Lista înlănţuită
cu reprezentarea înlănţuirilor pe tablou

După cum am arătat în **Cursul 4**, există două modalităţi de a reprezenta înlănţuirile pentru o listă înlănţuită:

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

1. Lista simplu înlănţuită (LSI) - repezentarea înlănţuirilor pe tablou

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

- **elementele** (informaţiile 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 înlănţuirilor folosind alocare dinamică) este **urm**[i]=0
  - în cazul în care vectorii **e** şi **urm** sunt indexaţi de la 0, echivalentul legăturii **NIL** va fi **urm**[i]=-1
- valoarea unui element din vectorul **urm** este un indice între 1 şi Max
- corespondenţa dintre implementarea listei înlănţuite prin adrese (ex. nod a cărui adresă de memorare este p) şi prin "indici" (nod indicat prin indicele i) este dată în Tabela 1.
- capacitatea vectorilor e şi urm poate fi mărită dinamic, dacă este necesar - numărul de elemente din listă depăşeşte numărul de locaţii alocat iniţial (a se vedea **vectorul dinamic**).
  - după redimensionare şi copierea elementelor, e necesară reiniţializarea listei înlănţuite a spaţiului liber (folosind locaţiile nou adăugate în vector)
  - în cazul redimensionării, considerând lista reprezentată simplu înlănţuit, operaţia $adaugaInceput$ va avea complexitatea timp **amortizată** $\theta(1)$.

	Alocare dinamică	**Înănţuiri pe tablou**
nod din listă	p	i
informaţia utilă a unui nod	$[p]$.e	e$[i]$
legătura unui nod	$[p]$.urm	urm$[i]$

Tabela 1: Corespondenţa dintre implementarea listei înlănţuite prin adrese şi prin "indici"

- LSI va memora *prim* (indicele la care este memorat pimul 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 înlănţuit, cu înlănţuirile reprezentate folosind un tablou având capacitatea maximă (iniţială) de 10 locaţii ($Max = 10$).

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 şi a înlănţurilor pe tablou, avem nevoie de un mecanism pentru gestionarea spaţiului liber rămas în tablouri pentru memorarea altor elemente.

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

În plus, trebuie furnizate 2 operaţii **rapide** ($\theta(1)$) pentru *alocarea* şi *delocarea* unui spaţiu liber

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

  - operaţiile **aloca**/ **dealoca** pe care le folosim în Pseudocod, conform convenţiilor noastre

Cum se face alocarea?

- se şterge prima valoare din lista înlănţuită a spaţiului liber (**primLiber**)
  - complexitatea timp a operaţiei e $\theta(1)$

Cum se face dealocarea?

- se adaugă (în faţă) o valoare în lista înlănţuită a spaţiului liber (înainte de **primLiber**)
  - complexitatea timp a operaţiei e $\theta(1)$

Pe lângă cele două operaţii anterior menţionate, vom avea nevoie de iniţializarea spaţiului liber din listă (la început, în constuctorul listei).

Cum se face iniţializarea listei spaţiului liber?

- toate locaţiile ($1..Max$) trebuie adăugate în lista înlănţuită a spaţiului liber şi **primLiber** setat corespunzător
- de exemplu, se poate iniţializa lista spaţiului liber astfel: $\boxed{\mathbf{1}} \rightarrow \boxed{\mathbf{2}} \rightarrow \boxed{\mathbf{3}} \rightarrow \ldots \boxed{Max}$
  - ulterior, datorită dinamicităţii listei (adăugări, ştergeri) lista spaţiului liber nu va mai conţine locaţiile în ordine
  - complexitatea timp a operaţiei e $\theta(Max)$ (operţia va fi decrisă în Pseudocod, mai jos)
- se poate alege orice modalitatate de înlănţuire a locaţiilor în lista spaţiului liber (ex. $Max \rightarrow Max$-1 $\rightarrow \cdots \rightarrow 1$, sau se pot înlănţui locaţiile în ordine aleatoare)

1.1. Exemplu LSI - reprezentarea înlănţuirilor 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 operaţiile de:

- *alocare* spaţiu liber în listă
  - pentru similitudine cu alocarea dinamică, vom numi această operaţie **alocă**

- *dealocare* spaţiu (care a fost liber)
  - pentru similitudine cu alocarea dinamică, vom numi această operaţie **dealocă**
- iniţializarea listei spaţiului liber
  - operaţia o vom numi **initSpatiuLiber**

De menţionat că operaţiile **aloca**, **dealoca**, **initSpatiuLiber** NU vor fi în interfaţa containerelor implementate folosind o LSI cu înlănţuiri pe tablou, ci doar în partea de implementare.

- în clasa care implementează containerul, operaţiile vor fi în secţiunea **private**, nu **public**

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

- Complexitate: $\theta(1))$

Trebuie menţiont faptul că spaţiul liber i rezultat în uma apelului **aloca**(l, i) pote fi 0, caz în care lista nu mai are spaţiu liber, fiind necesară redimensionarea.

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

- Complexitate: $\theta(1))$

```
Subalgoritm initSpatiuLiber(l, cp)
  {creează o LSI de lungime cp - tote poziţiile 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: Operaţia de iniţializare a spaţiului liber (**initSpatiuLiber**). $primLiber$=1

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

```
Subalgoritm creeaza(l, cp)
  {creează lista vidă }
  {se iniţializează lista spaţiului liber}
  initSpatiuLiber(l, cp)
  l.prim ← 0
SfSubalgoritm
```

- Complexitate: $\theta(cp)$)

Vom descrie, în continuare, operaţia **adaugaInceput**. Similar cu cazul listei înlănţuite cu alocare dinamică, vom folosi o funcţie auxiliară care creează un nod având o anumită informaţie utilă. Specificaţia funcţiei a fost descrisă în **Cursul 4**.

Crearea unui nod în lista simplu înlănţută cu înlănţuiri pe tablou (i este o poziţie liberă, e este informaţia utilă care trebuie memorată) este ilustrată mai jos.

indice	i
element	**e**
următor	**0**

Tabela 4: Crearea unui nod în lista simplu înlănţută cu înlănţuiri pe tablou (i este o poziţie liberă, e este informaţia utilă care trebuie memorată) este ilustrată mai jos

```
Functia creeazaNod(l, e)
  {dacă nu mai există spaţiu de memorare în listă}
  Daca l.primLiber = 0 atunci
    @realocare vectori
    @copiere elemente
    @reiniţializare spaţiu liber
  SfDaca
  aloca(l, i)
  {sigur i e diferit de 0}
  l.e[i] ← e
  l.urm[i] ← 0
  creeazaNod← i
SfFunctia
```

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

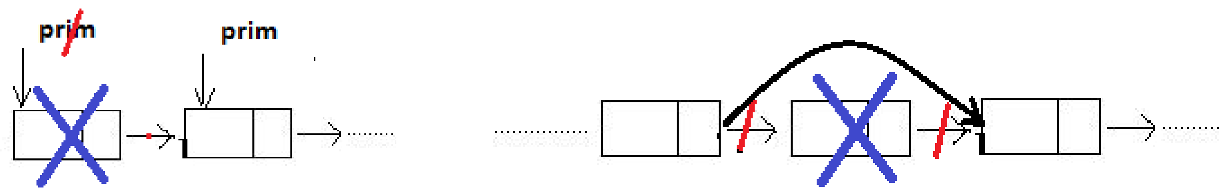
```
Subalgoritm adaugaInceput(l, e)
  {adaugă elementul e la începutul listei}
  nou ← creeazaNod(l, e)
  {adaugăm elementul e înainte de prim}
```

```
  l.urm[nou] ← l.prim
  l.prim ← nou
SfSubalgoritm
```

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

Subalgoritmul pentru ştergerea unui element de pe o poziţie p din listă (indicată prin indicele din tablou). Sunt 2 cazuri la ştergere:

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



(a) Ştergere *prim*.

(b) Ştergere nod p diferit de *prim*.

Figura 1

```
Subalgoritm sterge(l, p, e)
  {se şterge nodul p, p ≠ 0, e e valoarea ştearsă}
  {e e elementul şters }
  e ← l.e[p]
  Daca p = l.prim atunci
    {se şterge prim}
    l.prim ← l.urm[p]
  altfel
    {se parcurge pâna la nodul p}
    q ← l.prim
    {sigur p este în listă, prin precondiţie}
    CatTimp l.urm[q] ≠ p executa
      q ← l.urm[q]
    SfCatTimp
    {q este nodul care precede p }
    {se şterge nodul p }
    l.urm[q] ← l.urm[p]
  SfDaca
  {adăugăm locaţia p în lista spaţiului liber }
  dealoca(l, p)
SfSubalgoritm
```

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

1.2. Exemplu implementare iterator pe un container reprezentat sub foma unei unei LSI cu înlănţuiri pe tablou

Presupunem că avem un **Container** oarecare (de ex. Colecţie) reprezentat sub forma unei LSI cu înlănţuiri 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 conţină:

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

IteratorContainer
c : Container *//containerul pe care îl iterează*
curent: Intreg *//poziţia (indicele asociat) nodului curent al* LSI

Operaţiile specifice ale iteratorului (creează, valid, element, următor) le vom descrie, mai jos, în Pseudocod. Toate operaţiile 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


Functia 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
SfFunctia


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 referinţa curent a iteratorului}
  i.curent ← l.urrm[i.curent]
SfSubalgoritm
```

În directorul asociat Cursului 6 găsiţi implementarea parţială, în limbajul C++, a containerului **Colecţie** (reprezentarea este sub forma unei LSI care memorează toate elementele colecţiei, folosind reprezentarea înlănţuirilor pe vector STATIC).

TEMĂ. Scrieţi în Pseudocod/implementaţi operaţiile specifice pe LSI ordonată cu înlănţuirile repezentate pe tablou (LSIO) şi deduceţi complexităţile acestora.

2. Lista dublu înlănţuită (LDI) - reprezentarea înlănţuirilor pe tablou

Convenţiile de memorare sunt aceleaşi ca la LSI cu înlănţuiri 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 înlănţuit, cu înlănţuirile reprezentate folosind un tablou având capacitatea maximă de 8 locaţii ($Max = 10$). Locaţiile se cosideră 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

Observaţii

- Reprezentarea listei va fi

  <u>LDI</u>

  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 înlănţuită a spaţiului liber e suficient să fie simplu înlănţuită.
  - pentru exemplul din Tabela 3, lista înlănţuită a spaţiului liber este $\boxed{\mathbf{1}} \to \boxed{\mathbf{3}} \to \boxed{\mathbf{4}} \to \boxed{\mathbf{6}} \to \boxed{\mathbf{7}}$
  - operaţiile **aloca**, **dealoca**, **initSpatiuLiber** sunt ca şi la LSI, fiind valabil tot ce s-a discutat în Secţiunea 1.
  - capacitatea vectorilor *e*, *urm* şi *prec* poate fi mărită dinamic, dacă este necesar - numărul de elemente din listă depăşeşte numărul de locaţii alocat iniţial (a se vedea **vectorul dinamic**).

* după dimensionare şi copierea elementelor, e necesară reiniţializarea listei înlănţuite a spaţiului liber (folosind locaţiile nou adăugate în vector)

Vom descrie, în continuare, operaţia **adaugaInainte**. Similar cu cazul listei simplu înlănţuite cu înlănţuiri pe tablou, vom folosi o funcţie auxiliară care creează un nod având o anumită informaţie utilă. Specificaţia funcţiei a fost descrisă în **Cursul 4**.

Crearea unui nod în lista dublu înlănţută cu înlănţuiri pe tablou (i este o poziţie liberă, e este informaţia utilă care trebuie memorată) este ilustrată mai jos.

indice	i
element	**e**
următor	**0**
precedent	**0**

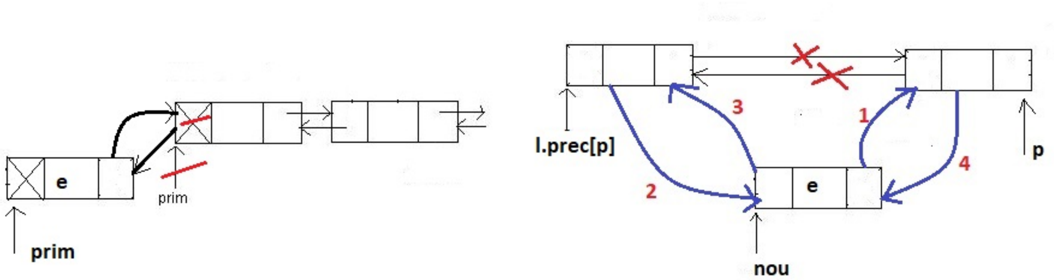
Tabela 6: Crearea unui nod în lista dublu înlănţuită cu înlănţuiri pe tablou (i este o poziţie liberă, e este informaţia utilă care trebuie memorată) este iluistrată mai jos

```
Functia creeazaNod(l, e)
  {dacă nu mai există spaţiu de memorare în listă}
  Daca l.primLiber = 0 atunci
    @realocare vectori
    @copiere
    @reiniţilizare spaţiu liber
  SfDaca
  aloca(l, i)
  l.e[i] ← e
  l.urm[i] ← 0
  l.pec[i] ← 0
  creeazaNod← 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 ≠ 0 este poziţia unui nod (indice) în l, e: TElement}
  {post: se adaugă e înaintea nodului p}
  nou ← creeazaNod(l, e)
  {dacă se adaugă înaintea primului nod }
  Daca p = l.prim atunci
    {se adaugă înainte de prim}
    l.urm[nou] ← l.prim
    {p este diferit de 0, prin precondiţie}
    l.prec[l.prim] ← nou
    {se actualizează prim}
    l.prim ← nou
  altfel
    {se adaugă între pecendentul lui p şi p}
    l.urm[nou] ← p
    l.urm[l.prec[p]] ← nou
    l.prec[nou] ← l.prec[p]
    l.prec[p] ← nou
  SfDaca
SfSubalgoritm
```

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

Observaţie

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

TEMĂ. Scrieţi în Pseudocod operaţiile specifice pe LDI/LDIO (ordonată) cu înlănţuirile repezentate pe tablou şi deduceţi complexităţile acestora.