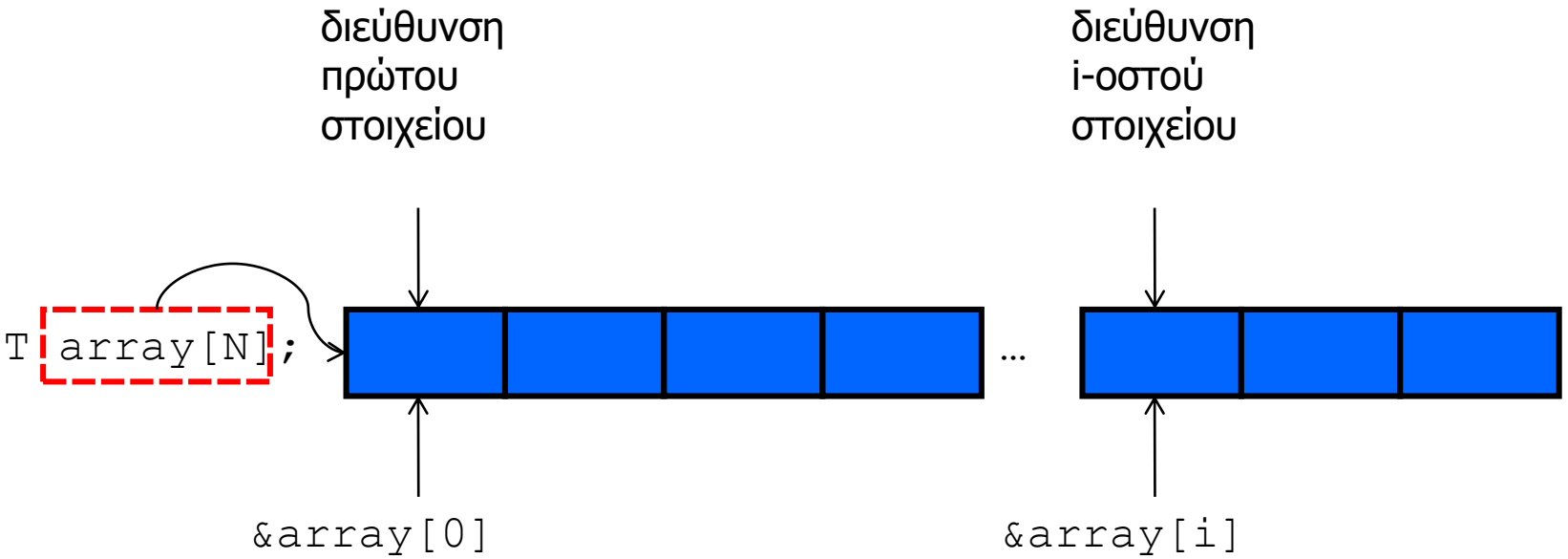
Διασυνδεδεμένες Δομές

Λίστες

Διασυνδεδεμένες δομές

- Η μνήμη ενός πίνακα δεσμεύεται **συνεχόμενα**
  - η πρόσβαση στο i-οστό στοιχείο είναι **άμεση**
    καθώς η διεύθυνση του είναι γνωστή εκ των προτέρων
    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
  - ουσιαστικά **δεν** υφίσταται «i-οστό» στοιχείο
  - αυτή η ορολογία ισχύει (απόλυτα) μόνο για πίνακες

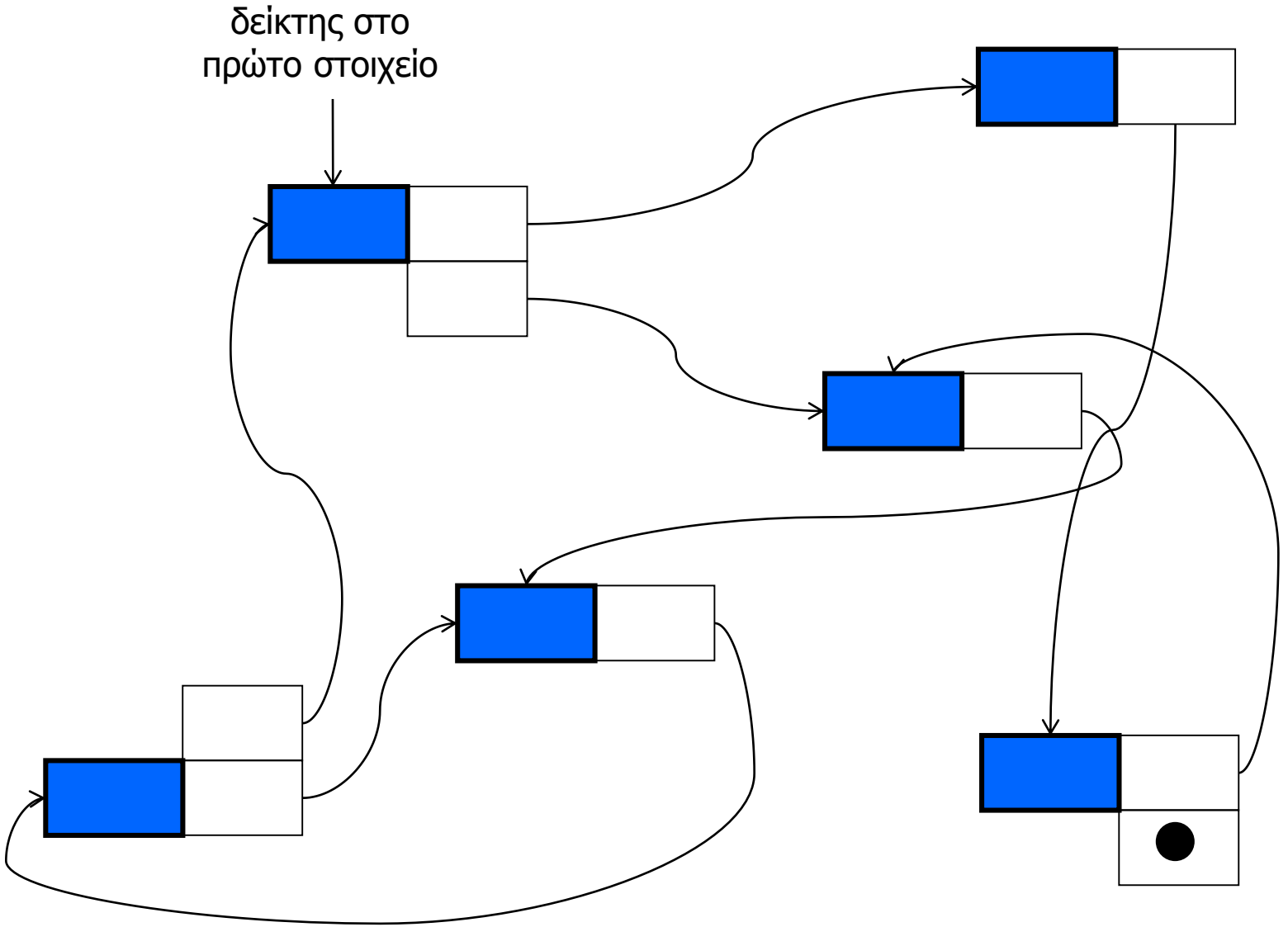


είναι

`array`

είναι

`array + i*sizeof(T)`


δείκτης στο
πρώτο στοιχείο

Κατασκευή διασυνδεδεμένων δομών

- Οι διασυνδεδεμένες δομές κατασκευάζονται **σταδιακά**, κόμβο προς κόμβο
  - όχι μονομιάς, όπως π.χ. οι πίνακες
- Κάθε νέος κόμβος πρέπει να **συνδεθεί** με τους ήδη υπάρχοντες κόμβους, με κατάλληλο τρόπο
  - έτσι ώστε να είναι δυνατή η προσπέλαση όλων των κόμβων
- Για κάθε κόμβο που απομακρύνεται, πρέπει να **προσαρμοστούν** οι διασυνδέσεις των υπολοίπων
- Κάθε δομή έχει ένα συγκεκριμένο (ίσως διαφορετικό) **τρόπο προσπέλασης**, σύμφωνα με τον οποίο
  - ορίζονται οι δείκτες διασύνδεσης των κόμβων
  - υλοποιούνται οι βασικές λειτουργίες αναζήτησης, προσθήκης, απομάκρυνσης

Αναδρομικές/επαναλαμβανόμενες δομές

- Μια από τις κλασικές μορφές διασυνδεδεμένων δομών είναι οι λεγόμενες **αναδρομικές** δομές
  - όταν ένα από τα πεδία της είναι δείκτης σε δομή ιδίου τύπου

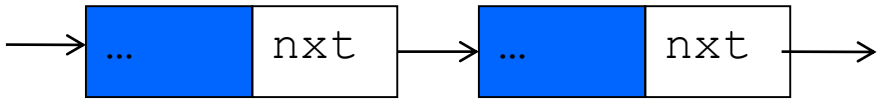
- Οι αναδρομικές δομές κατασκευάζονται με επανάληψη του ίδιου δομικού στοιχείου (κόμβου)

- 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

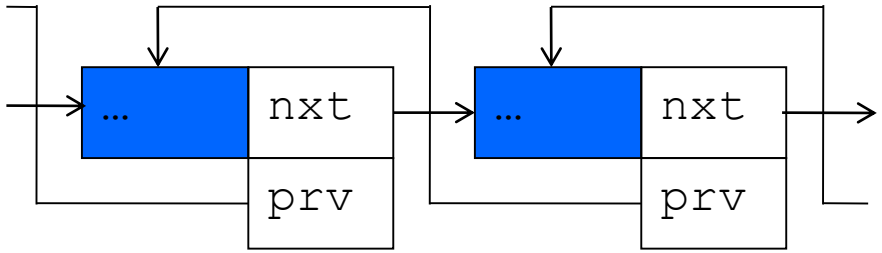
Αναζήτηση/διέλευση

- Η διεύθυνση ενός συγκεκριμένου κόμβου της δομής **δεν** μπορεί να υπολογιστεί με βάση την αρχή της δομής (την διεύθυνση του πρώτου κόμβου της)
  - οι κόμβοι βρίσκονται σε μη συνεχόμενες θέσεις της μνήμης
- Ο μόνος τρόπος να εντοπιστεί ένας συγκεκριμένος κόμβος (ή να διαπιστωθεί ότι δεν υπάρχει) είναι να **διανυθεί** η δομή, κόμβο προς κόμβο, μέχρι να ανακαλύψουμε τον επιθυμητό κόμβο (αν υπάρχει)
- Η μετάβαση από το «προηγούμενο» κόμβο στον «επόμενο» γίνεται σύμφωνα με τους κανόνες διασύνδεσης της εκάστοτε δομής

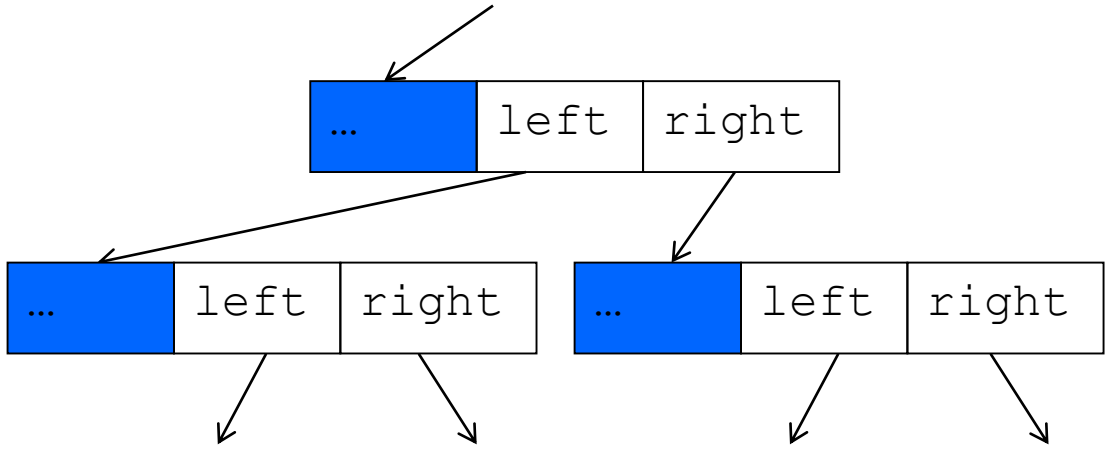
```
struct list {
  … /* data */
  struct list *nxt;
};
```



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



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



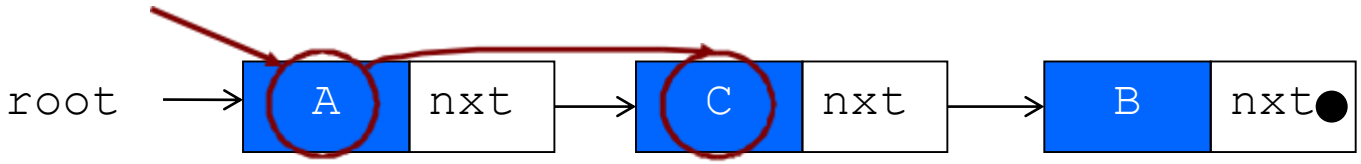
Απλά συνδεδεμένη λίστα

Ενδεικτική υλοποίηση

- Κάθε κόμβος περιέχει ένα **δείκτη** που αρχικοποιείται έτσι ώστε να δείχνει στον **επόμενο** κόμβο
- Η αρχή της λίστας (δείκτης στο πρώτο κόμβο) αποθηκεύεται σε μια (καθολική) μεταβλητή
- **Εισαγωγή:** ο νέος κόμβος εισάγεται ως πρώτος κόμβος της λίστας (δεν ελέγχουμε για διπλές τιμές)
- **Αναζήτηση:** αρχίζουμε από τον πρώτο κόμβο και διασχίζουμε την λίστα, κόμβο προς κόμβο, μέχρι να βρούμε τον κόμβο με την επιθυμητή τιμή
- **Απομάκρυνση:** ο κόμβος εντοπίζεται με αναζήτηση και παρακάμπτεται, συνδέοντας τον προηγούμενο κόμβο με τον επόμενο

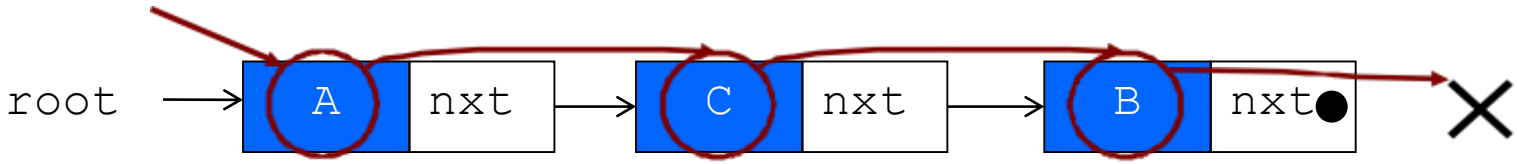
Αναζήτηση (επιτυχημένη)

find C



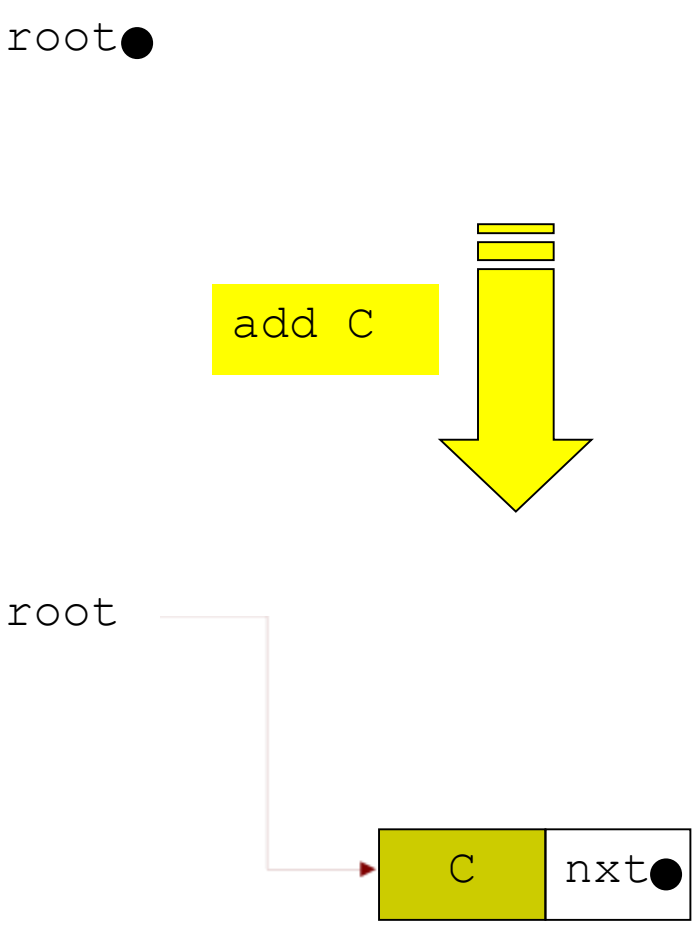
Αναζήτηση (αποτυχημένη)

```
find D
```

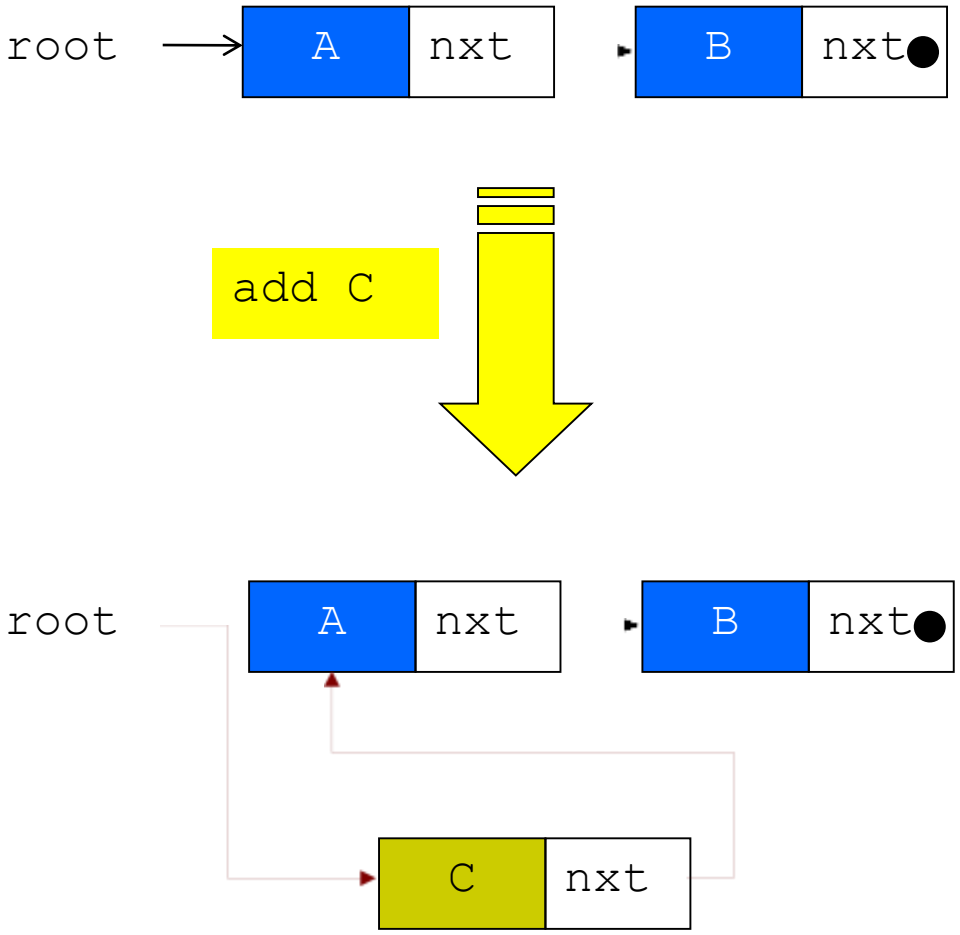


```
struct list {
  int v;
  struct list *nxt;
};

struct list *root;

void list_init() {
  root = NULL;
}

int list_hasElement(int v) {
  struct list *curr;

  for(curr=root; (curr != NULL) && (curr->v != v);
                                    curr=curr->nxt);
  return(curr != NULL);
}
```

Εισαγωγή (σε κενή λίστα)

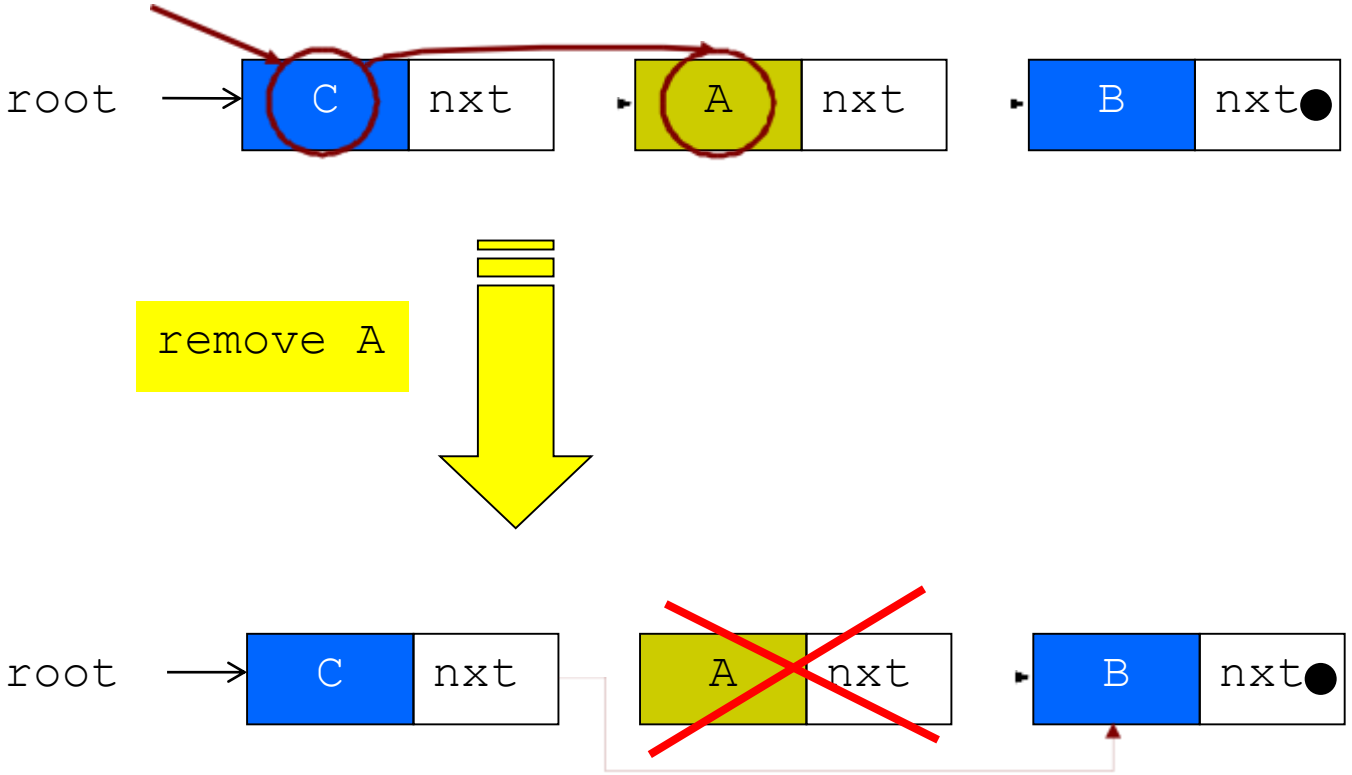


Εισαγωγή (σε μη κενή λίστα)

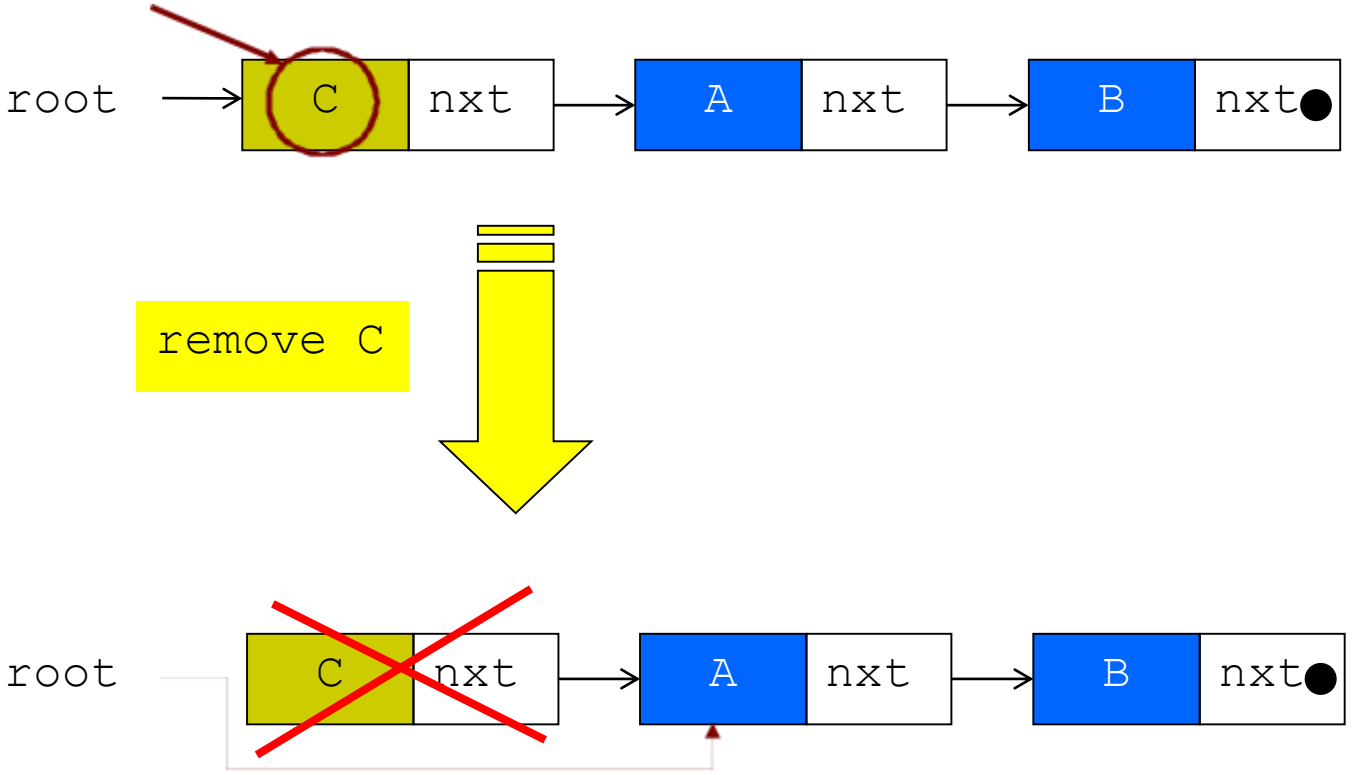


```
void list_insert(int v) {
  struct list *curr;

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

  curr->v = v;

  /* εισαγωγή νέου κόμβου στην αρχή της λίστας */
  curr->nxt = root;
  root = curr;

}
```

Απομάκρυνση (με προηγούμενο κόμβο)



Απομάκρυνση (χωρίς προηγούμενο κόμβο)



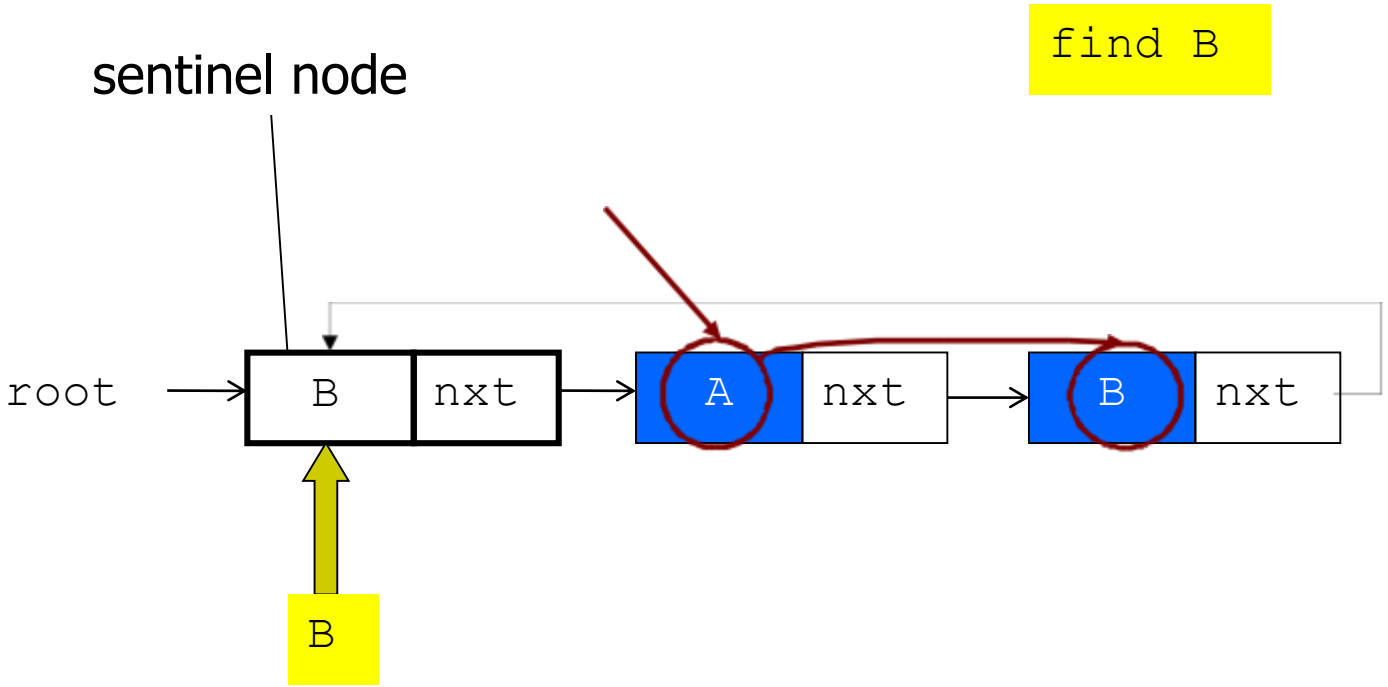
```
void list_remove(int v) {
  struct list *curr,*prev;

  for(prev=NULL,curr=root; (curr != NULL) && (curr->v != v);
                                   prev=curr,curr=curr->nxt);
  if (curr != NULL) {

    if (prev == NULL) {
      /* παράκαμψη πρώτου κόμβου της λίστας */
      root = curr->nxt;
    }
    else {
      /* παράκαμψη κόμβου με προηγούμενο κόμβο */
      prev->nxt = curr->nxt;
    }

    free(curr);
  }
}
```

Κυκλικά συνδεδεμένη λίστα με ειδικό τερματικό κόμβο

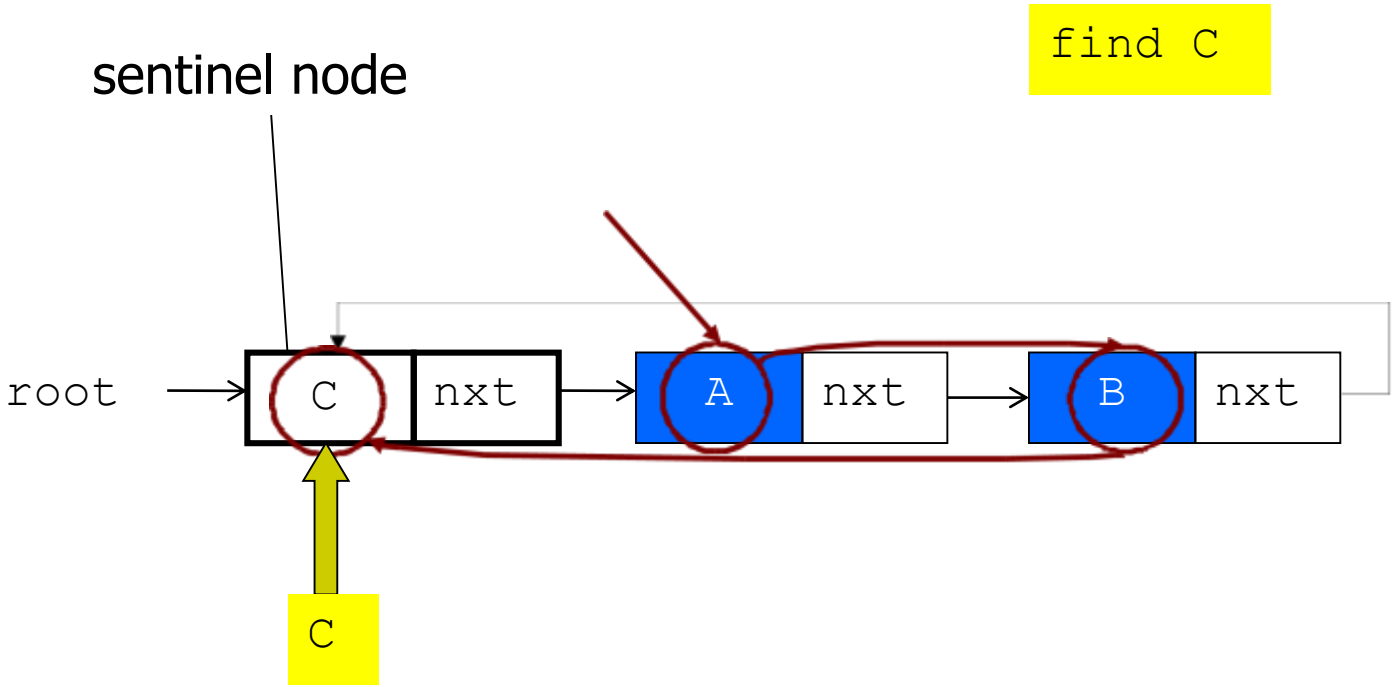
Χρήση τερματικού κόμβου

- Στην απλή λίστα, σε κάθε βήμα της αναζήτησης γίνονται δύο έλεγχοι
  - αν έχουμε φτάσει στο τέλος της λίστας
  - αν βρισκόμαστε στον επιθυμητό κόμβο
- Αν ο κόμβος υπάρχει, κατά μ.ο. χρειάζονται N/2 βήματα (N έλεγχοι), αλλιώς N βήματα (2N έλεγχοι)
- Επίσης, χρειάζεται ειδικός χειρισμός για την περίπτωση της απομάκρυνσης του πρώτου κόμβου
- **Βελτιστοποίηση:** υλοποιούμε τη λίστα ως **κυκλική** λίστα, και χρησιμοποιούμε έναν **άδειο** αρχικό κόμβο (που δεν έχει «πραγματικό» περιεχόμενο δεδομένα) ως **τερματικό κόμβο / sentinel** στην αναζήτηση

Αναζήτηση (πετυχημένη)

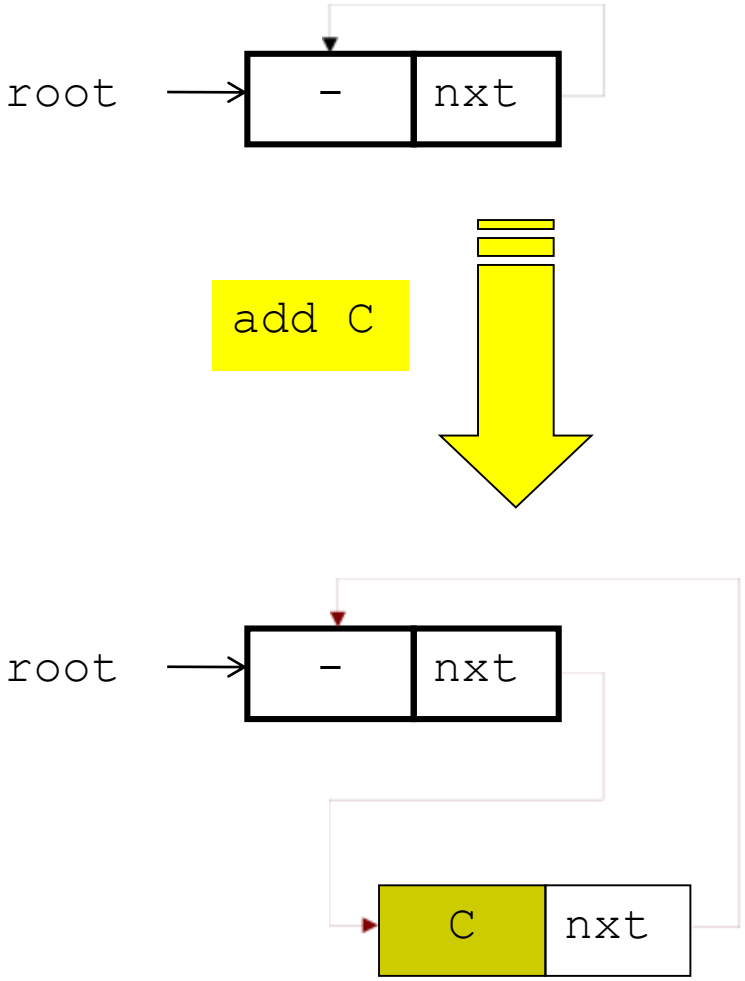


Αναζήτηση (αποτυχημένη)

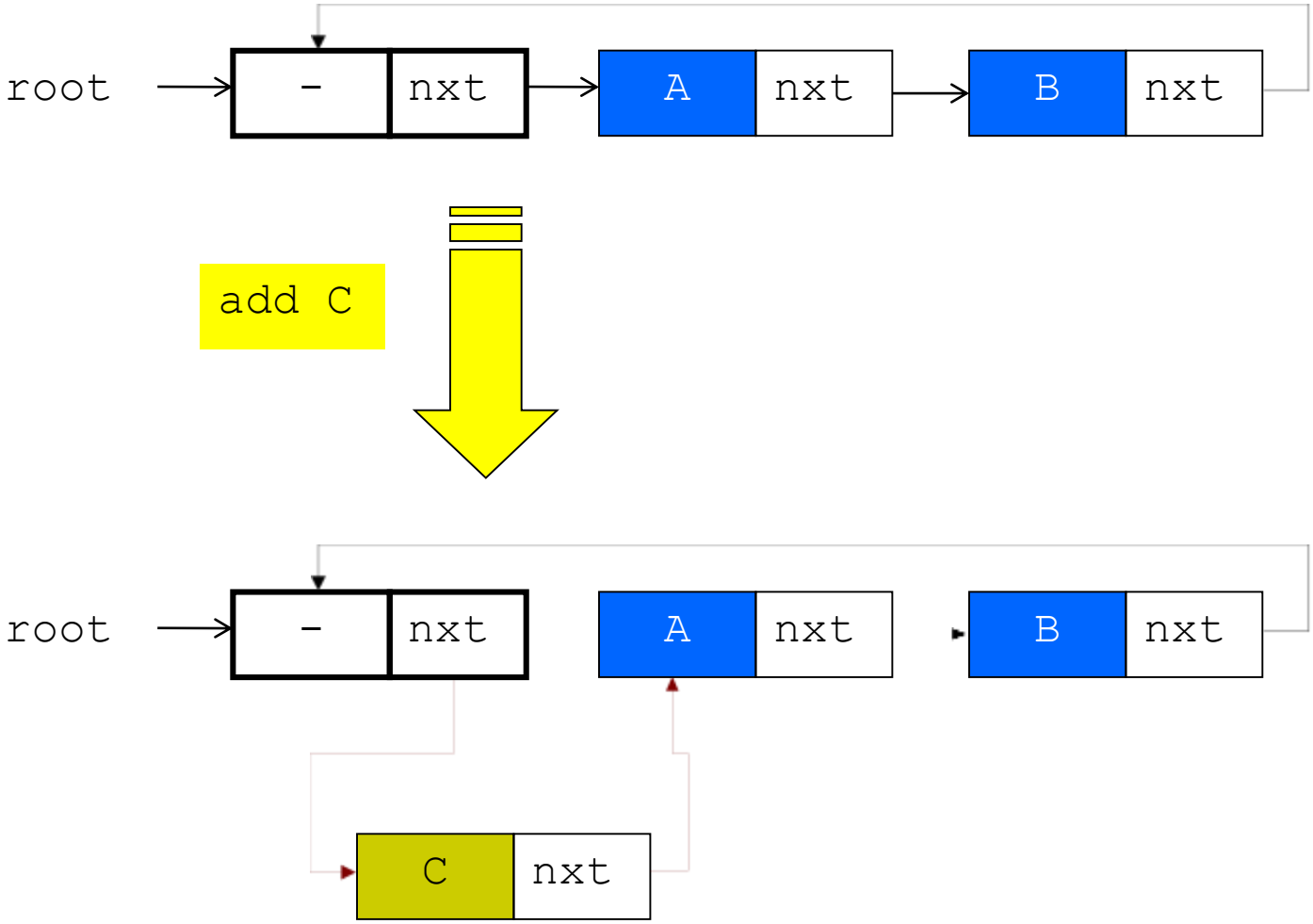


```
struct list {
  int v;
  struct list *nxt;
};

struct list *root;

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

int list_hasElement(int v) {
  struct list *curr;

  root->v = v;
  for(curr=root->nxt; curr->v!=v; curr=curr->nxt);
  return(curr != root);
}
```

Εισαγωγή (σε κενή λίστα)

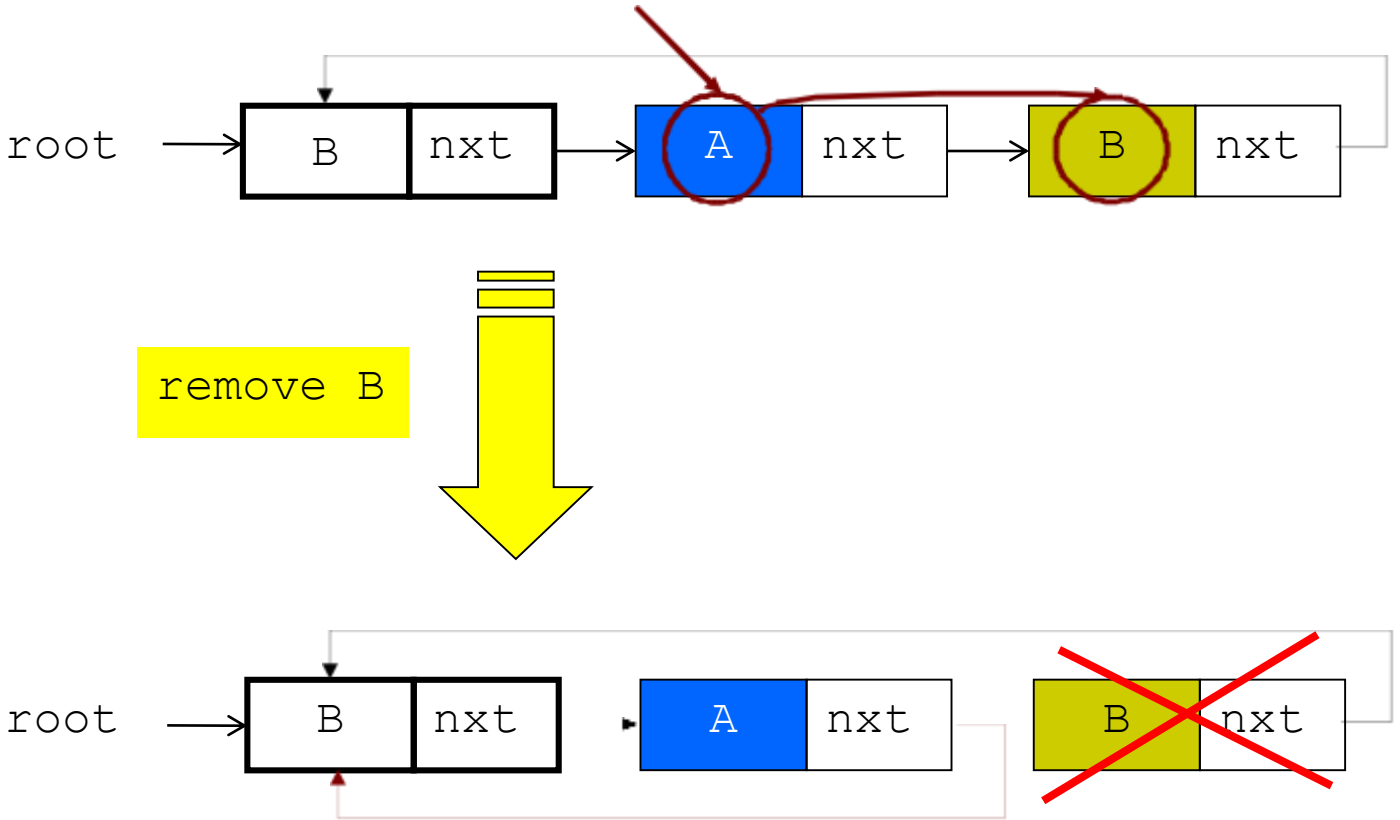


Εισαγωγή (σε μη κενή λίστα)

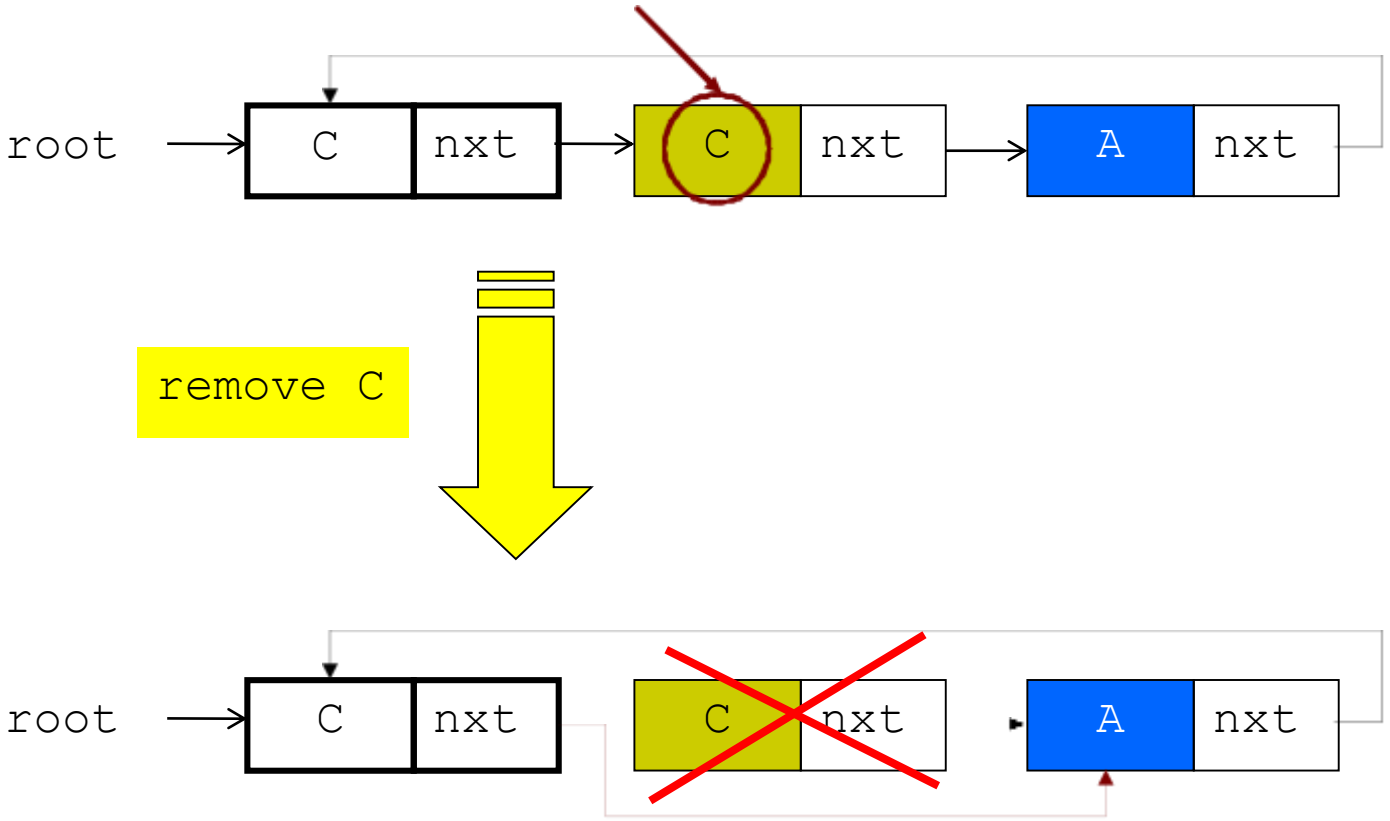


```
void list_insert(int v) {
  struct list *curr;

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

  curr->v = v;

  /* εισαγωγή νέου κόμβου στην αρχή της λίστας */
  curr->nxt = root->nxt;
  root->nxt = curr;

}
```

Απομάκρυνση (με προηγούμενο κόμβο)



Απομάκρυνση (χωρίς προηγούμενο κόμβο)



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

  root->v = v;
  for(prev=root,curr=root->nxt; curr->v!=v;
                                prev=curr,curr=curr->nxt);

  if (curr != root) {
    /* παράκαμψη κόμβου */
    prev->nxt = curr->nxt;
    free(curr);
  }
}
```

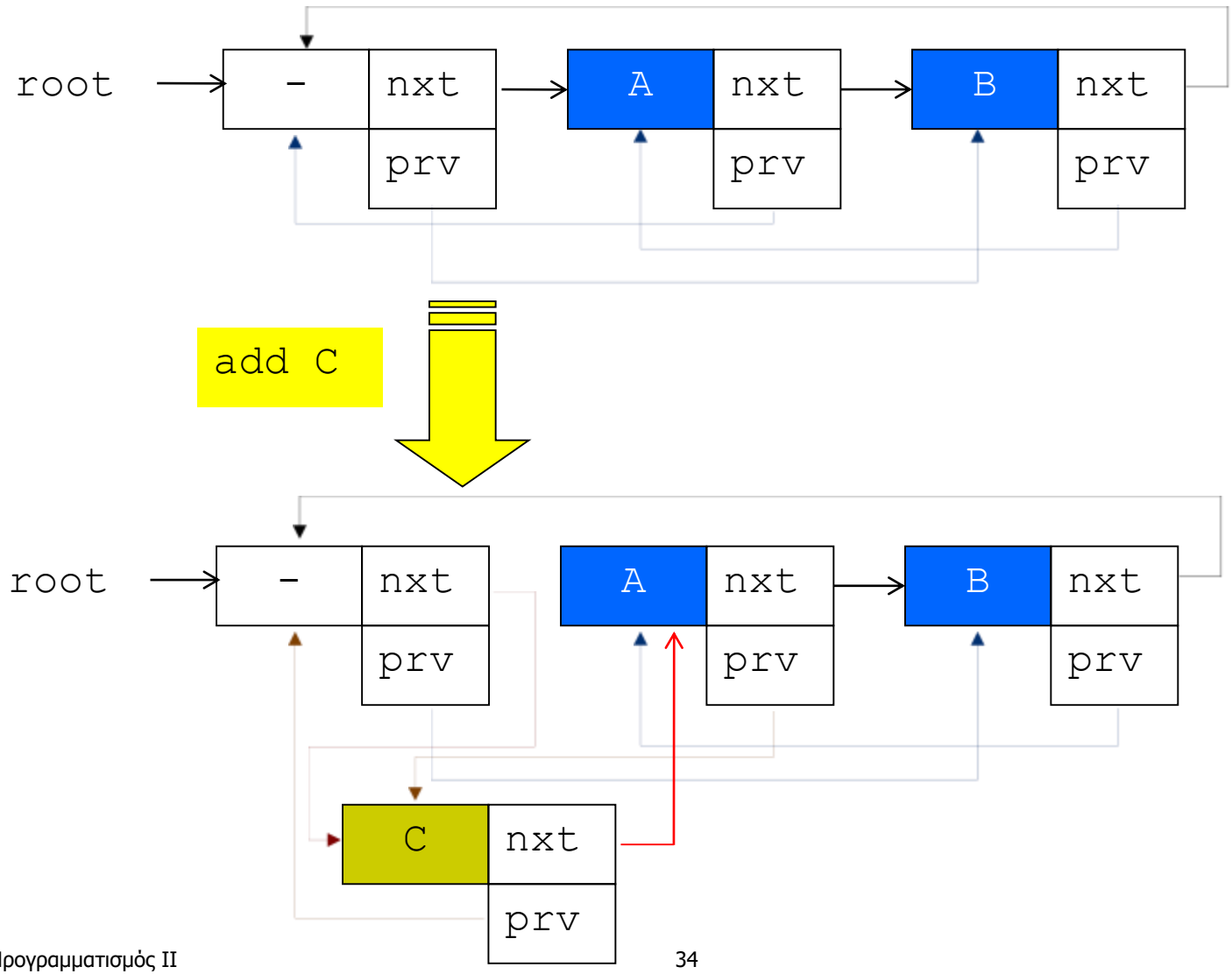
Διπλά συνδεδεμένη λίστα

Απλοποίηση απομάκρυνσης

- Στην απομάκρυνση, η λίστα διασχίζεται με **δύο** δείκτες, έναν στον κόμβο που ελέγχουμε και έναν στον προηγούμενο κόμβο, έτσι ώστε αν εντοπιστεί ο επιθυμητός κόμβος να τον παρακάμψουμε
- Αν η λίστα έχει Ν κόμβους, κατά μέσο όρο θα χρειαστούν Ν/2 βήματα και συνεπώς Ν αναθέσεις
- **Βελτιστοποίηση:** υλοποιούμε τη λίστα ως **διπλά συνδεδεμένη** λίστα, όπου κάθε κόμβος δείχνει στον επόμενο αλλά και στον προηγούμενο του
- Μπορούμε πλέον να απομακρύνουμε έναν κόμβο αρκεί να έχουμε ένα δείκτη σε αυτόν
  - δε χρειάζεται ένας δείκτης στον προηγούμενο κόμβο

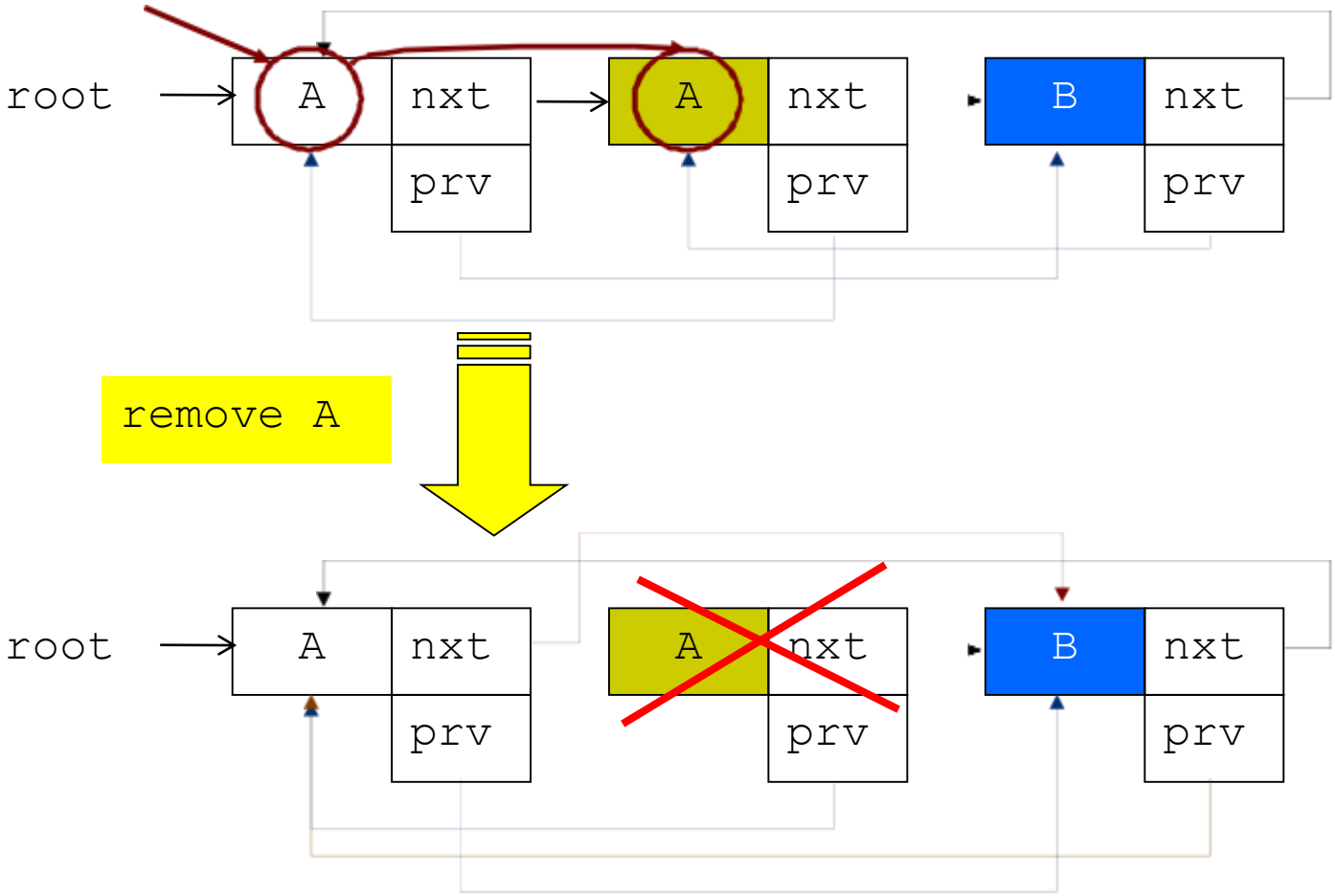
```
struct list2 {
  int v;
  struct list2 *nxt;
  struct list2 *prv;
};

struct list2 *root;

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

int list_hasElement(int v) {
  struct list2 *curr;

  root->v = v;
  for(curr=root->nxt; curr->v!=v; curr=curr->nxt);
  return(curr != root);
}
```

Εισαγωγή


root
-
nxt
prv
A
nxt
prv
B
nxt
prv
add C
root
-
nxt
prv
A
nxt
prv
B
nxt
prv
C
nxt
prv

```
void list_insert(int v) {
  struct list2 *curr;

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

  curr->v = v;

  /* εισαγωγή νέου κόμβου στην αρχή της λίστας */
  curr->nxt = root->nxt;
  curr->prv = root;
  curr->nxt->prv = curr;
  curr->prv->nxt = curr;

}
```

Απομάκρυνση



```
void list_remove(int v) {
  struct list2 *curr;

  root->v = v;
  for(curr=root->nxt; curr->v!=v; curr=curr->nxt);

  if (curr != root) {
    /* παράκαμψη κόμβου */
    curr->nxt->prv = curr->prv;
    curr->prv->nxt = curr->nxt;
    free(curr);
  }

}
```

Σύγκριση

- Απλά συνδεδεμένη λίστα
  - ένα πεδίο δείκτης
  - δύο συγκρίσεις ανά βήμα αναζήτησης
  - δύο αναθέσεις ανά βήμα αναζήτησης για απομάκρυνση
- Κυκλικά συνδεδεμένη λίστα με τερματικό
  - ένα πεδίο δείκτης και ένας τερματικός κόμβος
  - μια σύγκριση ανά βήμα αναζήτησης
  - δύο αναθέσεις ανά βήμα αναζήτησης για απομάκρυνση
- Διπλά συνδεδεμένη κυκλική λίστα με τερματικό
  - δύο πεδία δείκτες και ένας τερματικός κόμβος
  - μια σύγκριση ανά βήμα αναζήτησης
  - μια ανάθεση ανά βήμα αναζήτησης για απομάκρυνση

Ταξινομημένη λίστα

Επιτάχυνση της αναζήτησης

- Όταν αναζητείται ένας κόμβος με συγκεκριμένο περιεχόμενο, πρέπει να ελεγχθούν **όλοι** οι κόμβοι **μέχρι** να **βρεθεί** ο ζητούμενος ή μέχρι να φτάσουμε στο **τέλος** της λίστας
  - αν ο επιθυμητός κόμβος δεν υπάρχει στην λίστα, αναγκαστικά θα διασχισθεί **ολόκληρη** η λίστα
- **Βελτίωση:** ταξινόμηση των κόμβων της λίστας
  - σύμφωνα με κάποιο συνδυασμό των περιεχομένων τους ή ένα **πεδίο-κλειδί** που χρησιμοποιείται για την σύγκριση
- Η εισαγωγή αλλάζει, έτσι ώστε ο νέος κόμβος να τοποθετείται **μετά** από όλους τους κόμβους με «μικρότερες» τιμές, και **πριν** από όλους τους κόμβους με «μεγαλύτερες» τιμές

```
struct list {
  int v;
  struct list *nxt;
};

struct list *root;

void sortedlist_init() {
  root = (struct list *)malloc(sizeof(struct list));
  root->nxt = root;
}

int sortedlist_hasElement(int v) {
  struct list *curr;

  root->v = v;
  for(curr=root->nxt; curr->v<v; curr=curr->nxt);
  return((curr!=root)) && (curr->v==v));
}
```

```
void sortedlist_insert(int v) {
  struct list *curr,*prev;

  root->v = v;
  for(prev=root,curr=root->nxt; curr->v < v;
                                prev=curr,curr=curr->nxt);

  curr = (struct list *)malloc(sizeof(struct list));
  curr->v = v;
  curr->nxt = prev->nxt; prev->nxt = curr;
}
```

```
void sortedlist_remove(int v) {
  struct list *curr,*prev;

  root->v = v;
  for(prev=root,curr=root->nxt; curr->v < v;
                                prev=curr,curr=curr->nxt);
  if ((curr != root) && (curr->v == v)) {
    prev->nxt = curr->nxt;
    free(curr);
  }
}
```

Ταξινομημένη vs. μη-ταξινομημένη λίστα

- Η μεγάλη διαφορά στην αναζήτηση υπάρχει όταν ο επιθυμητός κόμβος **δεν** βρίσκεται στην λίστα
- Με ταξινομημένη λίστα χρειάζονται κατά **μέσο όρο N/2** βήματα, ενώ με μη-ταξινομημένη λίστα χρειάζονται **πάντα N** βήματα
- Αν ο κόμβος υπάρχει, και οι δύο μέθοδοι απαιτούν κατά **μέσο όρο N/2** βήματα – σκεφτείτε γιατί!
- Υπάρχει όμως κόστος: στην ταξινομημένη λίστα, πρέπει να γίνει **αναζήτηση** κατά την εισαγωγή
- Αν όμως δεν επιτρέπονται «διπλοί» κόμβοι, τότε η εισαγωγή υποχρεωτικά συμπεριλαμβάνει αναζήτηση (και στην μη-ταξινομημένη λίστα), οπότε η χρήση ταξινομημένης λίστας γίνεται **ακόμα** πιο ελκυστική

Λίστα vs. δυναμικός πίνακας

- Αν ο πίνακας περιέχει τα στοιχεία με τα δεδομένα, οι λειτουργίες προσθήκης/αφαίρεσης μπορεί να είναι (πολύ) πιο αργές από ότι σε μια λίστα, λόγω αντιγραφής / μετακίνησης των στοιχείων
- Αν ο πίνακας περιέχει δείκτες στα στοιχεία με τα δεδομένα, η απόδοση είναι παρόμοια με την λίστα
- Σε πίνακα με ταξινομημένα στοιχεία, η αναζήτηση μπορεί να υλοποιηθεί πιο αποδοτικά με δυαδική αναζήτηση (binary search)
  - γιατί δεν μπορεί να γίνει σε λίστα;
- Σε δυναμικό πίνακα μπορεί μπορεί να γίνεται «σιωπηρή» αντιγραφή δεδομένων κάθε φορά που αλλάζει το μέγεθος του (βλέπε `realloc`)