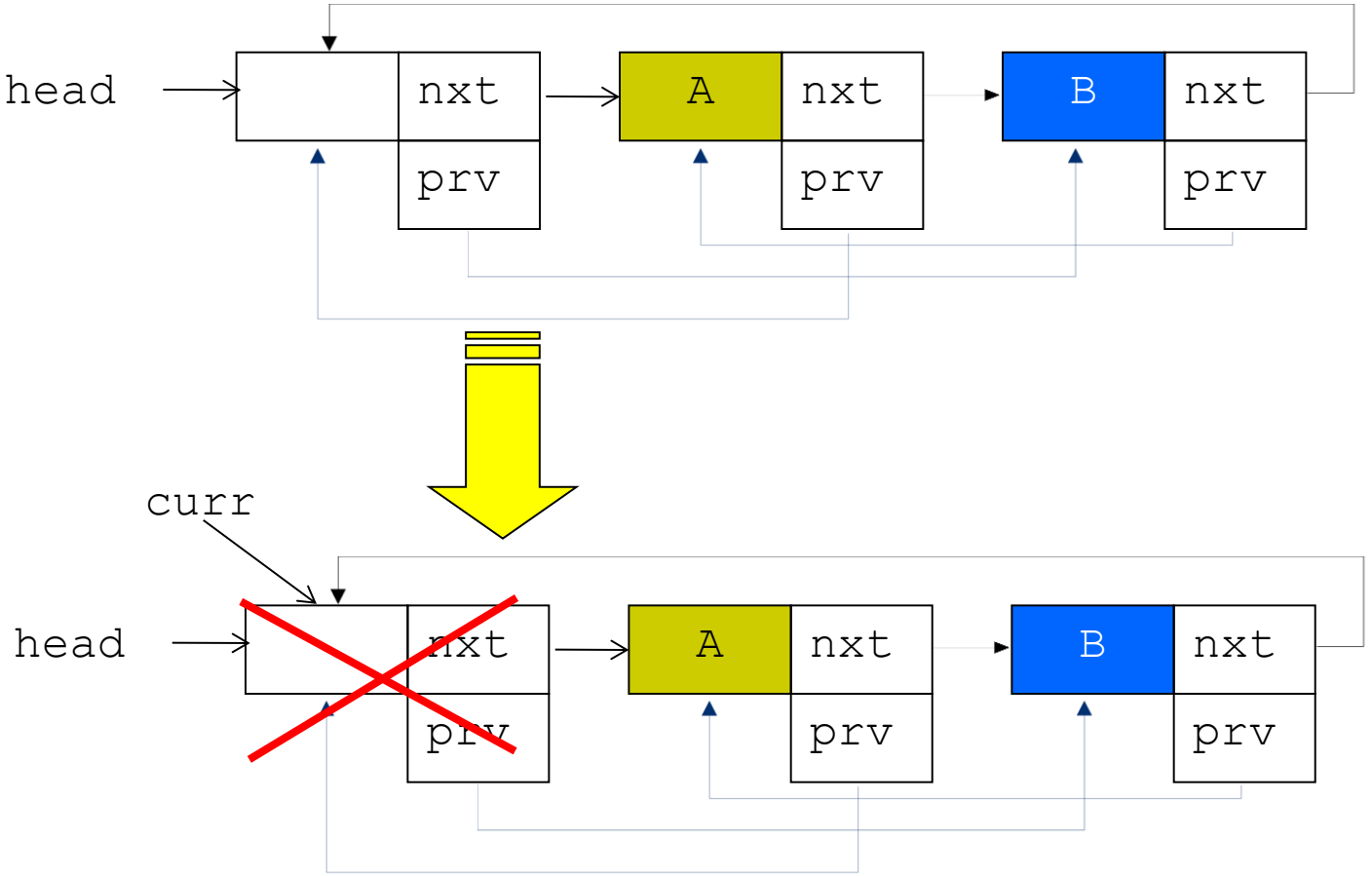
```
void list_insert(struct list2 *head, int key) {
  struct list2 *curr;

  curr = (struct list2 *)malloc(sizeof(struct list2));

  curr->key = key;

  curr->nxt = head->nxt;
  curr->prv = head;
  curr->nxt->prv = curr;
  head->nxt = curr;
}
```

Απομάκρυνση

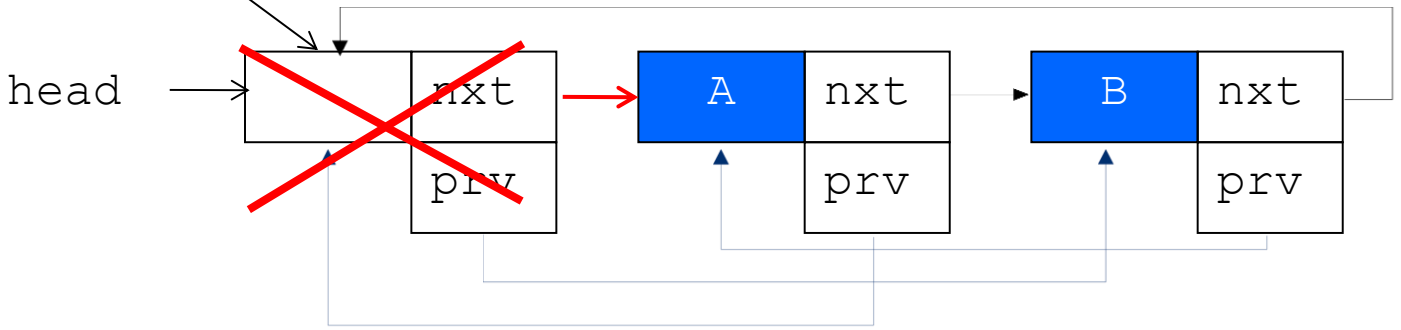


```
void list_remove(struct list2 *head, int key) {
  struct list2 *curr;

  head->key = key;
  for (curr=head->nxt; curr->key!=key curr=curr->nxt);

  if (curr != head) {
    curr->nxt->prv = curr->prv;
    curr->prv->nxt = curr->nxt;
    free(curr);
  }
}
```

Καταστροφή λίστας



Καταστροφή λίστας



- **Δεν** επιτρέπεται να προσπελάσουμε τα πεδία ενός κόμβου που ήδη απελευθερώσαμε!
- Πρέπει να ξέρουμε ποιος είναι ο επόμενος κόμβος ... **προτού** απελευθερώσουμε τον τρέχοντα
- Βλέπε λύση για την κυκλική λίστα με τερματικό

```
void list_destroy(struct list2 *head) {
  struct list2 *curr,*prev;

  curr = head;
  do {
    prev = curr;
    curr = curr->nxt;
    free(prev);
  } while (curr != head);
}
```

Επισκόπηση

- Απλή λίστα
  - δύο συγκρίσεις ανά βήμα αναζήτησης
  - δύο αναθέσεις ανά βήμα αναζήτησης για απομάκρυνση
- Κυκλική λίστα με τερματικό
  - μια σύγκριση ανά βήμα αναζήτησης
  - δύο αναθέσεις ανά βήμα αναζήτησης για απομάκρυνση
- Διπλά συνδεδεμένη κυκλική λίστα με τερματικό
  - μια σύγκριση ανά βήμα αναζήτησης
  - μια ανάθεση ανά βήμα αναζήτησης για απομάκρυνση
  - άμεση προσθήκη/απομάκρυνση στο / από το τέλος της λίστας (χωρίς ανάγκη για αναζήτηση)

Επιτάχυνση της αναζήτησης

- Όταν ένας κόμβος δεν υπάρχει, ελέγχονται **όλοι** οι κόμβοι μέχρι να φτάσουμε στο τέλος της λίστας
  - ανεξάρτητα από το αν χρησιμοποιούμε τερματικό

- **Λύση: Ταξινομημένη λίστα**
  - οι κόμβοι είναι στην λίστα με αύξουσα/φθίνουσα σειρά (ως προς κάποιο πεδίο-κλειδί ή συνδυασμό πεδίων)

- Αλλάζει η **διαδικασία εισαγωγής**
- Ένας νέος κόμβος πρέπει να τοποθετείται **μετά** από όλους τους κόμβους με «μικρότερες» τιμές, και **πριν** από όλους τους κόμβους με «μεγαλύτερες» τιμές

Ταξινομημένη vs. μη-ταξινομημένη λίστα

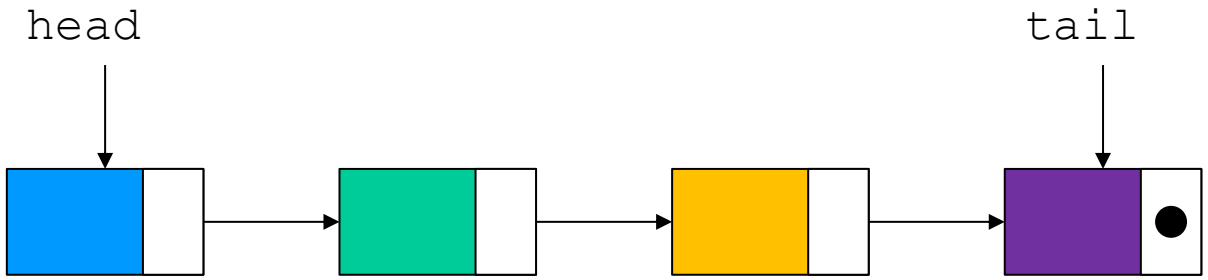
- Αν ο κόμβος υπάρχει, και οι δύο μέθοδοι απαιτούν κατά μέσο όρο $N/2$ βήματα – σκεφτείτε γιατί!
- Η διαφορά υπάρχει όταν ο αναζητούμενος κόμβος **δεν** βρίσκεται στην λίστα
- Σε ταξινομημένη λίστα με N κόμβους χρειάζονται κατά μέσο όρο **μόνο** $N/2$ βήματα
  - σε μη-ταξινομημένη λίστα χρειάζονται **πάντα** N βήματα
- Η ταξινομημένη λίστα γίνεται ακόμα πιο ελκυστική όταν **δεν** επιτρέπονται «διπλότυποι» κόμβοι
  - η εισαγωγή **υποχρεωτικά** περιλαμβάνει αναζήτηση, για να σιγουρευτούμε ότι δεν υπάρχει ήδη «ίδιος» κόμβος
- Ακόμα και σε ταξινομημένη λίστα **δεν** μπορεί να γίνει δυαδική αναζήτηση (binary search) – γιατί;

Ουρά προτεραιότητας FIFO με λίστα

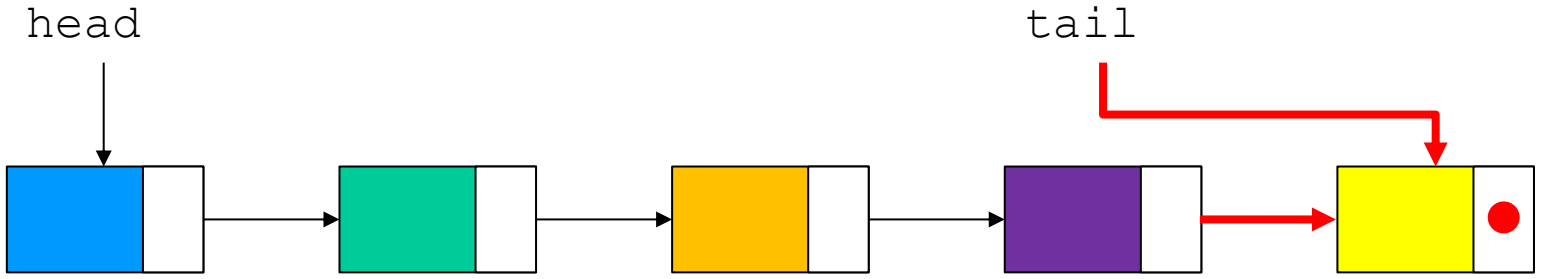
- FIFO: first in first out
- Τα δεδομένα πρέπει να βγαίνουν από την ουρά με την (χρονολογική) σειρά με την οποία μπήκαν

Ενδεικτική υλοποίηση με λίστα

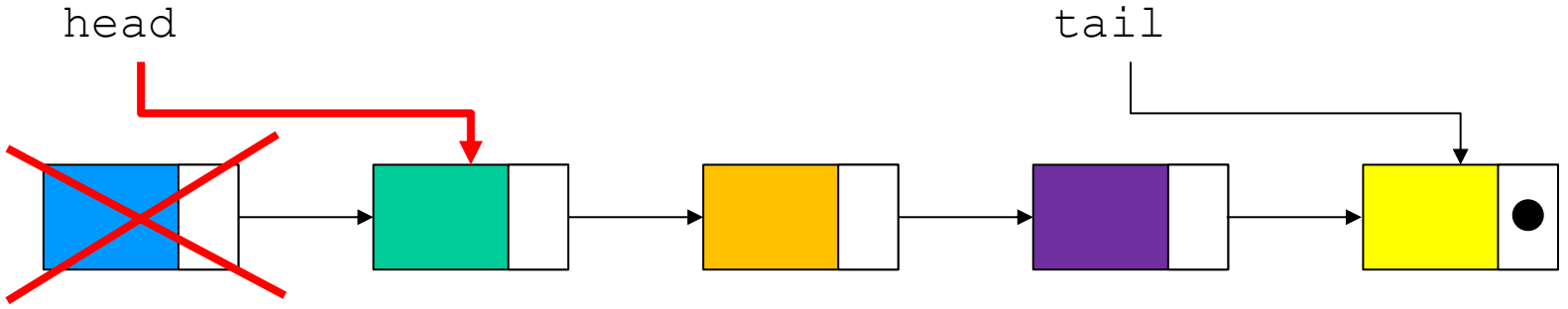
- Η προσθήκη γίνεται στο τέλος της λίστας
- Η απομάκρυνση γίνεται από την αρχή της λίστας
- Εκτός από τον δείκτη στην αρχή της λίστας, έχουμε και έναν δείκτη **στον τελευταίο κόμβο**
  - για να μην ψάχνουμε κάθε φορά για το τέλος της λίστας
- Κατάλληλη διαχείριση του επιπλέον δείκτη

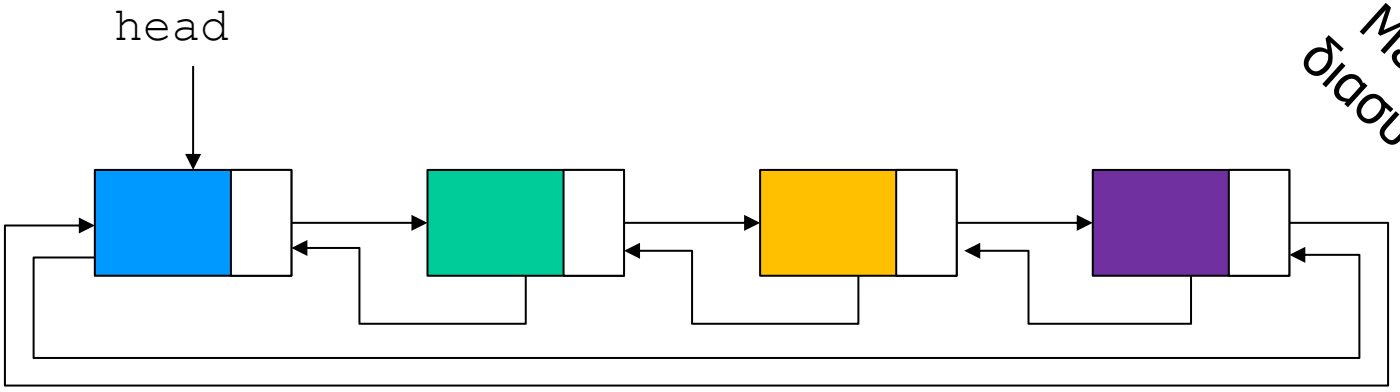


προσθήκη



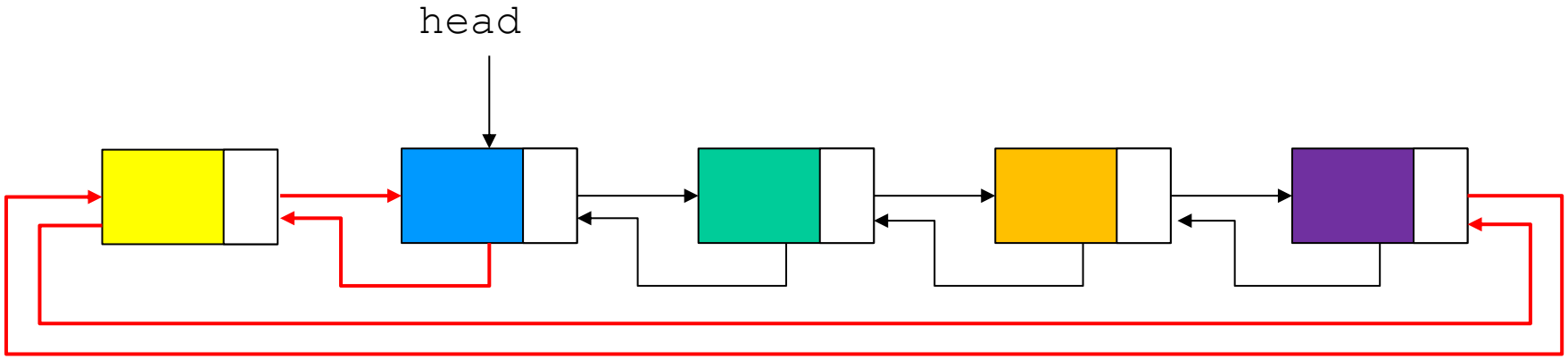
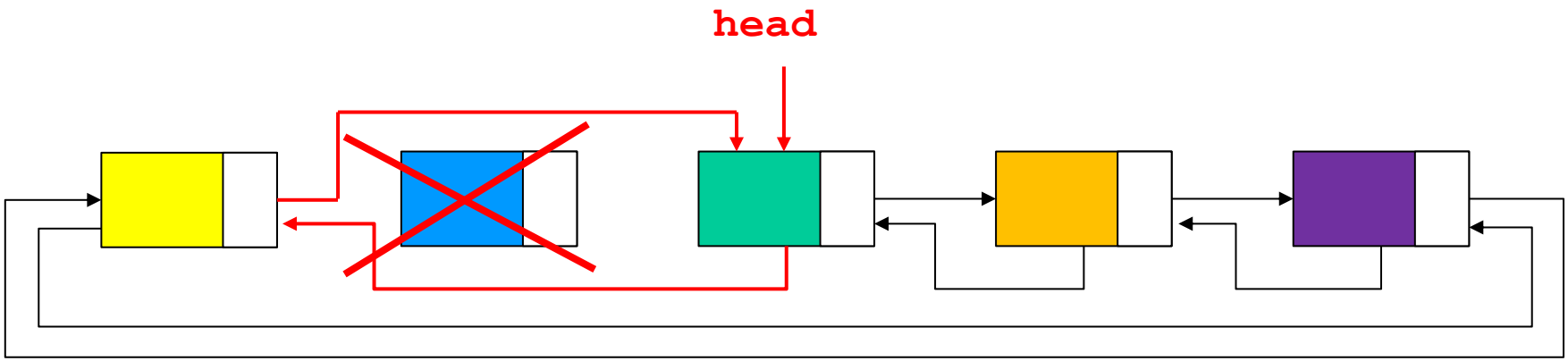
απομάκρυνση





Με κυκλική, διπλά διασυνδεμένη λίστα

προσθήκη



απομάκρυνση