

# Lista înlănțuită

## cu reprezentarea înlănțuirilor pe tablou

După cum am arătat în **Cursul 4**, există două modalități de a reprezenta înlănțuirile pentru o listă înlănțuită:

- folosind **alocare dinamică** (pointeri)
  - această modalitate de reprezentare a înlănțuirilor a fost discutată în **Cursul 4**.
- folosind **tablouri** statice/dinamice
  - această modalitate de reprezentare a înlănțuirilor o vom discuta în continuare.

### 1. Lista simplu înlănțuită (LSI) - reprezentarea înlănțuirilor pe tablou

---

Într-o astfel de reprezentare, atât elementele, cât și legăturile se memorează folosind două tablouri. Presupunem ca  $Max$  este capacitatea maximă de memorare a tabloului. Locațiile se consideră a fi numerotate de la 1 la  $Max$ .

- **elementele** (informațiile utile din nodurile listei) se memorează în tabloul  $e[1..Max]$  (indexarea poate fi  $0..Max-1$  la implementarea în C/C++)
- **legăturile** dintre nodurile listei se memorează în tabloul  $urm[1..Max]$  (indexarea poate fi  $0..Max-1$  la implementarea în C/C++)
  - $urm[i]$  este indicele din tabloul  $e$  unde se memorează nodul care urmează în listă după nodul  $i$
- echivalentul legăturii **NIL** (folosită în cazul reprezentării înlănțuirilor folosind alocare dinamică) este  $urm[i]=0$ 
  - în cazul în care vectorii  $e$  și  $urm$  sunt indexați de la 0, echivalentul legăturii **NIL** va fi  $urm[i]=-1$
- valoarea unui element din vectorul  $urm$  este un indice între 1 și  $Max$
- corespondența dintre implementarea listei înlănțuite prin adrese (ex. nod a cărui adresă de memorare este  $p$ ) și prin “indici” (nod indicat prin indicele  $i$ ) este dată în Tabela 1.
- capacitatea vectorilor  $e$  și  $urm$  poate fi mărită dinamic, dacă este necesar - numărul de elemente din listă depășește numărul de locații alocat inițial (a se vedea **vectorul dinamic**).
  - după redimensionare și copierea elementelor, e necesară reinițializarea listei înlănțuite a spațiului liber (folosind locațiile nou adăugate în vector)
  - în cazul redimensionării, considerând lista reprezentată simplu înlănțuit, operația *adaugaInceput* va avea complexitatea timp **amortizată**  $\theta(1)$ .

	Alocare dinamică	Înănțuiri pe tablou
nod din listă	$p$	$i$
informația utilă a unui nod	$[p].e$	$e[i]$
legătura unui nod	$[p].urm$	$urm[i]$

Tabela 1: Corespondența dintre implementarea listei înlănțuite prin adrese și prin “indici”

- LSI va memora *prim* (indicele la care este memorat primul nod al listei), eventual *ultim*
  - lista este **vidă** dacă  $prim = 0$  (sau -1, în cazul în care indexarea vectorilor începe de la 0).

### Exemplu

Să considerăm lista  $l=(a, b, c, d, e, f)$  reprezentată simplu înlănțuit, cu înlănțuirile reprezentate folosind un tablou având capacitatea maximă (inițială) de 10 locații ( $Max = 10$ ).

Reprezentarea este dată în Tabela 2.

Indice	1	2	3	4	5	6	7	8	9	10
e	c	-	e	a	-	b	-	-	d	f
urm	9	5	10	6	8	1	2	0	3	0

Tabela 2:  $prim=4$

În cazul memorării elementelor listei și a înlănțurilor pe tablou, avem nevoie de un mecanism pentru gestionarea spațiului liber rămas în tablouri pentru memorarea altor elemente.

- $\Rightarrow$  lista înlănțuită a spațiului liber, a cărui prim element îl numim **primLiber**
- pentru exemplul din Tabela 2, lista înlănțuită a spațiului liber poate fi  $\boxed{7} \rightarrow \boxed{2} \rightarrow \boxed{5} \rightarrow \boxed{8}$ 
  - în cazul de mai sus  $primLiber=7$ , iar ultimul spațiu liber este 8.
  - semnificația elementelor din lista spațiului liber este următoarea: locațiile din tabloul *e* (în care sunt memorate elementele) având indicii 7, 8, 5 și 2 sunt **libere** (nu au memorate elemente ale listei)

În plus, trebuie furnizate 2 operații **rapide** ( $\theta(1)$ ) pentru *alocarea* și *delocarea* unui spațiu liber

- similar cu ceea ce oferă compilatoarele pentru alocarea/dealocarea pointerilor (**new/delete** în C++)

- operațiile **aloca**/**dealoca** pe care le folosim în Pseudocod, conform convențiilor noastre

### Cum se face alocarea?

- se șterge prima valoare din lista înlănțuită a spațiului liber (**primLiber**)
  - complexitatea timp a operației e  $\theta(1)$

### Cum se face dealocarea?

- se adaugă (în față) o valoare în lista înlănțuită a spațiului liber (înainte de **primLiber**)
  - complexitatea timp a operației e  $\theta(1)$

Pe lângă cele două operații anterior menționate, vom avea nevoie de inițializarea spațiului liber din listă (la început, în constructorul listei).

### Cum se face inițializarea listei spațiului liber?

- toate locațiile ( $1..Max$ ) trebuie adăugate în lista înlănțuită a spațiului liber și **primLiber** setat corespunzător
- de exemplu, se poate inițializa lista spațiului liber astfel:  $\boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3} \rightarrow \dots \rightarrow \boxed{Max}$ 
  - ulterior, datorită dinamicității listei (adăugări, ștergeri) lista spațiului liber nu va mai conține locațiile în ordine
  - complexitatea timp a operației e  $\theta(Max)$  (operația va fi descrisă în Pseudocod, mai jos)
- se poate alege orice modalitate de înlănțuire a locațiilor în lista spațiului liber (ex.  $Max \rightarrow Max-1 \rightarrow \dots \rightarrow 1$ , sau se pot înlănțui locațiile în ordine aleatoare)

### 1.1. Exemplu LSI - reprezentarea înlănțuirilor pe tablou

Pentru reprezentarea LSI pe tablou, avem nevoie de următoarea structură

#### LSI

cp: Intreg //capacitatea de memorare a celor doi vectori  
 prim, primLiber: 0..cp //numere întregi, a căror valoare e în intervalul [0, cp]  
 e: TElement[] //vectorul de elemente  
 urm: Intreg[] //vectorul de legături - valorile sunt în intervalul [0, cp]

Vom începe prin a descrie operațiile de:

- *alocare* spațiu liber în listă
  - pentru similitudine cu alocarea dinamică, vom numi această operație **aloca**

- *dealocare* spațiu (care a fost liber)
  - pentru similitudine cu alocarea dinamică, vom numi această operație **dealloca**
- inițializarea listei spațiului liber
  - operația o vom numi **initSpatiuLiber**

De menționat că operațiile **aloca**, **dealoca**, **initSpatiuLiber** NU vor fi în interfața containerelor implementate folosind o LSI cu înlănțuiri pe tablou, ci doar în partea de implementare.

- în clasa care implementează containerul, operațiile vor fi în secțiunea **private**, nu **public**

```
Subalgoritm aloca(l, i)
  {furnizează un spațiu liber de indice i }
  {se șterge primul nod din lista spațiului liber}
  i ← l.primLiber
  l.primLiber ← l.urm[l.primLiber]
SfSubalgoritm
```

- Complexitate:  $\theta(1)$

Trebuie menționat faptul că spațiul liber *i* rezultat în urma apelului **aloca**(*l, i*) poate fi 0, caz în care lista nu mai are spațiu liber, fiind necesară redimensionarea.

```
Subalgoritm dealoca(l, i)
  {trece poziția i în lista spațiului liber }
  {se adaugă i la începutul listei spațiului liber}
  l.urm[i] ← l.primLiber
  l.primLiber ← i
SfSubalgoritm
```

- Complexitate:  $\theta(1)$

```
Subalgoritm initSpatiuLiber(l, cp)
  {creează o LSI de lungime cp - toate pozițiile din tablou sunt libere }
  Pentru i = 1, cp - 1 executa
    l.urm[i] ← i + 1
  SfPentru
  l.urm[cp] ← 0
  l.primLiber ← 1
SfSubalgoritm
```

- Complexitate:  $\theta(cp)$

Indice	1	2	3	...	$cp-1$	$cp$
e	-	-	-	-	-	-
urm	2	3	4	...	$cp$	0

Tabela 3: Operația de inițializare a spațiului liber (**initSpatiuLiber**).  $primLiber=1$

Descriem, în continuare, constructorul listei, care inițializează lista cu lista vidă.

```

Subalgoritm creeaza( $l, cp$ )
    { creează lista vidă }
    { se inițializează lista spațiului liber }
    initSpatiuLiber( $l, cp$ )
     $l.prim \leftarrow 0$ 
SfSubalgoritm

```

- Complexitate:  $\theta(cp)$

Vom descrie, în continuare, operația **adaugaInceput**. Similar cu cazul listei înlănțuite cu alocare dinamică, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă. Specificația funcției a fost descrisă în **Cursul 4**.

Crearea unui nod în lista simplu înlănțuită cu înlănțuiri pe tablou ( $i$  este o poziție liberă,  $e$  este informația utilă care trebuie memorată) este ilustrată mai jos.

indice	$i$
element	<b>e</b>
următor	<b>0</b>

Tabela 4: Crearea unui nod în lista simplu înlănțuită cu înlănțuiri pe tablou ( $i$  este o poziție liberă,  $e$  este informația utilă care trebuie memorată) este ilustrată mai jos

```

Funcția creeazaNod( $l, e$ )
    { dacă nu mai există spațiu de memorare în listă }
    Dacă  $l.primLiber = 0$  atunci
        @realocare vectori
        @reinițializare spațiu liber
    SfDacă
    aloca( $l, i$ )
    { sigur  $i$  e diferit de 0 }
     $l.e[i] \leftarrow e$ 
     $l.urm[i] \leftarrow 0$ 
    creeazaNod  $\leftarrow i$ 
SfFuncția

```

- Complexitate:  $\theta(1)$ ) amortizat, dacă se folosesc vectori dinamici (redimensionare)

```

Subalgoritm adaugaInceput( $l, e$ )
    { adaugă elementul  $e$  la începutul listei }
     $nou \leftarrow creeazaNod(l, e)$ 
    { adaugăm elementul  $e$  înainte de prim }
     $l.urm[nou] \leftarrow l.prim$ 

```

```

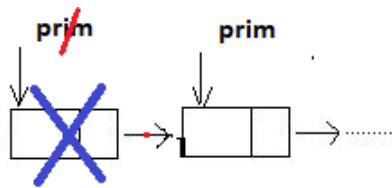
    l.prim ← nou
SfSubalgoritm

```

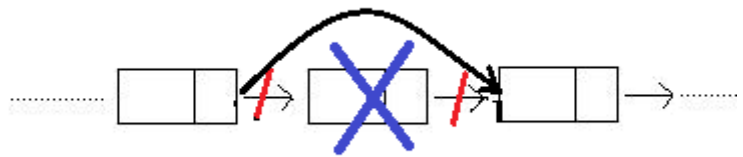
- Complexitate:  $\theta(1)$ ) amortizat, dacă se folosesc vectori dinamici (redimensionare)

Subalgoritmul pentru ștergerea unui element de pe o poziție  $p$  din listă (indicată prin indicele din tablou). . Sunt 2 cazuri la ștergere:

- se șterge  $prim$  (Figura 1 (a));
- se șterge un nod  $p$  diferit de  $prim$  (Figura 1 (b)).



(a) Ștergere  $prim$ .



(b) Ștergere nod  $p$  diferit de  $prim$ .

Figura 1

```

Subalgoritm sterge( $l, p, e$ )
    {se șterge nodul  $p$ ,  $p \neq 0$ ,  $e$  e valoarea ștersă}
    { $e$  e elementul șters }
     $e \leftarrow l.e[p]$ 
    Dacă  $p = l.prim$  atunci
        {se șterge  $prim$ }
         $l.prim \leftarrow l.urm[p]$ 
    altfel
        {se parcurge pâna la nodul  $p$ }
         $q \leftarrow l.prim$ 
        {sigur  $p$  este în listă, prin precondiție}
        CatTimp  $l.urm[q] \neq p$  executa
             $q \leftarrow l.urm[q]$ 
        SfCatTimp
        { $q$  este nodul care precede  $p$  }
        {se șterge nodul  $p$  }
         $l.urm[q] \leftarrow l.urm[p]$ 
    SfDacă
    {adăugăm locația  $p$  în lista spațiului liber }
    dealoca( $l, p$ )
SfSubalgoritm

```

- Complexitate:  $O(n)$ ,  $n$  fiind numărul de elemente din listă
  - cazul favorabil  $\theta(1)$  - șterg la început/sfârșit
  - cazul defavorabil  $\theta(n)$  - șterg penultimul element al listei

## 1.2. Exemplu implementare iterator pe un container reprezentat sub forma unei LSI cu înlănțuiri pe tablou

Presupunem că avem un **Container** oarecare (de ex. Colecție) reprezentat sub forma unei LSI cu înlănțuiri pe tablou, după cum urmează.

### Container

cp: Intreg //capacitatea de memorare a celor doi vectori  
prim, primLiber: 0..cp //numere întregi, a căror valoare e în intervalul [0, cp]  
e: TElement[] //vectorul de elemente  
urm: Intreg[] //vectorul de legături - valorile sunt în intervalul [0, cp]

În acest caz, iteratorul pe Container ar trebuie să conțină:

- o referință către container
- adresa unui nod din lista simplu înlănțuită folosită pentru reprezentarea containerului (*curent*)

### IteratorContainer

c : Container //containerul pe care îl iterează  
curent: Intreg //poziția (indicele) nodului curent al LSI

Operațiile specifice ale iteratorului (creează, valid, element, următor) le vom descrie, mai jos, în Pseudocod. Toate operațiile au complexitate timp  $\theta(1)$ .

#### Subalgoritm creeaza(*i, c*)

```
{pre: c este un container}
{post: se creează iteratorul i pe containerul c}
{      elementul curent al iteratorului referă primul element din c}
{se setează containerul în iterator}
i.c ← c
{se setează elementul curent al iteratorului}
i.curent ← c.prim
```

SfSubalgoritm

#### Funcția valid(*i*)

```
{pre: i este un iterator}
{post: se verifică dacă elementul curent este valid}
{iteratorul este valid dacă elementul curent este diferit de 0}
valid ← i.curent ≠ 0
```

SfFuncția

#### Subalgoritm element(*i, e*)

```
{pre: i este un iterator, i este valid}
{post: e este elementul indicat de curent}
e ← l.e[i.curent]
```

SfSubalgoritm

#### Subalgoritm urmator(*i*)

```

{pre: i este un iterator, i este valid}
{post: se deplasează referința curent a iteratorului}
i.curent ← l.urrm[i.curent]
SfSubalgoritm

```

În directorul asociat Cursului 6 găsiți implementarea parțială, în limbajul C++, a containerului **Colecție** (reprezentarea este sub forma unei LSI care memorează toate elementele colecției, folosind reprezentarea înlănțuirilor pe vector STATIC).

**TEMĂ.** Scrieți în Pseudocod/implementați operațiile specifice pe LSI ordonată cu înlănțuirile reprezentate pe tablou (LSIO) și deduceți complexitățile acestora.

## 2. Lista dublu înlănțuită (LDI) - reprezentarea înlănțuirilor pe tablou

Convențiile de memorare sunt aceleași ca la LSI cu înlănțuiri pe tablou, doar se adaugă un treilea vector **prec** pentru a memora legăturile spre nodurile precedente din listă.

Să considerăm lista  $l=(a, b, c)$  reprezentată dublu înlănțuit, cu înlănțuirile reprezentate folosind un tablou având capacitatea maximă de 8 locații ( $Max = 10$ ). Locațiile se consideră a fi numerotate de la 1 la  $Max$ .

Reprezentarea este dată în Tabela 3.

Indice	1	2	3	4	5	6	7	8
e	-	a	-	-	b	-	-	c
urm	3	5	4	6	8	7	0	0
prec		0			2			5

Tabela 5:  $prim=2$ ,  $ultim=8$ ,  $primLiber=1$

### Observații

- Reprezentarea listei va fi

#### LDI

cp: Intreg //capacitatea de memorare a celor doi vectori

prim, ultim, primLiber: 0..cp //numere întregi, a căror valoare e în  $[0, cp]$

e: TElement[] //vectorul de elemente

urm, prec: Intreg[] //vectorii de legături - valorile sunt în intervalul  $[0, cp]$

- lista înlănțuită a spațiului liber e suficient să fie simplu înlănțuită.

- pentru exemplul din Tabela 3, lista înlănțuită a spațiului liber este  $\boxed{1} \rightarrow \boxed{3} \rightarrow \boxed{4} \rightarrow \boxed{6} \rightarrow \boxed{7}$
- operațiile **aloca**, **dealoca**, **initSpatiuLiber** sunt ca și la LSI, fiind valabil tot ce s-a discutat în Secțiunea 1.
- capacitatea vectorilor *e*, *urm* și *prec* poate fi mărită dinamic, dacă este necesar - numărul de elemente din listă depășește numărul de locații alocate inițial (a se vedea **vectorul dinamic**).



- \* după dimensionare și copierea elementelor, e neceară reinițializarea listei înălțuite a spațiului liber (folosind locațiile nou adăugate în vector)

Vom descrie, în continuare, operația **adaugaInainte**. Similar cu cazul listei simplu înălțuite cu înălțuiri pe tablou, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă. Specificația funcției a fost descrisă în **Cursul 4**.

Crearea unui nod în lista dublu înălțuită cu înălțuiri pe tablou ( $i$  este o poziție liberă,  $e$  este informația utilă care trebuie memorată) este ilustrată mai jos.

indice	$i$
element	$e$
următor	$0$
precedent	$0$

Tabela 6: Crearea unui nod în lista dublu înălțuită cu înălțuiri pe tablou ( $i$  este o poziție liberă,  $e$  este informația utilă care trebuie memorată) este ilustrată mai jos

```

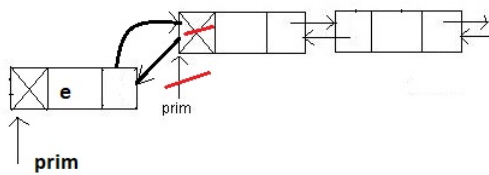
Functia creeazaNod( $l, e$ )
    {dacă nu mai există spațiu de memorare în listă}
    Daca  $l.\text{primLiber} = 0$  atunci
        @realocare vectori
        @reinițializare spațiu liber
    SfDaca
    aloca( $l, i$ )
     $l.e[i] \leftarrow e$ 
     $l.urm[i] \leftarrow 0$ 
     $l.pec[i] \leftarrow 0$ 
    creeazaNod  $\leftarrow i$ 
SfFunctia

```

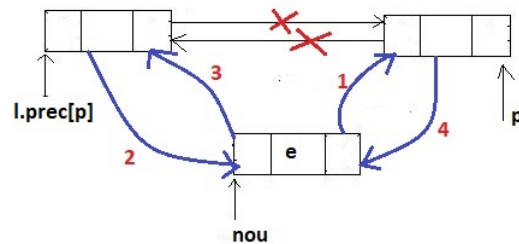
- Complexitate:  $\theta(1)$ ) amortizat, dacă se folosec vectori dinamici (redimensionare)

Subalgoritmul pentru adăugarea unui element înaintea unui nod din listă (indicat prin indicele la care e memorat în tablou). Sunt două cazuri care trebuie tratate

- adăugare înainte de  $prim$  (dacă  $p = prim$ ) (Figura 2(a))
- adăugare înainte de un nod  $p$  diferit de  $prim$  (Figura 2(b))



(a) Adăugare element la început.



(b) Adăugare element înaintea unui nod  $p$  diferit de  $prim$ .

Figura 2

Subalgoritm  $adaugaInainte(l, p, e)$

{pre:  $l$ : LDI,  $p$ ,  $p \neq 0$  este poziția unui nod (indice) în  $l$ ,  $e$ : TElement}

{post: se adaugă  $e$  înaintea nodului  $p$ }

$nou \leftarrow creeazaNod(l, e)$

{dacă se adaugă înaintea primului nod }

Dacă  $p = l.prim$  atunci

{se adaugă înainte de  $prim$ }

$l.urm[nou] \leftarrow l.prim$

{ $p$  este diferit de 0, prin precondiție}

$l.prec[l.prim] \leftarrow nou$

{se actualizează  $prim$ }

$l.prim \leftarrow nou$

altfel

{se adaugă între precedentul lui  $p$  și  $p$ }

$l.urm[nou] \leftarrow p$

$l.urm[l.prec[p]] \leftarrow nou$

$l.prec[nou] \leftarrow l.prec[p]$

$l.prec[p] \leftarrow nou$

SfDacă

SfSubalgoritm

- Complexitate:  $\theta(1)$  amortizat, dacă se folosesc vectori dinamici (redimensionare)

## Observație

- Implementarea iteratorului pe un container oarecare (de ex. Colecție) reprezentat sub forma unei LDI cu înlănțuiri pe tablou este similar cu cel descris în Secțiunea 1.2. Spre deosebire de cazul LSI, iteratorul poate fi bidirecțional.

**TEMĂ.** Scrieți în Pseudocod operațiile specifice pe LDI/LDIO (ordonată) cu înlănțuirile reprezentate pe tablou și deduceți complexitățile acestora.