

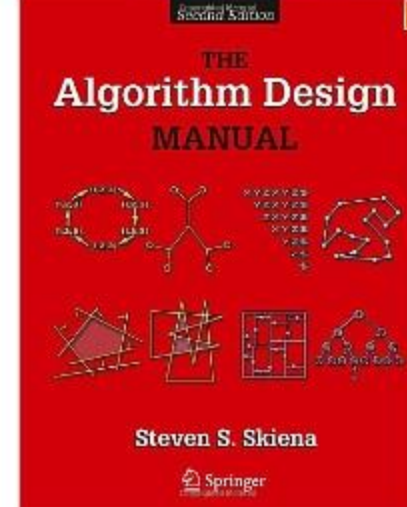
CURS 11:

Structuri liniare de date (II)

Motivatie

S. Skiena – The Algorithm Design Manual

<http://sist.sysu.edu.cn/~isslxm/DSA/textbook/Skiena.-TheAlgorithmDesignManual.pdf>



Interview Problems

3-27. [5] Determine whether a linked list contains a loop as quickly as possible without using any extra storage. Also, identify the location of the loop.

Se consideră două polinoame (fiecare cu câte 3 termeni) :

$$P[X]=3X^{100}-2X^2+1$$

$$Q[X]=2X^{50}+X^2-3X$$

și se dorește calculul sumei $S[X]=P[X]+Q[X]$

Propuneți o variantă de reprezentare eficientă atât în ce privește spațiul de memorie cât și volumul de calcul

Reminder: tipuri abstracte de date

Tip abstract de date = set de date asupra căruia se pot efectua operații specifice

Nivel: utilizare

Exemple: **lista (list)**, **stiva (stack)**, **coada (queue)**, movila (heap), dicționar (dictionary), tabel de dispersie (hash table), arbori de căutare

Reprezentare abstractă = modalitate de organizare a elementelor setului de date care permite efectuarea eficientă a operațiilor specifice

Nivel: proiectare

Exemple: **structură liniară**, structură arborescentă

Implementare = specificarea structurii de date folosind tipuri de date și construcții specifice unui limbaj de programare

Nivel: implementare

Exemple: tablouri (arrays), **liste înlanțuite (linked lists)**

Reminder: structuri liniare

Operații specifice listelor: **interogare = căutarea unui element**

Implementare: utilizând **tablouri**

- După poziție

- Elementul aflat pe o anumită poziție ($\Theta(1)$)
- Elementul următor/ anterior unui element specificat (element curent) ($\Theta(1)$)

- După valoare

- Elementul care conține o valoare specificată ($O(n)$)
- Elementul care conține cea mai mică/ mare valoare ($\Theta(n)$)

Reminder: structuri liniare

Operații specifice listelor: **modificare**

Implementare: utilizând **tablouri**

- Inserarea unui element

- La început ($\Theta(n)$)
- La sfârșit ($\Theta(1)$)
- Pe o poziție specificată relativ la alte elemente (înainte sau după un element dat) ($\Theta(n)$)

- Eliminarea (ștergerea) unui element

- De la început ($\Theta(n)$)
- De la sfârșit ($\Theta(1)$)
- De pe o poziție specificată relativ la alte elemente (înainte sau după un element dat) ($\Theta(n)$)

Reminder: structuri liniare

Exemplu: inserarea lui X după C

Zona contiguă (tablou)

A	B	C	D	E	F	
---	---	---	---	---	---	--



A	B	C		D	E	F
---	---	---	--	---	---	---

A	B	C	X	D	E	F
---	---	---	---	---	---	---

Cost: $O(n)$ transferuri

Obs. caz defavorabil: inserare la început

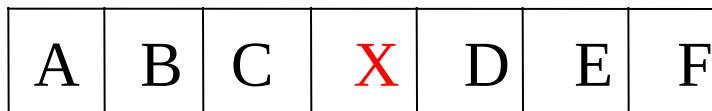
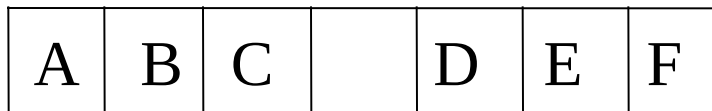
Ce se întâmplă dacă se dorește adăugarea unui element nou la lista și spațiul alocat inițial a fost utilizat în întregime?

- Se alocă altă zonă de dimensiune mai mare (de exemplu dublul dimensiunii curente a tabloului) unde sunt transferate elementele listei
- E abordarea specifică **tablourilor dinamice**
- Cost total extindere: $\Theta(n)$ (transferul celor n elemente)
- Cost extindere/element: $\Theta(1)$ (analiza amortizată)

Reminder: structuri liniare

Exemplu: inserarea lui X după C

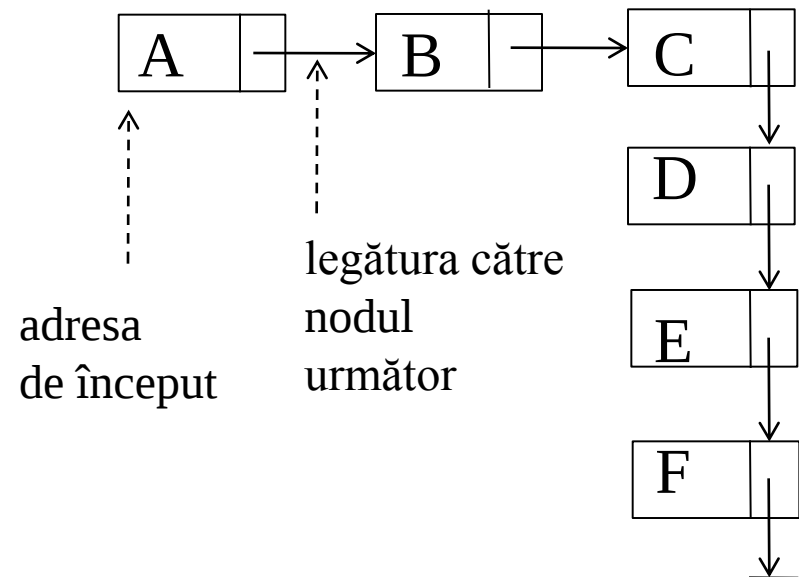
Zona contiguă (tablou)



Cost: $O(n)$ transferuri

Obs. caz defavorabil: inserare la început

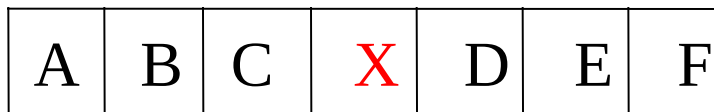
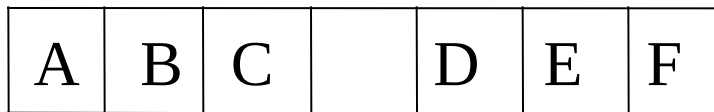
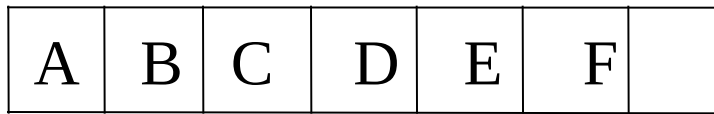
Altă variantă de implementare:
structură înlănțuită - relația „succesor direct” este implementată prin **legături** (referințe) între **noduri**



Reminder: lista

Exemplu: inserarea lui X după C

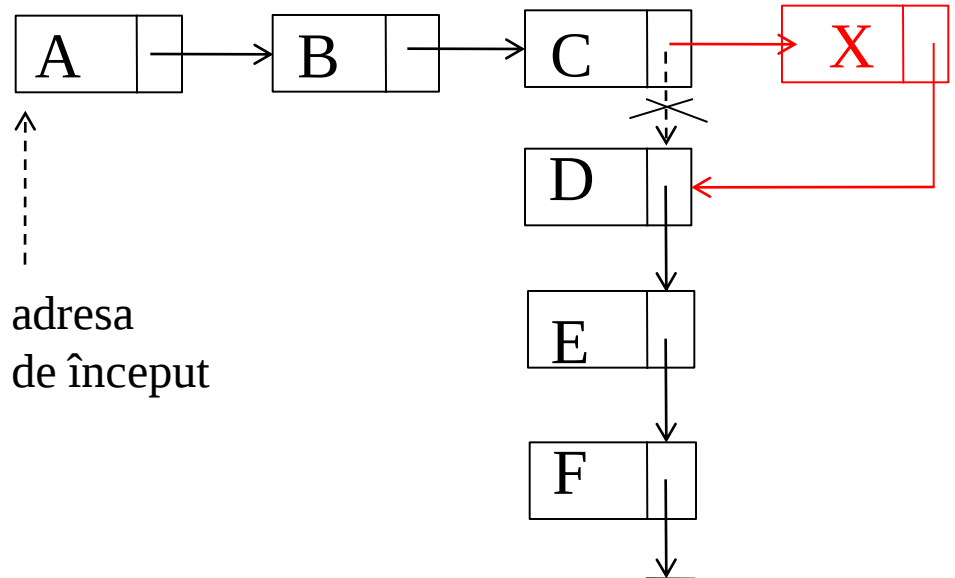
Zona contiguă (tablou)



Cost: $O(n)$ transferuri

Obs. caz defavorabil: inserare la început

Structură înlănțuită (relația „succesor direct” este implementată prin **legături** (referințe) între **noduri**) permite o implementare mai eficientă

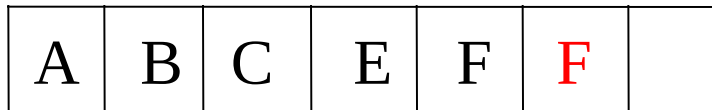
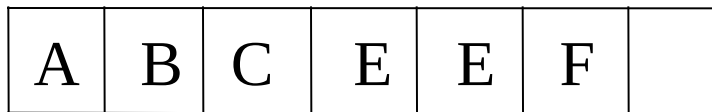


Cost: $\Theta(1)$ indiferent de poziția de inserare (crearea unui nod și modificarea unei referințe)

Reminder: lista

Exemplu: eliminarea lui D

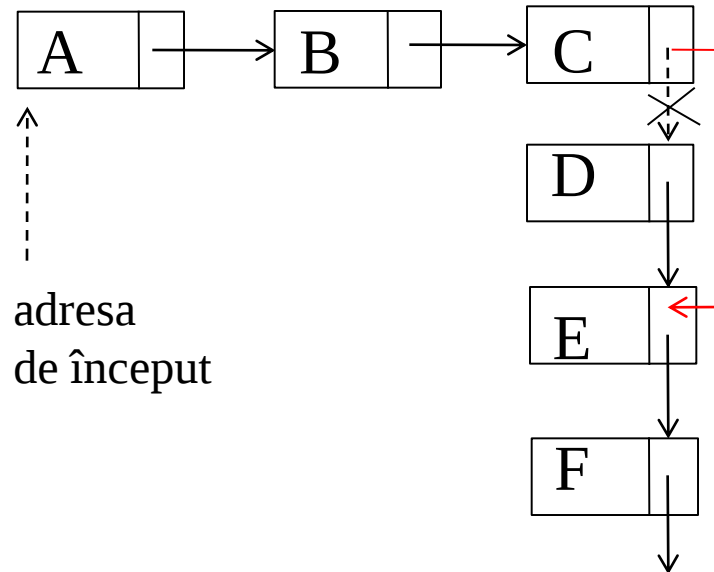
Zona contiguă (tablou)



Cost: $O(n)$ transferuri

Obs. caz defavorabil: eliminarea primului element

Structură înlănțuită (relația „succesor direct” este implementată prin **legături** (referințe) între **noduri**) permite o implementare mai eficientă



Cost: $\Theta(1)$ - indiferent de poziția nodului după care se șterge (modificarea unei referințe)

Reprezentarea structurilor înlănțuite

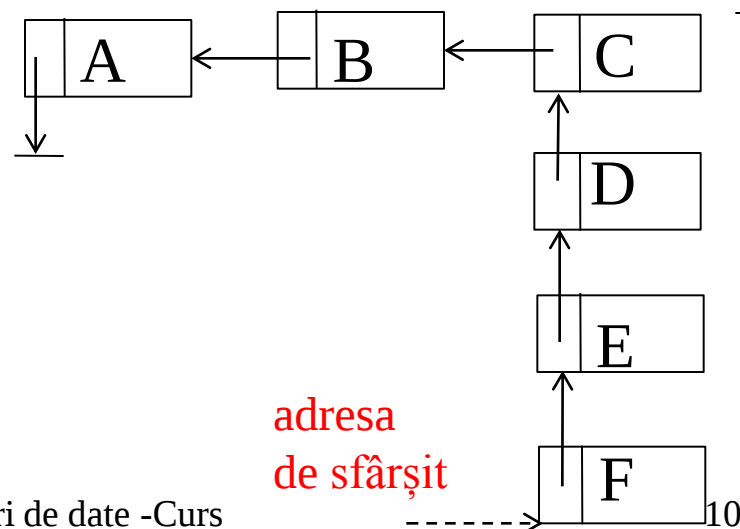
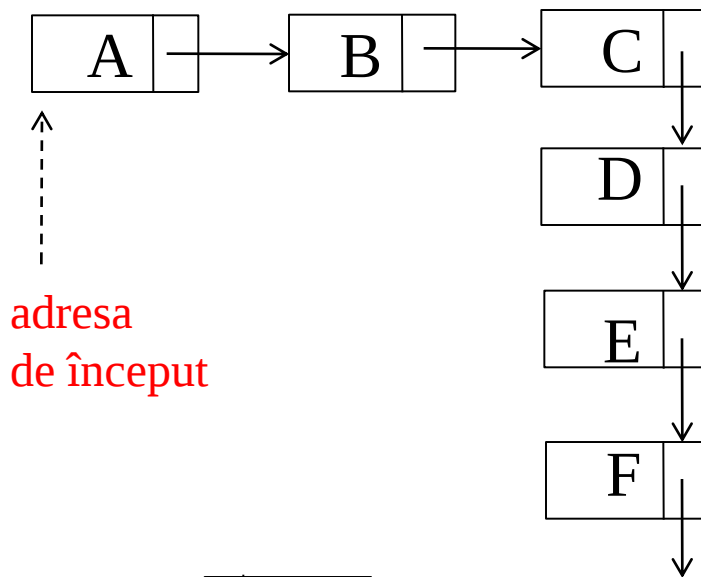
Ce ar trebui să cunoaștem ca să putem parcurge o structură înlănțuită?

Adresa de început – în ipoteza că fiecare nod conține referința către următorul nod

- Listă simplu înlănțuită care permite parcurgerea elementelor de la primul la ultimul

Adresa de sfârșit – în ipoteza că fiecare nod conține referința către nodul anterior

- Listă simplu înlănțuită care permite parcurgerea elementelor de la ultimul element la primul



Reprezentarea structurilor înlănțuite

Ce ar trebui să cunoaștem ca să putem parcurge o structură înlănțuită?

Adresa de început – în ipoteza că fiecare nod conține referința către următorul nod

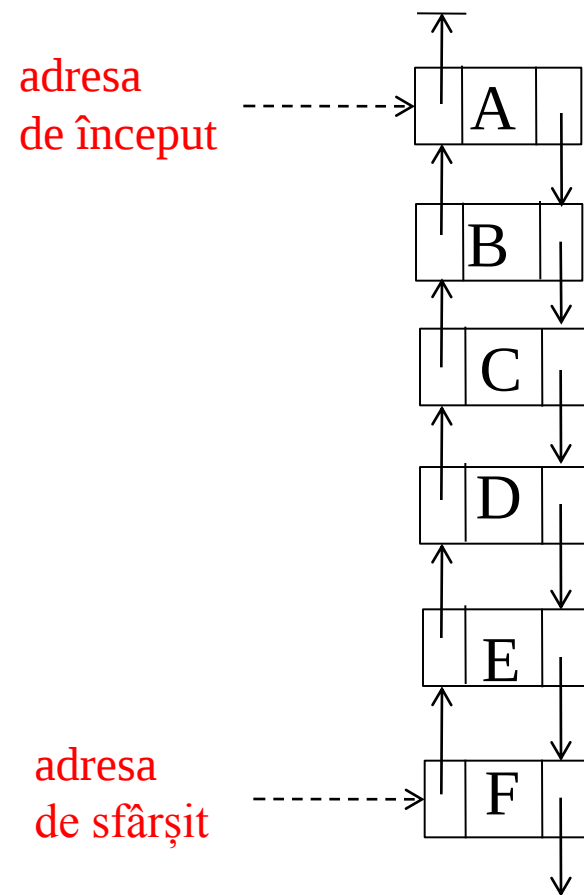
- Listă simplu înlănțuită care permite parcurgerea elementelor de la primul la ultimul

Adresa de sfârșit – în ipoteza că fiecare nod conține referința către nodul anterior

- Listă simplu înlănțuită care permite parcurgerea elementelor de la ultimul element la primul

Ambele adrese + referințe în fiecare nod către nodul anterior și către nodul următor

- **Lista dublu înlănțuită** care permite parcurgerea în ambele sensuri



Implementarea structurilor înlănțuite

Obs. lista L este constituită din noduri cu aceeași structură:

- Informația utilă (valorile stocate în listă)
- Referințe către:
 - Nodul anterior
 - Nodul următor

Fiecare nod este identificat prin **referința** către el (**ref**).

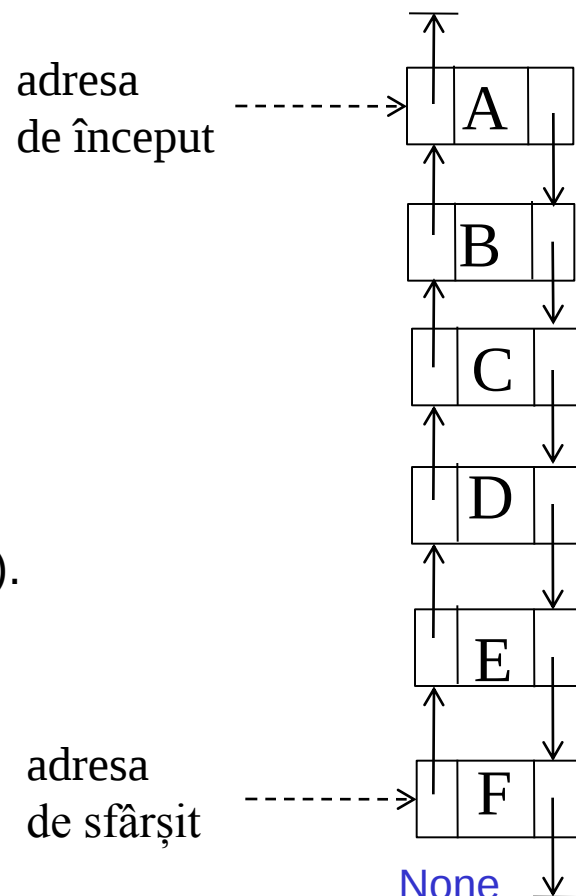
Exemplu descriere în pseudocod:

ref.info (informația utilă din nodul referit prin **ref**)

ref.ant (referința către nodul anterior)

ref.urm (referința către nodul următor)

Pentru specificarea unei liste simplu înlănțuite este suficientă referința către primul element (**L.inceput**)



Obs: **None** reprezintă **referința vidă**

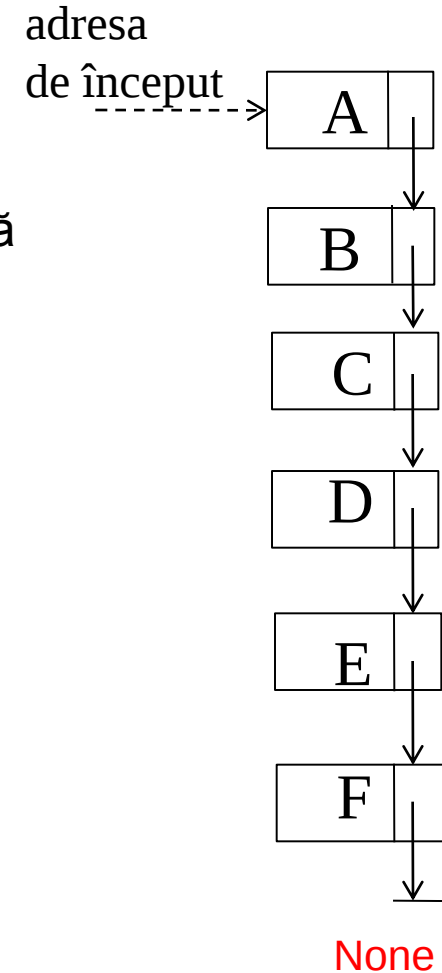
(nu există nod către care să facă referire)

Operații cu liste simplu înlănțuite

Determinarea nodului care conține o anumită informație

```
caut(L,x) // inceput = referință către primul element din listă
  ref ← L.inceput
  while (ref != None) and (ref.info != x) do
    ref ← ref.urm // trecerea la urmatorul element din lista
  endwhile
  return ref
```

Cost: $O(n)$ (n e numărul de elemente din listă)

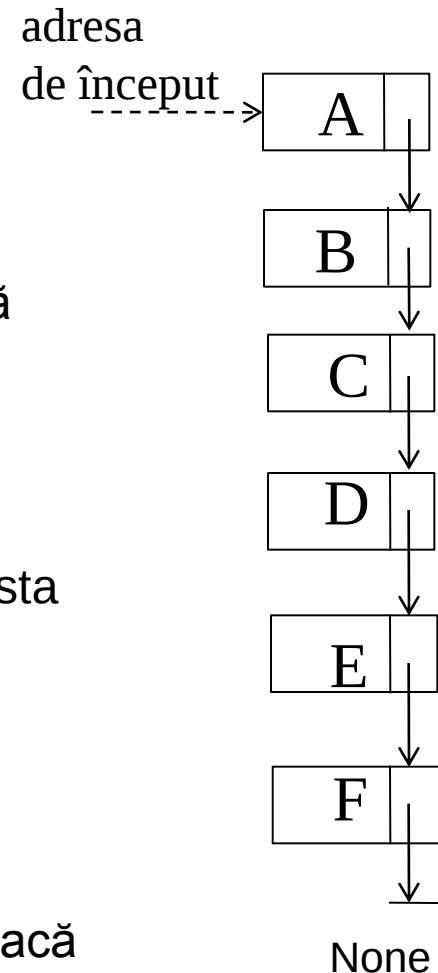


Operații cu liste simplu înlănțuite

Determinarea nodului corespunzător unei poziții
(al k-lea nod din listă)

```
caut(L,k) // inceput = referință către primul element din listă
  ref ← L.inceput
  i ← 1
  while (ref != None) and (i < k) do
    ref ← ref.urm    // trecerea la urmatorul element din lista
    i ← i + 1
  endwhile
  return ref
```

Cost: $O(k)$ (Obs: căutarea se poate termina mai repede dacă lista nu are k elemente)

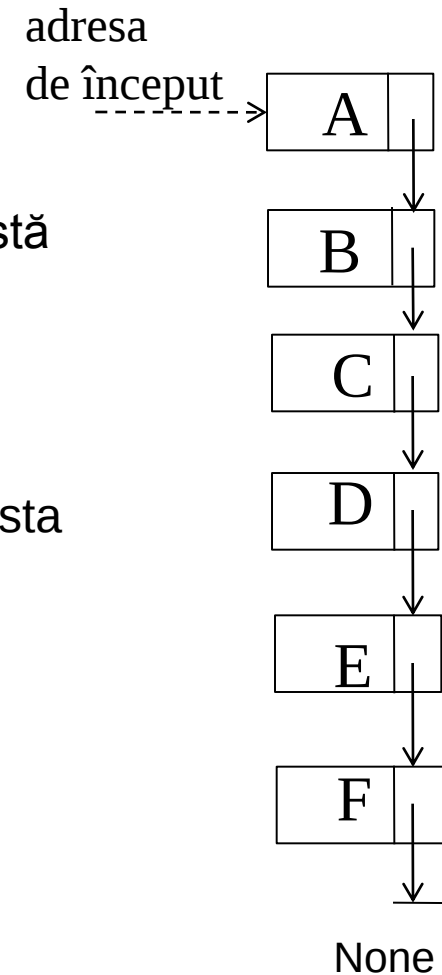


Operații cu liste simplu înlanțuite

Parcurgerea unei liste

```
parcure(L) // inceput = referință către primul element din listă
  ref ← L.inceput
  while (ref != None)
    < prelucrare element specificat prin ref >
    ref ← ref.urm // trecerea la următorul element din listă
  endwhile
```

Cost: $\Theta(n)$ (n e numărul de elemente din listă)

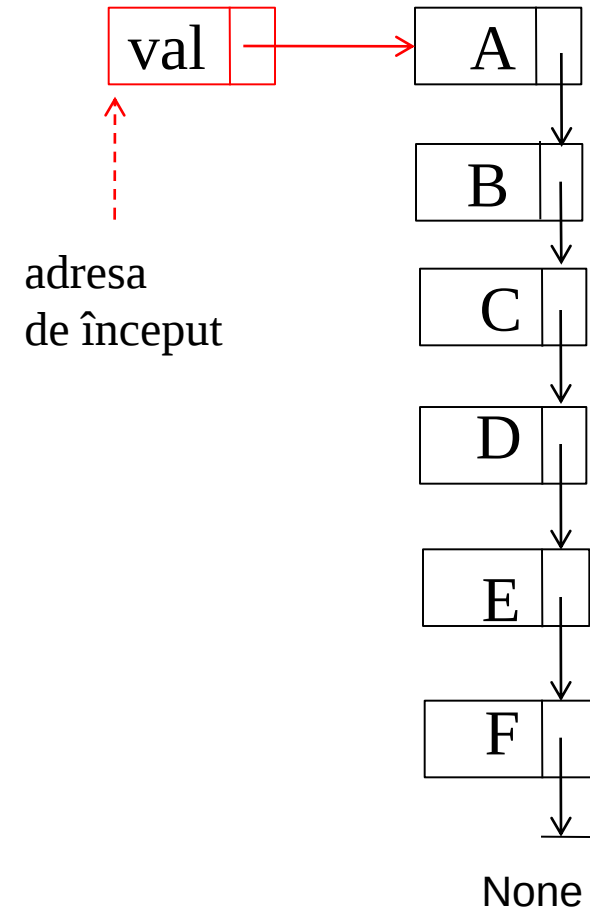


Operații cu liste simplu înlanțuite

Adăugare la începutul listei

```
adaugInceput(L,val)
  ref ← <creare nod nou>
  ref.info ← val
  ref.urm ← L.inceput
  L.inceput ← ref
  return L
```

Cost: $\Theta(1)$



Operații cu liste simplu înlănțuite

Insertie după elementul referit prin adr

inserareDupa(adr,val)

ref ← <creare nod nou>

ref.info ← val

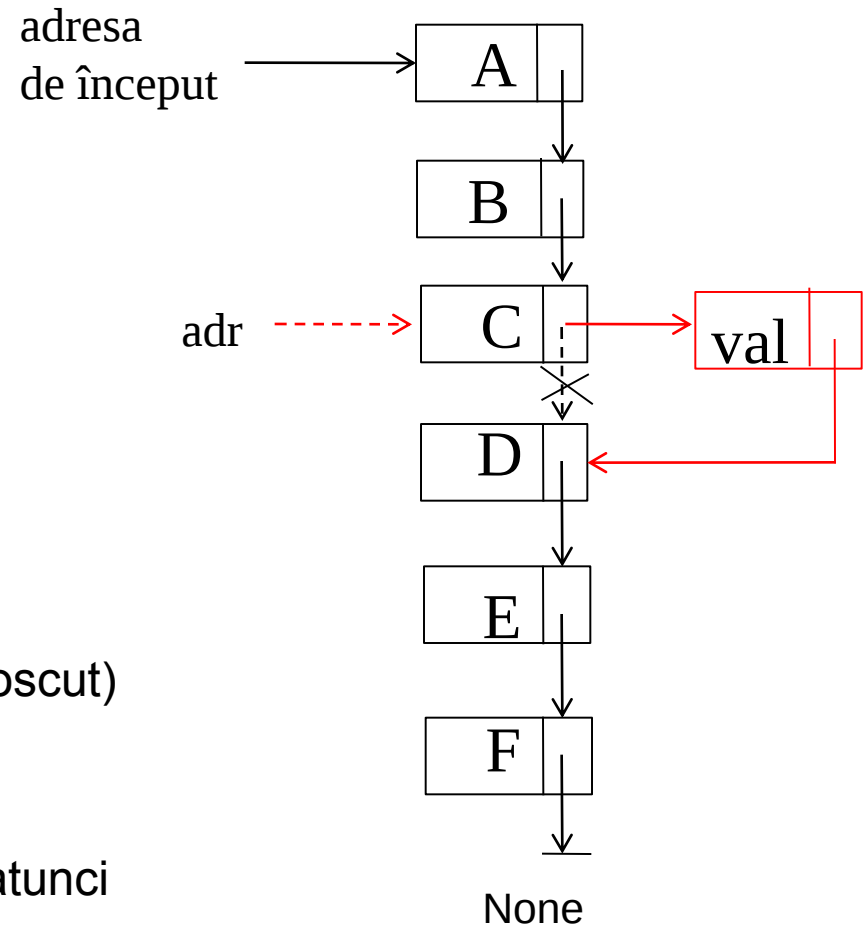
ref.urm ← adr.urm

adr.urm ← ref

return L

Cost: $\Theta(1)$ (cu condiția ca adr să fie cunoscut)

Obs: dacă se dorește inserarea după un element care conține o anumită valoare, atunci trebuie determinată prima dată adresa elementului ($O(n)$)



Operații cu liste simplu înlanțuite

Insertie înaintea elementului referit prin adr

inserareInainte(adr, val)

ref ← <creare nod nou>

ref.info ← adr.info // nodul de la adr se transfera la ref

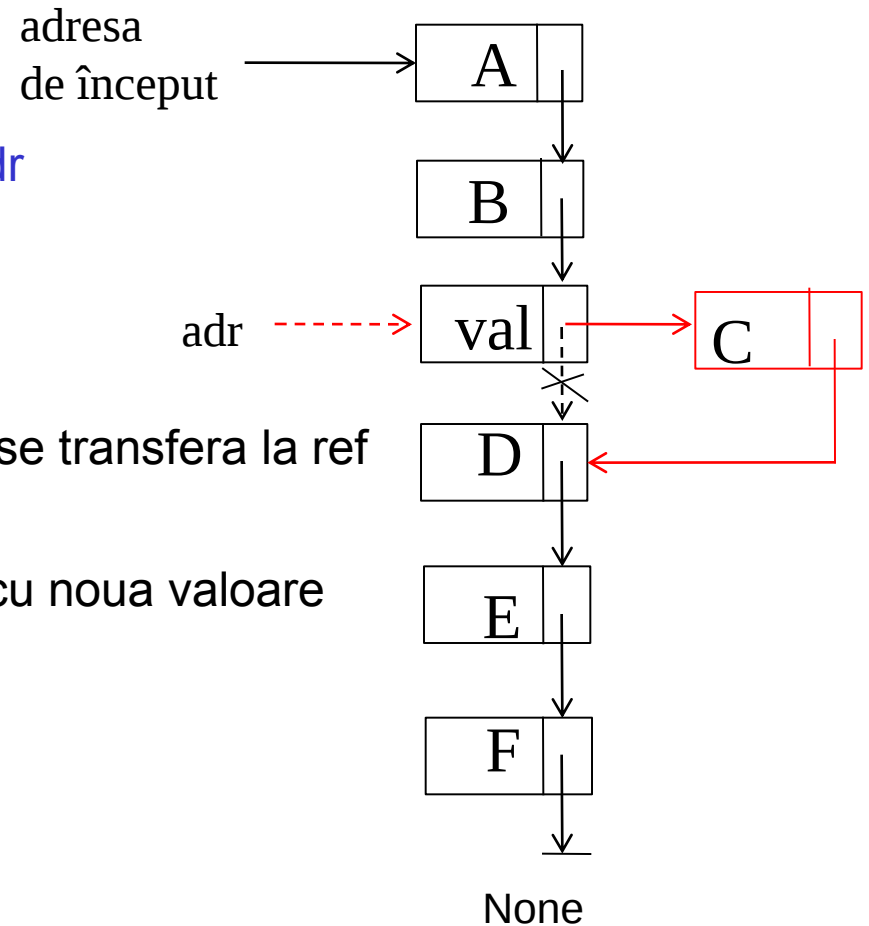
ref.urm ← adr.urm

adr.urm ← ref // adr se completeaza cu noua valoare

adr.info ← val

return L

Cost: $\Theta(1)$



Operații cu liste simplu înlanțuite

Stergerea (eliminarea) primului element

stergeInceput(L)

L.inceput $\leftarrow$ L.inceput.urm

return L

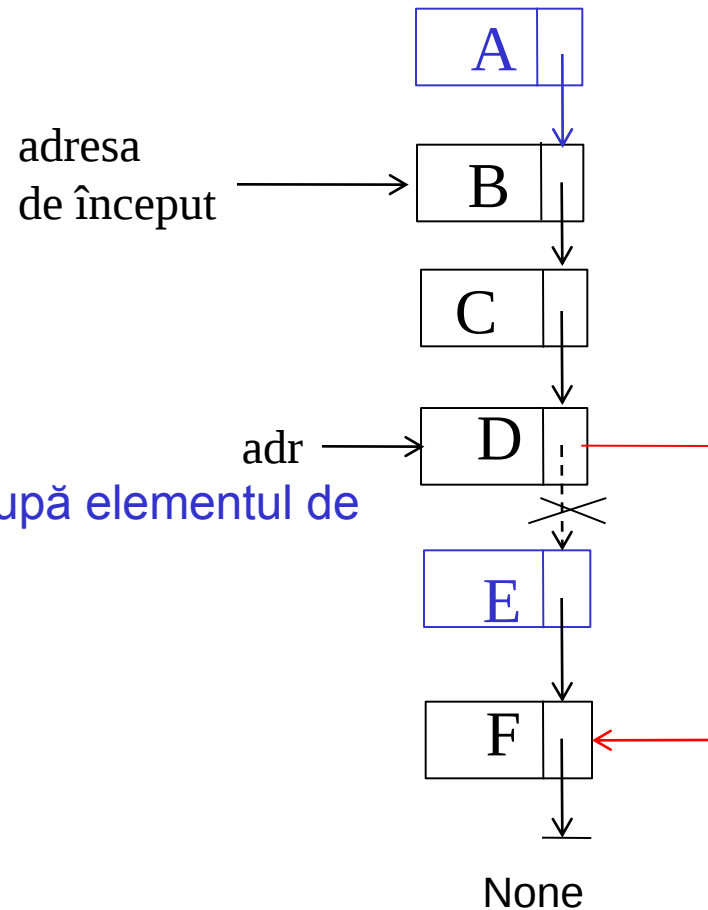
Stergerea (eliminarea) elementului aflat după elementul de la adresa adr

stergeDupa(L, adr)

adr.urm $\leftarrow$ adr.urm.urm

return L

Cost: $\Theta(1)$



Operații cu liste simplu înlanțuite

Stergerea (eliminarea) elementului de la adresa adr

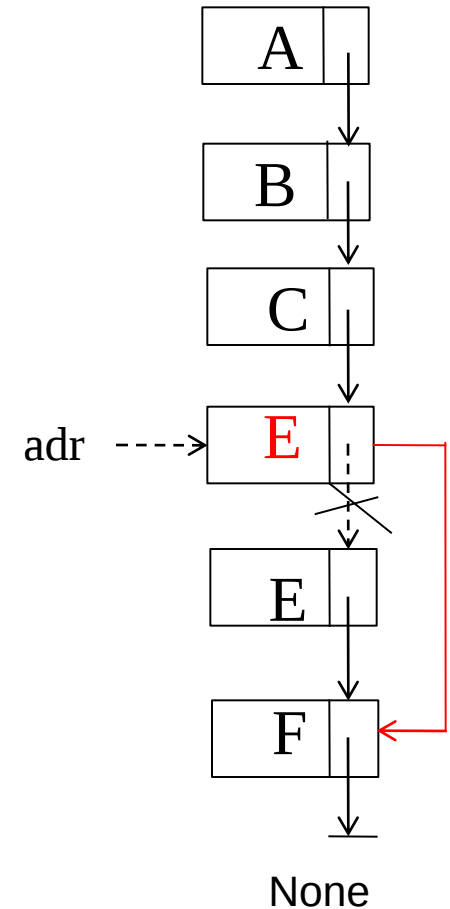
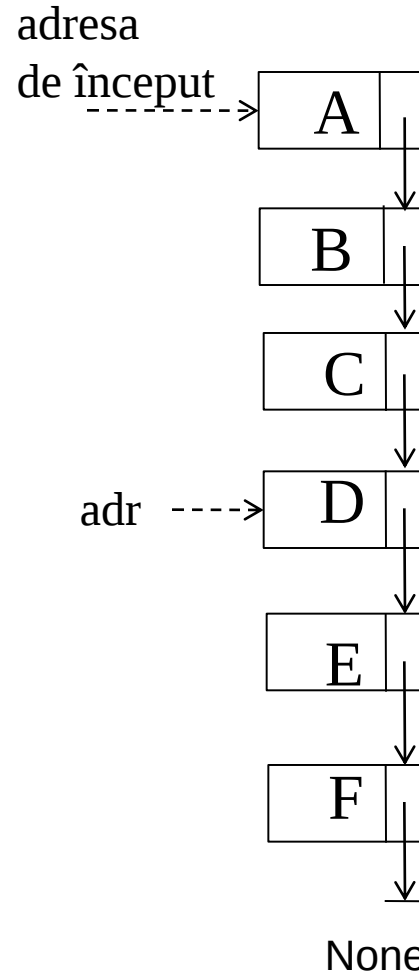
Sterge(L,adr)

adr.info ← adr.urm.info

adr.urm ← adr.urm.urm

return L

Cost: $\Theta(1)$



Exemplu implementare Python

```
class Nod:
    def __init__(self, val = None, urm = None):
        self.info = val
        self.urm = urm

class Lista:
    def __init__(self):
        self.inceput = None

    def adaugaInceput(self, x):
        if self.inceput == None:
            self.inceput = Nod(x, None)
        else:
            nodNou = Nod(x, self.inceput)
            self.inceput = nodNou

    def stergeInceput(self):
        self.inceput = self.inceput.urm
```

Exemplu implementare Python

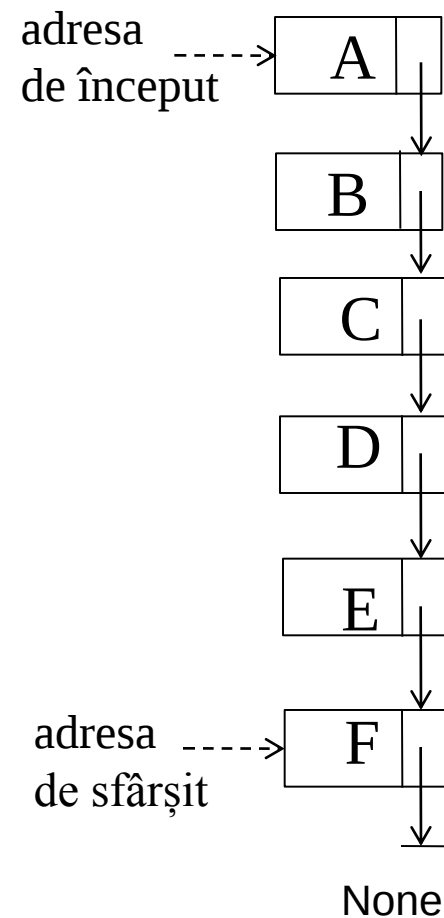
```
def parcurge(self):  
    ref=self.inceput  
    while (ref!=None):  
        print(ref.info)  
        ref=ref.urm
```

Crearea unei liste si efectuarea de operatii:

```
L=Lista() # lista vida  
L.adaugaInceput(1)  
    # lista are elementul 1  
L.adaugaInceput(2)  
    # lista are elementele 2,1  
L.adaugaInceput(3)  
    # lista are elementele 3,2,1  
L.stergeInceput()  
    # lista are elementele 2,1  
L.parcurge()  
    # afisarea elementelor listei
```

Liste simplu înlănțuite

- Care este **costul adăugării după ultimul element** într-o listă simplu înlănțuită (dacă se cunoaște doar adresa primului element din listă)?
 - Răspuns: $\Theta(n)$
- Cum putem reduce costul?
 - Răspuns: se reține și adresa ultimului element
- Care este **costul ștergerii ultimului element din listă** (dacă se cunosc adresele primului și ultimului element din listă)?
 - Răspuns: $\Theta(n)$
- Cum putem reduce costul?
 - Răspuns: se reține și adresa ultimului element și a penultimului => **liste dublu înlănțuite**



Liste dublu înlanțuite

- fiecare element conține o referință către elementul următor și una către elementul anterior (sunt implementate atât relația de precedență cât și cea de succesiune directă)

- Specificare în pseudocod a unei liste dublu înlanțuite:

L.inceput

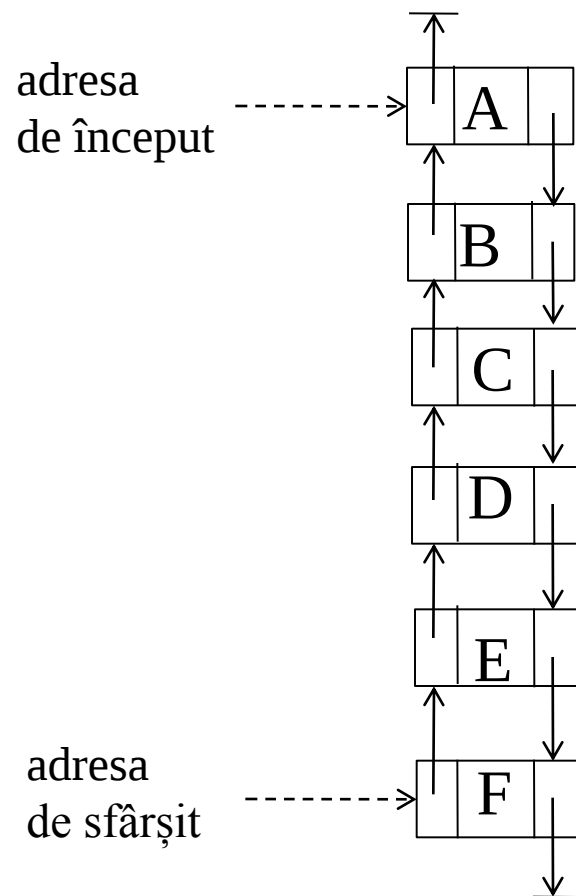
L.sfarsit

- Specificarea în pseudocod a câmpurilor unui nod din lista dublu înlanțuită (referit prin **ref**)

ref.info

ref.prec

ref.urm



Operații cu liste dublu înlanțuite

Adăugare la sfârșit

adaugLaSfarsit(L,val)

ref ← <creare nod nou>

ref.info ← val

ref.prec ← L.sfarsit

ref.urm ← None

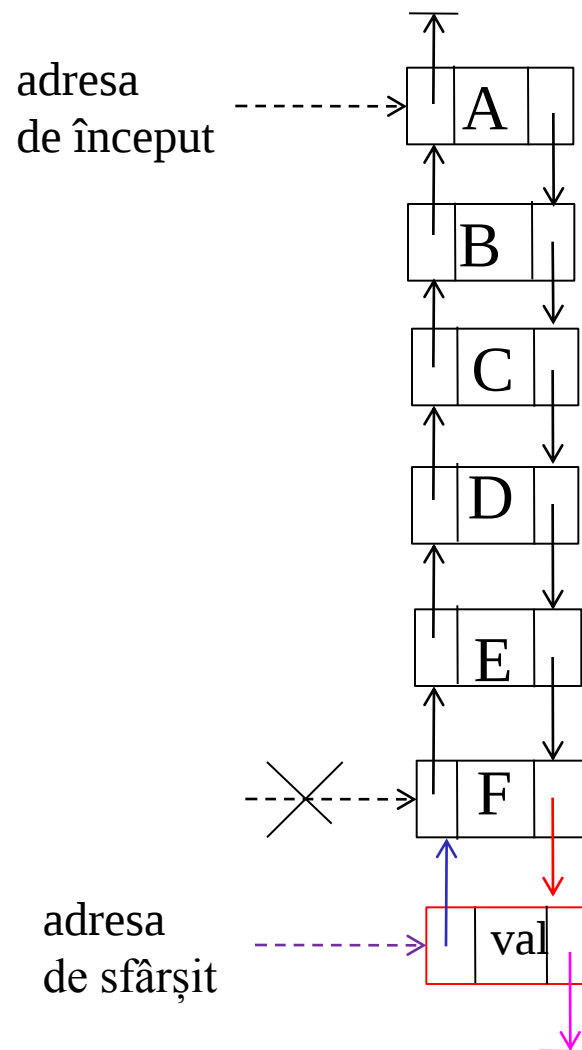
L.sfarsit.urm ← ref

L.sfarsit ← ref

return L

Cost: $\Theta(1)$

Obs: cazul în care lista este vidă
(L.inceput=L.sfarsit=None) trebuie tratat separat



Operații cu liste dublu înlănțuite

Adăugare după un nod specificat prin adresa sa

`inserareDupa(adr, val)`

`ref ← <creare nod nou>`

`ref.info ← val`

`ref.urm ← adr.urm`

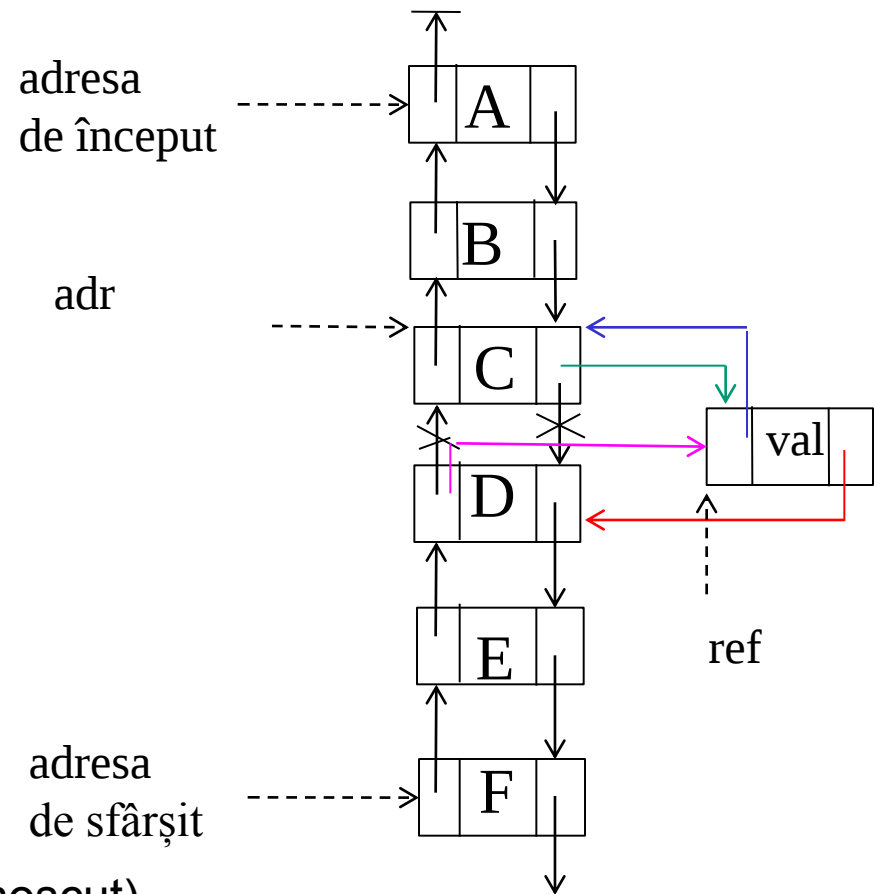
`ref.prec ← adr`

`adr.urm ← ref`

`ref.urm.prec ← ref`

Cost: $\Theta(1)$ (cu condiția ca `adr` să fie cunoscut)

Obs: dacă se dorește inserarea după un element care conține o anumită valoare, atunci trebuie determinată prima dată adresa elementului ($O(n)$)



Operații cu liste dublu înlănțuite

Adăugare înaintea unui nod specificat prin
adresa sa

inserareInainte(adr, val)

ref ← <creare nod nou>

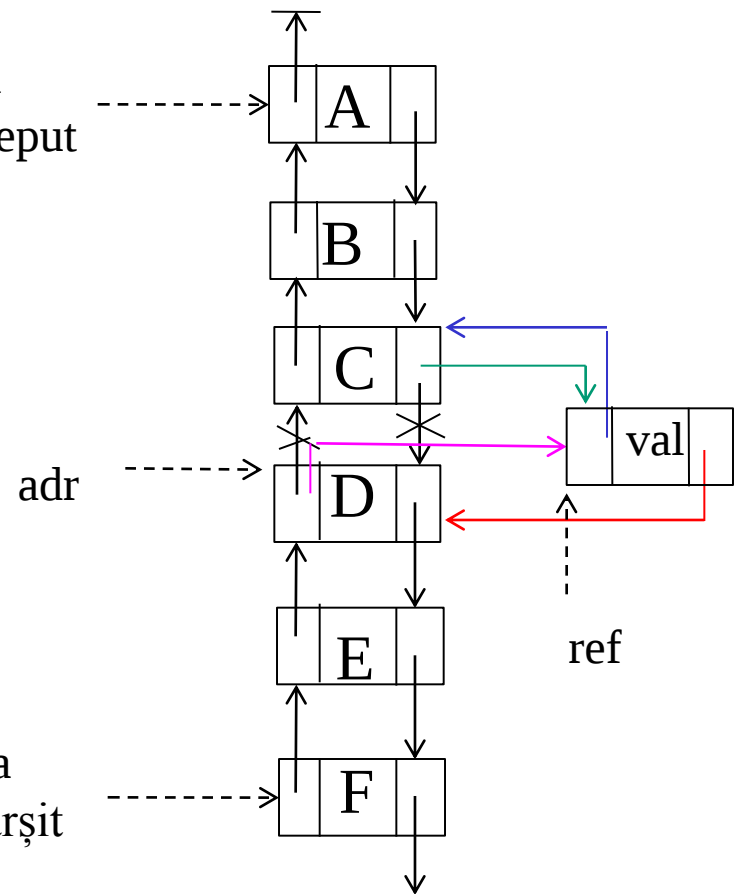
ref.info ← val

ref.prec ← adr.prec

ref.urm ← adr

adr.prec ← ref

ref.prec.urm ← ref



Cost: $\Theta(1)$ (cu condiția ca adr să fie cunoscut)

Operații cu liste dublu înlanțuite

Ștergerea ultimului/primului element

stergeUltimul(L)

$L.sfarsit \leftarrow L.sfarsit.prec$

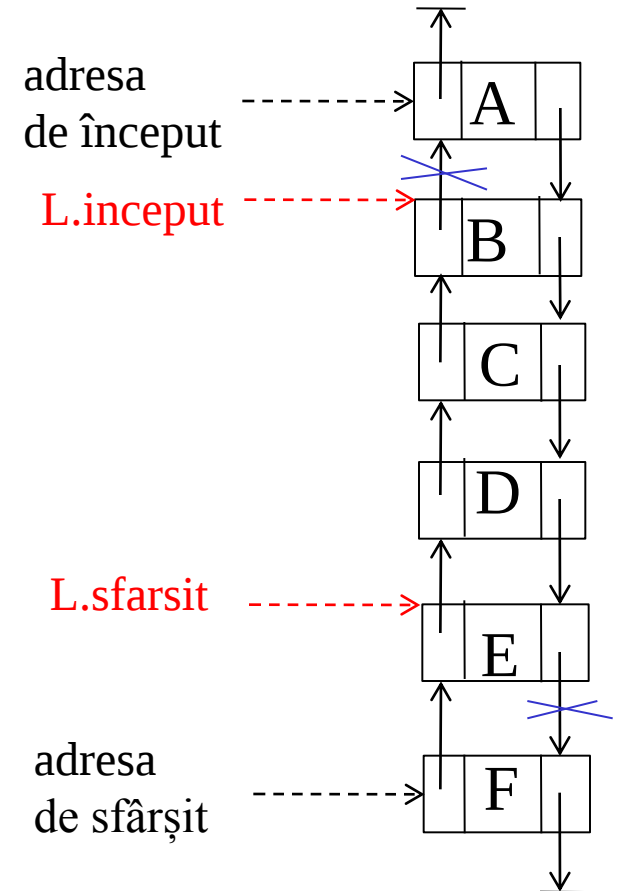
$L.sfarsit.urm \leftarrow None$

stergePrimul(L)

$L.inceput \leftarrow L.inceput.urm$

$L.inceput.prec \leftarrow None$

Cost: $\Theta(1)$



Operații cu liste dublu înlănțuite

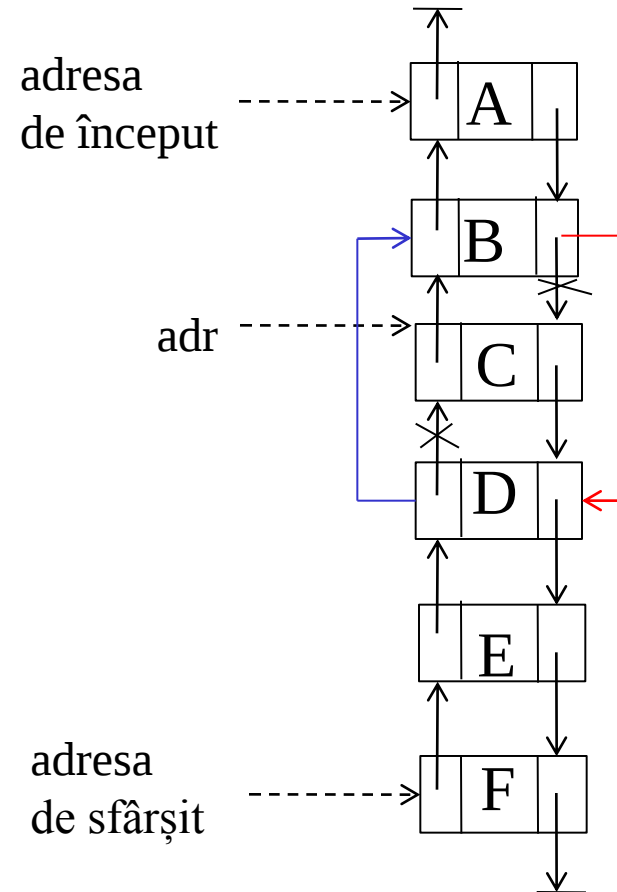
Ștergerea unui element specificat prin
adresa

sterge(adr)

$\text{adr.urm.prec} \leftarrow \text{adr.prec}$

$\text{adr.prec.urm} \leftarrow \text{adr.urm}$

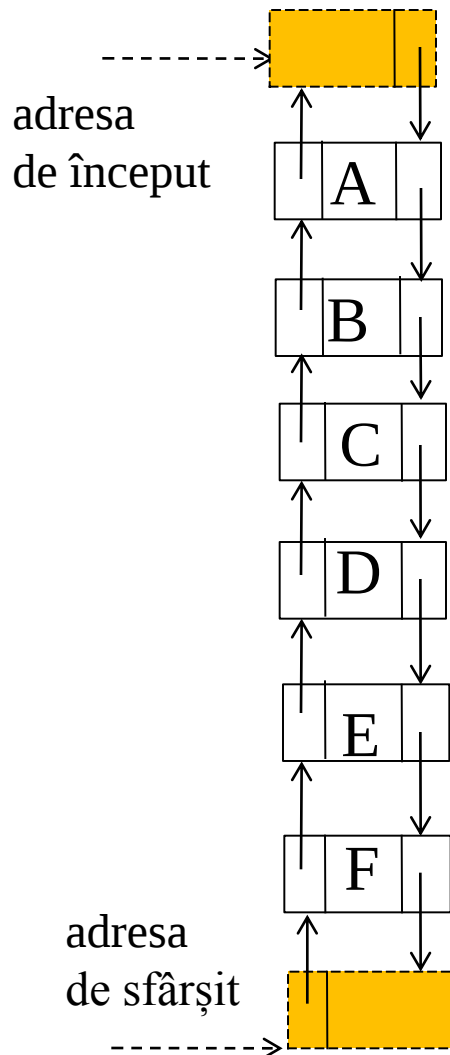
Cost: $\Theta(1)$



Operații cu liste dublu înlănțuite

Liste cu noduri fictive de început și sfârșit

- Nodurile fictive nu conțin informație utilă
- Lista vidă este constituită din cele două noduri ($L.inceput.urm=L.sfarsit$, $L.sfarsit.prec=L.inceput$)
- $L.inceput$, $L.sfarsit$ rămân neschimbate pe parcursul prelucrării listei
- Inserarea și ștergerea **se face întotdeauna în același mod** (se inserează între două noduri existente sau se șterge un nod aflat între două noduri existente)



Implementarea stivelor cu înlănțuiri

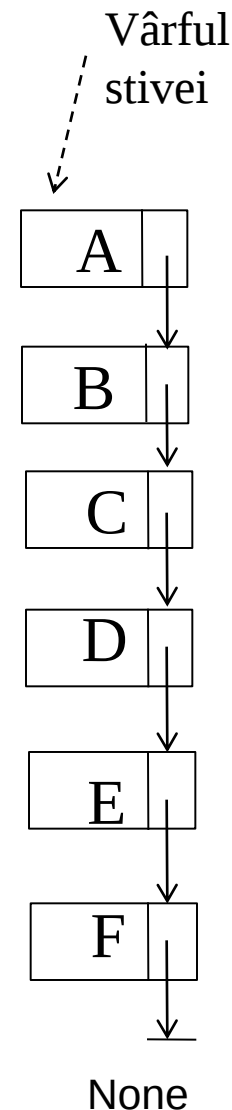
- **Motivație:** nu e necesară alocarea inițială a spațiului
- **Reminder:**

Stiva = LIFO (Last In First Out)

Operații cu/pe stive:

- Inserare (push) $\Leftrightarrow$ adăugare la început
- Stergere/Citire (pop) $\Leftrightarrow$ ștergerea primului element
- Verificare vârf $\Leftrightarrow$ consultarea primului element
- Stivă vidă $\Leftrightarrow$ adresa de început e None

Care variantă de listă înlănțuită e mai potrivită pentru implementarea stivelor?



Implementarea stivelor cu înlănțuiri

clasa pentru stiva implementata prin lista simplu inlantuita

```
class nod:
```

```
    def __init__(self,info=None,urm=None):
```

```
        self.info=info
```

```
        self.urm=urm
```

```
class stiva:
```

```
    def __init__(self):
```

```
        self.varf=None
```

```
    def push(self,e):
```

```
        self.varf=nod(e,self.varf)
```

```
    def top(self):
```

```
        if self.varf!=None:
```

```
            return self.varf.info
```

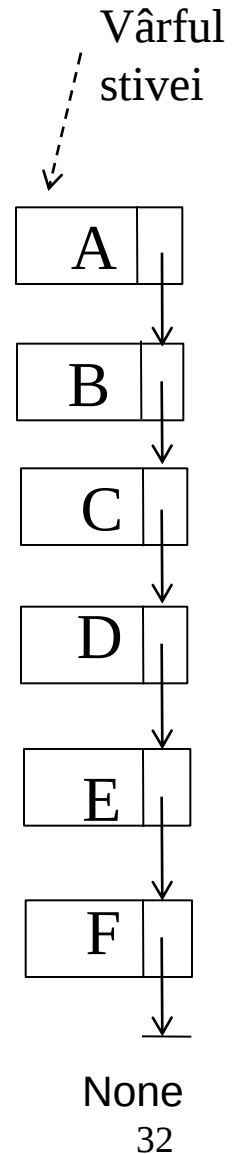
```
    def pop(self):
```

```
        if self.varf!=None:
```

```
            val=self.varf.info
```

```
            self.varf=self.varf.urm
```

```
            return val
```



Implementarea cozilor cu înlănțuiri

- Motivație: nu e necesară alocarea inițială a spațiului

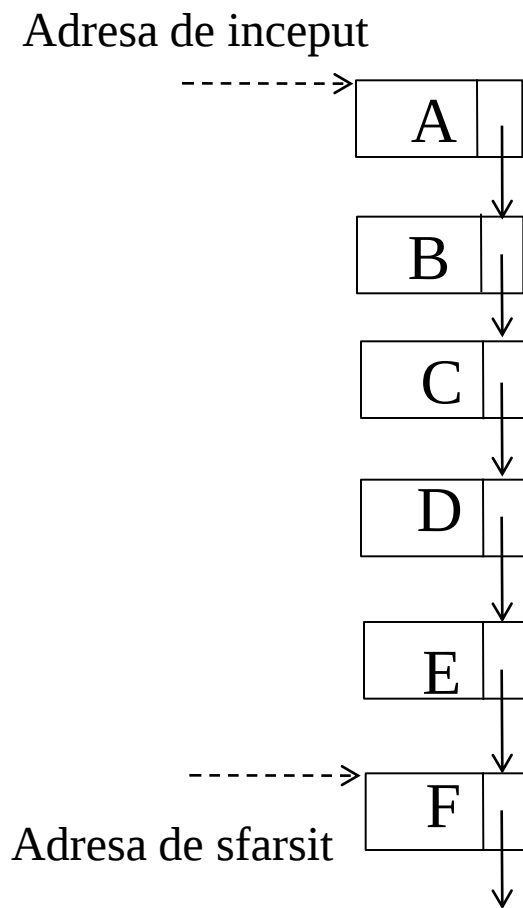
- Reminder:

Coadă = FIFO (First In First Out)

Operații cu cozi:

- Stergere din fața cozii $\Leftrightarrow$ ștergerea primului element
- Adăugare la sfârșitul cozii $\Leftrightarrow$ adăugare după ultimul element
- Consultarea primului element
- Coadă vidă $\Leftrightarrow$ adresa de început e None

Care variantă de listă înlănțuită e mai potrivită pentru implementarea cozilor simple? Dar a cozilor circulare?



Implementarea cozilor cu înlănțuiri

clasa pentru coada implementata prin lista inlantuita

```
class nod:
```

```
    def __init__(self,info=None,urm=None):
```

```
        self.info=info
```

```
        self.urm=urm
```

```
class coada:
```

```
    def __init__(self):
```

```
        self.fata=None
```

```
        self.spate=None
```

```
    def adauga(self,e):
```

```
        if self.spate==None:
```

```
            self.fata=self.spate=nod(e,None)
```

```
        else:
```

```
            self.spate.urm=nod(e,None)
```

```
            self.spate=self.spate.urm
```

```
    def extrage(self):
```

```
        if self.fata!=None:
```

```
            val=self.fata.info
```

```
            self.fata=self.fata.urm
```

```
            if self.fata==None:
```

```
                self.spate=None
```

```
            return val
```

```
    def parcurge(self):
```

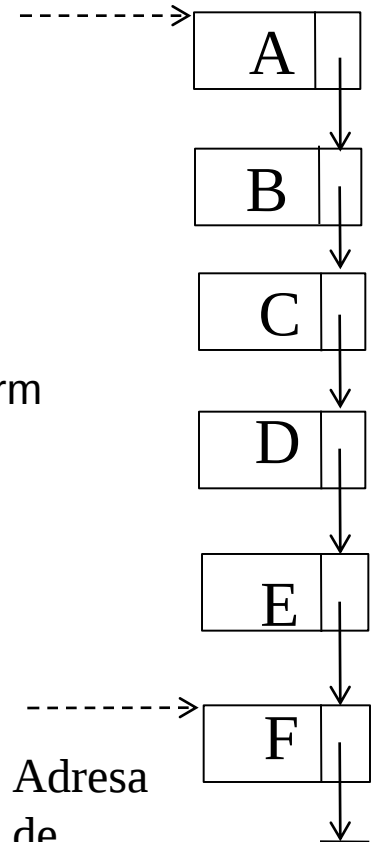
```
        ref=self.fata;
```

```
        while (ref!=None):
```

```
            print(ref.info)
```

```
            ref=ref.urm
```

Adresa de inceput



Implementarea cozilor cu înlănțuiri

coada circulara (e suficient sa se specifice adresa ultimului element)

```
class nod:
```

```
    def __init__(self, info=None, urm=None):
```

```
        self.info=info
```

```
        self.urm=urm
```

```
class coada:
```

```
    def __init__(self):
```

```
        self.spate=None
```

```
    def adauga(self,e):
```

```
        if self.spate==None:
```

```
            self.spate=nod(e,None)
```

```
            self.spate.urm=self.spate
```

```
        else:
```

```
            ref=self.spate.urm
```

```
            self.spate.urm=nod(e,ref)
```

```
            self.spate=self.spate.urm
```

```
    def extrage(self):
```

```
        if self.spate!=None:
```

```
            ref=self.spate.urm
```

```
            self.spate.urm=ref.urm
```

```
            return ref.info
```

```
    def rotire(self):
```

```
        self.spate=self.spate.urm
```

```
    def parcurge(self):
```

```
        if self.spate!=None:
```

```
            ref=self.spate.urm;
```

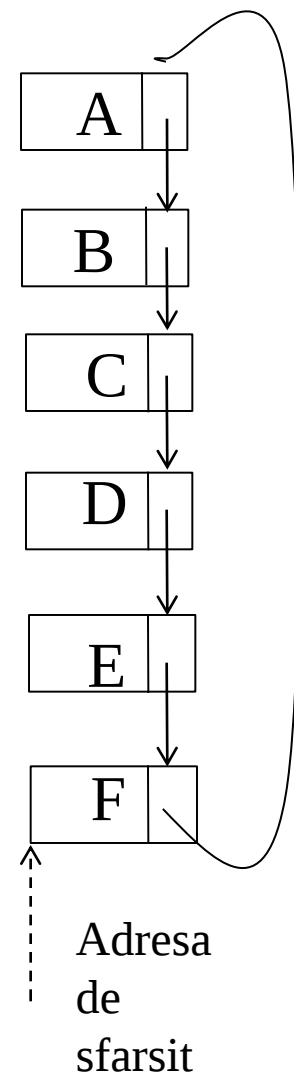
```
            print(ref.info)
```

```
            ref=ref.urm
```

```
            while (ref!=self.spate.urm):
```

```
                print(ref.info)
```

```
                ref=ref.urm
```



Sumar

- Liste simple și dublu înlanțuite
- Liste speciale: Stive, Cozi
- Operații cu liste:
 - **Căutare**
 - **Parcurgere**
 - **Adăugare**
 - **Ștergere**
 - Copiere
 - Împărțire
 - Îmbinare
 - Sortare
- Când sunt mai bune tablourile? Când sunt mai bune listele înlanțuite?

Cursul următor va fi despre...

... tehnica reducerii (decrease and conquer)

Intrebare de final

Ce valoare conține nodul din lista L (schema alăturata) specificat prin:

`L.inceput.urm.urm.prec`

Variante de răspuns: adresa de început

- a) A
- b) B
- c) C
- d) D
- e) E
- f) F

adresa de sfârșit

