Υλοποίηση Λιστών

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

Ευθύγραμμές Απλά και Διπλά Συνδεδεμένες Λίστες

Κυκλικές Απλά και Διπλά Συνδεδεμένες Λίστες

Τεχνικές Μείωσης Μνήμης

Λίστες

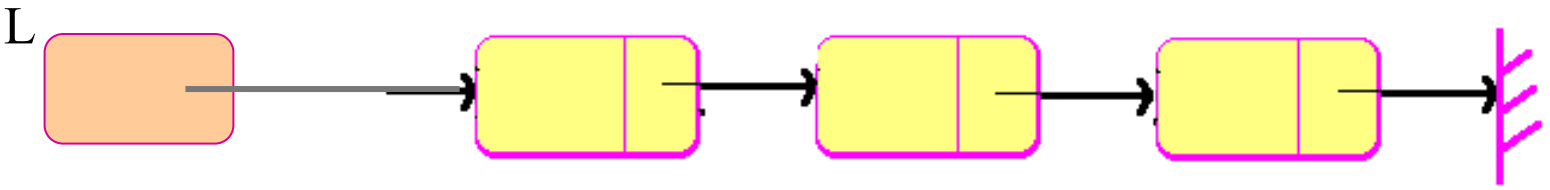
- Ο ΑΤΔ λίστα ορίζεται ως μια ακολουθία στοιχείων συνοδευόμενη από πράξεις που επιτρέπουν εισαγωγή και εξαγωγή στοιχείων σε οποιαδήποτε θέση της λίστας.
- Οι βασικές πράξεις περιλαμβάνουν τις εξής:

Concatenate(L_1, L_2)	δημιούργησε μια νέα λίστα που περιέχει τα στοιχεία της λίστας L_1 ακολουθούμενα από τα στοιχεία της L_2
Access(L, i)	επέστρεψε το i-οστό στοιχείο της L
Sublist (L,i, j)	επέστρεψε τη λίστα που ξεκινά από το i-οστό και τελειώνει στο j-οστό στοιχείο της L.
Insert_After(L, x, i)	εισήγαγε το x μετά από το i-οστό στοιχείο της L.
Delete(L, i)	αφαίρεσε το i-οστό στοιχείο της L

- Στη συνέχεια θα μελετήσουμε υλοποιήσεις του ΑΤΔ με δυναμική χορήγηση μνήμης.

Ευθύγραμμες Απλά Συνδεδεμένες Λίστες

- Μία λίστα μπορεί να υλοποιηθεί ως μια συνδεδεμένη λίστα (με παρόμοιο τρόπο όπως μια στοίβα).



- Πιθανές δηλώσεις κόμβων είναι:

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

```
typedef struct list {
NODE  *top;
. . .
} LIST;
```

Ευθύγραμμες Απλά Συνδεδεμένες Λίστες

- Πιο κάτω ορίζονται κάποιες χρήσιμες πράξεις.
- *Εύρεση Κόμβου με συγκεκριμένο στοιχείο*:

```
bool find(LIST *L, key val){
    return findnode((*L).top, val);
}

bool findnode(NODE *P, key val){
for ( Q = P; Q != NULL; Q = (*Q).next);
    if ((*Q).data == val)
          return TRUE;
return FALSE;
}
```

- Αναδρομική Εκδοχή της findnode;;

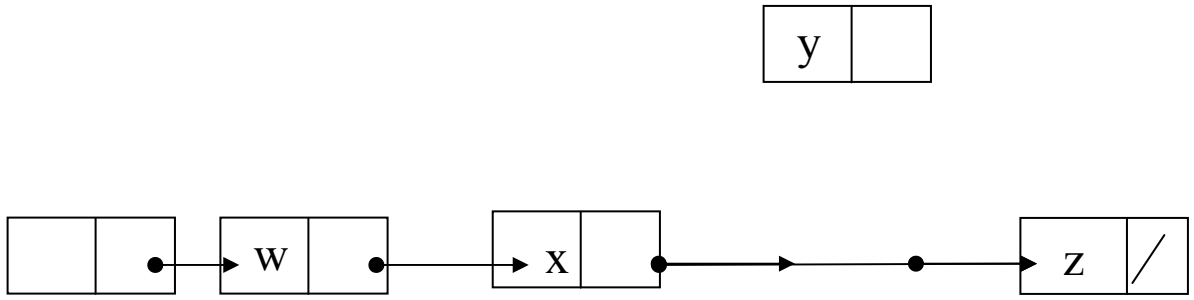
Ευθύγραμμες Απλά Συνδεδεμένες Λίστες

```
bool rec-find(LIST *L, key val){
  return rec-findnode((*L).top, val);
}

bool rec-findnode(NODE *P, key val){
   if (P == NULL)
       return FALSE;
   else
       if ((*P).data == val)
          return TRUE;
       else
          return rec-findnode((*P).next, val);
}
```

Ευθύγραμμες Απλά Συνδεδεμένες Λίστες

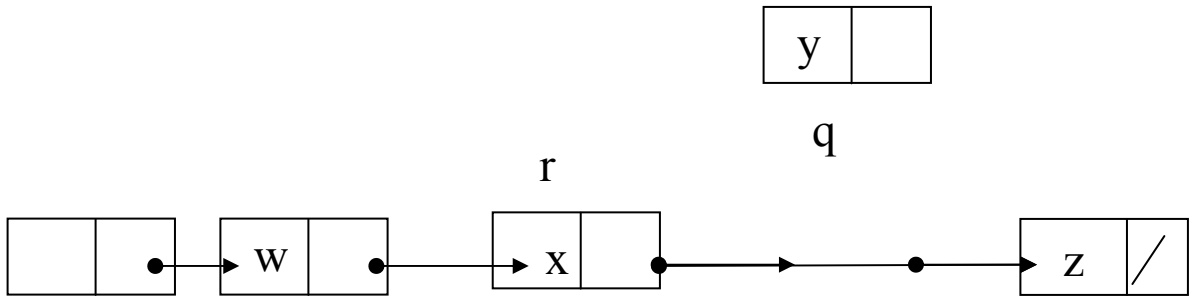
- Με παρόμοιο τρόπο μπορούμε να ορίσουμε διαδικασία `findpointer(LIST *L, key val)` που επιστρέφει δείκτη προς κόμβο της λίστας που περιέχει το στοιχείο `val`, αν υπάρχει.

- *Εισαγωγή Κόμβου μετά από συγκεκριμένο κόμβο*
  `insert(LIST *L, key x,y)`



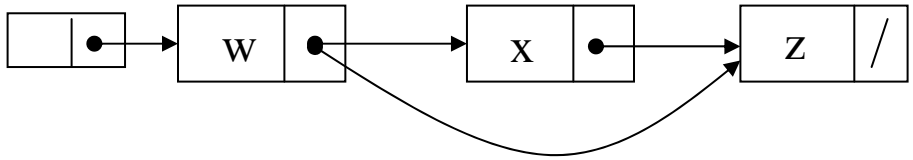
Ευθύγραμμες Απλά Συνδεδεμένες Λίστες

```
insert(LIST *L, key x,y) {
        r = findpointer(L,x);
        if r == NULL error
        else q = (NODE *)malloc(sizeof(NODE));
            (*q).data = y;
            (*q).next = (*r).next;
            (*r).next = q;
  }
```



Ευθύγραμμες Απλά Συνδεδεμένες Λίστες

- Εξαγωγή Κόμβου με συγκεκριμένη πληροφορία



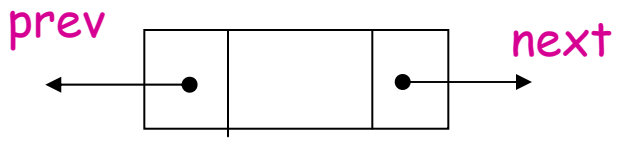

```
delete(LIST *L, key x){
  if (*L).top == NULL) return error

  Q = (*L).top;
  if (*Q).data == x
    (*L).top = (*Q).next; free(Q);
  else
    R = Q; Q = (*Q).next;
    while(Q != NULL){
       if ((*Q).data == x) break;
       else R = Q; Q = (*Q).next;
    }
  if Q == null  error
  else (*R).next = (*Q).next; free(Q);
}
```

Πίνακες / Συνδεδεμένες Λίστες

- Όταν υλοποιούμε λίστες με πίνακες χρειάζεται να γνωρίζουμε το μέγιστο μέγεθος της λίστας εκ των προτέρων.
- Χρήση Χώρου
  - Πίνακας: καταλαμβάνεται ο ίδιος χώρος άσχετα με τον αριθμό των στοιχείων που είναι αποθηκευμένα.
  - Συνδεδεμένη Λίστα: μέγεθος της λίστας × (χώρος ενός στοιχείου + χώρος ενός δείκτη)
- Χρόνος εισαγωγής/εξαγωγής στην αρχή
  - πίνακας: $\Theta(n)$
  - συνδεδεμένη λίστα: $\Theta(1)$
- Χρόνος εύρεσης k-οστού στοιχείου
  - πίνακας: $\Theta(1)$
  - συνδεδεμένη λίστα: $\Theta(k)$

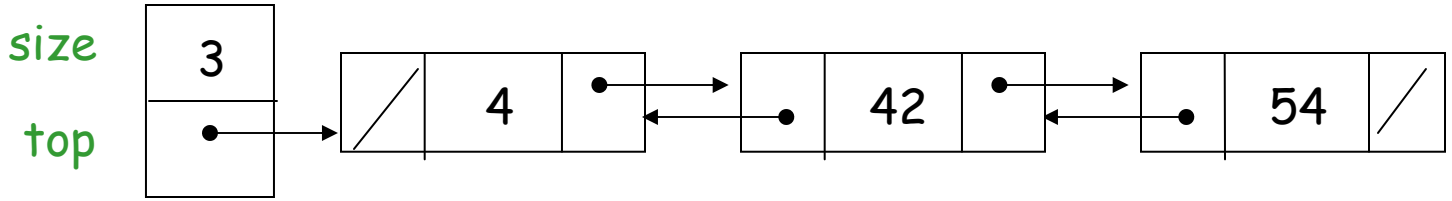
Ευθύγραμμες Διπλά Συνδεδεμένες Λίστες

- Διπλά συνδεδεμένη λίστα (doubly-linked list) ονομάζεται μια λίστα κάθε κόμβος της οποίας κρατά πληροφορίες και για τον επόμενο και για τον προηγούμενο κόμβο:



- Με αυτό τον τρόπο δίνεται η ευχέρεια μετακίνησης μέσα στη λίστα και προς τις δύο κατευθύνσεις.

- Παράδειγμα Λίστας:



Διπλά Συνδεδεμένες Λίστες

- Ποιες δομές χρειάζονται για υλοποίηση μιας διπλά συνδεδεμένης λίστας;
- Ένας κόμβος ορίζεται από το πιο κάτω structure:

```
typedef struct dlnode {
        int             data;
        struct dlnode *prev;
        struct dlnode *next;
} DLNODE;
```

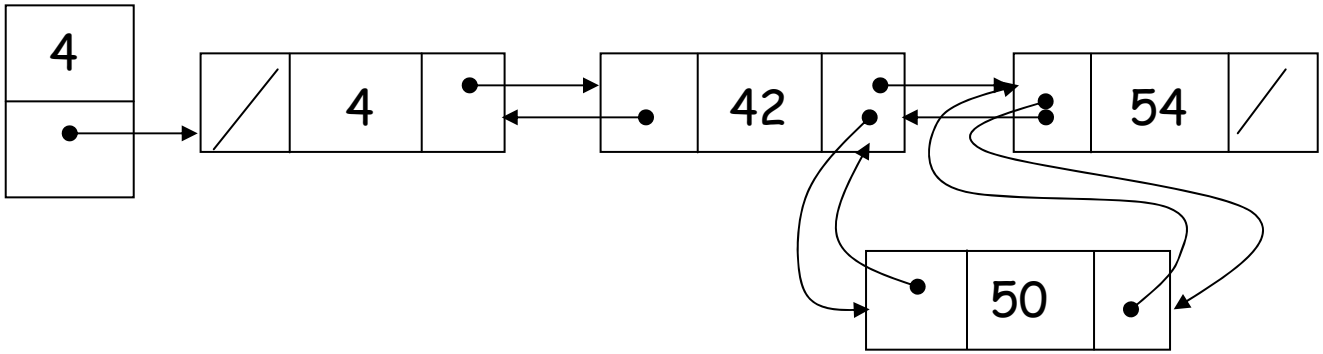
- Ο κόμβος που ορίζει τη διπλά συνδεδεμένη λίστα είναι ο ίδιος με αυτό που ορίζει μια στοίβα:

```
typedef struct dllist {
        DLNODE  *top;
        ...
} DLLIST;
```

Διπλά Συνδεδεμένες Λίστες

- Προφανώς η εισαγωγή στοιχείου σε κάποιο σημείο μιας διπλά συνδεδεμένης λίστας περιέχει κάποια επιπλέον πολυπλοκότητα από την εισαγωγή σε μια απλά συνδεδεμένη λίστα. Αυτό γιατί κάθε νεοδημιούργητος κόμβος πρέπει να συνδεθεί και με τον επόμενο και με τον προηγούμενο κόμβο στη λίστα. Παρόμοια, κατά τις εξαγωγές στοιχείων.

- Παράδειγμα εισαγωγής του στοιχείου 50 μετά το 42 στη λίστα της διαφάνειας 10:



Η συνάρτηση put

- Να ορίσετε συνάρτηση put(l, x, y) η οποία τοποθετεί το στοιχείο x μετά από το στοιχείο y μέσα στη λίστα l.

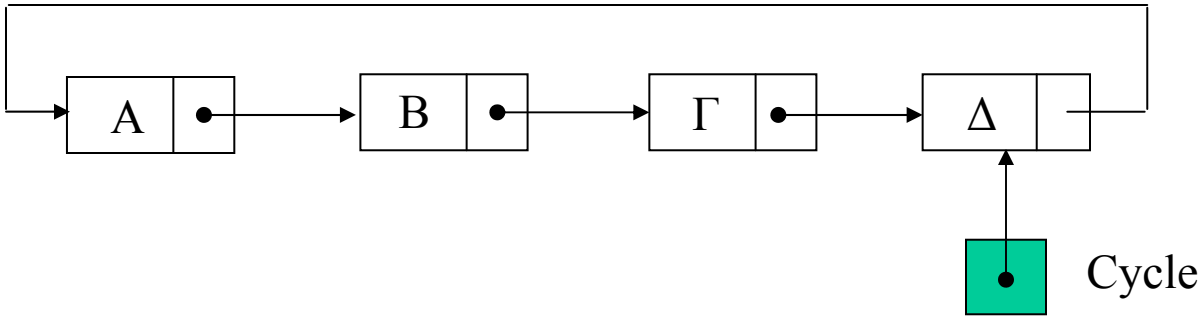
```
put(DLLIST *l, int x, int y){
  if (l->top == NULL)
    printf("The list is empty, no insertion was made\n");
  else
    putnode(l->top, x, y);
}
```

Η συνάρτηση puttnode

```
void  putnode(DLNODE *p, int x, int y){
  DLNODE  *q;
  if ( p->data == y ) {
       q = (DLNODE *)malloc(sizeof(DLNODE));
       q->data = x;
       q->next = p->next;
       q->prev = p;
       if ((p->next) != NULL) (p->next)->prev = q;
       p->next= q;
  }
  else if (p == NULL)
       printf("Element %d does not exist,
                no insertion was made\n", y);
  else
       putnode(p->next, x, y);
}
```

Κυκλικές Απλά Συνδεδεμένες Λίστες

- Μια κυκλική λίστα μπορεί να παίξει το ρόλο της ουράς, της οποίας και τα δύο άκρα είναι προσιτά με τη βοήθεια ενός μόνο δείκτη.



- Ο κόμβος που ορίζει την κυκλική απλά συνδεδεμένη λίστα είναι:

```
typedef struct clist {
        DLNODE  *back;
        …
} CLIST;
```

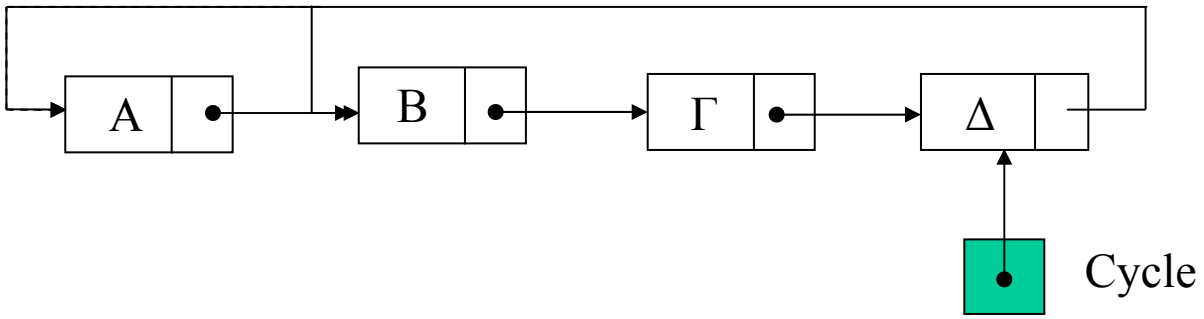
- Το πίσω άκρο, δηλαδή το άκρο όπου γίνονται οι εισαγωγές, είναι αυτό που δείχνεται από τον δείκτη στη λίστα (Cycle). Άρα οι εξαγωγές γίνονται ακριβώς μετά από τον κόμβο που δείχνεται από τον δείκτη Cycle.

Κυκλικές Απλά Συνδεδεμένες Λίστες

- Ο κόμβος εξόδου μπορεί να διαγραφεί με τη διαδικασία

```
delete(CLIST *Cycle) {
        Q = Cycle->back;
        if (Q == NULL) return;
        if (Q->next = Q) Cycle->back = NULL
        print (Q->next)->data;
        P = (Q->next)->next;
        free(Q->next); Q->next=P;
}
```

- Εφαρμογή της διαδικασίας στη λίστα της Διαφ. 5-14 έχει σαν αποτέλεσμα:



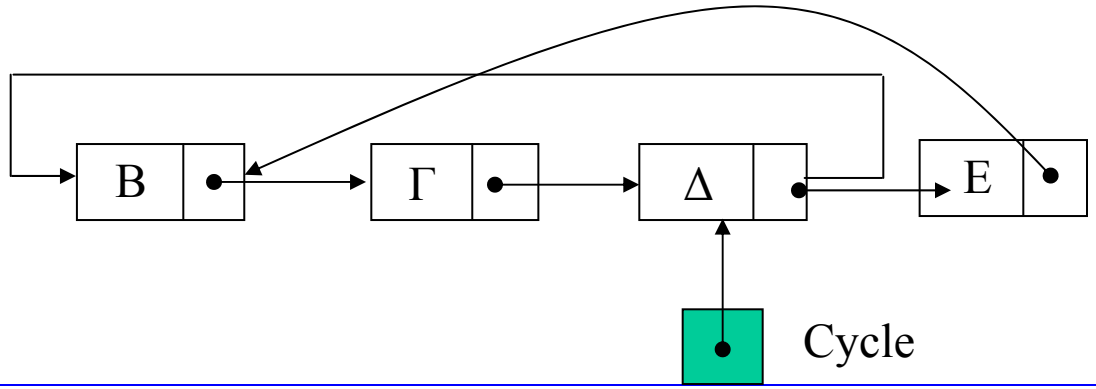
Κυκλικές Απλά Συνδεδεμένες Λίστες

- Εισαγωγές μπορούν να γίνουν και στα δύο άκρα σε χρόνο O(1):

```
insertfront (key x, CLIST *Cycle){
    Q = (NODE *)malloc(sizeof(NODE));
    Q->data = x;
    Q->next = (Cycle->back)->next;
    (Cycle->back)->next = Q;
}
```

Υπόθεση: Η λίστα έχει τουλάχιστον 1 στοιχείο.

- Μετά από την διαδικασία insertfront(E, Cycle) η λίστα της διαφάνειας 15 έχει ως εξής:

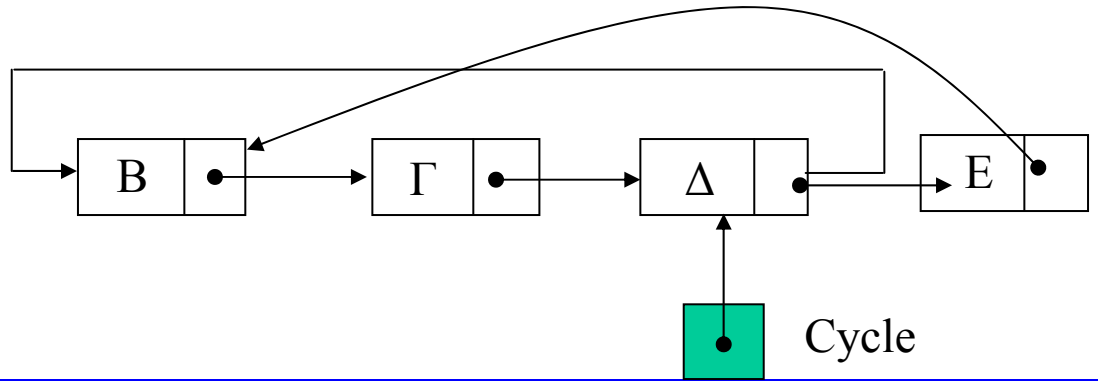


Κυκλικές Απλά Συνδεδεμένες Λίστες

```
insertback ( key x, CLIST *Cycle){
   Q = (NODE *)malloc(sizeof(NODE));
   Q->data = x;
   Q->next = (Cycle->back)->next;
   (Cycle->back)->next = Q;
   Cycle->back = Q;
}
```

Υπόθεση: Η λίστα έχει τουλάχιστον 1 στοιχείο.

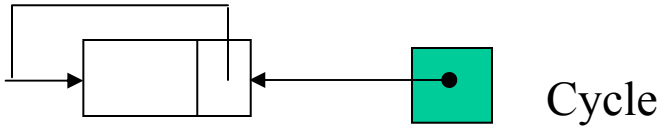
- Μετά από την διαδικασία insertback(E, Cycle) η λίστα της διαφάνειας 15 έχει ως εξής:



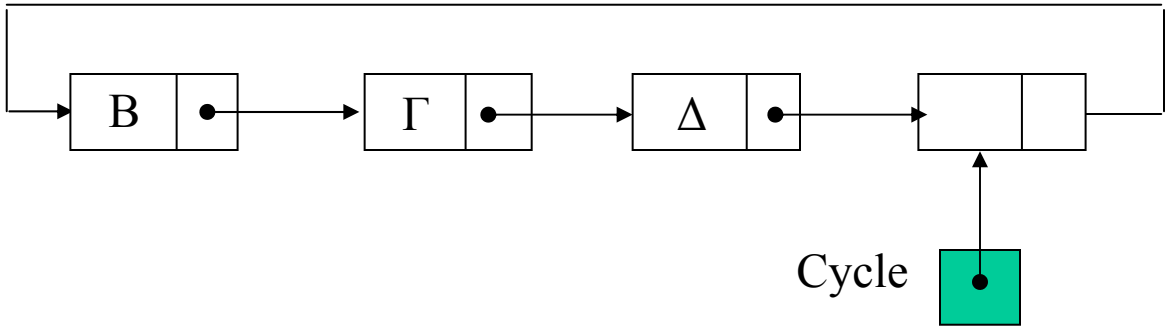
Κυκλικές Απλά Συνδεδεμένες Λίστες

- Υπάρχει η εξής ασυνέχεια σε κυκλικά συνδεδεμένες λίστες: ενώ η μη-κενή λίστα δεν έχει καμιά μηδενική τιμή συνδέσμων, η κενή λίστα παριστάνεται από ένα μηδενικό σύνδεσμο. Αυτή η ασυνέχεια προκαλεί πρόσθετους ελέγχους στις διάφορες χρήσιμες πράξεις.

- Για να επιτύχουμε ομοιομορφία μεταξύ των δυο ειδών λίστας μπορούμε να εισαγάγουμε κόμβους κεφαλής, δηλαδή κόμβους που δεν περιέχουν πληροφορίες. Για παράδειγμα η άδεια ουρά δίνεται ως



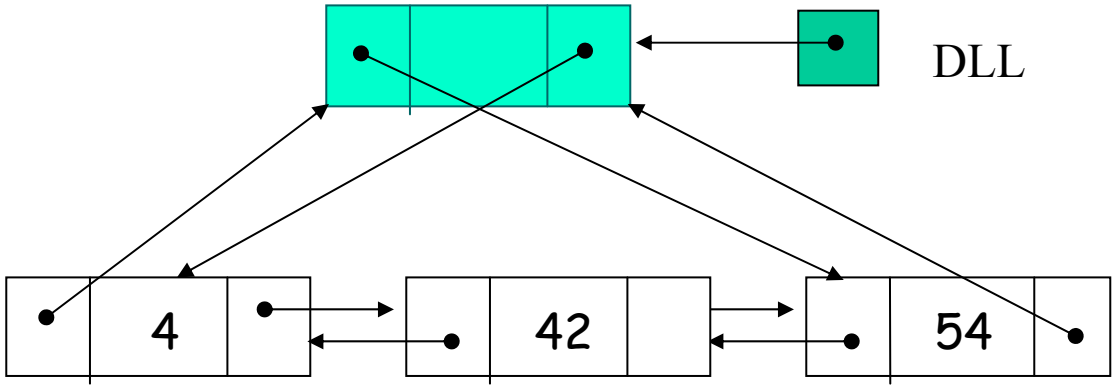
ενώ η ουρά στην αρχή της διαφάνειας 14 υλοποιείται ως



Κυκλικές Διπλά Συνδεδεμένες Λίστες

- Κυκλική λίστα της οποίας ο κάθε κόμβος δείχνει και στον επόμενο και στον προηγούμενο.

- Με την εισαγωγή κεφαλής σε τέτοιες λίστες παίρνουμε τις κυκλικές διπλά συνδεδεμένες λίστες με κεφαλή.



Τεχνικές Μείωσης Χώρου

- Οι διπλά συνδεδεμένες λίστες έχουν το μειονέκτημα πως απαιτούν την ύπαρξη δύο πεδίων δεικτών σε κάθε κόμβο.
- Πολλές από τις αποθηκευμένες πληροφορίες επαναλαμβάνονται: κάθε δείκτης αποθηκεύεται σε δυο κόμβους, τον επόμενο και τον προηγούμενο με αποτέλεσμα τη σπατάλη χώρου.
- Μπορούμε να συμπτύξουμε σε ένα πεδίο τύπου δείκτη τις διευθύνσεις και του επόμενου και του προηγουμένου κόμβου;

Τεχνικές Μείωσης Χώρου

- *Ορισμός* (Exclusive or): $a \oplus b = 0 \Leftrightarrow a = b$, δηλ.

a	b	a ⊕ b
1	1	0
1	0	1
0	1	1
0	0	0

- Αν $a_1a_2\ldots a_n$, $b_1b_2\ldots b_n$ είναι συμβολοσειρές και $a_i \oplus b_i = c_i$ τότε ορίζουμε $a_1a_2\ldots a_n \oplus b_1b_2\ldots b_n = c_1c_2\ldots c_n$

- *Ιδιότητες του Exclusive-or*

  $a \oplus b = b \oplus a$ $\qquad$ $a \oplus (b \oplus c) = (a \oplus b) \oplus c$

  $a \oplus a = 0$ $\qquad$ $a \oplus 0 = a$

Τεχνικές Μείωσης Χώρου

Μέθοδος μείωσης χώρου

Θεωρούμε ότι κάθε κόμβος της λίστας αποτελείται από δύο πεδία: το πεδίο `data` για αποθήκευση δεδομένων και το πεδίο `Link` που περιέχει *ένα* pointer. Ας υποθέσουμε πως η λίστα χρησιμοποιείται για την αποθήκευση n δεδομένων. Έστω ότι η διεύθυνση του κόμβου που περιέχει το i-οστό δεδομένο είναι η M_i. Θεωρούμε επίσης δύο επιπλέον κόμβους με διευθύνσεις M_{n+1} και M_{n+2}.

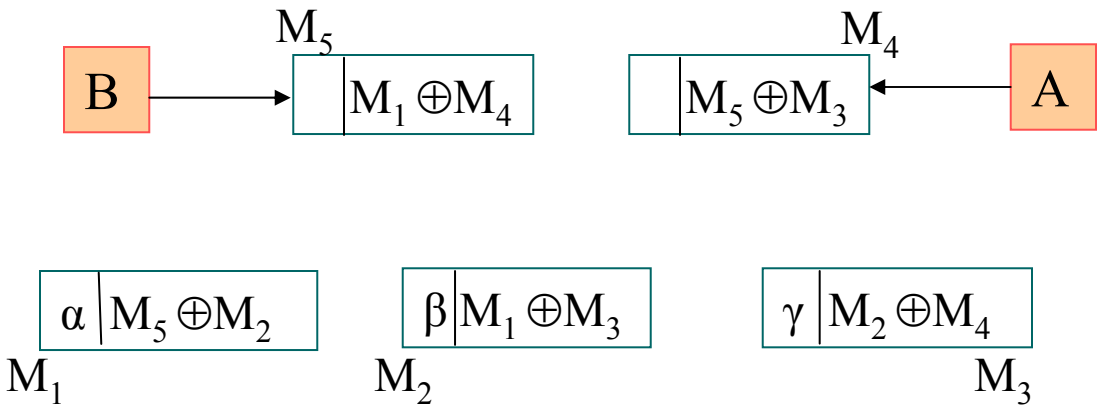
- Για κάθε κόμβο, το πεδίο `Link` περιέχει το exclusive-or δύο διευθύνσεων: της διεύθυνσης του κόμβου που προηγείται και της διεύθυνσης αυτού που ακολουθεί. Δηλαδή

$$(*M_i).Link = M_{(i-1) mod (n+2)} \oplus M_{(i+1) mod (n+2)}.$$

- Για τη διέλευση της λίστας απαιτούνται δύο δείκτες A, B, οι οποίοι να δείχνουν πάντοτε σε δυο διαδοχικούς κόμβους. Αρχικά οι A,B, δείχνουν στους κόμβους M_{n+1} και M_{n+2}.

Τεχνικές Μείωσης Χώρου

- Ο επόμενος κόμβος δίνεται από το A⊕Link(B), και ο προηγούμενος κόμβος από το Link(A) ⊕ B.



π.χ. $A \oplus Link(B) = M_4 \oplus M_1 \oplus M_4 = M_1$

$Link(A) \oplus B = M_5 \oplus M_5 \oplus M_3 = M_3$