Ενότητα 2
Στοίβες – Ουρές - Λίστες

Λίστες

Γραμμική λίστα (linear list) είναι ένα σύνολο από έστω $n \geq 0$ στοιχεία ή κόμβους, e_1, ..., e_n, τα οποία είναι διατεταγμένα με γραμμική σειρά. Το e_1 είναι το πρώτο στοιχείο της λίστας και το e_n είναι το τελευταίο στοιχείο της λίστας. Το στοιχείο e_k προηγείται του στοιχείου e_{k+1} και έπεται του στοιχείου e_{k-1}, $1 < k < n$.

- $|L|$: μήκος λίστας ($|L| = n$)
- <>: κενή λίστα

Λειτουργίες που συνήθως υποστηρίζονται από λίστες

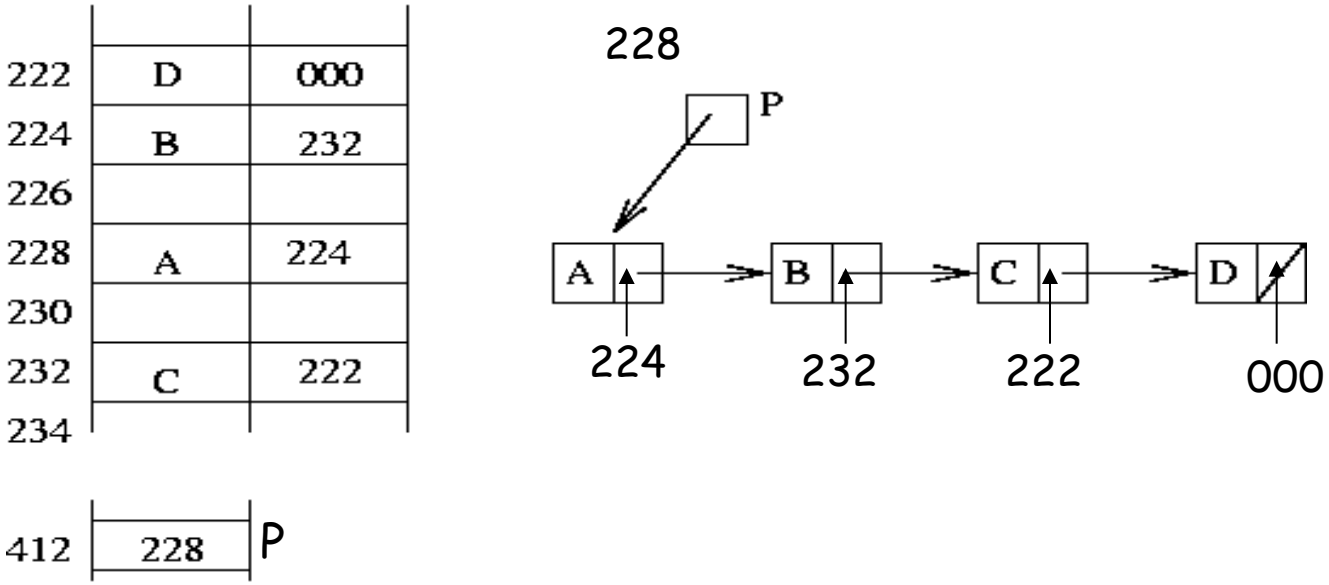
- ***Access(L,j):*** Επιστρέφει το j-οστό στοιχείο της λίστας ή ένα μήνυμα λάθους αν j είναι > $|L|$.
- ***Length(L):*** Επιστρέφει $|L|$, το μήκος της λίστας.
- ***Concat(L_1,L_2):*** Επιστρέφει μια λίστα που είναι το αποτέλεσμα της συνένωσης των δύο λιστών L_1 και L_2 σε μία.
- ***MakeEmptyList():***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

Τρόποι Υλοποίησης Λιστών

Στατικές Λίστες – Υλοποίηση με πίνακες

- Όλα τα στοιχεία της λίστας αποθηκεύονται σε πίνακα.

Συνδεδεμένες Λίστες – Χρήση δεικτών



Σχήμα 1.4: Lewis & Denenberg, Data Structures & Their Algorithms, Addison-Wesley, 1991

Τρόποι Υλοποίησης Λιστών

Θετικά δυναμικών έναντι στατικών λιστών

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

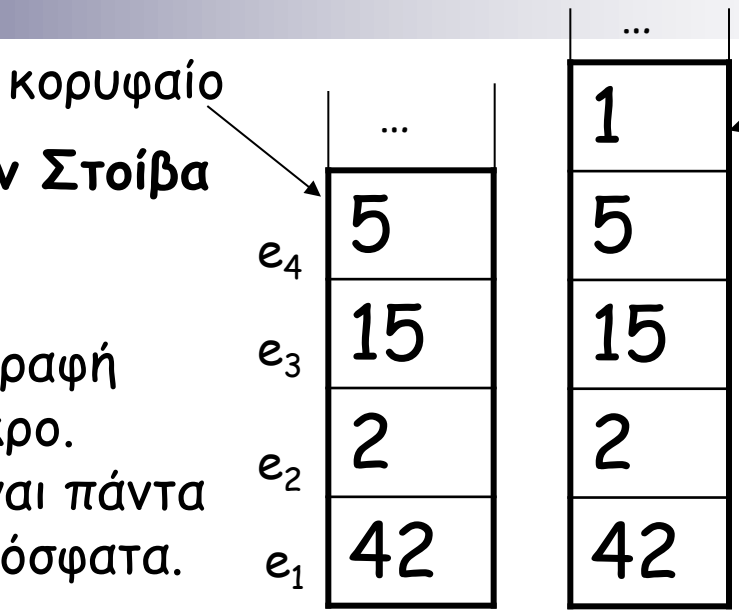
Αρνητικά δυναμικών έναντι στατικών λιστών

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

Στοίβες

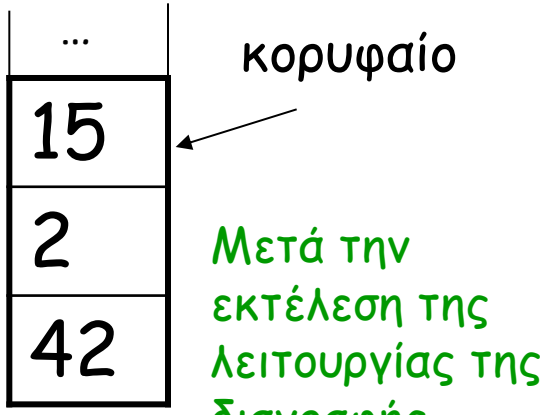
Αφηρημένος τύπος δεδομένων Στοίβα (Stack)

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)*: διαγραφή και επιστροφή του κορυφαίου στοιχείου της S
- *Push(x,S)*: εισαγωγή του στοιχείου x στην κορυφή της στοίβας
- *MakeEmptyStack()*: επιστρέφει την <>.
- *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

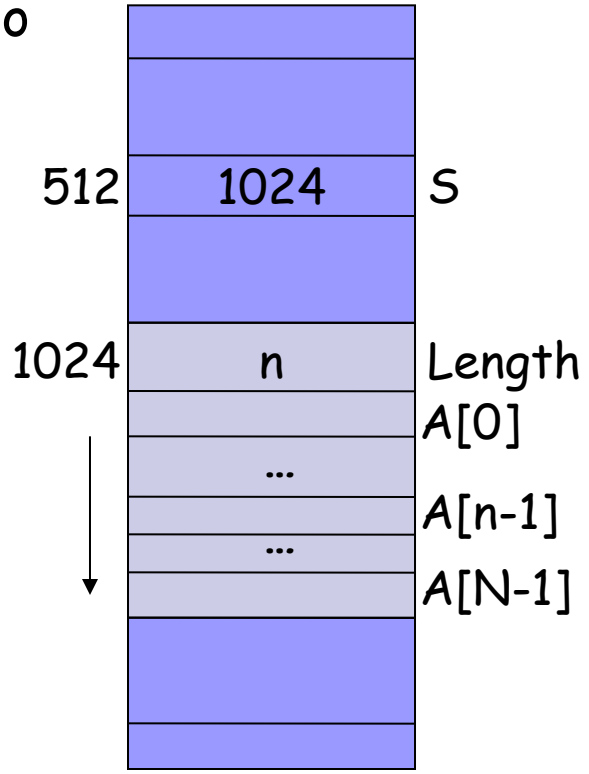
Μια στατική στοίβα υλοποιείται με τη χρήση ενός μονοδιάστατου πίνακα *A* και ενός ακεραίου Length που υποδηλώνει το τρέχον μέγεθος της στοίβας. Ο πίνακας έχει ένα προκαθορισμένο πλήθος θέσεων N. Μια στοίβα με n ≤ N στοιχεία καταλαμβάνει τα στοιχεία *A*[0], … , *A*[n-1] του πίνακα.

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

Έστω Type ο τύπος των στοιχείων της στοίβας.

Έστω S ένας δείκτης σε ένα struct που έχει δύο πεδία, τον πίνακα *A* και τον ακέραιο Length και αναπαριστά μια στοίβα.

- Αν S->Length == 0, η στοίβα είναι άδεια.
- Αν S->Length == N, η στοίβα είναι γεμάτη.



Υλοποίηση Λειτουργιών Στοίβας

Pointer *MakeEmptyStack(void)*

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

Χρονική Πολυπλοκότητα: $\Theta(1)$

boolean *IsEmptyStack(pointer S)*

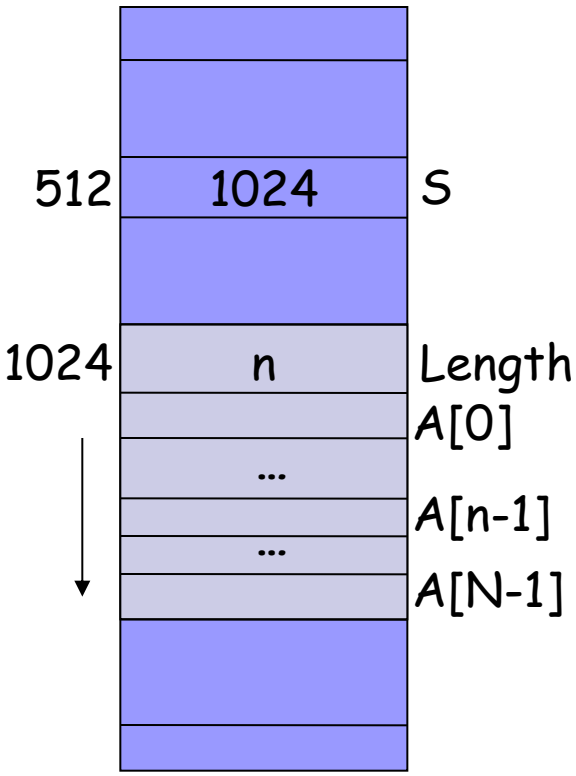
```
if (S->Length == 0) return 1;
else return 0;
```

Χρονική Πολυπλοκότητα: $\Theta(1)$

Type *Top(pointer S)*

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

Χρονική Πολυπλοκότητα: $\Theta(1)$



Συνολικός Απαιτούμενος Χώρος Μνήμης:
Ανεξάρτητα από τον αριθμό των στοιχείων που έχουν εισαχθεί στη στοίβα: N

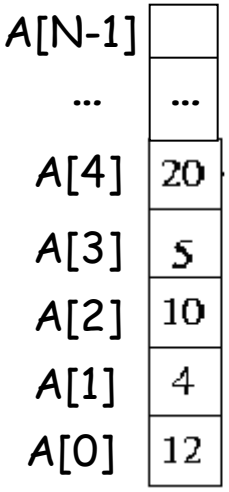
Υλοποίηση Λειτουργιών Στοίβας

```
Type Pop(Pointer S)
    if (IsEmptyStack(S)) then error
    else {
        x = Top(S);
        S->Length = S->Length –1;
    }
    return x;
```

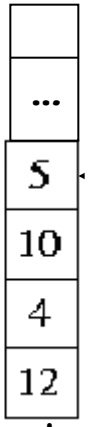
Χρονική Πολυπλοκότητα; $\Theta(1)$

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

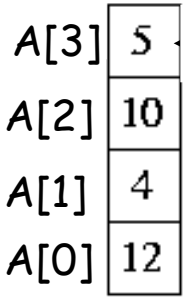
Χρονική Πολυπλοκότητα; $\Theta(1)$



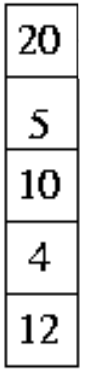
Length = 5



Length = 4



Length = 4



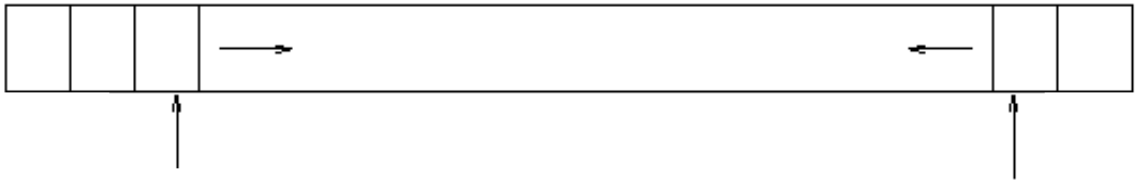
Length = 5

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 1η στοίβα ξεκινάει από τη θέση A[0] και αναπτύσσεται προς τα δεξιά, ενώ η 2η ξεκινάει από τη θέση A[N-1] και αναπτύσσεται προς τα αριστερά.

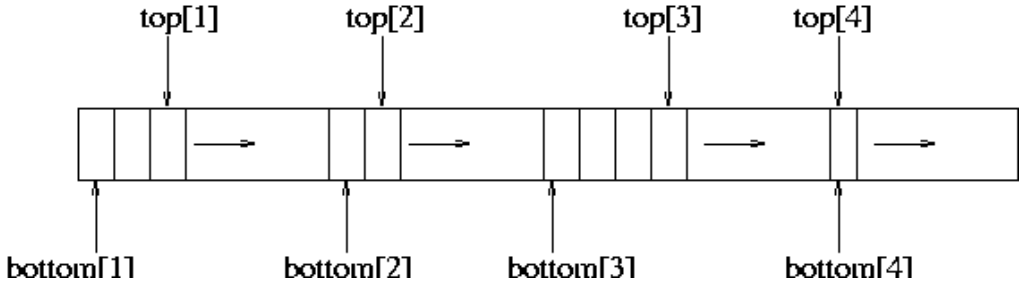


κορυφαίο στοιχείο στοίβας 1

κορυφαίο στοιχείο στοίβας 2

Παράδειγμα 2: k Στοίβες

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



Στοίβα ως Συνδεδεμένη Λίστα

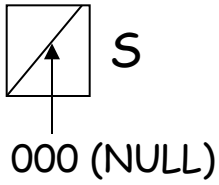
pointer *MakeEmptyStack()*
    return NULL;

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

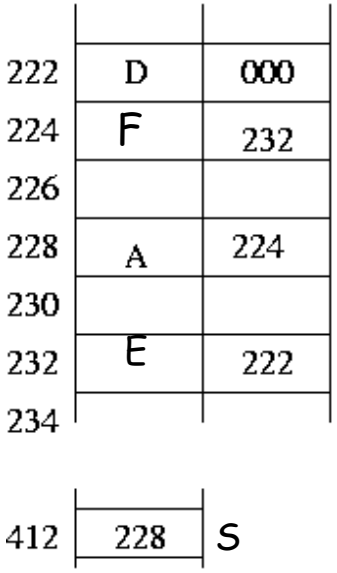
Type *Top(pointer S)*
if IsEmptyStack(S) then
    error;
else return S->data;

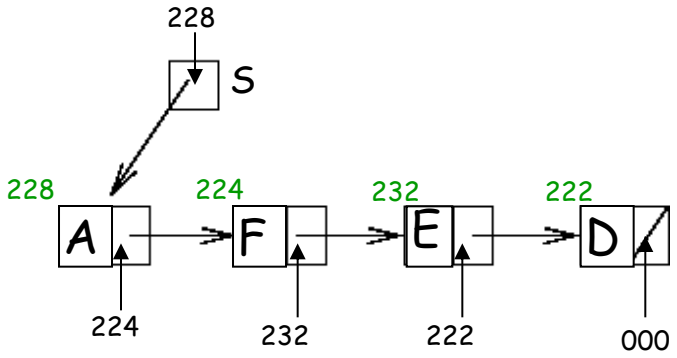
Χρονική Πολυπλοκότητα κάθε μιας από τις παραπάνω λειτουργίες;
Θ(1)

Αρχικά

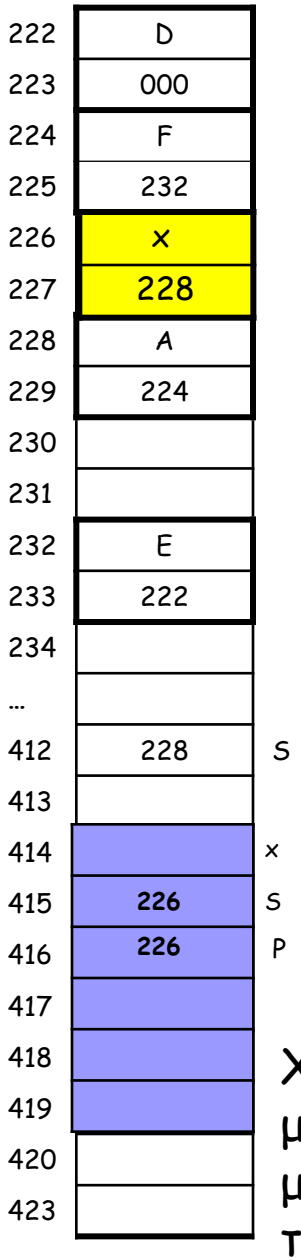


Μετά την εισαγωγή των D,E,F,A (με αυτή τη σειρά) στη στοίβα

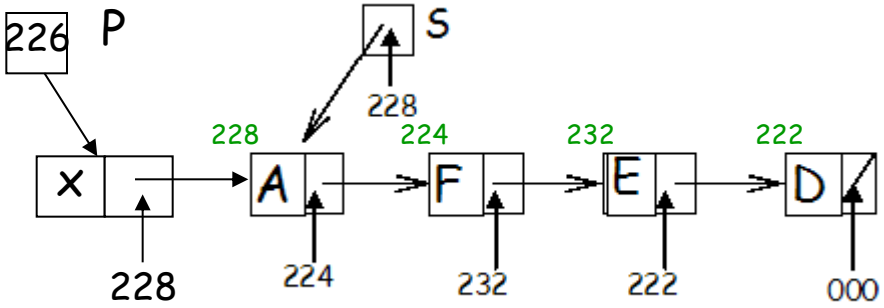




Εισαγωγή σε Στοίβα



Χώρος στη μνήμη για τις μεταβλητές της Push



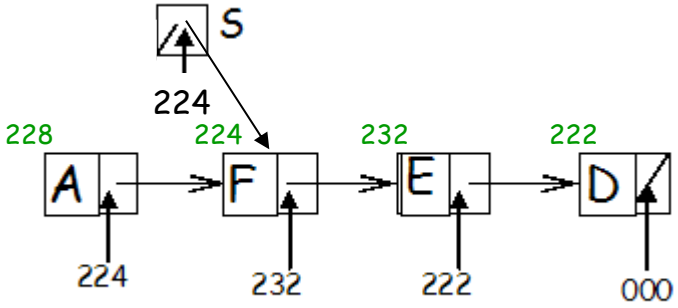

```
pointer Push(info x, pointer S)
    pointer P;      /* temporary pointer */
    P = NewCell(NODE);   /* malloc() */
    P->data = x;
    P->next = S;
   S = P;
   return S;
```

Χρονική Πολυπλοκότητα; Θ(1)

Διαγραφή από Στοίβα

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

Χώρος στη μνήμη για τις μεταβλητές της Pop




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

Χρονική Πολυπλοκότητα;
Θ(1)

Μνήμη;
δεδομένα & (n+1) δείκτες (αν η στοίβα έχει n στοιχεία)

Ουρά

Αφηρημένος Τύπος Δεδομένων Ουρά (Queue)

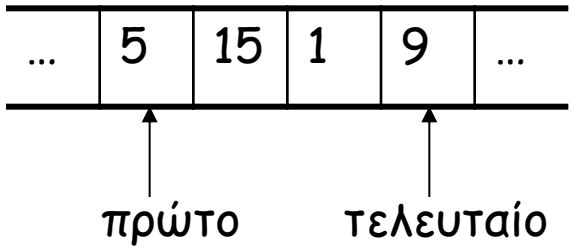
- Λίστα που μπορεί να τροποποιείται μόνο με την εισαγωγή στοιχείων στο ένα άκρο της και τη διαγραφή στοιχείων από το άλλο. Το στοιχείο που αφαιρείται είναι πάντα αυτό που έχει παραμείνει στην ουρά για το μεγαλύτερο χρονικό διάστημα.

Λειτουργίες

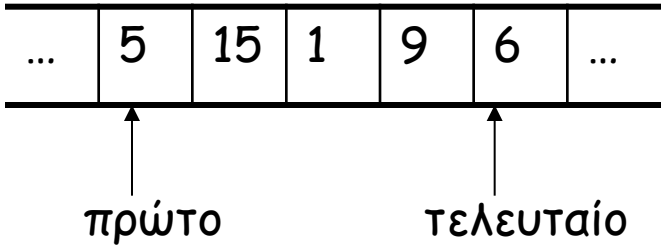
- *Enqueue(x,Q)*: Εισαγωγή στοιχείου με τιμή x στο τέλος (back) της ουράς Q
- *Dequeue(Q)*: Διαγραφή του πρώτου στοιχείου (front) της Q και επιστροφή της τιμής του
- *Front(Q)*: επιστρέφει το πρώτο στοιχείο της Q.
- *MakeEmptyQueue()*: επιστρέφει <>, την κενή ουρά.
- *IsEmptyQueue(Q)*: επιστρέφει TRUE αν Q == <> και FALSE διαφορετικά.

- Η μέθοδος επεξεργασίας των δεδομένων ουράς λέγεται **«Εξαγωγή κατά σειρά εισαγωγής» (First In – First Out, FIFO)**.

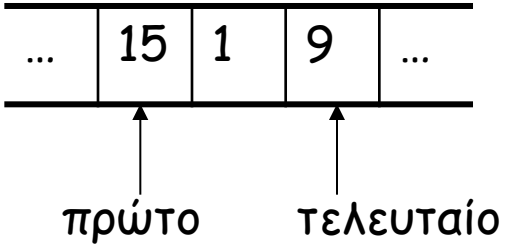
Ουρά Q

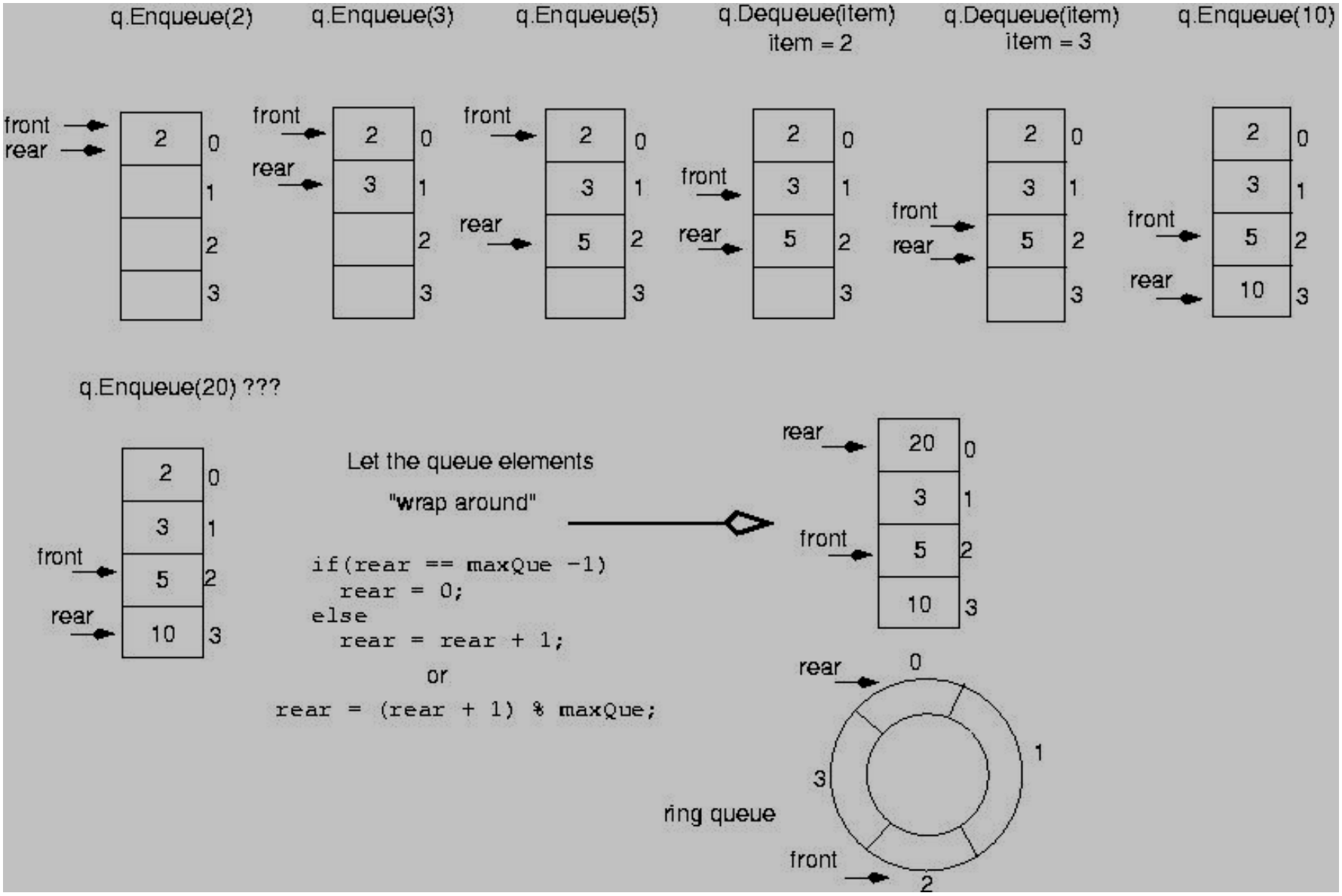


Μετά την εκτέλεση της λειτουργίας Enqueue(Q,6)

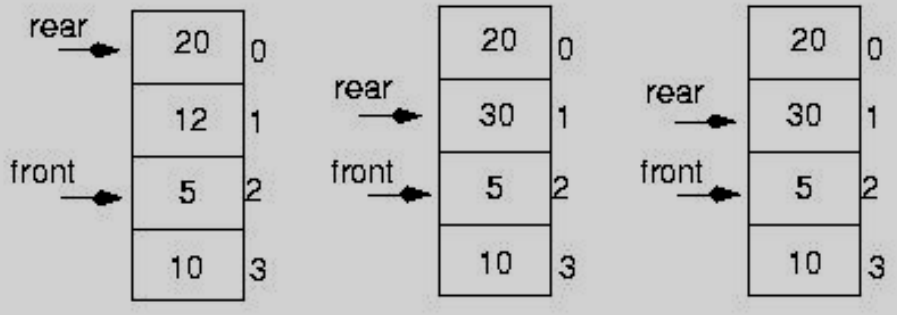


Μετά την εκτέλεση της λειτουργίας Dequeue(Q))




q.Enqueue(2)
q.Enqueue(3)
q.Enqueue(5)
q.Dequeue(item)
item = 2
q.Dequeue(item)
item = 3
q.Enqueue(10)
front
rear
2
3
5
10
0
1
2
3
q.Enqueue(20) ???
Let the queue elements
"wrap around"
if(rear == maxQue -1)
rear = 0;
else
rear = rear + 1;
or
rear = (rear + 1) % maxQue;
20
ring queue

q.Enqueue(30) q.Enqueue(50) ???

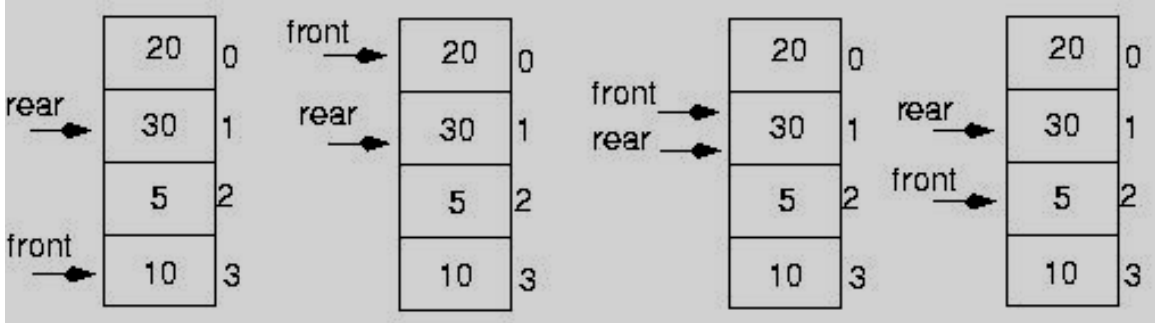


The queue is full !!

What is the condition for a full queue ?

```
rear + 1 == front
```

q.Dequeue(item) item = 5 q.Dequeue(item) item = 10 q.Dequeue(item) item = 20 q.Dequeue(item) item = 30



The queue is empty !!

What is the condition for an empty queue ?

```
rear + 1 == front
```

We cannot distinguish between the two cases !!!

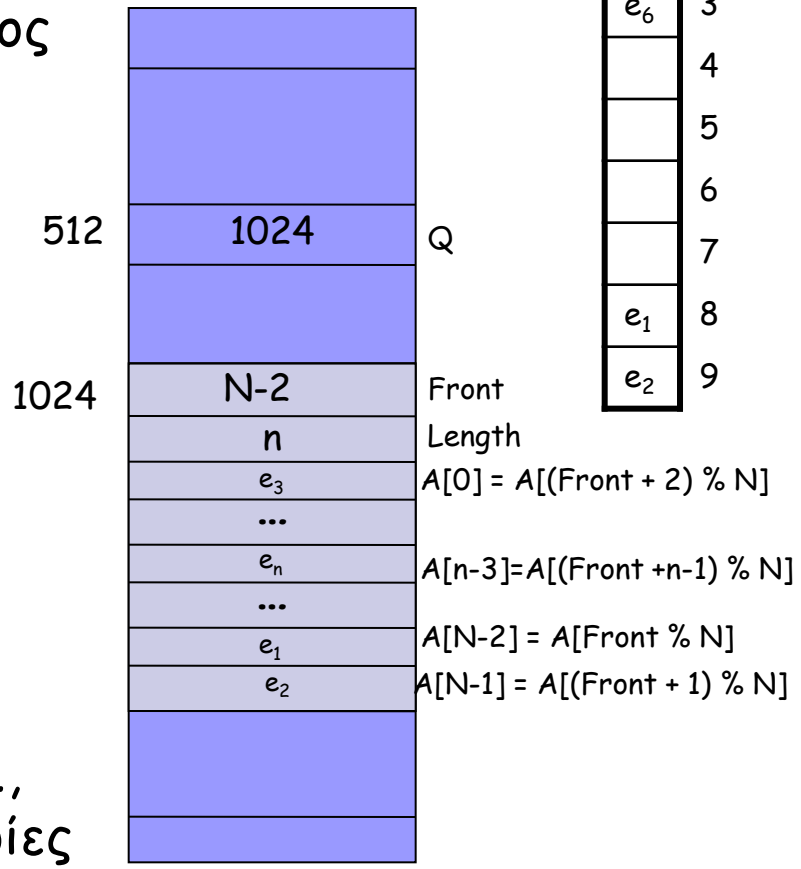
Στατικές Ουρές – Υλοποίηση με Πίνακα

Η στατική ουρά υλοποιείται ως ένα struct (στη C) με πεδία έναν πίνακα A με προκαθορισμένο πλήθος θέσεων N και δύο ακεραίους:

- Length που υποδηλώνει το τρέχον μέγεθος 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->Length == 0, η ουρά είναι άδεια.
- Αν Q->Length == N, η ουρά είναι γεμάτη.
- e_1, …, e_n: στοιχεία ουράς
- A[Front mod N], A[(Front + 1) mod N], …, A[(Front + n - 1) mod N]: θέσεις στις οποίες είναι αποθηκευμένα τα e_1, …, e_n.



Υλοποίηση Λειτουργιών Ουράς

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

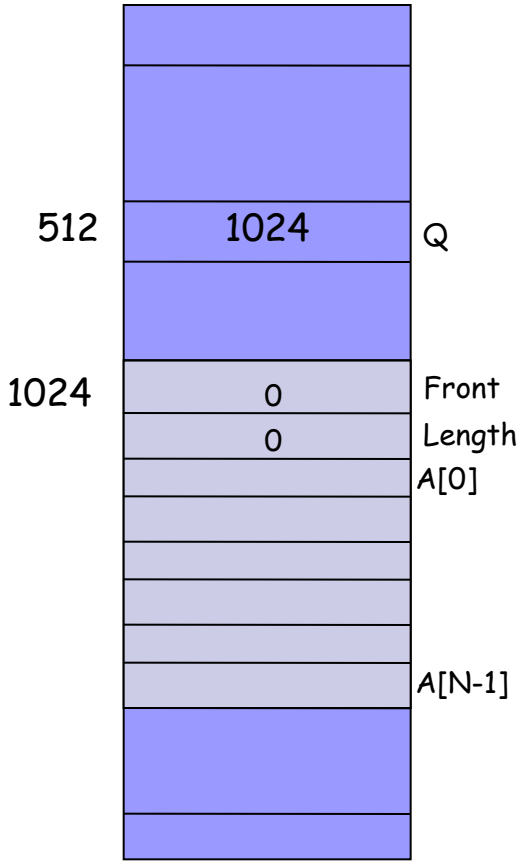
Χρονική Πολυπλοκότητα: $\Theta(1)$

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

Χρονική Πολυπλοκότητα: $\Theta(1)$

```
Type Front(pointer Q)
    if (IsEmptyQueue(Q)) 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

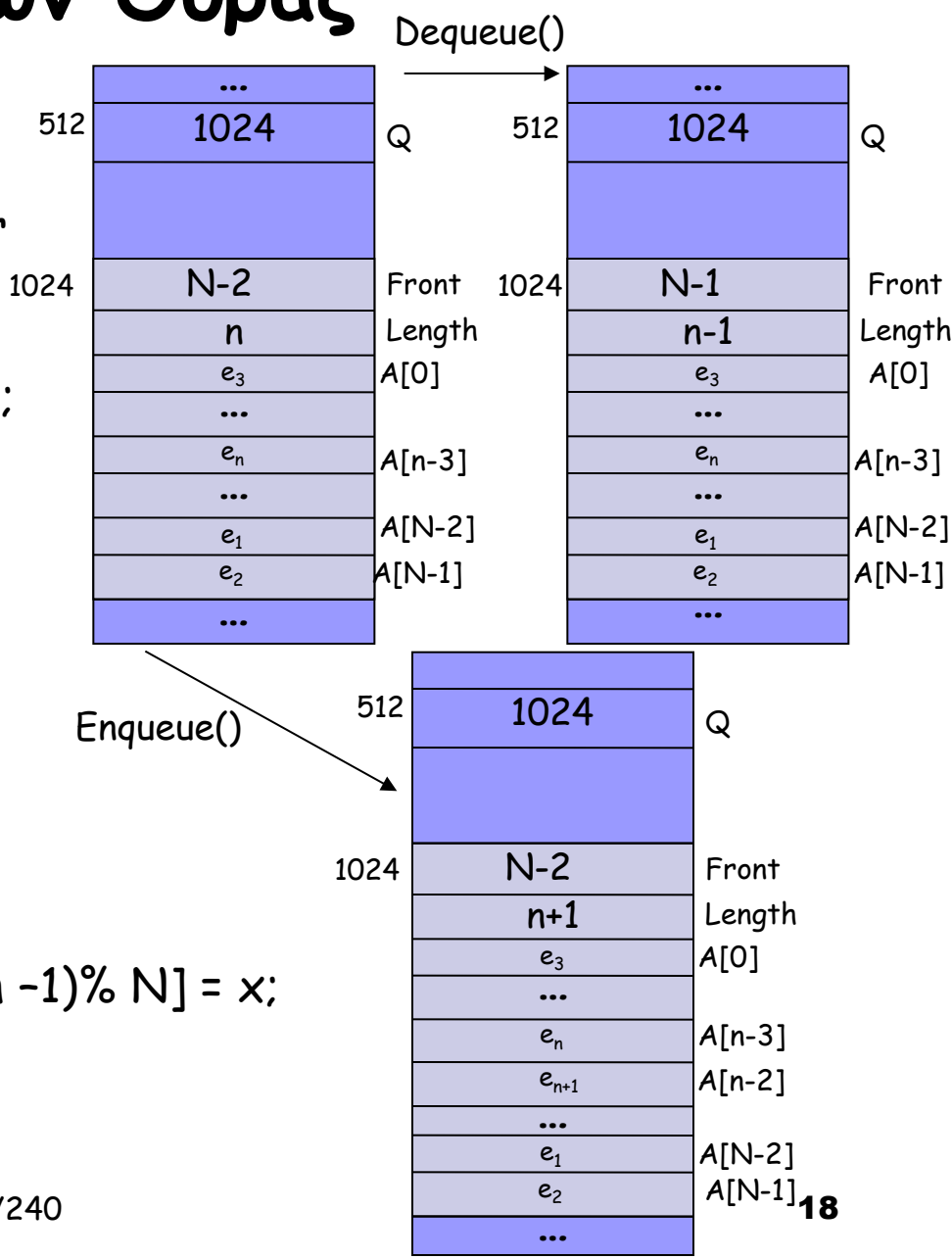
Υλοποίηση Λειτουργιών Ουράς

```
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;
    }
```

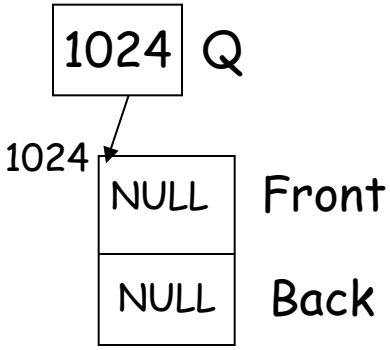
Χρονική Πολυπλοκότητα; $\Theta(1)$



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

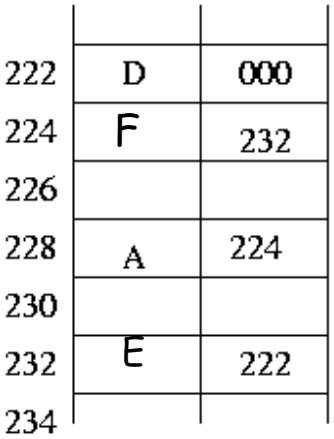


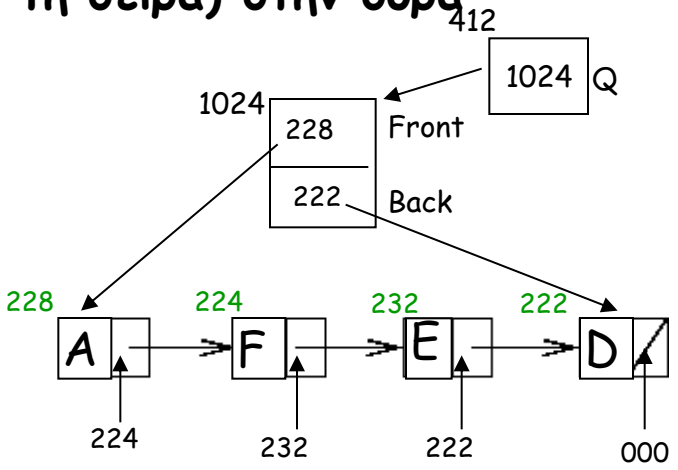
pointer *MakeEmptyQueue(void)*
```
pointer Q; /* temporary pointer */
Q = NewCell(Queue); /* malloc */
Q->Front = Q->Back = NULL;
return Q;
```

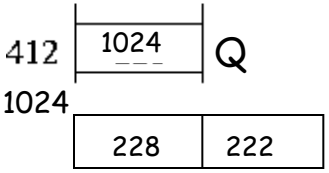
boolean *IsEmptyQueue(pointer Q)*
```
if (Q->Front == NULL) then
    return TRUE;
else return FALSE;
```

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

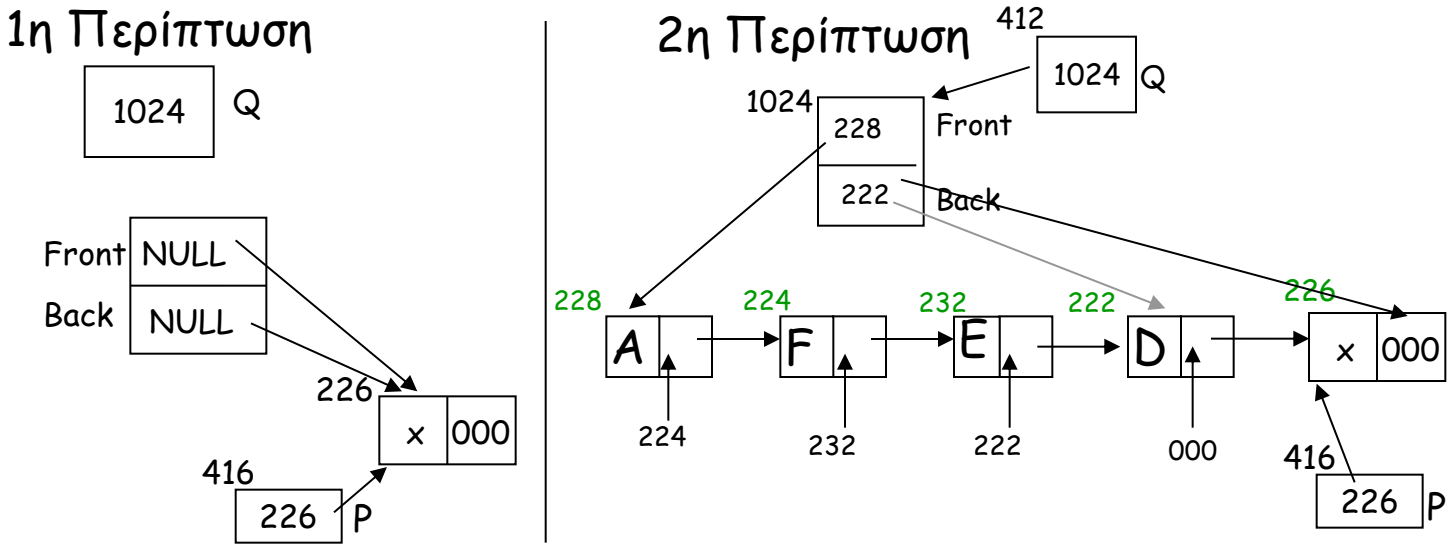






Χρονική Πολυπλοκότητα κάθε μιας από αυτές τις λειτουργίες;
$\Theta(1)$

Εισαγωγή σε Ουρά

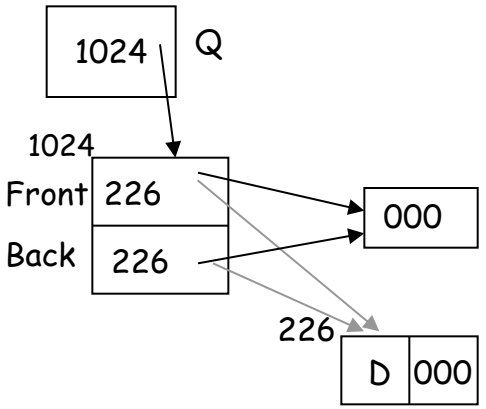



```
void Enqueue(Type x, pointer Q)
    pointer P;                    /* temporary pointer */
    P = NewCell(Node);
    P->data = x;
    P->next = NULL;
    if (IsEmptyQueue(Q)) then Q->Front = P;
    else Q->Back->next = P;
    Q->Back = P;
```

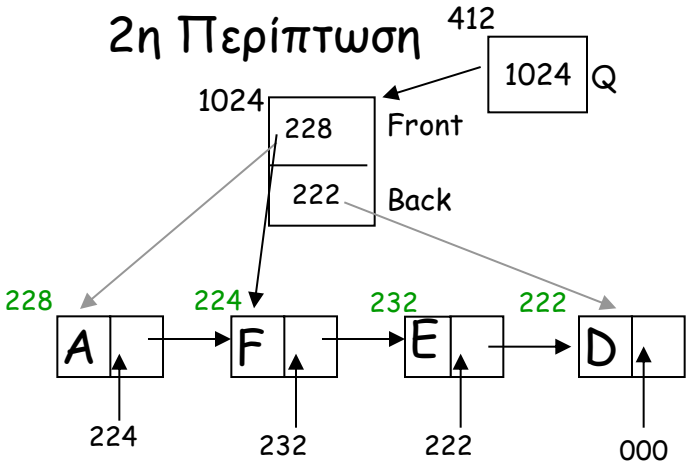
Χρονική Πολυπλοκότητα; $\Theta(1)$

Διαγραφή από Ουρά

1η Περίπτωση



2η Περίπτωση



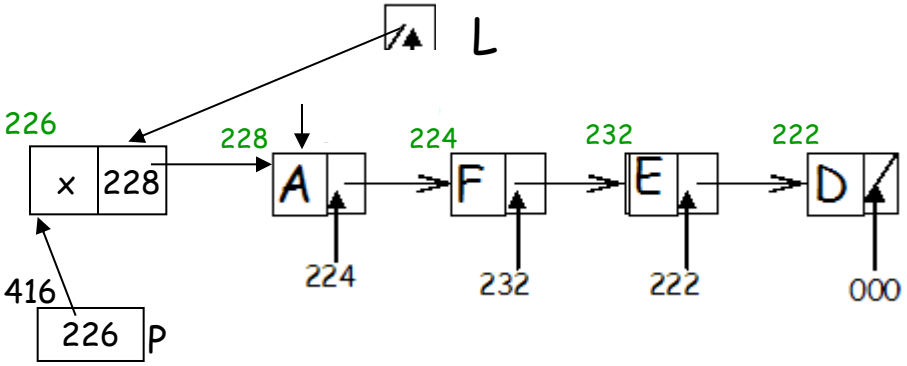

```
Type Dequeue(pointer Q)
   if (IsEmptyQueue(Q)) then error;
   else {
      x = (Q->Front)->data;
      Q->Front = (Q->Front)->next;
      if  (Q->Front == NULL) then
            Q->Back = NULL;
      return x;
   }
```

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

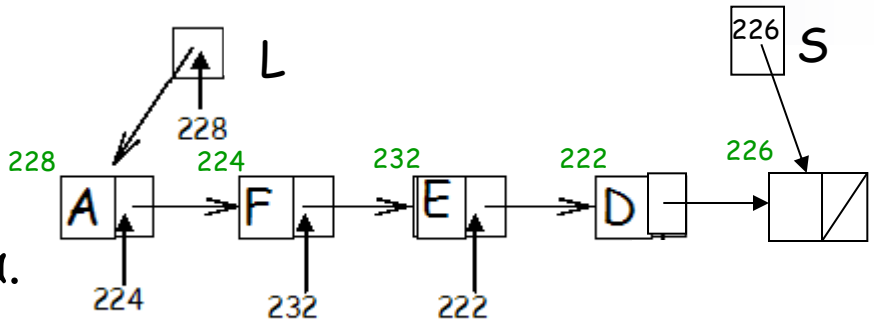
```
Pointer ListInsert(Pointer L, int x)
        pointer p;
        p = newcell(node);
        p->data = x;
        p->next = L;
        L = p;
        return L;
}
```

Αναζήτηση σε Λίστα

```
boolean ListSearch(Pointer L, Type x) {
        pointer q = L;
        while (q != NULL && q->data != x)
                q = q->next;
        if (q != NULL) return TRUE;
        else return FALSE;
}
```

Άσκηση: Υλοποιείστε τη Delete().

Κόμβος Φρουρός



Προς επίλυση Πρόβλημα

Αναζήτηση ενός στοιχείου x στη λίστα.

Λύση με κόμβο φρουρό

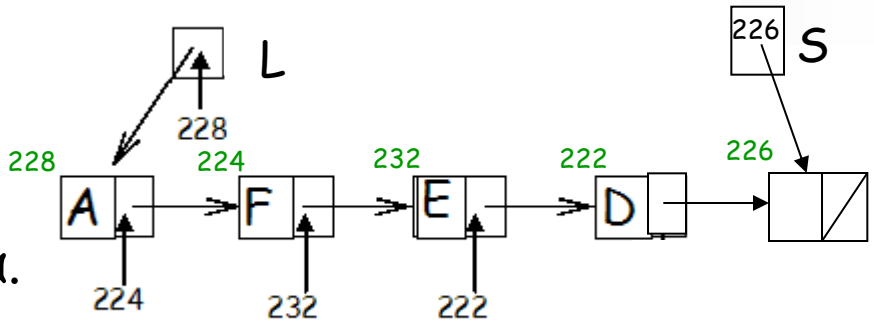
- Έχουμε εξ αρχής τοποθετήσει ένα κόμβο στη λίστα που λέγεται κόμβος φρουρός. Ο κόμβος αυτός είναι πάντα ο τελευταίος στη λίστα και χρησιμοποιείται μόνο για τη διαχείριση της λίστας (δηλαδή δεν θεωρείται στοιχείο της λίστας).
- Ένας δείκτης δείχνει μόνιμα σε αυτόν τον κόμβο.
- Κατά την αναζήτηση, η τιμή που αναζητείται αρχικά αποθηκεύεται στον κόμβο αυτό (π.χ. στο πεδίο data του struct του).
- Στη συνέχεια, εκτελείται διάσχιση της λίστας με τον γνωστό αλγόριθμο αναζήτησης για το στοιχείο αυτό.
- Το στοιχείο θα βρεθεί σίγουρα, είτε νωρίτερα σε κάποια θέση άλλη από τον κόμβο φρουρό ή στον κόμβο φρουρό.
- Στην 1η περίπτωση,

  η αναζήτηση είναι επιτυχημένη.
- Στην 2η περίπτωση, όχι.

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

Τι κερδίζουμε με τη χρήση κόμβου φρουρού;

Κόμβος Φρουρός



Προς επίλυση Πρόβλημα

Αναζήτηση ενός στοιχείου x στη λίστα.

Λύση με κόμβο φρουρό

- Έχουμε εξ αρχής τοποθετήσει ένα κόμβο στη λίστα που λέγεται κόμβος φρουρός. Ο κόμβος αυτός είναι πάντα ο τελευταίος στη λίστα και χρησιμοποιείται μόνο για τη διαχείριση της λίστας (δηλαδή δεν θεωρείται στοιχείο της λίστας).
- Ένας δείκτης δείχνει μόνιμα σε αυτόν τον κόμβο.
- Κατά την αναζήτηση, η τιμή που αναζητείται αρχικά αποθηκεύεται στον κόμβο αυτό (π.χ. στο πεδίο data του struct του).
- Στη συνέχεια, εκτελείται διάσχιση της λίστας με τον γνωστό αλγόριθμο αναζήτησης για το στοιχείο αυτό.
- Το στοιχείο θα βρεθεί σίγουρα, είτε νωρίτερα σε κάποια θέση άλλη από τον κόμβο φρουρό ή στον κόμβο φρουρό.
- Στην 1η περίπτωση,

  η αναζήτηση είναι επιτυχημένη.
- Στην 2η περίπτωση, όχι.

```
//Με κόμβο φρουρό!
boolean ListSearch(Type x) {
    S->data = x;        // S = sentinel
    pointer q = L;
    while (q->data != x)
        q = q->next;
    return (q != S);
}
```

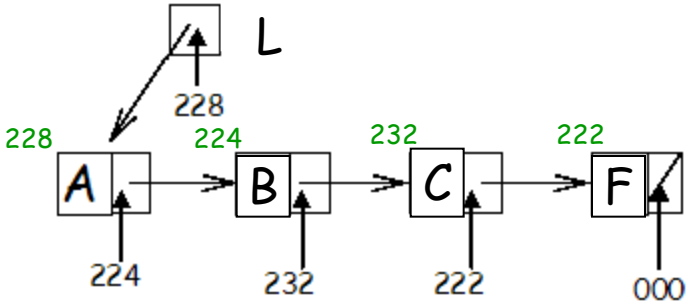
Τι κερδίζουμε με τη χρήση κόμβου φρουρού;

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);
```

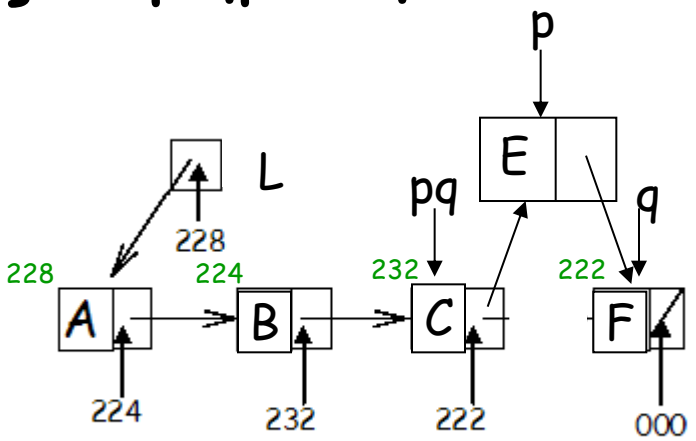
Εισαγωγή Στοιχείου σε Ταξινομημένη Λίστα

```
Pointer LLInsert(Type x, pointer L)
    pointer pq, q,p;    /* temporary pointers */

    q = L;
    pq = NULL;
    while (q != NULL) and (q->data < x) {
          pq = q;
          q = q->next;
    }

   if (q != NULL) and (q->data == x) then return;
                                  /* x is already in list */

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



Διάσχιση Λίστας

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

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

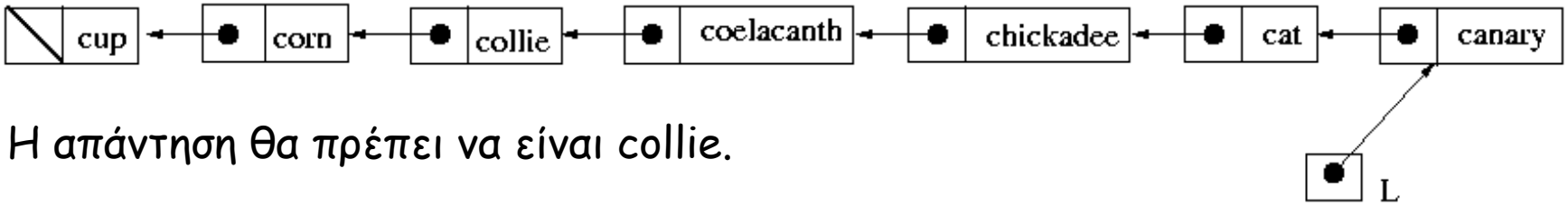
Πρόβλημα 1

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 = crabapple

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

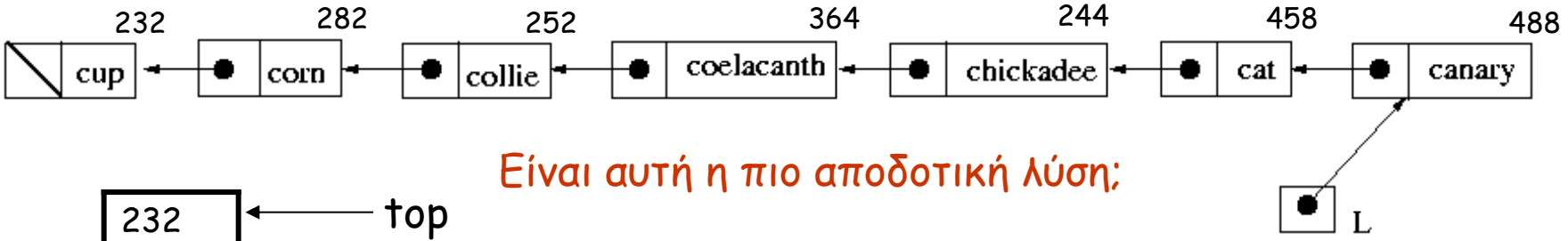


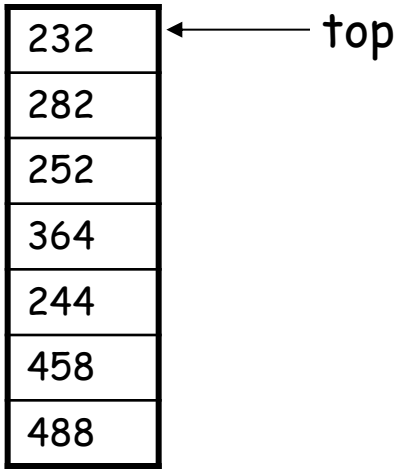
Η απάντηση θα πρέπει να είναι collie.

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?

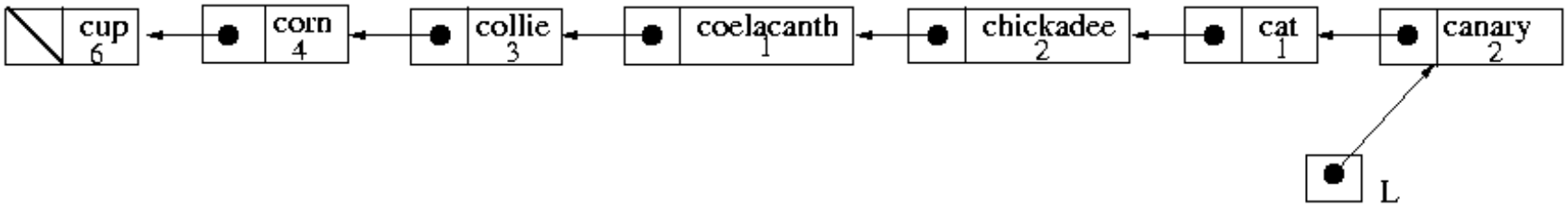
Διασχίσεις Zig-Zag

Έστω ότι κάθε κόμβος της λίστας έχει τα εξής πεδία:

- word: αλφαριθμητικό
- num: ακέραιος
- next: δείκτης στον επόμενο κόμβο

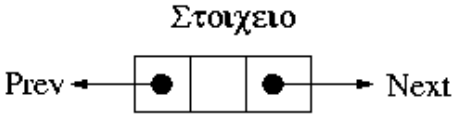
Πρόβλημα 2

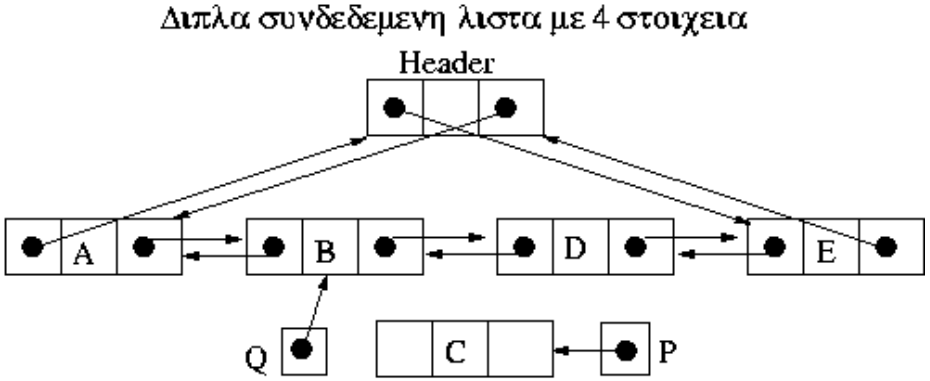
Δίδεται ένα αλφαριθμητικό w. Έστω ότι το w υπάρχει στη λίστα σε κάποιον κόμβο p του οποίου το πεδίο num έχει τιμή n. Αναζητείται η τιμή του πεδίου word του κόμβου που προηγείται του p κατά n θέσεις στη λίστα.



Παρουσιάστε αλγόριθμο που να επιλύει το πρόβλημα.

Διπλά Συνδεδεμένες Λίστες





Σχήμα 3.5: Lewis & Denenberg, Data Structures & Their Algorithms, Addison-Wesley, 1991

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

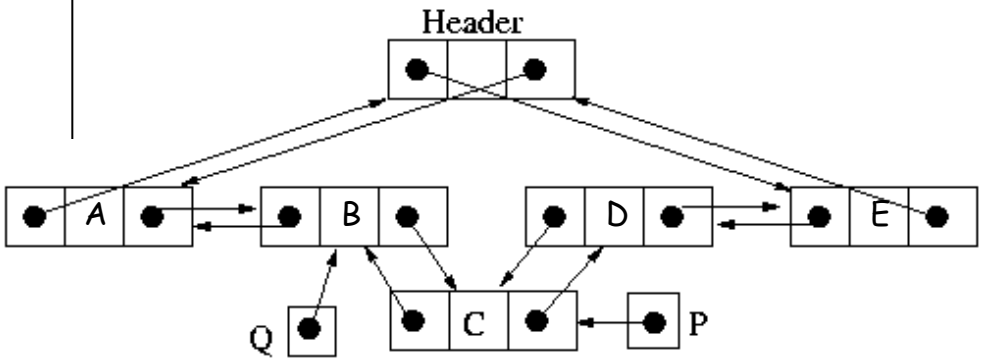
void DoublyLinkedInsert(pointer P,Q)
/* insert node pointed to by P just after node pointed to by Q */

$$\begin{pmatrix} P->Prev \\ P->Next \\ Q->Next \\ Q->Next->Prev \end{pmatrix} \leftarrow \begin{pmatrix} Q \\ Q->Next \\ P \\ P \end{pmatrix}$$

Διαγραφή κόμβου P από τη λίστα

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

$$\begin{pmatrix} P->Prev->Next \\ P->Next->Prev \end{pmatrix} \leftarrow \begin{pmatrix} P->Next \\ P->Prev \end{pmatrix}$$



Σχήμα 3.5: Lewis & Denenberg, Data Structures & Their Algorithms, Addison-Wesley, 1991

Τεχνικές Επιμεριστικής Ανάλυσης

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

Μέθοδοι Επιμεριστικής Ανάλυσης

- Η αθροιστική μέθοδος
- Η λογιστική μέθοδος
- Η μέθοδος του δυναμικού (δεν θα διδαχθεί σε αυτό το μάθημα)

Μέθοδοι Επιμεριστικής Ανάλυσης

- **Αθροιστική Μέθοδος**
  - Καθορισμός ενός πάνω φράγματος T(n) στο συνολικό κόστος μιας ακολουθίας n λειτουργιών.
  - Το επιμεριστικό κόστος κάθε λειτουργίας είναι T(n)/n.

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

Επιμεριστική Ανάλυση – Αθροιστική Μέθοδος

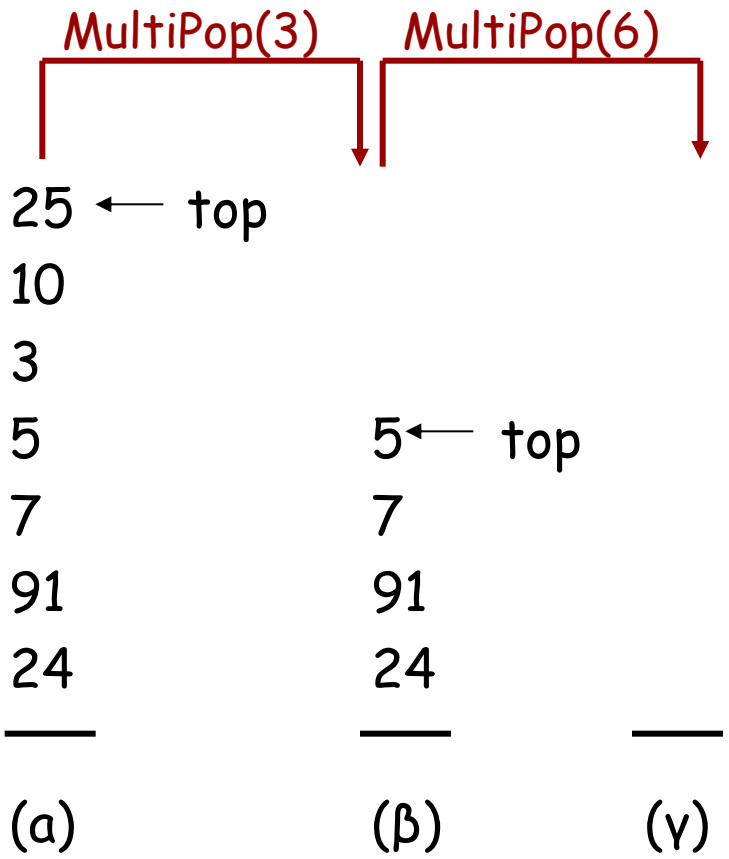
- Αποδεικνύουμε ότι ∀ n, οποιαδήποτε ακολουθία n λειτουργιών απαιτεί συνολικά το πολύ T(n) βήματα.

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 (!IsEmptyStack() AND k ≠0) {
        Pop();
        k = k-1;
    }
}
```



Επιμεριστική 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 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 MultiPop() είναι το πολύ όσο το πλήθος των Push().
- Το πλήθος των λειτουργιών Push() που θα εκτελεστούν είναι $O(n)$.

➢ **Η επιμεριστική χρονική πολυπλοκότητα κάθε λειτουργίας είναι $O(n)/n = O(1)$.**

Επιμεριστική Ανάλυση - Λογιστική Μέθοδος

- Καθορισμός του επιμεριστικού κόστους κάθε λειτουργίας. Διαφορετικές λειτουργίες μπορεί να έχουν διαφορετικά επιμεριστικά κόστη.

- Το επιμεριστικό κόστος των λειτουργιών μπορεί να είναι μεγαλύτερο ή μικρότερο από το πραγματικό τους κόστος. Το «κέρδος» από λειτουργίες με μεγαλύτερο από το πραγματικό επιμεριστικό κόστος αποθηκεύεται σε συγκεκριμένα αντικείμενα της δομής ως πίστωση και χρησιμοποιείται αργότερα για την «πληρωμή» λειτουργιών με επιμεριστικό κόστος μικρότερο από το πραγματικό τους.

- Το συνολικό επιμεριστικό 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 επιμεριστικό κόστος σε κάθε λειτουργία.

Επιμεριστική Ανάλυση - Λογιστική Μέθοδος

Παράδειγμα 1 – Στοίβα που υποστηρίζει τη λειτουργία MultiPop()

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

Το επιμεριστικό κόστος κάθε λειτουργίας είναι O(1).

Θα αποδείξουμε ότι για οποιαδήποτε ακολουθία n λειτουργιών, το συνολικό επιμεριστικό 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()).

Αναφορές

Το υλικό της ενότητας αυτής περιέχεται στα ακόλουθα βιβλία:

- Harry Lewis and Larry Denenberg, *Data Structures and Their Algorithms*, Harper Collins Publishers, Inc., New York, 1991
  - Chapter 3: Lists

- Cormen, Leiserson, Rivest & Stein, Εισαγωγή στους Αλγορίθμους, Πανεπιστημιακές Εκδόσεις Κρήτης, 2006.
  - Κεφάλαιο 17: Αντισταθμιστική Ανάλυση