TAD Lista (LIST)

Observaţii:

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

Tipul Abstract de Date LISTA:

domeniu:

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

operaţii:

- 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)
  $pre:$ $l \in L, p \in TPozitie, p$ poziție validă
  $post:$ $urmator = q \in TPozitie$,
  $q = \begin{cases} \text{poziția următoare poziției } p \text{ din lista } l, \\ \qquad \text{dacă } p \text{ nu e poziția ultimului element din lista } l \\ \perp, \quad \text{dacă } p \text{ e poziția ultimului element din lista } l \end{cases}$

  @ aruncă excepție dacă p nu e validă

- anterior(l, p)
  $pre:$ $l \in L, p \in TPozitie, p$ poziție validă
  $post:$ $anterior = q \in TPozitie,$
  $$q = \begin{cases} \text{poziția precedentă poziției } p \text{ din lista } l, \\ \qquad \text{dacă } p \text{ nu e poziția primului element din lista } l \\ \perp, \qquad \text{dacă } p \text{ e poziția primului element din lista } l \end{cases}$$
  @ aruncă excepție dacă p nu e validă

- element(l, p, e)
  $pre:$ $l \in L, p \in TPozitie,$ valid(p)
  $post:$ $e \in TElement, \quad e =$ elementul de pe poziția p din l

  @ aruncă excepție dacă p nu e validă

- pozitie (l, e)
  $pre:$ $l \in L, e \in TElement,$
  $post:$ $pozitie = p \in TPozitie,$
  $$p = \begin{cases} \text{prima poziție a elementului } e \text{ din lista } l, & \text{dacă } e \in l \\ \perp, \qquad \text{dacă } e \notin l \end{cases}$$

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

  @ aruncă excepție dacă p nu e validă

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

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

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

  @ aruncă excepție dacă p nu e validă

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

  @ aruncă excepție dacă p nu e validă

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

  @ aruncă excepție dacă p nu e validă

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

- vida (l)

$$pre: \quad l \in L$$
$$post: \quad vida = \begin{cases} true, & \text{dacă } l = \Phi \\ false, & \text{dacă } l \neq \Phi \end{cases}$$

- dim(l)

$$pre: \quad l \in L$$
$$post: \quad dim = n \in Natural,$$
$$n = \text{numărul de elemente ale listei } l$$

- distruge(l)
{destructor}

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

- iterator(l, i)

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

Observaţii

- Operaţia **cauta** poate fi specificată mai general
  - returnează prima *poziţie* pe care apare un element în listă, dacă elementul e găsit în listă
  - returnează *poziţie invalidă* dacă elementul nu e găsit în listă
- Din perspectiva unei ierarhii de containere
  - **Lista** este o **Colecţie**
  - **Vector Dinamic** este o **Listă**
    - **Vectorul Dinamic** poate fi văzut ca o **Listă** reprezentată *secvenţial*
- Există anumite dezavantaje induse de folosirea unui parametru de tip $TPozitie$ în interfaţa listei:
  1. Tipurile de referinţe concrete folosite diferă în funcţie de reprezentarea listei.
  2. Interfaţa listei este destul de greoaie şi nesigură prin faptul că expune în exterior poziţiile (referinţele la locaţiile din listă).
     - acesta este motivul pentru care bibliotecile existente particularizează tipul $TPozitie$ expus în interfaţa containerului Listă (după cum se va vedea în continuare)

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

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

- dacă se doreşte reprezentare *secvenţială*, se va folosi **vector**.

2. **Java - List**

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

Modalităţi de implementare a unei liste

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

Analiza complexităţii timp a celor mai importante operaţii ale containerului Lista în funcţie de implementarea acesteia

În Tabelul 1 vom considera, comparativ

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

Notăm cu n numărul de elemente din listă. Observăm faptul că reprezentarea dublu înlănţuită este cea mai eficientă ca şi timp, dar ocupă spaţiu de memorare suplimentar pentru legături (pentru a reduce spaţiul 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 specificaţii ale containerului **Lista** descrise anterior(STL/Java).

Lista - cu poziţie indice (indexată)

- corespunde modului în care este specificată lista în Java.
- *poziţia* 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 şi ştergeri la orice poziţie (poziţia unui element reprezintă indicele acestuia în cadrul listei).

Operaţie	Reprezentare secvenţială	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: Complexităţi timp ale operaţiilor.

- O poziţie i în cadrul listei l este *validă* dacă $1 \leq i \leq lungime(l)$.
- Se simplifică interfaţa
  - interfaţa este aceeaşi cu a unui **Vector Dinamic**

Specificaţia Listei indexate este dată mai jos **domeniu:**

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

operaţii:

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

  @ aruncă excepţie dacă i nu e valid
- sterge (l, i, e)
  $pre:$ $l \in L, l = (e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \cdots, e_n), i \in Intreg, i$ poziţie validă
  $post:$ $e \in TElement, e =$ elementul de pe poziţia i din l
  $l' = (e_1, \ldots, e_{i-1}, e_{i+1}, \cdots, e_n)$
  (pozitie$(l', e) = i$)

  @ aruncă excepţie dacă i nu e valid

- cauta (l, e)

  $pre:$ $l \in L, e \in TElement$

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

- element (l, i, e)

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

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

  @ aruncă excepție dacă i nu e valid

- modifica (l, i, e)

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

  $post:$ elementul de pe poziția i din $l' = e$

  @ aruncă excepție dacă i nu e valid

- vida (l)

  $pre:$ $l \in L$

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

- dim (l)

  $pre:$ $l \in L$

  $post:$ $dim = n \in Intreg$,
  $n =$ numărul de elemente din lista l

- iterator(l, i)

  $pre:$ $l \in L$

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

- distruge(l)

  $pre:$ $l \in L$

  $post:$ l 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, operaţia **element**.
Reprezentarea listei este

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

Lista
  prim: ↑ Nod*//adresa primului nod din listă*

```
Subalgoritm element(l, i, e)
  {pre:  l:  Lista, i:Intreg, 1 ≤ i ≤ lungime(l), e:TElement }
  {post:  e este al i-lea element al listei }
  {se parcurge până la al i-lea element }
  p ← l.prim
  {se parcurg i-1 legături }
  Pentru i = 1, i − 1 executa
    p ← [p].urm
```

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

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

Să considerăm sublgoritmul **tiparire** care tipăreşte 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)
    @tipăreşte 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)
    @tipăreşte e
  SfPentru
SfSubalgoritm
```

Lista - cu poziţie iterator

- corespunde modului în care este specificată lista în STL.
- *poziţia* este dată de un *iterator* pe listă $\Rightarrow TPozitie = Iterator$.
- se simplifică interfaţa
  - operaţiile *următor*, *anterior*, *valid* şi *element* sunt operaţiile pe *iterator*

Enumerăm, mai jos, operaţiile din interfaţa Listei în care accesul e pe baza unei poziţii date de un iterator, fără a mai da specificaţia completă a operaţiilor (specificaţiile sunt cele indicate la containerul generic **Lista**, dar cu $TPozitie = IteratorLista$).

Operaţii din interfaţă:

- 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 poziţie iterator, folosind o LDI alocată dinamic. Descriem mai jos, în Pseudocod, operaţia **adaugaDupa**.
Reprezentarea listei şi a iteratorului pe listă sunt date mai jos

Nod
e: TElement *//infomaţia 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*//referinţă către listă*
curent:↑ Nod*//adresa nodului curent din listă*

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

```
Functia creeazaNod(l, e)
  {pre:  l:  Lista,e:  TElement}
  {post:  se returnezã un ↑ Nod conţinând e ca informaţie utilă}
  {se alocă un spaţiu de memorare pentru un Nod }
  {p:  ↑ Nod}
  aloca(p)
  [p].e ← e
  [p].urm ← NIL
  [p].prec ← NIL
  {rezultatul returnat de funcţie}
  creeazaNod ← p
SfFunctia
```

- Complexitate: $\theta(1)$)

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

```
    {dacă p este ultimul nod al listei }
    Daca p = l.ultim atunci
      {p este diferit de NIL, din precondiţie }
      [l.ultim].urm ← nou
      [nou].prec ← l.ultim
      {se actualizează ultim}
      l.ultim ← nou
    altfel
      {se adaugă între p şi [p].urm}
      [nou].urm ← [p].urm
      [[p].urm].prec ← nou
      [p].urm ← nou
      [nou].prec← p
    SfDaca
  SfSubalgoritm
```

- Complexitate: $\theta(1)$)

În directorul TAD Lista (pe pagina cursului, curs 5) găsiţi implementarea parţială, în limbajul C++, a containerului **Lista** cu poziţie iterator (reprezentarea este sub forma unei LDI, folosind alocare dinamică pentru reprezentarea înlănţuirilor).