

Operații pe liste înlănțuite

Reprezentarea înlănțuirilor folosind alocare dinamică

- Vom prezenta, în cele ce urmează, în Pseudocod, implementarea unor operații pe liste înlănțuite, folosind alocare dinamică pentru reprezentarea înlănțuirilor
- Vom prezenta reprezentarea și câteva operații specifice pe următoarele structuri de date:
 - lista simplu înlănțuită
 - lista dublu înlănțuită
 - lista simplu înlănțuită sortată

- Reamintim convențiile de notații Pseudocod pentru pointeri
 - Pentru a indica pointeri (adrese ale unor zone de memorie), vom folosi caracterul \uparrow , cu alte cuvinte dacă vrem să declarăm un pointer p care referă un număr întreg, acest lucru îl vom scrie în următoarea manieră:

$$p : \uparrow Intreg$$

Conținutul locației referite de pointerul p îl vom nota $[p]$.

- Pointerul nul (care nu referă nimic) îl vom nota prin NIL.
- Operațiile de alocare, respectiv dealocare a pointerilor le vom nota:
 - * $aloca(p)$
 - * $dealoca(p)$

Lista simplu înlănțuită (LSI)

Pentru reprezentarea LSI avem nevoie de două structuri: una pentru un **nod** și o structură pentru listă.

NodLSI

e: TElement //informația utilă a nodului

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

LSI

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

{ultim: \uparrow NodLSI} //eventual adresa ultimului nod din listă

- În general, pentru o LSI se memorează doar adresa primului element din listă (**prim**). Se poate memora și adresa ultimului element din listă (**ultim**), caz în care operația de adăugare la sfârșit va avea complexitate timp constantă $\theta(1)$.

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

Figura 1 ilustrează crearea unui nod cu o informație utilă e .

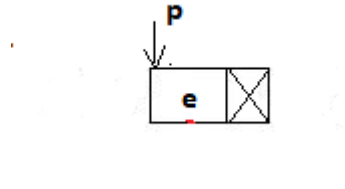


Figura 1: Creare nod cu informație utilă e .

```

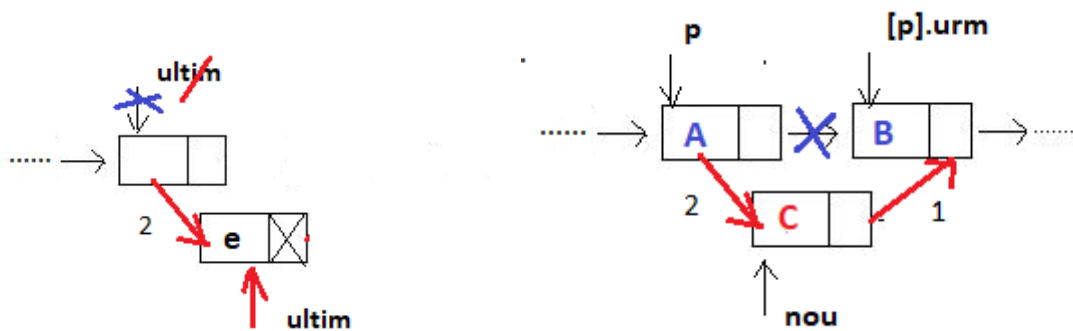
Functia creeazaNodLSI( $lsi, e$ )
{pre:  $lsi$ : LSI,  $e$ : TElement}
{post: se returnează un  $\uparrow$  NodLSI conținând  $e$  ca informație utilă}
{se alocă un spațiu de memorare pentru un NodLSI }
{ $p$ :  $\uparrow$  NodLSI}
aloca( $p$ )
 $[p].e \leftarrow e$ 
 $[p].urm \leftarrow \text{NIL}$ 
{rezultatul returnat de funcție}
creeazaNodLSI  $\leftarrow p$ 
SfFunctia

```

- Complexitate: $\theta(1)$

Subalgoritmul pentru adăugarea unui element la finalul listei este descris mai jos. Considerăm că lista memorează și adresa ultimului nod din listă (**ultim**).

Figura 2 ilustrează adăugarea la sfârșit și adăugarea după un anumit nod p .



(a) Adăugare element la sfârșit.

(b) Adăugare element după nod p diferit de $ultim$.

Figura 2

```

Subalgoritm adaugaSfarsit( $lsi, e$ )
{pre:  $lsi$ : LSI,  $e$ : TElement}
{post: se adaugă  $e$  la finalul  $lsi$ }
nou  $\leftarrow$  creeazaNodLSI( $lsi, e$ )

```

```

{dacă lista nu e vidă }
Dacă  $lsi.ultim \neq NIL$  atunci
    {se adaugă după ultim}
    [ $lsi.ultim$ ].urm  $\leftarrow nou$ 
altfel
    {nodul adăugat este și primul}
     $lsi.prim \leftarrow nou$ 
SfDacă
    {se actualizează ultim}
     $lsi.ultim \leftarrow nou$ 
SfSubalgoritm

```

- Complexitate: $\theta(1)$

Subalgoritmul pentru adăugarea unui element după un nod din listă (indicat prin adresa sa - pointer). Considerăm că lista memorează și adresa ultimului nod din listă (**ultim**). Sunt două cazuri care trebuie tratate

- adăugare după *ultim* (dacă $p = ultim$) (Figura 2(a))
- Adăugare element după nod p diferit de *ultim* (Figura 2(b))

```

Subalgoritm adaugaDupa( $lsi, p, e$ )
{pre:  $lsi$ : LSI,  $p : \uparrow \text{NodLSI}$ ,  $p \neq NIL$  este adresa unui nod din  $lsi$ ,  $e$ : TElement}
{post: se adaugă  $e$  după nodul indicat de  $p$ }
 $nou \leftarrow \text{creeazaNodLSI}(lsi, e)$ 
{dacă se adaugă după ultimul nod }
Dacă  $p = lsi.ultim$  atunci
    {se adaugă după ultim care e diferit de NIL, din precondiție}
    [ $lsi.ultim$ ].urm  $\leftarrow nou$ 
    {se actualizează ultim}
     $lsi.ultim \leftarrow nou$ 
altfel
    {se adaugă între  $p$  și [ $p$ ].urm}
    [ $nou$ ].urm  $\leftarrow [p].urm$ 
    [ $p$ ].urm  $\leftarrow nou$ 
SfDacă
SfSubalgoritm

```

- Complexitate: $\theta(1)$

Subalgoritmul pentru ștergerea unui nod din listă (indicat prin adresa sa - pointer). Considerăm că lista memorează și adresa ultimului nod din listă (**ultim**). Sunt 3 cazuri la ștergere:

- se șterge *prim* (Figura 3 (a));
- se șterge *ultim* (Figura 3 (b));
- se șterge un nod p diferit de *prim* și *ultim* (Figura 4).

```

Subalgoritm sterge( $lsi, p, e$ )
{pre:  $lsi$ : LSI,  $p : \uparrow \text{NodLSI}$ ,  $p \neq NIL$  este adresa unui nod din listă}

```

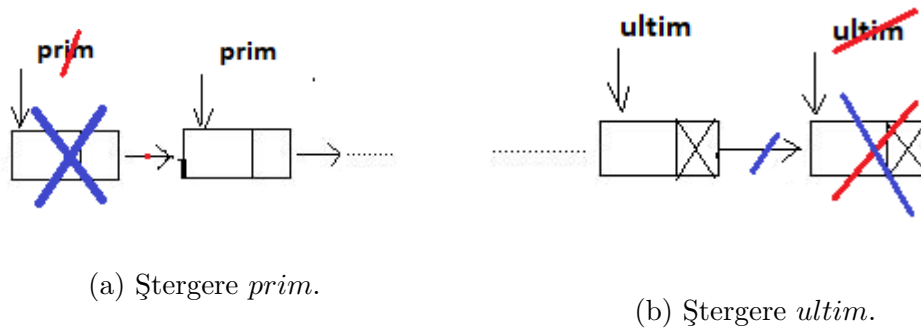


Figura 3

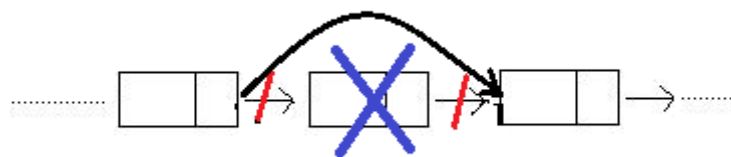


Figura 4: Ștergere nod p diferit de *prim* și *ultim*.

```

{post: se șterge din listă nodul indicat de  $p$ ,  $e$ : TElement este elementul
șters}
{elementul șters}
 $e \leftarrow [p].e$ 
{dacă se șterge primul element al listei}
Dacă  $p = lsi.prim$  atunci
    {se modifică prim}
     $lsi.prim \leftarrow [p].urm$ 
    {dacă noul prim e NIL, atunci lista e vidă}
    Dacă  $lsi.prim = NIL$  atunci
         $lsi.ultim \leftarrow NIL$ 
    SfDacă
altfel
    {se parcurge până la nodul  $p$ }
     $q \leftarrow lsi.prim$ 
    {sigur  $p$  este în listă, prin precondiție}
    CatTimp  $[q].urm \neq p$  executa
         $q \leftarrow [q].urm$ 
    SfCatTimp
    { $q$  este nodul care precede  $p$ }
    {dacă se șterge ultimul element al listei}
    Dacă  $p = lsi.ultim$  atunci
         $lsi.ultim \leftarrow q$ 
        {lista nu poate fi vidă, s-a tratat cazul la început}
    altfel
        {se șterge nodul  $p$ }
         $[q].urm \leftarrow [p].urm$ 
    SfDacă
SfDacă

```

```

    {dealocăm spațiul de memorare pentru  $p$  }
    dealoca( $p$ )
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

Iterator pe un container reprezentat folosind o LSI

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

NodLSI

e: TElement *//informația utilă nodului*
 urm: \uparrow NodLSI *//adresa la care e memorat următorul nod*

Container

prim: \uparrow NodLSI *//adresa primului nod din listă*

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

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

IteratorContainer

c : Container *//containerul pe care îl iterează*
 curent: \uparrow NodLSI *//adresa unui nod 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 }
 $i.c \leftarrow c$ 
{se setează elementul curent al iteratorului }
 $i.curent \leftarrow c.prim$ 

```

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 NIL }

```

```
valid  $\leftarrow i.curent \neq \text{NIL}$   
SfFunctia
```

```
Subalgoritm element( $i, e$ )  
  {pre:  $i$  este un iterator,  $i$  este valid}  
  {post:  $e$  este elementul indicat de curent}  
   $e \leftarrow [i.curent].e$   
SfSubalgoritm
```

```
Subalgoritm urmator( $i$ )  
  {pre:  $i$  este un iterator,  $i$  este valid}  
  {post: se deplasează referința curent a iteratorului}  
   $i.curent \leftarrow [i.curent].urm$   
SfSubalgoritm
```

În directorul asociat Cursului 4 găsiți implementarea parțială, în limbajul C++, a containerului **Colecție** (reprezentarea este sub forma unei LSI care memorează toate elementele colecției, folosind alocare dinamică pentru reprezentarea înlănțuirilor).

TEMĂ. Scrieți în Pseudocod/implementați restul operațiilor specifice pe LSI și deduceți complexitățile acestora:

- adăugare element la începutul listei ($adaugaInceput(lsi, e)$)
- adăugare element înaintea unui nod dat ($adaugaInainte(lsi, p, e)$)
- căutarea unui element dat în listă ($cauta(lsi, e)$)

Lista dublu înlănțuită (LDI)

Pentru reprezentarea LDI avem nevoie de două structuri: una pentru un **nod** și o structură pentru listă.

NodLDI

e : TElement //informația utilă nodului
 urm : \uparrow NodLDI //adresa la care e memorat următorul nod
 $prec$: \uparrow NodLDI //adresa la care e memorat nodul precedent

LDI

$prim$: \uparrow NodLDI //adresa primului nod din listă
 $ultim$: \uparrow NodLDI // adresa ultimului nod din listă

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

```
Functia creeazaNodLDI( $ldi, e$ )  
  {pre:  $ldi$ : LDI,  $e$ : TElement}  
  {post: se returnează un  $\uparrow$  NodLDI conținând  $e$  ca informație utilă}  
  {se alocă un spațiu de memorare pentru un NodLDI }
```

```

{p: ↑ NodLDI}
aloca(p)
[p].e ← e
[p].urm ← NIL
[p].prec ← NIL
{rezultatul returnat de funcție}
creeazaNodLDI ← p
SfFunctia

```

- Complexitate: $\theta(1)$

Subalgoritmul pentru adăugarea unui element înaintea unui nod din listă (indicat prin adresa sa - pointer). Sunt două cazuri care trebuie tratate

- adăugare înainte de *prim* (dacă $p = \text{prim}$) (Figura 5(a))
- adăugare înainte de un nod *p* diferit de *prim* (Figura 5(b))

Figura 5 ilustrează adăugarea la început și adăugarea înaintea unui nod *p* diferit de *prim*.

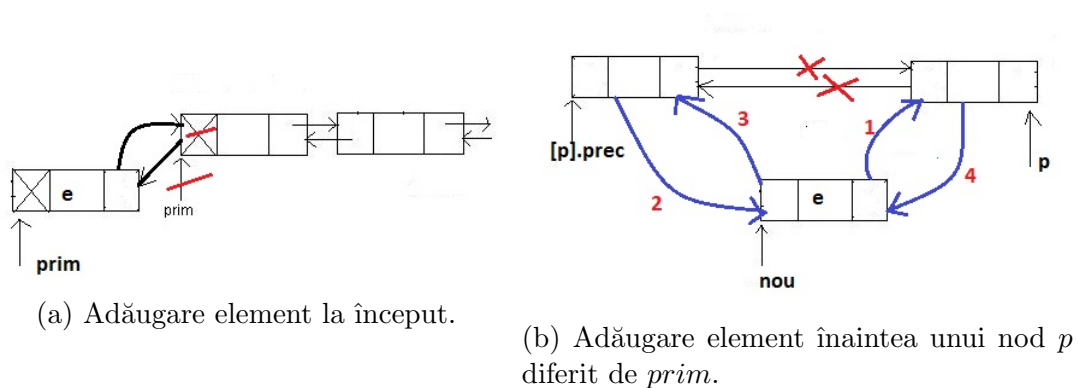


Figura 5

Subalgoritm *adaugaInainte*(*ldi*, *p*, *e*)

```

{pre: ldi: LDI, p: ↑ NodLDI, p ≠ NIL este adresa unui nod din ldi, e: TElement}
{post: se adaugă e înaintea nodului indicat de p}
nou ← creeazaNodLDI(ldi, e)
{dacă se adaugă înaintea primului nod }
Daca p = ldi.prim atunci
    {se adaugă înainte de prim}
    [nou].urm ← ldi.prim
    {p este diferit de NIL, prin precondiție}
    [ldi.prim].prec ← nou
    {se actualizează prim}
    ldi.prim ← nou
altfel
    {se adaugă între [p].prec și p}
    [nou].urm ← p
    [[p].prec].urm ← nou
    [nou].prec ← [p].prec

```

```

    [p].prec ← nou
SfDaca
SfSubalgoritm

```

- Complexitate: $\theta(1)$

TEMA

- Scrieți în Pseudocod/implementați restul operațiilor specifice pe LDI și deduceți complexitățile acestora:
 - adăugare element la începutul și sfârșitul listei
 - adăugare element după un nod dat
 - căutarea unui element dat în listă
 - ștergerea unui nod din listă
- Similar cu ce s-a prezentat pentru LSI, scrieți în Pseudocod operațiile pe iteratorul unui **Container** oarecare (de ex. Colecție) reprezentat sub forma unei LDI, folosind alocare dinamică pentru reprezentarea înlănțuirilor.

Lista simplu înlănțuită sortată/ordonată (LSIO)

Într-o LSIO, elementele sunt de **TComparabil** (**TElement**=**TComparabil**) și sunt memorate în ordine în raport cu o anumită relație de ordine $\mathcal{R} \subseteq \mathbf{TComparabil} \times \mathbf{TComparabil}$ (*reflexivă, tranzitivă și antisimetrică*)

- de exemplu, dacă $\mathcal{R} = \leq$, atunci elementele vor fi stocate în ordine crescătoare:
x. 4 7 9 11
- ca implementare, relația va fi o funcție (pointer spre funcție în C++).

De exemplu,

```

typedef int TComparabil;
typedef TComparabil TElement;
typedef bool(*Relatie)(TElement, TElement);

//relatia <=
bool relatie1(TElement e1, TElement e2) {
    if (e1 <= e2) return true;
    else return false;
}

```

Pentru reprezentarea LSIO avem nevoie de două structuri: una pentru un **nod** și o structură pentru listă.

NodLSIO

e: TElement //informația utilă nodului

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

LSIO

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

\mathcal{R} : Relatie //relația de ordine între elemente

Pe LSIO va fi definită o singură operație de **adăugare**, numită, de obicei, *inserare*, care va insera un element în LSIO astfel încât să se păstreze, după inserare, relația de ordine între elemente. Pentru operația de *inserare*, vom folosi funcția auxiliară **creeazaNodLSI** definită anterior pentru LSI.

- notăm prin $a\mathcal{R}b$ faptul că a e în relația \mathcal{R} cu b (de ex. $a \leq b$).

De exemplu, dacă vrem să inserăm un element e într-o LSIO în care $\mathcal{R} = \leq$, 3 7 9 12 18, identificăm 2 cazuri:

- adăugăm înaintea primului element: dacă $e = 1$, atunci lista devine 1 3 7 9 12 18
 - dacă se memora și ultim, se trataa și cazul adăugării după ultimul element
- adăugăm undeva în interiorul listei: dacă $e = 10$, atunci lista devine 3 7 9 10 12 18

Subalgoritm *insereza*($lsio, e$)

{pre: $lsio$: LSIO, e :TElement }

{post: se adaugă în LSIO elementul e , păstrând relația de ordine între elemente}

$nou \leftarrow creeazaNod(lsio, e)$

{dacă se adaugă la începutul listei }

Daca ($lsio.prim = NIL$) \vee ($e \mathcal{R} [lsio.prim].e$) atunci

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

$lsio.prim \leftarrow nou$

altfel

{se parcurge pâna la nodul q după care trebuie adăugat nou }

{dacă lista e 1 2 5, relația e \leq și vrem să adăugăm 3, ne oprim pe nodul q cu informația 2}

$q \leftarrow lsio.prim$

CatTimp ($[q].urm \neq NIL$) $\wedge \neg (e \mathcal{R} [[q].urm].e)$ executa

$q \leftarrow [q].urm$

SfCatTimp

{ q este nodul după care se adaugă nou }

$[nou].urm \leftarrow [q].urm$

$[q].urm \leftarrow nou$

SfDaca

SfSubalgoritm

- Complexitate: $O(n)$, n fiind numărul de elemente din listă
 - cazul favorabil $\theta(1)$ - inserez la început
 - cazul defavorabil $\theta(n)$ - adaug la finalul listei

Operația de ștergere din LSIO este similară cu cea definită anterior pe LSI. De asemenea, iteratorul pe un container reprezentat folosind o LSIO este similar cu cel definit anterior pe LSI.

TEMĂ. Analizați cazul listei dublu înlanțuite sortate/ordonate - similar cu ce s-a discutat în Secțiunea anterioară.