

Liste

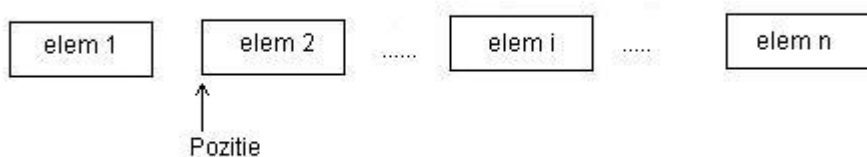
- In limbajul uzual cuvântul “listă” referă o “înșirare, într-o anumită *ordine*, a unor nume de persoane sau de obiecte, a unor date etc.”
- Exemple de liste sunt multiple: listă de cumpărături, listă de prețuri, listă de studenți, etc.
 - *Ordinea* în listă poate fi interpretată
 - ca un fel de „legătură” între elementele listei: după prima cumpărătură urmează a doua cumpărătură, după a doua cumpărătură urmează a treia cumpărătură, etc)
sau
 - poate fi văzută ca fiind dată de numărul de ordine al elementului în listă (1-a cumpărătură, a 2-a cumpărătură, etc).
 - Tipul de date `Listă` care va fi definit în continuare permite implementarea în aplicații a acestor situații din lumea reală.
- O **listă** o putem vedea ca pe o secvență de elemente $\langle l_1, l_2, \dots, l_n \rangle$ de un același tip (`TElement`), fiecare element având o *poziție* bine determinată în cadrul listei, existând o ordine între pozițiile elementelor în cadrul listei
 - Lista poate fi văzută ca o colecție dinamică de elemente în care este esențială ordinea elementelor.
 - Numărul n de elemente din listă se numește *lungimea* listei.
 - O listă de lungime 0 se va numi lista *vidă*.
 - Caracterul de dinamicitate al listei este dat de faptul că lista își poate modifica în timp lungimea prin adăugări și ștergeri de elemente în/din listă.
- *Poziția* elementelor în cadrul listei este esențială, astfel accesul, ștergerea și adăugarea se pot face pe orice *poziție* în listă.
- **Lista liniară**
 - o structură care fie este vidă (nu are nici un element), fie
 - are un unic prim element;
 - are un unic ultim element;
 - fiecare element din listă (cu excepția ultimului element) are un singur succesor;
 - fiecare element din listă (cu excepția primului element) are un singur predecesor.
 - Într-o listă liniară se pot insera elemente, șterge elemente, se poate determina succesorul (predecesorul) unui element aflat pe o anumită *poziție*, se poate accesa un element pe baza *poziției* sale în listă.

- Fiecare element al unei liste liniare are o *poziție* bine determinată în cadrul listei.
 - este importantă prima *poziție* în cadrul listei
 - *poziția* unui element este relativă la listă
 - *poziția* unui element din listă
 - identifică elementul din listă
 - poziția elementului predecesor și poziția elementului succesor în listă (dacă acestea există)
 - ordine între pozițiile elementelor în cadrul listei.

Poziția unui element în cadrul listei poate fi văzută în diferite moduri:

1. ca fiind dată de **rangul** (numărul de ordine al) elementului în cadrul listei.
 - similitudine cu tablourile, *poziția* unui element în listă fiind *indexul* acestuia în cadrul listei.
 - Într-o astfel de abordare, lista este văzută ca un tablou dinamic în care se pot accesa/adăuga/șterge elemente pe orice poziție în listă.
2. ca fiind dată de o **referință** la locația unde se stochează elementul listei (ex: pointer spre locația unde se memorează elementul).

Pentru a asigura generalitatea, vom abstractiza noțiunea de **poziție** a unui element în listă și vom presupune că elementele listei sunt accesate prin intermediul unei **poziții** generice.



- O **Poziție** p într-o listă o considerăm **validă** dacă este poziția unui element al listei.
 - dacă p ar fi un pointer spre locația unde se memorează un element al listei, atunci p este **valid** dacă este diferit de pointerul nul sau de orice altă adresă care nu reprezintă adresa de memorare a unui element al listei.
 - dacă p ar fi rangul (numărul de ordine al) elementului în listă, atunci p este **valid** dacă e cuprins între 1 și numărul de elemente din listă.

TAD Lista (LIST)

Observații:

1. Tipul (abstract) de date $TPozitie$ abstractizează noțiunea de poziție a unui element în listă (pentru a se asigura generalitatea).
2. O poziție $p \in TPozitie$ din lista l o numim *poziție validă* dacă este poziția unui element din lista l .
3. În domeniului de valori a $TPozitie$, notată cu \perp o valoare specială p car o vom numi poziție nedefinită. Poziția nedefinită \perp nu este o poziție *validă* (conform celor menționate anterior).
4. Lista vidă o notăm cu Φ .

Tipul Abstract de Date LISTA:

domeniu:

$\mathcal{L} = \{l \mid l \text{ este o listă cu elemente de tip } TElement, \text{ fiecare element având o poziție unică în } l \text{ de tip } TPozitie\}$

operații:

- $creeaza(l)$
 $\{creează o listă vidă\}$
 $pre : true$
 $post : l \in L, l = \Phi$
- $prim(l)$
 $pre : l \in L$
 $post : prim = p \in TPozitie,$
 $p = \begin{cases} \text{poziția primului element din lista } l, & \text{dacă } l \neq \Phi \\ \perp, & \text{dacă } l = \Phi \end{cases}$
- $ultim(l)$
 $pre : l \in L$
 $post : ultim = p \in TPozitie,$
 $p = \begin{cases} \text{poziția ultimului element din lista } l, & \text{dacă } l \neq \Phi \\ \perp, & \text{dacă } l = \Phi \end{cases}$
- $valid(l, p)$
 $pre : l \in L, p \in TPozitie$
 $post : valid = \begin{cases} true, & \text{dacă } p \text{ este o poziție a unui element din lista } l \\ false, & \text{altfel} \end{cases}$
- $următor(l, p)$
 $pre : l \in L, p \in TPozitie, p \text{ poziție validă}$
 $post : urmator = q \in TPozitie,$
 $q = \begin{cases} \text{poziția următoare poziției } p \text{ din lista } l, & \\ \text{dacă } p \text{ nu e poziția ultimului element din lista } l & \\ \perp, & \text{dacă } p \text{ e poziția ultimului element din lista } l \end{cases}$
@ aruncă excepție dacă p nu e validă

- $\text{anterior}(l, p)$

$pre : l \in L, p \in TPozitie, p \text{ poziție validă}$
 $post : anterior = q \in TPozitie,$

$$q = \begin{cases} \text{poziția precedentă poziției } p \text{ din lista } l, & \text{dacă } p \text{ nu e poziția primului element din lista } l \\ \perp, & \text{dacă } p \text{ e poziția primului element din lista } l \end{cases}$$

@ aruncă excepție dacă p nu e validă
- $\text{element}(l, p, e)$

$pre : l \in L, p \in TPozitie, \text{valid}(p)$
 $post : e \in TElement, e = \text{elementul de pe poziția } p \text{ din } l$

@ aruncă excepție dacă p nu e validă
- $\text{pozitie}(l, e)$

$pre : l \in L, e \in TElement,$
 $post : pozitie = p \in TPozitie,$

$$p = \begin{cases} \text{prima poziție a elementului } e \text{ din lista } l, & \text{dacă } e \in l \\ \perp, & \text{dacă } e \notin l \end{cases}$$
- $\text{modifică}(l, p, e)$

$pre : l \in L, p \in TPozitie, \text{valid}(p), e \in TElement$
 $post : \text{elementul de pe poziția } p \text{ din } l' = e$

@ aruncă excepție dacă p nu e validă
- $\text{adaugaInceput}(l, e)$

$pre : l \in L, e \in TElement$
 $post : \text{elementul } e \text{ a fost adăugat la începutul listei } l$
 $(l' = e \oplus l)$
- $\text{adaugaSfarsit}(l, e)$

$pre : l \in L, e \in TElement$
 $post : \text{elementul } e \text{ a fost adăugat la sfârșitul listei } l$
 $(l' = l \oplus e)$
- $\text{adaugaDupa}(l, p, e)$

$pre : l \in L, p \in TPozitie, \text{valid}(p), e \in TElement$
 $post : \text{elementul } e \text{ a fost inserat în lista } l \text{ după poziția } p,$
 $\text{pozitie}(l', e) = \text{urmator}(l', p)$

@ aruncă excepție dacă p nu e validă
- $\text{adaugaInainte}(l, p, e)$

$pre : l \in L, p \in TPozitie, \text{valid}(p), e \in TElement$
 $post : \text{elementul } e \text{ a fost inserat în lista } l \text{ înaintea poziției } p,$
 $\text{pozitie}(l', e) = \text{anterior}(l', p)$

@ aruncă excepție dacă p nu e validă
- $\text{sterge}(l, p, e)$

$pre : l \in L, p \in TPozitie, \text{valid}(p)$
 $post : e \in TElement, \text{elementul } e \text{ de pe poziția } p \text{ a fost șters din } l$

@ aruncă excepție dacă p nu e validă
- $\text{cauta}(l, e)$

$pre : l \in L, e \in TElement$
 $post : \text{cauta} = \begin{cases} \text{adevarat}, & \text{dacă } e \text{ a fost găsit în lista } l \\ \text{fals}, & \text{altfel} \end{cases}$

- $vida(l)$

$$pre : l \in L$$

$$post : vida = \begin{cases} true, & \text{dacă } l = \Phi \\ false, & \text{dacă } l \neq \Phi \end{cases}$$
- $dim(l)$

$$pre : l \in L$$

$$post : dim = n \in Natural,$$

$$n = \text{numărul de elemente ale listei } l$$
- $distruge(l)$

$$\{\text{destructor}\}$$

$$pre : l \in L$$

$$post : l \text{ a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)}$$
- $iterator(l, i)$

$$pre : l \in L$$

$$post : i \in \mathcal{I}, i \text{ este un iterator pe lista } l$$

Observații

- Operația **cauta** poate fi specificată mai general
 - returnează prima *poziție* pe care apare un element în listă, dacă elementul e găsit în listă
 - returnează *poziție invalidă* dacă elementul nu e găsit în listă
- Din perspectiva unei ierarhii de containere
 - **Lista** este o **Colecție**
 - **Vector Dinamic** este o **Listă**
 - * **Vectorul Dinamic** poate fi văzut ca o **Listă** reprezentată *secvențial*
- Există anumite dezavantaje induse de folosirea unui parametru de tip $TPozitie$ în interfața listei:
 1. Tipurile de referințe concrete folosite diferă în funcție de reprezentarea listei.
 2. Interfața listei este destul de greoaie și nesigură prin faptul că expune în exterior pozițiile (referințele la locațiile din listă).
 - acesta este motivul pentru care bibliotecile existente particularizează tipul $TPozitie$ expus în interfața containerului Listă (după cum se va vedea în continuare)

Implementări ale containerului **Listă** în biblioteci existente:

1. STL - list

- *poziția* este dată de un *iterator* pe listă $\Rightarrow TPozitie = Iterator$.
- în STL, *list* e văzut ca și un container de tip *secvență*: elementele sunt aranjate într-o ordine (liniară) strictă.
- reprezentarea este dublu *înlănțuită*
 - dacă se dorește reprezentare simplu *înlănțuită*, se va folosi **forward_list**.

- dacă se dorește reprezentare *secvențială*, se va folosi **vector**.

2. Java - List

- *poziția* este văzută ca un indice $\Rightarrow TPozitie = Intreg$.
 - permite accesarea elementelor din listă prin intermediul indicilor (ca la reprezentarea secvențială - **Vector Dinamic**)
- dacă se dorește reprezentare *înlănțuită* a listei, se va folosi **Linked List**.

Modalități de implementare a unei liste

- memorând elementele sale **secvențial** într-un tablou/vector (dinamic)
 - accesul la elementele listei este *direct* ($\theta(1)$)
- memorând elementele sale **înlănțuit** într-o listă înlănțuită
 - accesul la elementele listei este *secvențial* ($O(n)$)
 - lista înlănțuită poate fi
 - * simplu înlănțuită (LSI)
 - * dublu înlănțuită (LDI)

Analiza complexității timp a celor mai importante operații ale containerului Lista în funcție de implementarea acestuia

În Tabelul 1 vom considera, comparativ

- reprezentare secvențială folosind un vector dinamic (poziția este indice);
- reprezentare simplu înlănțuită (LSI) cu alocare dinamică (poziția este adresa de memorare a unui nod);
- reprezentare dublu înlănțuită (LDI) cu alocare dinamică (poziția este adresa de memorare a unui nod);

Notăm cu n numărul de elemente din listă. Observăm faptul că reprezentarea dublu înlănțuită este cea mai eficientă ca și timp, dar ocupă spațiu de memorare suplimentar pentru legături (pentru a reduce spațiul de memorare se pot folosi liste de tip XOR - a se vedea cursul 4).

Vom particulariza, în cele ce urmează, TAD-ul generic **Lista**, atfel încât să regăsim cele două specificații ale containerului **Lista** descrise anterior (STL/Java).

Lista - cu poziție indice (indexată)

- corespunde modului în care este specificată lista în Java.
- *poziția* este văzută ca un indice $\Rightarrow TPozitie = Intreg$.
 - permite accesarea elementelor prin intermediul indicilor
- Accesul la elemente se face pe baza rangului, se permit inserări și ștergeri la orice poziție (poziția unui element reprezintă indicele acestuia în cadrul listei).

Operație	Reprezentare secvențială	Reprezentare folosind o LSI alocată dinamic	Reprezentare folosind o LDI alocată dinamic
creeaza	$\theta(1)$	$\theta(1)$	$\theta(1)$
prim	$\theta(1)$	$\theta(1)$	$\theta(1)$
ultim	$\theta(1)$	$\theta(1)$ - dacă memorăm ultim $O(n)$ - fără a memora ultim	$\theta(1)$
următor	$\theta(1)$	$\theta(1)$	$\theta(1)$
anterior	$\theta(1)$	$O(n)$	$\theta(1)$
adaugaInceput	$\theta(n)$	$\theta(1)$	$\theta(1)$
adaugaSfarsit	$\theta(1)$ amortizat	$\theta(1)$ - dacă memorăm ultim $\theta(n)$ - fără a memora ultim	$\theta(1)$
adaugaDupa	$O(n)$	$\theta(1)$	$\theta(1)$
adaugaInainte	$O(n)$	$O(n)$	$\theta(1)$
sterge	$O(n)$	$O(n)$	$\theta(1)$

Tabela 1: Complexități timp ale operațiilor.

- O poziție i în cadrul listei l este *validă* dacă $1 \leq i \leq \text{lungime}(l)$.
- Se simplifică interfața
 - interfața este aceeași cu a unui **Vector Dinamic**

Specificația Listei indexate este dată mai jos

domeniu:

$$L = \{l \mid l = [e_1, e_2, \dots, e_n], e_i \in TElement \ \forall i = 1, 2, \dots, n\}$$

operații:

- creeaza (l)
 - $pre : \text{true}$
 - $post : l \in L, l = \Phi$ lista vidă
- adaugaSfarsit(l, e)
 - $pre : l \in L, e \in TElement$
 - $post : \text{elementul } e \text{ a fost adăugat la sfârșitul listei } l$
($l' = l \oplus e$)
- adauga (l, i, e)
 - $pre : l \in L, e \in TElement, i \in Integ,$
 i poziție validă în $l \vee i = \text{lungime}(l) + 1$
 - $post : l' = (e_1, \dots, e_{i-1}, e, e_i, e_{i+1}, \dots, e_n)$
($\text{pozitie}(l', e) = i$)

@ aruncă excepție dacă i nu e valid
- sterge (l, i, e)
 - $pre : l \in L, l = (e_1, \dots, e_{i-1}, e_i, e_{i+1}, \dots, e_n), i \in Integ, i$ poziție validă
 - $post : e \in TElement, e = \text{elementul de pe poziția } i \text{ din } l$
 $l' = (e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n)$
($\text{pozitie}(l', e) = i$)

@ aruncă excepție dacă i nu e valid

- $\text{cauta}(l, e)$

$$\text{pre} : l \in L, e \in TElement$$

$$\text{post} : \text{cauta} = \begin{cases} i, & \text{dacă } i \text{ e prima poziție pe care } e \text{ a fost găsit în lista } l \\ -1, & e \notin L \end{cases}$$
- $\text{element}(l, i, e)$

$$\text{pre} : l \in L, i \in Intreg, i \text{ poziție validă}$$

$$\text{post} : e \in TElement, e = \text{elementul de pe poziția } i \text{ din } l$$

@ aruncă excepție dacă i nu e valid
- $\text{modifica}(l, i, e)$

$$\text{pre} : l \in L, i \in Intreg, i \text{ poziție validă}, e \in TElement$$

$$\text{post} : \text{elementul de pe poziția } i \text{ din } l' = e$$

@ aruncă excepție dacă i nu e valid
- $\text{vida}(l)$

$$\text{pre} : l \in L$$

$$\text{post} : \text{vida} = \begin{cases} true, & \text{dacă } l = \Phi \\ false, & \text{altfel} \end{cases}$$
- $\text{dim}(l)$

$$\text{pre} : l \in L$$

$$\text{post} : \text{dim} = n \in Intreg,$$

$$n = \text{numărul de elemente din lista } l$$
- $\text{iterator}(l, i)$

$$\text{pre} : l \in L$$

$$\text{post} : i \in \mathcal{I}, i \text{ este un iterator pe lista } l$$
- $\text{distruge}(l)$

$$\text{pre} : l \in L$$

$$\text{post} : l \text{ a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)}$$

Exemplu

Considerăm reprezentarea Listei indexate folosind o LSI alocată dinamic. Descriem mai jos, în Pseudocod, operația **element**.

Reprezentarea listei este

Nod

e: TElement //informația utilă nodului

urm: \uparrow Nod //adresa la care e memorat următorul nod

Lista

prim: \uparrow Nod//adresa primului nod din listă

Subalgoritm $\text{element}(l, i, e)$

{pre: l : Lista, i :Intreg, $1 \leq i \leq \text{lungime}(l)$, e :TElement }

{post: e este al i -lea element al listei }

{se parcurge până la al i -lea element }

$p \leftarrow l.\text{prim}$

{se parcurg $i-1$ legături }

Pentru $i = 1, i - 1$ executa

$p \leftarrow [p].\text{urm}$


```

SfPentru
{p este al i-lea nod }
e ← [p].e
SfSubalgoritm

```

- Complexitate: $O(n)$, n fiind numărul de elemente din listă

Să considerăm subalgoritmul **tiparire** care tipărește elementele unei liste indexate reprezentate folosind o LSI alocată dinamic. Tipărirea trebuie realizată folosind iteratorul, în caz contrar, tipărirea se va realiza în timp pătratic în raport cu numărul de elemente din listă.

1. folosind un iterator: complexitate timp $\theta(n)$, n fiind numărul de elemente ale listei

```

Subalgoritm tiparire(l)
{pre: l: Lista}
{post: se tipăresc elementele listei}
iterator(l,i)
CatTimp valid(i) executa
    element(i,e)
    @tipărește e
    urmator(i)
SfCatTimp
SfSubalgoritm

```

2. folosind accesul la elemente prin indici: complexitate timp $\theta(n^2)$, n fiind numărul de elemente ale listei

```

Subalgoritm tiparire(l)
{pre: l: Lista}
{post: se tipăresc elementele listei}
Pentru i = 1, dim(l) executa
    element(l,i,e)
    @tipărește e
SfPentru
SfSubalgoritm

```

Lista - cu poziție iterator

- corespunde modului în care este specificată lista în STL.
- *poziția* este dată de un *iterator* pe listă $\Rightarrow TPozitie = Iterator$.
- se simplifică interfața
 - operațiile *următor*, *anterior*, *valid* și *element* sunt operațiile pe *iterator*

Enumerăm, mai jos, operațiile din interfața Listei în care accesul e pe baza unei poziții date de un iterator, fără a mai da specificația completă a operațiilor (specificațiile sunt cele indicate la containerul generic **Lista**, dar cu $TPozitie = IteratorLista$).

Operații din interfață:

- creeaza ($l : Lista$)
- vida ($l : Lista$)
- dim ($l : Lista$)
- IteratorLista prim($l : Lista$)

- *TElement* element(*l* :Lista, *poz*:IteratorLista)
- *TElement* modifica(*l* :Listă, *poz*:IteratorLista, *e* : *TElement*)
- adaugaInceput(*l* :Listă, *e* : *TElement*)
- adaugaSfarsit(*l* :Listă, *e* : *TElement*)
- adauga (*l* :Listă, *poz*:IteratorListă, *e* : *TElement*)
- *TElement* sterge(*l* :Lista, *poz*:IteratorLista)
- IteratorLista cauta(*l* :Lista, *e* : *TElement*)
- distruge (*l* : Lista)

Exemplu

Considerăm reprezentarea Listei cu poziție iterator, folosind o LDI alocată dinamic. Descriem mai jos, în Pseudocod, operația **adaugaDupa**.

Reprezentarea listei și a iteratorului pe listă sunt date mai jos

Nod

e: TElement //informația utilă nodului
urm: ↑ Nod //adresa la care e memorat următorul nod
prec: ↑ Nod //adresa la care e memorat nodul anterior

Lista

prim: ↑ Nod//adresa primului nod din listă
ultim: ↑ Nod//adresa ultimului nod din listă

IteratorLista

l: Lista//referință către listă
curent:↑ Nod//adresa nodului curent din listă

Pentru operația de adăugare, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă.

Funcția creeazaNod(*l*,*e*)

```
{pre: l: Lista, e: TElement}
{post: se returnează un ↑ Nod conținând e ca informație utilă}
{se alocă un spațiu de memorare pentru un Nod }
{p: ↑ Nod}
aloca(p)
[p].e ← e
[p].urm ← NIL
[p].prec ← NIL
{rezultatul returnat de funcție}
creeazaNod ← p
```

SfFuncția

- Complexitate: $\theta(1)$

Subalgoritm adaugaDupa(*l*,*i*,*e*)

```
{pre: l: Lista, i: IteratorLista, i este valid, e: TElement}
{post: se adaugă e după nodul curent al lui i}
nou ← creeazaNod(l,e)
p ← i.curent
{se va adaugă după p }
```

```

{dacă  $p$  este ultimul nod al listei }
Dacă  $p = l.\text{ultim}$  atunci
    { $p$  este diferit de NIL, din precondiție }
    [ $l.\text{ultim}$ ].urm  $\leftarrow$   $nou$ 
    [ $nou$ ].prec  $\leftarrow$   $l.\text{ultim}$ 
    {se actualizează ultim}
     $l.\text{ultim} \leftarrow nou$ 
altfel
    {se adaugă între  $p$  și [ $p$ ].urm}
    [ $nou$ ].urm  $\leftarrow$  [ $p$ ].urm
    [[ $p$ ].urm].prec  $\leftarrow$   $nou$ 
    [ $p$ ].urm  $\leftarrow$   $nou$ 
    [ $nou$ ].prec  $\leftarrow$   $p$ 
SfDaca
SfSubalgoritm

```

- Complexitate: $\theta(1)$)

În directorul TAD Lista (Curs 5) găsiți implementarea parțială, în limbajul C++, a containerului **Lista cu poziție iterator** (reprezentarea este sub forma unei LDI, folosind alocare dinamică pentru reprezentarea înlanțuirilor).

Concluzii - Liste

- Memorarea elementelor listei **secvențial** într-un tablou unidimensional (vector).
 - eficientă pentru acele liste în care se fac multe operații de adăugare la sfârșit, accesare și mai puține inserări.
 - dacă se folosește un tablou static, deficiența este dată de gestionarea inefficientă a spațiului de memorare (este deseori necesar să se supraestimeze spațiul necesar memorării elementelor).
 - tabloul dinamic exclude dezavantajul tablourilor statice de stabilire statică a capacității maxime a unei liste, dar totuși rămâne dezavantajul dat de ineficiența operațiilor de inserare și ștergere a elementelor din interiorul listei. Inserările și ștergerile, într-o astfel de listă, se fac dificil deoarece necesită deplasări ale elementelor.
- Reprezentarea **înlănțuită**.
 - spațiu adițional pentru memorarea legăturilor - ceea ce conduce la creșterea complexității-spațiu
 - gestionarea memoriei se face mai eficient
 - operațiile de inserare și ștergere se pot face mult mai eficient.
- Decizia asupra alegerii modului de implementare a unei liste depinde de gradul de dinamicitate al listei și de tipul aplicațiilor în care urmează a fi folosită:
 - Dacă actualizările (inserări, ștergeri) sunt rare, este preferată reprezentarea folosind tablouri.
 - Dacă actualizările sunt dese, este preferată reprezentarea înlănțuită.
- În funcție de restricțiile de acces și actualizare a elementelor unei liste, există diferite specializări ale listelor: *stive*, *cozi*, *cozi complete*, liste liniare generalizate.