PROGRAMARE ȘI STRUCTURI DE DATE

CURS 10

Lect. dr. Oneț-Marian Zsuzsanna

Facultatea de Matematică și Informatică UBB
în colaborare cu NTT Data

Cuprins

Pointeri

- Orice variabilă definită are o adresă de memorie, care arată locul unde variabila respectivă este stocată în memorie.
- Un *pointer* este un tip de date care reține o adresă de memorie. O variabilă de tip pointer poate să rețină o adresă de memorie (probabil adresa unei alte variabile).
- În pseudocod, o variabilă de tip *pointer* este reprezentată de o săgeată în sus: $\uparrow$
  - a: $\uparrow$ Întreg - *a* este o variabilă care reține adresa unei alte variabile de tip Întreg
  - c: $\uparrow$ Colecție - *c* este o variabilă care reține adresa unei variabile de tip Colecție
  - etc.

Pointeri II

- Dacă am un pointer, prin el pot accesa variabila adresa căreia o am în pointer (se numește *dereferențiere*).
- În pseudocod pentru *dereferențiere* folosim notația [ ].
  - Dacă a este un pointer la un Întreg, $[a]$ reprezintă numărul de la adresa a
  - Dacă c este un pointer la o Colecție, $[c]$ reprezintă colecția de la adresa c.

Pointeri III

numar: Întreg *//definim o variabilă de tip Întreg*
adresa: ↑Întreg *//definim o variabilă de tip pointer la un Întreg*
numar = 99
adresa = @adresa variabilei numar *//reținem adresa variabilei număr*
scrie numar

Pointeri III

numar: Întreg *//definim o variabilă de tip Întreg*
adresa: ↑Întreg *//definim o variabilă de tip pointer la un Întreg*
numar = 99
adresa = @adresa variabilei numar *//reținem adresa variabilei număr*
scrie numar *//va afișa 99*
scrie adresa

Pointeri III

numar: Întreg *//definim o variabilă de tip Întreg*
adresa: ↑Întreg *//definim o variabilă de tip pointer la un Întreg*
numar = 99
adresa = @adresa variabilei numar *//reținem adresa variabilei număr*
scrie numar *//va afișa 99*
scrie adresa *//va afișa o adresă - de ex 12A961*
scrie [adresa]

Pointeri III

numar: Întreg *//definim o variabilă de tip Întreg*
adresa: ↑Întreg *//definim o variabilă de tip pointer la un Întreg*
numar = 99
adresa = @adresa variabilei numar *//reținem adresa variabilei număr*
scrie numar *//va afișa 99*
scrie adresa *//va afișa o adresă - de ex 12A961*
scrie [adresa] *//va afișa valoarea variabilei de la adresa adresă, adică 99*
[*adresa*] = 199
scrie numar

Pointeri III

numar: Întreg *//definim o variabilă de tip Întreg*
adresa: ↑Întreg *//definim o variabilă de tip pointer la un Întreg*
numar = 99
adresa = @adresa variabilei numar *//reținem adresa variabilei număr*
scrie numar *//va afișa 99*
scrie adresa *//va afișa o adresă - de ex 12A961*
scrie [adresa] *//va afișa valoarea variabilei de la adresa adresă, adică 99*
[*adresa*] = 199
scrie numar *//va afișa 199*
scrie adresa

Pointeri III

numar: Întreg *//definim o variabilă de tip Întreg*
adresa: ↑Întreg *//definim o variabilă de tip pointer la un Întreg*
numar = 99
adresa = @adresa variabilei numar *//reținem adresa variabilei număr*
scrie numar *//va afișa 99*
scrie adresa *//va afișa o adresă - de ex 12A961*
scrie [adresa] *//va afișa valoarea variabilei de la adresa adresă, adică 99*
[*adresa*] = 199
scrie numar *//va afișa 199*
scrie adresa *//va afișa aceeași adresă ca înainte, 12A961*
scrie [adresa]

Pointeri III

numar: Întreg *//definim o variabilă de tip Întreg*
adresa: ↑Întreg *//definim o variabilă de tip pointer la un Întreg*
numar = 99
adresa = @adresa variabilei numar *//reținem adresa variabilei număr*
scrie numar *//va afișa 99*
scrie adresa *//va afișa o adresă - de ex 12A961*
scrie [adresa] *//va afișa valoarea variabilei de la adresa adresă, adică 99*
[*adresa*] = 199
scrie numar *//va afișa 199*
scrie adresa *//va afișa aceeași adresă ca înainte, 12A961*
scrie [adresa] *//va afișa valoarea variabilei de la adresa adresă, adică 199*

- O variabilă de tip pointer care nu reține (încă) o adresă validă, în general este setat la valoarea *NIL*. *NIL* reprezintă o adresă invalidă.

Pointeri IV

- Anumite limbaje de programare permit accesare adreselor de memorie (deci pot fi definite variabile de tip pointer), alte limbaje de programare nu permit acest lucru.
- De exemplu în C++, programatorul poate să decidă dacă vrea o variabilă de tip Colecție sau o variabilă de tip pointer la Colecție.
- În Java, nu putem accesa adresa de memorie unei variabile, și nici nu putem defini variabile de tip pointer.

Pointeri V

- În Java nu putem accesa adresa de memorie a unei variabile. Astfel nu există nici dereferențiere.
- În schimb, orice variabilă de tip clasă este reținută prin adresa ei.
- Am discutat că la tablouri se reține intern adresa primului element din tablou. La obiecte (variabile de tip clasă) intern se reține adresa obiectului. Deci practic nu avem pointeri în Java, pentru că (aproape) tot e pointer (cu excepția tipurilor primitive: int, boolean, double, etc.).

Pointeri VI

```
Colectie col = new Colectie()
//col este adresa unde e reținută colecția, chiar dacă nu îl vedem așa
System.out.println(col.dimensiune()) //va afișa 0
Colecie col2 = col //col2 reține aceeași adresă.
col2.adauga(45)
System.out.println(col.dimensiune()) //va afișa 1
```

- În Java o variabilă care nu reține nici-o adresă (încă) se inițializează cu *null*

Lista înlănțuită

- Am discutat că Vectorul Dinamic rezolvă dezavantajul principal al unui tablou simplu: necesitatea de a stabili de la început numărul de elemente.

Lista înlănțuită

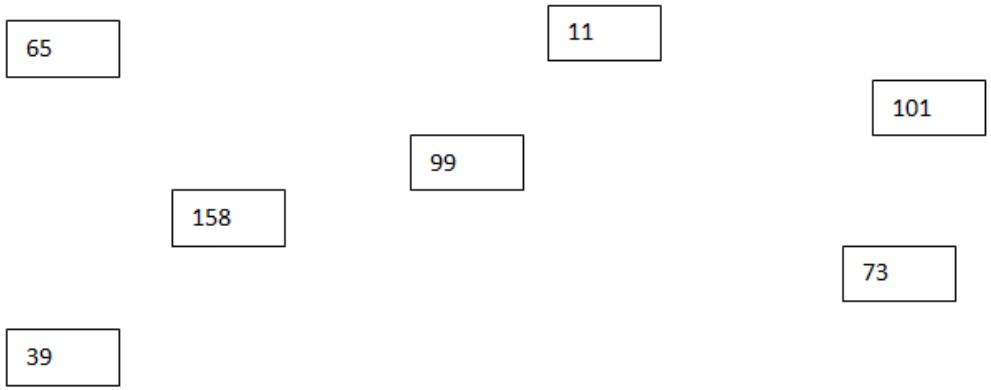
- Am discutat că Vectorul Dinamic rezolvă dezavantajul principal al unui tablou simplu: necesitatea de a stabili de la început numărul de elemente.
- Dar mai rămâne un dezavantaj: de multe ori, vectorul ocupă zonă de memorie chiar dacă nu sunt elemente acolo. Operația care mărește Vectorul Dinamic în general dublează mărimea tabloului (și capacitatea), și majoritatea pozițiilor nu sunt ocupate. Evident, ele pot fi ocupate pe parcurs, dacă se tot adaugă elemente, dar dacă nu sunt alte adăugări, atunci rămâne zonă ocupată degeaba.
- O altă problemă la Vector Dinamic (și la tablou simplu) apare dacă trebuie să adăugăm elemente la început, sau să ștergem elemente de la început. În ambele situații toate elementele trebuie mutate.

Lista înlănțuită II

- *Lista înlănțuită* este o structură de date care rezolvă prima problemă, ea ocupă tot timpul doar atât spațiu cât este necesar pentru elementele existente. Nu există spațiu extra, pentru elementele care poate vor fi adăugate în viitor.
- Lista înlănțuită este alcătuită de fapt din *noduri*, fiecare nod conține câte un element.
- Aceste noduri care compun lista înlănțuită nu trebuie să fie în zone consecutive de memorie, ele pot fi oriunde în memorie.

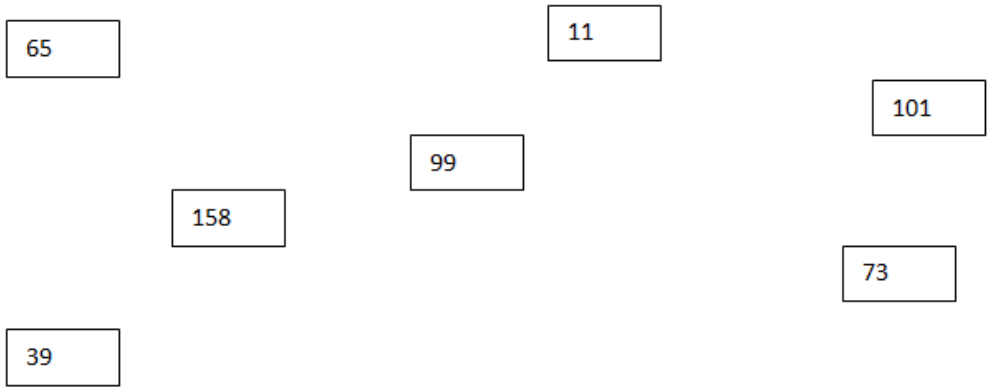
Lista înlănțuită III

- Să prespunem că avem o listă înlănțuită cu următoarele 7 noduri (în fiecare nod vedem elementul reținut în nod):



Lista înlănțuită III

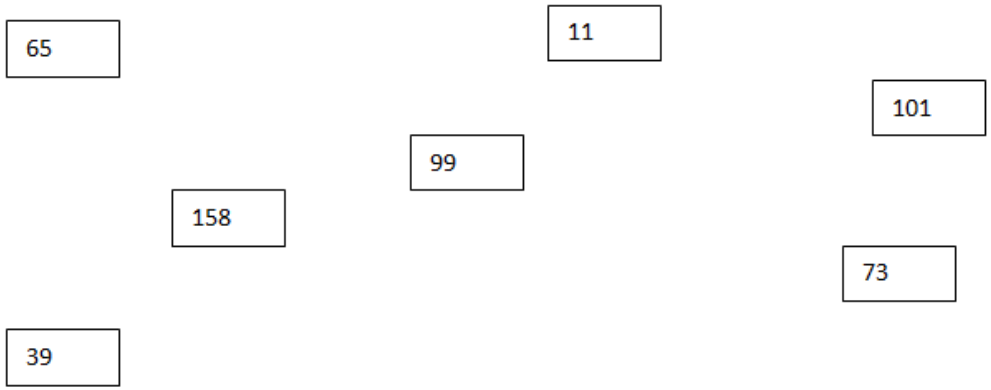
- Să prespunem că avem o listă înlănțuită cu următoarele 7 noduri (în fiecare nod vedem elementul reținut în nod):



- Care este problema cu acestă listă?

Lista înlănțuită III

- Să prespunem că avem o listă înlănțuită cu următoarele 7 noduri (în fiecare nod vedem elementul reținut în nod):



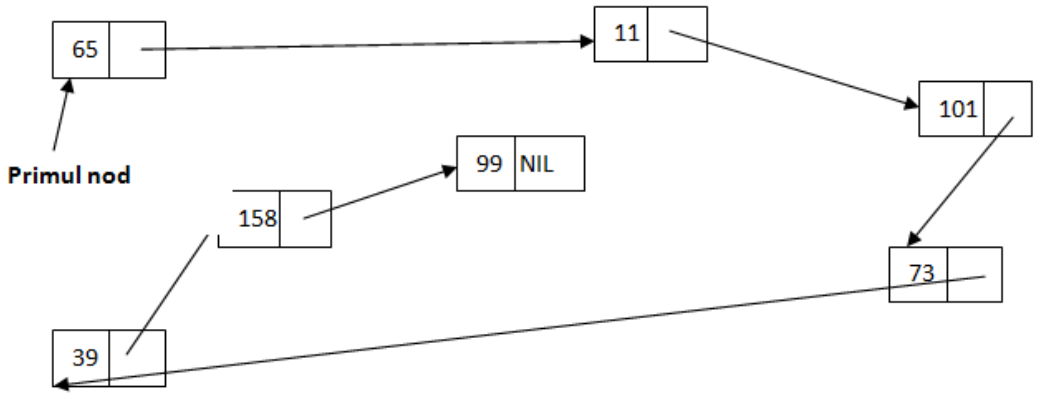
- Care este problema cu acestă listă?
  - Nu știm care e primul element
  - Nu știm care e ultimul element
  - Nu știm în ce ordine vin elementele

Lista înlănțuită IV

- Pentru a putea reține ordinea elementelor, fiecare nod conține pe lângă elementul reținut și *adresa nodului următor* (un pointer spre nodul următor).
- Ultimul nod din listă reține ca adresa nodului următor valoarea *NIL*.

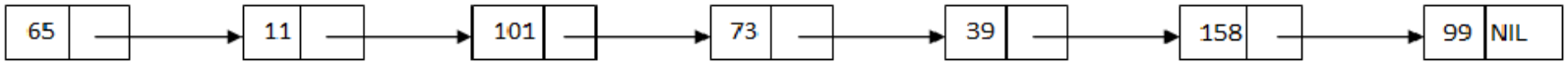
Lista înlănțuită IV

- Pentru a putea reține ordinea elementelor, fiecare nod conține pe lângă elementul reținut și *adresa nodului următor* (un pointer spre nodul următor).
- Ultimul nod din listă reține ca adresa nodului următor valoarea *NIL*.



Lista înlănțuită V

- Chiar dacă nodurile nu sunt în zone de memorie consecutive, în general, pe desen, vom pune nodurile unul după altul:



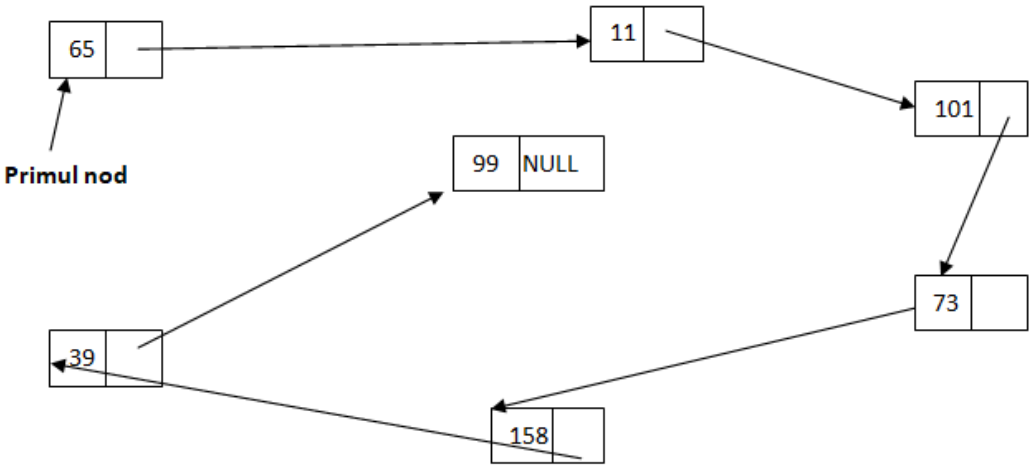
Lista înlănțuită V

- Lista din figura anterioară se numește *listă simplu înlănțuită*, pentru că fiecare nod are legătură spre nodul următor. Există și *listă dublu înlănțuită* în care fiecare nod are legătură spre nodul anterior și următor.
- Deocamdată vom vorbi de lista simplu înlănțuită.

Lista înlănțuită V

- Lista din figura anterioară se numește *listă simplu înlănțuită*, pentru că fiecare nod are legătură spre nodul următor. Există și *listă dublu înlănțuită* în care fiecare nod are legătură spre nodul anterior și următor.
- Deocamdată vom vorbi de lista simplu înlănțuită.
- Am văzut că prin legături (pointeri spre nodul următor) putem stabili ordinea nodurilor. Ceea ce trebuie să reținem pentru o listă simplu înlănțuită, este adresa primului nod. Dacă am adresa primului nod, de acolo pot ajunge la orice element.

Lista înlănțuită V



Lista simplu înlănțuită - reprezentare

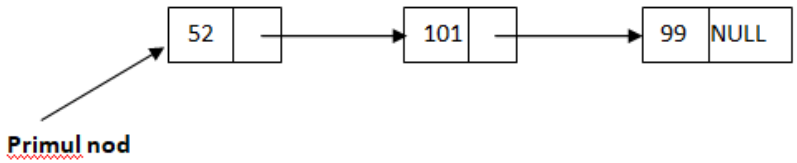
- Dacă vrem să implementăm o listă simplu înlănțuită, avem nevoie de 2 structuri:

Nod:
elem: TElem
urm: ↑ Nod

ListaSimpluInlantuita:
prim: ↑ Nod

Lista simplu înlănțuită - reprezentare

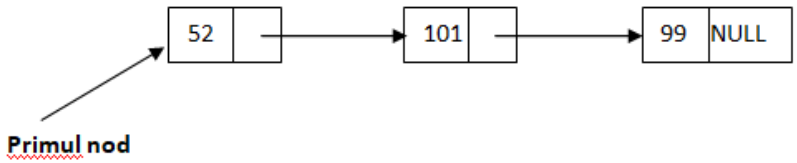
- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim -

Lista simplu înlănțuită - reprezentare

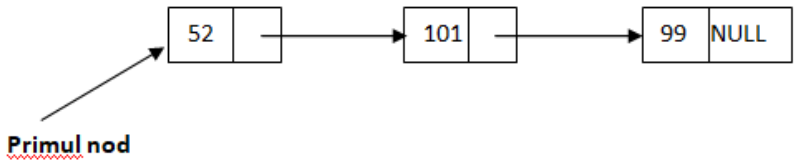
- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim - adresa primului nod din listă (pointer la nodul cu 52)
[*lst.prim*].*elem*

Lista simplu înlănțuită - reprezentare

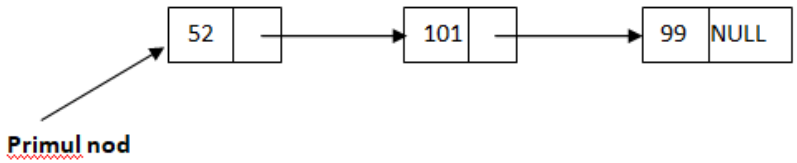
- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim - adresa primului nod din listă (pointer la nodul cu 52)
[*lst.prim*].*elem* - elementul din primul nod (valoarea 52)
[*list.prim*].*urm* -

Lista simplu înlănțuită - reprezentare

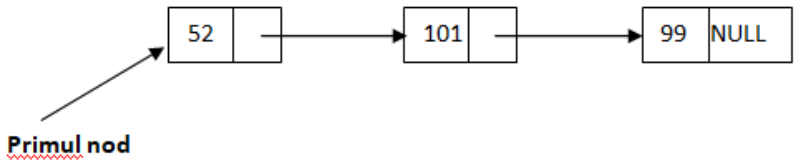
- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim - adresa primului nod din listă (pointer la nodul cu 52)
[*lst.prim*].*elem* - elementul din primul nod (valoarea 52)
[*list.prim*].*urm* - adresa celui de-al doilea nod (pointer la nodul cu 101)
[[*lst.prim*].*urm*].*elem* -

Lista simplu înlănțuită - reprezentare

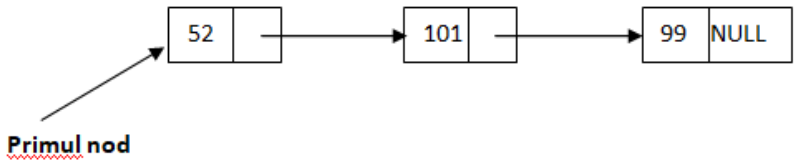
- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim - adresa primului nod din listă (pointer la nodul cu 52)
[*lst.prim*].*elem* - elementul din primul nod (valoarea 52)
[*list.prim*].*urm* - adresa celui de-al doilea nod (pointer la nodul cu 101)
[[*lst.prim*].*urm*].*elem* - elementul din al 2-lea nod (valoarea 101)
[[*lst.prim*].*urm*].*urm* -

Lista simplu înlănțuită - reprezentare

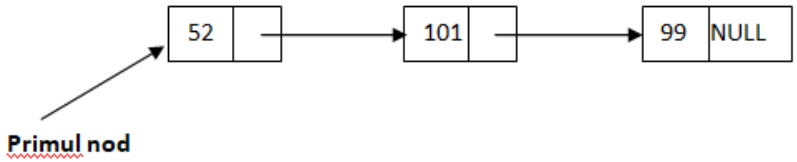
- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim - adresa primului nod din listă (pointer la nodul cu 52)
[*lst.prim*].*elem* - elementul din primul nod (valoarea 52)
[*list.prim*].*urm* - adresa celui de-al doilea nod (pointer la nodul cu 101)
[[*lst.prim*].*urm*].*elem* - elementul din al 2-lea nod (valoarea 101)
[[*lst.prim*].*urm*].*urm* - adresa celui de-al treilea nod (pointer la nodul cu 99)
[[[*lst.prim*].*urm*].*urm*].*elem* -

Lista simplu înlănțuită - reprezentare

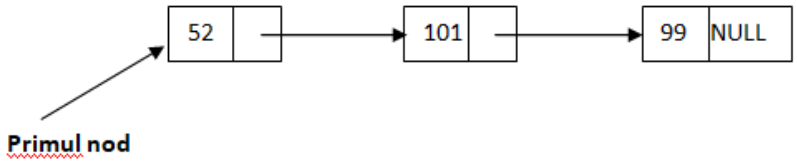
- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim - adresa primului nod din listă (pointer la nodul cu 52)
[*lst.prim*].*elem* - elementul din primul nod (valoarea 52)
[*list.prim*].*urm* - adresa celui de-al doilea nod (pointer la nodul cu 101)
[[*lst.prim*].*urm*].*elem* - elementul din al 2-lea nod (valoarea 101)
[[*lst.prim*].*urm*].*urm* - adresa celui de-al treilea nod (pointer la nodul cu 99)
[[[*lst.prim*].*urm*].*urm*].*elem* - elementul din al 3-lea nod (valoarea 99)
[[[*lst.prim*].*urm*].*urm*].*urm* -

Lista simplu înlănțuită - reprezentare

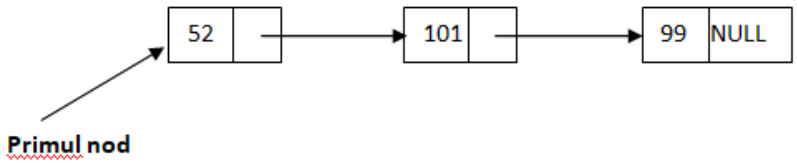
- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim - adresa primului nod din listă (pointer la nodul cu 52)
[*lst.prim*].*elem* - elementul din primul nod (valoarea 52)
[*list.prim*].*urm* - adresa celui de-al doilea nod (pointer la nodul cu 101)
[[*lst.prim*].*urm*].*elem* - elementul din al 2-lea nod (valoarea 101)
[[*lst.prim*].*urm*].*urm* - adresa celui de-al treilea nod (pointer la nodul cu 99)
[[[*lst.prim*].*urm*].*urm*].*elem* - elementul din al 3-lea nod (valoarea 99)
[[[*lst.prim*].*urm*].*urm*].*urm* - adresa celui de-al patrulea nod. Din moment ce nu există decât 3 noduri, această adresă este NIL
[[[[*lst.prim*].*urm*].*urm*].*urm*].*elem* -

Lista simplu înlănțuită - reprezentare

- Să presupunem că *lst* este o variabilă de tip *ListaSimpluInlantuita* care conține exact 3 elemente:



lst.prim - adresa primului nod din listă (pointer la nodul cu 52)
[*lst.prim*].*elem* - elementul din primul nod (valoarea 52)
[*list.prim*].*urm* - adresa celui de-al doilea nod (pointer la nodul cu 101)
[[*lst.prim*].*urm*].*elem* - elementul din al 2-lea nod (valoarea 101)
[[*lst.prim*].*urm*].*urm* - adresa celui de-al treilea nod (pointer la nodul cu 99)
[[[*lst.prim*].*urm*].*urm*].*elem* - elementul din al 3-lea nod (valoarea 99)
[[[*lst.prim*].*urm*].*urm*].*urm* - adresa celui de-al patrulea nod. Din moment ce nu există decât 3 noduri, această adresă este NIL
[[[[*lst.prim*].*urm*].*urm*].*urm*].*elem* - teoretic al 4-lea element. Dacă sunt doar 3 elemente, eroare.

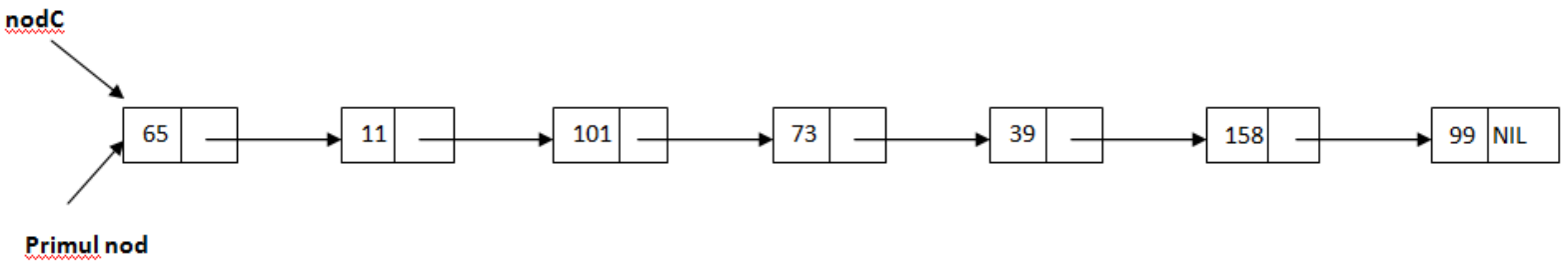
LSI - parcurgere

- Evident, pentru a parcurge o listă întreagă ne trebuie o metodă mai generală decât .urm.urm.urm...
- Vom considera o variabilă de tip pointer la nod, și cu această variabilă vom parcurge lista.

```
//presupunem că suntem în clasa Listă
nodC: ↑ Nod //nodC va reține adresa unui nod
nodC = this.prim //nodC reține primul nod din lst
cât timp nodC ≠ NULL execută
    elem = [nodC].elem //elementul din nodul curent
    @facem ceva cu elem
    nodC = [nodC].urm //trecem cu variabila nodC la nodul următor
sf_cât timp
```

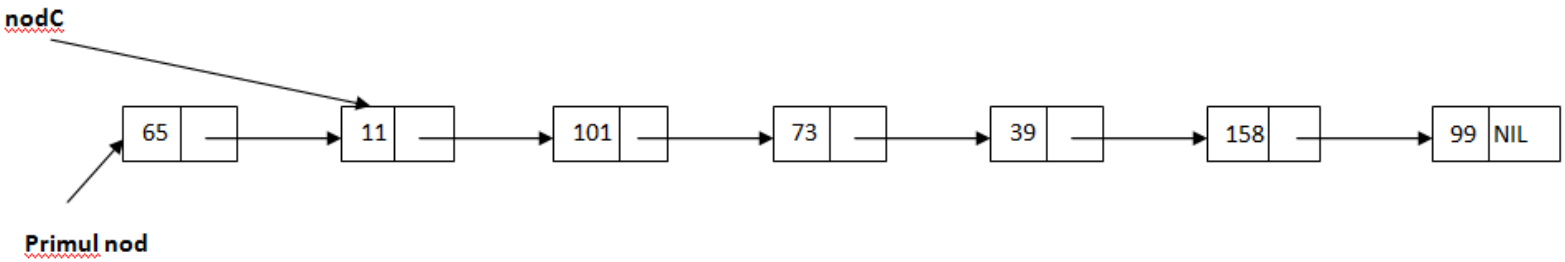
LSI - parcurgere II

- Inițial, variabila *nodC* reține adresa primului nod din listă



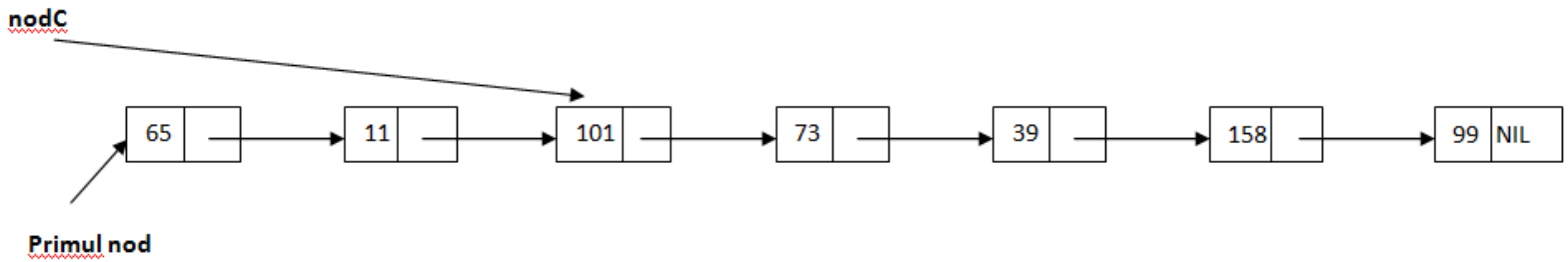
LSI - parcurgere III

- După prima execuție a instrucțiunii: *nodC = [nodC].urm*



LSI - parcurgere IV

- După a doua execuție a instrucțiunii: *nodC = [nodC].urm*



- Și tot așa mai departe, până ajungem la ultimul element, care are ca *urm* valoarea *NIL*. Când facem *nodC = [nodC].urm*, *nodC* devine *NIL*. Acum am terminat de parcurs toate elementele.

LSI - adăugare la început

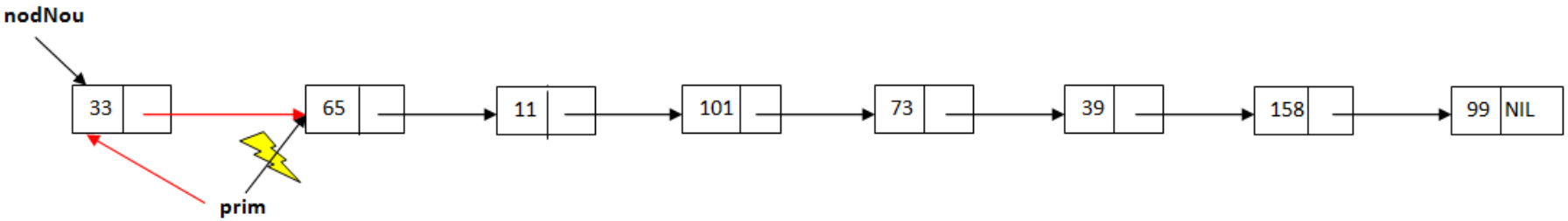
- Să presupunem că vrem să inserăm un element nou la începutul unei liste simplu înlănțuite.
- De fiecare dată când inserăm un element nou, trebuie să creăm un nod nou în care să punem elementul, după care trebuie să setăm niște legături ca nodurile să vină în ordinea în care vrem.
- La lista înlănțuită adăgarea sau ștergerea unui element tot timpul implică doar setarea unor legături, niciodată nu vom *muta* elementele dintr-un nod în altul (sau nu vom muta un nod la o altă adresă).

LSI - adăugare la început

- Dacă pun element la începutul listei, atunci următorul elementului nou va fi prim-ul de până acum, iar prim-ul listei va fi elementul nou.
- Trebuie să considerăm și cazul când lista nu conține nici-un element. În acest caz *lst.prim* are valoarea NIL și nodul de inserat va fi primul nod.

LSI - adăugare la început

- Adăugarea elementului 33 la începutul listei simple înlănțuite
- Vom crea un nod nou (numit *nodNou*) în care punem valoarea 33.
- Cu roșu sunt marcate legăturile noi care trebuie create la adăugare, iar cu fulger galben este marcată legătura care va fi desfăcută.



LSI - adăugare la început

```
subalgoritm adaugaInceput( el: TElem) este
    nodNou: ↑ Nod //creăm un nod nou...
    [nodNou].elem = el //...și îi setăm câmpurile
    [nodNou].urm = NIL
    dacă this.prim == NIL atunci //lista e vidă
        this.prim = nodNou
    altfel
        [nodNou].urm = this.prim
        this.prim = nodNou
    sf_dacă
sf_subalgoritm
```

- Complexitate:

LSI - adăugare la început

```
subalgoritm adaugaInceput( el: TElem) este
    nodNou: ↑ Nod //creăm un nod nou...
    [nodNou].elem = el //...și îi setăm câmpurile
    [nodNou].urm = NIL
    dacă this.prim == NIL atunci //lista e vidă
        this.prim = nodNou
    altfel
        [nodNou].urm = this.prim
        this.prim = nodNou
    sf_dacă
sf_subalgoritm
```

- Complexitate: $\Theta(1)$

LSI - adăugare la sfârșit I

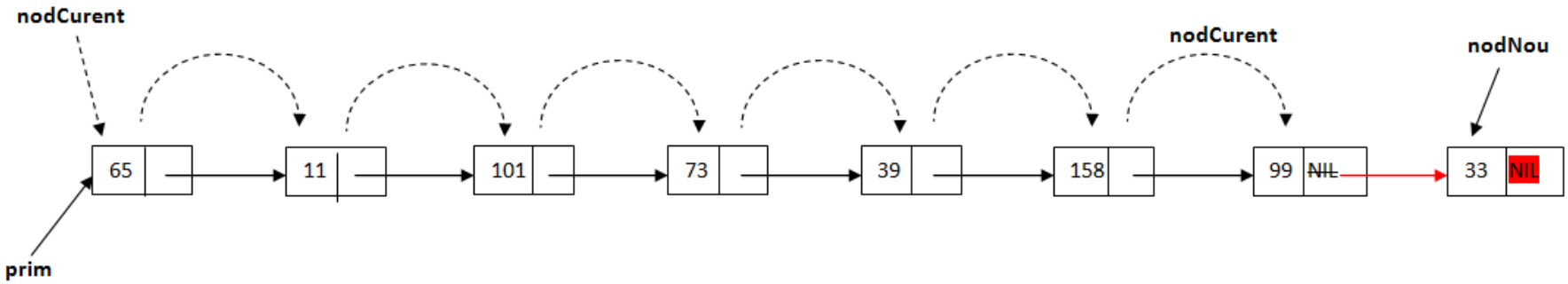
- Să adăugăm un element la sfârșitul unei liste simplu înlănțuite.

LSI - adăugare la sfârșit I

- Să adăugăm un element la sfârșitul unei liste simplu înlănțuite.
- Trebuie să parcurgem lista, nod cu nod, până ajungem la ultimul nod (cel care are câmpul *urm* egal cu NIL). Dacă suntem la ultimul nod, legăm câmpul urm de nodul pe care vrem să îl inserăm.

LSI - adăugare la sfârșit II

- Adăugarea elementului 33 la sfârșitul listei simple înlănțuite
- Vom crea un nod nou (numit *nodNou*) în care punem valoarea 33.
- Avem nevoie de *nodCurent* cu care parcurgem nodurile până ajungem la ultimul nod.
- Linia punctată arată parcurgerea cu *nodCurent*, iar cu roșu sunt marcate legăturile noi care trebuie create.



LSI - adăugare la sfârșit III

```
subalgoritm adaugaSfarsit(el: TElem) este
    nodNou: ↑ Nod //creăm un nod nou...
    [nodNou].elem = el //...și îi setăm câmpurile
    [nodNou].urm = NIL
    dacă this.prim == NIL atunci //lista e vidă
        this.prim = nodNou
    altfel
        nodCurent: ↑ Nod //o variabilă cu care vom parcurge lista
        nodCurent = this.prim
        câttimp [nodCurent].urm ≠ NIL execută
            nodCurent = [nodCurent].urm
        sf_câttimp //nodCurent acum e ultimul nod
        [nodCurent].urm = nodNou
    sf_dacă
sf_subalgoritm
```

LSI -adăugare la sfârșit IV

- Complexitate:

LSI -adăugare la sfârșit IV

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

LSI -adăugare pe poziție I

- Să adăugăm un element pe o anumită poziție în lista simplu înlănțuită.

LSI -adăugare pe poziție I

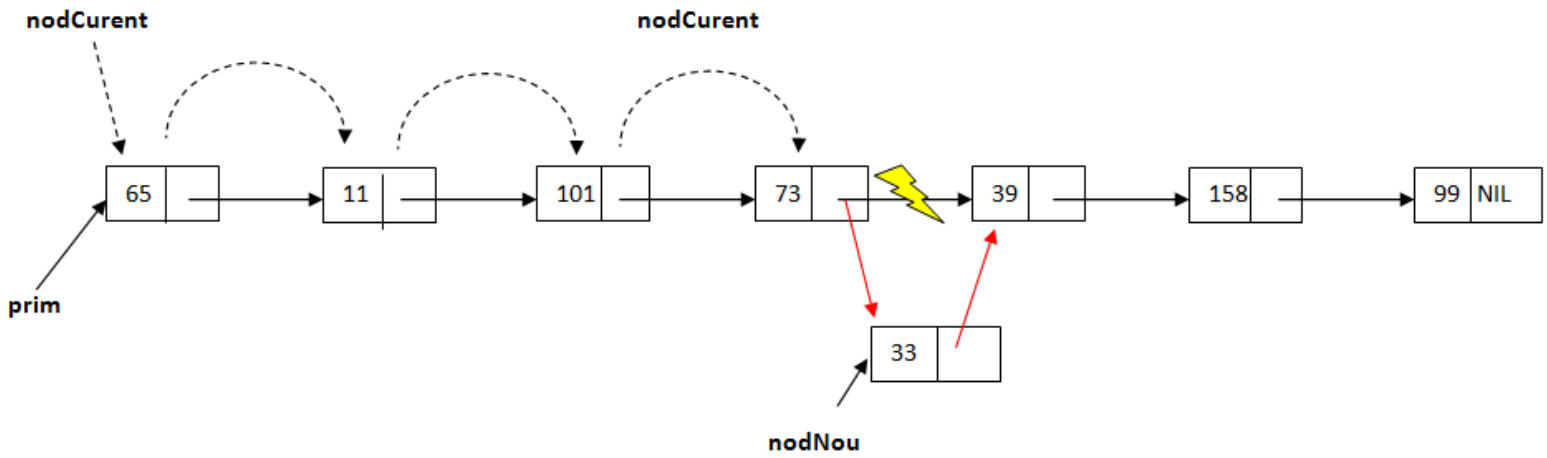
- Să adăugăm un element pe o anumită poziție în lista simplu înlănțuită.
- Prima dată trebuie să găsim poziția. Plecând de la primul element, parcurgem lista până ajungem la poziția cerută. Ne oprim când găsim nodul *după care trebuie să inserăm*, altfel nu vom putea seta legăturile în mod corect. Tratăm separat cazul când elementul trebuie inserat pe prima poziție (în acest caz nu exista nod după care să inserăm). (cazul când vrem să inserăm după ultimul nod va fi acoperit de cazul general - există nod după care inserez).

LSI - adăugare pe poziție II

- La acele operații care lucrează cu poziții, trebuie să verificăm și dacă poziția e corectă.
- La Vector Dinamic aveam câmpul *len*, pe care l-am folosit să verificăm dacă o poziție este validă.
- Pentru a simplifica lucrurile, putem introduce și la ListaSimpluÎnlănțuită câmpul lungime.
- Dacă nu avem câmp lungime, în timp ce parcurgem lista trebuie să verificăm și dacă am ieșit din listă.
- Deocamdată presupunem reprezentare fără câmpul lungime.

LSI - adăugare pe poziție III

- Adăugarea elementului 33 pe poziția 4.
- Vom crea un nod nou (numit *nodNou*) în care punem valoarea 33.
- Avem nevoie de *nodCurent* cu care parcurgem nodurile până ajungem la elementul după care inserăm (elementul de pe poziția 3).
- Linia punctată arată parcurgerea cu *nodCurent*, cu roșu sunt marcate legăturile noi care trebuie create, iar fulgerul galben marchează legătura care va fi desfăcută.



subalgoritm adaugaPozitie(el: TElem, poz: întreg) **este**
  dacă *poz* $<$ 0 **atunci**
    @aruncă excepție, poziția nu poate fi negativă
  sf_dacă
  nodNou: ↑ Nod *//creăm un nod nou...*
  [nodNou].elem = el *//...și îi setăm câmpurile*
  [nodNou].urm = NIL
  dacă poz == 0 **atunci** *//vrem să inserăm pe prima poziție*
    [nodNou].urm = this.prim
    this.prim = nodNou
  altfel
    nodCurent: ↑ Nod *//o variabilă cu care vom parcurge lista*
    pozCurent : Întreg *//pentru a număra pozițiile*
    pozCurent = 0
    nodCurent = this.prim
    câttimp pozCurent $<$ poz-1 **SI** nodCurent $\neq$ NIL **execută**
      nodCurent = [nodCurent].urm
      pozCurent = pozCurent + 1
    sf_câttimp *//continuăm pe pagina următoare*

```
    dacă pozCurent == poz - 1 ȘI nodCurent ≠ NIL atunci
//pozitia e validă
        //nodCurent este nodul după care inserăm
        //prima dată setăm legătura noului nod
         [nodNou].urm = [nodCurent].urm
         [nodCurent].urm = nodNou
    sf_dacă
  sf_dacă
sf_subalgoritm
```

LSI - adăugare pe poziție IV

- Complexitate:

LSI - adăugare pe poziție IV

- Complexitate: $O(n)$

LSI - adăugare pe poziție V

- Ce se modifică dacă am câmpul lungime în reprezentarea listei?

LSI - adăugare pe poziție V

- Ce se modifică dacă am câmpul lungime în reprezentarea listei?
- Putem verifica la început direct daca poziția primită ca parametru e o poziție validă
- Dacă știm că poziția este validă, nu ne mai trebuie condiția în ciclu *while* cu *nodCurent* diferit de NIL.
- De fapt, putem folosi un ciclu *for* în loc de ciclu *while*.

```
subalgoritm adaugaPozitie(el:TElem, poz:întreg) este
    dacă poz < 0 SAU poz > lungime atunci
        @aruncă excepție, poziție invalidă
    sf_dacă
     nodNou: ↑ Nod //creăm un nod nou...
     [nodNou].elem = el //...și îi setăm câmpurile
     [nodNou].urm = NIL
     dacă poz == 0 atunci //vrem să inserăm pe prima poziție
         [nodNou].urm = this.prim
         this.prim = nodNou
         this.lungime = this.lungime+ 1
    altfel
         nodCurent: ↑ Nod //o variabilă cu care vom parcurge lista
         nodCurent = this.prim
        pentru i = 0, poz-1, 1 execută
            nodCurent = [nodCurent].urm
        sf_pentru //continuăm pe pagina următoare
```

```
    //nodCurent este nodul după care inserăm
    //prima dată setăm legătura noului nod
     [nodNou].urm = [nodCurent].urm
     [nodCurent].urm = nodNou
    this.lungime = this.lungime + 1
  sf_dacă
sf_subalgoritm
```