

Ενότητα 2

Στοιίβες - Ουρές - Λίστες

Λίστες

Γραμμική λίστα (linear list) είναι ένα σύνολο από $n \geq 0$ στοιχεία ή κόμβους $L_1, \dots, L_n$, τα οποία είναι διατεταγμένα με γραμμική σειρά. Το στοιχείο L_1 είναι το πρώτο στοιχείο και το στοιχείο L_n το τελευταίο στοιχείο της λίστας. Το στοιχείο L_k προηγείται του στοιχείου L_{k+1} και έπεται του στοιχείου L_{k-1} , $1 < k < n-1$.

- L_1 : κεφαλή (head)
- L_n : ουρά (tail)
- $|L|$: μήκος λίστας ($|L| = n$)
- $\langle \rangle$: κενή λίστα

Λειτουργίες που συνήθως υποστηρίζονται από λίστες

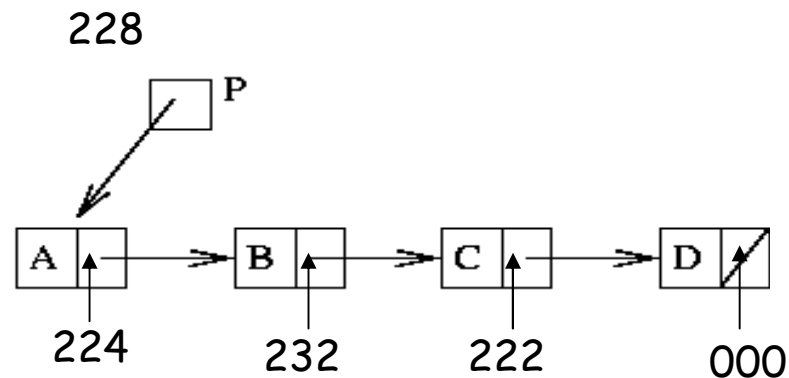
- ***Access(L, j)***: Επιστρέφει το j -οστό στοιχείο της λίστας ή ένα μήνυμα λάθους αν j είναι $> |L|$.
- ***Length(L)***: Επιστρέφει $|L|$, το μήκος της λίστας.
- ***Concat(L₁, L₂)***: Επιστρέφει μια λίστα που είναι το αποτέλεσμα της συνένωσης των δύο λιστών L_1 και L_2 σε μία.
- ***MakeEmptyList()***: επιστρέφει $\langle \rangle$, την κενή λίστα.
- ***IsEmptyList(L)***: επιστρέφει true αν $L == \langle \rangle$, false διαφορετικά.

Τρόποι Υλοποίησης Λιστών

- Στατικές Λίστες - Υλοποίηση με πίνακες
 - Όλα τα στοιχεία της λίστας αποθηκεύονται σε πίνακα.
- Συνδεδεμένες Λίστες - Χρήση δεικτών

222	D	000
224	B	232
226		
228	A	224
230		
232	C	222
234		

412	228	P
-----	-----	---



Θετικά δυναμικών έναντι στατικών λιστών

- ☺ Εισαγωγή/διαγραφή νέων στοιχείων γίνεται πολύ εύκολα
- ☺ Ο συνολικός αριθμός στοιχείων δεν χρειάζεται να είναι γνωστός εξ αρχής

Αρνητικά δυναμικών έναντι στατικών λιστών

- ☹ Απαιτούν περισσότερη μνήμη (λόγω των δεικτών).
- ☹ Ποια είναι η πολυπλοκότητα χρόνου για την ανάκτηση του j-οστού στοιχείου στη λίστα;

Στοιίβες

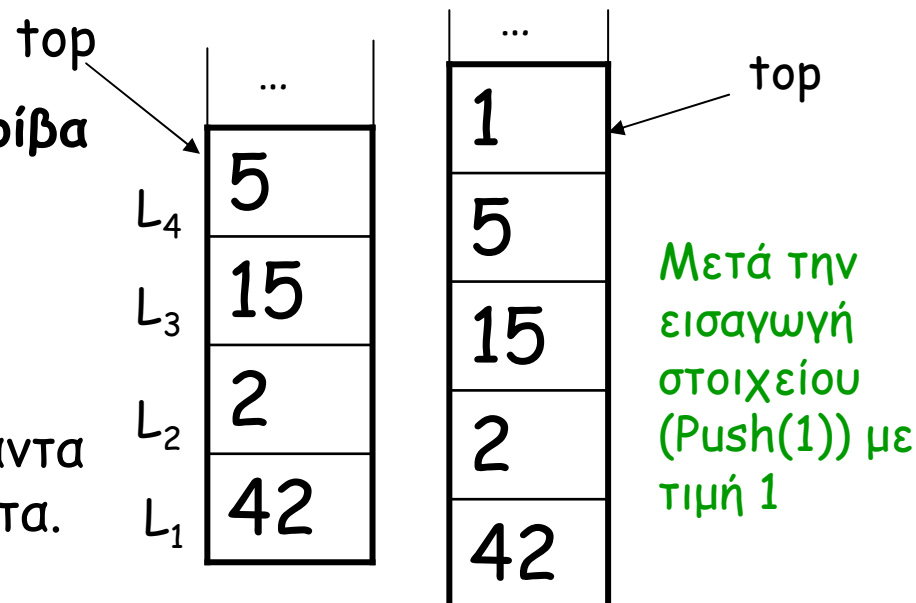
Αφηρημένος τύπος δεδομένων Στοιίβα (Stack)

Μια **στοίβα** είναι μια λίστα που υποστηρίζει εισαγωγή και διαγραφή στοιχείων μόνο στο ένα της άκρο. Το στοιχείο που αφαιρείται είναι πάντα αυτό που έχει εισαχθεί πιο πρόσφατα.

Λειτουργίες

- $Top(S)$: επιστρέφει το **κορυφαίο** στοιχείο της S (δηλαδή αυτό που έχει εισαχθεί τελευταίο)
- $Pop(S)$: (λειτουργία διαγραφής) διαγραφή και επιστροφή του κορυφαίου στοιχείου της S
- $Push(x, S)$: (λειτουργία εισαγωγής) εισαγωγή του στοιχείου x στην κορυφή της στοίβας
- $MakeEmptyStack()$: επιστρέφει την $\langle \rangle$.
- $IsEmptyStack(S)$: επιστρέφει true αν $|S| = 0$ και false διαφορετικά.

Η μέθοδος επεξεργασίας των δεδομένων της στοίβας λέγεται «**Εξαγωγή κατά ανάστροφη σειρά εισαγωγής**» (Last In - First Out, LIFO).



Στατικές Στοίβες - Υλοποίηση με Πίνακα

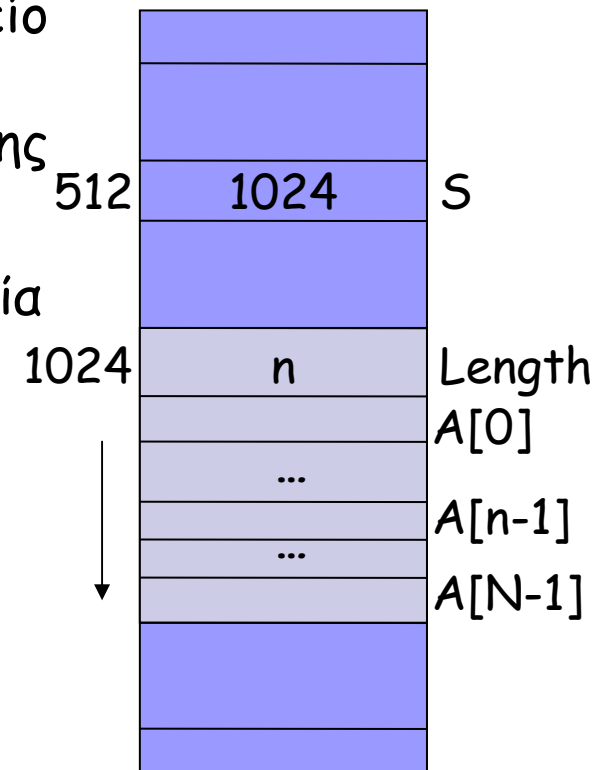
Μια **στατική** στοίβα υλοποιείται με τη χρήση ενός μονοδιάστατου πίνακα A . Ο πίνακας έχει ένα προκαθορισμένο πλήθος θέσεων N . Μια στοίβα με $n \leq N$ στοιχεία καταλαμβάνει τα στοιχεία $A[0], \dots, A[n-1]$ του πίνακα.

- Το $A[n-1]$ είναι το κορυφαίο (ή τελευταίο) στοιχείο της στοίβας
- Το $A[0]$ είναι το βαθύτερο (ή πρώτο) στοιχείο της στοίβας

Η στοίβα υλοποιείται ως ένα struct (στη C) με πεδία τον πίνακα A και έναν ακέραιο $Length$ που υποδηλώνει το μέγεθος της στοίβας (δηλαδή το τρέχον πλήθος στοιχείων στη στοίβα).

Έστω S ένας δείκτης στο struct μιας στοίβας και έστω $Type$ ο τύπος των στοιχείων της στοίβας.

- Αν $S \rightarrow Length == 0$, η στοίβα είναι άδεια.
- Αν $S \rightarrow Length == N$, η στοίβα είναι γεμάτη.



Υλοποίηση Λειτουργιών Στοίβας

```
void MakeEmptyStack(void)
    Length = 0;
```

Χρονική Πολυπλοκότητα: $\Theta(1)$

```
boolean IsEmptyStack(void)
    /* return (Length == 0); */
    if (Length == 0) return 1;
    else return 0;
```

Χρονική Πολυπλοκότητα: $\Theta(1)$

```
Type Top(void)
    if (IsEmptyStack()) then error;
    else (return(A[Length - 1]));
```

Χρονική Πολυπλοκότητα: $\Theta(1)$

Συνολικός Απαιτούμενος Χώρος Μνήμης:

Ανεξάρτητα από τον αριθμό των στοιχείων που έχουν εισαχθεί στη στοίβα: N

```
Pointer MakeEmptyStack(void)
    pointer S;
    S = newcell(STACK);
    S->Length = 0;
    return S;
```

```
boolean IsEmptyStack(pointer S)
    if (S->Length == 0) return 1;
    else return 0;
```

```
Type Top(pointer S)
    if (IsEmptyStack(S)) then error;
    else (return((S->A)[S->Length - 1]));
```

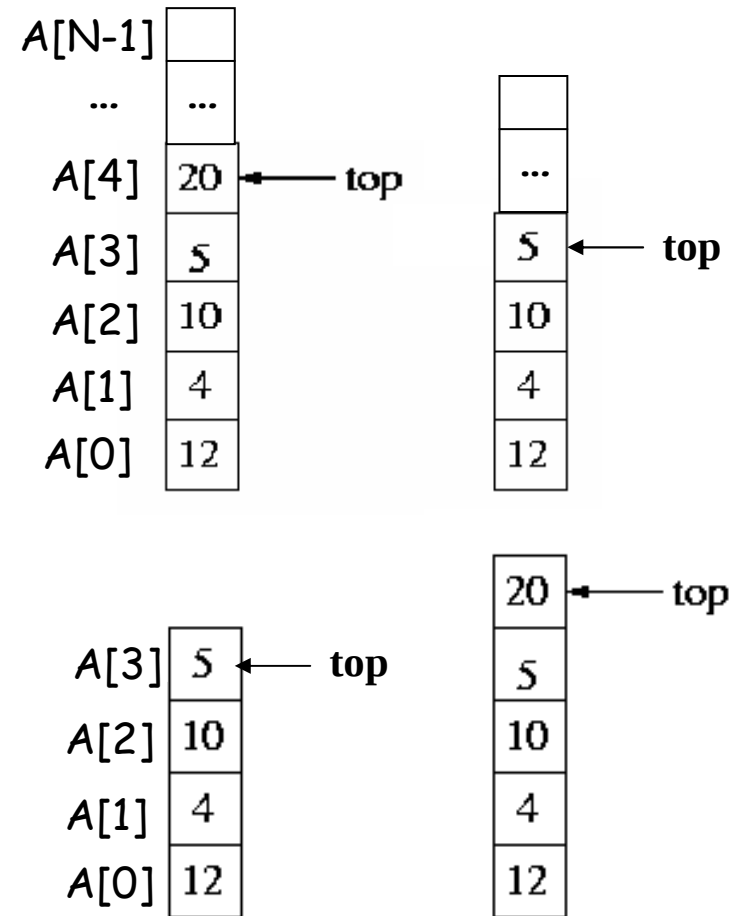
Υλοποίηση Λειτουργιών Στοίβας

```
Type Pop(void)
  if (Length == 0) then error
  else {
    x = Top();
    Length = Length - 1;
  }
  return x;
```

Χρονική Πολυπλοκότητα: $\Theta(1)$

```
void Push(Type x)
  if (Length == N) then error
  else {
    Length = Length + 1;
    A[Length-1] = x;
  }
```

Χρονική Πολυπλοκότητα: $\Theta(1)$

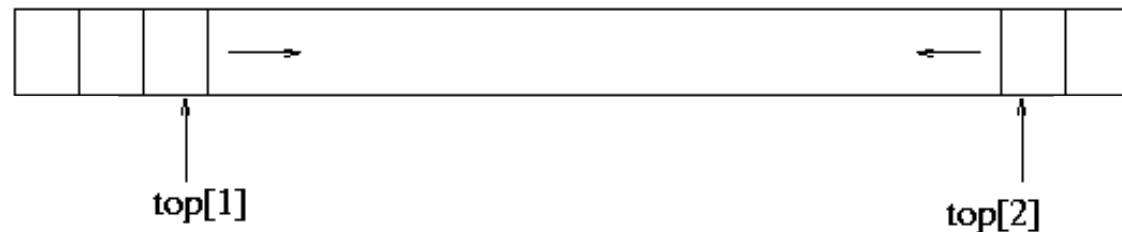


Πολλαπλή Στατική Στοίβα

Περισσότερες από μια στοίβες που υλοποιούνται χρησιμοποιώντας έναν πίνακα.

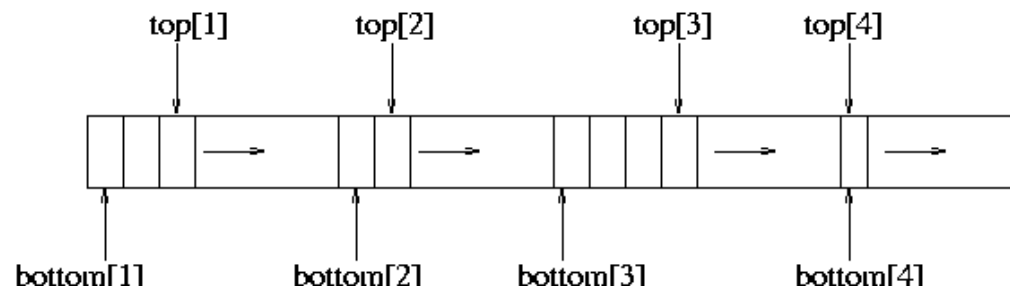
Παράδειγμα 1: Δύο Στοίβες

- Έστω $A[0...N-1]$ ο πίνακας που χρησιμοποιείται για την αποθήκευση των στοιβών.
- Η 1^η στοίβα ξεκινάει από τη θέση $A[0]$ και αναπτύσσεται προς τα δεξιά, ενώ η 2^η ξεκινάει από τη θέση $A[N-1]$ και αναπτύσσεται προς τα αριστερά.



Παράδειγμα 2: n Στοίβες

- Ο πίνακας χωρίζεται σε n ίσα τμήματα (στο παρακάτω σχήμα $n = 4$).



Στοιίβα ως Συνδεδεμένη Λίστα

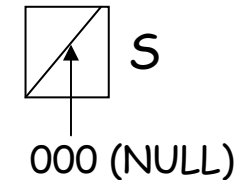
```
pointer MakeEmptyStack()
    return NULL;
```

```
boolean IsEmptyStack(pointer S)
    if (S == NULL) return TRUE;
    else return FALSE;
```

```
Type Top(pointer S)
    if IsEmptyStack(S) then
        error;
    else return S->data;
```

Χρονική Πολυπλοκότητα κάθε μιας
από τις παραπάνω λειτουργίες;
 $\Theta(1)$

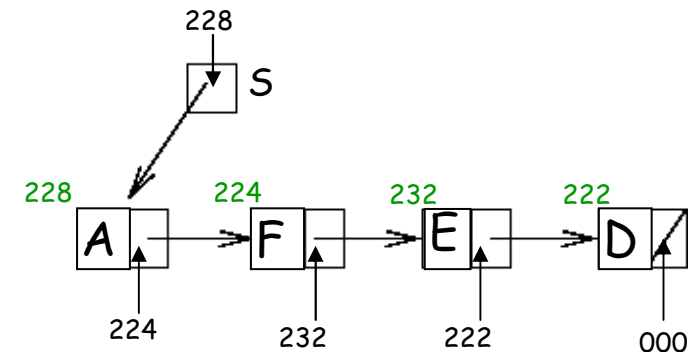
Αρχικά



Μετά την εισαγωγή των D,E,F,A
(με αυτή τη σειρά) στη στοιίβα

222	D	000
224	F	232
226		
228	A	224
230		
232	E	222
234		

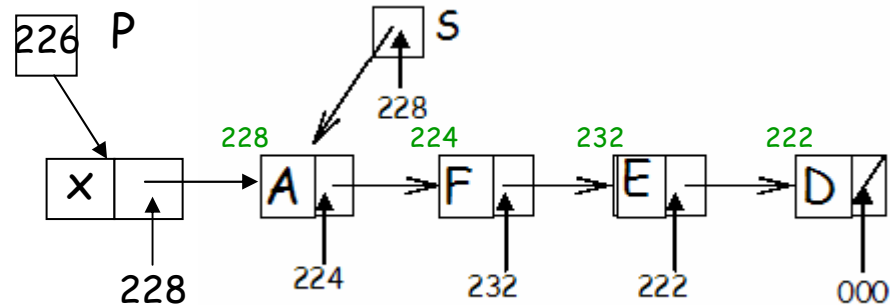
412	228	S
-----	-----	---



Εισαγωγή σε Στοίβα

222	D
223	000
224	F
225	232
226	x
227	228
228	A
229	224
230	
231	
232	E
233	222
234	
...	
412	228
413	
414	x
415	226
416	226
417	
418	
419	
420	
423	

Χώρος στη
μνήμη για τις
μεταβλητές
της Push



void Push(info x, pointer S)

pointer P; /* temporary pointer */

P = NewCell(NODE); /* malloc() */

P->data = x;

P->next = S;

S = P;

/* Αυτό στη C δεν έχει
το επιθυμητό αποτέλεσμα! */

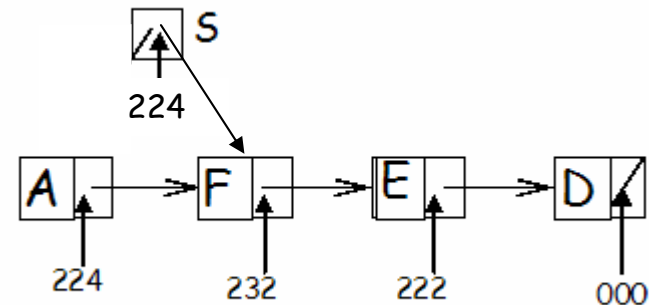
Χρονική Πολυπλοκότητα;

$\Theta(1)$

Διαγραφή από Στοιίβα

222	D
223	000
224	F
225	232
226	
227	
228	A
229	224
230	
231	
232	E
233	222
234	
...	
412	228
413	
414	224
415	A
416	
417	
418	
419	
420	
423	

Χώρος στη
μνήμη για τις
μεταβλητές
της Pop



```

info Pop(pointer S)
  char x;
  if (IsEmptyStack(S)) then error;
  else
    x = Top(S);
    S = S->next;
    return x;

```

Χρονική Πολυπλοκότητα: $\Theta(1)$

Μνήμη:
δεδομένα & (n+1) δείκτες (αν
η στοιίβα έχει n στοιχεία)

Ουρά

Αφηρημένος Τύπος Δεδομένων Ουρά (Queue)

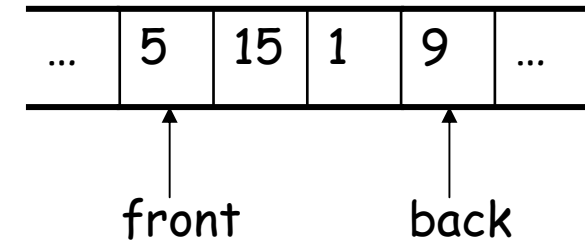
- Λίστα που μπορεί να τροποποιείται μόνο με την εισαγωγή στοιχείων στο ένα άκρο της και τη διαγραφή στοιχείων από το άλλο. Το στοιχείο που αφαιρείται είναι πάντα αυτό που έχει παραμείνει στην ουρά για το μεγαλύτερο χρονικό διάστημα.

Λειτουργίες

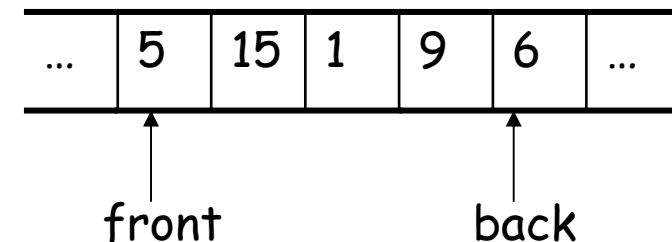
- *Enqueue(x, Q)*: Εισαγωγή στοιχείου με τιμή x στο τέλος (back) της ουράς Q
- *Dequeue(Q)*: Διαγραφή του πρώτου στοιχείου της Q (δηλαδή αυτό που βρίσκεται στην αρχή (front)) και επιστροφή της τιμής του
- *Front(Q)*: επιστρέφει το πρώτο στοιχείο της Q .
- *MakeEmptyQueue()*: επιστρέφει $\langle \rangle$, την κενή ουρά.
- *IsEmptyQueue(Q)*: επιστρέφει TRUE αν $Q == \langle \rangle$ και FALSE διαφορετικά.
- Η μέθοδος επεξεργασίας των δεδομένων ουράς λέγεται «Εξαγωγή κατά σειρά εισαγωγής» (First In - First Out, FIFO).

ΗΥ240 - Παναγιώτα Φατούρου

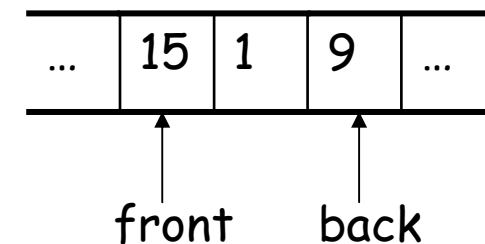
Ουρά Q



Μετά την εκτέλεση της λειτουργίας Enqueue(Q,6)



Μετά την εκτέλεση της λειτουργίας Dequeue(Q)



Στατικές Ουρές - Υλοποίηση με Πίνακα

Μια **στατική** ουρά υλοποιείται με τη χρήση ενός μονοδιάστατου πίνακα A . Ο πίνακας έχει ένα προκαθορισμένο πλήθος θέσεων N .

Η ουρά υλοποιείται ως ένα struct (στη C) με πεδία τον πίνακα A και δύο ακεραίους:

- $Length$ που υποδηλώνει το μέγεθος της ουράς (δηλαδή το τρέχον πλήθος στοιχείων της ουράς)

- $Front$ που υποδηλώνει τη θέση του πρώτου στοιχείου της ουράς στον πίνακα.

Έστω Q ένας δείκτης στο struct μιας ουράς και έστω $Type$ ο τύπος των στοιχείων της ουράς.

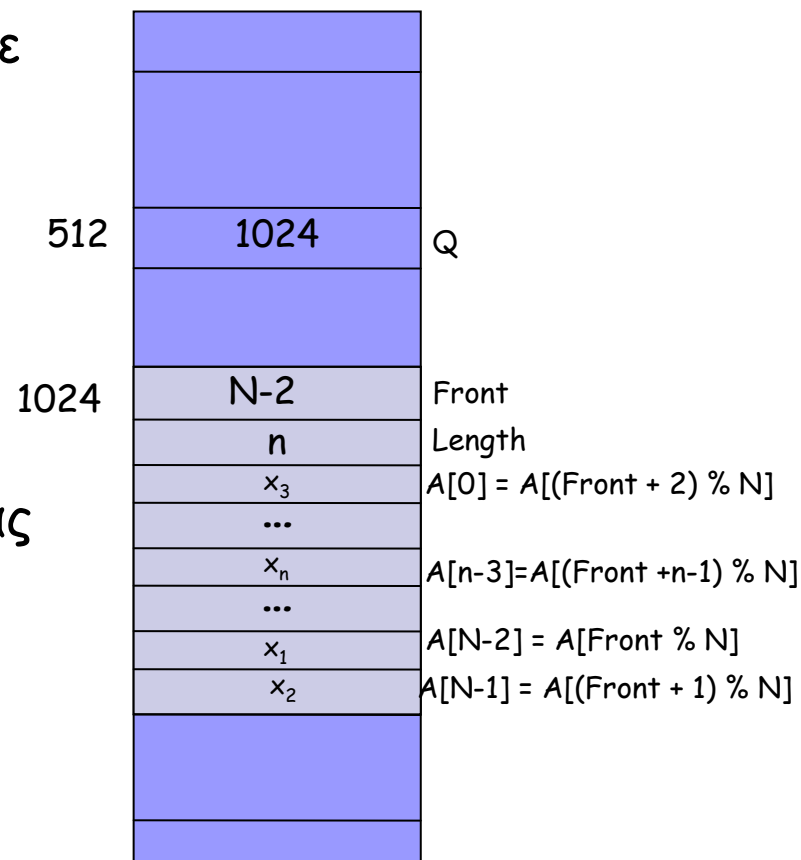
- Αν $Q \rightarrow Length == 0$, η ουρά είναι άδεια.

- Αν $Q \rightarrow Length == N$, η ουρά είναι γεμάτη.

- $x_1, \dots, x_n$: στοιχεία ουράς

- $A[Front \bmod N], A[(Front + 1) \bmod N],$

$A[(Front + 2) \bmod N], \dots, A[(Front + n - 1) \bmod N]$: θέσεις στις οποίες είναι αποθηκευμένα τα $x_1, \dots, x_n$.



Υλοποίηση Λειτουργιών Ουράς

```
void MakeEmptyQueue(void)
    pointer Q; /* temporary pointer */
    Q = NewCell(Queue); /* malloc() */
    Q->Front = 0;
    Q->Length = 0;
    return Q;
```

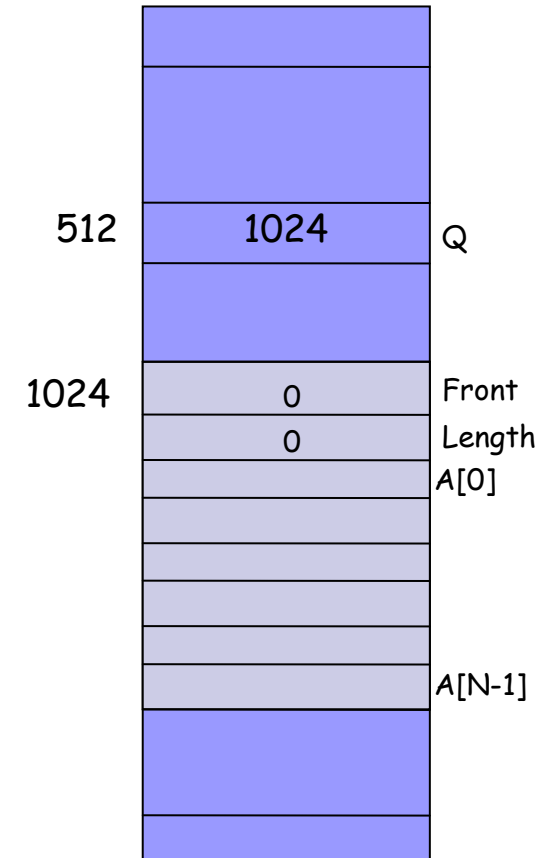
Χρονική Πολυπλοκότητα: $\Theta(1)$

```
boolean IsEmptyQueue(pointer Q)
    if (Q->Length == 0) return 1;
    else return 0;
```

Χρονική Πολυπλοκότητα: $\Theta(1)$

```
Type Front(pointer Q)
    if (IsEmptyQueue()) then error;
    else (return((Q->A)[Q->Front]));
```

Χρονική Πολυπλοκότητα: $\Theta(1)$



Συνολικός Απαιτούμενος Χώρος Μνήμης:
Ανεξάρτητα από τον αριθμό των στοιχείων που έχουν εισαχθεί στην ουρά: N

Υλοποίηση Λειτουργιών Ουράς

```

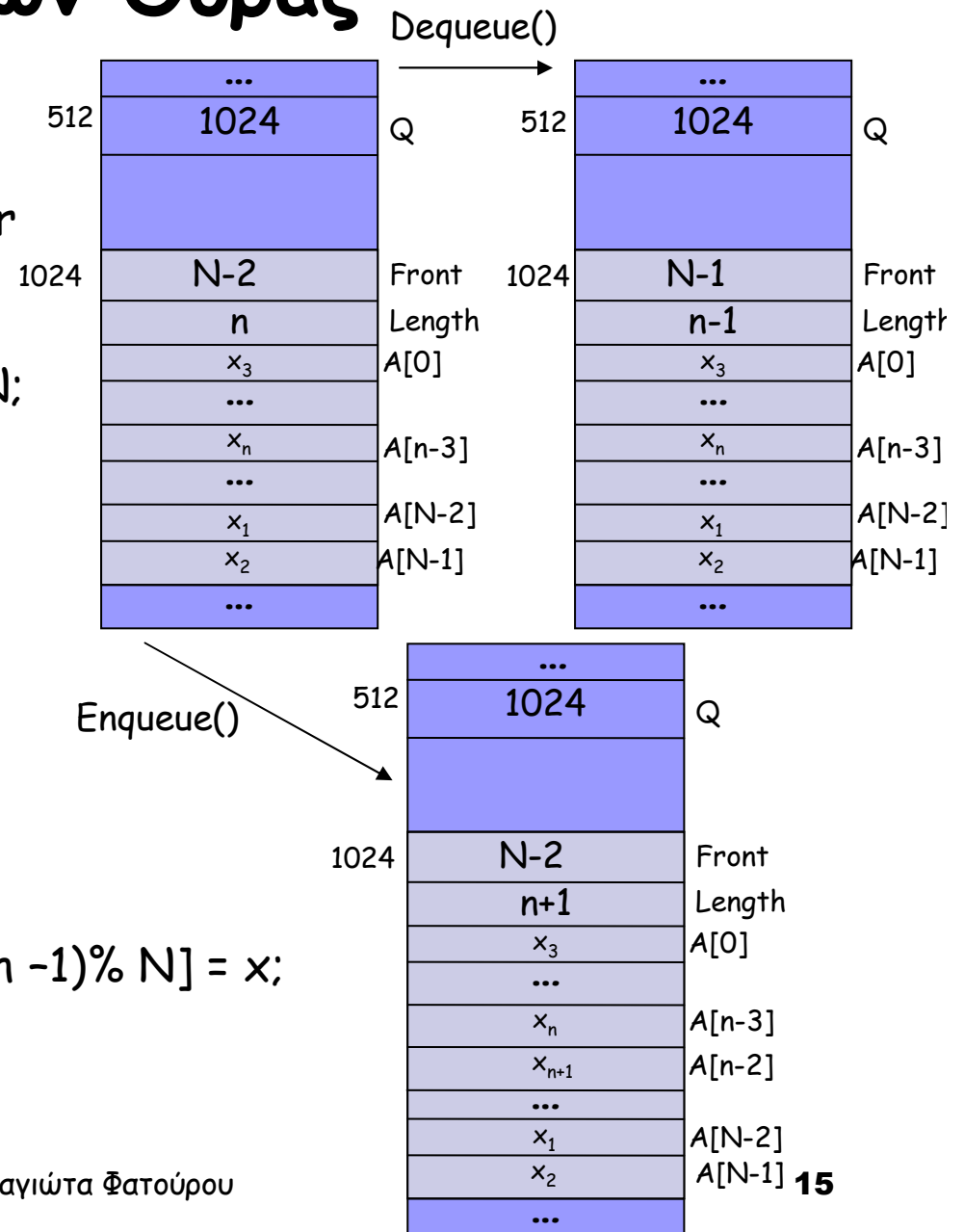
Type Dequeue(pointer Q)
  if (IsEmptyQueue(Q)) then error
  else {
    x = Front(Q);
    Q->Front = (Q->Front+1) mod N;
    Q->Length = Q->Length -1;
    return x;
  }
    
```

Χρονική Πολυπλοκότητα: $\Theta(1)$

```

void Enqueue(pointer Q, Type x)
  if (Q->Length == N) then error
  else {
    Q->Length = Q->Length+1;
    (Q->A)[(Q->Front + Q->Length -1)% N] = x;
  }
    
```

Χρονική Πολυπλοκότητα: $\Theta(1)$



Ουρά ως Συνδεδεμένη Λίστα

Node: struct με πεδία:

- data: πληροφορία αποθηκευμένη στο στοιχείο
- next: δείκτης στο επόμενο στοιχείο

Queue: struct με πεδία δύο δείκτες:

- Front: δείκτης στο πρώτο στοιχείο
- Back: δείκτης στο τελευταίο στοιχείο

pointer *MakeEmptyQueue(void)*

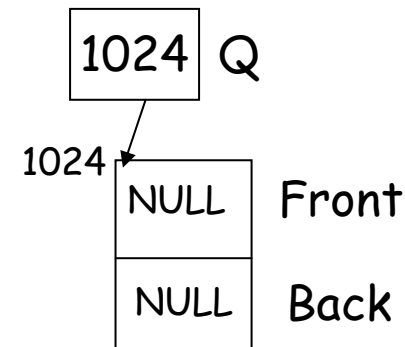
```
pointer Q; /* temporary pointer */
Q = NewCell(Queue); /* malloc */
Q->Front = Q->Back = NULL;
return Q;
```

boolean *IsEmptyQueue(pointer Q)*

```
if (Q->Front == NULL) then
    return TRUE;
else return FALSE;
```

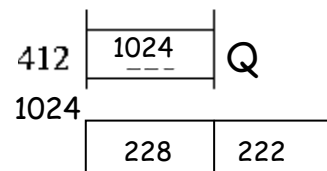
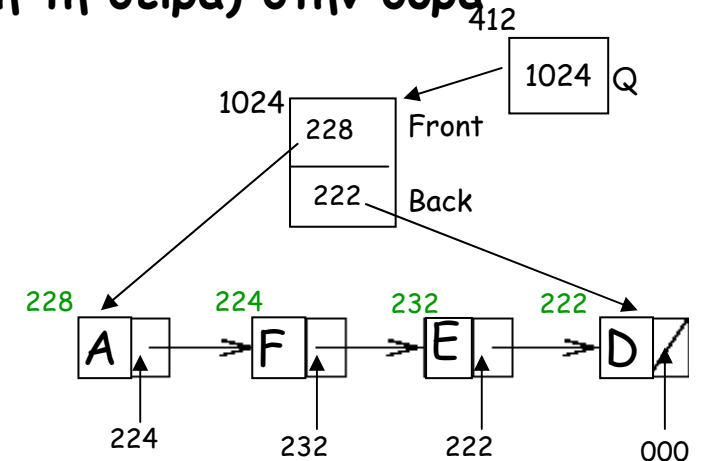
Type *Front(pointer Q)*

```
if (IsEmptyQueue(Q)) then error;
else return ((Q->Front)->data);
```



Μετά την εισαγωγή των A,F,E,D
(με αυτή τη σειρά) στην ουρά

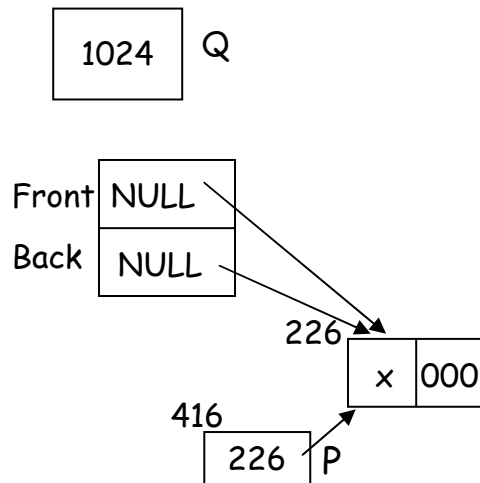
228	D	000
224	F	232
226		
228	A	224
230		
232	E	222
234		



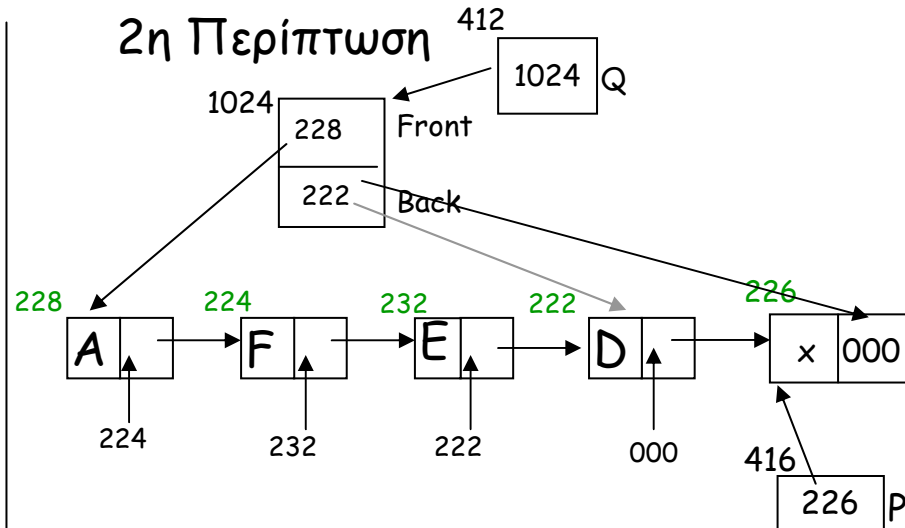
Χρονική Πολυπλοκότητα κάθε
μιας από αυτές τις λειτουργίες:
 $\Theta(1)$

Εισαγωγή σε Ουρά

1η Περίπτωση



2η Περίπτωση



void *Enqueue*(Type x, pointer Q)

```

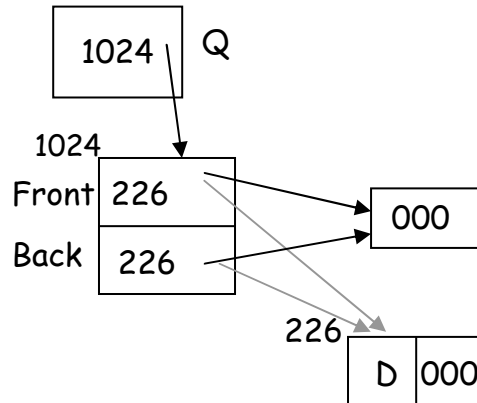
    pointer P;                      /* temporary pointer */
    P = NewCell(Node);
    P->data = x;
    P->next = NULL;
    if (IsEmptyQueue(Q)) then Q->Front = P;
    else Q->Back->next = P;
    Q->Back = P;

```

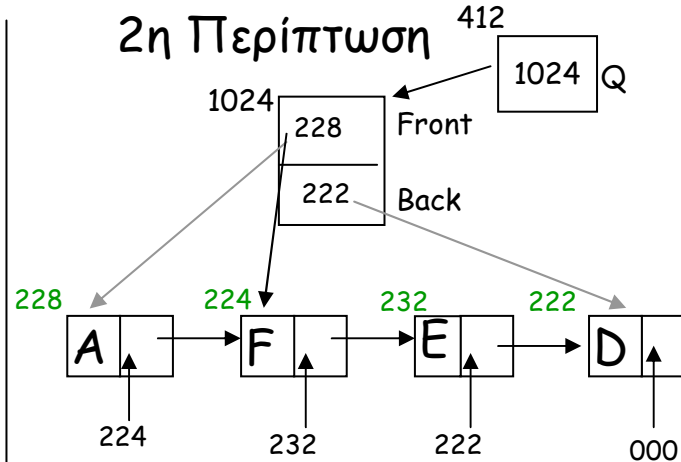
Χρονική Πολυπλοκότητα: $\Theta(1)$

Διαγραφή από Ουρά

1η Περίπτωση



2η Περίπτωση



```

Type Dequeue(pointer Q)
  if (IsEmptyQueue(Q)) then error;
  else {
    x = (Q->Front)->data;
    Q->Front = (Q->Front)->next;
    if (Q->Front == NULL) then
      Q->Back = NULL;
    return x;
  }
    
```

Χρονική Πολυπλοκότητα;

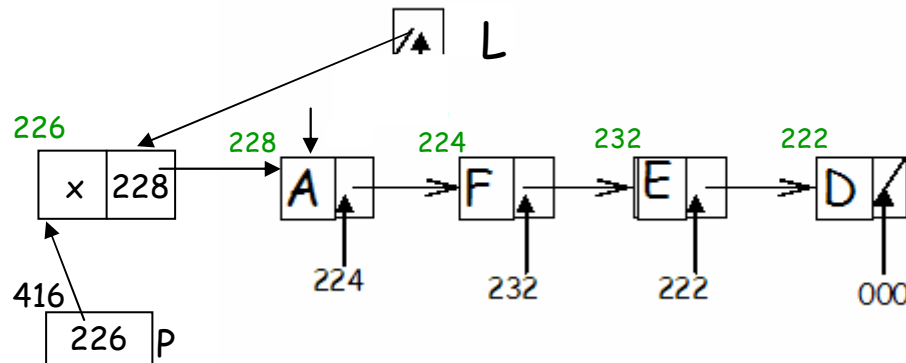
$\Theta(1)$

Μνήμη;

δεδομένα & $(n+3)$ δείκτες (αν η ουρά έχει n στοιχεία)

Συνδεδεμένες Λίστες

Έστω ότι κάθε στοιχείο της λίστας (struct node) έχει δύο πεδία, έναν ακέραιο data και το δείκτη next. Ένας δείκτης L δείχνει στο πρώτο στοιχείο της λίστας.



Εισαγωγή σε Λίστα

```
void ListInsert(int x)
{
    pointer p;
    p = newcell(node);
    p->data = x;
    p->next = L;
    L = p;
}
```

Αναζήτηση σε Λίστα

```
boolean ListSearch(Type x) {
    pointer q = L;
    while (q != NULL && q->data != x)
        q = q->next;
    return (q == NULL);
}
```

Άσκηση: Υλοποιείστε τη Delete().

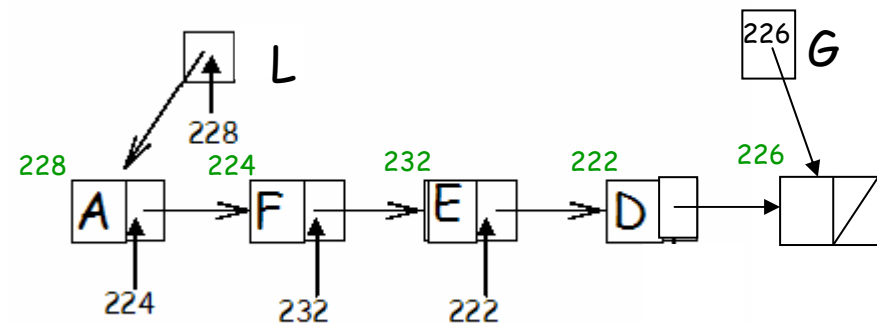
Κόμβος Φρουρός

Αναζήτηση κόμβου με συγκεκριμένη τιμή. Αν ο κόμβος δεν υπάρχει στη λίστα εισάγεται.

- ❑ Υπάρχει ένας κόμβος (που είναι τελευταίος πάντα στη λίστα) που λέγεται κόμβος φρουρός.
- ❑ Ένας δείκτης δείχνει μόνιμα σε αυτόν τον κόμβο.
- ❑ Κατά την αναζήτηση οποιουδήποτε κόμβου, η προς αναζήτηση τιμή αρχικά αποθηκεύεται στον κόμβο αυτό.
- ❑ Εκτελείται διάσχιση της λίστας και αναζήτηση της τιμής. Αν βρεθεί στον κόμβο φρουρό η τιμή αυτή δεν υπάρχει στη λίστα.

Τι κερδίζουμε με τη χρήση κόμβου φρουρού;

```
boolean ListSearch(Type x) {  
    pointer q = L;  
    while (q != NULL && q->data != x)  
        q = q->next;  
    return (q == NULL);  
}
```



Εισαγωγή Στοιχείου σε Ταξινομημένη Λίστα

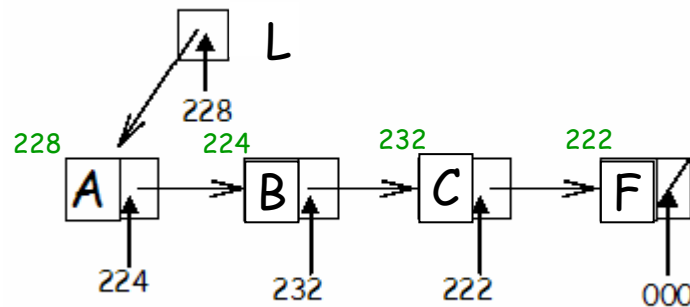
Κάθε κόμβος της λίστας περιέχει π.χ. έναν ακέραιο data και ένα δείκτη next στον επόμενο κόμβο. Έστω L ένας δείκτης στο πρώτο στοιχείο της λίστας. Η λίστα είναι ταξινομημένη.

Πρόβλημα προς επίλυση

Εισαγωγή νέου στοιχείου στη λίστα, έτσι ώστε η λίστα να εξακολουθήσει να είναι ταξινομημένη. Έστω x ο 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.

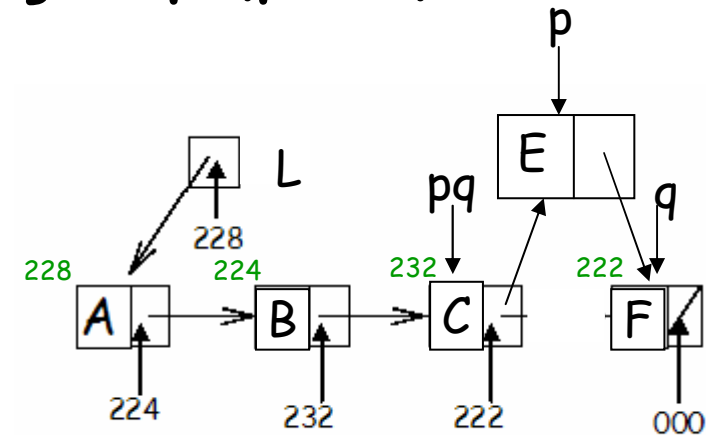


```
pointer q = L;  
while (q != NULL && q->data < x)  
    q = q->next;  
return (q == NULL);
```

Εισαγωγή Στοιχείου σε Ταξινομημένη Λίστα

```
void LLInsert(Type k, pointer L)
    pointer C, ptr; /* temporary pointers */
    q = L;
    pq = NULL;
    while (q != NULL) and (q->data > K) {
        pq = q;
        q = q->next;
    }
    if (q != NULL) and (q->data == K) then return;
        /* k is already in list */

    p = NewCell(Node);    /* malloc */
    p->data = K;
    p->next = q;
    if (pq == NULL) then L = p;
    else pq->next = p;
```



Διάσχιση Λίστας

Εκτέλεση επίσκεψης σε ένα ή σε κάποια στοιχεία μιας λίστας που έχουν μια ιδιότητα.

Θεωρούμε λίστα που περιέχει strings (αλφαριθμητικά) και είναι λεξικογραφικά ταξινομημένη.

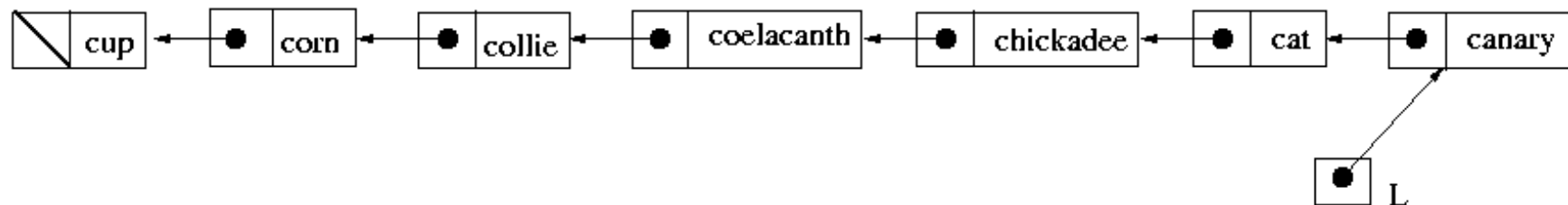
Πρόβλημα 1

Δεδομένου ενός αλφαριθμητικού w , ζητείται το τελευταίο αλφαριθμητικό στη λίστα που προηγείται αλφαβητικά του w και τελειώνει με το ίδιο γράμμα όπως το w .

Παράδειγμα

$w = \text{crabapple}$

$L = \langle \text{canary}, \text{cat}, \text{chickadee}, \text{coelacanth}, \text{collie}, \text{corn}, \text{cup} \rangle$.



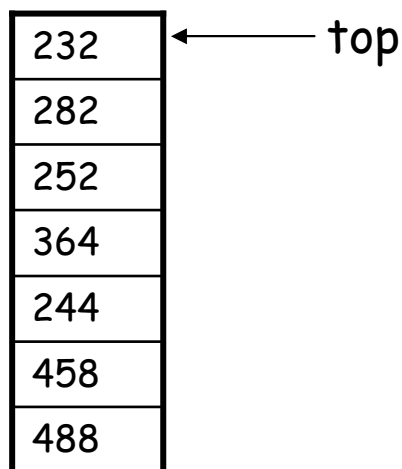
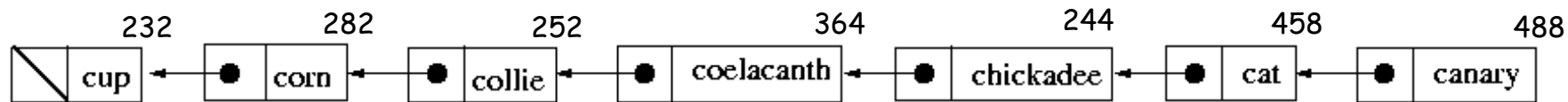
Η απάντηση θα πρέπει να είναι *collie*.

Πιθανοί Αλγόριθμοι Επίλυσης Προβλήματος 1

Αλγόριθμος 1

Διασχίζουμε τη λίστα μέχρι να βρούμε την πρώτη λέξη που είναι αλφαβητικά μεγαλύτερη από την crabapple (στο παράδειγμα την cup), κρατώντας σε μια στοίβα δείκτες στους κόμβους που διασχίσαμε.

Εξάγουμε έναν-έναν τους δείκτες από τη στοίβα και εξετάζουμε τα structs στα οποία δείχνουν (με αυτό τον τρόπο πραγματοποιούμε αντίστροφη διάσχιση της λίστας) μέχρι να βρούμε την πρώτη λέξη που τελειώνει σε e.



Είναι αυτή η πιο αποδοτική λύση;

Αλγόριθμος 2

Διασχίζουμε τη λίστα ξεκινώντας από τον 1ο κόμβο της διατηρώντας ένα βοηθητικό δείκτη στο τελευταίο στοιχείο που διασχίσαμε και είχε την επιθυμητή ιδιότητα.

Πως θα συγκρίνατε την πολυπλοκότητα των δύο αλγορίθμων?

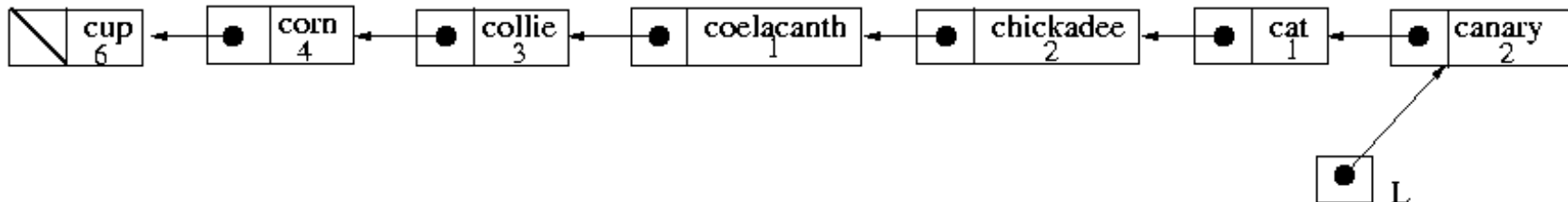
Διασχίσεις Zig-Zag

Έστω ότι κάθε κόμβος της λίστας έχει τα εξής πεδία:

- string: λέξη
- num: ακέραιος
- next: δείκτης στον επόμενο κόμβο

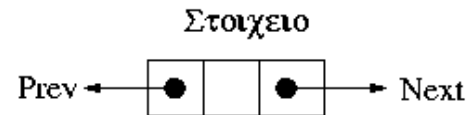
Πρόβλημα 2

Δίδεται ένα αλφαριθμητικό w . Έστω ότι το w υπάρχει στη λίστα σε κάποιον κόμβο p του οποίου το πεδίο num έχει τιμή n . Αναζητείται η τιμή του πεδίου $string$ του κόμβου που προηγείται του p κατά n θέσεις στη λίστα.

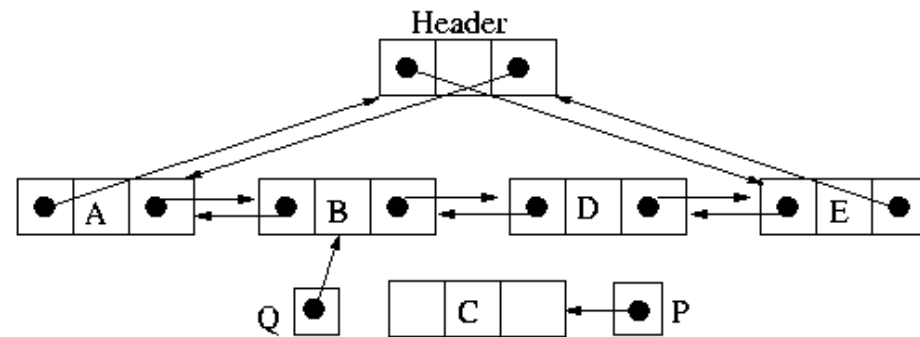


Παρουσιάστε αλγόριθμο που να επιλύει το πρόβλημα.

Διπλά Συνδεδεμένες Λίστες



Διπλά συνδεδεμένη λίστα με 4 στοιχεία



Κάθε κόμβος μιας διπλά συνδεδεμένης λίστας αποθηκεύει δείκτες και προς το επόμενο και προς το προηγούμενο στοιχείο του κόμβου.

Διασχίσεις Zig-Zag είναι εύκολα υλοποιήσιμες!

Διπλά Συνδεδεμένες Λίστες

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

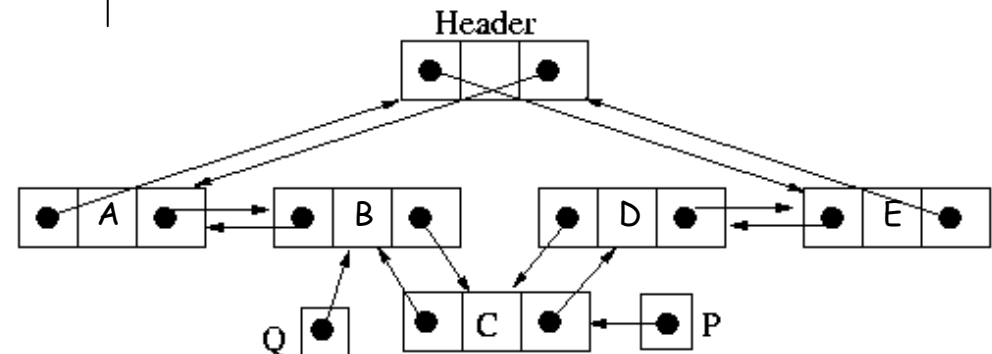
```
void DoublyLinkedInsert(pointer P,Q)
/* insert node pointed to by P just
after node pointed to by Q */
```

$$\begin{pmatrix} P \rightarrow \text{Prev} \\ P \rightarrow \text{Next} \\ Q \rightarrow \text{Next} \\ Q \rightarrow \text{Next} \rightarrow \text{Prev} \end{pmatrix} \leftarrow \begin{pmatrix} Q \\ Q \rightarrow \text{Next} \\ P \\ P \end{pmatrix}$$

Διαγραφή κόμβου P από τη λίστα

```
void DoublyLinkedDelete(pointer P)
/* delete node P from its doubly
linked list */
```

$$\begin{pmatrix} P \rightarrow \text{Prev} \rightarrow \text{Next} \\ P \rightarrow \text{Next} \rightarrow \text{Prev} \end{pmatrix} \leftarrow \begin{pmatrix} P \rightarrow \text{Next} \\ P \rightarrow \text{Prev} \end{pmatrix}$$



Τεχνικές Επιμερισμένης Ανάλυσης

Η επιμερισμένη ανάλυση μελετά τη συμπεριφορά χειρότερης περίπτωσης ενός αλγορίθμου ή δομής καθώς υποβάλλεται σε μια ακολουθία από n λειτουργίες.

■ Η αθροιστική μέθοδος

- Καθορισμός ενός πάνω φράγματος $T(n)$ στο συνολικό κόστος μιας ακολουθίας n λειτουργιών.
- Το επιμερισμένο κόστος κάθε λειτουργίας είναι $T(n)/n$.

■ Η λογιστική μέθοδος

- Καθορισμός ενός επιμερισμένου κόστους για κάθε λειτουργία. Διαφορετικές λειτουργίες μπορεί να έχουν διαφορετικά επιμερισμένα κόστη.
- Το επιμερισμένο κόστος των λειτουργιών μπορεί να είναι μεγαλύτερο ή μικρότερο από το πραγματικό τους κόστος.
- Η πίστωση από λειτουργίες με μεγαλύτερο από το πραγματικό επιμερισμένο κόστος αποθηκεύεται σε συγκεκριμένα αντικείμενα της δομής και χρησιμοποιείται αργότερα για την «πληρωμή» λειτουργιών με επιμερισμένο κόστος μικρότερο από το πραγματικό τους.

■ Η μέθοδος του δυναμικού (δεν θα διδαχθεί σε αυτό το μάθημα)

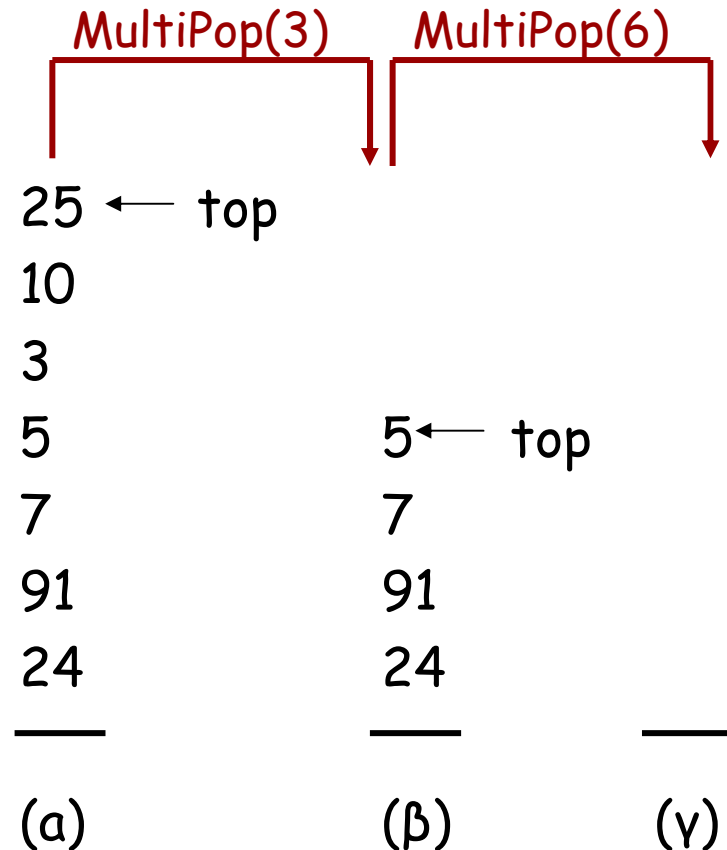
Επιμερισμένη Ανάλυση - Αθροιστική Μέθοδος

- Αποδεικνύουμε ότι $\forall n$, οποιαδήποτε ακολουθία η λειτουργιών απαιτεί συνολικά το πολύ $T(n)$ βήματα.
- Το επιμερισμένο κόστος κάθε λειτουργίας είναι επομένως $T(n)/n$.

Παράδειγμα 1 - Στοίβα με **MultiPop()**

Έστω μια δομή στοίβας που υποστηρίζει τις ακόλουθες λειτουργίες:

- **Push(x)**: Εισαγωγή του στοιχείου x στην κορυφή της στοίβας.
- **Pop()**: Διαγραφή και επιστροφή του στοιχείου που βρίσκεται στην κορυφή της στοίβας.
- **MultiPop(k)**: Διαγραφή των k πρώτων (υψηλότερων) στοιχείων της στοίβας. Αν υπάρχουν λιγότερα από k στοιχεία στη στοίβα, διαγράφονται όλα.



```
MultiPop(k) {  
    while not StackEmpty() and k ≠ 0 {  
        Pop();  
        k = k-1;  
    }  
}
```

Επιμερισμένη Ανάλυση - Αθροιστική Μέθοδος

- Η χρονική πολυπλοκότητα των $Push()$ και $Pop()$ είναι $O(1)$.
Θεωρούμε ότι το κόστος κάθε μιας εξ αυτών είναι 1.
- Ποιο είναι το κόστος της $MultiPop(k)$ αν η στοίβα περιέχει s στοιχεία?
 $O(\min\{s, k\})$
- Ποιο είναι το κόστος (χειρότερης περίπτωσης) μιας ακολουθίας n λειτουργιών στη στοίβα;
 $O(n^2)$

Εύρεση Αυστηρού Άνω Φράγματος

Ισχυρισμός: Κάθε ακολουθία από n $Push()$, $Pop()$ και $MultiPop()$ ξεκινώντας από μια άδεια στοίβα έχει χρονική πολυπλοκότητα $O(n)$.

Γιατί ισχύει αυτό;

- Το πλήθος των διαγραφών από τη στοίβα δεν μπορεί να υπερβαίνει το πλήθος των λειτουργιών $Push()$ στη στοίβα! Το πλήθος των $Pop()$ συμπεριλαμβανομένων των $Pop()$ που καλούνται από λειτουργίες $MultiPop()$ είναι το πολύ όσο το πλήθος των $Push()$.
 - Το πλήθος των λειτουργιών $Push()$ που θα εκτελεστούν είναι $O(n)$.
- Η επιμερισμένη χρονική πολυπλοκότητα κάθε λειτουργίας είναι $O(n)/n = O(1)$!



Επιμερισμένη Ανάλυση - Λογιστική Μέθοδος

- Καθορισμός του επιμερισμένου κόστους κάθε λειτουργίας. Διαφορετικές λειτουργίες μπορεί να έχουν διαφορετικά επιμερισμένα κόστη.
- Το επιμερισμένο κόστος των λειτουργιών μπορεί να είναι μεγαλύτερο ή μικρότερο από το πραγματικό τους κόστος. Το «κέρδος» από λειτουργίες με μεγαλύτερο από το πραγματικό επιμερισμένο κόστος αποθηκεύεται σε συγκεκριμένα αντικείμενα της δομής ως πίστωση και χρησιμοποιείται αργότερα για την «πληρωμή» λειτουργιών με επιμερισμένο κόστος μικρότερο από το πραγματικό τους.
- Το συνολικό επιμερισμένο κόστος οποιασδήποτε ακολουθίας λειτουργιών πρέπει να αποτελεί άνω φράγμα του συνολικού πραγματικού κόστους της ακολουθίας $\Rightarrow$ Το συνολικό κέρδος (πίστωση) που είναι συσχετισμένο με τα αντικείμενα της δομής κάθε χρονική στιγμή πρέπει να είναι μη-αρνητικό.

Παρατήρηση

Σε αντίθεση με την αθροιστική μέθοδο, η λογιστική μέθοδος δεν αποδίδει το ίδιο επιμερισμένο κόστος σε κάθε λειτουργία.

Επιμερισμένη Ανάλυση - Λογιστική Μέθοδος

Παράδειγμα 1 - Στοίβα που υποστηρίζει τη λειτουργία MultiPop()

Πραγματικό Κόστος Λειτουργιών		Επιμερισμένο Κόστος Λειτουργιών	
Push()	1	Push()	2
Pop()	1	Pop()	0
MultiPop(k)	$\min\{k,s\}$	MultiPop(k)	0

Το επιμερισμένο κόστος κάθε λειτουργίας είναι $O(1)$!

Θα αποδείξουμε ότι για οποιαδήποτε ακολουθία η λειτουργιών, το συνολικό επιμερισμένο κόστος αποτελεί άνω φράγμα του συνολικού πραγματικού κόστους.

- ❑ Υποθέτουμε ότι κάθε μονάδα κόστους αναπαρίσταται από 1 ευρώ.
- ❑ Κάθε φορά που πραγματοποιείται μια Push(), το 1 εκ των 2 ευρώ χρησιμοποιείται για το κόστος της Push(), ενώ το άλλο αποθηκεύεται στο νέο στοιχείο που εισάγεται στη δομή.
- ❑ Το έξτρα ευρώ που είναι αποθηκευμένο σε κάθε στοιχείο της δομής θα χρησιμοποιηθεί για την «πληρωμή» της Pop() του στοιχείου από τη δομή (είτε αυτή καλείται άμεσα από τον χρήστη είτε έμμεσα μέσω μιας MultiPop()).