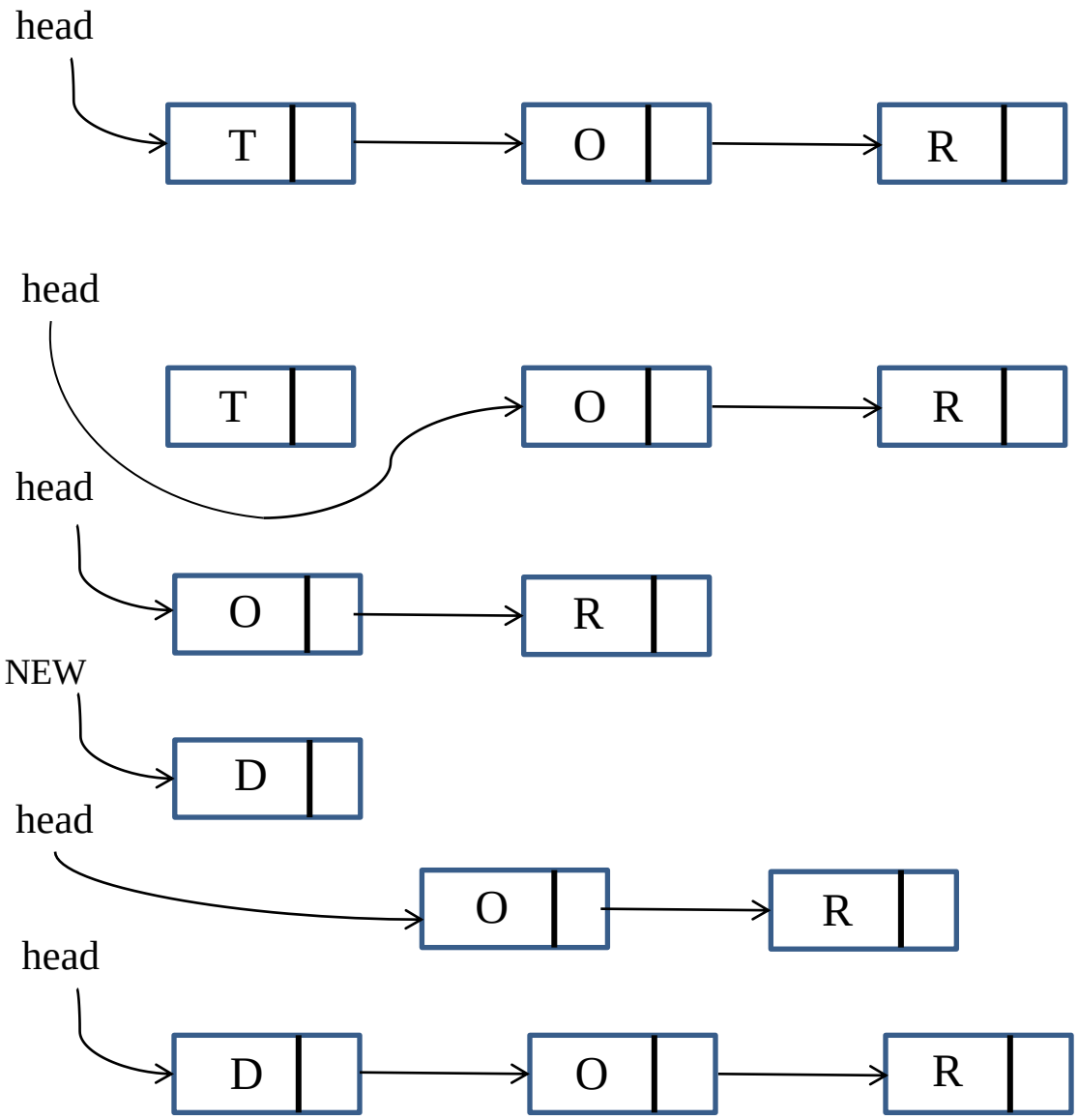
Stiva - continuare

Implementare prin listă înlănţuită



Ştergerea/Inserarea unui nod într-o stivă

Acest cod implementează **TAD** STIVĂ prin listă înlănţuită prezentat în slide-ul precedent.

Este utilizată funcţia auxiliară NEW pentru a aloca spaţiu de memorie pentru un nod, să seteze valorile din argumentele funcţiei şi să returneze un pointer spre nod.

```
#include <stdlib.h>
#include "Item.h"
typedef struct STACKnode* link;
struct STACKnode { Item itemj link next; };
static link head;

link NEW (Item item, link next)
{ link x = malloc(sizeof *x);
x->item = item;
x->next = next;
return x;
}
```

```
void STACKinit(int maxN)
{ head = NULL; }

int STACKempty()
{ return head == NULL; }

STACKpush(Item item)
{head = NEW(item, head); }

Item STACKpop ()
{ Item item = head->item;
link t = head->next;
 free(head); head = t;
return item;
}
```

Coadă la un magazin de carne din Bucuresti inainte de 1990

şi

iPad2 Coadă la un magazin Apple din New York 2011

Coada

Cozi (engl. Queues)

O coadă este o listă specială cu următoarea caracteristică:

- Principiul (FIFO) First In – First Out:
  - inserarea se face numai la sfârşitul listei
  - ştergerea se face numai de la începutul listei

Operaţii pe structuri de tipul coadă

- Put: un element este inserat la sfârşitul cozii
- Get: primul element al cozii este îndepărtat

Exemple de utilizare a cozilor:

- Playlist la iTunes
- Bufferul (zona tampon) de date la iPod
- Comenzile de tipărire în coada de aşteptare a imprimantei
- Comenzile clienţilor la portalele Web

Cozi FIFO şi cozi generalizate

Coada first-in, first-out (FIFO) este un alt **TAD** fundamental care este similar stivei, dar care foloseşte regula opusă la decizia care nod să fie îndepărtat de către *delete*. În loc de a îndepărta nodul cel mai recent inserat, se va îndepărta cel care se află de cel mai mult timp în coadă .

Cozile FIFO sunt numeroase în viaţa de zi cu zi: când aşteptăm în rând să intrăm la un concert/meci sau să cumpărăm la un supermarket vom fi serviţi conform unei discipline FIFO.

Definitie:*(Sedgewick)* O coadă FIFO *este un TAD care cuprinde două operaţii de bază: insert* (put) *adaugă un nou articol/ item şi delete* (get) *şterge articolul/item-ul care a fost adăugat de cel mai mult timp.*

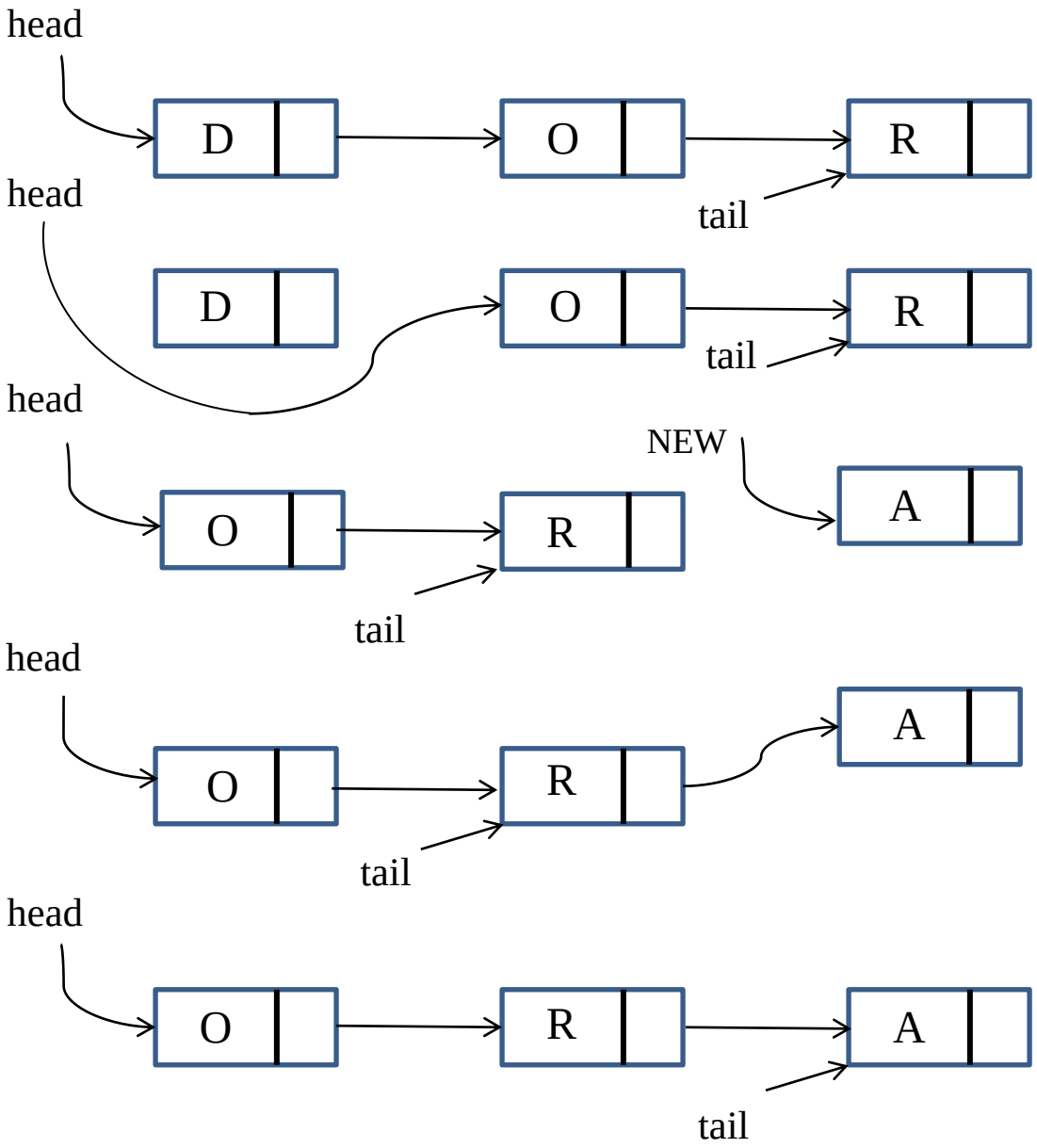
Interfaţă TAD coadă FIFO

Identică cu interfaţa de la stivă exceptând numele structurii. Cele două TAD-uri diferă numai în specificaţie, ceea ce nu se reflectă în cod.

```
void QUEUEinit(int);
 int QUEUEempty();
void QUEUEput(Item);
Item QUEUEget () ;
```

```
void STACKinit(int);
 int STACKempty();
void STACKpush(Item);
Item STACKpop();
```

OBS: pentru un compilator, să zicem, cele două interfeţe sunt identice! Această observaţie subliniază faptul că abstractizarea în sine, pe care programatorii în mod obişnuit nu o definesc formal, este o componentă esenţială a unui TAD.



Ştergerea/Inserarea unui nod într-o coadă

Pentru a implementa TAD – ul *Coadă* FIFO printr-o listă înlănţuită, păstrăm înregistrările în listă în ordine – de la cel mai puţin recent inserat la cel mai recent inserat (vezi slide). Această ordine ne permite o implementare eficientă a operaţiilor cozii.

Se utilizează 2 pointeri pe listă:

- unul pe începutul listei (head) – astfel încât să *obţinem /get* primul element din coadă

- unul pe sfârşit (tail) – astfel încât să putem *insera/put* un nou element în coadă.

Coadă FIFO implementată printr-o **listă înlănţuită**

Acest program păstrează un pointer tail pe ultimul nod al listei, astfel încât funcţia `QUEUEput` poate adăuga un nod nou legând acel nod de nodul referit de `tail` şi apoi actualizând `tail` să pointeze spre noul nod.

Funcţiile `QUEUEget`, `QUEUEinit` şi `QUEUEempty` sunt toate identice cu omoloagele lor de la stiva implementată ca listă înlănţuită în programul 4.5

```
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"
typedef struct QUEUEnode* link;
struct QUEUEnode { Item item; link next; };
static link head, tail;

link NEW (Item item, link next)
{ link x = malloc(sizeof *x);
x->item = item;
x->next = next;
return x;
}
void QUEUEinit(int maxN)
{ head = NULL; }

int QUEUEempty 0
{ return head == NULL; }
```

```
QUEUEput(Item item)
{
if (head == NULL)
  { head = (tail = NEW(item, head));
    return;
  }
tail->next = NEW(item, tail->next);
tail = tail->next;
}

Item QUEUEget ()
{ Item item = head->item;
link t = head->next;
free(head); head = t;
return item;
}
```

Coada poate fi implementată şi printr-un array cu toate că trebuie să avem grijă să păstrăm timpul de execuţie constant atât pentru operaţia *put* cât şi pentru *get*. Acest scop ne impune să nu mutăm elementele cozii în array. La fel cum am procedat la coada implementată ca listă înlănţuită vom lucra cu doi indici pe array: unul pe începutul cozii şi unul pe sfârşitul ei.

Conţinutul cozii vor fi elementele între cei 2 indici. Pentru a obţine un element *get* îl vom îndepărta de la începutul cozii (head) şi vom incrementa indexul de head, iar pentru a insera un element în coadă *put* îl vom adăuga la sfârşitul cozii (tail) şi vom incrementa indexul de tail.

O secvenţă de operaţii *put* şi *get* dă impresia deplasării cozii de-a lungul array-ului. Când atinge capătul array-ului se va reveni la început (vezi implementarea).

Coadă FIFO implementată printr-un **array**

Conţinutul cozii vor fi elementele între cei 2 indici.
Dacă *head* şi *tail* sunt egale considerăm coada că este vidă; dar dacă *put* le face egale, atunci vom considera coada că este plină.
Nu verificăm asemenea conditii de eroare, dar făcând dimensiunea array-ului cu 1 mai mare decât numărul maxim de elemente la care se aşteaptă clientul, putem extinde programul să facă asemenea verificări.

```
#include <stdlib.h>
#include "Item.h"
static Item *q;
static int N, head, tail;

void QUEUEinit(int maxN)
{ q = malloc((maxN+1)*sizeof(Item));
N = maxN+1; head = N; tail = 0; }

int QUEUEempty()
{ return head % N == tail; }

void QUEUEput(Item item)
{ q[tail++] = item; tail = tail %N; }

Item QUEUEget()
{ head = head % N; return q[head++]; }
```

Proprietate: *Operaţiile* get *şi* put *pentru un TAD coadă pot fi implementate astfel încât să se execute în timp constant indiferent dacă se utilizează o listă înlănţuită sau un array.*

Proprietatea este evidentă analizând cele 2 implementări.

Într-unul din cursurile viitoare vom analiza un tip special de coadă:
coada cu priorităţi


Și chiar de nu voi fi un far, ci o candelă, ajunge. Și chiar de nu voi fi nici candelă, tot ajunge, fiindcă m-am străduit să aprind lumina.
Nicolae Titulescu


Sa ai un Paste luminos si plin de...
PACE
CREDINTA
IUBIRE
SPERANTA
ÎMPLINIRI ŞI BUCURII !