ΕΝΟΤΗΤΑ 3
ΓΡΑΜΜΙΚΕΣ ΛΙΣΤΕΣ

Γραμμικές Λίστες

Ορισμός

Γραμμική λίστα (linear list) είναι ένα σύνολο από n >= 0 κόμβους L[0], L[1], ..., L[n-1], όπου το στιχείο L[0] είναι το πρώτο στοιχείο (ή ο πρώτος κόμβος), ενώ το στοιχείο L[k] προηγείται του στοιχείου L[k+1] και έπεται του στοιχείου L[k-1], 0 < k < n-1.

- L[0]: κεφαλή (head)
- L[n-1]: ουρά (tail)
- |L|: μήκος λίστας (|L| = n)
- < >: κενή λίστα

Λειτουργίες που συνήθως υποστηρίζονται από λίστες:

- *Access(L,j):* Επιστρέφει L[j] ή ένα μήνυμα λάθους αν j είναι < 0 ή > |L|-1.
- *Length(L):* Επιστρέφει |L|.
- *Concat(L_1, L_2):* επιστρέφει μια λίστα: το αποτέλεσμα της συνένωσης των 2 λιστών σε 1.
- *MakeEmptyList():* επιστρέφει < >.
- *IsEmptyList(L):* επιστρέφει true αν L = < >, false διαφορετικά.

Είδη Γραμμικών Λιστών

Σειριακή Λίστα**:** καταλαμβάνει συνεχόμενες θέσεις κύριας μνήμης

Συνδεδεμένη Λίστα: οι κόμβοι βρίσκονται σε απομακρυσμένες θέσεις συνδεδεμένες όμως μεταξύ τους με δείκτες.

Στατικές Λίστες: μέγιστος αριθμός στοιχείων είναι εξ αρχής γνωστός (υλοποίηση με σειριακές λίστες).

Δυναμικές Λίστες: ο μέγιστος αριθμός στοιχείων δεν είναι γνωστός. Επιτρέπεται η επέκταση ή συρρίκνωση της λίστας κατά την εκτέλεση του προγράμματος (υλοποίηση με συνδεδεμένες λίστες).

- Στοίβα
- Ουρά

Αφηρημένος τύπος δεδομένων Στοίβα (Stack)

Λίστα που μπορεί να τροποποιείται μόνο με την εισαγωγή και διαγραφή στοιχείων στο ένα της άκρο.

Λειτουργίες

Top(S): επιστρέφει το κορυφαίο στοιχείο της S (αντίστοιχο Access(S, |S|-1)).

Pop(S): (λειτουργία ώθησης) διαγραφή και επιστροφή του κορυφαίου στοιχείου της S

Push(x,S): (λειτουργία απώθησης) εισαγωγή του στοιχείου x στην κορυφή της στοίβας

MakeEmptyStack(): επιστρέφει την < >.

IsEmptyStack(S): επιστρέφει true αν η |S| = 0, διαφορετικά false.

Η μέθοδος επεξεργασίας των δεδομένων στοίβας λέγεται **«Τελευταίο Μέσα – Πρώτο Έξω» (Last In – First Out, LIFO).**

Αφηρημένος τύπος δεδομένων Ουρά (Queue)

Λίστα που μπορεί να τροποποιείται μόνο με την εισαγωγή στοιχείων στο ένα άκρο της και τη διαγραφή στοιχείων στο άλλο.

Λειτουργίες

Enqueue(x,Q): Εισαγωγή x στο τέλος της Q (αντίστοιχο Concat(Q, <x>)).

Dequeue(Q): Διαγραφή & επιστροφή του πρώτου στοιχείου της Q (Q = <Q[1], …, Q[|Q|-1]> και το Q[0] επιστρέφεται).

Front(Q): επιστρέφει Q[0].

MakeEmptyQueue(): επιστρέφει < >.

IsEmptyQueue(Q): επιστρέφει true αν |Q| = 0, false διαφορετικά.

Η μέθοδος επεξεργασίας των δεδομένων ουράς λέγεται **«Πρώτο Μέσα – Πρώτο Έξω» (First In – First Out, FIFO).**

Σειριακές Γραμμικές Λίστες

Στατικές Στοίβες

Μια στατική στοίβα υλοποιείται με τη χρήση ενός μονοδιάστατου πίνακα.

- $S = \langle x_0, \ldots, x_{n-1} \rangle$ η στοίβα & A[0 … N-1] ο πίνακας.
- $A[j] = x_j$
- Η στοίβα καταλαμβάνει το κομμάτι A[0 … n-1].
- A[n-1]: κορυφαίο στοιχείο της στοίβας
- A[0]: βαθύτερο στοιχείο της στοίβας

Η στοίβα υλοποιείται ως μια δομή (struct στη C) με πεδία τον πίνακα Infos και τον ακέραιο Length (μέγεθος στοίβας).

S: δείκτης σε στοίβα

info: τύπος στοιχείων του πίνακα (Infos(S)).

Length(S) = 0: άδεια στοίβα

Length(S) = N: γεμάτη στοίβα

Υλοποίηση Λειτουργιών Στοίβας

function *MakeEmptyStack():***pointer**

```
S = newcell(Stack)
Length(S) = 0
return S
```

function *IsEmptyStack(S):* **boolean**

```
return (Length(S) == 0)
```

function *Top(pointer S):* **info**

```
if (IsEmptyStack(S)) then error
else (return(Infos(S)[Length(S) – 1]))
```

Χρονική Πολυπλοκότητα

MakeEmptyStack(): Θ(1)

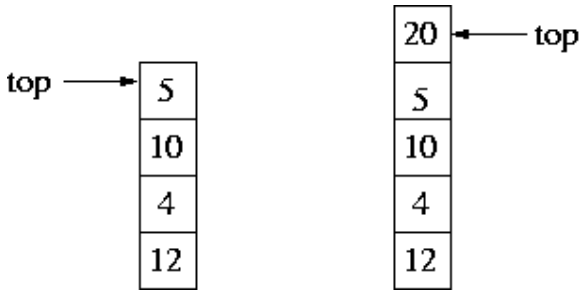
IsEmptyStack(): Θ(1)

Top(): Θ(1)

Συνολικός Απαιτούμενος Χώρος Μνήμης

Ανεξάρτητα από τον αριθμό στοιχείων: N

Υλοποίηση Λειτουργιών Στοίβας



function *Pop(pointer S):* **info**

```
if (Length(S) == 0) return error
else
    x = Top(S);
    Length(S) = Length(S) –1
    return x;
```

procedure *Push(info x, pointer S)*

```
if (Length(S) == N) then error
else
    Lenth(S) = Lenth(S) + 1;
    Infros(S)[Length(S)-1] = x;
```

Χρονική Πολυπλοκότητα

Pop(): Θ(1)

Push(): Θ(1)

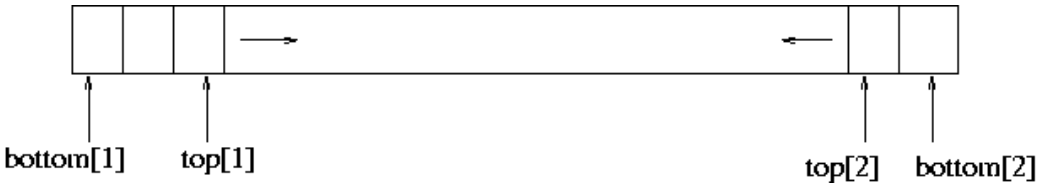
Πολλαπλή Στοίβα

Περισσότερες από μια στοίβες που υλοποιούνται σε συνεχομένες θέσεις μνήμης.

Παράδειγμα 1: Δύο Στοίβες

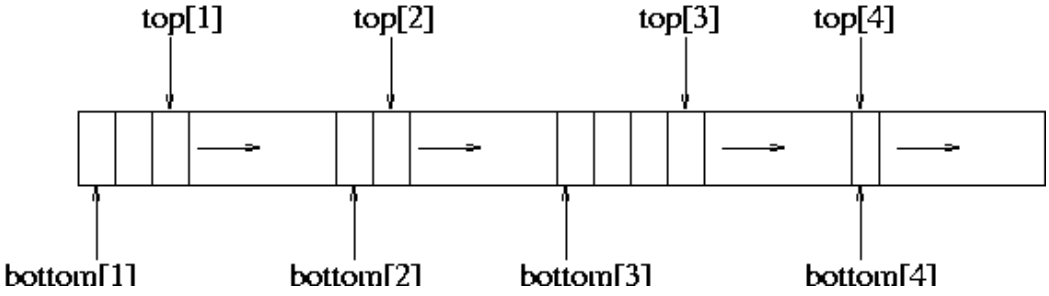
Έστω Stack[0…n-1] ο πίνακας που χρησιμοποιείται για την αποθήκευση των λιστών.

Η 1η στοίβα ξεκινάει από τη θέση stack[0] και αναπτύσσεται προς τα δεξιά, ενώ η 2η ξεκινάει από τη θέση Stack[n-1] και αναπτύσσεται προς τα αριστερά.



Παράδειγμα 2: n Στοίβες

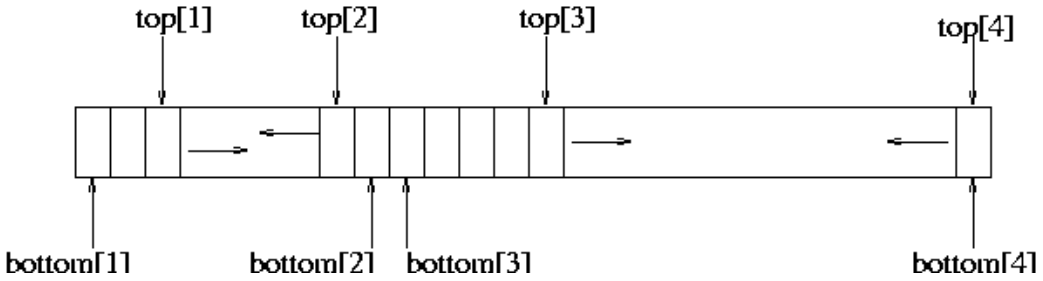
Η πολλαπλή στοίβα χωρίζεται σε n ίσα τμήματα.



Τι γίνεται σε περίπτωση υπερχείλισης?

Καλύτερες Υλοποιήσεις n-Πολλαπλής Στοίβας

Δικατευθυνόμενη Πολλαπλή Στοίβα



Εφαρμογές Σειριακών Γραμμικών Στοιβών

Αναδρομή

Υπολογισμός του n!

Αναδρομική Λύση

```
function factorial(integer n): integer
    if (n == 0) then return 1
    else return (n * factorial(n-1))
```

Μη Αναδρομική Λύση

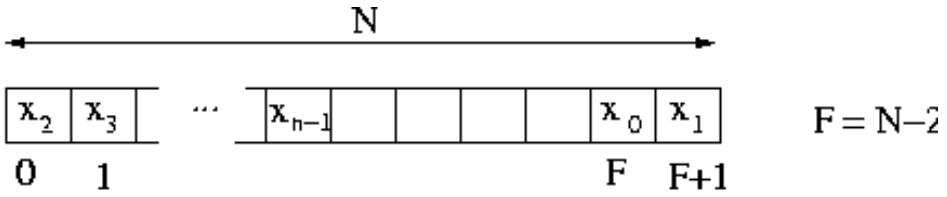
```
function factorial(integer n): integer
    integer j, product;

    j = n
    product = 1
    while (j > 0)
        product = j * product
        j = j-1
    return product
```

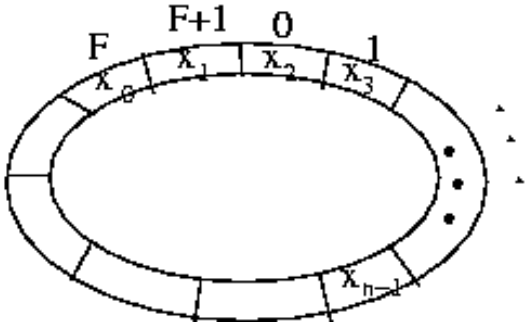
Στατική Ουρά

Q: ουρά, δομή με τρία πεδία:

- A=Infos(Q): πίνακας με στοιχεία
- F=Front(Q): θέση πρώτου στοιχείου
- n=Length(Q): συνολικός αριθμός στοιχείων.

Κυκλική Στατική Ουρά





- $x_0, \ldots, x_{n-1}$: στοιχεία ουράς
- A[F], A[(F+1) mod N], A[(F+2) mod N] , … , A[(F+n-1) mod N]: θέσεις στις οποίες είναι αποθηκευμένα τα $x_0, \ldots, x_{n-1}$.

Υλοποίηση Λειτουργιών Κυκλικής Ουράς

function *MakeEmptyQueue():***pointer**

```
Q = NewCell(Queue)
Front(Q) = 0
Legth(Q) = 0
Return Q
```

function *IsEmptyQueue(pointer Q):***boolean**

```
return (Length(Q) == 0)
```

procedure *Front(pointer Q)*

```
If IsEmptyQueue(Q) then error
else return Infos(Q)[Front(Q)]
```

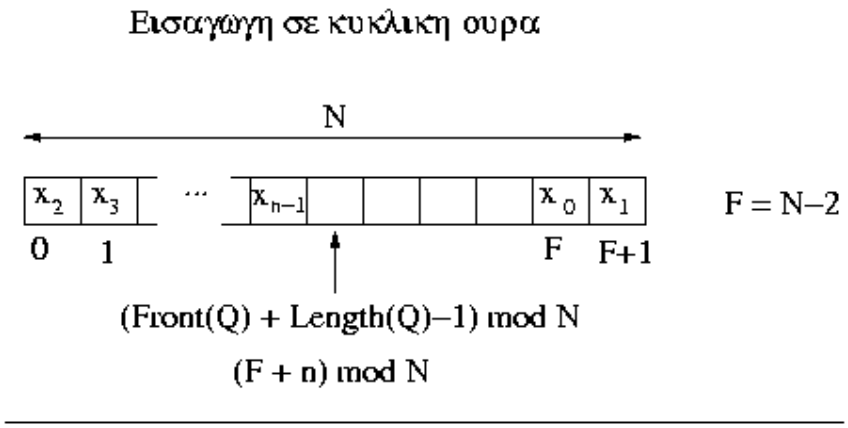
Πολυπλοκότητα

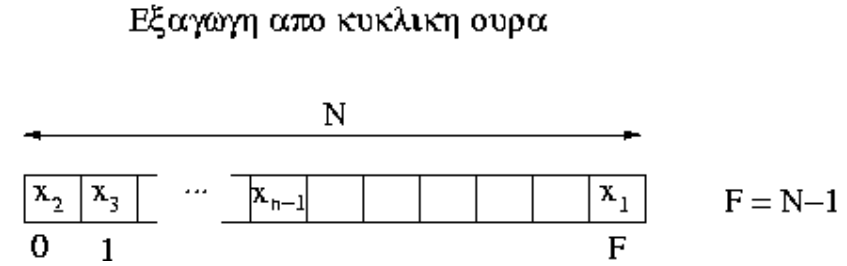
Ίδια με την υλοποίηση στοίβας

Χρόνος εκτέλεσης κάθε λειτουργίας: Θ(1)

Χρησιμοποιούμενος χώρος μνήμης: N

Υλοποίηση Λειτουργιών Κυκλικής Ουράς





function *Dequeue(pointer L):***info**

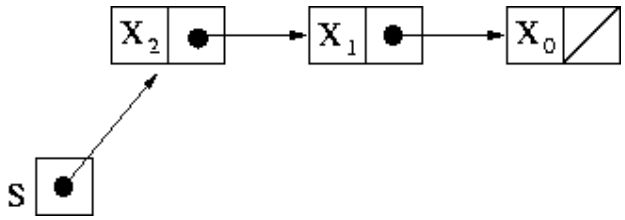
```
if IsEmptyQueue(Q) then error
else
    x = Infos(Q)[Front(Q)]
    Front(Q) = (Front(Q)+1) mod N
    Length(Q) = Length(Q) –1
    return x
```

procedure *Enqueue(info x, pointer Q)*

```
if (Length(Q) == N) then error
else
   Length(Q) = Length(Q)+1
   Infos(Q)[(Front(Q) + Length(Q) –1) mod N] = x
```

Συνδεδεμένες Γραμμικές Λίστες

Στοίβα ως Συνδεδεμένη Λίστα



S: δείκτης σε δομή (που ονομάζεται Node) με πεδία :

- Next: δείκτης στο επομένο στοιχείο της στοίβας
- Info: πληροφορία αποθηκευμένη στο στοιχείο

Υλοποίηση Λειτουργιών

function *MakeEmptyStack():* **pointer**

return NULL

function *IsEmptyStack(pointer S):* **boolean**

return (S == NULL);

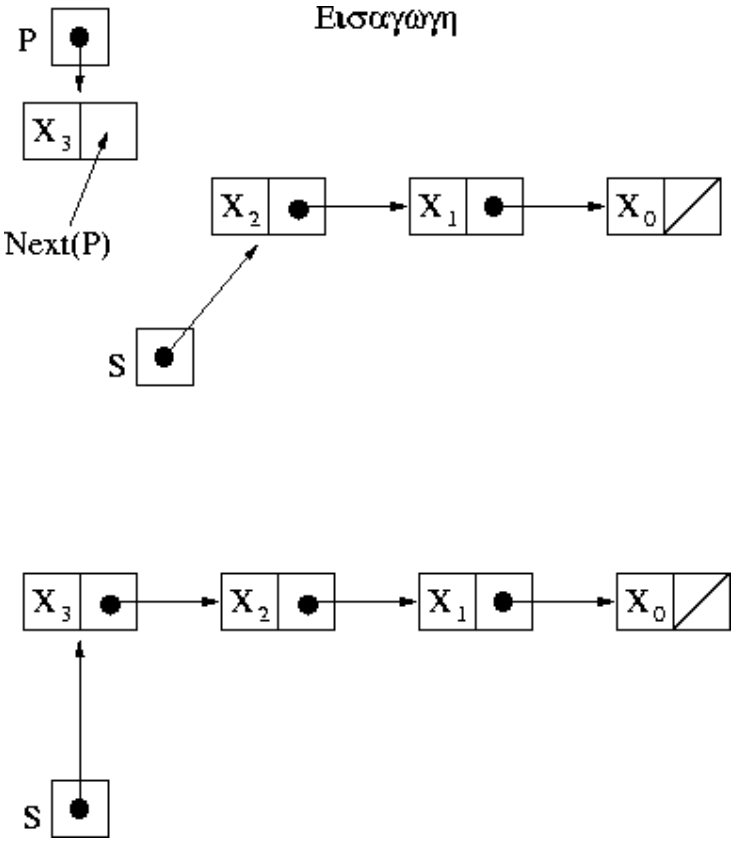
function *Top(pointer S):* **info**

if IsEmptyStack(S) then error

else return Info(S)

Χρόνος εκτέλεσης κάθε λειτουργίας: Θ(1)

Υλοποίηση Λειτουργιών Συνδεδεμένης Στοίβας



procedure *Push(info x, pointer S)*

P = NewCell(Node)

Info(P) = x

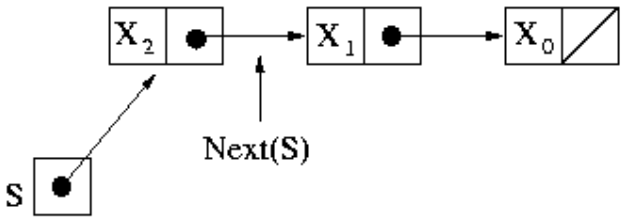
Next(P) = S

S = P

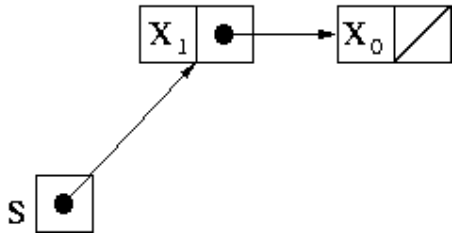
Απαιτούμενος χρόνος: Θ(1)

Υλοποίηση Λειτουργιών Συνδεδεμένης Στοίβας

Διαγραφη



S = Next(S)



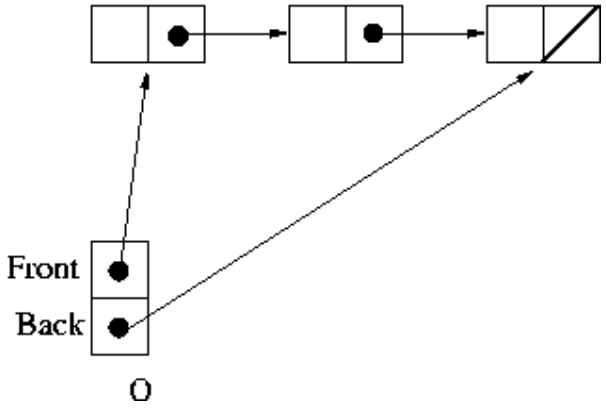
function *Pop(pointer S):* **info**

```
if IsEmptyStack(S) then error
else
    x = Top(S)
    S = Next(S)
    return x
```

Απαιτούμενος Χρόνος: Θ(1)

Extra μνήμη (για δείκτες): n (όπου n: # στοιχείων)

Ουρά ως Συνδεδεμένη Λίστα



Ουρά Q: δομή (που ονομάζεται Queue) 2 δεικτών (Front(Q) και Back(Q)) σε δομή (που ονομάζεται Node) 2 πεδίων:

- Next: δείκτης στο επομένο στοιχείο της στοίβας
- Info: πληροφορία αποθηκευμένη στο στοιχείο

Υλοποίηση Λειτουργιών Ουράς

function *MakeEmptyQueue():* **pointer**

```
Q = NewCell(Queue)
Front(Q) = Back(Q) = NULL
return Q
```

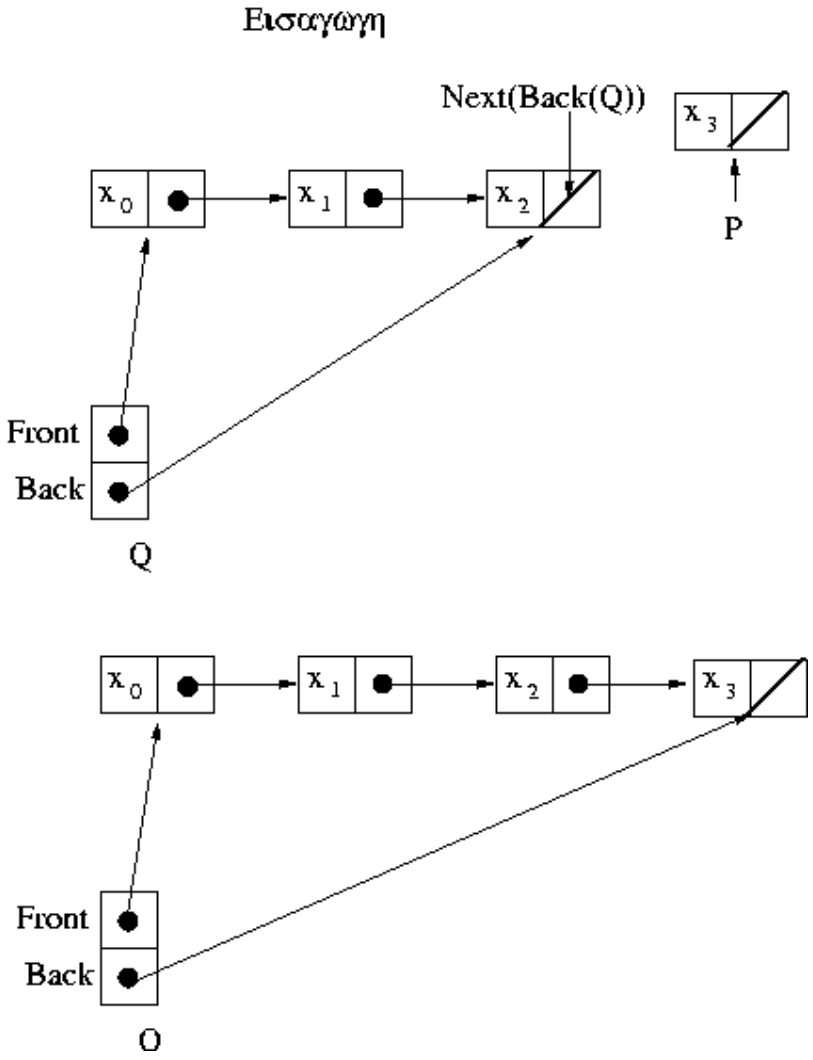
function *IsEmptyQueue(pointer Q):* **boolean**

```
return (Front(Q) == NULL)
```

function *Front(pointer Q):* **info**

```
if IsEmptyQueue(Q) then error
else return Info(Front(Q))
```

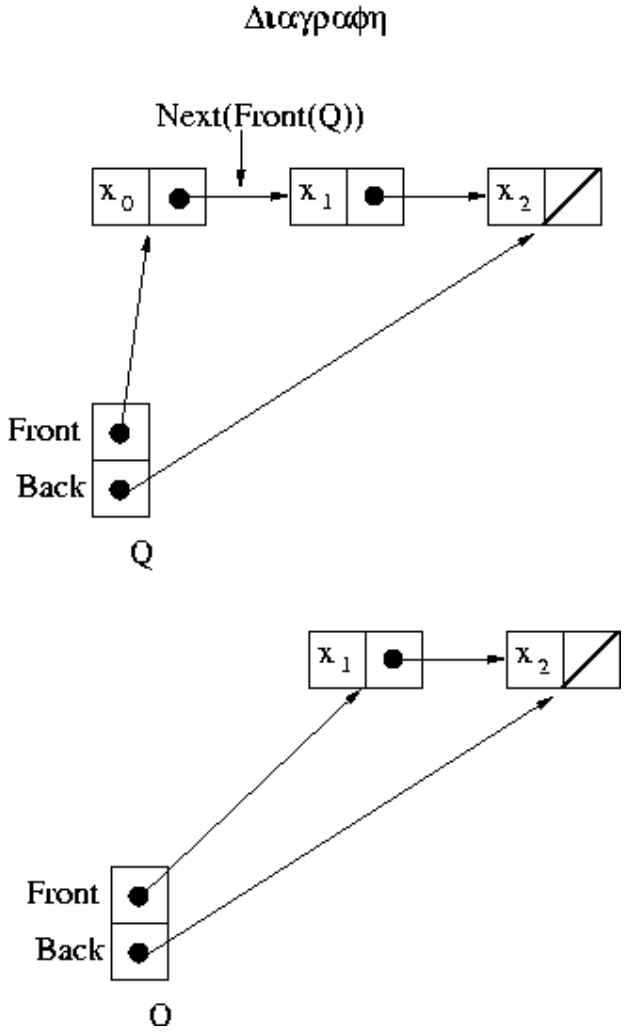
Υλοποίηση Λειτουργιών Ουράς



procedure *Enqueue(info x, pointer Q)*

P = NewCell(Node)
Info(P) = x
Next(P) = NULL
if IsEmptyQueue(Q) then Front(Q) = P
else Next(Back(Q)) = P
Back(Q) = P

Υλοποίηση Λειτουργιών Ουράς



function *Dequeue(pointer Q):* **info**

```
if IsEmptyQueue(Q) then error
else
    x = Info(Front(Q))
    Front(Q) = Next(Front(Q))
    if  (Front(Q) == NULL) then
        Back(Q) = NULL
    return x
```

Πολυπλοκότητα

Ίδια με εκείνη για στοίβες

Παράδειγμα Εφαρμογής Συνδεδεμένων Λιστών

Παράσταση πολυωνύμων

$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

Χρήση λίστας για την αποθήκευση των συντελεστών.

Κόμβος Φρουρός

Αναζήτηση Κόμβου με συγκεκριμμένη τιμή. Αν ο κόμβος δεν υπάρχει στη λίστα εισάγεται.

Εισαγάγεται ένας κόμβος (τελευταίος πάντα στη λίστα) που λέγεται κόμβος φρουρός.
Ένας δείκτης δείχνει μόνιμα σε αυτόν τον κόμβο.
Η προς αναζήτηση τιμή αρχικά αποθηκεύεται στον κόμβο αυτό.
Εκτελείται διάσχιση της λίστας και αναζήτηση της τιμής.
Αν βρεθεί στο κόμβο φρουρό γίνεται η εισαγωγή. Διαφορετικά όχι.

Σε τι μας βοηθάει ο κόμβος φρουρός?

2ο Παράδειγμα Εφαρμογής Συνδεδεμένων Λιστών

Αραιοί Πίνακες

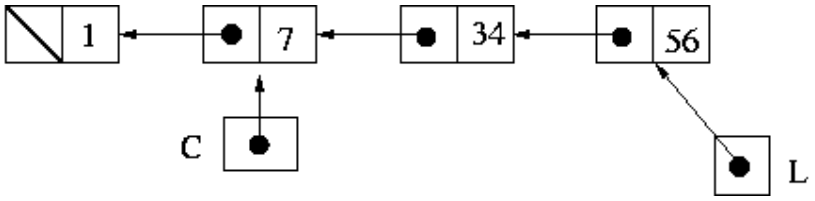
- Αποθήκευση των μη-μηδενικών στοιχείων σε μια λίστα.

- Κάθε στοιχείο της λίστας αντιστοιχεί σε ένα μη-μηδενικό στοιχείο του πίνακα.

- Προσπέλαση ενός στοιχείο απαιτεί χρόνο ανάλογο του αριθμού των μη-μηδενικών στοιχείων στον πίνακα.

- Εισαγωγές στοιχείων είναι ωστόσο δυνατές (δηλαδή ενός μηδενικού στοιχείου του πίνακα σε μη-μηδενικό είναι δυνατές).

Εισαγωγή Στοιχείου σε Ταξινομημένη Λίστα



Ο κάθε κόμβος της λίστας περιέχει π.χ. έναν ακέραιο και έναν δείκτη στον επόμενο κόμβο:
Num: αποθηκευμένος ακέραιος στον κόμβο
Next: δείκτης στον επόμενο κόμβο της λίστας

L: δείκτης στο πρώτο στοιχείο της λίστας

Πρόβλημα προς επίλυση
Εισαγωγή νέου στοιχείου στη λίστα, έτσι ώστε η λίστα να εξακολουθήσει να είναι ταξινομημένη.

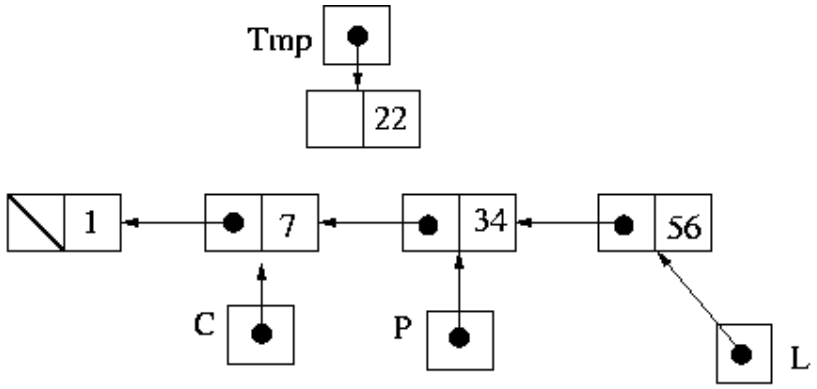
K: προς εισαγωγή ακέραιος

Πρόβλημα με την εισαγωγή στοιχείου σε ταξινομημένη λίστα:

Είναι δυνατή η εισαγωγή ενός στοιχείου μόνο ως επόμενου κόμβου κάποιου δεδομένου κόμβου και όχι ως προηγούμενου.

Λύση

Χρήση ενός βοηθητικού δείκτη P (που δείχνει πάντα στο προηγούμενο από το τρέχον στοιχείο).



Εισαγωγη σε ταξινομημενη λιστα

```
procedure LLInsert(integer K)
        C = L
        P = NULL
        while (C != NULL) and (Num(C) > K) do
                P = C
                C = Next(C)
        if (C != NULL) and (Num(C) == K) then
                return
        Tmp = NewCell(Node)
        Num(Tmp) = K
        if (P == NULL) then
                Next(Tmp) = NULL
                L = Tmp
        else
                Next(Tmp) = S
                Next(P) = Tmp
```

Διάσχιση Λίστας

Εκτέλεση επίσκεψης σε ένα ή σε κάποια προκαθορισμένα στοιχεία μιας λίστας με κάποια προκαθορισμένη σειρά.

Θεωρούμε λίστα που περιέχει strings (λέξεις) & είναι λεξικογραφικά ταξινομημένη.

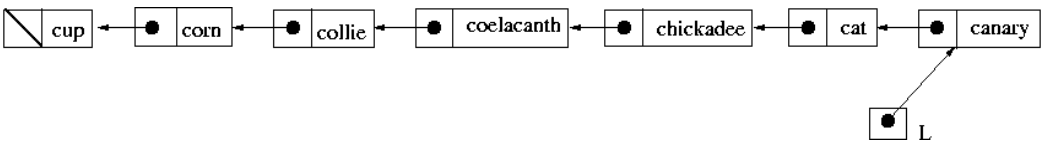
Πρόβλημα

Δεδομένης λέξης w, βρείτε την τελευταία λέξη στη λίστα που προηγείται αλφαβητικά της w και τελειώνει με το ίδιο γράμμα όπως η w.

Παράδειγμα

w = crabapple

L = <canary, cat, chickadee, coelacanth, collie, corn, cup>.

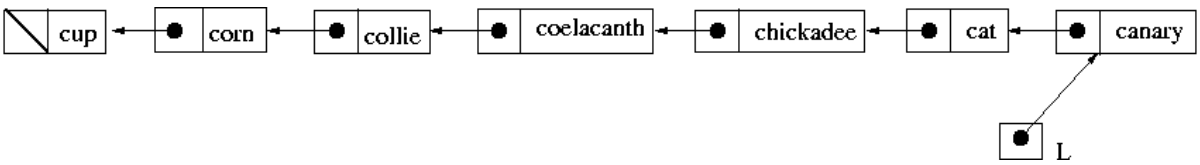


Η απάντηση θα πρέπει να είναι collie.

Πιθανοί Αλγόριθμοι

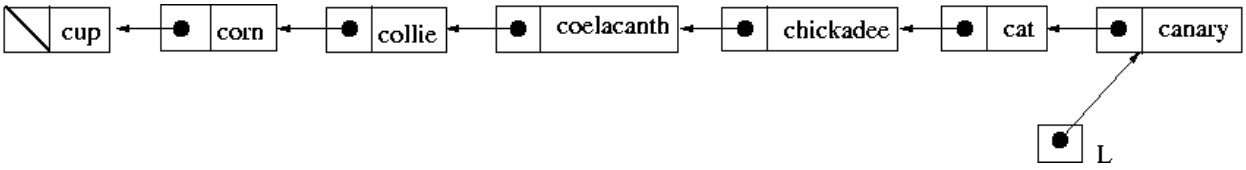
Αλγόριθμος 1

1) Διασχίζουμε τη λίστα προς τα εμπρός ως να βρούμε την πρώτη αλφαβητικά «μεγαλύτερη» λέξη από την crabapple, κρατώντας δείκτες προς τα πίσω (στο παράδειγμα την cup).
2) Ακολουθούμε τους δείκτες προς τα πίσω (που προσθέσαμε κατά την διάσχιση προς τα μπρος) μέχρι να βρούμε την πρώτη λέξη που τελειώνει σε e.



Αλγόριθμος 2

Διασχίζουμε τη λίστα προς τα εμπρός κρατώντας έναν δεύτερο (βοηθητικό) δείκτη στο τελευταίο στοιχείο που είδαμε να έχει την ζητούμενη ιδιότητα.



Ψευδοκώδικας

function FindLast(pointer L, string w): string
/*find the last word in L ending with the same letter as w*/
/* return NULL if there is no such word */

```
P = NULL
C = L
while (C != NULL) and (String(C) < w) do
    if (String(C) ends with the
                same letter as w) then P = C
    C = Next(C)
If (P == NULL) then return NULL
else return String(P)
```

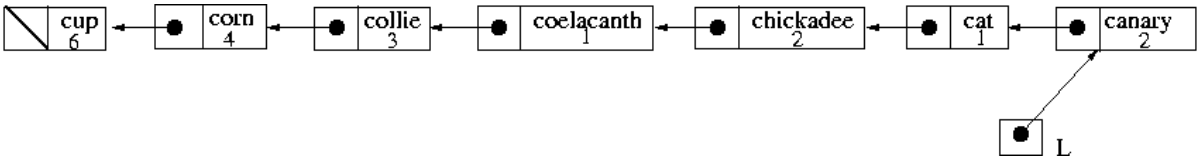
Πως θα συγκρίνατε την πολυπλοκότητα των δύο αλγορίθμων?

Διασχίσεις Zig-Zag

Παράδειγμα

Έστω ότι κάθε κόμβος έχει τα εξής πεδία:
- String: λέξη
- Num: ακέραιος
- Next: δείκτης στον επόμενο κόμβο

Δεδομένης λέξης w της λίστας η οποία σχετίζεται με τον αριθμό n, βρείτε την λέξη που προηγείται της w κατά n θέσεις στη λίστα.



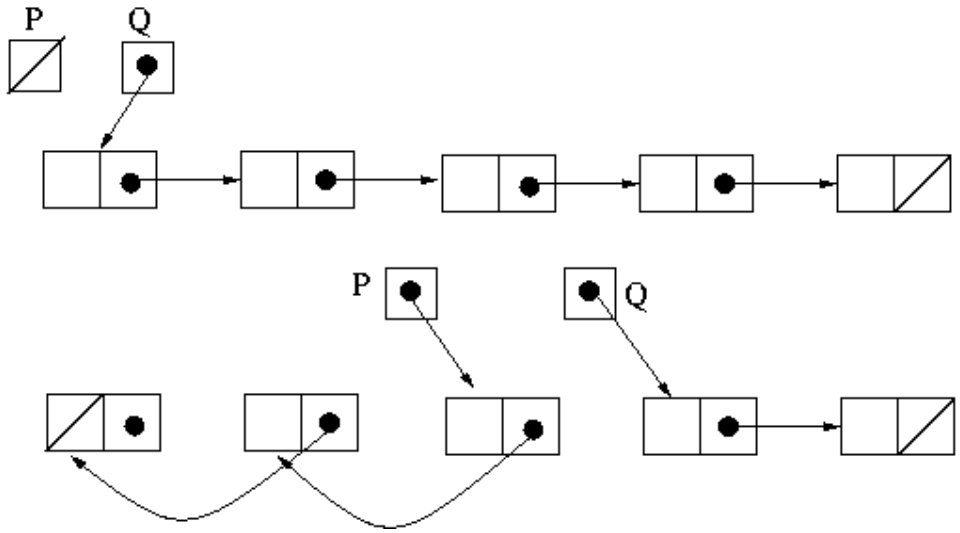
Αλγόριθμος

1. Αναζήτηση της w στη λίστα με ταυτόχρονη κράτηση δεικτών προς τα πίσω
2. Οπισθοδρόμηση κατά n θέσεις στη λίστα.

Η λύση με χρήση δεικτών προς τα πίσω είναι ακριβή σε μνήμη!!!! Γιατί?

Υπάρχει λύση φθηνή σε μνήμη?

Μέθοδος Αναστροφής Δεικτών Λίστας



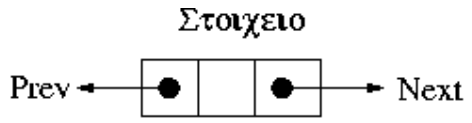
Λειτουργίες

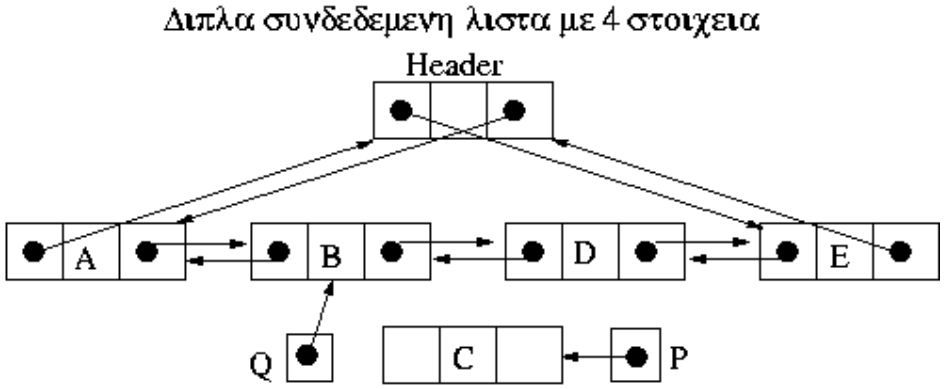
StartTraversal(L):

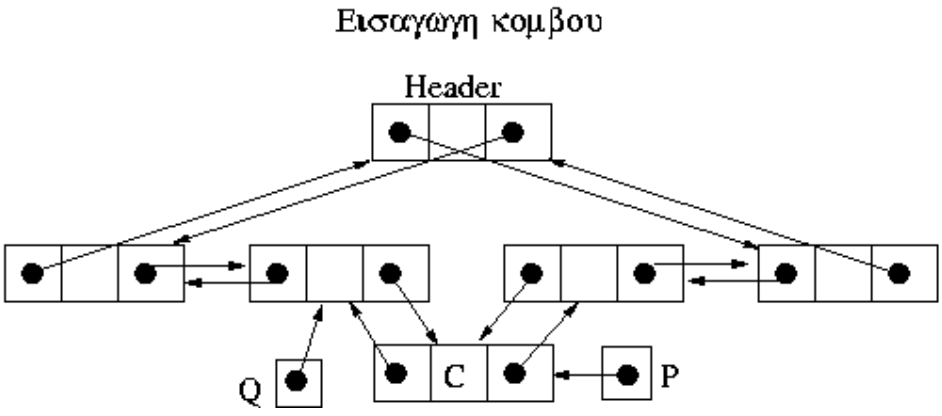
Forward(P,Q): *Back(P,Q):*

Διπλά Συνδεδεμένες Λίστες

Οι κόμβοι μιας διπλά συνδεδεμένης λίστας περιέχουν δείκτες και προς τα εμπρός και προς τα πίσω και άρα διασχίσεις Zig-Zag είναι εύκολα υλοποιήσιμες.







Λειτουργίες Διπλά Συνδεδεμένης Λίστας

Εισαγωγή κόμβου στον οποίο δείχνει ο P μετά τον κόμβο στον οποίο δείχνει ο Q:

procedure DoublyLinkedInsert(pointer P,Q):
/* insert node pointed to by P just after node pointed to by Q */

Διαγραφή κόμβου P από τη λίστα
Procedure DoublyLinkedDelete(pointer P):
/* delete node P from its doubly linked list */

Πολυπλοκότητα?

ERROR: syntaxerror
OFFENDING COMMAND: --%ztokenexec_continue--

STACK:

false