PROGRAMARE ȘI STRUCTURI DE DATE

CURS 11

Lect. dr. Oneț-Marian Zsuzsanna

Facultatea de Matematică și Informatică UBB
în colaborare cu NTT Data

În cursul 9 și 10

- Vector dinamic
- Ansamblu
- Lista simplu înlănțuită

Cuprins

Lista înlănțuită

- Lista înlănțuită este o structură de date alcătuită din *noduri*, fiecare nod conține câte un element și adresa nodului următor.
- Reținem din listă adresa primului nod.
- Aceste noduri care compun lista înlănțuită nu trebuie să fie în zone consecutive de memorie, ele pot fi oriunde în memorie.

Lista simplu înlănțuită - reprezentare

- Dacă vrem să implementăm o listă simplu înlănțuită, avem nevoie de 2 structuri:

Nod:
elem: TElem
urm: ↑ Nod

ListaSimpluInlantuita:
prim: ↑ Nod

LSI - ștergere de la început I

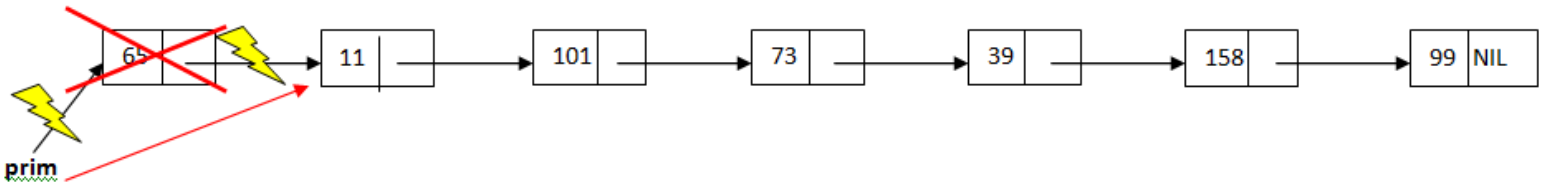
- Să vedem cum putem șterge primul element dintr-o listă simplu înlănțiută.

LSI - ștergere de la început I

- Să vedem cum putem șterge primul element dintr-o listă simplu înlănțiută.
- Dacă vrem să ștergem primul element pur și simplu modificăm valoarea *prim*-ului, să fie nodul următor.
- Evident, trebuie să verificăm să nu avem listă vidă (în acest caz nu putem șterge nimic).
- Presupunem reprezentare fără câmpul lungime.

LSI - ștergere de la început II

- Ștergem primul element din listă (elementul 65).
- Modificăm valoarea *prim*-ului, să arate spre al 2-lea nod.
- Cu roșu este marcată legătura nouă și fulgerul galben arată legăturile șterse.



LSI - ștergere de la început III

```
subalgoritm stergeInceput() este:
    dacă this.prim != NIL atunci
        this.prim = [this.prim].urm
    sf_dacă
sf_subalgoritm
```

- Complexitate:

LSI - ștergere de la început III

```
subalgoritm stergeInceput() este:
    dacă this.prim != NIL atunci
        this.prim = [this.prim].urm
    sf_dacă
sf_subalgoritm
```

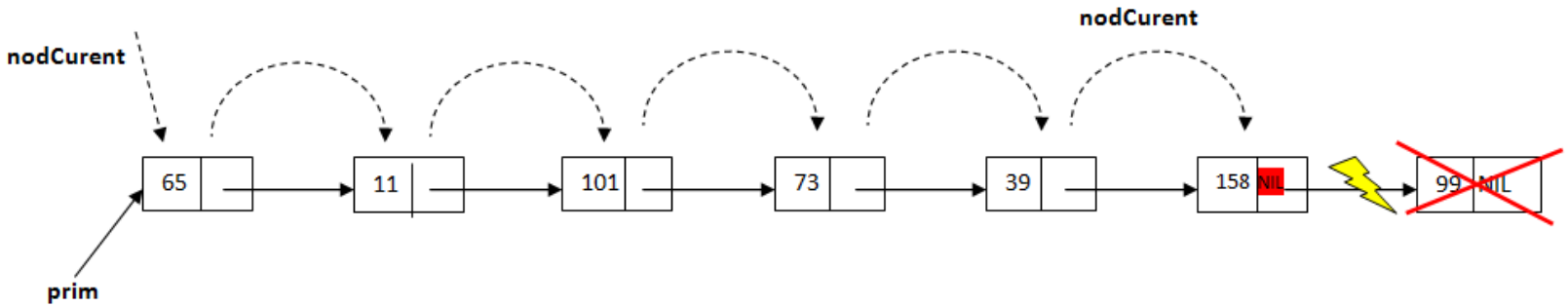
- Complexitate: $\Theta(1)$

LSI - ștergere de la sfârșit I

- Pentru a șterge ultimul element, trebuie să parcurgem lista, până ajungem la penultimul element. Legătura *urm* al penultimul element este spre ultimul element, cel pe care vrem să ștergem. Vom seta această legătură la *NIL* pentru a rupe legătura cu ultimul element.
- Ultimul element este cel care are *urm* egal cu *NIL*, penultimul element este cel care are *urm*-ul *urm*-ului egal cu *NIL*.
- Trebuie să verificăm ca lista să nu fie vidă (nu avem ce șterge) și să nu aibă un singur element (nu avem penultim element). Dacă lista are un singur element, acesta va fi șters.
- Presupunem reprezentare fără câmpul lungime

LSI - ștergere de la sfârșit II

- Ștergem ultimul element din listă (elementul 99)
- Cu *nodCurent* ne deplasăm până la penultimul element (liniile punctate)
- Cu fulger galben este marcată legătura care este ștearsă.



LSI - ștergere de la sfârșit III

```
subalgoritm stergeSfarsit () este:
    dacă this.prim != NIL atunci
        dacă [this.prim].urm == NIL atunci //avem un singur element
            this.prim = NIL
        altfel
            nodCurent: ↑ Nod //variabila cu care parcurgem lista
            nodCurent = this.prim
            câttimp [[nodCurent].urm].urm != NIL execută
                nodCurent = [nodCurent].urm
            sf_câttimp //acum [[nodCurent].urm].urm este NIL
            [nodCurent].urm = NIL
        sf_dacă
    sf_dacă
sf_subalgoritm
```

- Complexitate:

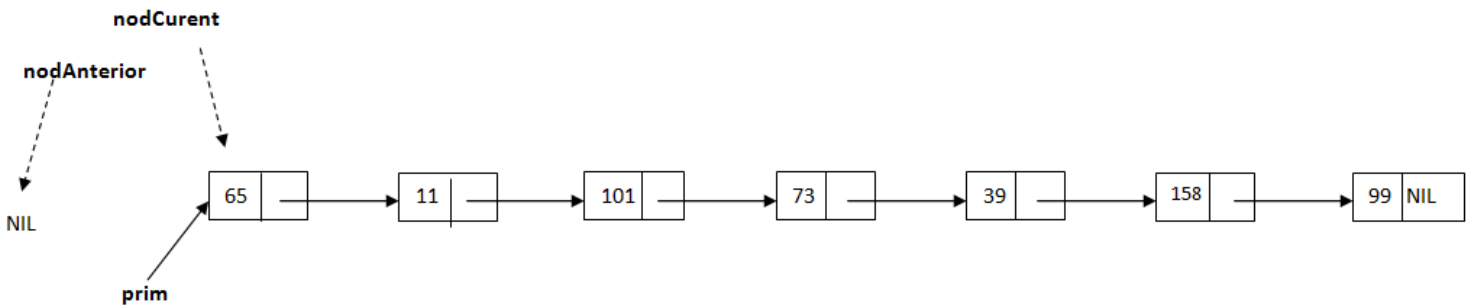
LSI - ștergere de la sfârșit III

```
subalgoritm stergeSfarsit () este:
    dacă this.prim != NIL atunci
        dacă [this.prim].urm == NIL atunci //avem un singur element
            this.prim = NIL
        altfel
            nodCurent: ↑ Nod //variabila cu care parcurgem lista
            nodCurent = this.prim
            câttimp [[nodCurent].urm].urm != NIL execută
                nodCurent = [nodCurent].urm
            sf_câttimp //acum [[nodCurent].urm].urm este NIL
            [nodCurent].urm = NIL
        sf_dacă
    sf_dacă
sf_subalgoritm
```

- Complexitate: $\Theta(n)$ - unde n este numărul de elemente din listă

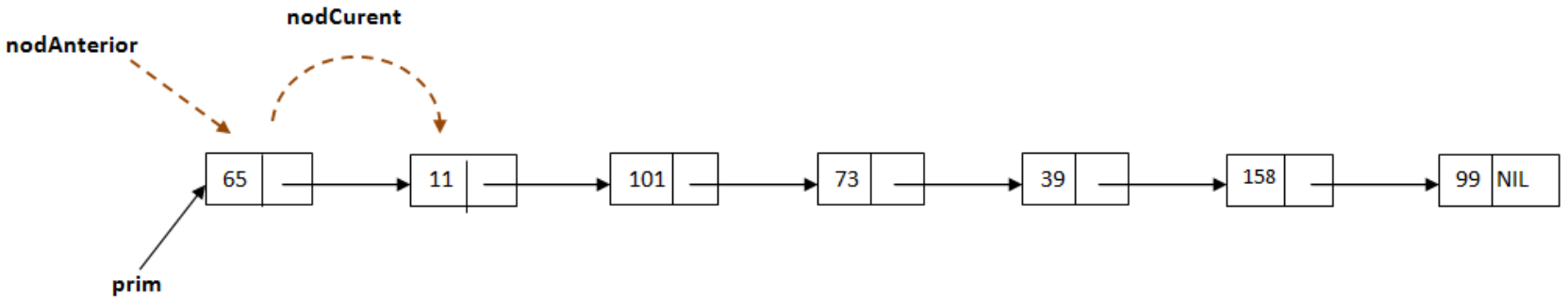
LSI - ștergere de la sfârșit IV

- Există o altă variantă de a parcurge lista, pentru a evita să folosim *[[nodCurent].urm].urm*.
- Putem folosi pentru parcurgere 2 variabile de tip nod (de ex. *nodCurent* și *nodAnterior*), care vor reține adresele pentru 2 noduri consecutive.
  - *nodCurent* este primul nod, *nodAnterior* este *NIL*
  - *nodCurent* este al 2-lea nod, *nodAnterior* este primul nod
  - *nodCurent* este al 3-lea nod, *nodAnterior* este al 2-lea nod
  - ...
  - *nodCurent* este ultimul nod, *nodAnterior* este penultimul nod

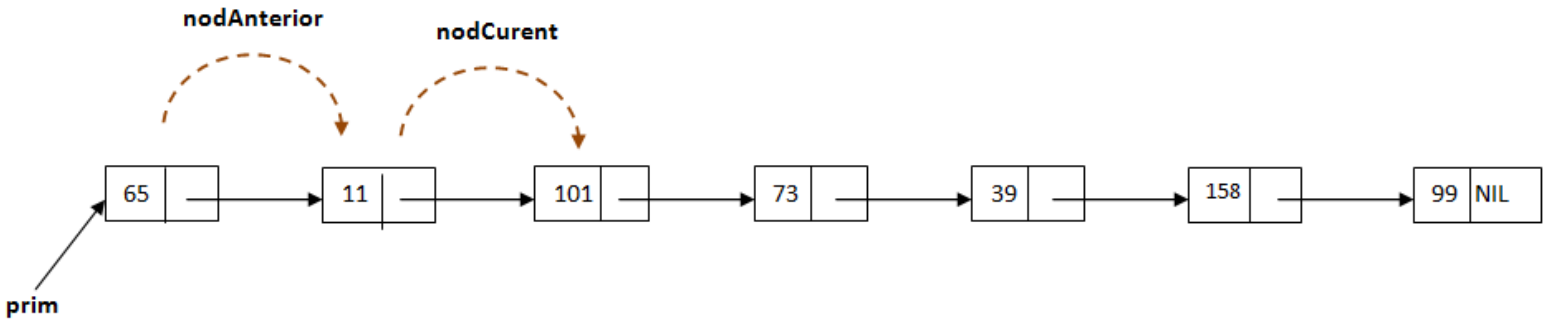


LSI - ștergere de la sfârșit V

- După o iterație (din parcurgere):

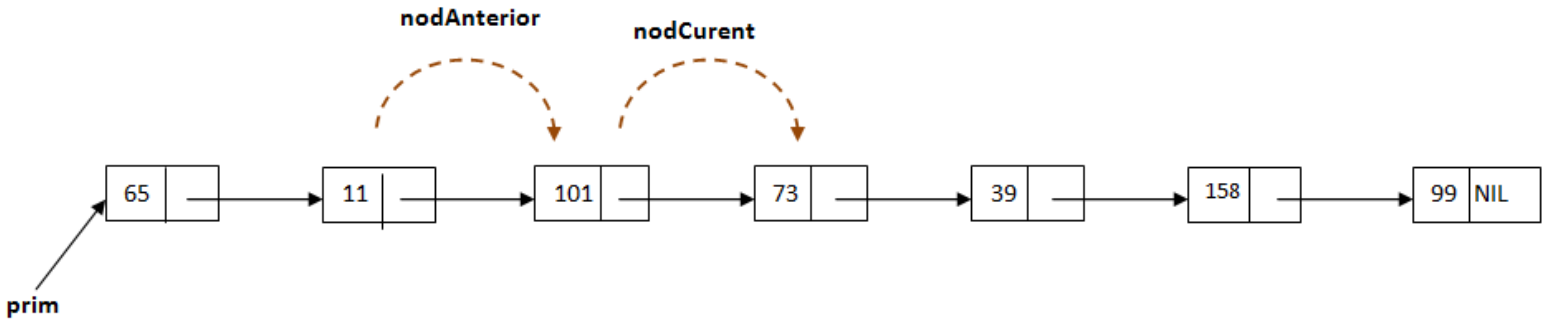


- După a 2-a iterație (din parcurgere):

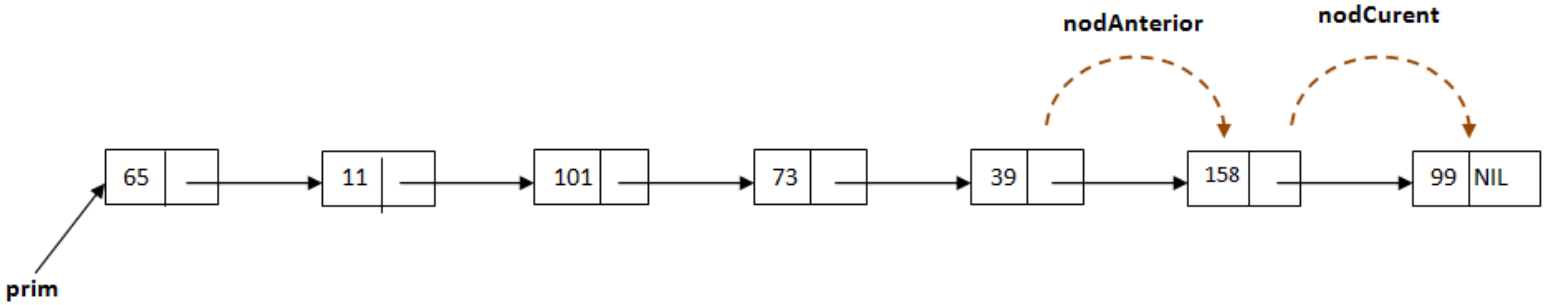


LSI - ștergere de la sfârșit VI

- După a 3-a iterație (din parcurgere):



- După ultima iterație (din parcurgere):



LSI - ștergere de la sfârșit VII

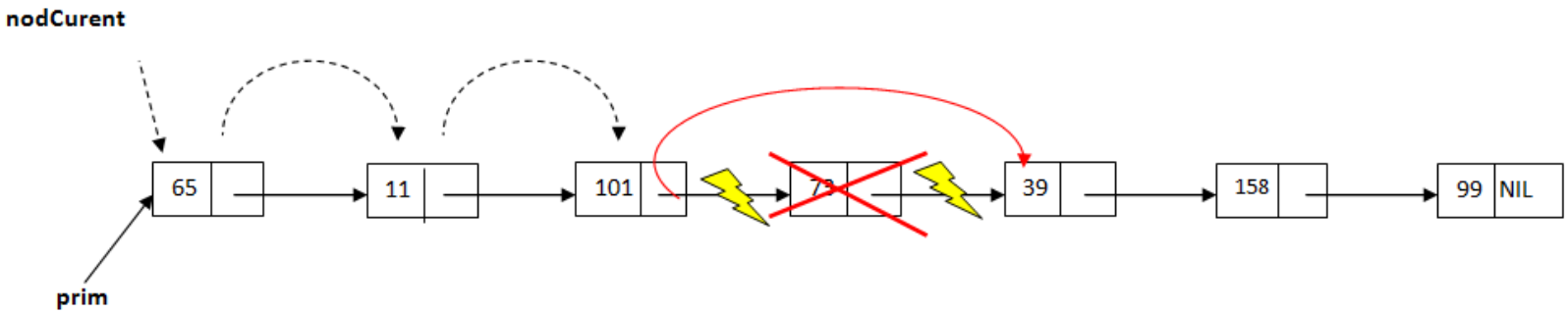
```
subalgoritm stergeSfarsit () este:
    dacă this.prim != NIL atunci
        dacă [this.prim].urm = NIL atunci //avem un singur element
            this.prim = NIL
        altfel
            nodCurent: ↑ Nod
            nodCurent = this.prim
            nodAnterior: ↑ Nod
            nodAnterior = NIL
            câttimp [nodCurent].urm != NIL execută
                nodAnterior = nodCurent
                nodCurent = [nodCurent].urm
            sf_câttimp //acum nodCurent trebuie șters
            [nodAnterior].urm = NIL
        sf_dacă
    sf_dacă
sf_subalgoritm
```

LSI - ștergere de pe poziție I

- Pentru a șterge un element de pe o poziție, trebuie să parcurgem lista până când găsim nodul de *după* care ștergem.
- Dacă am găsit nodul de după care ștergem, pur și simplu setăm câmpul *urm* al acestui nod, la *urm*-ul nodului următor (care este nodul șters).
- Trebuie să verificăm dacă ștergem elementul de pe prima poziție (este caz special, aici nu există nod de după care ștergem) și dacă lista e vidă (nu avem ce șterge).

LSI - ștergere de pe poziție II

- Să ștergem elementul de pe poziția 3 (numărul 73)
- Folosim un *nodCurent* să ne deplasăm pe poziția 2.
- Setăm legătura nodului de pe poziția 2 la nodul de pe poziția 4.



LSI - ștergere de pe poziție III

```
subalgoritm stergePozitie(poz:întreg) este:
    dacă this.prim != NIL atunci
        dacă poz == 0 atunci
            this.prim = [this.prim].urm
        altfel
            nodCurent: ↑ Nod
            nodCurent = this.prim
            pozCurent: Întreg
            pozCurent = 0
            //continuăm pe pagina următoare
```

LSI - ștergere de pe poziție IV

```
        cât timp pozCurent < poz - 1 ȘI nodCurent != NIL exec.
            nodCurent = [nodCurent].urm
            pozCurent = pozCurent + 1
        sf_cât timp
        dacă pozCurent==poz-1 ȘI nodCurent!=NIL ȘI [nodCurent].urm!=NIL at.
            [nodCurent].urm = [[nodCurent].urm].urm
        altfel
            @aruncă excepție, poziție invalidă
        sf_dacă
    sf_dacă
  sf_dacă
sf_subalgoritm
```

- Complexitate:

LSI - ștergere de pe poziție IV

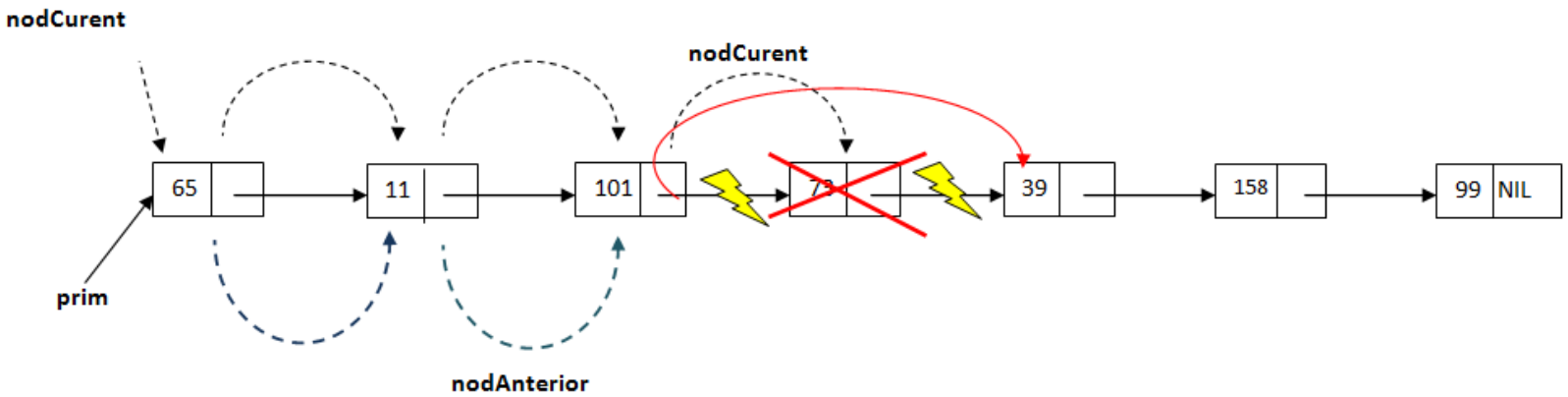
```
    cât timp pozCurent < poz - 1 ȘI nodCurent != NIL exec.
        nodCurent = [nodCurent].urm
        pozCurent = pozCurent + 1
    sf_cât timp
    dacă pozCurent==poz-1 ȘI nodCurent!=NIL ȘI [nodCurent].urm!=NIL at.
        [nodCurent].urm = [[nodCurent].urm].urm
    altfel
        @aruncă excepție, poziție invalidă
    sf_dacă
  sf_dacă
 sf_dacă
sf_subalgoritm
```

- Complexitate: $O(n)$ - unde n este numărul de elemente din listă

LSI - ștergere de pe poziție V

- Există și la acest algoritm varianta în care folosim 2 variabile de tip nod: *nodCurent* și *nodAnterior*. Ne deplasăm până când *nodCurent* este nodul de șters, iar *nodAnterior* este nodul de dinaintea lui.

- În acest caz, pentru a șterge elementul *nodCurent*, vom seta *urm*-ul lui *nodAnterior* la *urm*-ul lui *nodCurent*.



LSI - ștergere de pe poziție VI

```
subalgoritm stergePozitie(poz:întreg) este:
//poz - poziția de unde ștergem. Presupunem că poz >= 0
   dacă this.prim != NIL atunci
      dacă poz == 0 atunci
         this.prim = [this.prim].urm
      altfel
         nodCurent: ↑ Nod
         nodCurent = this.prim
         nodAnterior: ↑ Nod
         nodAnterior = NIL
         pozCurent: Întreg
         pozCurent = 0
         //continuăm pe pagina următoare
```

LSI - ștergere de pe poziție VII

```
        câttimp pozCurent < poz ȘI nodCurent != NIL exec.
            nodAnterior = nodCurent
            nodCurent = [nodCurent].urm
            pozCurent = pozCurent + 1
        sf_câttimp
        dacă pozCurent == poz ȘI nodCurent != NIL at.
            [nodAnterior].urm = [nodCurent].urm
        altfel
            @aruncă excepție, poziție invalidă
        sf_dacă
    sf_dacă
sf_dacă
sf_subalgoritm
```

- Complexitate:

LSI - ștergere de pe poziție VII

```
        câttimp pozCurent < poz ȘI nodCurent != NIL exec.
            nodAnterior = nodCurent
            nodCurent = [nodCurent].urm
            pozCurent = pozCurent + 1
        sf_câttimp
        dacă pozCurent == poz ȘI nodCurent != NIL at.
            [nodAnterior].urm = [nodCurent].urm
        altfel
            @aruncă excepție, poziție invalidă
        sf_dacă
    sf_dacă
sf_dacă
sf_subalgoritm
```

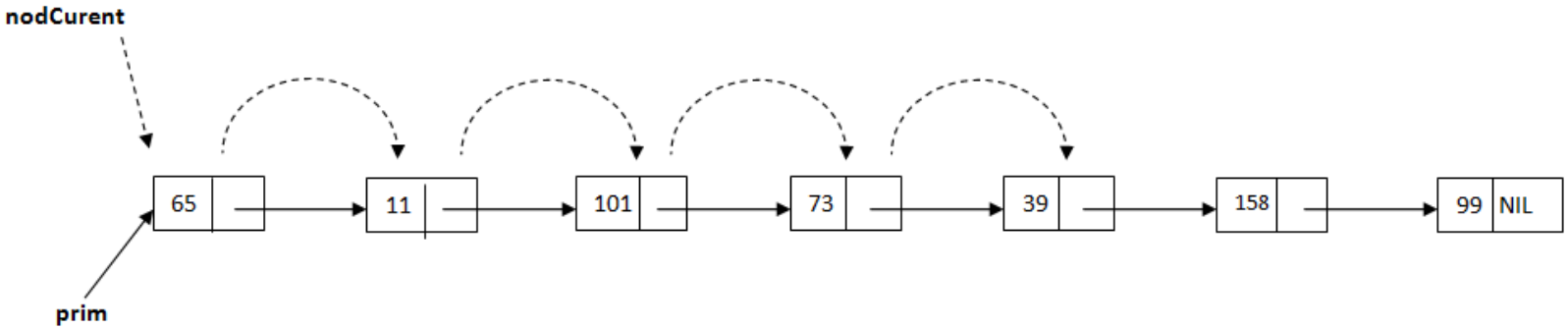
- Complexitate: $O(n)$ - unde n este numărul de elemente din listă

LSI - Returnarea unui element de pe o poziție

- Pentru a returna un element de pe o poziție, trebuie să parcurgem lista până ajungem la poziția respectivă.
- Asemenea parcurgeri am făcut și la adăugare pe poziție, și la ștergere de pe poziție, numai atunci ne-am deplasat doar până la nodul de dinaintea poziției cerute. Acum ne vom deplasa până la poziția cerută.
- Evident, trebuie să verificăm să existe poziția (dacă nu există poziția, nodul folosit pentru parcurgere - *nodCurent* - va deveni NIL).

LSI - Returnarea unui element de pe o poziție II

- Vrem elementul de pe poziția 4.
- Folosim *nodCurent* care inițial e pe poziția 0, și ne deplasăm până ajungem la poziția 4.



LSI - Returnarea unui element de pe o poziție III

funcție element (poz:întreg) **este:**
  dacă poz >= 0 **atunci**
    nodCurent: ↑ Nod
    nodCurent = this.prim
    pozCurent: Întreg
    pozCurent = 0
    câttimp pozCurent < poz **ȘI** nodCurent != NIL **execută**
      pozCurent = pozCurent + 1
      nodCurent = [nodCurent].urm
    sf_câttimp
    //continuăm pe pagina următoare

LSI - Returnarea unui element de pe o poziție IV

```
    dacă nodCurent != NIL atunci
        returnează [nodCurent].elem
    sf_dacă
  sf_dacă
  @aruncăm excepție sau returnăm NIL, poziția nu există
sf_funcție
```

- Complexitate:

LSI - Returnarea unui element de pe o poziție IV

```
    dacă nodCurent != NIL atunci
        returnează [nodCurent].elem
    sf_dacă
  sf_dacă
  @aruncăm excepție sau returnăm NIL, poziția nu există
sf_funcție
```

- Complexitate: $O(n)$

LSI - Iterator

- Cum putem defini un iterator pentru o listă simplu înlănțuită?
- Un iterator este folosit pentru a parcurge o structură de date, element - cu - element. Orice iterator trebuie să rețină un *element curent* (care diferă de la o structură la alta)
- Cum putem reține un element curent pentru o listă simplu înlănțuită?

LST - Iterator II

- La VectorDinamic am discutat că elementul curent din iterator este un index, o poziție din cadrul vectorului.
- La listă înlănțuită nu e o idee bună să reținem un index. O operație al iteratorului este *element*, operația care returnează elementul curent. Dacă iteratorul reține poziția elementului curent (cum facem la Vector Dinamic), pentru a returna elementul curent trebuie să parcurgem toată lista de la început până la poziția respectivă.
- La o listă înlănțuită, elementul curent din iterator, este un *nod*.

LST - Iterator - Reprezentare

- Un Iterator pentru o listă simplu înlănțuită poate fi reprezentată în modul următor (structura Nod este cea folosită și la listă) :

IteratorLSI:
  list: LSI
  curent: ↑ Nod

LSI - Iterator - creează

- Operația *creează* trebuie să inițializeze câmpurile pentru iterator.

LSI - Iterator - creează

- Operația *creează* trebuie să inițializeze câmpurile pentru iterator.

subalgoritm creeaza(lst) **este:**
//lst - este lista pentru care vrem să creăm iteratorul.
    this.list = lst
    this.curent = lst.prim
sf_subalgoritm

- Complexitate:

LSI - Iterator - creează

- Operația *creează* trebuie să inițializeze câmpurile pentru iterator.

subalgoritm creeaza(lst) **este:**
//lst - este lista pentru care vrem să creăm iteratorul.
    this.list = lst
    this.curent = lst.prim
sf_subalgoritm

- Complexitate: $\Theta(1)$

LSI - Iterator - element

- Cum returnăm elementul curent din iterator?

LSI - Iterator - element

- Cum returnăm elementul curent din iterator?

funcție element() **este**:
    returnează [this.curent].elem
sf_funcție

- Complexitate:

LSI - Iterator - element

- Cum returnăm elementul curent din iterator?

funcție element() **este**:
    returnează [this.curent].elem
sf_funcție

- Complexitate: $\Theta(1)$

LSI - Iterator - următor

- Cum trecem la elementul următor?

LSI - Iterator - următor

- Cum trecem la elementul următor?

subalgoritm următor() **este**:
  this.curent = [this.curent].urm
sf_subalgoritm

- Complexitate:

LSI - Iterator - următor

- Cum trecem la elementul următor?

subalgoritm următor() **este**:
  this.curent = [this.curent].urm
sf_subalgoritm

- Complexitate: $\Theta(1)$

LSI - Iterator - valid

- Cum verificăm dacă elementul curent din iterator este valid?

LSI - Iterator - valid

- Cum verificăm dacă elementul curent din iterator este valid?

```
funcție valid() este:
    dacă this.curent == NIL atunci
        returnează false
    altfel
        returnează true
sf _funcție
```

- Complexitate:

LSI - Iterator - valid

- Cum verificăm dacă elementul curent din iterator este valid?

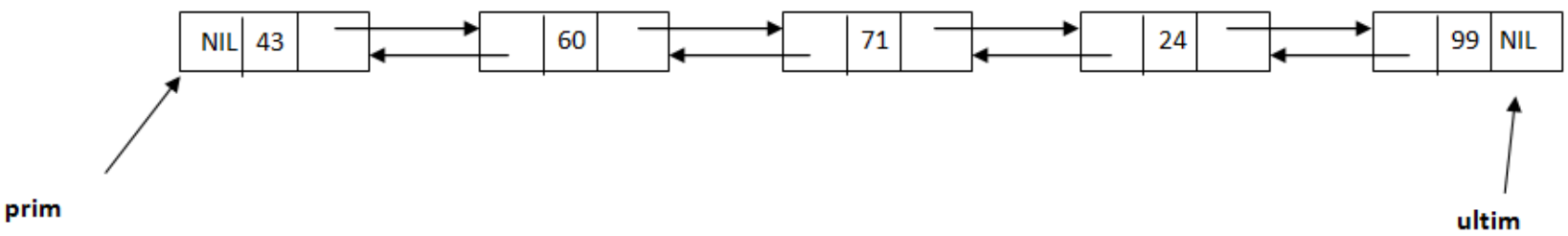
```
funcție valid() este:
    dacă this.curent == NIL atunci
        returnează false
    altfel
        returnează true
sf _funcție
```

- Complexitate: $\Theta(1)$

Lista dublu înlănțuită

- Până acum am discutat de lista simplu înlănțuită, în care fiecare nod reține adresa nodului următor.
- Există și listă *dublu înlănțuită* în care fiecare nod reține adresa nodului următor și adresa nodului anterior.
- Lista simplu înlănțuită poate fi parcursă într-o singură direcție (înainte), lista dublu înlănțuită poate fi parcursă în ambele direcții (pot să parcurg lista de la capăt spre început).

LDI - Exemplu



- Fiecare nod are legătură pentru nodul următor și anterior.
- Legătura pentru următor la ultimul nod este *NIL*
- Legătura pentru anteriorul la primul nod este *NIL*
- Nu reținem doar *primul* nod, ci și *ultimul*.

LDI - Reprezentare

- Avem nevoie de 2 structuri și la LDI: *Nod* și *ListaDubluInlantuita*

Nod:
elem: TElem
urm: ↑ Nod
ant: ↑ Nod

LDI:
prim: ↑ Nod
ultim: ↑ Nod

LDI - Operații

- Operațiile sunt similare cu operațiile de la lista simplu înlănțuită, trebuie doar să setăm mai multe legături și să nu uităm că reținem și *ultimul nod* (de exemplu când adaug sau șterg de la sfârșitul listei, se modifică valoarea ultimului nod).
- Acele operații care nu modifică structura listei (element, cauta, lungime, toate operațiile iteratorului) rămân la fel ca la lista simplu înlănțuită.
- Vom implementa câteva dintre operațiile pe lista dublu înlănțuită ca să vedem diferența.

LDI - Creează

- Operația creează trebuie să inițializeze câmpurile listei.

subalgoritm creeaza() **este:**
    this.prim = NIL
    this.ultim = NIL
sf_subalgoritm

- Complexitate:

LDI - Creează

- Operația creează trebuie să inițializeze câmpurile listei.

subalgoritm creeaza() **este:**
    this.prim = NIL
    this.ultim = NIL
sf_subalgoritm

- Complexitate: $\Theta(1)$

LDI - AdaugaÎnceput I

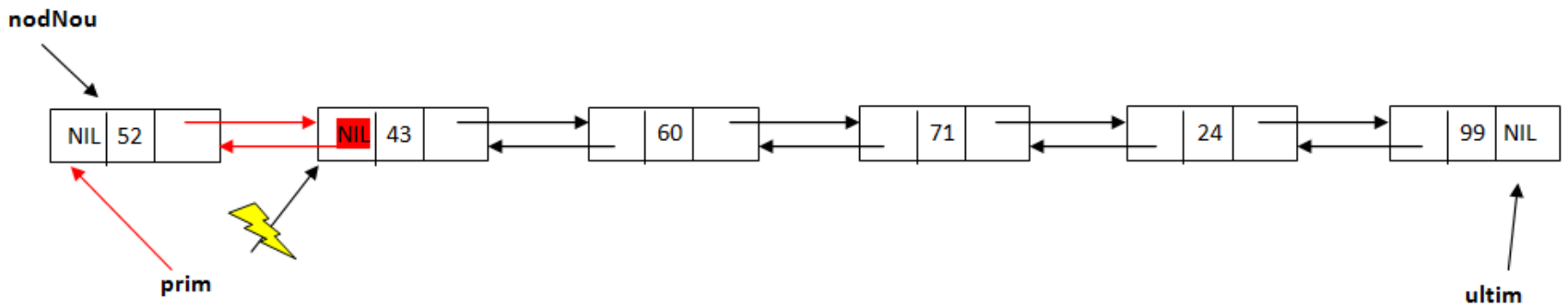
- Cum adăugăm un element la începutul listei?

LDI - AdaugaÎnceput I

- Cum adăugăm un element la începutul listei?
- Trebuie să creăm un nod nou, și să-l punem la începutul listei.
- Se va modifica *prim*-ul listei, dar dacă lista era vidă, atunci și *ultim*-ul (dacă lista este vidă, adăugând un element nou, acel element va fi și primul și ultimul element)
- Trebuie să setăm și legătura pentru *ant*, nu doar *urm*.

LDI - AdaugaÎnceput II

- Să adăugăm la începutul listei elementul 52.
- Vom crea un nod nou, în care punem elementul și care va deveni primul nod.
- Cu roșu sunt marcate legăturile noi, iar cu fulger galben legătura care va fi desfăcută.
- Comparați figura cu cel de la adăugare început la LSI.



LDI - Adauga Început III

```
subalgoritm adaugaInceput(el:TElem) este:
    nodNou: ↑ Nod //nodul nou
    [nodNou].urm = NIL
    [nodNou].ant = NIL
    [nodNou].elem = el
    dacă this.prim == NIL atunci
        this.prim = nodNou
        this.ultim = nodNou
    altfel
        //începem cu legăturile nodului nou
        [nodNou].urm = this.prim
        [this.prim].ant = nodNou
        this.prim = nodNou
    sf_dacă
sf_subalgoritm
```

- Complexitate:

LDI - Adauga Început III

```
subalgoritm adaugaInceput(el:TElem) este:
    nodNou: ↑ Nod //nodul nou
    [nodNou].urm = NIL
    [nodNou].ant = NIL
    [nodNou].elem = el
    dacă this.prim == NIL atunci
        this.prim = nodNou
        this.ultim = nodNou
    altfel
        //începem cu legăturile nodului nou
        [nodNou].urm = this.prim
        [this.prim].ant = nodNou
        this.prim = nodNou
    sf_dacă
sf_subalgoritm
```

- Complexitate: $\Theta(1)$

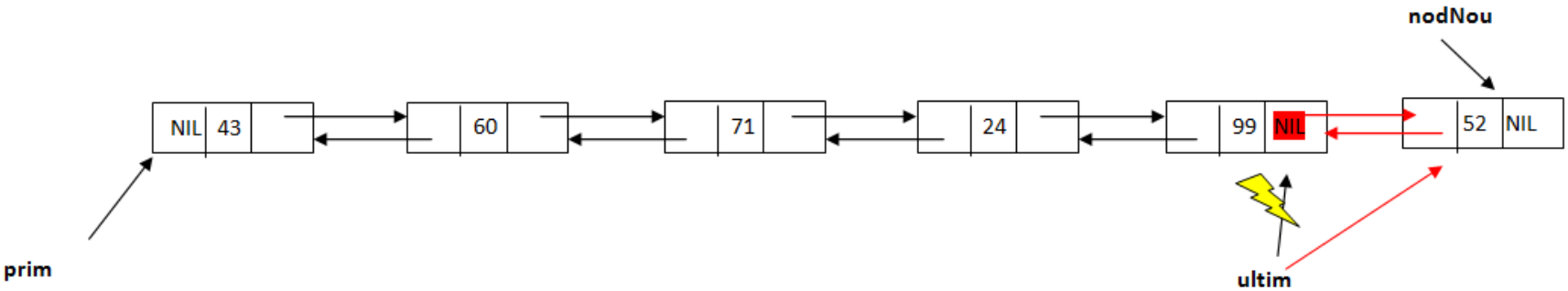
LDI - Adaugă Sfârșit

- Cum adăugăm un element la sfârșitul listei?

LDI - Adaugă Sfârșit

- Cum adăugăm un element la sfârșitul listei?
- La adăugare la sfârșit facem similar ca la adăugarea la început.

- Nu trebuie să parcurgem lista ca să ajungem la ultimul element (cum am făcut la lista simplu înlănțuită) pentru că avem ultimul nod reținut.

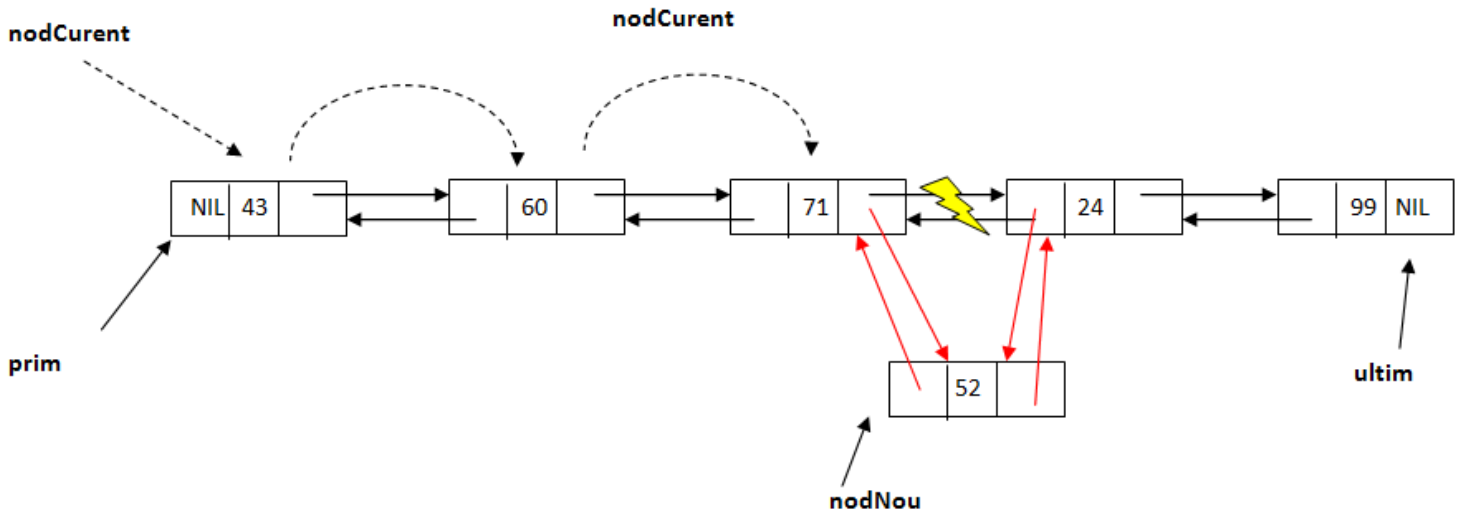


LDI - Adaugăre pe poziție I

- La adăugare pe poziție trebuie să parcurgem lista, până când ajungem la poziția unde vrem să adăugăm.
- Nu trebuie să ne oprim neapărat la nodul după care inserăm (cum am făcut la lista simplu înlănțuită), pentru că putem să ne deplasăm în ambele direcții. Dar, înainte de implementare trebuie să ne decidem dacă ne oprim la nodul *înainte* sau *după* care adăugăm. Noi vom implementa varianta când căutăm nodul după care adăugăm.
- Trebuie să verificăm dacă inserăm pe prima sau ultima poziție (în acest caz se modifică valoarea *prim* sau *ultim*)

LDI - Adăugare pe poziție II

- Să adăugăm elementul 52 pe poziția 3.



LDI - Adaugare pe poziție III

subalgoritm adaugaPozitie(el:TElem, poz: întreg) **este:**
//poz - poziția pe care adăugăm. Presupunem că poz >= 0
    nodNou: ↑ Nod
    [nodNou].urm = NIL
    [nodNou].ant = NIL
    [nodNou].elem = el
    dacă poz == 0 **atunci** *//adăugăm pe prima poziție*
        dacă this.prim == NIL **atunci** *//și lista e vidă*
            this.prim = nodNou
            this.ultim = nodNou
        altfel *//adaugam pe prima poz, dar lista nu e vida*
//continuăm pe pagina următoare

LDI - Adăugare pe poziție IV

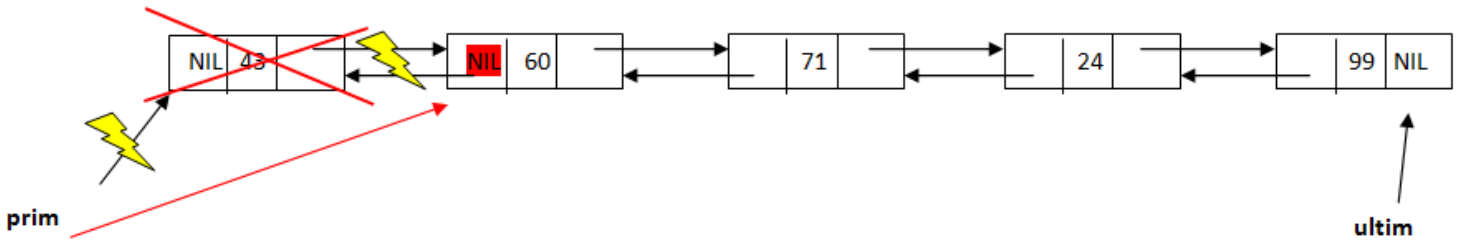
```
        [nodNou].urm = this.prim
        [this.prim].ant = nodNou
        this.prim = nodNou
    sf_dacă
  altfel
  //trebuie să căutăm nodul după care inserăm
    nodCurent: ↑ Nod
    nodCurent = this.prim
    pozCurent: Întreg
    pozCurent = 0
    cât timp pozCurent < poz - 1 ȘI nodCurent != NIL execută
      pozCurent = pozCurent + 1
      nodCurent = [nodCurent].urm
    sf_cât timp
//continuăm pe pagina următoare
```

LDI - Adăugare pe poziție IV

```
    //inserăm dupa nodCurent, dacă există
    dacă pozCurent == poz - 1 ȘI nodCurent ≠ NIL atunci
        dacă nodCurent == this.ultim atunci //adăugăm după ultimul
            [nodNou].ant = nodCurent
            [nodCurent].urm = nodNou
            this.ultim = nodNou
        altfel
            [nodNou].ant = nodCurent
            [nodNou].urm = [nodCurent].urm
            [[nodCurent].urm].ant = nodNou
            [nodCurent].urm = nodNou
        sf_dacă
    altfel
        @aruncă excepție poziție invalidă
    sf_dacă
  sf_dacă
sf_subalgoritm
```

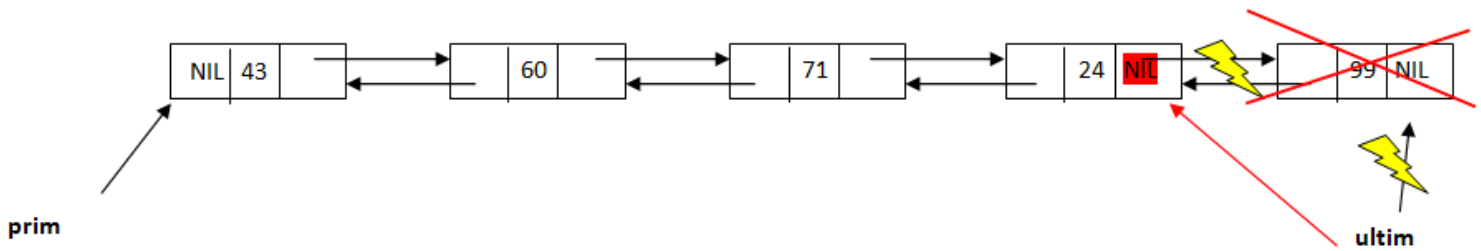
LDI - Ștergere început

- Când vrem să ștergem primul element, pur și simplu setăm *prim*-ul listei să arate spre nodul următor.
- Trebuie să verificăm ca lista să nu fie vidă (nu avem ce șterge) sau să nu aibă un singur element (în acest caz lista devine vidă și se modifică și *prim* și *ultim*).



LDI - Ștergere sfârșit I

- Când vrem să ștergem ultimul element, nu trebuie să parcurgem lista, folosim câmpul *ultim*.
- Setăm *ultim* la *ant*-ul ultimului nod.
- Trebuie să verificăm ca lista să nu fie vidă (nu avem ce șterge) sau să nu aibă un singur element (în acest caz lista devine vidă și se modifică și *prim* și *ultim*).



LDI - Ștergere sfârșit II

```
subalgoritm stergeSfarsit() este:
    dacă this.prim != NIL atunci
        dacă this.prim == this.ultim atunci //lista are un singur element
            this.prim = NIL
            this.ultim = NIL
        altfel
            this.ultim = [this.ultim].ant
            [this.ultim].urm = NIL
        sf_dacă
    sf_dacă
sf_subalgoritm
```

- Complexitate:

LDI - Ștergere sfârșit II

```
subalgoritm stergeSfarsit() este:
    dacă this.prim != NIL atunci
        dacă this.prim == this.ultim atunci //lista are un singur element
            this.prim = NIL
            this.ultim = NIL
        altfel
            this.ultim = [this.ultim].ant
            [this.ultim].urm = NIL
        sf_dacă
    sf_dacă
sf_subalgoritm
```

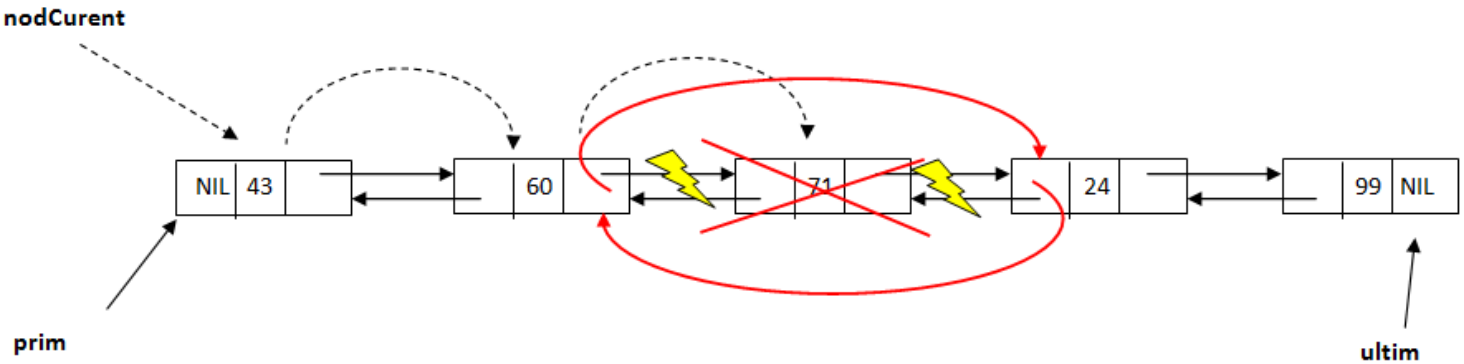
- Complexitate: $\Theta(1)$

LDI - Ștergere de pe poziție I

- Pentru a șterge un element de pe o poziție, trebuie să luăm un nod curent și să ne deplasăm cu acel nod până la poziția de unde ștergem.
- Putem să ne deplasăm până la elementul șters, nu trebuie să ne oprim înainte de poziție (ca la lista simplu înlănțuită).
- Pentru a șterge elementul trebuie să setăm 2 legături (următorul anteriorului și anteriorul următorului).
- Trebuie să verificăm dacă ștergem primul sau ultimul element (se modifică *prim* sau *ultim*).

LDI - Ștergere de pe poziție II

- Să ștergem elementul de pe poziția 2.



Comparație I

- Listele înlănțuite au aceleași operații ca Vectorul Dinamic.
- Să vedem complexitatea pentru fiecare operație, pentru fiecare reprezentare.
- Pentru fiecare operație este marcată cu verde structura de date care are complexitate minimă pentru operația respectivă.
- Notația:
  - VD - Vector Dinamic
  - LSI - Lista simplu înlănțuită
  - LDI - Lista dublu înlănțuită

Comparație II

Operație	VD	LSI	LDI
creează			

Comparație II

Operație	VD	LSI	LDI
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput			

Comparație II

Operație	**VD**	**LSI**	**LDI**
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit			

Comparație II

Operație	VD	LSI	LDI
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie			

Comparație II

Operație	**VD**	**LSI**	**LDI**
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput			

Comparație II

Operație	**VD**	**LSI**	**LDI**
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit			

Comparație II

Operație	**VD**	**LSI**	**LDI**
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie			

Comparație II

Operație	**VD**	**LSI**	**LDI**
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie	$O(n)$	$O(n)$	$O(n)$
element			

Comparație II

Operație	VD	LSI	LDI
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie	$O(n)$	$O(n)$	$O(n)$
element	$\Theta(1)$	$O(n)$	$O(n)$
cauta			

Comparație II

Operație	**VD**	**LSI**	**LDI**
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie	$O(n)$	$O(n)$	$O(n)$
element	$\Theta(1)$	$O(n)$	$O(n)$
cauta	$O(n)$	$O(n)$	$O(n)$
lungime			

Comparație II

Operație	VD	LSI	LDI
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie	$O(n)$	$O(n)$	$O(n)$
element	$\Theta(1)$	$O(n)$	$O(n)$
cauta	$O(n)$	$O(n)$	$O(n)$
lungime	$\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$
Iterator creează			

Comparație II

Operație	VD	LSI	LDI
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie	$O(n)$	$O(n)$	$O(n)$
element	$\Theta(1)$	$O(n)$	$O(n)$
cauta	$O(n)$	$O(n)$	$O(n)$
lungime	$\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$
Iterator creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator element			

Comparație II

Operație	VD	LSI	LDI
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie	$O(n)$	$O(n)$	$O(n)$
element	$\Theta(1)$	$O(n)$	$O(n)$
cauta	$O(n)$	$O(n)$	$O(n)$
lungime	$\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$
Iterator creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator element	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator următor			

Comparație II

Operație	VD	LSI	LDI
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie	$O(n)$	$O(n)$	$O(n)$
element	$\Theta(1)$	$O(n)$	$O(n)$
cauta	$O(n)$	$O(n)$	$O(n)$
lungime	$\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$
Iterator creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator element	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator următor	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator valid			

Comparație II

Operație	VD	LSI	LDI
creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
adaugaInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
adaugaSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta 1$
adaugaPozitie	$O(n)$	$O(n)$	$O(n)$
stergeInceput	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
stergeSfarsit	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
stergePozitie	$O(n)$	$O(n)$	$O(n)$
element	$\Theta(1)$	$O(n)$	$O(n)$
cauta	$O(n)$	$O(n)$	$O(n)$
lungime	$\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$	$\Theta(n)$ sau $\Theta(1)$
Iterator creează	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator element	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator următor	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Iterator valid	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Parcurgere completă cu iterator	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Comparație III

- Din tabelul cu comparația complexităților putem observa:
  - Lista dublu înlănțuită are complexități mai bune (sau la fel de bune) ca lista simplu înlănțuită
  - Listele înlănțuite au un avantaj când lucrăm cu începutul listei/vectorului
  - Vectorul dinamic are complexitate mai bună la accesarea unui element de pe o poziție, dar dacă parcurgem toate elementele cu iterator, complexitatea este egală.
  - Vectorul dinamic are complexitate mai bună la operația lungime.

Exemplu în Java

- În Java TAD Lista (*List*) are 2 implementări:
  - *ArrayList* - o listă implementată pe un Vector Dinamic
  - *LinkedList* - o listă implementată pe o listă dublu înlănțuită
- Ambele implementări au aceleași operații, când avem nevoie de o listă putem folosi oricare (din perspectiva operațiilor), dar, după cum am văzut, aceste operații au complexități diferite.

Parcurgerea unei liste

- Dacă vrem să parcurgem elementele unei liste, avem 2 opțiuni: folosim un iterator sau folosim indici (poziții)
- Mai jos aveți cele 2 variante

```
//presupunem că list este o listă cu elemente de tip TElem
e: TElem
pentru i = 0, list.lungime(), 1 execută
    e = list.element(i)
    scrie e
sf_pentru
```

Parcurgerea unei liste

```
//presupunem că list este o listă cu elemente de tip TElem
e: TElem
it : Iterator
it = list.iterator()
câttimp it.valid() execută
    e = it.element()
    scrie e
    it.urmator()
sf_câttimp
```

Parcurgerea unei liste

```
//presupunem că list este o listă cu elemente de tip TElem
e: TElem
it : Iterator
it = list.iterator()
câttimp it.valid() execută
    e = it.element()
    scrie e
    it.urmator()
sf_câttimp
```

- Cât este complexitatea celor două parcurgeri, dacă lista este reprezentată pe un Vector Dinamic?

Parcurgerea unei liste

```
//presupunem că list este o listă cu elemente de tip TElem
e: TElem
it : Iterator
it = list.iterator()
câttimp it.valid() execută
    e = it.element()
    scrie e
    it.urmator()
sf_câttimp
```

- Cât este complexitatea celor două parcurgeri, dacă lista este reprezentată pe un Vector Dinamic?
- Cât este complexitatea celor două parcurgeri, dacă lista este reprezentată pe o listă înlănțuită?

Exemplu 2

- Să ne gândim la problema următoare: avem o listă de numere, și vrem să *scădem* din această listă o valoare. Această scădere poate însemna să dispară anumite elemente (suma lor totală trebuie să fie egală cu *valoare*) respectiv anumite elemente pot fi înlocuite cu una sau mai multe valori mai mici. Nu există o soluție unică pentru această problemă.
- De exemplu, dacă avem: [2, 5, 3, 7, 4] și vrem să scădem 8:
  - putem elimina 5 și 3 și rămânem cu [2, 7, 4]
  - putem elimina 7 și 4 și adăuga un 3 în listă: [2, 5, 3, 3]
  - putem elimina 4 și 5 și adăuga o valoare de 1 în listă: [2, 3, 7, 1]

Exemplu 2 - Implementare Java

- Aveți mai jos o implementare în Java pentru problema respectivă. Este o implementare *bună* (eficientă)?

```
public static void test(ArrayList<Integer> list, int valoare) {
    while (valoare > 0 && list.size() > 0) {
        Integer elem = list.remove(0);
        if (elem <= valoare) {
            valoare = valoare - elem;
        } else {
            Integer ramane = elem - valoare;
            for (int i = 0; i < ramane; i++) {
                list.add(0, 1); //adăugăm pe poziția 0, valoarea 1
            }
                valoare = 0;
        }
    }
}
```

Stivă - Implementare

- Ce structuri de date putem folosi dacă vrem să implementăm o stivă?

Stivă - Implementare

- Ce structuri de date putem folosi dacă vrem să implementăm o stivă?
- Pentru reprezentarea unei Stive putem folosi:
  - Vector Dinamic
  - Listă Simplu Înlănțuită
  - Listă Dublu Înlănțuită

Stivă - reprezentare folosind un Vector Dinamic

- Dacă vrem să implementăm o Stivă folosind un Vector Dinamic, unde ar trebui să punem vârful Stivei, pentru a avea complexitate cât mai bună pentru operații?
- Vârful Stivei este *capătul* la care adăugăm, și de unde ștergem.

Stivă - reprezentare folosind un Vector Dinamic

- Dacă vrem să implementăm o Stivă folosind un Vector Dinamic, unde ar trebui să punem vârful Stivei, pentru a avea complexitate cât mai bună pentru operații?
- Vârful Stivei este *capătul* la care adăugăm, și de unde ștergem.
- Teoretic putem pune vârful Stivei în 2 locuri:
  - La începutul vectorului (vom adăga la început, și vom șterge de la început)
  - La finalul vectorului (vom adăuga la sfârșit și vom șterge de la sfârșit)
- Pentru a avea complexitate $\Theta(1)$ pentru toate operațiile, vom pune vârful stivei la finalul vectorului.

Stivă - reprezentare folosind un Vector Dinamic

- Cum putem reprezenta o Stivă, folosind un Vector Dinamic?
- Ce câmpuri ar trebui să conțină structura Stivă?

Stivă - reprezentare folosind un Vector Dinamic

- Cum putem reprezenta o Stivă, folosind un Vector Dinamic?
- Ce câmpuri ar trebui să conțină structura Stivă?

Stiva:
varf: Întreg
cap: Întreg
elemente: TElement[]

- Câmpul *vârf* reprezintă și numărul de elemente din vector, adică lungimea vectorului (la alte TAD-ul reprezentate pe Vector Dinamic, câmpul a fost numit *len*). La Stivă se potrivește mai bine denumirea *vârf*.

Stivă - reprezentare folosind o LSI

- Dacă vrem să reprezentăm o Stivă folosind o LSI, unde ar trebui să punem vârful Stivei, pentru a avea complexitate cât mai bună pentru operații?

Stivă - reprezentare folosind o LSI

- Dacă vrem să reprezentăm o Stivă folosind o LSI, unde ar trebui să punem vârful Stivei, pentru a avea complexitate cât mai bună pentru operații?
- Teoretic putem pune vârful Stivei în 2 locuri:
  - La începutul listei (vom adăga la început, și vom șterge de la început)
  - La finalul listei (vom adăuga la sfârșit și vom șterge de la sfârșit)
- Pentru a avea complexitate $\Theta(1)$ pentru toate operațiile, vom pune vârful stivei la începutul listei.

Stivă - reprezentare folosind o LSI

- Cum putem reprezenta o Stivă, folosind o Listă Simplu Înlănțuită?
- Ce câmpuri ar trebui să conțină structura Stivă?

Stivă - reprezentare folosind o LSI

- Cum putem reprezenta o Stivă, folosind o Listă Simplu Înlănțuită?
- Ce câmpuri ar trebui să conțină structura Stivă?

Nod:
elem: TElement
urm: ↑ Nod

Stiva:
varf: ↑ Nod

- Câmpul *vârf* este adresa primului Nod, la alte TAD-uri l-am numit *prim*.
- Ne-ar ajuta cu ceva dacă am reține în structura Stivă și numărul de elemente?

Stivă - reprezentare folosind o LDI

- Dacă vrem să reprezentăm o Stivă folosind o LDI, unde ar trebui să punem vârful Stivei, pentru a avea complexitate cât mai bună pentru operații?

Stivă - reprezentare folosind o LDI

- Dacă vrem să reprezentăm o Stivă folosind o LDI, unde ar trebui să punem vârful Stivei, pentru a avea complexitate cât mai bună pentru operații?
- Teoretic putem pune vârful Stivei în 2 locuri:
  - La începutul listei (vom adăga la început, și vom șterge de la început)
  - La finalul listei (vom adăuga la sfârșit și vom șterge de la sfârșit)
- Pentru a avea complexitate $\Theta(1)$ pentru toate operațiile, putem pune vârful oriunde.

Stivă - reprezentare folosind o LDI

- Cum putem reprezenta o Stivă, folosind o Listă Dublu Înlănțuită?
- Ce câmpuri ar trebui să conțină structura Stivă?

Stivă - reprezentare folosind o LDI

- Cum putem reprezenta o Stivă, folosind o Listă Dublu Înlănțuită?
- Ce câmpuri ar trebui să conțină structura Stivă?

Nod:
elem: TElement
urm: ↑ Nod
ant: ↑ Nod

Stiva:
varf: ↑ Nod

Coadă - implementare

- Ce structuri de date putem folosi dacă vrem să implementăm o coadă?

Coadă - implementare

- Ce structuri de date putem folosi dacă vrem să implementăm o coadă?
- Pentru reprezentarea unei Cozi putem folosi:
  - Vector Dinamic
  - Listă Simplu Înlănțuită
  - Listă Dublu Înlănțuită
- Vom discuta pentru fiecare structură de date cum putem reprezenta o Coadă (unde punem *front* și unde punem *end*), încercând să avem complexitate $\Theta(1)$ pentru toate operațiile.

Coadă - reprezentare folosind un Vector Dinamic

- Dacă vrem să implementăm o Coadă folosind un Vector Dinamic, unde ar trebui să punem *front* și unde ar trebui să punem *end*-ul, pentru a avea complexitate cât mai bună pentru operații?

Coadă - reprezentare folosind un Vector Dinamic

- Dacă vrem să implementăm o Coadă folosind un Vector Dinamic, unde ar trebui să punem *front* și unde ar trebui să punem *end*-ul, pentru a avea complexitate cât mai bună pentru operații?
- Teoretic avem 2 variante:
  - Punem *front* la începutul vectorului și *end* la sfârșit (vom șterge de la început, și vom adăuga la final)
  - Punem *end* la începutul vectorului și *front* la sfârșit (vom adăuga la început, și vom șterge de la final)

Coadă - reprezentare folosind un Vector Dinamic

- Dacă vrem să implementăm o Coadă folosind un Vector Dinamic, unde ar trebui să punem *front* și unde ar trebui să punem *end*-ul, pentru a avea complexitate cât mai bună pentru operații?
- Teoretic avem 2 variante:
  - Punem *front* la începutul vectorului și *end* la sfârșit (vom șterge de la început, și vom adăuga la final)
  - Punem *end* la începutul vectorului și *front* la sfârșit (vom adăuga la început, și vom șterge de la final)
- Operațiile de adăugare la început și ștergere de la început au complexitate $\Theta(n)$, deci oricare variantă alegem, vom avea o operație cu complexitate $\Theta(n)$.

Coadă - reprezentare folosind un Vector Dinamic

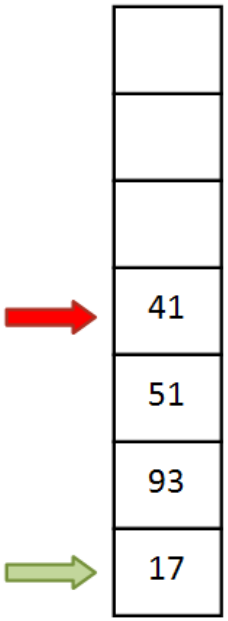
- Puteți să vă gândiți la o reprezentare pentru Coadă, folosind un Vector Dinamic astfel încât să avem totuși complexitate $\Theta(1)$ pentru toate operațiile?

Coadă - reprezentare folosind un Vector Dinamic

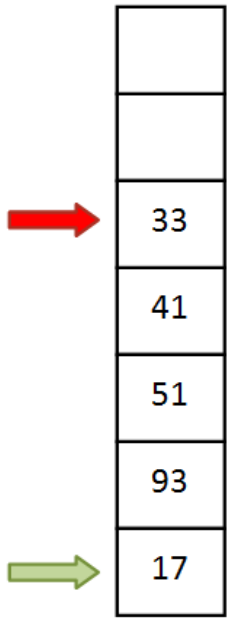
- Puteți să vă gândiți la o reprezentare pentru Coadă, folosind un Vector Dinamic astfel încât să avem totuși complexitate $\Theta(1)$ pentru toate operațiile?
- Reprezentarea anterioară presupune că ori *front*-ul, ori *end*-ul cozii se găsește pe prima poziție din vectorul dinamic, și când avem de adăugat sau șters de la începutul vectorului vom muta restul elementelor.
- Putem să reprezentăm Coada pe un Vector Dinamic, fără să forțăm ca *front* sau *end* să fie la începutul vectorului (pe poziția 0), dar trebuie să reținem separat *front* și *end*.
- Această reprezentare se numește reprezentare pe Vector Circular.

Coadă - Exemplu cu Vector Circular

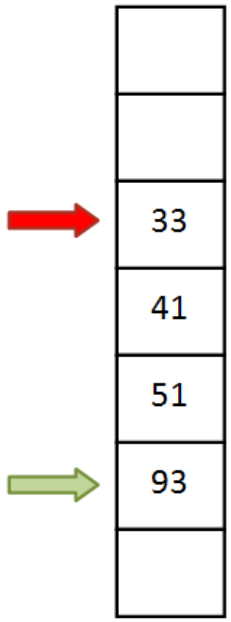
- Aceasta este coada noastră (săgeata verde e *front*, săgeata roșie este *end*):



- Adăugăm numărul 33 :

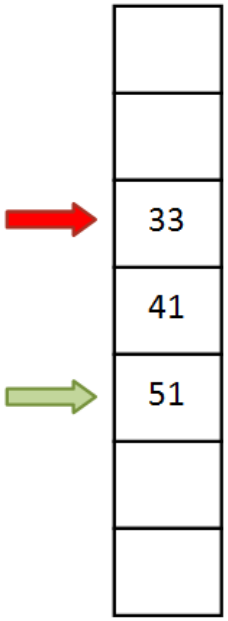


- Ștergem un element (dar nu mutăm restul):

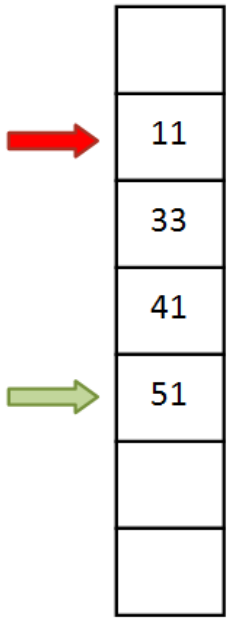


Coadă - Exemplu cu Vector Circular

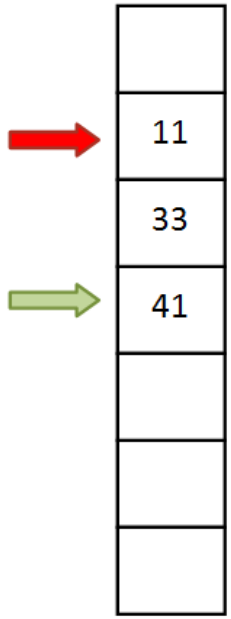
- Ștergem încă un element:



- Adăugăm numărul 11:

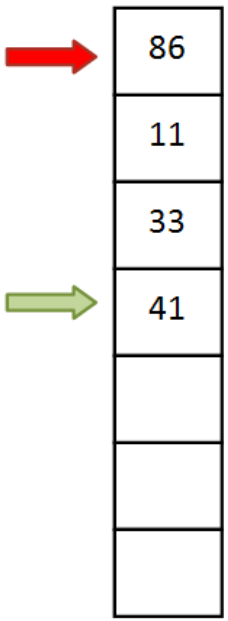


- Ștergem un element:

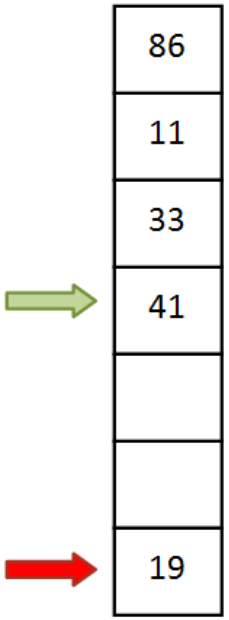


Coadă - Exemplu cu Vector Circular

- Adăugăm numărul 86:



- Adăugăm numărul 19:



Coadă - reprezentare pe Vector Circular

- Cum putem reprezenta o Coadă pe un Vector Circular?
- Ce câmpuri ar trebui să aibă structura Coada?

Coadă - reprezentare pe Vector Circular

- Cum putem reprezenta o Coadă pe un Vector Circular?
- Ce câmpuri ar trebui să aibă structura Coada?

Coada:
cap: Întreg
front: Întreg
end: Întreg
elemente: TElement[]

- Să vedem cum putem implementa fiecare operație pentru această reprezentare (vom începe cu operațiile mai simple).

Coadă - pe Vector Circular - creeaza

- Operația creează inițializează câmpurile din Coada.
- Vom nota *front* și *end* pentru o Coadă vidă cu valoarea -1.

subalgoritm creeaza() **este:**
  this.cap = 10 *//alegem noi o capacitate inițială*
  this.front = -1
  this.end = -1
  this.elemente = @tablou cu 10 elemente
sf_subalgoritm

- Complexitate:

Coadă - pe Vector Circular - creeaza

- Operația creează inițializează câmpurile din Coada.
- Vom nota *front* și *end* pentru o Coadă vidă cu valoarea -1.

subalgoritm creeaza() **este:**
  this.cap = 10 *//alegem noi o capacitate inițială*
  this.front = -1
  this.end = -1
  this.elemente = @tablou cu 10 elemente
sf_subalgoritm

- Complexitate: $\Theta(1)$

Coadă - pe Vector Circular - vida

- Cum verificăm dacă Coada este vidă?

Coadă - pe Vector Circular - vida

- Cum verificăm dacă Coada este vidă?

```
subalgoritm vida() este:
    dacă this.front == -1 atunci
        returnează Adevărat
    altfel
        returnează Fals
    sf_dacă
sf_subalgoritm
```

- Complexitate:

Coadă - pe Vector Circular - vida

- Cum verificăm dacă Coada este vidă?

```
subalgoritm vida() este:
    dacă this.front == -1 atunci
        returnează Adevărat
    altfel
        returnează Fals
    sf_dacă
sf_subalgoritm
```

- Complexitate: $\Theta(1)$

Coadă - pe Vector Circular - element

- Element trebuie să returneze elementul de la *front*-ul cozii. Trebuie să verificăm să nu avem coadă vidă.

```
subalgoritm element() este:
    dacă this.front != -1 atunci
        returnează this.elemente[this.front]
    sf_dacă
//aruncăm exceptie pentru că avem o coada vida
sf_subalgoritm
```

- Complexitate:

Coadă - pe Vector Circular - element

- Element trebuie să returneze elementul de la *front*-ul cozii. Trebuie să verificăm să nu avem coadă vidă.

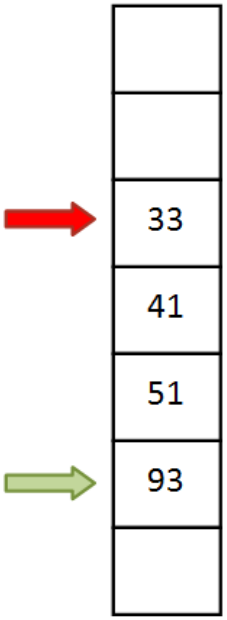
```
subalgoritm element() este:
    dacă this.front != -1 atunci
        returnează this.elemente[this.front]
    sf_dacă
//aruncăm exceptie pentru că avem o coada vida
sf_subalgoritm
```

- Complexitate: $\Theta(1)$

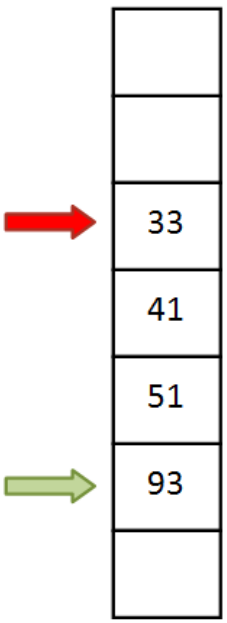
Coadă - pe Vector Circular - sterge

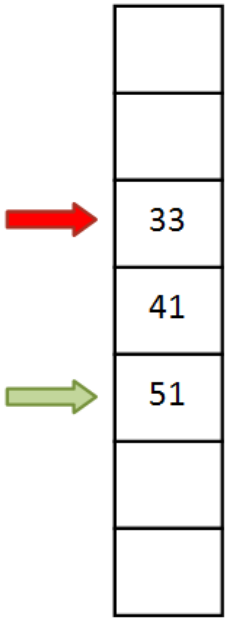
- La operația sterge trebuie să verificăm dacă:
  - Coada este vidă (nu avem ce șterge)
  - Coada are un singur element (dacă coada are un singur element și îl ștergem, *front* și *end* trebuie setat la -1)
  - Altfel există 2 situații:

Coadă - pe Vector Circular - sterge



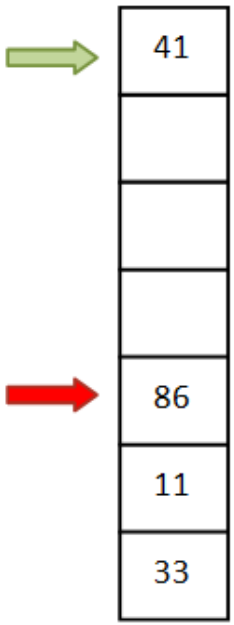
Coadă - pe Vector Circular - sterge



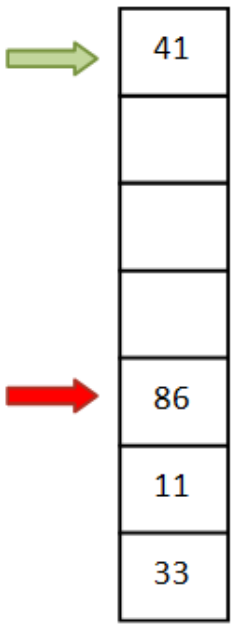


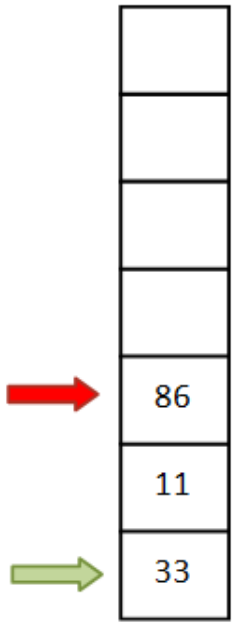
- Dacă șterg, *front* crește cu 1:

Coadă - pe Vector Circular - sterge



Coadă - pe Vector Circular - sterge





- Dacă șterg, *front* trece la începutul vectorului.

Coadă pe Vector Circular - sterge

```
funcție sterge () este:
    dacă this.front != -1 atunci
        elemSters: TElem
        elemSters = this.elemente[this.front]
        dacă this.front == this.end atunci //avem un singur element
            this.front = -1
            this.end = -1
        altfel dacă this.front == this.cap atunci
            this.front = 0
        altfel
            this.front = this.front + 1
        sf_dacă
        returnează elemSters
    altfel
        @aruncă excepție, coada e vidă
    sf_dacă
sf_funcție
```

- Complexitate:

Coadă pe Vector Circular - sterge

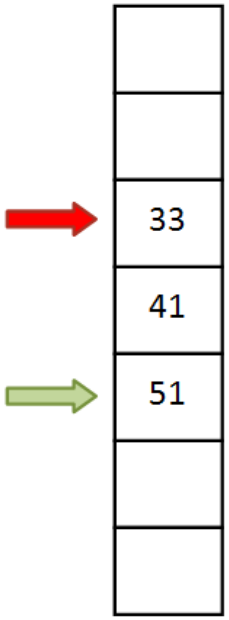
```
funcție sterge () este:
    dacă this.front != -1 atunci
        elemSters: TElem
        elemSters = this.elemente[this.front]
        dacă this.front == this.end atunci //avem un singur element
            this.front = -1
            this.end = -1
        altfel dacă this.front == this.cap atunci
            this.front = 0
        altfel
            this.front = this.front + 1
        sf_dacă
        returnează elemSters
    altfel
        @aruncă excepție, coada e vidă
    sf_dacă
sf_funcție
```

- Complexitate: O(1)

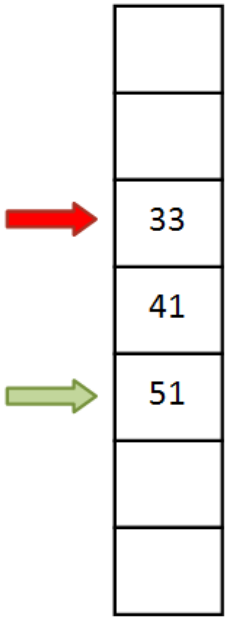
Coadă pe Vector Circular - adauga

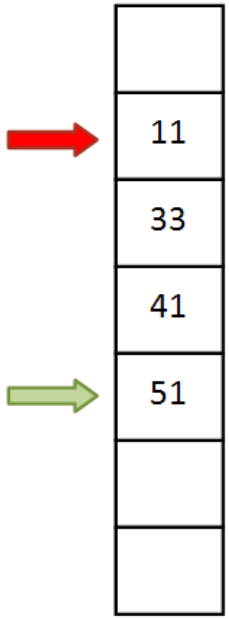
- La adăugare trebuie să verificăm dacă:
  - Coada este vidă (punem elementul pe poziția 0)
  - Coada este plină (dacă nu sunt locuri libere, trebuie să alocăm un vector mai mare)
  - Altfel, există 2 variante:

Coadă - pe Vector Circular - adaugă



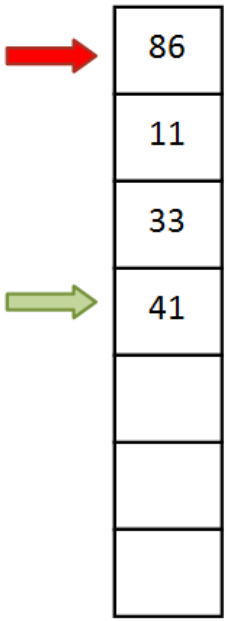
Coadă - pe Vector Circular - adaugă



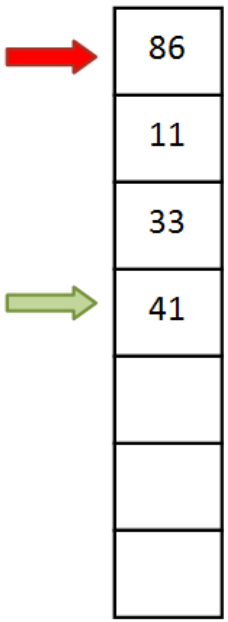


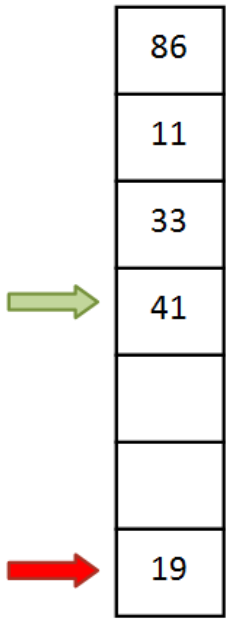
- Dacă adaug, *end* crește cu 1:

Coadă - pe Vector Circular - adauga



Coadă - pe Vector Circular - adauga

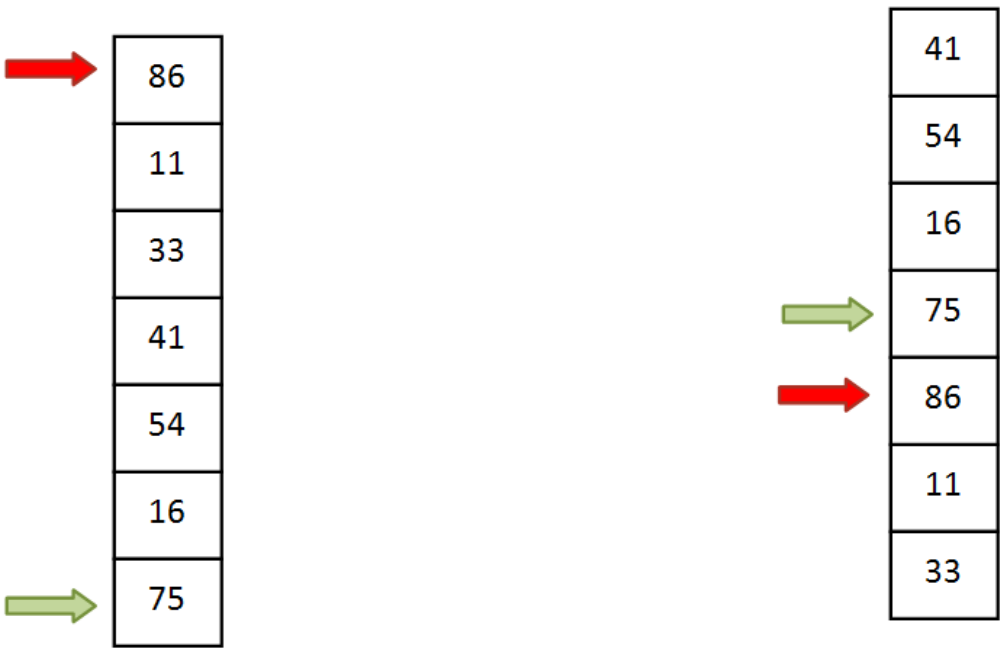




- Dacă adaug, *end* trece la începutul vectorului.

Coadă - pe Vector Circular - adauga

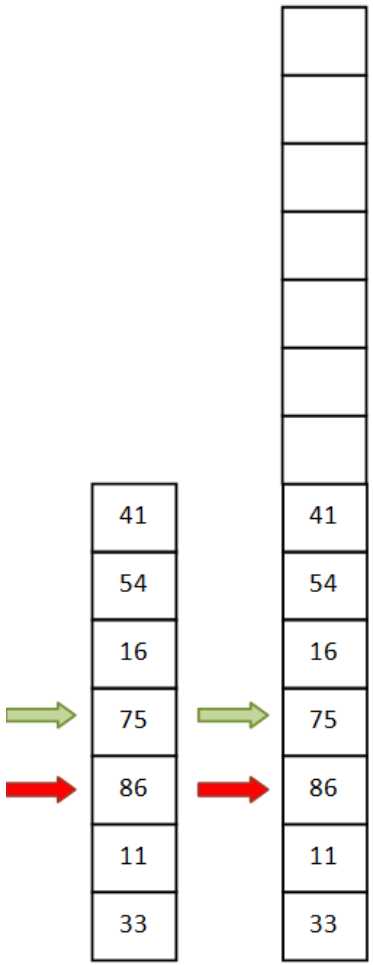
- De unde știm că avem o coadă plină?
- Există 2 situații:



- În ambele situații ordinea în care elementele au fost adăugate în coadă este: 75, 16, 54, 41, 33, 11, 86

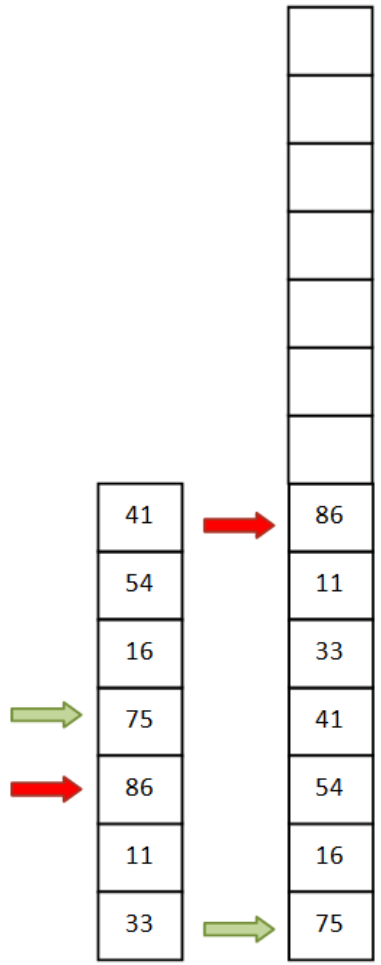
Coadă - pe Vector Circular - adauga

- Dacă vectorul este plin și alocăm unul mai mare, nu este suficient să copiem elementele din tabloul vechi.
- Tot nu avem unde să punem elementul nou.



Coadă - pe Vector Circular - adauga

- Trebuie să copiem elementele în așa fel încât să *îndreptăm* coada.
- Pe poziția 0 vom pune elementul de pe poz *front*.
- Pe poziția 1 vom pune elementul de pe poz *front*+ 1.
- ...
- Pe poziția *cap* vom pune elementul de pe poz *end*.
- Regula: pe poziția *i* vom pune elementul de pe poziția: ((front + i) % cap)



```
subalgoritm adauga(e: TElem) este:
    dacă this.front == -1 atunci
        this.elemente[0] = e
        this.front = 0
        this.end = 0
        @return
    altfel dacă (this.front==0 Și this.end==this.cap) SAU
this.end==this.front-1 atunci
        elementeNou: TElement[this.cap * 2]
        i: Întreg
        pentru i = 0, this.cap, 1 execută
            elementeNou[i] = this.elemente[(this.front + i) % this.cap]
        sf_pentru
        this.elemente = elementeNou
        this.front = 0
        this.end = this.cap-1
        this.cap = this.cap * 2
    sf_dacă
//acum sigur avem loc liber unde să punem elementul
//continuăm pe pagina următoare
```

Coadă - pe Vector Circular - adauga

dacă this.end $\neq$ this.cap **atunci**
  this.elemente[this.end+1] = e
  this.end = this.end + 1
altfel
  this.elemente[0] = e
  this.end = 0
sf_dacă
sf_subalgoritm

- Complexitate:

Coadă - pe Vector Circular - adauga

dacă this.end $\neq$ this.cap **atunci**
  this.elemente[this.end+1] = e
  this.end = this.end + 1
altfel
  this.elemente[0] = e
  this.end = 0
sf_dacă
sf_subalgoritm

- Complexitate: $\Theta(1)$

Coadă - reprezentare folosind o LSI

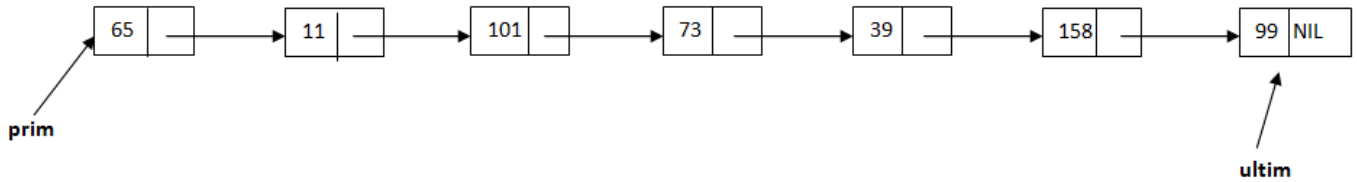
- Dacă vrem să reprezentăm o Coadă folosind o LSI, unde ar trebui să punem *front* și unde ar trebui să punem *end*-ul, pentru a avea complexitate cât mai bună pentru operații?

Coadă - reprezentare folosind o LSI

- Dacă vrem să reprezentăm o Coadă folosind o LSI, unde ar trebui să punem *front* și unde ar trebui să punem *end*-ul, pentru a avea complexitate cât mai bună pentru operații?
- Teoretic avem 2 variante:
  - Punem *front* la începutul listei și *end* la sfârșit (vom șterge de la început, și vom adăuga la final)
  - Punem *end* la începutul listei și *front* la sfârșit (vom adăuga la început, și vom șterge de la final)
- La o LSI în general avem acces direct doar la primul nod, deci orice operație vrem să facem la sfârșitul listei, trebuie să parcurgem lista, și vom avea complexitate $\Theta(n)$.

Coadă - reprezentare folosind o LSI

- Chiar dacă lista este simplu înlănțuită, deci nodurile au legătură doar spre nodul următor, putem să reținem și ultimul nod din listă.



- Având acces la primul și ultimul nod, ce operație pot face în $\Theta(1)$ la finalul listei? Adăugare sau ștergere?

Coadă - reprezentare folosind o LSI

- Pentru a șterge un element de la finalul listei, am nevoie și de penultimul nod (la care nu am acces), deci ștergerea ia $\Theta(n)$. Dar, pot adăuga un element nou la finalul listei, fără să am nevoie de alt nod din listă.
- Deci, dacă aleg să pun *front* (de unde ștergem) la începutul listei, și *end* (unde adăugăm) la finalul listei, și reținem primul și ultimul nod din listă, putem implementa o Coadă unde toate operațiile au $\Theta(1)$.

Nod:
elem: TElement
urm: ↑ Nod

Coada:
front: ↑ Nod //*e primul nod din listă*
end: ↑ Nod //*e ultimul nod din listă*

Coadă - reprezentare folosind o LDI

- Dacă vrem să reprezentăm o Coadă folosind o LDI, unde ar trebui să punem *front* și unde ar trebui să punem *end*-ul, pentru a avea complexitate cât mai bună pentru operații?

Coadă - reprezentare folosind o LDI

- Dacă vrem să reprezentăm o Coadă folosind o LDI, unde ar trebui să punem *front* și unde ar trebui să punem *end*-ul, pentru a avea complexitate cât mai bună pentru operații?
- Teoretic avem 2 variante:
  - Punem *front* la începutul listei și *end* la sfârșit (vom șterge de la început, și vom adăuga la final)
  - Punem *end* la începutul listei și *front* la sfârșit (vom adăuga la început, și vom șterge de la final)
- Din moment ce într-o LDI pot adăuga și șterge la ambele capete în $\Theta(1)$, pot alege oricare variantă.