Operaţii pe liste înlănţuite

Reprezentarea înlănţuirilor folosind alocare dinamică

- Vom prezenta, în cele ce urmează, în Pseudocod, implementarea unor operaţii pe liste înlănţuite, folosind alocare dinamică pentru reprezentarea înlănţuirilor
- Vom prezenta reprezentarea şi câteva operaţii specifice pe următoarele structuri de date:
  - lista simplu înlănţuită
  - lista dublu înlănţuită
  - lista simplu înlănţuită sortată
- Reamintim convenţiile de notaţii 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:

    $$p : \uparrow Intreg$$

    Conţinutul locaţiei referite de pointerul p îl vom nota $[p]$.
  - Pointerul nul (care nu referă nimic) îl vom nota prin NIL.
  - Operaţiile de alocare, respectiv dealocare a pointerilor le vom nota:
    - aloca(p)
    - dealoca(p)

Lista simplu înlănţuită (LSI)

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

NodLSI
  e: TElement *//infomaţia 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 şi adresa ultimului element din listă (**ultim**), caz în care operaţia de adăugare la sfârşit va avea complexitate timp constantă $\theta(1)$.

Pentru operaţiile de adăugare, vom folosi o funcţie auxiliară care creează un nod având o anumită informaţie utilă dată.

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

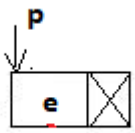


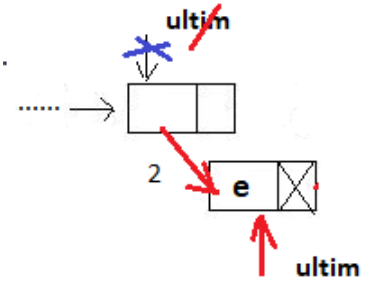
Figura 1: Creare nod cu informaţie utilă e.

```
Functia creeazaNodLSI(lsi, e)
  {pre: lsi: LSI,e: TElement}
  {post: se returnează un ↑ NodLSI conţinând e ca informaţie utilă}
  {se alocă un spaţiu de memorare pentru un NodLSI }
  {p: ↑ NodLSI}
  aloca(p)
  [p].e ← e
  [p].urm ← NIL
  {rezultatul returnat de funcţie}
  creeazaNodLSI ← p
SfFunctia
```

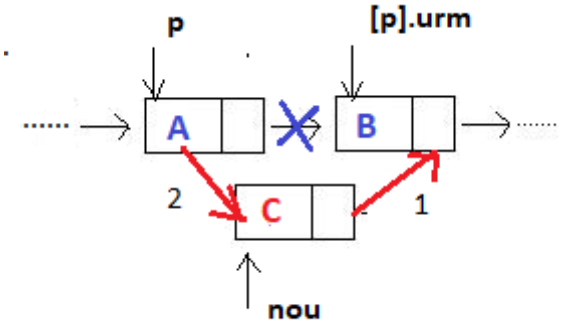
- Complexitate: $\theta(1)$)

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

Figura 2 ilustrează adăugarea la sfârşit şi adăugarea după un anumit nod p.



(a) Adăugare element la sfârşit.



(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 ← creeazaNodLSI(lsi, e)
```

```
  {dacă lista nu e vidă }
  Daca lsi.ultim ≠ NIL  atunci
    {se adaugă după ultim}
    [lsi.ultim].urm ← nou
  altfel
    {nodul adăugat este şi primul}
    lsi.prim ← nou
  SfDaca
  {se actualizează ultim}
  lsi.ultim ← nou
SfSubalgoritm
```

- Complexitate: $\theta(1)$)

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

- adăugare după *ultim* (dacă $p = ultim$) (Figura2(a))
- Adăugare element după nod p diferit de *ultim* (Figura 2(b))

```
Subalgoritm adaugaDupa(lsi, p, e)
  {pre:  lsi:  LSI, p :↑ NodLSI, p ≠ NIL este adresa unui nod din lsi, e:  TElement}
  {post:  se adaugă e după nodul indicat de p}
  nou ← creeazaNodLSI(lsi, e)
  {dacă se adaugă după ultimul nod }
  Daca p = lsi.ultim atunci
    {se adaugă după ultim care e diferit de NIL, din precondiţie}
    [lsi.ultim].urm ← nou
    {se actualizează ultim}
    lsi.ultim ← nou
  altfel
    {se adaugă între p şi [p].urm}
    [nou].urm ← [p].urm
    [p].urm ← nou
  SfDaca
SfSubalgoritm
```

- Complexitate: $\theta(1)$)

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

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

```
Subalgoritm sterge(lsi, p, e)
  {pre:  lsi:  LSI, p :↑ NodLSI, p ≠ NIL este adresa unui nod din listă}
```

(a) Ştergere *prim*.

(b) Ştergere *ultim*.

Figura 3

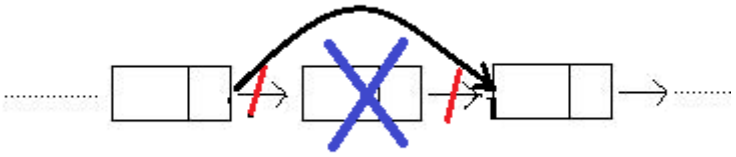


Figura 4: Ştergere nod p diferit de *prim* şi *ultim*.

```
{post:  se şterge din listă nodul indicat de p, e:  TElement este elementul
şters}
{elementul şters}
e ← [p].e
{dacă se şterge primul element al listei}
Daca p = lsi.prim atunci
  {se modifică prim}
  lsi.prim ← [p].urm
  {dacă noul prim e NIL, atunci lista e vidă}
  Daca  lsi.prim = NIL atunci
    lsi.ultim ← NIL
  SfDaca
altfel
  {se parcurge pâna la nodul p}
  q ← lsi.prim
  {sigur p este în listă, prin precondiţie}
  CatTimp [q].urm ≠ p executa
    q ← [q].urm
  SfCatTimp
  {q este nodul care precede p }
  {dacă se şterge ultimul element al listei }
  Daca p = lsi.ultim atunci
    lsi.ultim ← q
    {lista nu poate fi vidă, s-a tratat cazul la început}
  altfel
    {se şterge nodul p }
    [q].urm ← [p].urm
  SfDaca
SfDaca
```

```
  {dealocăm spaţiul de memorare pentru p }
  dealoca(p)
SfSubalgoritm
```

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

Iterator pe un container reprezentat folosind o LSI

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

NodLSI
e: TElement *//infomaţia utilă nodului*
urm: ↑ NodLSI *//adresa la care e memorat următorul nod*

Container
prim: ↑ NodLSI *//adresa primului nod din listă*

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: ↑ NodLSI *//adresa unui nod curent al* LSI

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

```
  valid ← i.curent ≠ NIL
SfFunctia

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

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

În directorul asociat Cursului 4 găsiţi implementarea parţială, în limbajul C++, a containerului **Colecţie** (reprezentarea este sub forma unei LSI care memorează toate elementele colecţiei, folosind alocare dinamică pentru reprezentarea înlănţuirilor).

TEMĂ. Scrieţi în Pseudocod/implementaţi restul operaţiilor specifice pe LSI şi deduceţi complexităţile 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))
- ştergere după un nod dat (stergDupa(lsi, p, e))

Lista dublu înlănţuită (LDI)

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

NodLDI
  e: TElement *//infomaţia utilă nodului*
  urm: ↑ NodLDI *//adresa la care e memorat următorul nod*
  prec: ↑ NodLDI *//adresa la care e memorat nodul precedent*

LDI
  prim: ↑ NodLDI *//adresa primului nod din listă*
  ultim: ↑ NodLDI *// adresa ultimului nod din listă*

Pentru operaţiile de adăugare, vom folosi o funcţie auxiliară care creează un nod având o anumită informţie utilă.

```
Functia creeazaNodLDI(ldi, e)
  {pre: ldi: LDI,e: TElement}
```

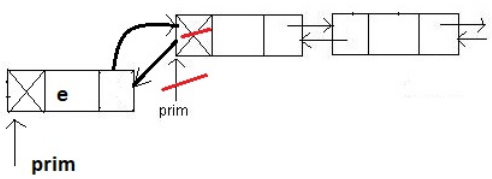
```
  {post: se returnează un ↑ NodLDI conţinând e ca informaţie utilă}
  {se alocă un spaţiu de memorare pentru un NodLDI }
  {p: ↑ NodLDI}
  aloca(p)
  [p].e ← e
  [p].urm ← NIL
  [p].prec ← NIL
  {rezultatul returnat de funcţie}
  creeazaNodLDI ← p
SfFunctia
```

- Complexitate: $\theta(1)$)

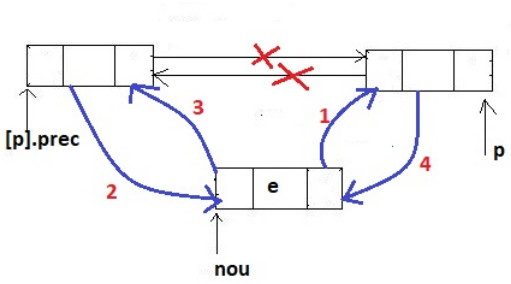
Subalgoritmul pentru adăugarea unui element înaintea unui nod din listă (indicat prin adresa sa - pointer). Sunt două cazuri care trebuie tratate

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

Figura 5 ilustrează adăugarea la început si adăugarea înaintea uni nod p diferit de *prim*.



(a) Adăugare element la început.



(b) Adăugare element 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 precondiţie}
    [ldi.prim].prec ← nou
    {se actualizează prim}
    ldi.prim ← nou
  altfel
    {se adaugă între [p].prec şi p}
    [nou].urm ← p
```

```
      [[p].prec].urm ← nou
      [nou].prec ← [p].prec
      [p].prec← nou
    SfDaca
  SfSubalgoritm
```

- Complexitate: $\theta(1)$)

TEMĂ

- Scrieţi în Pseudocod/implementaţi restul operaţiilor specifice pe LDI şi deduceţi complexităţile acestora:
  - adăugare element la începutul şi sfârşitul listei
  - adăugare element după un nod dat
  - căutarea unui element dat în listă
  - ştergerea unui nod din listă
- Similar cu ce s-a prezentat pentru LSI, scrieţi în Pseudocod operaţiile pe iteratorul unui **Container** oarecare (de ex. Colecţie) reprezentat sub forma unei LDI, folosind alocare dinamică pentru repezentarea înlănţuirilor.

Lista simplu înlănţuită sortată/ordonată (LSIO)

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

- de exemplu, dacă $\mathcal{R} = \leq$, atunci elementele vor fi stocate în ordine crescătoare: x. 4 7 9 11
- ca implementare, relaţia va fi o funcţie (pointer spre funcţie 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** şi o structură pentru listă.

NodLSIO
e: TElement *//infomaţia utilă nodului*
urm: ↑ NodLSIO *//adresa la care e memorat următorul nod*

LSIO
prim: ↑ NodLSI *//adresa primului nod din listă*
$\mathcal{R}$: Relatie *//relaţia de ordine întrre elemente*

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

- notăm prin $a\mathcal{R}b$ faptul că a e în relaţia $\mathcal{R}$ 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 şi ultim, se trataa şi 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 relaţia de ordine între elemente}
  nou ← creeazaNod(lsio, e)
  {dacă se adaugă la începutul listei }
  Daca (lsio.prim = NIL) ∨ (e R [lsio.prim].e)  atunci
    [nou].urm← lsio.prim
    lsio.prim ← nou
  altfel
    {se parcurge pâna la nodul q după care trebuie adăugat nou}
    {dacă lista e 1 2 5, relaţia e ≤ şi vrem să adăugăm 3, ne oprim pe nodul
    q cu informaţia 2}
    q ← lsio.prim
    CatTimp ([q].urm ≠ NIL) ∧ ¬ (e R [[q].urm].e) executa
      q ← [q].urm
    SfCatTimp
    {q este nodul după care se adaugă nou }
    [nou].urm ← [q].urm
    [q].urm ← 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

Operaţia de ştergere 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. Analizaţi cazul listei dublu înlănţuite sortate/ordonate - similar cu ce s-a discutat în Secţiunea anterioară.