STRUCTURI DE DATE SI ALGORITMI

CURS I, II
INTRODUCERE, ALGORITM, ANALIZA ALGORITMILOR, LISTE, STIVE, COZI

AGENDA

- Prezentare curs, echipa, reguli
- Putin despre algoritmi …
- Structuri de date
  - Abstracte: Lista, stiva, coada
  - Implementari: lista simplu/dublu inlantuita, etc.

STRUCTURI DE DATE SI ALGORITMI

- Echipa:
  - Camelia Lemnaru (curs),
    - Sala M03 (mansarda, intrarea pe langa P01, etaj 2), str. G. Baritiu
    - Tel. 0264-401479
    - Camelia.Lemnaru@cs.utcluj.ro
  - Iulia Costin, Matei Craciun, Ionel Giosan, Mircea Muresan, Anca Ciurte, Andra Petrovai, **Camelia Lemnaru**
  - Pagina curs:
    - Moodle: https://moodle.cs.utcluj.ro/

OBIECTIVE

- Familiarizarea cu structuri de date fundamentale si algoritmi care opereaza pe acestea
- Familiarizarea cu diferite proprietati ale structurilor de date si algoritmilor
- Dezvoltarea capacitatii de a analiza comparativ algoritmii (proprietati, timp de rulare, memorie folosita, etc)
- Dezvoltarea capacitatii de a identifica structuri de date si algoritmi potriviti, atunci cand acestia sunt disponibili
- Dezvoltarea capacitatii de a dezvolta algoritmi si structuri de date noi, atunci cand nu exista disponibile

ORGANIZARE

- Cursuri
  - participare, implicare prin intrebari
- Sesiuni de laborator
  - Lucrarea de laborator parcursa inaintea sesiunii de laborator!!!
  - 3 tipuri de sarcini:
    - obligatorii (depunctare!)
    - aditionale (optionale, de fixare a cunostintelor)
    - extra credit (bonus, 9->10)
  - Cadrul didactic de la laborator este acolo in primul rand sa va ajute, in al doilea rand sa va evalueze

EVALUARE

- Activitate practica – 30% (laborator)
  - 0.45*Test1 + 0.45*Test2 + 0.1*Extra credit
    - extra credit maxim -> min 4 probleme extra credit
      - probleme extra credit - max 2 saptamani pt a fi prezentate

- Evaluare pe parcursul semestrului - 20%
  - Partial, in timpul cursului
    - Dupa curs 6?
- Evaluare finala - 50%
  - sesiunea de vara

BIBLIOGRAFIE RECOMANDATA

- Aho, Hopcroft, Ullman [AHU]: Data Structures and Algorithms, Addison-Wesley, 427 pages, 1987.
  - Cartea clasică se SDA, introductivă, valorosa, nu acopera exhaustiv tematica; Pascal extins
- Cormen, Leiserson, Rivest, (Stein) [CLR, CLRS]: Introduction to Algorithms. MIT Press / McGraw Hill, (2nd edition), 1028 pages, 1990 (2002).
  - BIBLIA de Algoritmi Fundamentali (anul II), scrisa pentru toate nivelele, cu capitole introductive, dar si detaliate (in a doua parte a cartii); pseudocod

BIBLIOGRAFIE EXTINSA

- Preiss. Data Structures and Algorithms with object-Oriented Design Patterns in C++, John Wiley and Sons, 660 pages, 1999.

- Knuth. The Art of Computer Programing, Addison Wesley, 3 volume, multiple editii.

- Skiena. The Algorithm Design Manual. http://sist.sysu.edu.cn/~isslxm/DSA/textbook/Skiena.-.TheAlgorithmDesignManual.pdf
(recomandata de recruiterii Google, DAR!! nu suficient de teoretica)

RESURSE ADITIONALE

- visualgo.net
  - vizualizare, unealta interactiva
- http://algoviz.org/
- coursera.org
  - Data structures: Measuring and Optimizing Performance (U.C. San Diego) - in desfasurare
  - …

CUPRINS CURS

- Introducere. Algoritm. Structura abstracta de date. Analiza eficientei
- Liste. Stive. Cozi
- *ADT* pentru colectii: dictionare, tabele de dispersie, cozi de prioritati
- Arbori. Arbori binari. Arbori binari de cautare
- Arbori binari de cantare echilibrati: AVL, rosu si negru, arbori perfect echilibrati
- Grafuri: reprezentare; definitii, proprietati, problematici, traversari
- Tehnici de dezvoltare a algoritmilor / generale de cautare: backtracking, branch and bound, greedy, divide et impera, programare dinamica

REZOLVAREA UNEI PROBLEME

- **Model exact** al solutiilor valide
    - gasirea unui astfel de model - 50% problema rezolvata
        - experienta, cunostinte de matematica, inginerie software, algoritmica, etc.
- Dupa ce dispunem de modelul matematic, putem **specifica o solutie in termenii acelui model**

CONSTRUIREA MODELULUI

- Provocari:
  - cum modelam **obiectele reale** ca si **entitati matematice**
  - definirea **multimii de operatii** care manipuleaza entitatile
  - modalitatea de **stocare in memorie** (cum le agregam, cum le stocam propriu-zis) - **STRUCTURI DE DATE!**
  - **algoritmii** care realizeaza operatiile - **ALGORITMI!**

DEFINITII

- **Problema computationala:** O specificare in termeni generali a *intrarilor* si *iesirilor* si relatia dorita dintre acestea.
- **Instanta de problema:** O colectie particulara de intrare pentru problema data.
- **Algoritm:** O metoda de rezolvare a unei probleme care poate fi implementata de un calculator.
- **Program:** Implementarea explicita a unui algoritm

EXEMPLU - SORTAREA

- Problema:
  - *Intrare:* O secventa/sir de *n* numere <a_1, a_2, …, a_n>
  - *Iesire:* O permutare a numerelor <a_1, a_2, …, a_n> a.i. $a_i \leq a_j$, $i < j$
- Instanta: sirul <7, 5, 4, 10, 5>
- Algoritmi:
  - sortare prin selectie
  - sortare prin insertie
  - sortare rapida (Quick sort)
  - etc

UN ALGORITM....

- trebuie sa fie:
- corect
- eficient
- *usor de implementat*

PROIECTAREA ALGORITMULUI

```
AlgorithmDesign(informal problem)
 1   formalize problem (mathematically) [Step 0]
 2   repeat
 3       devise algorithm [Step 1]
 4       analyze correctness [Step 2]
 5       analyze efficiency [Step 3]
 6       refine
 7   until algorithm good enough
 8   return algorithm
```

EXEMPLU DE ALGORITM - ROBOT TOUR OPTMIZATION

- Problema: *Robot Tour Optimization*

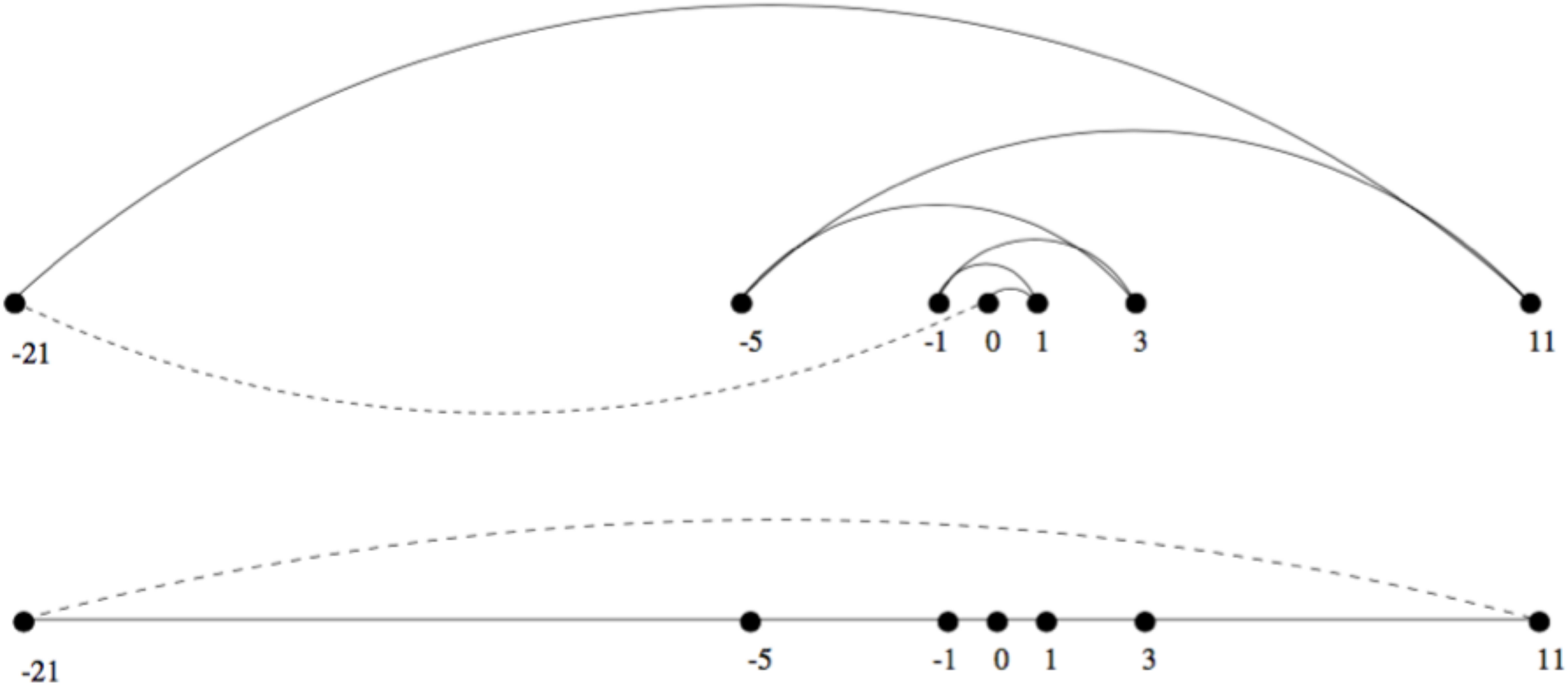
- Intrare: O multime *S* de *n* puncte in plan.

- Iesire: Care este *cel mai scurt tur ciclu* (e. cycle tour) care viziteaza fiecare punct din multimea S?

ROBOT TOUR OPTIMIZATION

```
NearestNeighbor(P)
Pick and visit an initial point p0 from P
   p = p0
   i = 0
   While there are still unvisited points
      i=i+1
      Select pi to be the closest unvisited
          point to pi-1
      Visit pi
Return to p0 from pn-1
```

NearestNeighbor



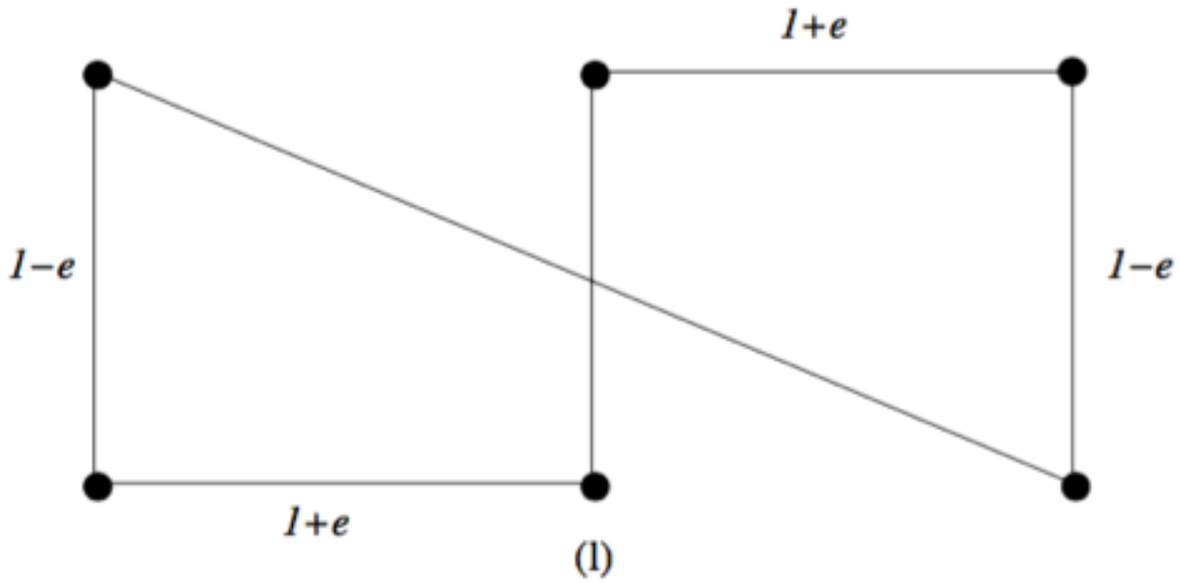
Algoritmul nu gaseste neaparat turul de lungime minima!

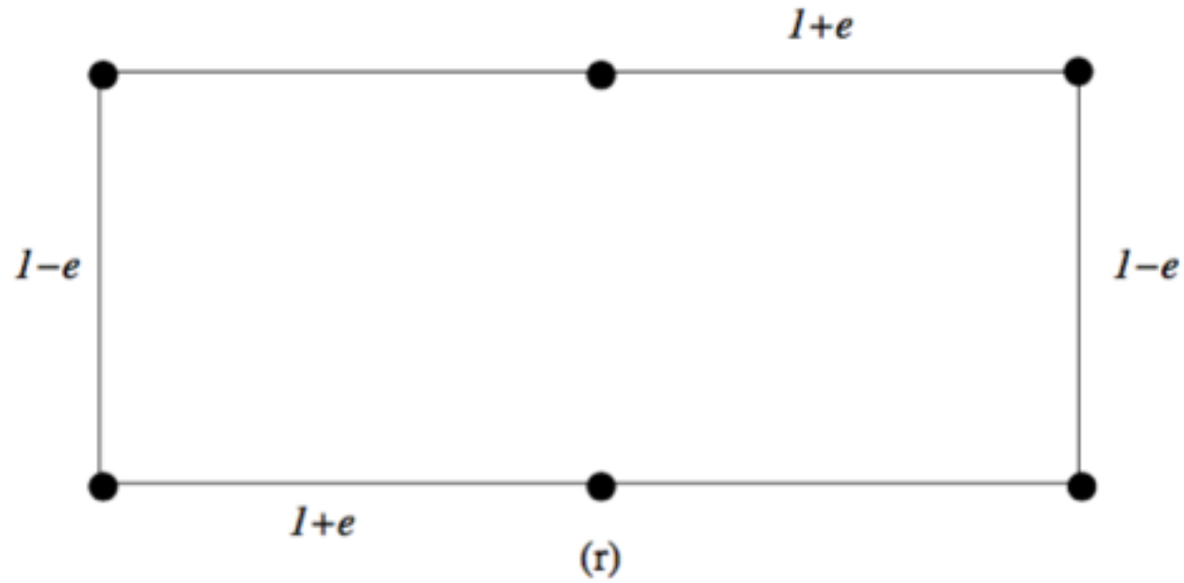
Imaginea a fost preluata din S. Skiena - The Algorithm Design Manual

ROBOT TOUR OPTIMIZATION

```
ClosestPair(P)
Let n be the number of points in set P
For i = 1 to n - 1 do
    d=∞
    For each pair of endpoints (s,t) from
     distinct vertex chains
       if dist(s,t)≤d then
          sm=s,tm=t,and d=dist(s,t)
   Connect (sm, tm) by an edge
Connect the two endpoints by an edge
```

ClosestPair





Nici acest algoritm nu gaseste neaparat turul de lungime minima!

Imaginea a fost preluata din S. Skiena - The Algorithm Design Manual

ROBOT TOUR OPTIMIZATION

```
OptimalTSP(P)
   d=∞
   For each of the n! permutations Pi of point set P
      If (cost(Pi) ≤ d) then
         d = cost(Pi) and Pmin = Pi
Return Pmin
```

DE TINUT MINTE…

- Esenta unui algoritm este o idee!
- Este important ca algoritmul sa fie exprimat clar! (pseudocod, limbaj natural, limbaj de programare)
- Corectitudinea trebuie demonstrata! Incorectitudinea - contra-exemplu (verificabilitate, simplitate)
- Contra-exemple: dimensiune mica, exhaustiv, slabiciuni -> egalitate, extreme

ANALIZA ALGORITMILOR

- Stabilirea cantitatilor de **resurse** necesare rularii algoritmului (de regula in functie de **dimensiunea** intrarii).
- Resurse:
  - timp
  - memorie (spatiu)
  - numar de accese la memoria secundara
  - numar de operatii aritmetice de baza
  - traficul de retea
- Formal, se defineste timpul de rulare al unui algoritm pe o intrare particulara ca fiind numarul de operatii de baza (atribuiri, comparatii) efectuate de algoritm pe acea intrare.

ANALIZA ALGORITMILOR - ESENTA

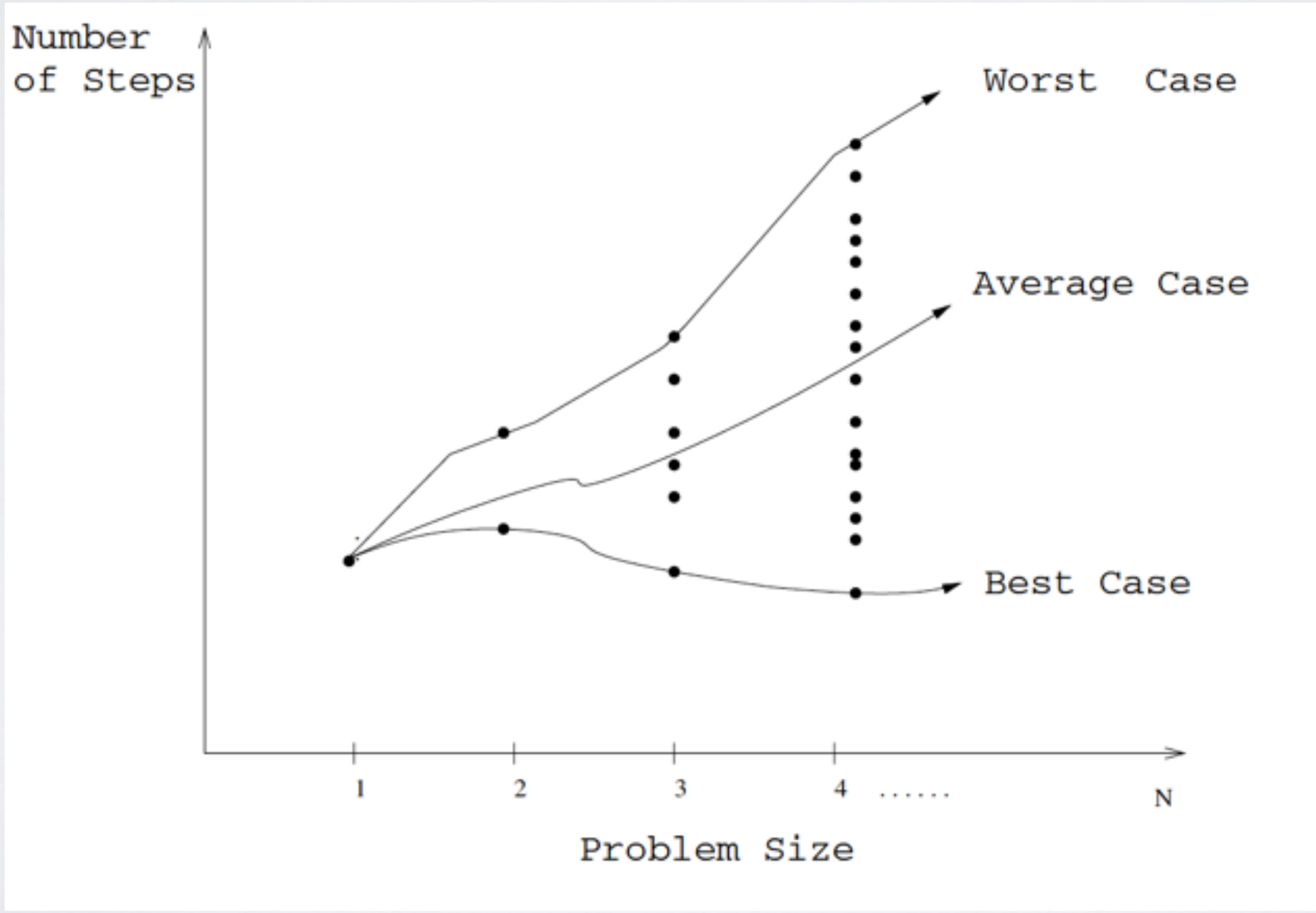
- maniera independenta de masina si limbaj
- unelte:
  - Modelul de calcul RAM
  - Analiza asimptotica a cazului defavorabil

MODELUL RAM DE CALCUL

- orice operatie simpla se executa intr-o unitate de timp
- buclele si sub-rutinele nu sunt operatii simple (compozitie de mai multe operatii simple)
- accesul la memorie se executa intr-o unitate de timp (memorie nelimitata, nu se face diferenta intre cache si disc)

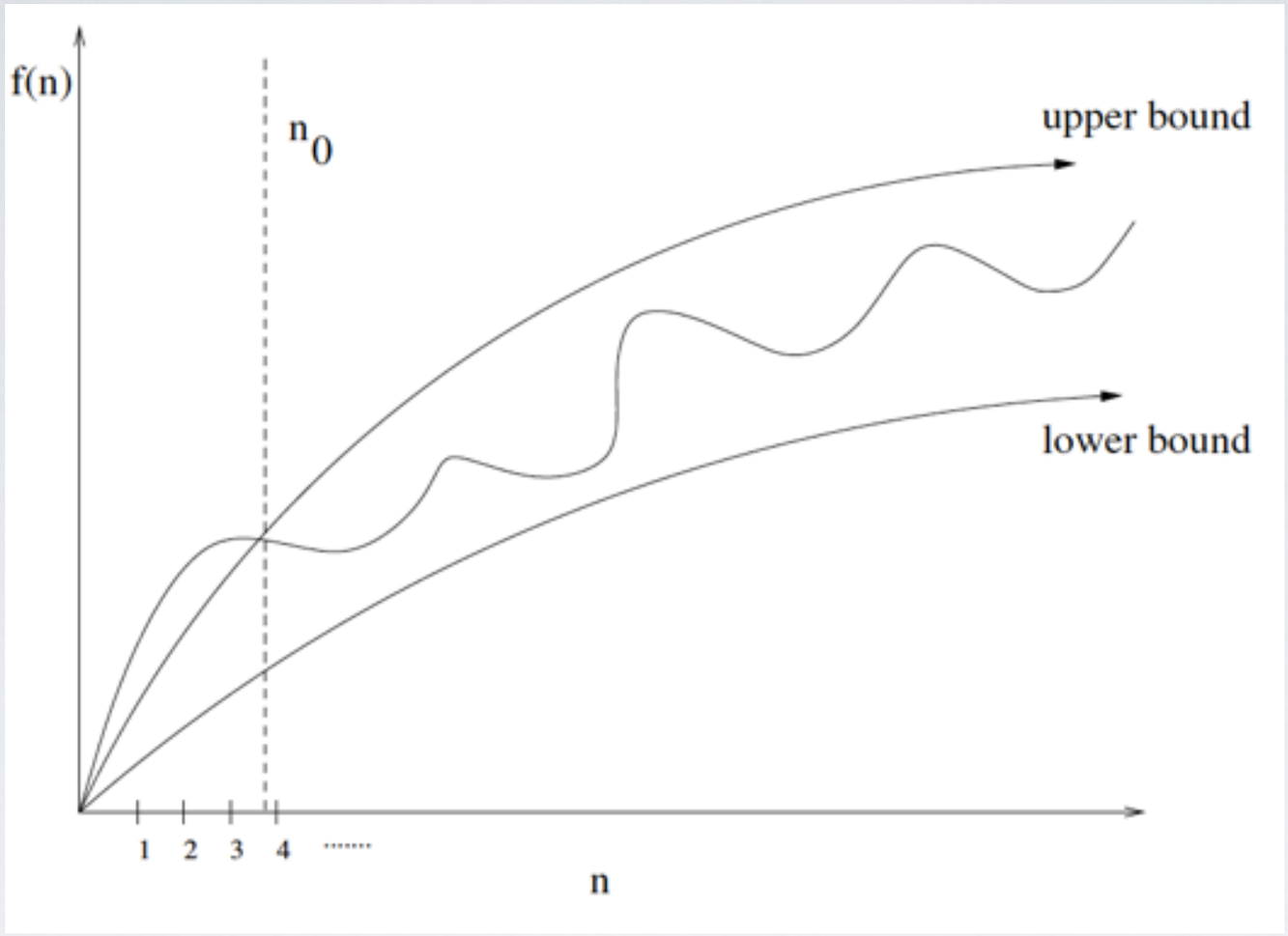
CAZ FAVORABIL, DEFAVORABIL, MEDIU STATISTIC

- modelul RAM aplicat pe TOATE instantele posibile (e.g. sortare)



O (BIG OH)

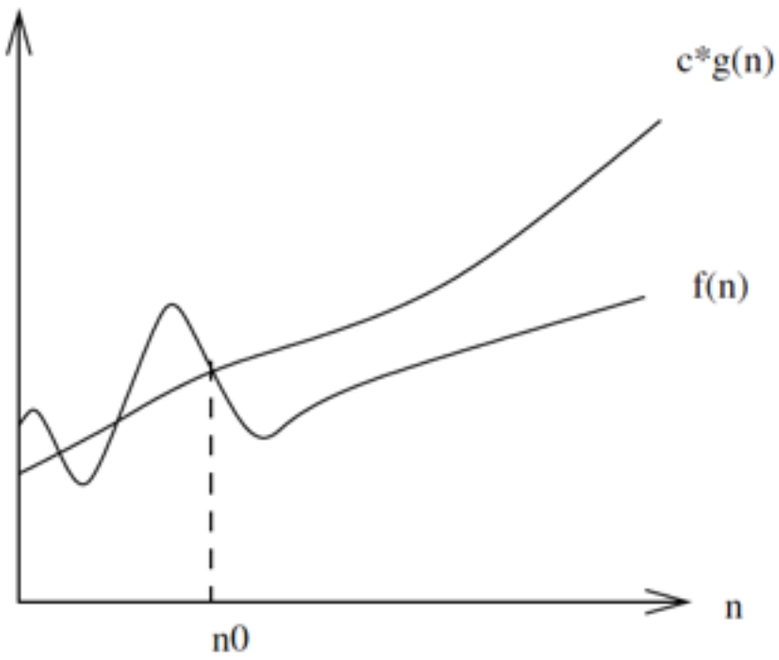
- extrem de greu de specificat functiile de complexitate exact!
  - prea multe “spykes/bumps”
  - prea multe detalii pentru a putea fi specificate exact (detalii neinteresante de cod)
- => limite superioare/inferioare



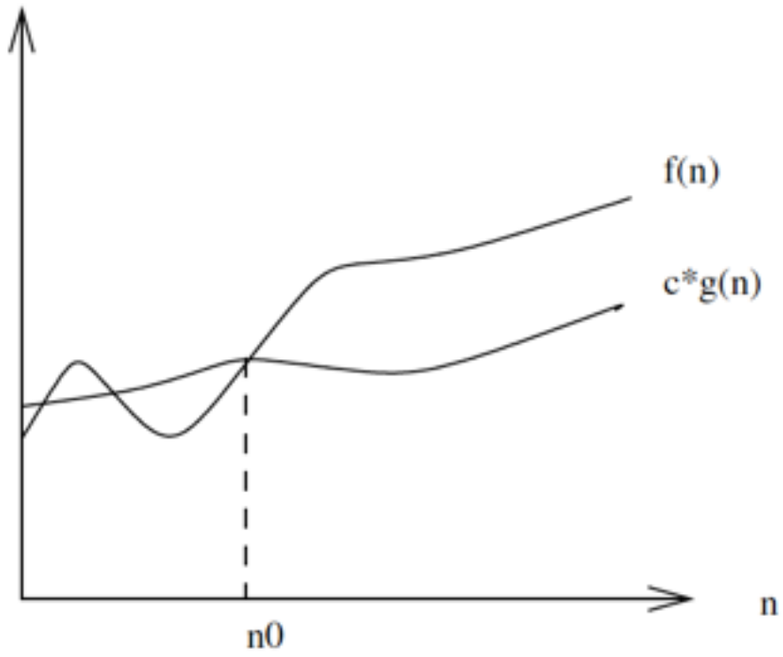
Upper bound: $f(n) = O(g(n)), \exists\, c, n_0 \;\; s.t. \; f(n) < c \cdot g(n), \forall n > n_0$

Lower bound: $f(n) = \Omega(g(n)), \exists\, c, n_0 \;\; s.t. \; f(n) \geq c \cdot g(n), \forall n > n_0$

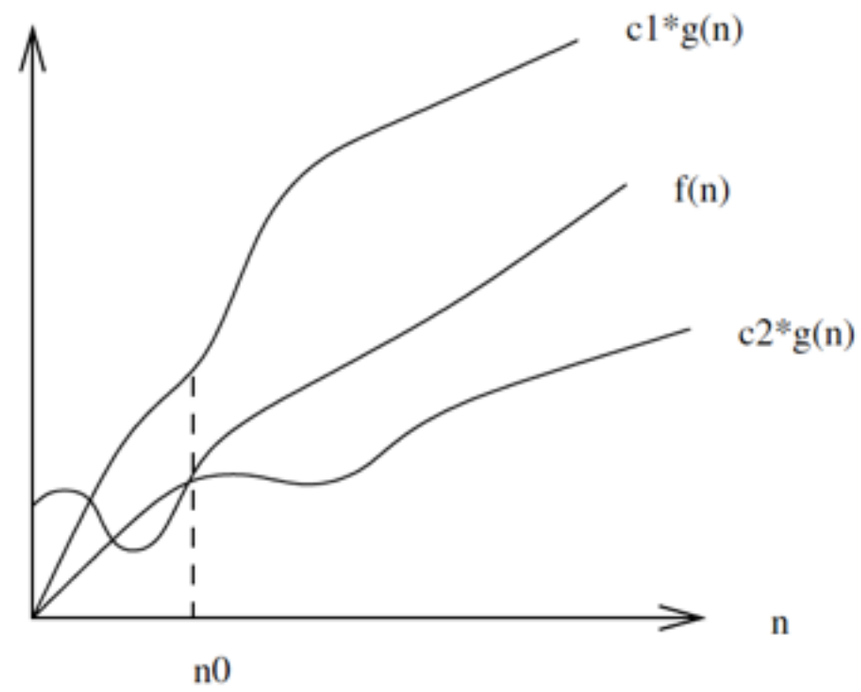
Tight bound: $f(n) = \Theta(g(n)), \exists\, c_1, c_2, n_0 \;\; s.t. \; f(n) \geq c_1 \cdot g(n) \; and$
$f(n) < c_2 \cdot g(n), \forall n > n_0$



(a)



(b)



(c)

TIP DE DATA ABSTRACT (ADT) *vs* STRUCTURI DE DATE (DS)

- sinonime?

- ADT - model matematic pentru *tipuri de date*
  - semantica d.p.d.v. utilizator (valori posibile, operatii) -> **comportament**!

- DS - reprezentari concrete ale datelor
  - **implementare**

ADT *vs* DS

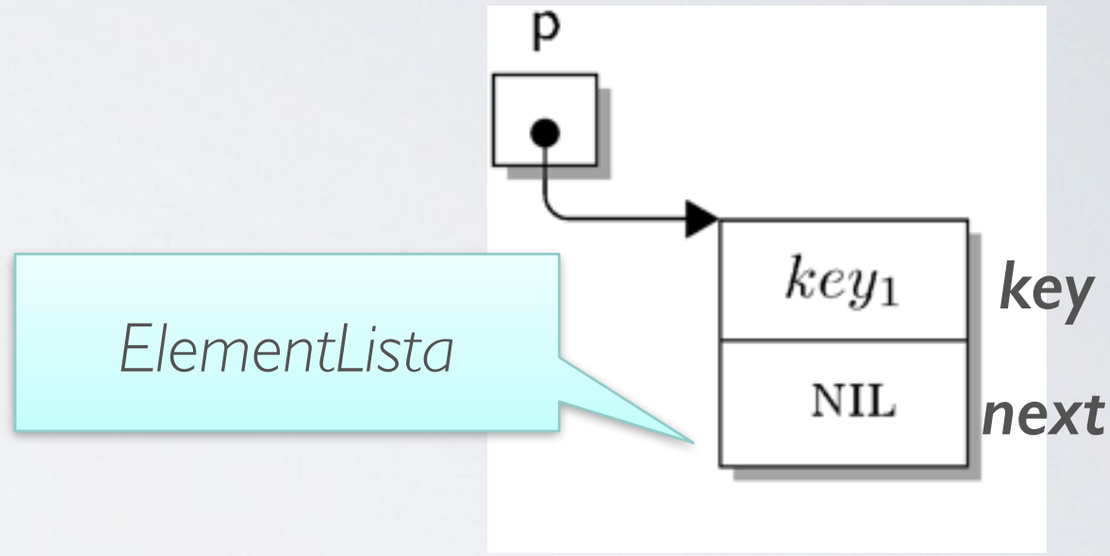
- *Lista* - o colectie *ordonata* de elemente, posibil duplicate (container)
  - *ordonata* nu inseamna neaparat *sortata* in acest context! (elementele apar unul dupa celalalt, acces secvential)
- O lista poate fi implementata fie folosind un *vector (array)*, fie o *lista simplu/dublu inlantuita*
- *particular - java.util.ArrayList, java.util.LinkedList,* sau *o biblioteca specifica C++ (*din *Standard Template Library).*

LISTA (ADT)

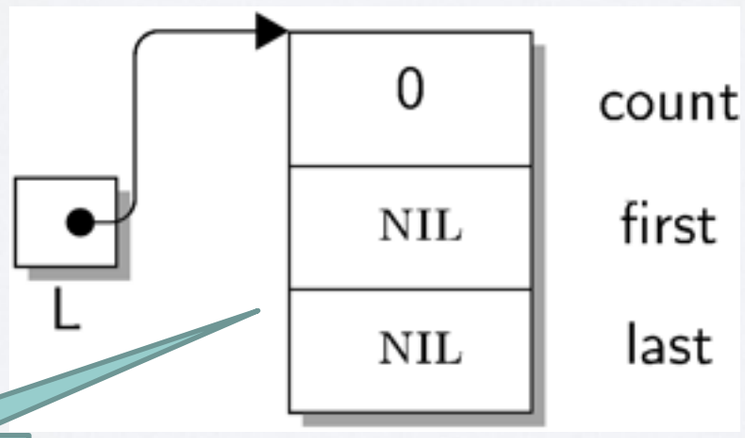
- O colectie de elemente (*ElementLista*), poate contine duplicate
- Operatii fundamentale:
  - insert(x): Adauga ElementLista *x* la inceputul listei (poate si la sfarsit, in ordine)
    - Intrare: ElementLista; Iesire: nimic
  - delete(x): Sterge ElementLista *x*; eroare daca lista e goala
    - Input: pointer catre elementul de sters; Iesire: nimic
  - search(k): cauta ElementLista care are cheia *k*
    - Intrare: cheia de cautat; Iesire: pointer catre ElementLista sau nil daca nu s-a gasit
- Operatii aditionale:
  - size(): returneaza numarul de operatii din lista
    - Intrare: nimic; Iesire: intreg
  - isEmpty(): returneaza valoare booleana care semnaleaza daca lista e goala
    - Intrare: nimic; Iesire: boolean
  - first(): returneaza, fara a sterge, primul element din lista; eroare daca lista este goala (head())
    - Intrare: none; Iesire: ElementLista
  - last(): returneaza, fara a sterge, ultimul element din lista; eroare daca lista este goala
    - Intrare: none; Iesire: ElementLista
  - prev(x), next(x): returneza ElementLista care precede/succede ElementLista x
  - tail(): returneaza restul listei, fara primul element
  - createEmpty()

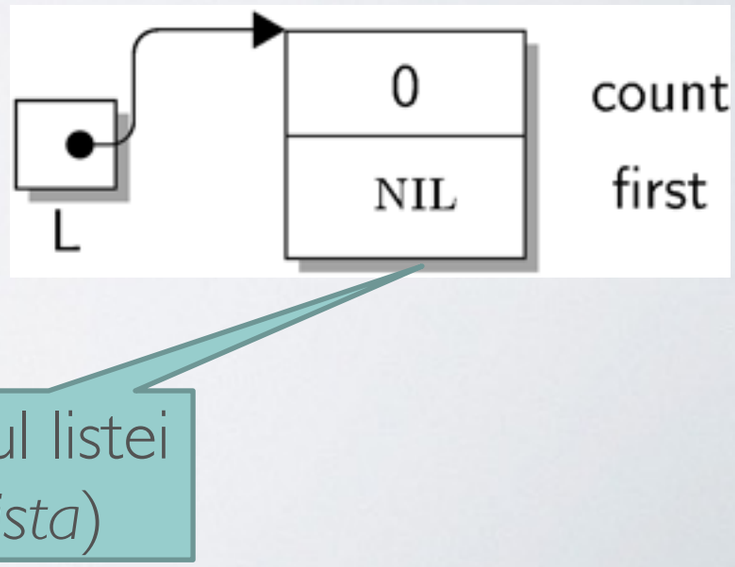
LISTA SIMPLU INLANTUITA

- Inlantuire intr-o singura directie
- Elementele listei - structura separata (*ElementLista*)



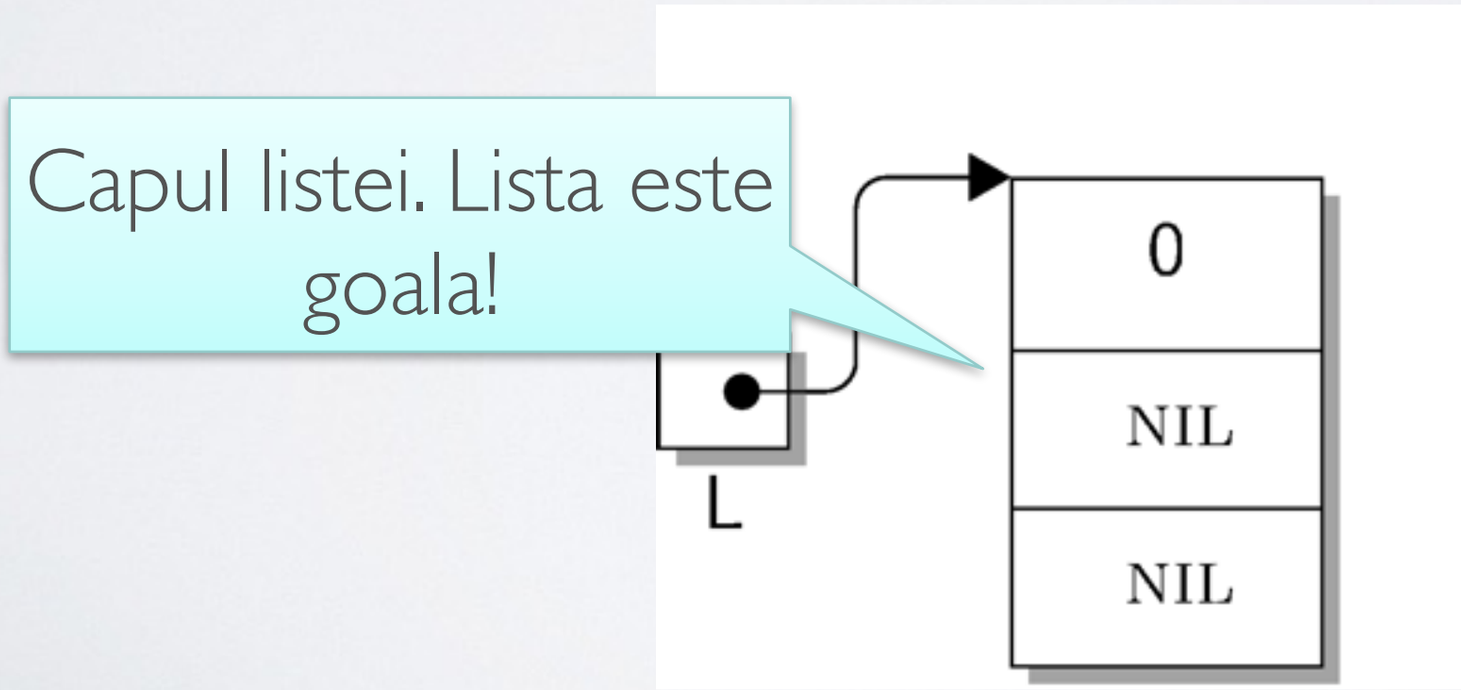
- Structura *Lista* (capul listei):
  - 2 (3) campuri:
    - count
    - first
    - (last)





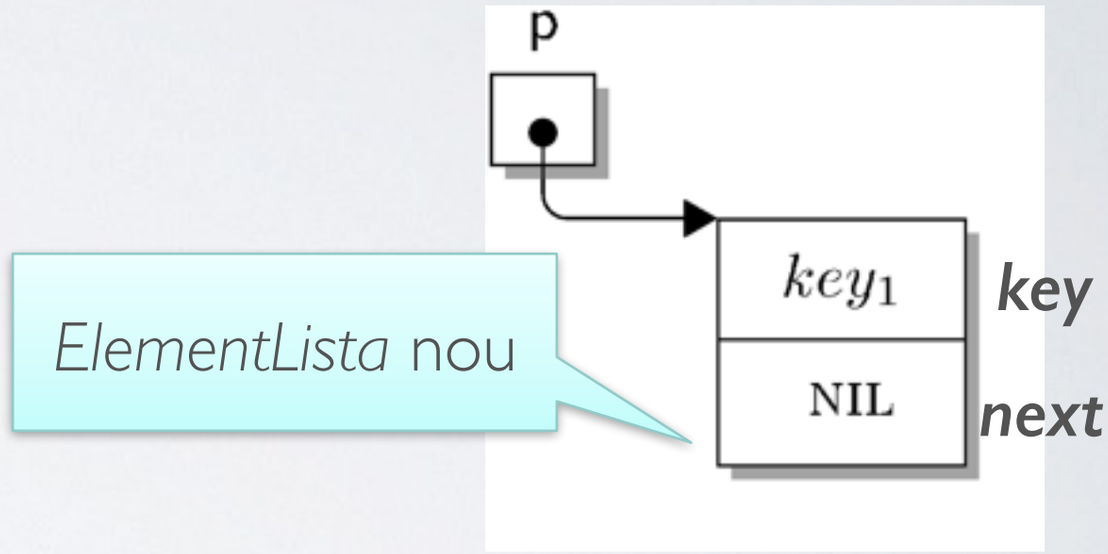
LISTA SIMPLU INLANTUITA CREARE

- *createEmpty()*:
  - Creeaza capul listei
  - Seteaza *L* sa pointeze catre capul listei



LISTA SIMPLU INLANTUITA
INSERT

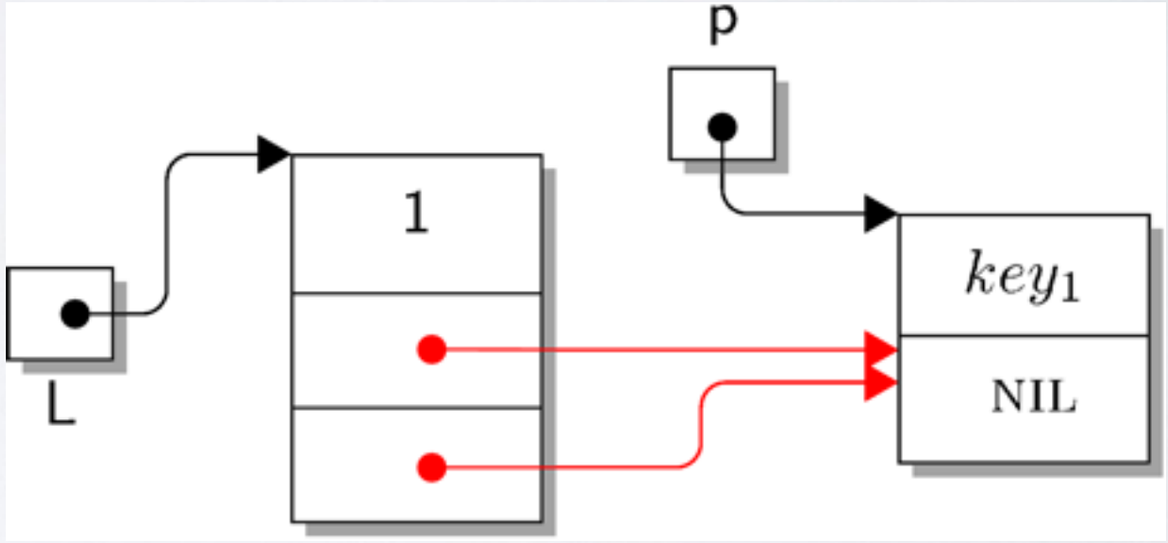
- *Insert* - optiuni
  1. (intotdeauna) la inceput
  2. la ambele capete (inceput si sfarsit)
  3. la pozitia k
  4. ordonata
- Trebuie creat intai elementul de inserat !!!



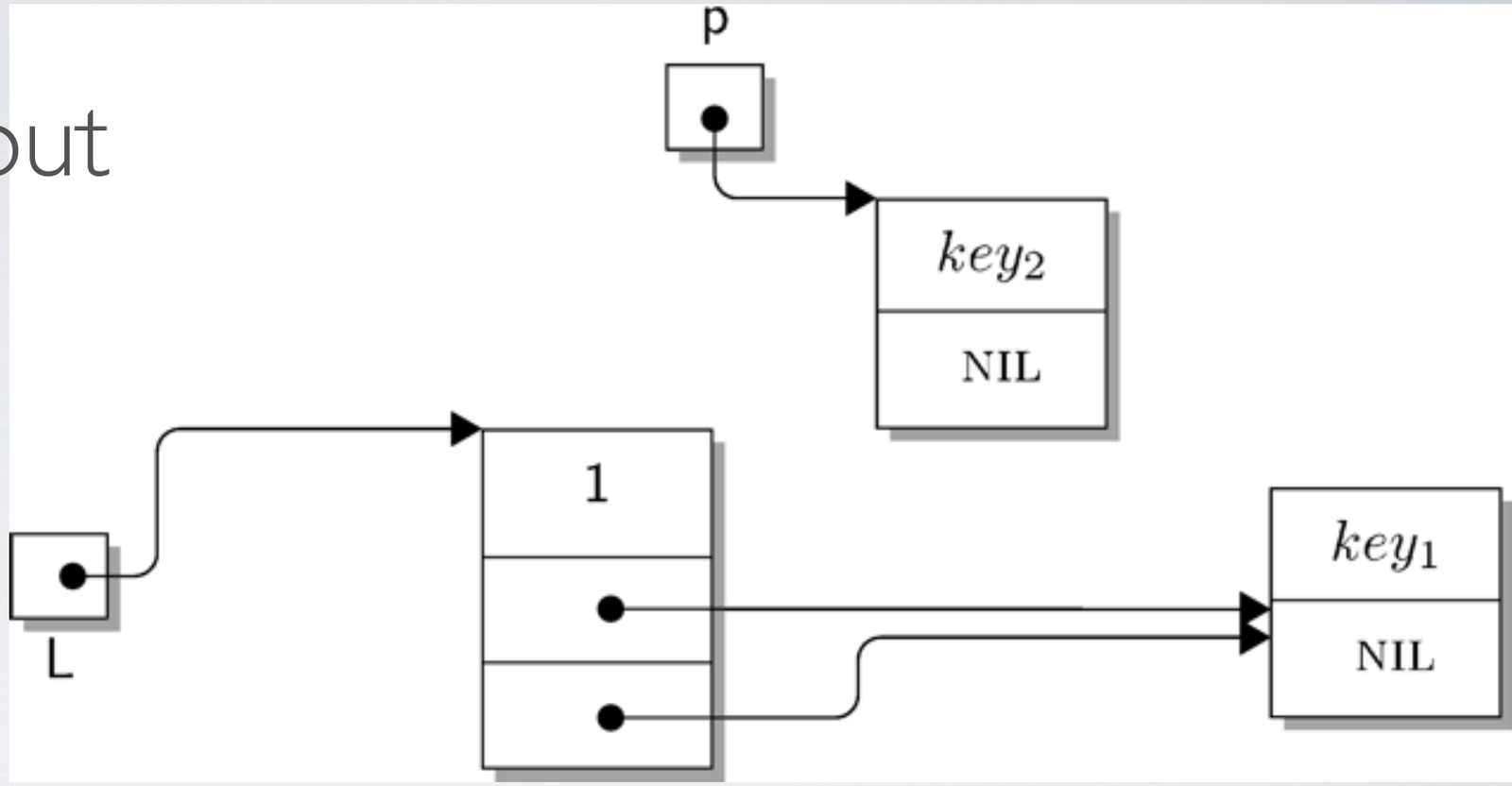
  - Caz 1: adaugare la inceput, el va fi noul *head*
  - Caz 2: adaugare la sfarsit (noul *last*?)
  - Caz 3 & 4:
    - cautare loc
    - refacere legaturi

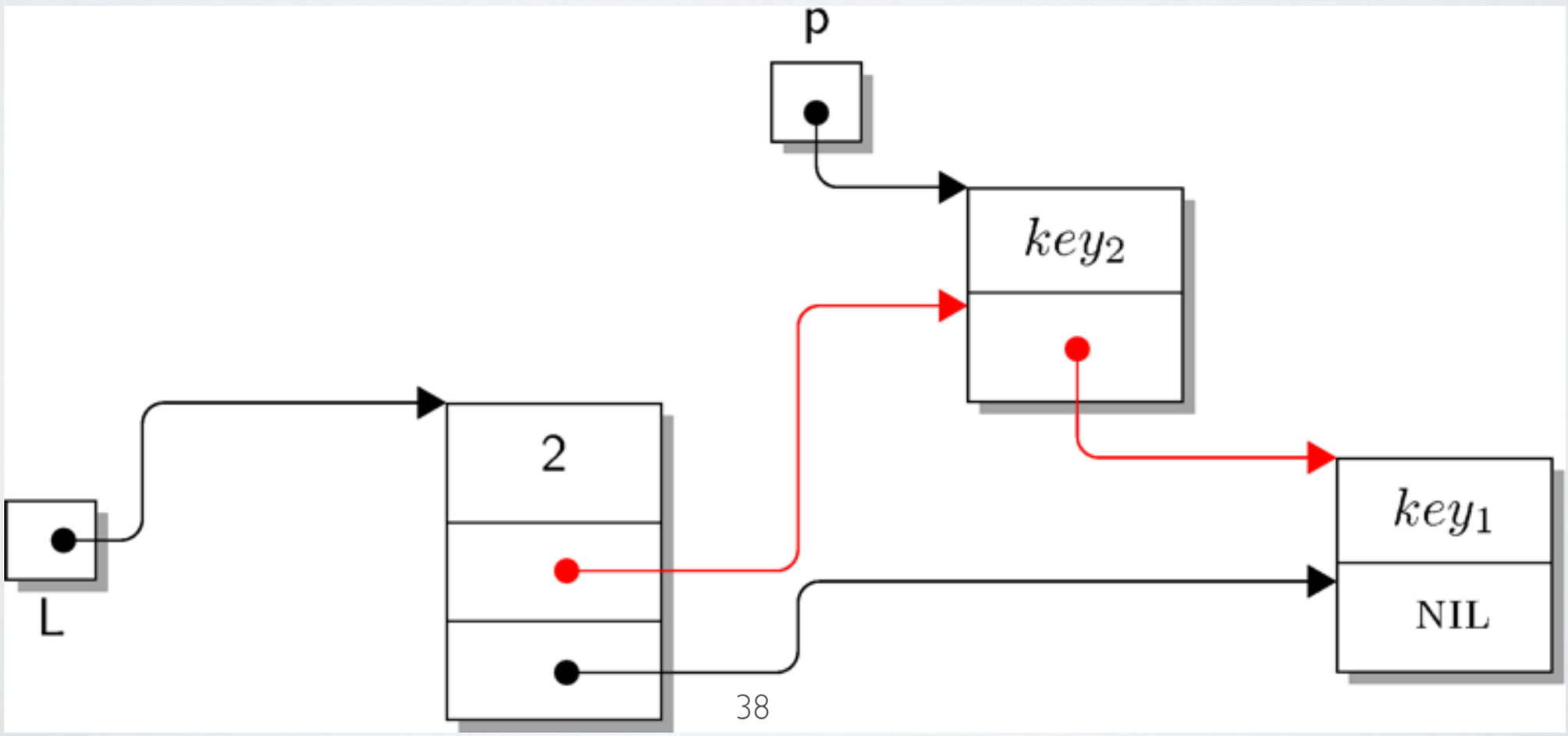
LISTA SIMPLU INLANTUITA INSERT

1. Adaugare la inceput
   - Daca lista e goala
     - Similar adaugare la final (append)



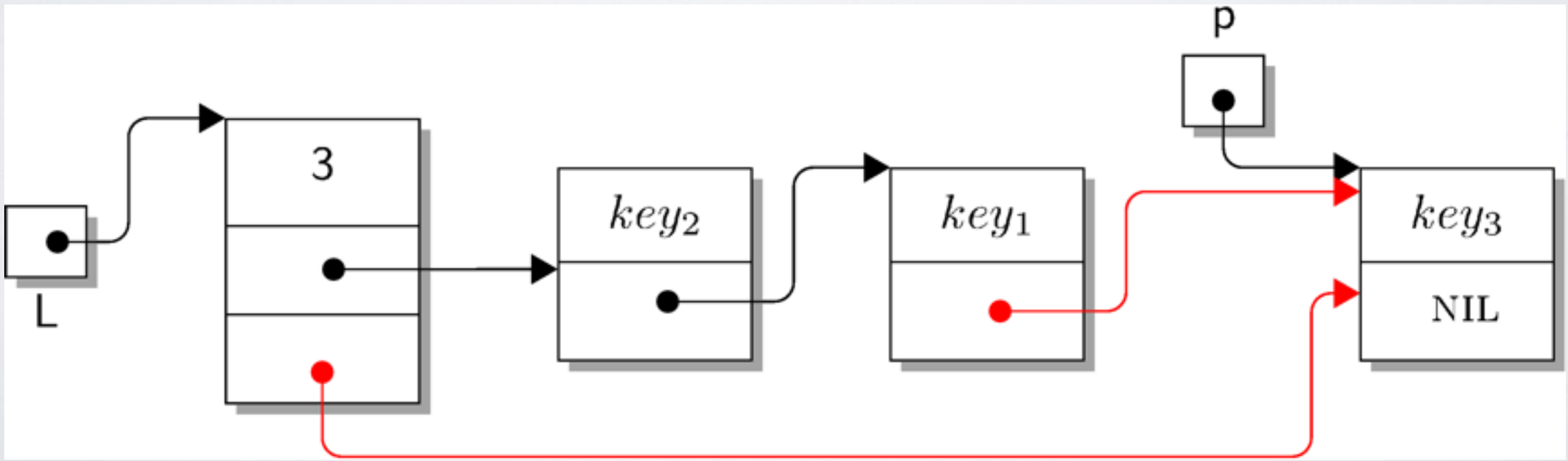
1. Adaugare la inceput
   - Lista nu e goala





LISTA SIMPLU INLANTUITA INSERT

2. Adaugare la final (*append*)
   - Lista goala: la fel ca si la inserare la inceput
   - Lista care contine elemente:



LISTA SIMPLU INLANTUITA INSERT

3. Inserare in lista *ordonata* sau la pozitia *k:*
   - Lista goala: ca si mai inainte
   - Lista cu elemente:
     - Inainte de primul nod (i.e. inserare la inceput)
     - Dupa ultimul nod (i.e. append)
     - **In interiorul listei**
       - trebuie traversata lista, si stabiliti doi pointeri:
         - nodul curent (va deveni *next* pt nodul inserat)
         - nodul anterior (al carui pointer *next* va referi nodul nou inserat)
- Exercitiu!!!!

EXEMPLU COD: *INSERT FIRST*

```
void insert_first(list **first, item_type x)
{
 list *p; /* temporary pointer */
 p = malloc( sizeof(list) );
 p->item = x;
 p->next = *first;
  *first = p;
}
```

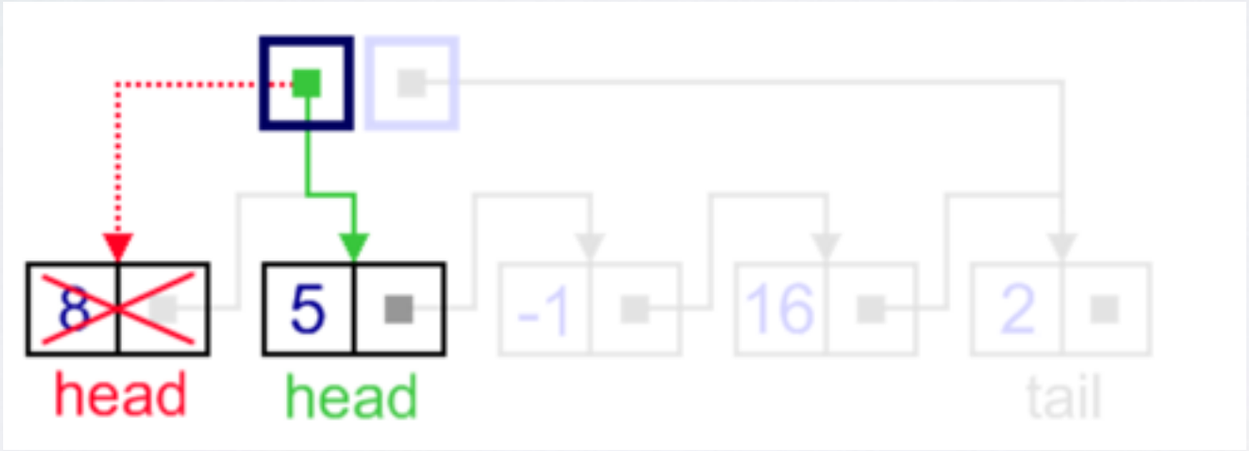
SEARCH

```
list *search_list(list *l, item_type x){
   if (l == NULL) return(NULL);
   if (l->item == x)
      return(l);
   else
      return( search_list(l->next, x) );
}
```

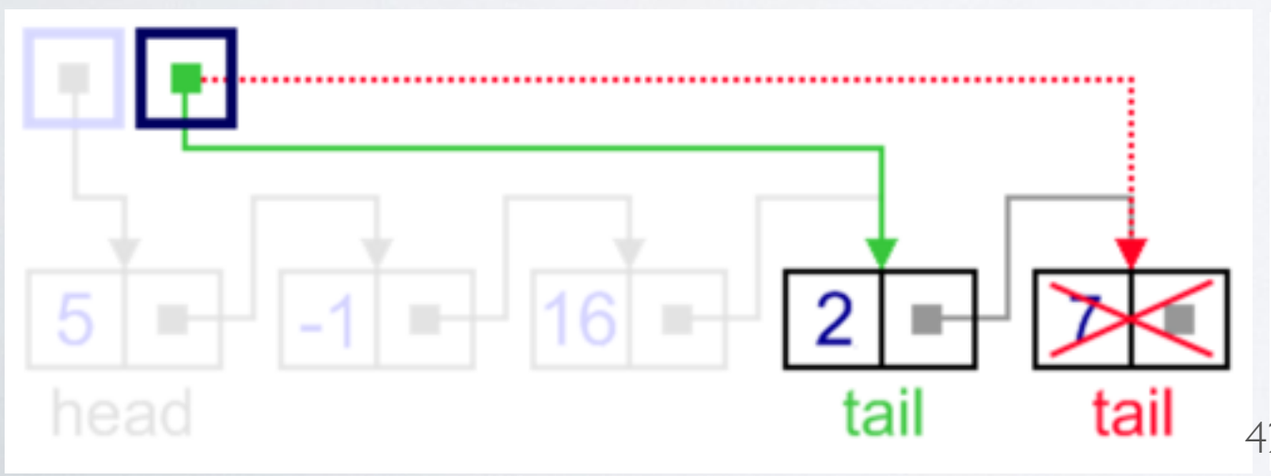
LISTA SIMPLU INLANTUITA DELETE

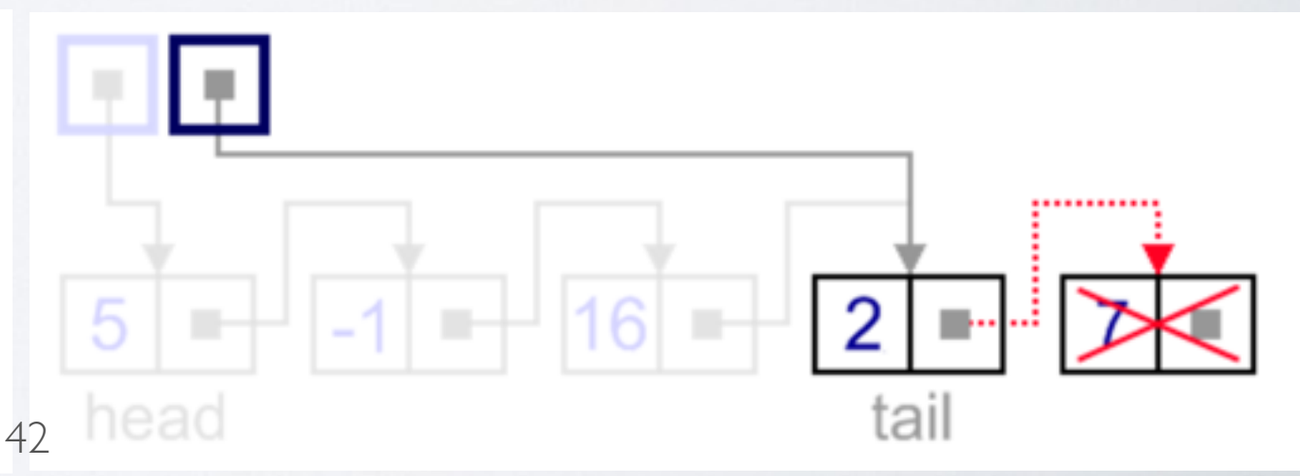
- *Delete* - cazuri speciale

1. primul element



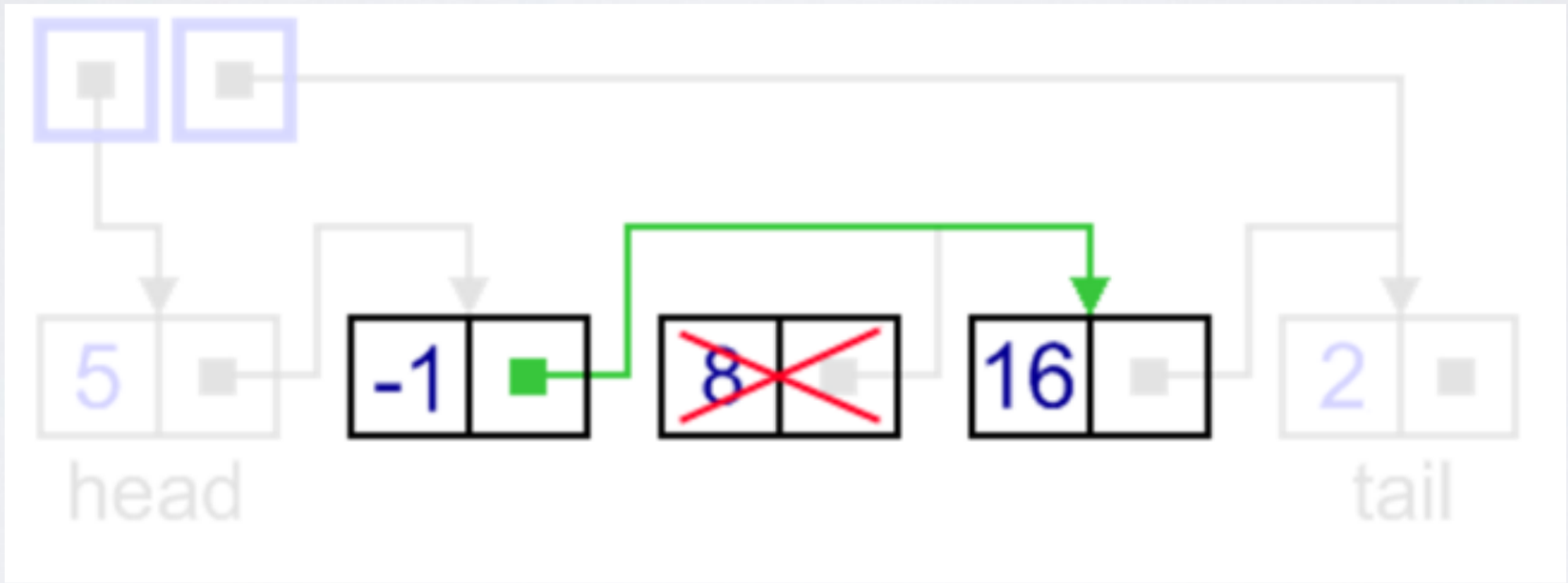
2. ultimul element





LISTA SIMPLU INLANTUITA DELETE

- *Delete* - cazul general



```
list *predecessor_list(list *l, item_type x){
   if ((l == NULL) || (l->next == NULL)) {
      printf("Error: predecessor sought on null list.\n");
      return(NULL);
   }
   if ((l->next)->item == x)
      return(l);
 else
      return(predecessor_list(l->next, x));
}
```

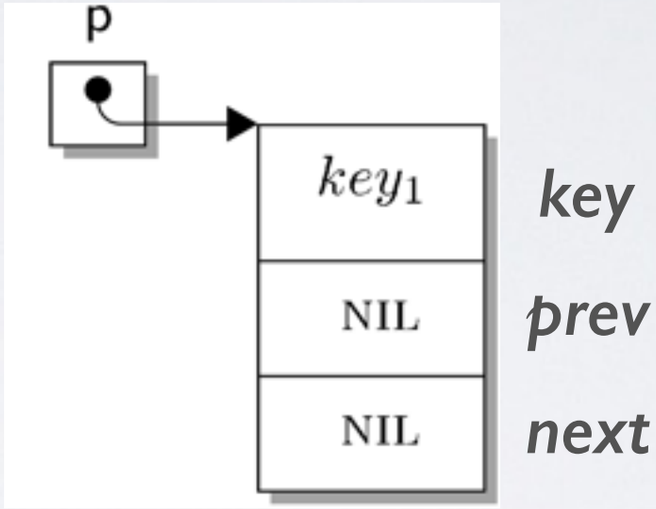
DELETE

```
delete_list(list **l, item_type x){
  list *p; /* item pointer */
  list *pred; /* predecessor pointer */
  p = search_list(*l,x);
  if (p != NULL) {
     pred = predecessor_list(*l,x);
     if (pred == NULL) /* update head */
        *l = p->next;
     else
        pred->next = p->next;
     free(p); /* free memory used by node */
  }
}
```

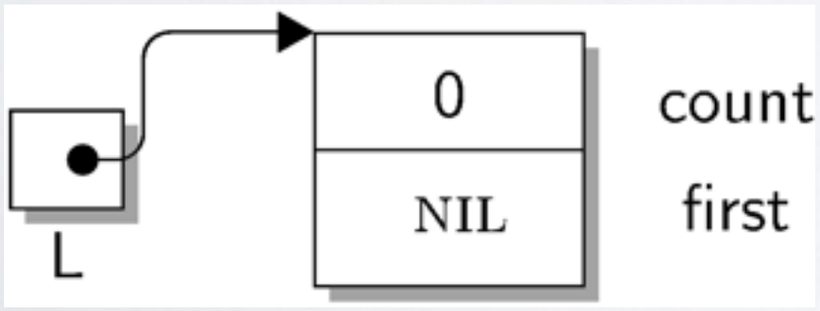
LISTA DUBLU INLANTUITA

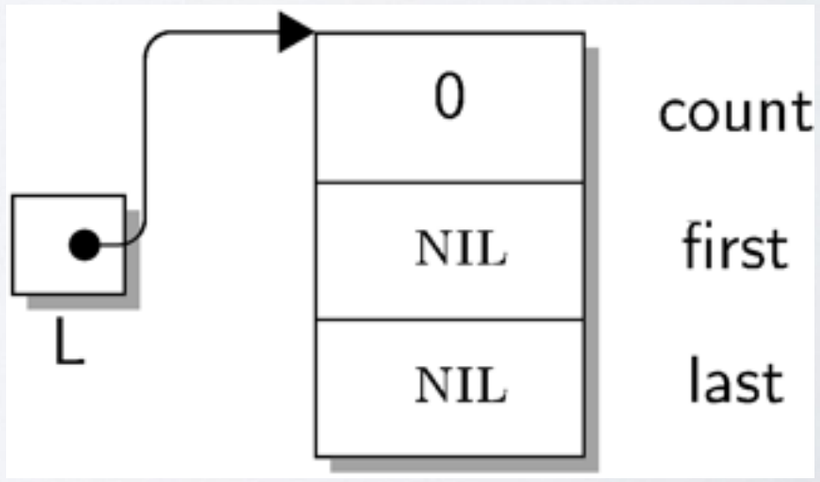
- Inlantuire bi-directionala

- *ElementLista*:



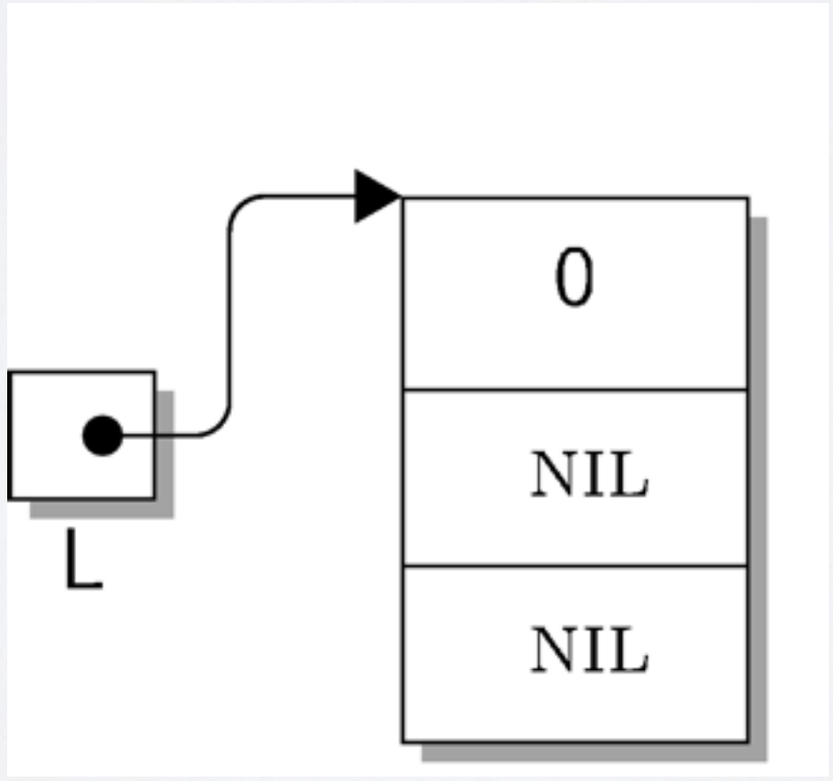
- Capul listei (*Lista*):





LISTA DUBLU INLANTUITA

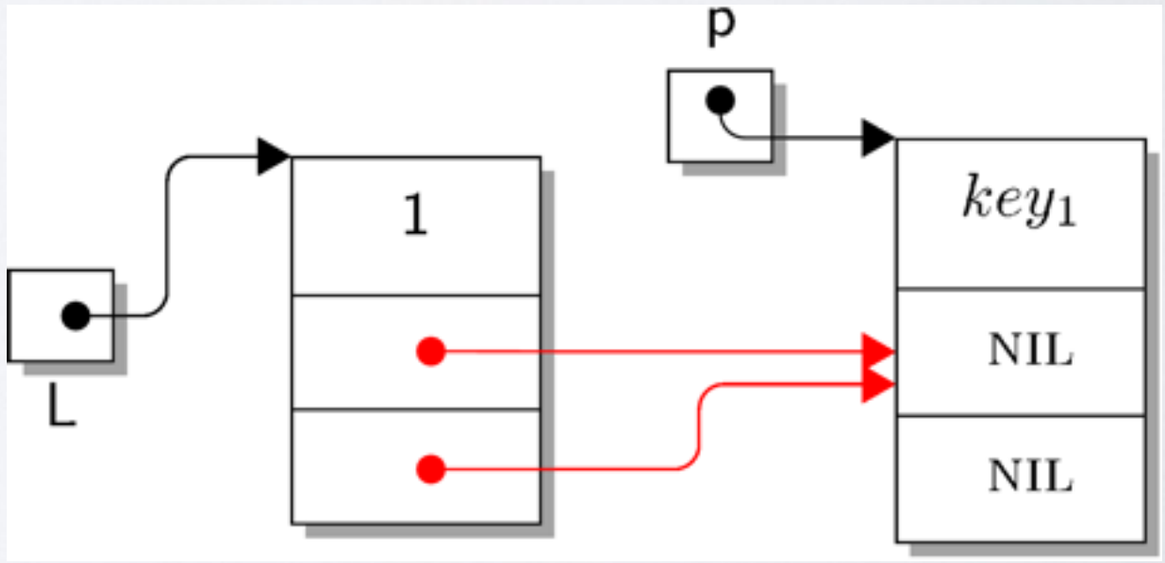
- *createEmpty()*:
  - Creeaza capul listei
  - Seteaza *L* sa pointeze catre capul listei



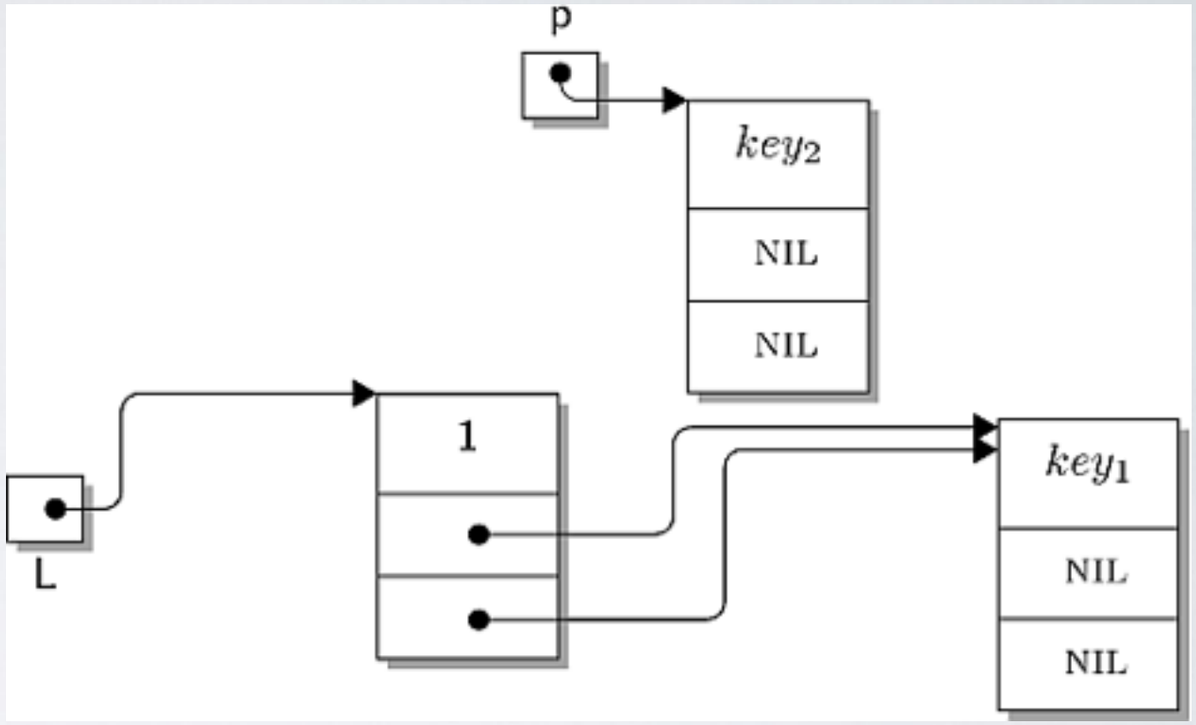
LISTA DUBLU INLANTUITA INSERT

Ca si la liste simplu inlantuite, initial trebuie creat nodul de inserat (ElementLista) !!

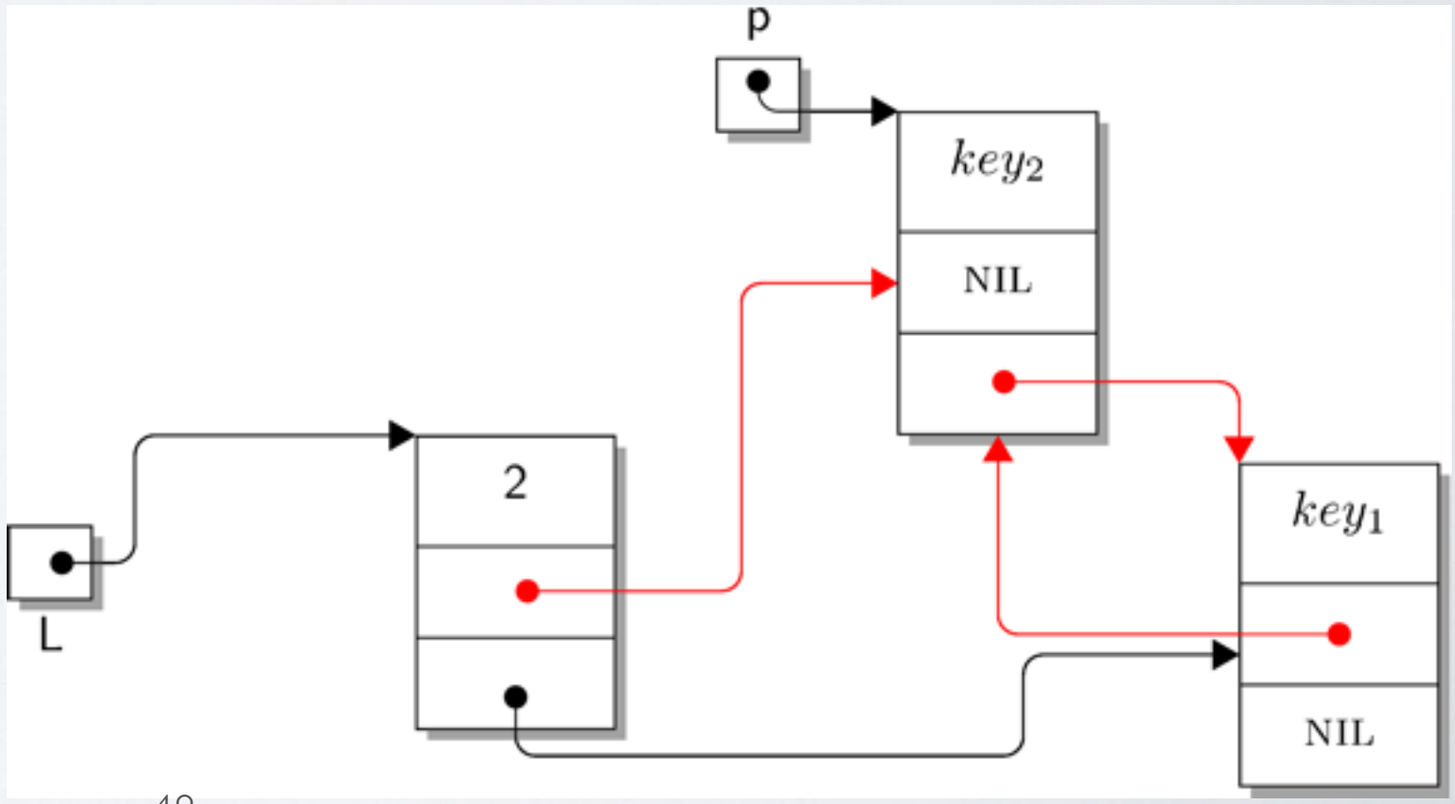
1. Inserare la inceput:
   - Lista goala
     - La fel si la *append*



LISTA DUBLU INLANTUITA INSERT



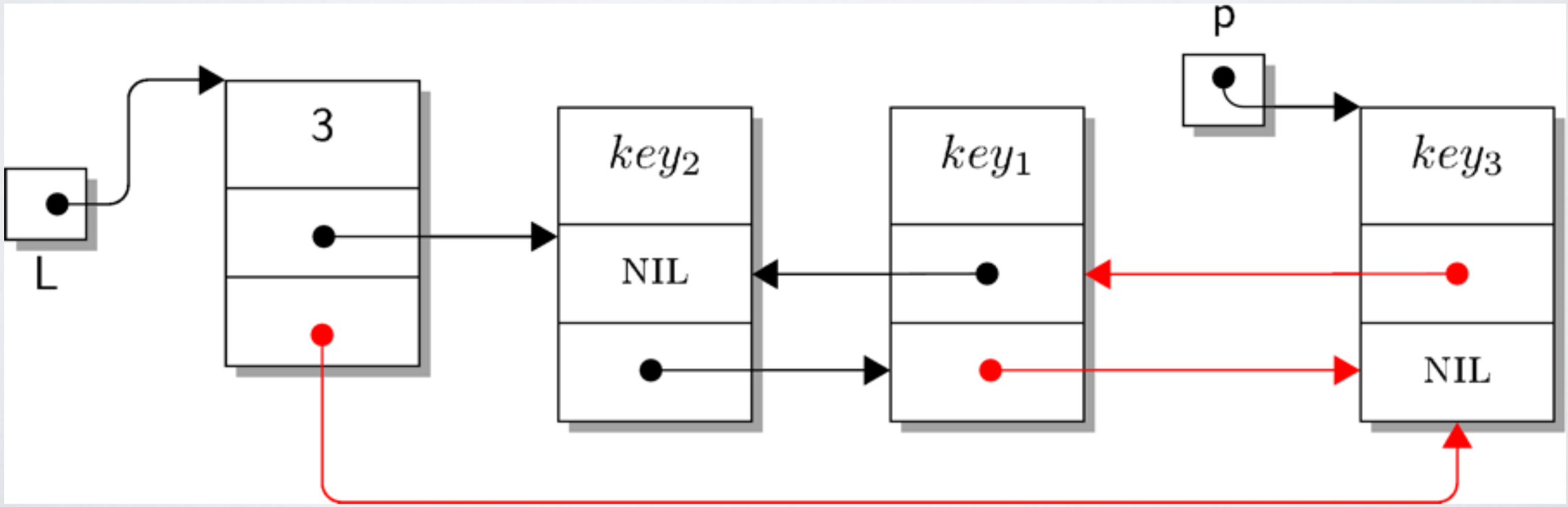
1. Inserare la inceput:
   - Lista nu e goala



LISTA DUBLU INLANTUITA INSERT

2. Inserare la final (append):

   - Lista goala - ca si la insertFirst
   - Lista nu e goala



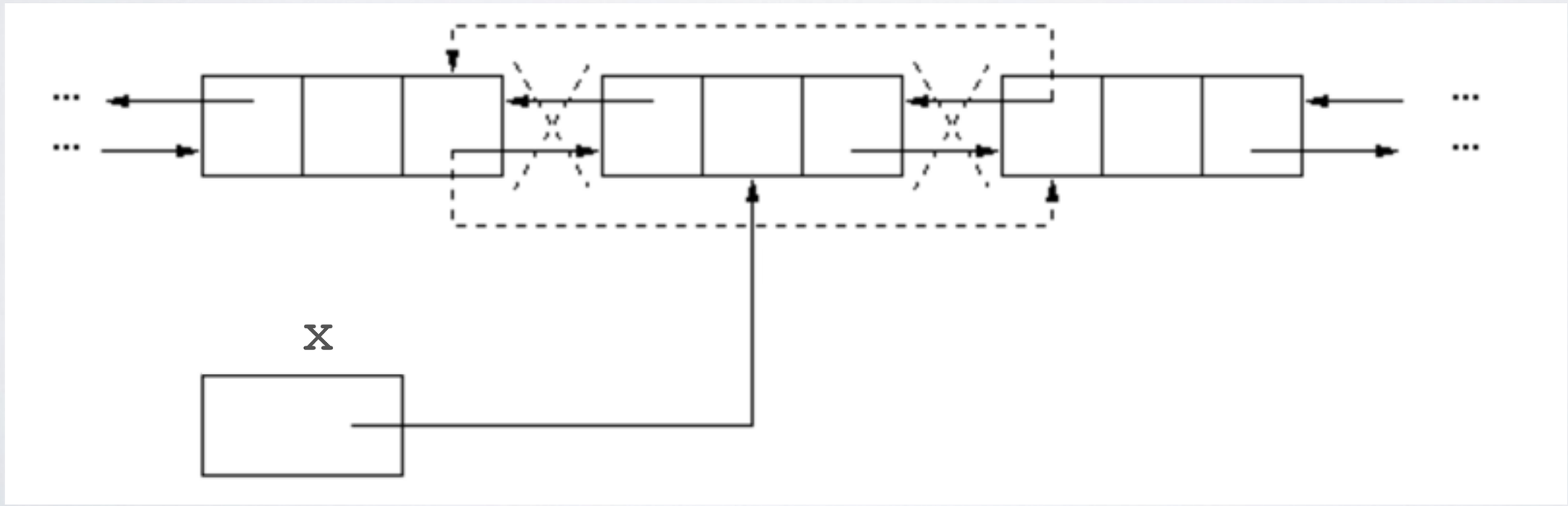
LISTA DUBLU INLANTUITA INSERT

3. Inserare in lista ordonata, sau a pozitia k
   - Lista goala: ca si mai inainte
   - Lista cu elemente:
     - Inainte de primul nod (i.e. inserare la inceput)
     - Dupa ultimul nod (i.e. append)
     - **In interiorul listei**
       - trebuie traversata lista, si stabilit un pointer (*de ce nu 2?*):
         - nodul curent (va deveni *next* pt nodul inserat)
         - nodul inserat va deveni *prev* pentru nodul curent
   - **Exercitiu!!!!**

LISTA DUBLU INLANTUITA DELETE

```
delete(L, x)
    if prev[x] != NIL then
        next[prev[x]]  = next[x]
    else
        head[L] = next[x]
    if next[x] != NIL then
        prev[next[x]] = prev[x]
```

Obs: lista are doar first (head), nu si last



LISTE INLANTUITE CU SANTINELA

- O **santinela** este un element de lista *dummy* care ne permite sa eliminam cazurile speciale (NIL).

- nu contine informatie

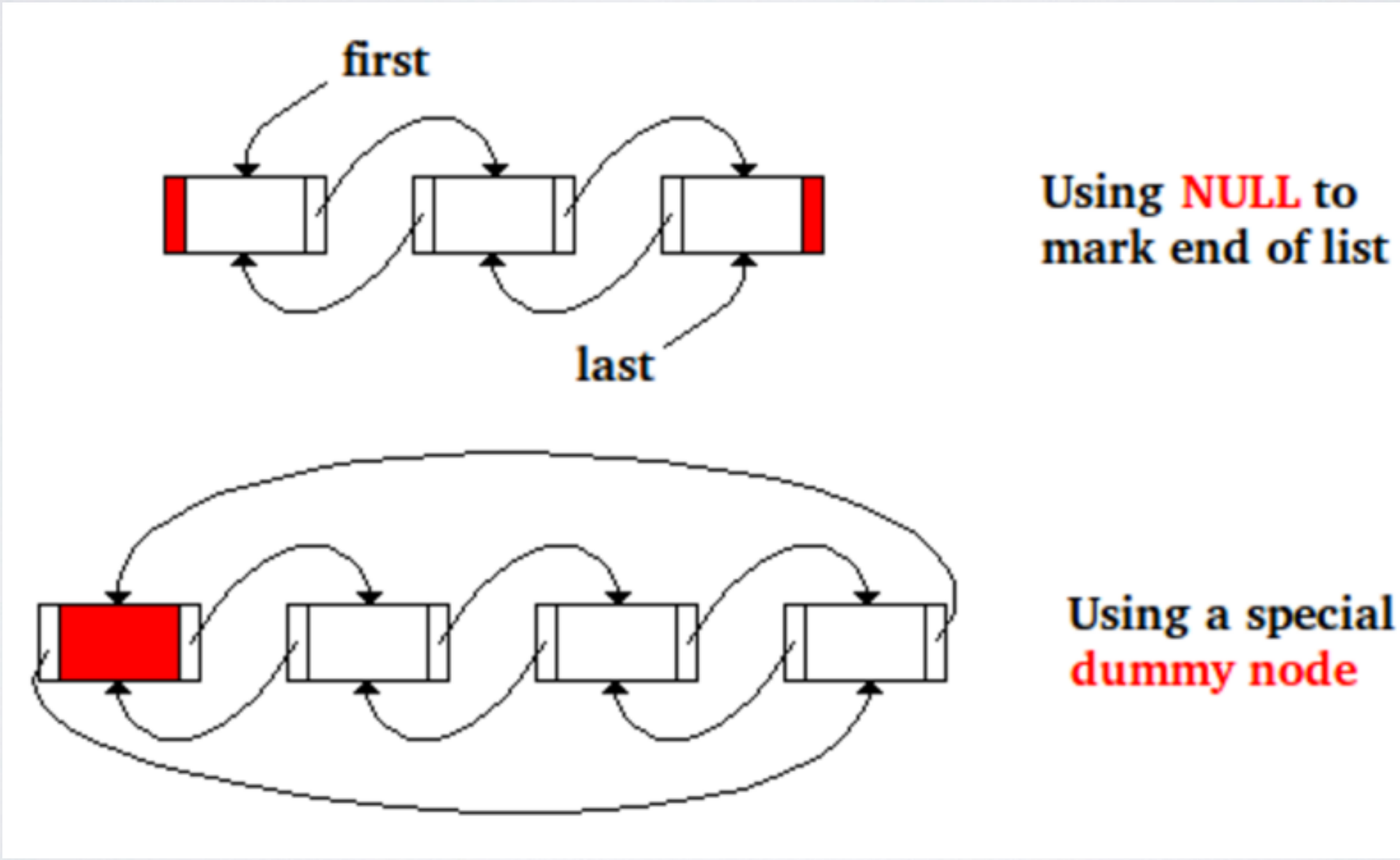
- contine toate campurile celorlalte elemente de lista

LISTA DUBLU INLANTUITA: INSERT, DELETE CU SANTINELA

```
//insert nod n inaintea lui x
n->next = x;
n->prev = x ? x->prev : last;
if (n->prev) n->prev->next = n;
        else first = n;
if (n->next) n->next->prev = n;
        else last = n;


delete(L, x)
    if prev[x] != NIL then
        next[prev[x]]  = next[x]
    else
        head[L] = next[x]
    if next[x] != NIL then
        prev[next[x]] = prev[x]
```

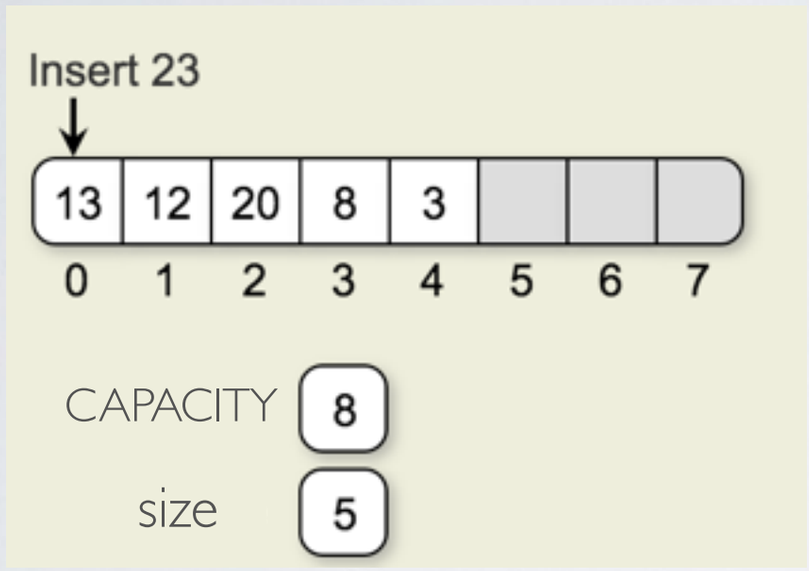
LISTA DUBLU INLANTUITA - FARA/CU SANTINELA

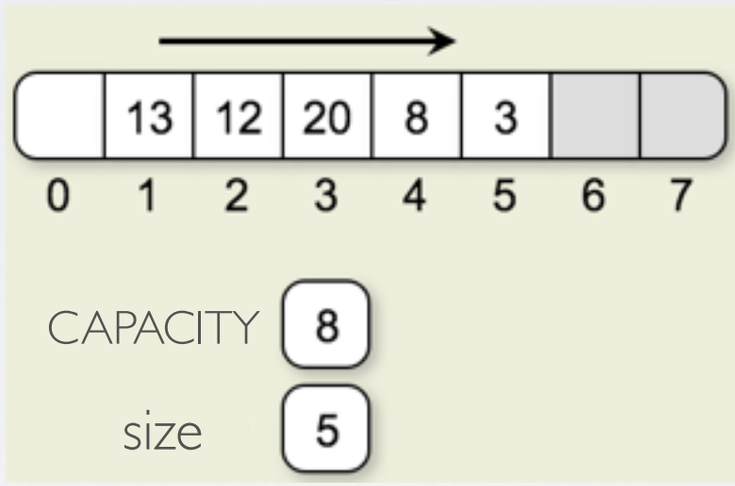


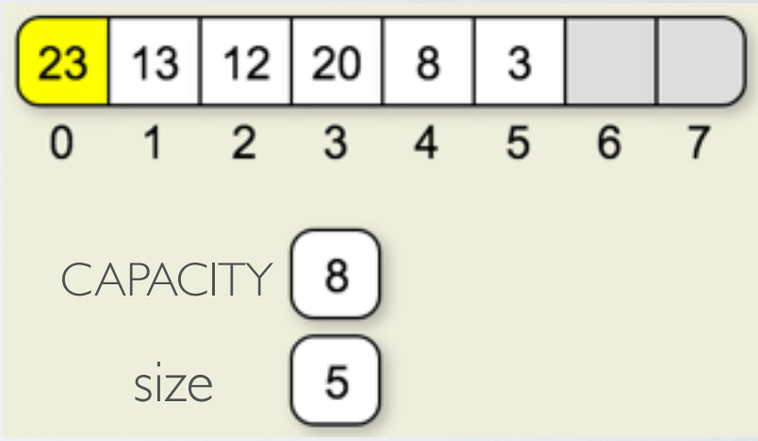
LISTA - IMPLEMENTARE VECTOR

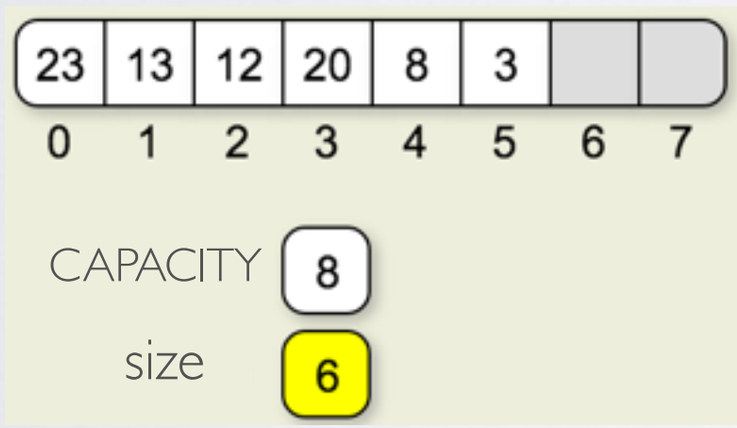
- Structura:
  - vector: int myList[CAPACITY]
  - dimensiune: int size
- Operatii:
  - search(k)
  - insertFirst(x), insertLast(x), insertK(x, k), insertOrd(x)
  - delete(x)
  - size(), capacity(),….

LISTA - IMPLEMENTARE
VECTOR - **INSERT**







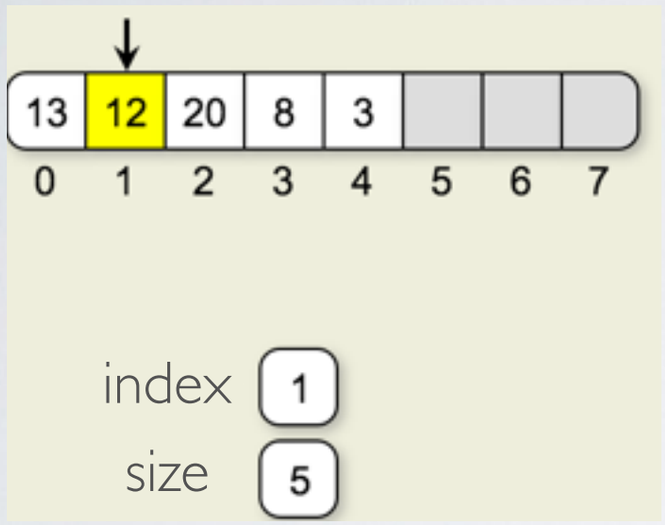


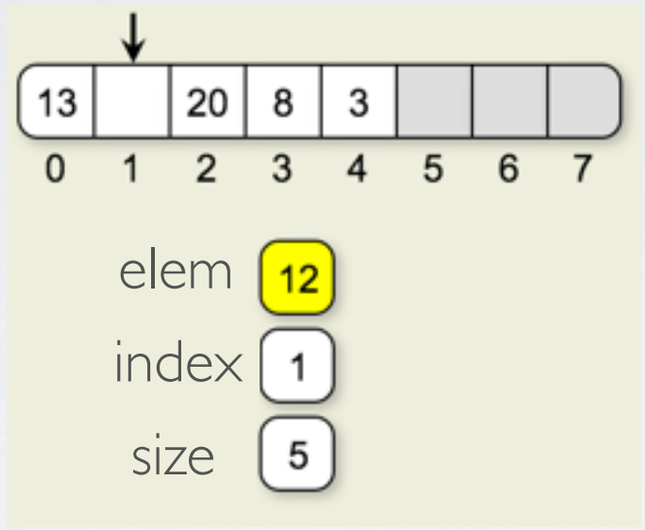
insertK(x, pos) - mutate toate elementele aflate dupa pozitia de inserare catre dreapta

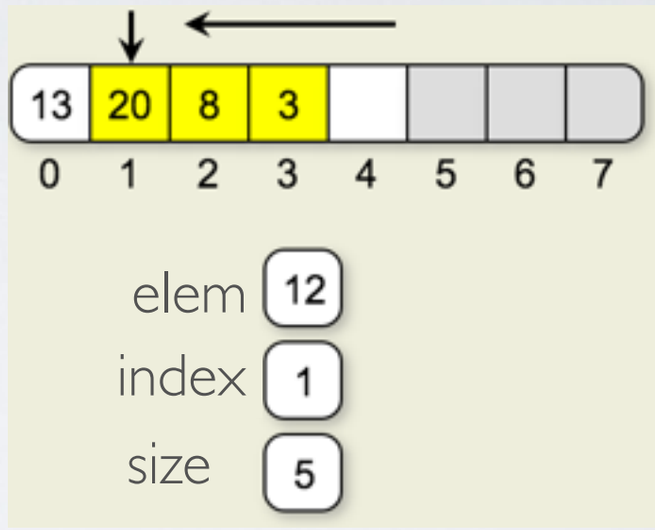
- pus elementul *x* pe pozitia *pos*
- size++

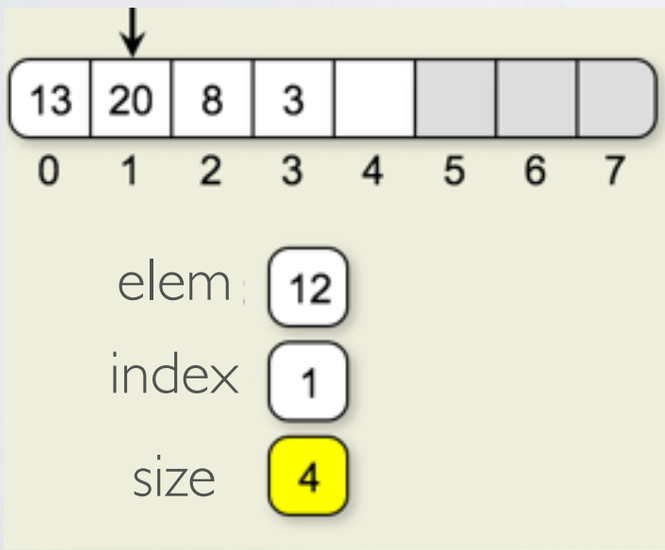
??? Ce se intampla daca size = CAPACITY

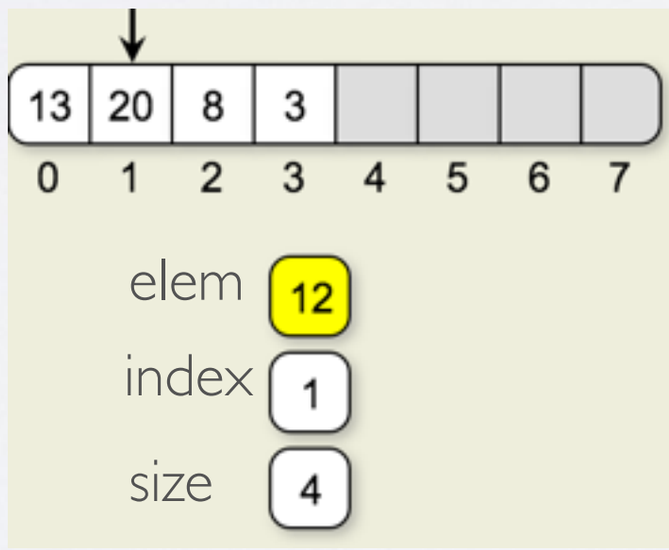
LISTA - IMPLEMENTARE VECTOR - **DELETE**











VECTOR *VS* LISTA INLANTUITA

	VECTOR	LISTA INLANTUITA
+	• timp de acces constant, dupa index (acces direct) • eficienta memorie (nu exista legaturi, alte info aditionale) • localitatea memoriei (memoria cache)	• *no overflow* (decat daca memoria e plina) • inserare si stergere eficienta • cu inregistrari mari, mutarea de pointeri este mai eficienta decat mutarea intregii inregistrari
-	• *overflow* - dimensiune fixa (vectori *dinamici*, analiza amortizata)	• memorie aditionala pt. stocarea pointerilor • accesul aleator nu e eficient • accesarea pe baza de pointeri inlantuiti nu exploateaza principiile memoriei cache

STIVA

- o colectie de elemente cu politica de acces (inserare/stergere) de tip *LIFO* (*Last-In-First-Out*)
  - elementele sunt inserate astfel incat, la orice moment, doar cel mai recent element inserat poate fi eliminat
- Inserare - la **inceput** - "push" *onto the stack*
- Stergere - de la **inceput** - "pop" *off the stack*

STIVA (ADT)

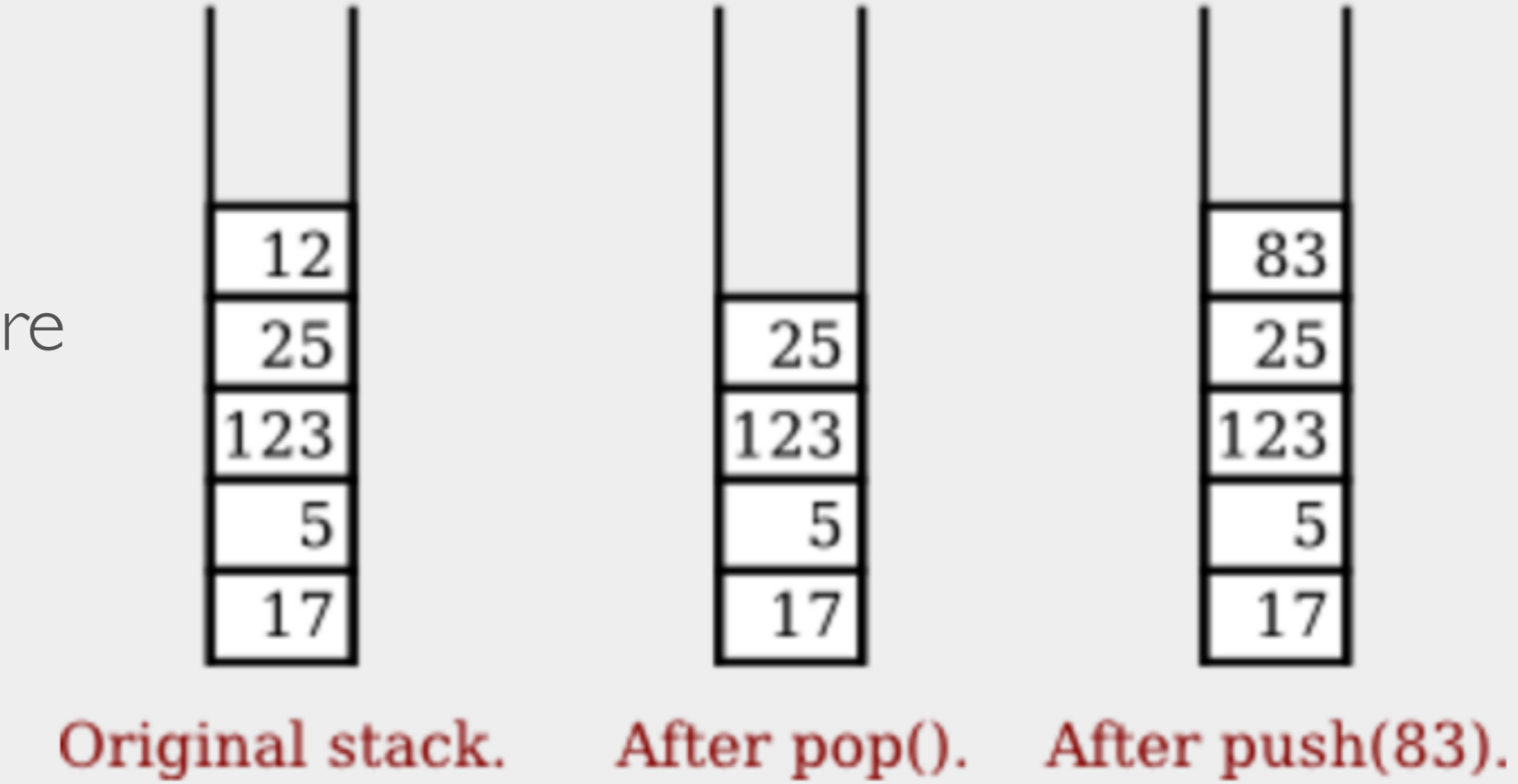
- Operatii fundamentale:
  - push(x): insereaza elementul *x in varful* stivei (en. *top, stack pointer*)
    - Intrare: ElementStiva; Iesire: nimic
  - pop(): Sterge elementul aflat in varful stivei, si il returneaza; daca stiva e goala, mesaj de eroare
    - Intrare: nimic; Iesire: ElementStiva
- Operatii aditionale (nu sunt fundamentale, dar utile):
  - size(): returneaza numarul de elemente din stiva
    - Intrare: nimic; Iesire: intreg
  - isEmpty(): semnaleaza daca stiva este goala
    - Intrare: nimic; Iesire: boolean
  - top(): returneaza elementul din varful stivei, fara a-l sterge; daca stiva este goala, mesaj de eroare.
    - Intrare: nimic; Iesire: ElementStiva

STIVA

```
PUSH(S, x)
1   top[S] ← top[S] + 1
2   S[top[S]] ← x
```

```
POP(S)
1   if STACK-EMPTY(S)
2       then error "underflow"
3       else top[S] ← top[S] − 1
4            return S[top[S] + 1]
```

STIVA - Implementare cu vector



STIVA - Implementare cu lista simplu inlantuita:

- first
- push(x) -> insertFirst(x)
- pop() -> delete head element

STIVA - RELEVANTA

- Stiva apare in programe
  - Mecanism cheie in apelul/iesirea functii/proceduri
  - Recursivitatea - stiva
    - Apel: push *stack frame*
    - Iesire: pop *stack frame*

- *Stack frame*
  - argumente functie
  - variabile locale
  - adresa de iesire

COADA

- o colectie de elemente cu politica de acces (inserare/stergere) de tip *FIFO* (*First-In-First-Out*)
- elementele sunt inserate astfel incat, la orice moment, elementul care a fost inserat la cel mai indepartat moment poate fi eliminat (cel mai *vechi*)
- Inserare: la ***sfarsit*** - *"enqueue"*
- Stergere: de la ***inceput*** - *"dequeue"*

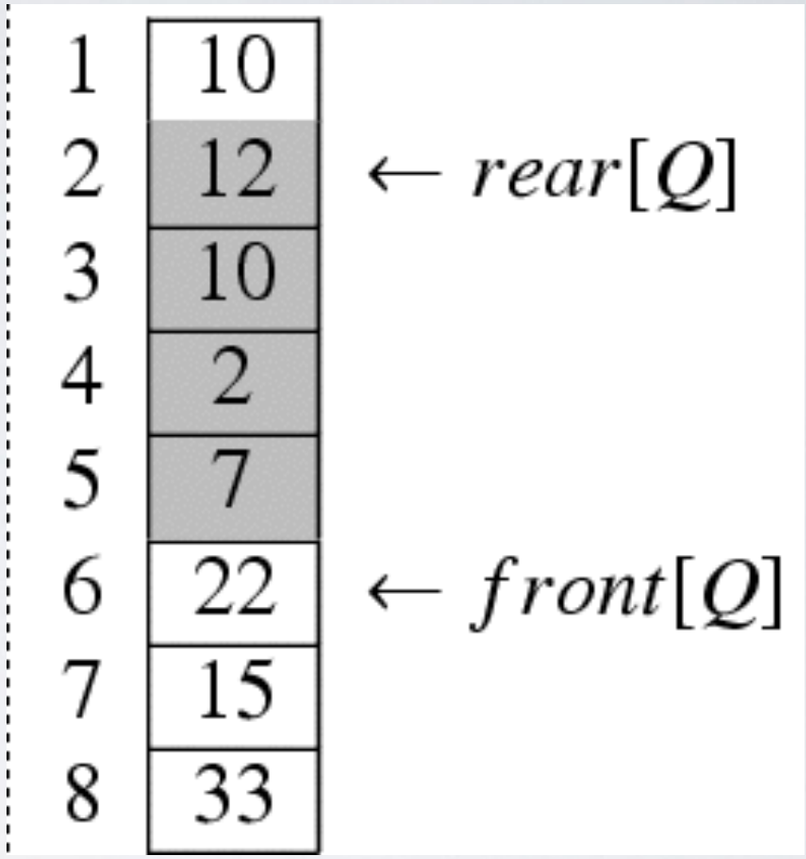
COADA (ADT)

- Operatii fundamentale:
  - enqueue(o): Insereaza elementul *o* la sfarsitul cozii
    - Intrare: ElementCoada; Iesire: nimic
  - dequeue(): Sterge elementul aflat la inceputul cozii si il returneaza; eroare daca nu exista elemente in coada
    - Intrare: nimic; Iesire: ElementCoada
- Operatii aditionale (nu sunt fundamentale, dar utile):
  - size(): returneaza numarul de elemente din coada
    - Intrare: nimic; Iesire: intreg
  - isEmpty(): semnaleaza daca coada este goala
    - Intrare: nimic; Iesire: boolean
  - front(): returneaza elementul din varful cozii, fara a-l sterge; daca coada este goala, mesaj de eroare.
    - Intrare: nimic; Iesire: ElementCoada

Coada circulara implementata ca vector

```
ENQUEUE(Q, x)
1   Q[rear(Q)] ← x
2   if rear[Q] = length[Q]
3       then rear[Q] ← 1
4       else rear[Q] ← rear[Q] + 1

DEQUEUE(Q, x)
1   x ← Q[front(Q)]]
2   if front[Q] = length[Q]
3       then front[Q] ← 1
4       else front[Q] ← front[Q] + 1
5   return x
```



Coada implementata ca lista simplu inlantuita:

- first, last
- enqueue(x) -> InsertLast
- dequeue() -> deleteFirst

DE RETINUT…

- SD - organiza, accesa si manipula date
- SD potrivita -> algoritm eficient
- ADT vs DS (abtractizare, separare what/how)
  - Lista, Stiva, Coada
  - Lista simplu/dublu inlantuita, vector

STRUCTURI CONTIGUE *VS* INLANTUITE

- Structura contigue - vectori, matrici (graf reprezentat cu matrice de adiacenta), heap-uri, tabele de dispersie.

- Structuri inlantuite - liste, arbori, grafuri reprezentate prin liste de adiacenta

STRUCTURI LINIARE *VS* IERARHICE

- Structura liniara - vector, lista

- Structuri ierarhice - heap-uri, arbori, grafuri,