Προγραμματισμός Ι (HY120)

Διάλεξη 15:
Διασυνδεμένες Δομές - Λίστες

Χρήστος Δ. Αντωνόπουλος
11/12/2017

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

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 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.
  - Σημείωση: ακριβώς για αυτό το λόγο, σε αυτές τις δομές δεν υφίσταται η έννοια «i-οστό» στοιχείο.

διεύθυνση
πρώτου
στοιχείου

διεύθυνση
i-οστού
στοιχείου

```
T array[N];
```

…

`&array[0]`

είναι

`array`

`&array[i]`

είναι

`array + i*sizeof(T)`

διεύθυνση
πρώτου
στοιχείου

Προσθήκη και απομάκρυνση στοιχείων

- 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.

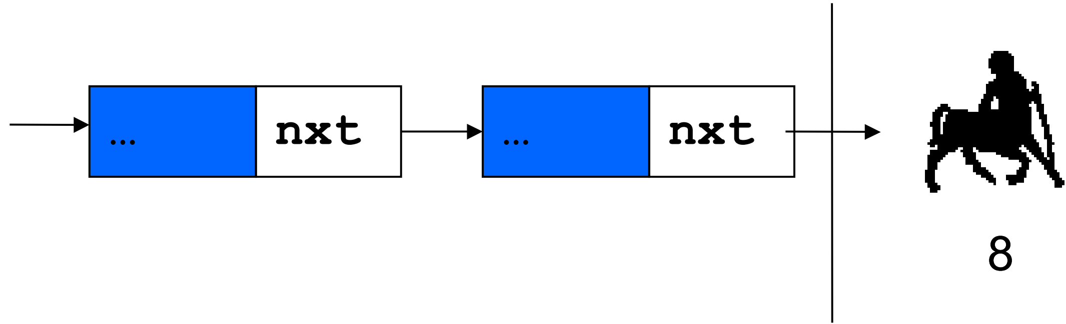
Αναδρομικές / επαναλαμβανόμενες δομές

- Μια από τις κλασικές μορφές διασυνδεδεμένων δομών είναι οι «αναδρομικές» δομές.
  - Μια δομή ονομάζεται αναδρομική όταν ένα από τα πεδία της είναι ένας δείκτης σε δομή ιδίου τύπου.
  - Οι αναδρομικές δομές κατασκευάζονται μέσα από την επανάληψη του ίδιου **δομικού στοιχείου** (κόμβου).
- Οι διασυνδέσεις μεταξύ των κόμβων (δηλαδή η αρχικοποίηση των πεδίων δεικτών) γίνεται σύμφωνα με τους κανόνες που διέπουν την συγκεκριμένη δομή, **και οι οποίοι διαφέρουν ανά περίπτωση**.

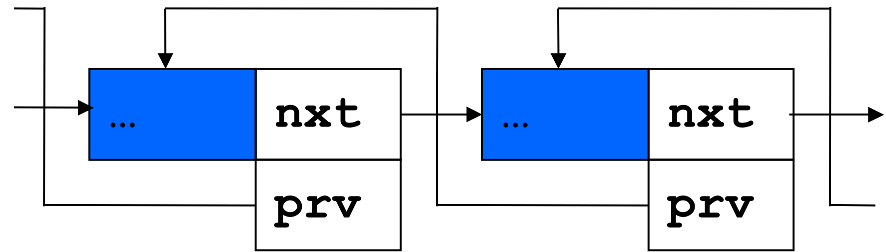
Αναζήτηση / διέλευση

- Η διεύθυνση ενός συγκεκριμένου κόμβου της δομής **δε** μπορεί να υπολογιστεί με βάση την αρχή της δομής (την διεύθυνση του πρώτου κόμβου της).
  - Ο λόγος είναι ότι οι κόμβοι είναι αποθηκευμένοι 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.

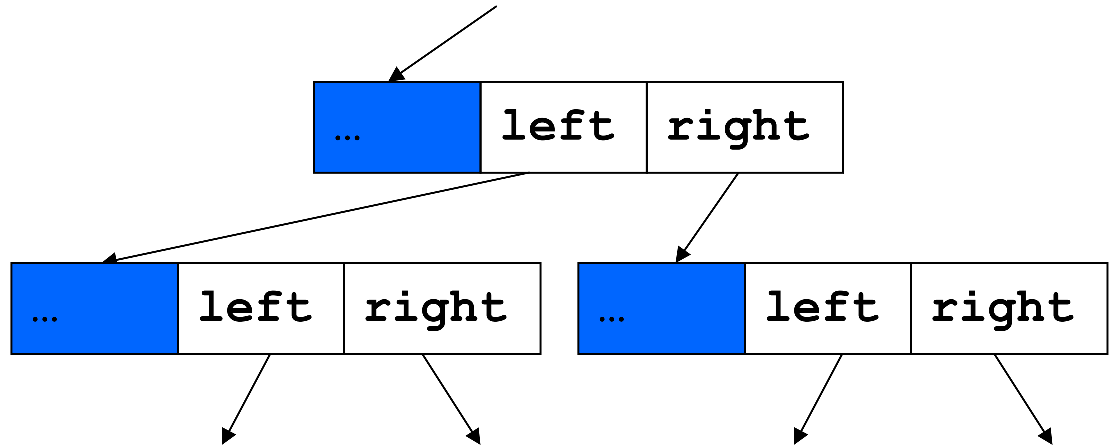
```
struct list {
  … /* δεδομένα */
  struct list *nxt;
};
```




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




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



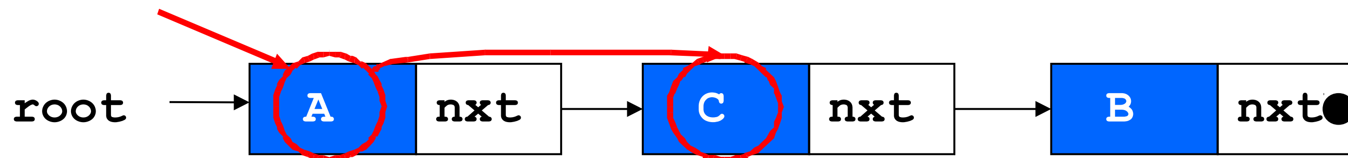
Λίστες

Ενδεικτική υλοποίηση απλής λίστας

- Κάθε κόμβος περιέχει ένα δείκτη που αρχικοποιείται έτσι ώστε να δείχνει στον επόμενο κόμβο.
- Η αρχή της λίστας (δείκτης στο πρώτο κόμβο) αποθηκεύεται σε κατάλληλη (καθολική) μεταβλητή, στην οποία έχει πρόσβαση ο κώδικας των πράξεων.
- **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».

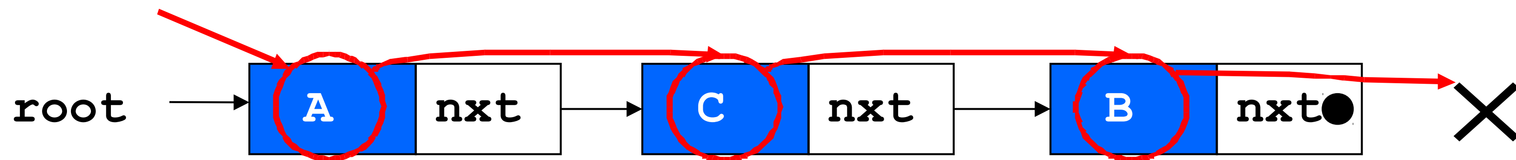
Αναζήτηση (επιτυχημένη)

find C



Αναζήτηση (αποτυχμένη)

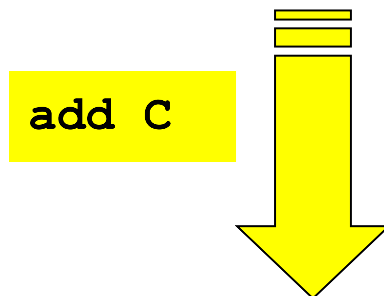
find D

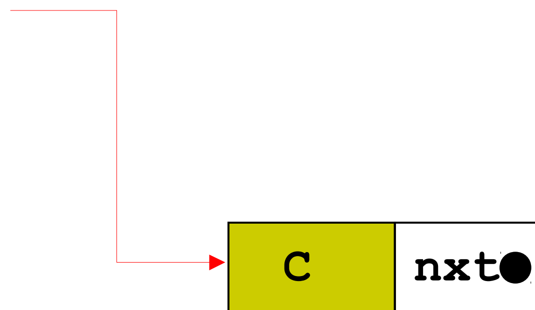


```
struct list {
  int v;
  struct list *nxt;
};

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);
}
```

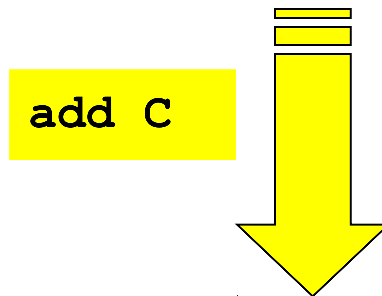
Εισαγωγή (σε κενή λίστα)

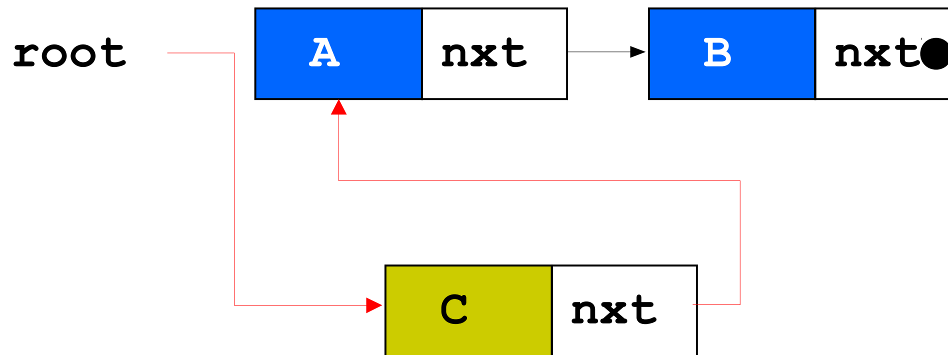
root●





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 στοιχείο)

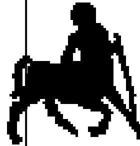
root → C | nxt → A | nxt → B | nxt●

remove A

root → C | nxt → B | nxt●   (A | nxt removed)

Απομάκρυνση (χωρίς προηγούμενο στοιχείο)

```
root → [C | nxt] → [A | nxt] → [B | nxt●]

remove C

root → [C | nxt] (crossed out)
root ─────────────→ [A | nxt] → [B | nxt●]
```

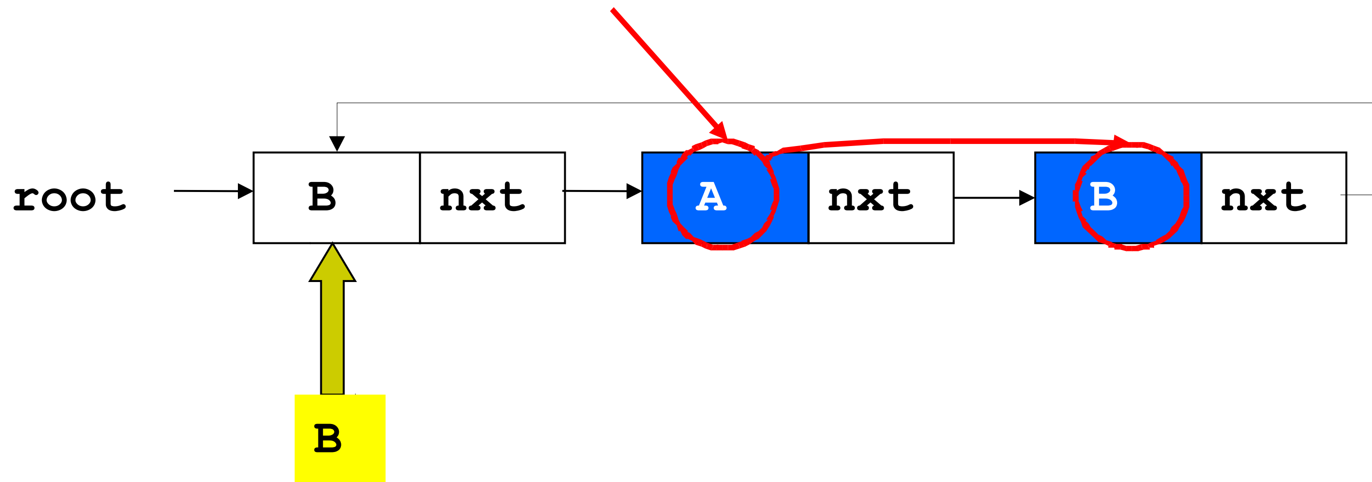



```
void list_remove(int v) {
  struct list *curr,*prev;

  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);
  }
}
```

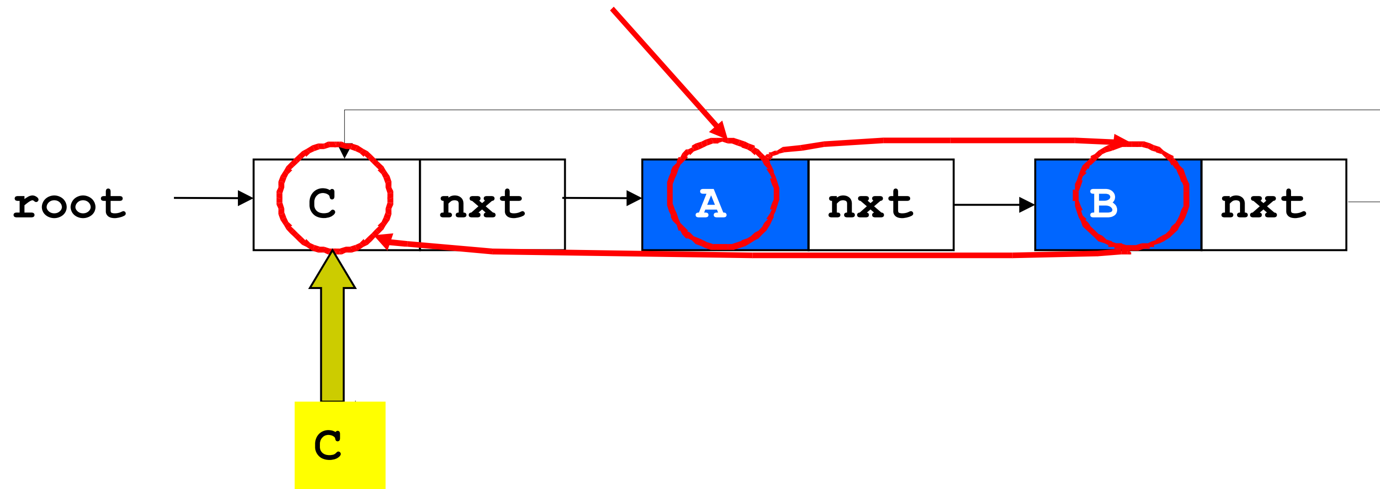
Κυκλικά συνδεδεμένη λίστα με τερματικό

- Σε κάθε βήμα της αναζήτησης γίνονται δύο έλεγχοι,
  - Κατά πόσο έχουμε φτάσει στο τέλος της λίστας και
  - Κατά πόσο εντοπίσαμε τον επιθυμητό κόμβο.
- Αν η λίστα έχει N κόμβους, κατά μέσο όρο θα χρειαστούν N/2 βήματα και συνεπώς N έλεγχοι.
- Παρατήρηση: Ειδική περίπτωση της απομάκρυνσης του πρώτου κόμβου. Μπορούμε να απλοποιήσουμε τη λειτουργία της απομάκρυνσης;
  - Μείωση Πολυπλοκότητας: υλοποιούμε τη λίστα ως **κυκλική** λίστα, και χρησιμοποιούμε έναν **άδειο** αρχικό κόμβο που δεν έχει «πραγματικά» δεδομένα και χρησιμεύει σαν **τερματικός** κόμβος (sentinel) στην αναζήτηση.

Αναζήτηση (επιτυχημένη)

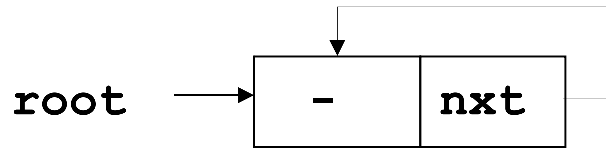


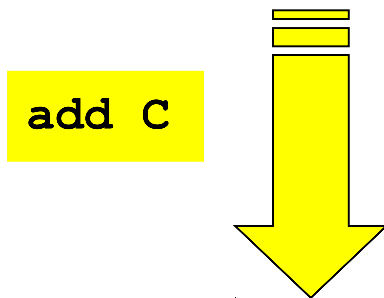
Αναζήτηση (αποτυχημένη)

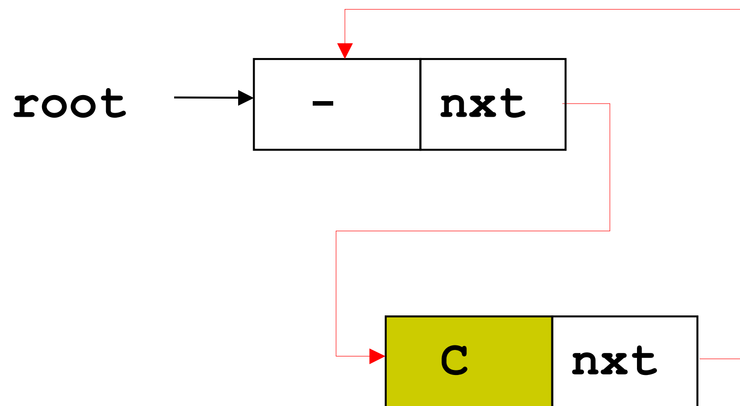


```
struct list {
  int v;
  struct list *nxt;
};

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);
}
```

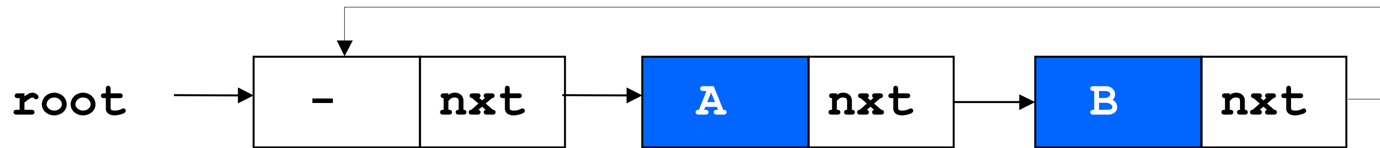
Εισαγωγή (σε κενή λίστα)



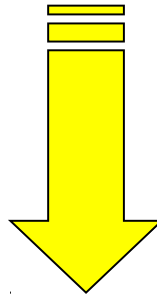


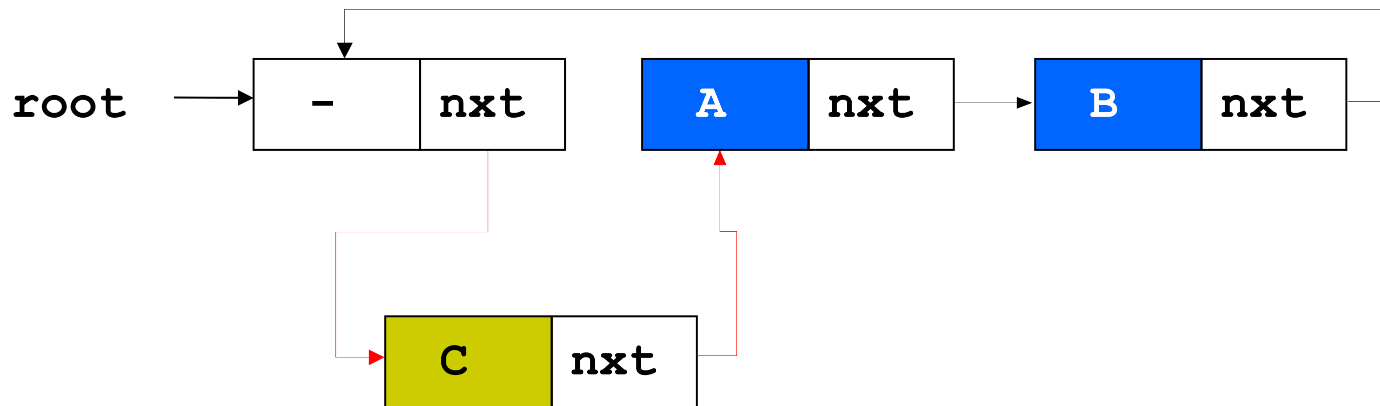


Εισαγωγή (σε μη κενή λίστα)



add C





```
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 στοιχείο)

Απομάκρυνση (χωρίς προηγούμενο στοιχείο)

root → [C | nxt] → [C | nxt] → [A | nxt]

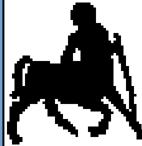
remove C

root → [C | nxt] → [A | nxt]

```
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 με προηγούμενο κόμβο */
    prev->nxt = curr->nxt;

    free(curr);
  }
}
```

Διπλά συνδεδεμένη κυκλική λίστα

- Στην απομάκρυνση, η λίστα διασχίζεται ανανεώνοντας σε κάθε βήμα **δύο** δείκτες, έναν στον κόμβο που ελέγχουμε και ένα στον αμέσως 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, οπότε η λίστα μπορεί να διανυθεί ανανεώνοντας ένα δείκτη.
- Μπορούμε πλέον να απομακρύνουμε έναν κόμβο αρκεί να έχουμε ένα δείκτη σε αυτόν.
  - Δε χρειάζεται να «θυμόμαστε τον προηγούμενο



```
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 με προηγούμενο κόμβο */
    curr->nxt->prv = curr->prv;
    curr->prv->nxt = curr->nxt;

    free(curr);
  }

}
```

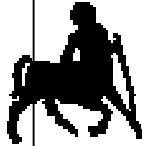
Σύγκριση

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

Ταξινομημένες λίστες

- Όταν αναζητείται ένας κόμβος με συγκεκριμένο περιεχόμενο πρέπει να ελεγχθούν όλοι οι κόμβοι της λίστας μέχρι αυτός να βρεθεί (ή να φτάσουμε στο τέλος της λίστας).
  - Αν ο επιθυμητός κόμβος δεν υπάρχει στην λίστα, αναγκαστικά θα διασχισθεί ολόκληρη η λίστα.
- Η αναζήτηση μπορεί να βελτιωθεί αν οι κόμβοι της λίστας είναι ταξινομημένοι
  - Σύμφωνα με κάποιο συνδυασμό των περιεχομένων τους
    - Συνήθως ένα πεδίο **κλειδί** που χρησιμοποιείται για σύγκριση.
- Η υλοποίηση της **εισαγωγής 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 και μη ταξινομημένη λίστα

- Η διαφορά της αναζήτησης ανάμεσα σε ταξινομημένη και μη ταξινομημένη λίστα είναι στην περίπτωση που ο επιθυμητός κόμβος **δεν** βρίσκεται στην λίστα.
  - Σε αυτή τη περίπτωση η αναζήτηση σε ταξινομημένη λίστα απαιτεί **κατά μέσο** όρο N/2 βήματα, ενώ σε μη ταξινομημένη λίστα απαιτεί **εγγυημένα** N βήματα.
- Αν ο επιθυμητός κόμβος υπάρχει στην λίστα, τότε και οι δύο μέθοδοι απαιτούν κατά μέσο όρο N/2 βήματα.
- Τι πληρώνω;
  - Αναζήτηση για κάθε εισαγωγή
  - Αν η λίστα **δεν** επιτρέπεται να έχει **διπλούς** κόμβους (με τα ίδια περιεχόμενα ή πεδίο κλειδί με ίδια τιμή), τότε η εισαγωγή υποχρεωτικά συμπεριλαμβάνει και **αναζήτηση**, οπότε η χρήση ταξινομημένης λίστας γίνεται ακόμα πιο ελκυστική.

Γρήγορη αναζήτηση σε ταξινομημένη λίστα;

- Αντίθετα με ένα ταξινομημένο πίνακα, δεν μπορεί να γίνει δυαδική αναζήτηση σε μια ταξινομημένη λίστα.
  - Δεν υπάρχει απ' ευθείας πρόσβαση στον i-οστό κόμβο
  - Το κόστος για να φτάσουμε στον i-οστό κόμβο είναι **ήδη** αυτό μιας γραμμικής αναζήτησης.
- Παρατήρηση: μια δομή που μπορεί να χρησιμοποιηθεί για δυαδική αναζήτηση καθώς και την ευέλικτη διαχείριση μνήμης με μικρό κόστος «μετακίνησης» δεδομένων, είναι ένας δυναμικός πίνακας από δείκτες.
- Υπάρχει μια άλλη διασυνδεδεμένη δομή μέσω της οποίας μπορεί να επιτευχθεί γρήγορη αναζήτηση…