STRUCTURI DE DATE SI ALGORITMI

CURS I
INTRODUCERE, ALGORITM, ANALIZA ALGORITMILOR
LISTE: LISTE SIMPLU INLANTUITE, VECTORI

AGENDA

- Prezentare curs, echipa, reguli
- Putin despre dezvoltarea algoritmilor …
- Structuri de date
    - Abstracte: Lista
    - Implementari: lista simplu inlantuita, vector

STRUCTURI DE DATE SI ALGORITMI - ECHIPA

Seria A: Raluca Brehar (curs)

Sala 6 str. G. Baritiu

Tel. 0264-401484

Raluca.Brehar@cs.utcluj.ro

Seria B+CSC: Camelia Lemnaru (curs)

Sala M03, str. G. Baritiu

Tel. 0264-401479

Camelia.Lemnaru@cs.utcluj.ro

Laborator:

Robert Varga, Ionel Giosan, Iulia Costin, Dan Toderici, Dan Domnita, Horatiu Florea, Vlad Andrei Negru

Pagini curs:

Moodle: https://moodle.cs.utcluj.ro/course/view.php?id=639

Teams seria A:

Teams seria B+CSC: https://bit.ly/3V0neci

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 ***proiecta algoritmi si structuri de date noi***, atunci cand nu exista disponibile

ORGANIZARE

- Cursuri
- participare, implicare prin intrebari, exercitii practice la tabla
- 2h – prezentare teoretica, 1h exercitii practice la tabla
- 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) marcate cu *
- Cadrul didactic de la laborator este acolo in primul rand sa va ajute, in al doilea rand sa va evalueze

EVALUARE

- Activitate **practica** pe parcursul semestrului – **30%** (laborator)
  - 0.4*Test1 + 0.6*Test2 + Bonus
    - Bonus laborator:
      - 1 punct pentru rezolvarea a minim 4 probleme extra credit - max 2 saptamani pt a fi prezentate
- Evaluare finala - **60%** (examen scris)
  - sesiunea de vara
- Quiz curs – **10%** (curs)
  - 2-3 teste **NE**anuntate, pe parcursul semestrului, in timpul cursului
- Bonus pentru participare activa la curs

BIBLIOGRAFIE RECOMANDATA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Cormen, Leiserson, Rivest, (Stein) [CLR, CLRS]: **Introduction to Algorithms**. MIT Press / McGraw Hill, (3rd, 4th editions).
  - BIBLIA de Algoritmi Fundamentali (anul II), scrisa pentru toate nivelele, cu capitole introductive, dar si detaliate (in a doua parte a cartii); pseudocod
- Kleinberg, Tardos [KT]: **Algorithm Design**, Addison-Wesley, 427 pages, 2005.
  - Resursa buna pentru tehnici de dezvoltare a algoritmilor, analiza algoritmilor, algoritmi pe grafuri, P vs. NP; pseudocod
  - http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS222/files/materials/Algorithm%20Design.pdf

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

- https://www.geeksforgeeks.org/
- visualgo.net
  - vizualizare, unealta interactiva
- http://algoviz.org/
- http://www.algolist.net/Data_structures/
- coursera.org
  - Data structures: Measuring and Optimizing Performance (U.C. San Diego)
  - …

CUPRINS CURS

1. Introducere. Algoritm. Analiza eficientei. Tip de data abstracta, structura de date.
2. Liste. Stive. Cozi
3. Arbori. Arbori binari. Arbori binari de cautare
4. Arbori binari de cautare echilibrati: AVL, arbori perfect echilibrati
5. Arbori multicai de cautare echilibrati – B-trees
6. Tabele de dispersie
7. Structuri de date pentru cozi de prioritati (Heap).
7. Structuri de date pentru multimi disjuncte
8. Grafuri: reprezentare; definitii, proprietati, problematici, traversari
9. Tehnici de dezvoltare a algoritmilor / generale de cautare: backtracking, branch and bound, greedy, divide et impera, programare dinamica
10. Sortarea

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 a relatiei dorite intre 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 $<a_1, a_2, \ldots, a_n>$
  - *Iesire:* O permutare a numerelor $<a'_1, a'_2, \ldots, a'_n>$ a.i. $a'_i \leq a'_j$, $i < j$
- Instanta: sirul <7, 5, 4, 10, 5>
- Algoritmi:
  - sortare prin selectie
  - sortare prin inserare
  - sortare rapida (Quick sort)
  - etc

UN ALGORITM….

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

"The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct." (Knuth)

PROIECTAREA ALGORITMULUI

ProiectareaAlgoritmului: (problema informala)

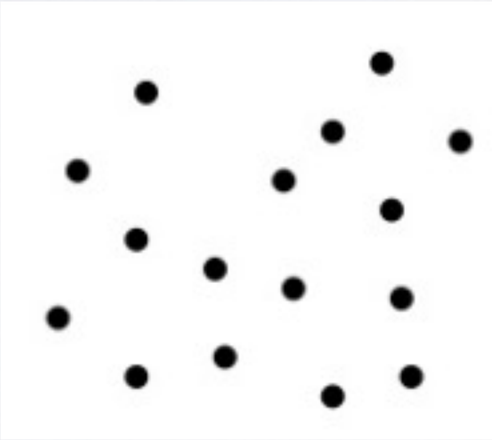
```
Se formalizeaza problema (matematic) [Pas 0]
repeta
        Se concepe algoritmul [Pas 1]
        Se analizeaza corectitudinea [Pas 2]
        Se analizeaza eficienta [Pas 3]
        Rafineaza
pana cand algoritmul este suficient de bun
returneaza algoritm
```

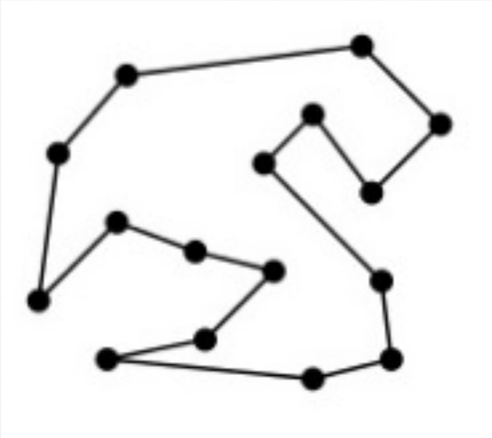
EXEMPLU DE ALGORITM - ROBOT TOUR OPTMIZATION

- Problema: *Robot Tour Optimization*

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

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





ROBOT TOUR OPTIMIZATION

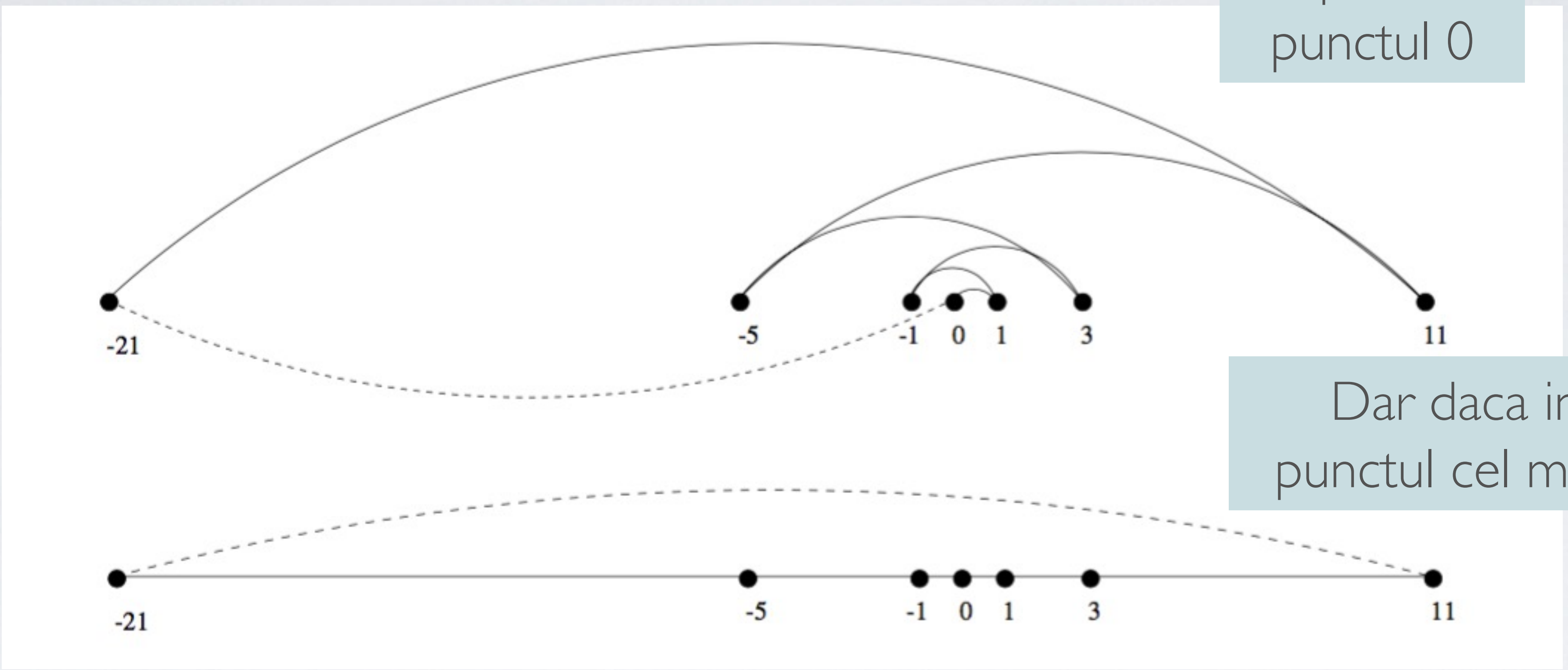
Ideea 1: Cel mai apropiat vecin

O solutie este sa incepem la un punct p0, apoi trecem la cel mai apropiat vecin al sau, p1, si de la p1 continuam cu cel mai apropiat vecin al sau, p2 s.a.m.d. pana cand au fost vizitate toate punctele.

```
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
```

Nearest Neighbor

Incepem din punctul 0



Dar daca incepem cu punctul cel mai din stanga?

Algoritmul nu gaseste neaparat turul de lungime minima!

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

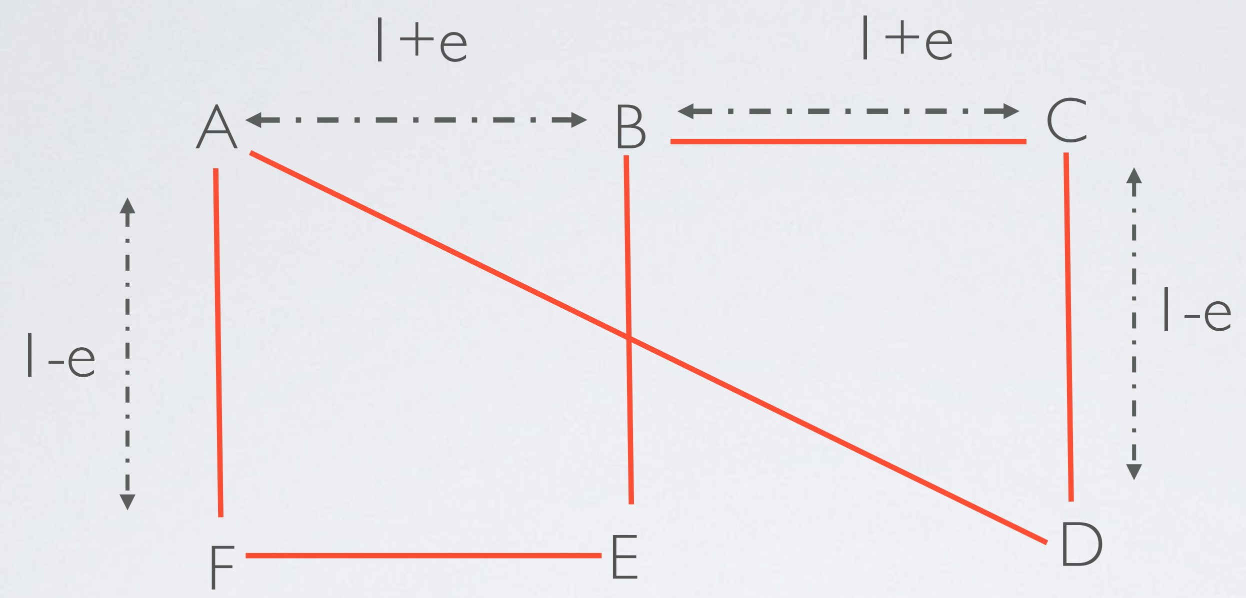
ROBOT TOUR OPTIMIZATION

Ideea 2: Cea mai apropiata pereche:

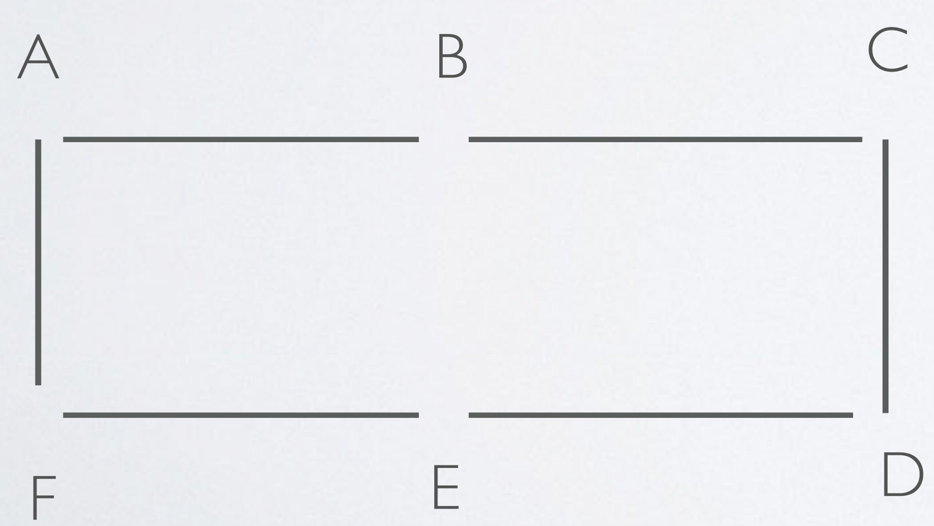
Se conecteaza repetitiv cea mai apropiata pereche de puncte a caror conexiune nu genereaza un ciclu sau o ramura cu 3 cai, pana cand toate punctele genereaza un singur ciclu.

```
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! O solutie mai buna ar fi:



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

ROBOT TOUR OPTIMIZATION

Ideea 3: Cautare exhaustiva

Incercam toate permutarile posibile si o selectam pe cea care minimizeaza lungimea totala.

```
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
```


UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

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)
  - "*Program testing can be used to show the presence of bugs, but never to show their absence!*" (E.W. Dijkstra)

  - 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 efectuate de algoritm pe acea intrare.

ANALIZA ALGORITMILOR - ESENTA

- maniera independenta de masina si limbaj
- unelte:
  - Modelul de calcul RAM (Random Access Machine)
  - Analiza asimptotica a cazului defavorabil

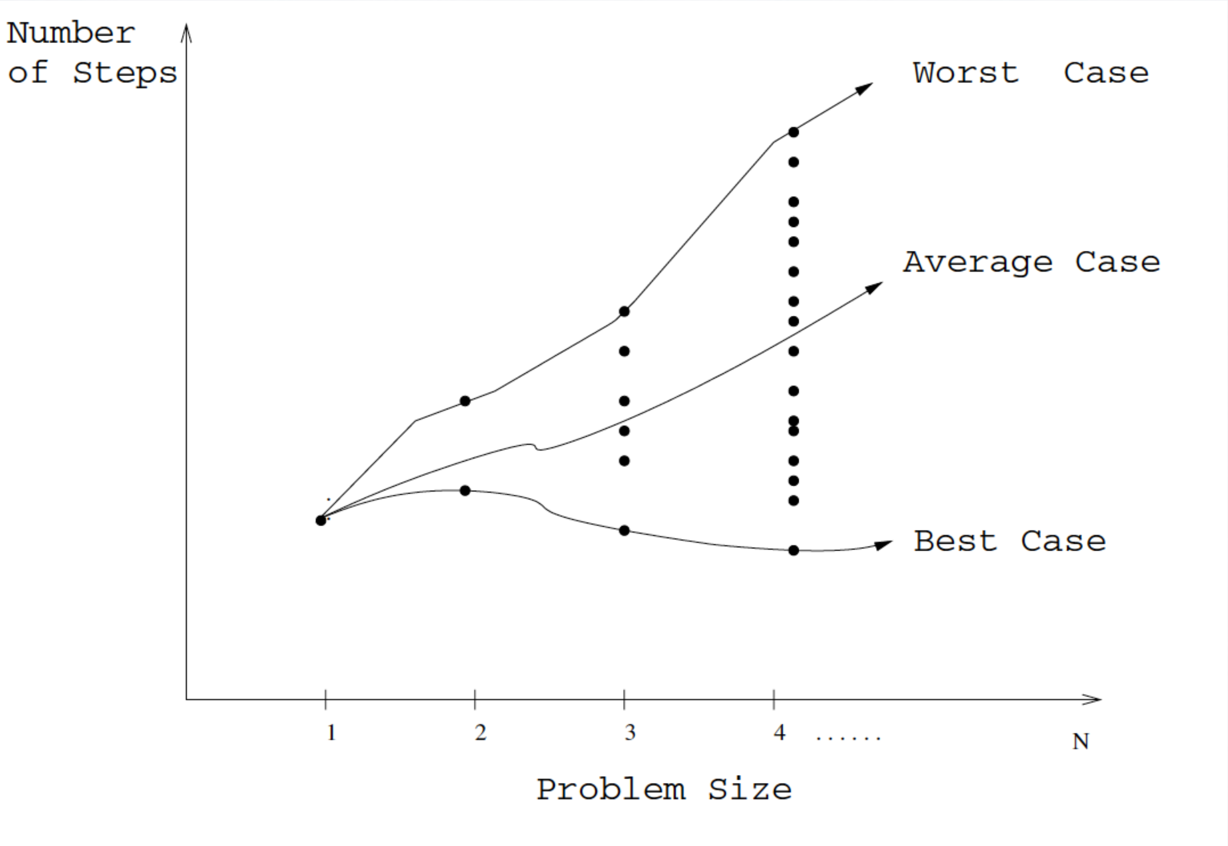
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

MODELUL RAM DE CALCUL

- orice operatie simpla (+, -, *, =, if, apel) se executa intr-o unitate de timp
- buclele si sub-rutinele nu sunt operatii simple (compozitie de mai multe operatii simple, <u>dependente de nr. de repetitii</u>)
- 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)



Imagine preluata din S. Skiena - The Algorithm Design Manual

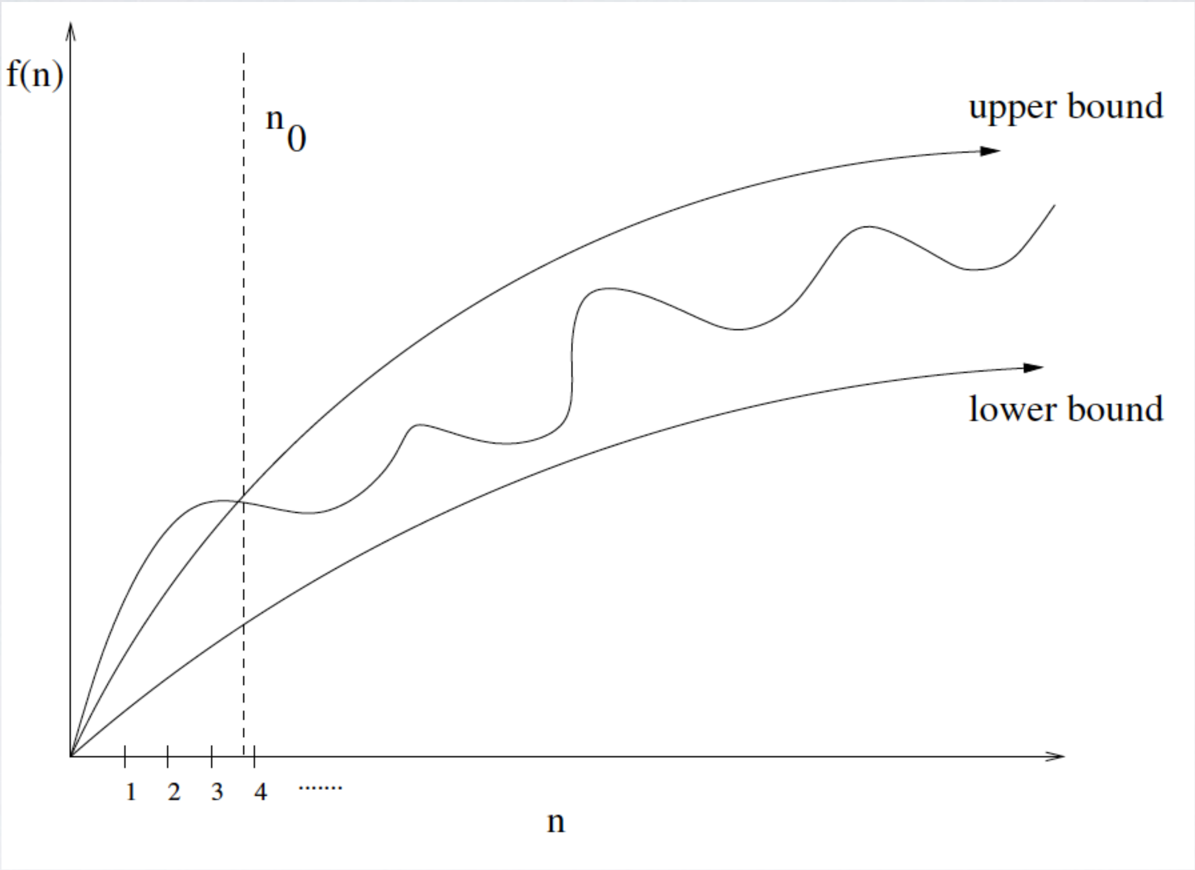
O (BIG OH)

- extrem de greu de specificat functiile de complexitate <u>exact</u>!

  - prea multe iregularitati ("spykes/bumps")
    - Cautarea binara mai rapida pe vectori de dimensiune 2^k-1

  - prea multe detalii pentru a putea fi specificate exact (detalii neinteresante de cod)

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

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

PROPRIETATI ALE LIMITELOR ASIMPTOTICE

- Tranzitivitate
  - $If\ f = O(g)\ and\ g = O(h), then\ f = O(h)$
  - $If\ f = \ \Omega(g)\ and\ g = \Omega(h), then\ f = \Omega(h)$
- Suma
  - $Suppose\ that\ f\ and\ g\ are\ two\ functions,$
    $such\ that\ for\ some\ other\ function\ h, we\ have:$
    $f = O(h)\ and\ g = O(h). Then\ f + g = O(h).$

O(1)

- O(1) descrie un algoritm care se va rula in timp (sau spatiu) constant, indiferent de dimensiunea datelor de intrare.

```
int getFirstElement(int x[], int n){
   return x[0];
}
```

Presupunem ca sirul are *n* elemente initializate.

O(n)

- O(n) descrie un algoritm al carui numar de operatii va creste liniar cu dimensiunea datelor de intrare (increment/viteza constanta).

```
int hasElement(int x[], int n, int el){
   for (int i = 0; i < n; i++)
      if (x[i] == el) return i;
   return -1;
}
```

$O(n^2)$

- $O(n^2)$ reprezinta un algoritm al carui numar de operatii este direct proportional cu patratul dimensiunii datelor de intrare.

```
int hasDuplicates(int x[], int n){
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++){
         if (i==j) continue;
         if (x[i] == x[j]) return 1;
      }
   return 0;
}
```

$O(2^n)$

- $O(2^n)$ – algoritmii ai caror numar de operatii (i.e. timp) se dubleaza cu fiecare incrementare a dimensiunii datelor de intrare.
- Curba de crestere a unei functii $O(2^n)$ este exponentiala.
- Exemplu – calculul sirului Fibonacci

```
int Fibonacci(int number){
      if (number <= 1) return number;
      return Fibonacci(number-2) + Fibonacci(number-1);
}
```

O(logn)

O(logn) reprezinta un algoritm al carui numar de operatii creste cel mult logaritmic cu dimensiunea datelor de intrare (viteza “scade”)

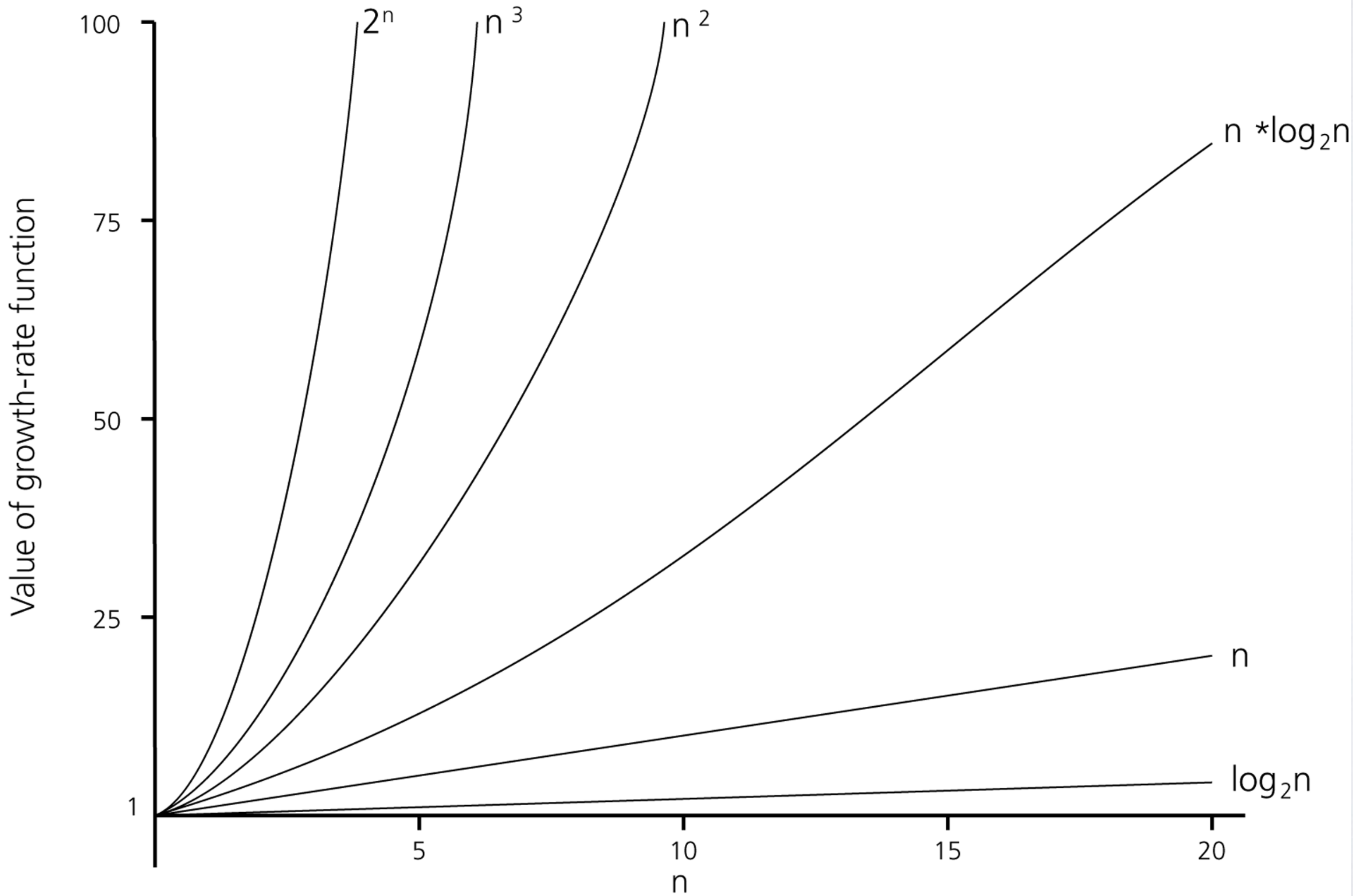
Baza logaritmului e de regula 2, dar nu conteaza, in teorie (schimbarea de baza nu modifica cresterea logaritmica)

```
int binarySearch_Left(int A[], int value, int low, int high){
   //invariants: value > A[i] for all i<low
   // and value <= A[i] for all i>high
   if (high < low) return low;
   int mid = (low+high)/2;
   if (A[mid] >= value)
       return binarySearch_Left(A, value, low, mid-1);
   else
       return binarySearch_Right(A, value, mid+1, high);
}
```

(a)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

(b)



DE LA TEORIE LA PRACTICA

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Timpii de rulare (rotunjiti in sus) pentru diferite clase de complexitate, obtinuti pe un procesor care executa 1 mil. de operatii/sec

STRUCTURI DE DATE

TIP DE DATA ABSTRACTA (ADT) *VS* STRUCTURI DE DATE (DS)

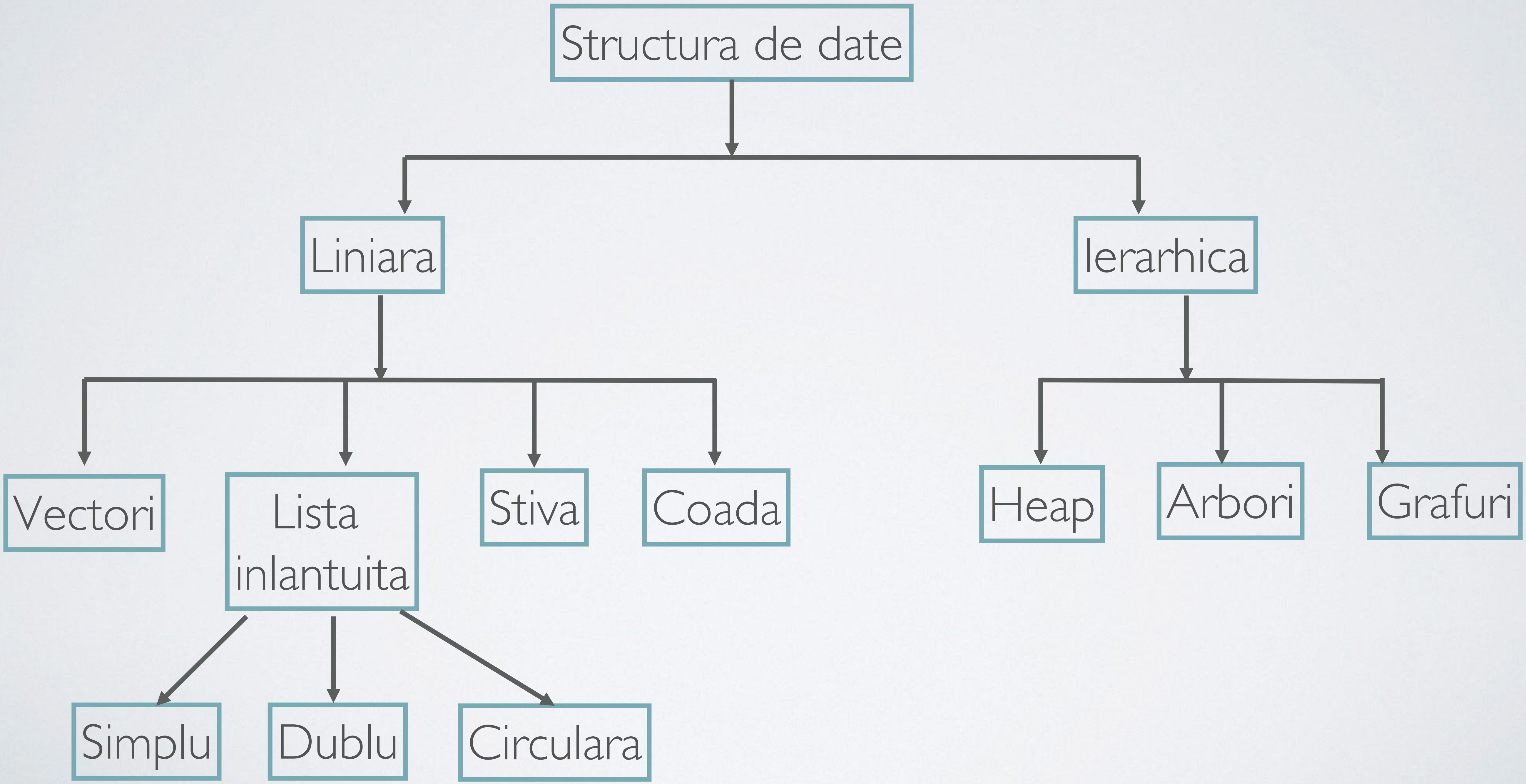
- sinonime?
- ADT - model matematic pentru *tipuri de date*
  - semantica d.p.d.v. utilizator (valori posibile, operatii) -> **comportament**!
- DS - reprezentari concrete ale datelor
  - **implementare**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

ADT *vs* DS

- *Lista* - o colectie de elemente, posibil duplicate (container); specifica o multime de operatii, ca si functionalitate (i.e. ce fac)
- O lista poate fi implementata fie folosind un *sir (array)*, fie o *lista simplu/dublu inlantuita*
- *particular - java.util.ArrayList, java.util.LinkedList,* sau *biblioteca std::list (C++, lista dublu inlantuita la baza).*

TIPURI DE STRUCTURI DE DATE



STRUCTURI DE DATE LINIARE - OPERATII

- Adaugarea unui element in structura:
  - Alocarea dinamica a memoriei pentru noul element
  - Initializarea noului element alocat
  - Legarea elementului in structura
- Cautarea unui element
- Stergerea unui element din structura:
  - Stergerea logica – eliminarea elementului prin modificarea adreselor de legatura
  - Stergerea fizica – eliberarea memoriei ocupate de elementul respectiv.
- Parcurgerea (traversarea) elementelor structurii

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

LISTA

Definitie: O lista este o secventa liniara cu un numar arbitrar de elemente (posibil duplicate), avand urmatoarele operatii fundamentale:

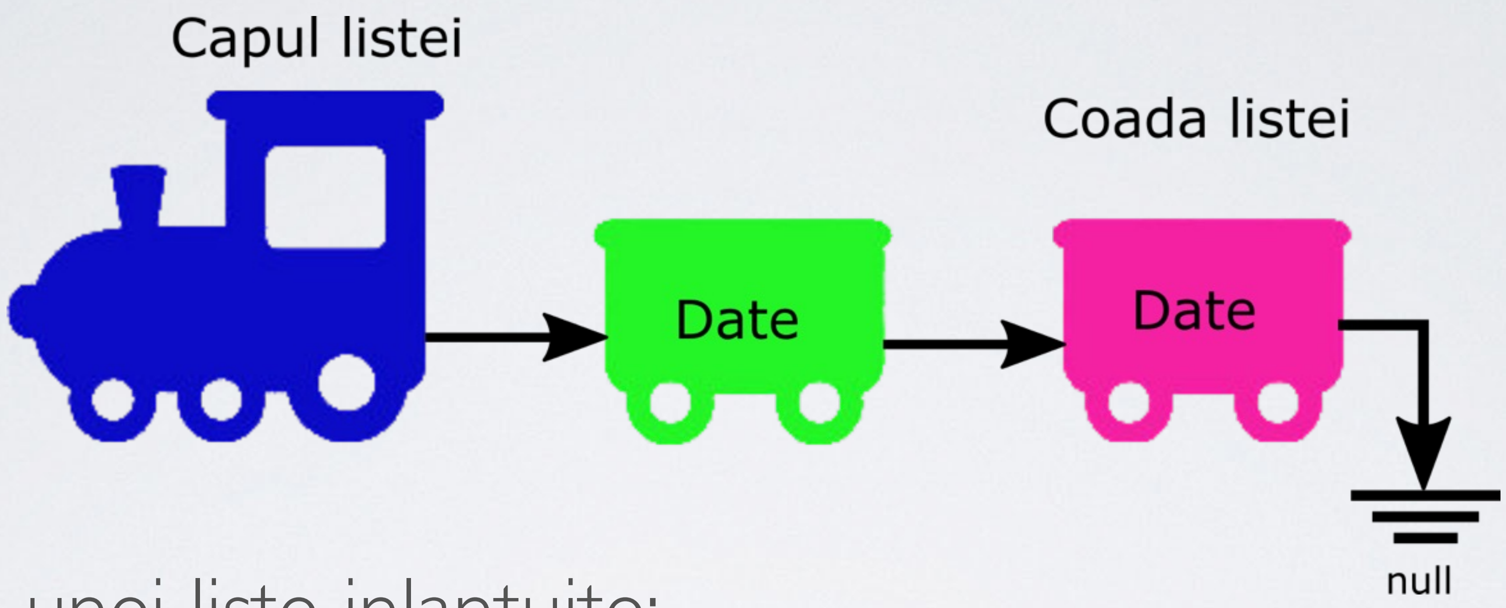
- insert(x): Adauga element *x* la inceputul listei (poate si la sfarsit, in ordine, inainte/dupa o anumita cheie)
  - Intrare: element (sau cheie, cheie_dupa, …etc); Iesire: nimic
- delete(x): Sterge element *x* (poate si primul, ultimul, cheie); eroare daca lista e goala
  - Intrare: pointer catre elementul de sters (cheia, nimic); Iesire: nimic
- search(k): cauta element care are cheia *k*
  - Intrare: cheia de cautat; Iesire: pointer catre element sau NULL daca nu s-a gasit

LISTA

Operatii aditionale:

- size(): returneaza numarul de elemente 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
  - Intrare: none; Iesire: element
- last(): returneaza, fara a sterge, ultimul element din lista; eroare daca lista este goala
  - Intrare: nimic; Iesire: element
- prev(x), next(x): returneza elementul care precede/sucede elementul x
- tail(): returneaza restul listei, fara primul element
- createEmpty(): creeaza o lista vida.

LISTA SIMPLU INLANTUITA O PRIVIRE INFORMALA



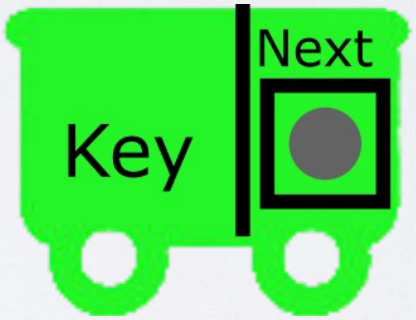
Model simplu al unei liste inlantuite:

- Avem un nod mai important – capul listei (locomotiva)
- De la capul listei se poate naviga din nod in nod (in vagoane) pana ajungem la sfarsit.
- La adaugarea si stergerea elementelor in lista e ca si cum am adauga un vagon la tren. Vagonul trebuie conectat cu vagonul (vagoanele) din vecinatate (de dinainte si/sau de dupa)!
- Cum marcam ca am ajuns dupa ultimul vagon, i.e. dupa coada listei ? – folosim NIL (null).

LISTA **SIMPLU** INLANTUITA

- Inlantuire intr-o singura directie.
- Definim o structura pentru un *element al listei* (***Nod***):

```
typedef struct node {
    int key;
    struct node *next;
} NodeT;
```



Structura ***Lista*** *are urmatoarele campuri*:

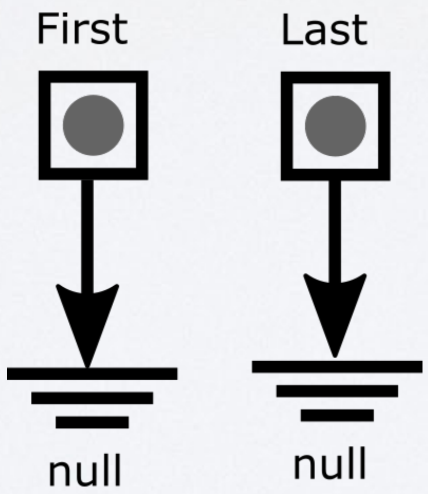
int count;   //optional

NodeT *first;

NodeT *last;   //optional

LISTA SIMPLU INLANTUITA CREARE

- *createEmpty()*:
  - first = NULL; last = NULL; count = 0;
  - Lista este goala



LISTA SIMPLU INLANTUITA INSERT

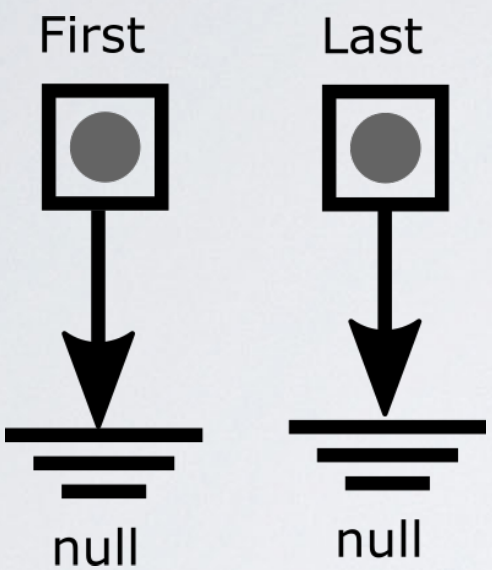
Insert - optiuni

1. la inceput
2. la sfarsit
3. inainte/dupa o cheie
4. la pozitia k
5. ordonata

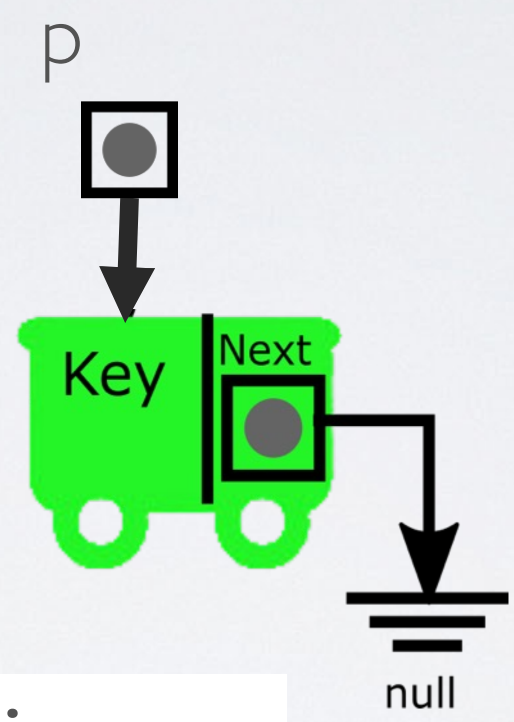
Trebuie creat intai elementul de inserat !!!

- Caz 1: adaugare la inceput, el va fi noul *first*
- Caz 2: adaugare la sfarsit (noul *last*, daca e cazul)
- Caz 3, 4 & 5:
  - cautare loc
  - refacere legaturi

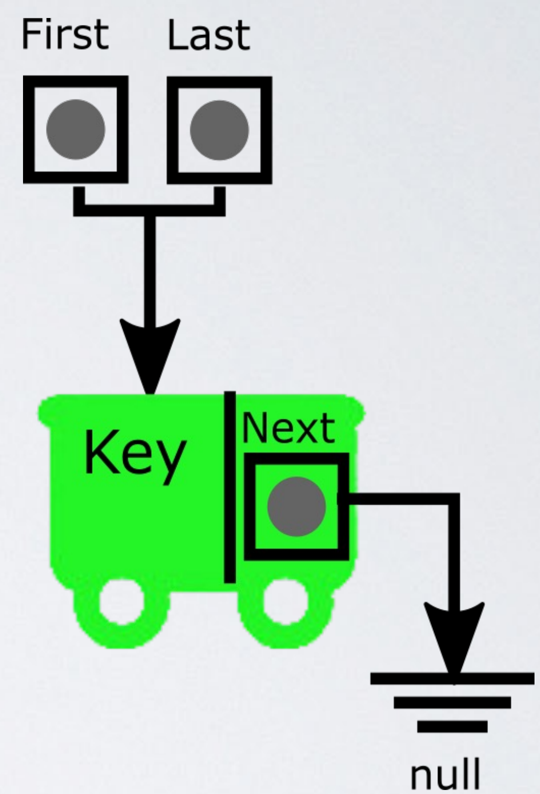
LISTA SIMPLU INLANTUITA INSERT_*: LISTA GOALA




```
NodeT *first = NULL, *last = NULL;
```




```
NodeT *p = (NodeT *) malloc(sizeof(NodeT));
p->key = Key;
p->next = NULL;
```

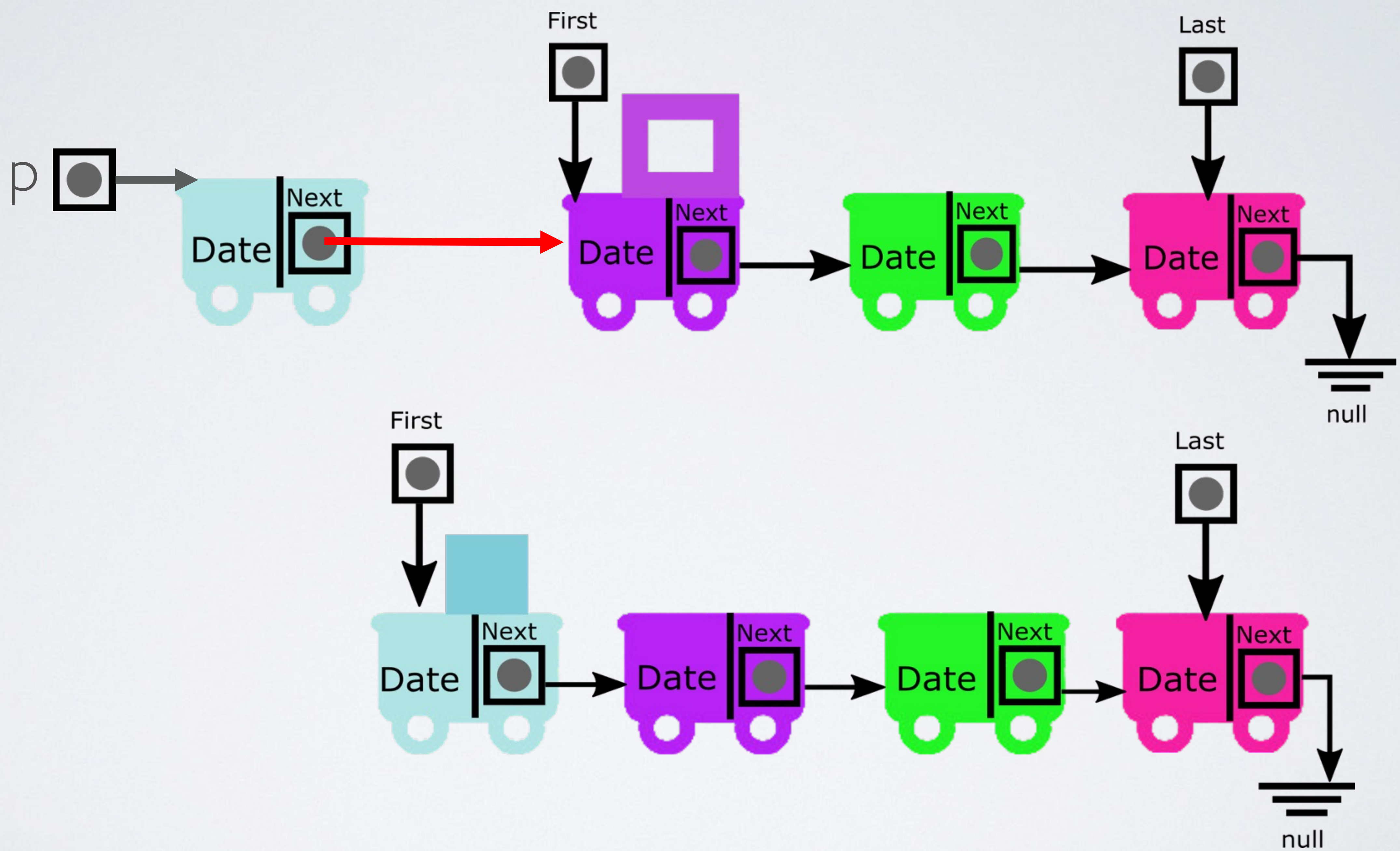



```
if (first == NULL)
{   /* empty list */
    first = p;
    last    = p;
}
```

I. Adaugare la inceput

- Lista nu e goala

```
if (first != NULL)
{
     p->next  = first;
     first    = p;
}
```

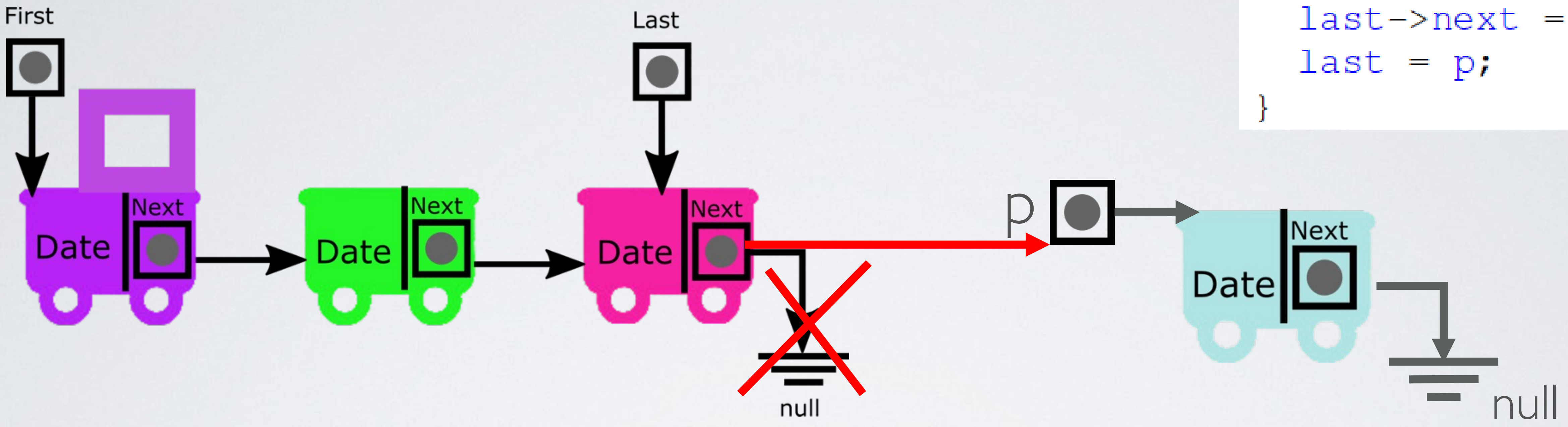


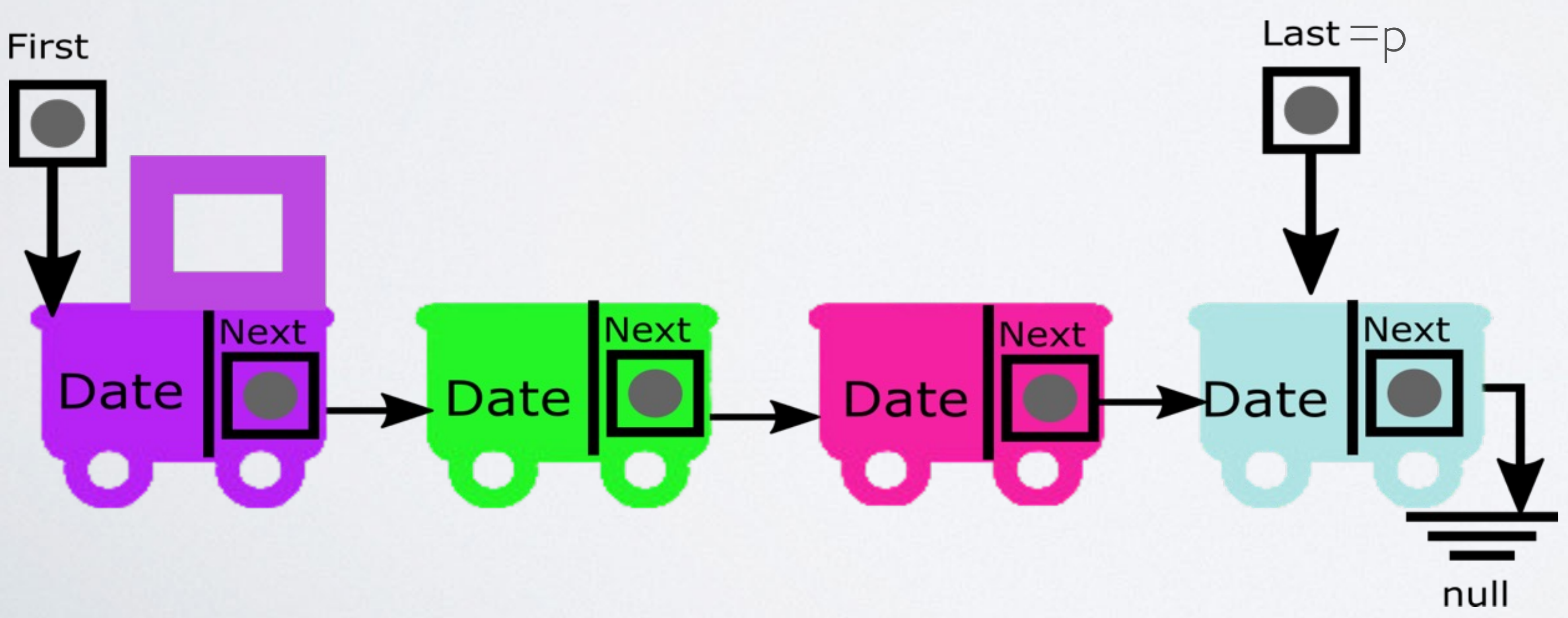
2. Adaugare la final (*append*)

Lista goala: la fel ca si la inserare la inceput

Lista care contine elemente:

```
if ( last != NULL )
{
  p->next = NULL;
  last->next = p;
  last = p;
}
```

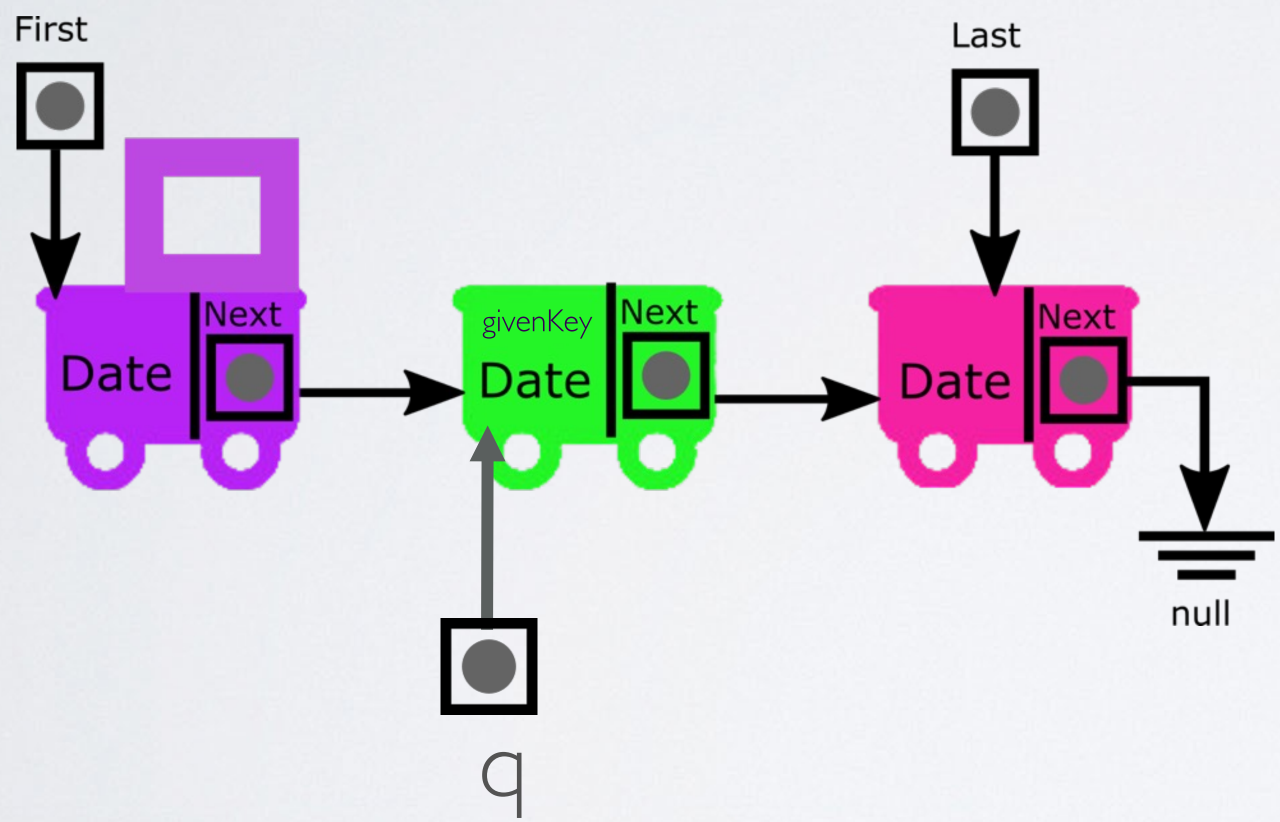




LISTA SIMPLU INLANTUITA INSERT DUPA UN NOD

Inserarea dupa un nod care are cheia *givenKey:*

1. Se cauta nodul care contine cheia *givenKey*

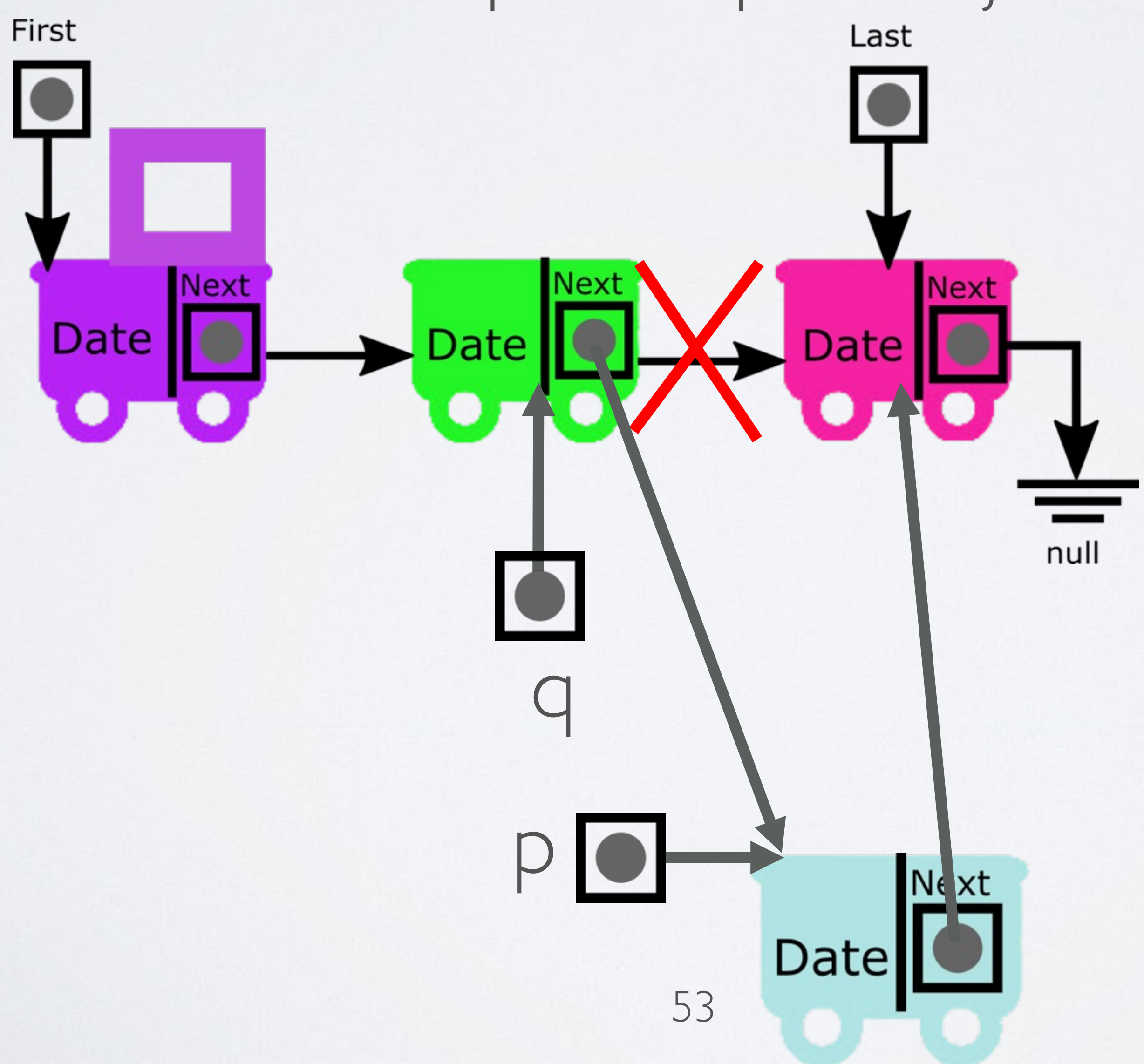



```
NodeT *q;
q = first;
while(q != NULL){
    if (q->key == givenKey) break;
    q = q->next;
}
```

LISTA SIMPLU INLANTUITA INSERT DUPA UN NOD

Inserarea dupa un nod care are cheia *givenKey:*

1. Se cauta nodul care contine cheia *givenKey* (*q* in exemplul de cod)
2. Se insereaza nodul de pointer p si se ajusteaza legaturile.

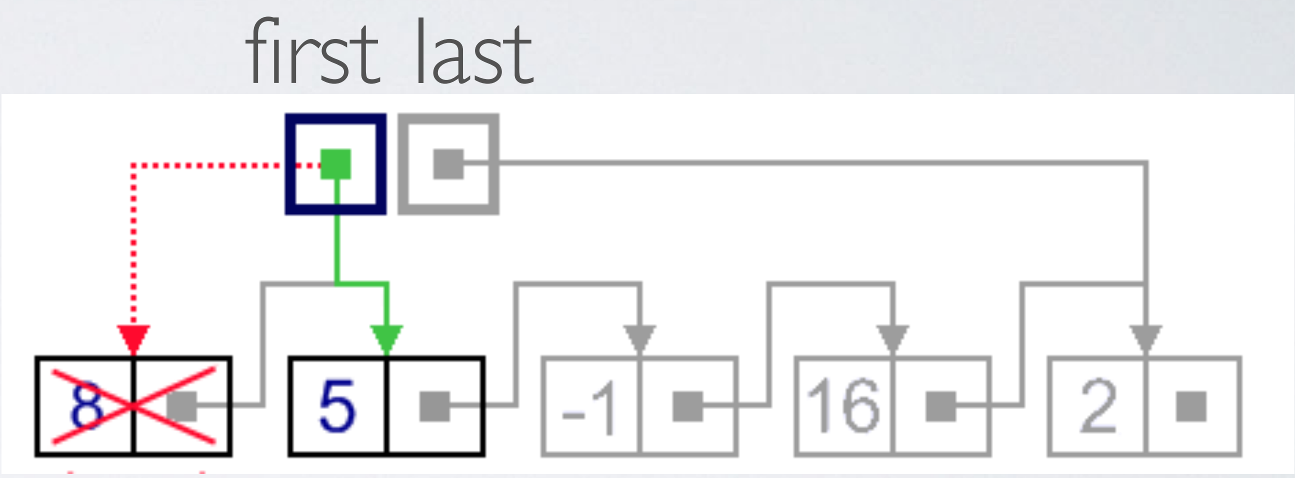


Cod?

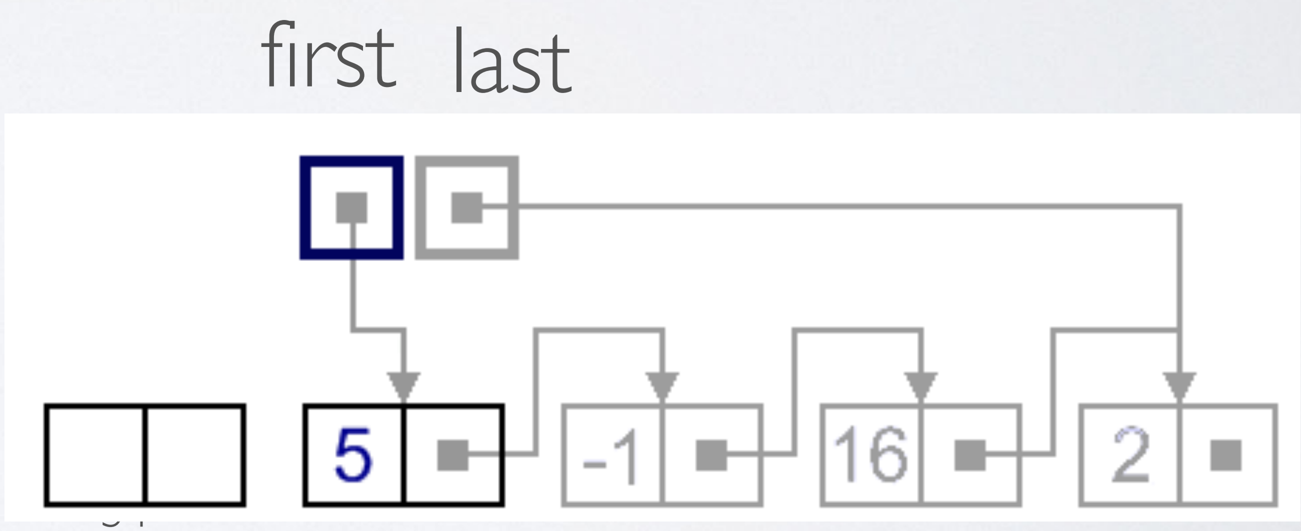
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

LISTA SIMPLU INLANTUITA DELETE – PRIMUL ELEMENT

I. Actualizam nodul first ca sa pointeze catre nodul urmator lui.



Ce se intampla daca lista contine doar 1 element inainte de stergere?

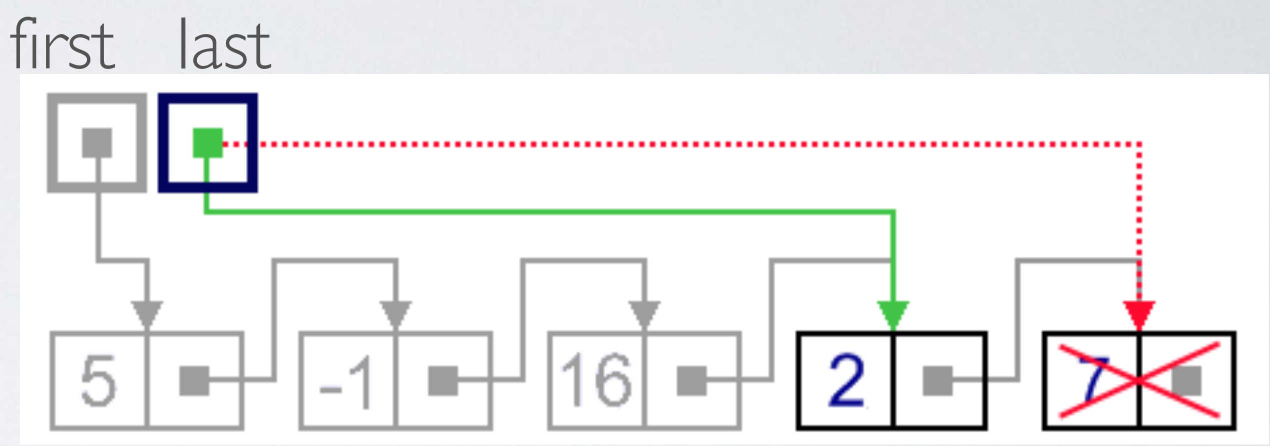


II. Stergem efectiv nodul.

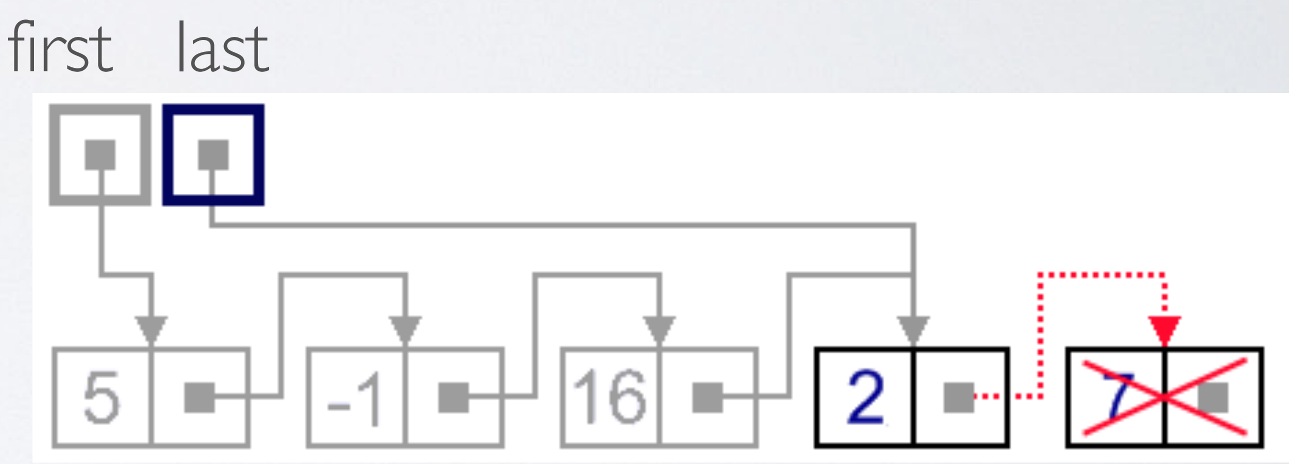
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

LISTA SIMPLU INLANTUITA DELETE – ULTIMUL ELEMENT

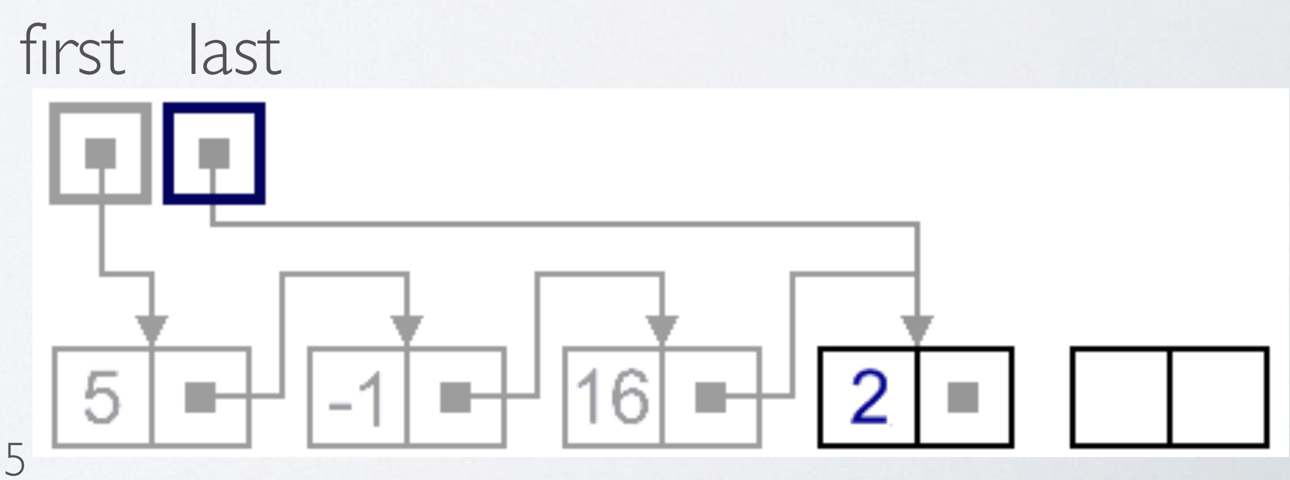
1. Actualizeaza last sa pointeze la nodul anterior lui. E nevoie de o parcurgere a listei !



2. Se modifica pointerul next al noului element last sa pointeze la NULL.



3. Se sterge fizic vechiul element last.

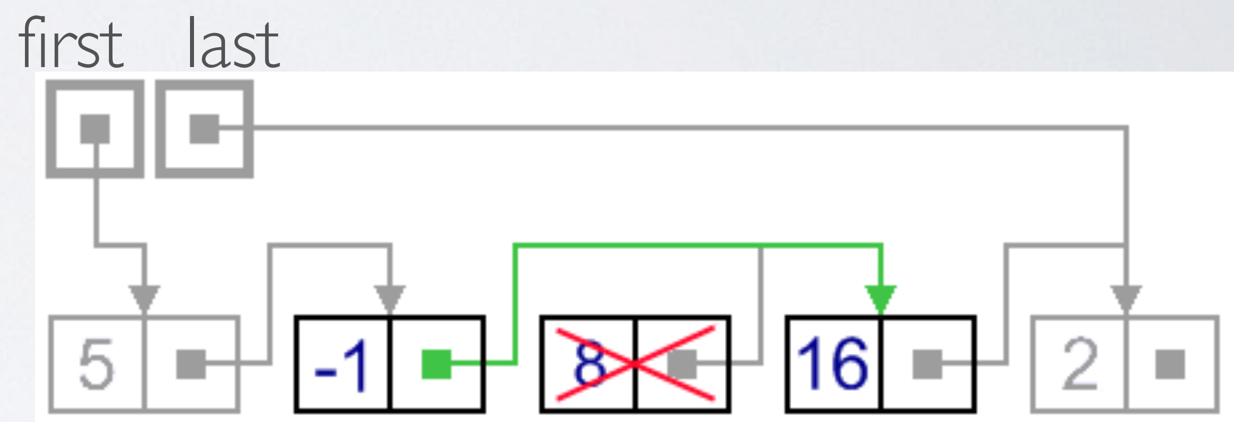


Ce se intampla daca lista contine doar 1 element inainte de stergere?

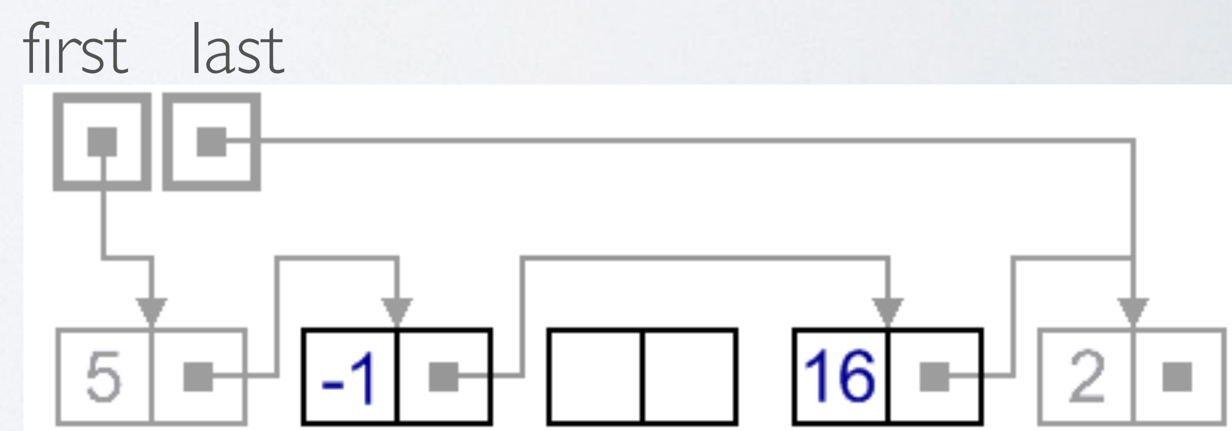
LISTA SIMPLU INLANTUITA DELETE – CAZ GENERAL

Elementul apare intre doua alte elemente existente! Nu se actualizeaza *first* si *last*.

1. Actualizeaza campul next al elementului anterior sa pointeze catre elementul urmator al elementului care se sterge – necesita parcurgere!



2. Se sterge fizic elementul.

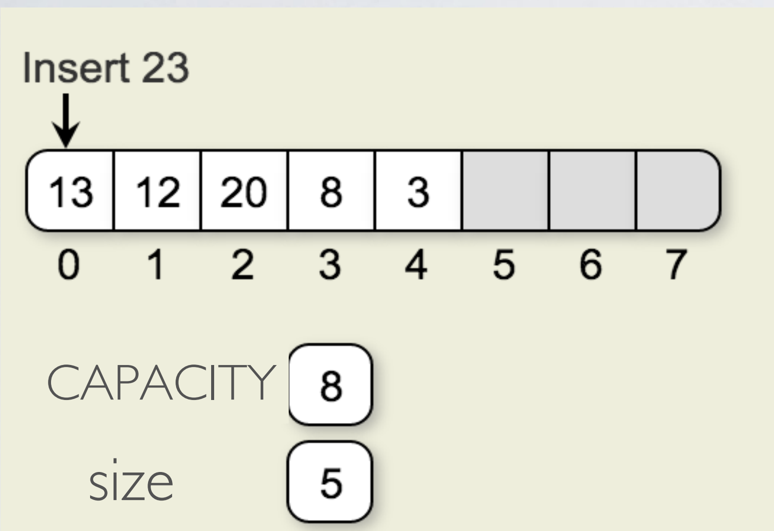


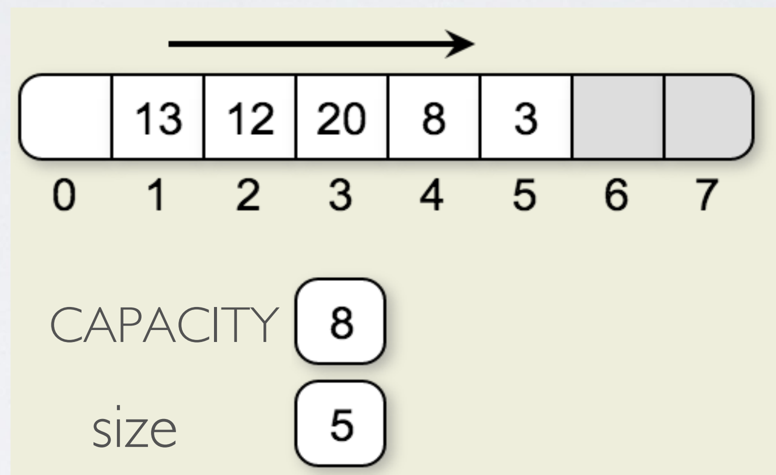
LISTA - IMPLEMENTAREA SECVENTIALA (VECTOR **STATIC**)

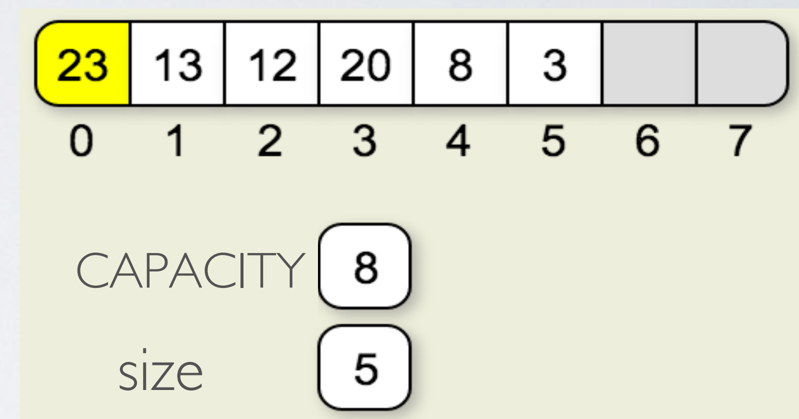
- Structura:
  - vector: int myList[CAPACITY]
  - dimensiune: int size
- Operatii:
  - search(k)
  - insert_first(x), insert_last(x), insert_before_k(x, k), insert_after_k(x, k), insert_ord(x)
  - delete_first(), delete_last(), delete(x)
  - size(), capacity(),….

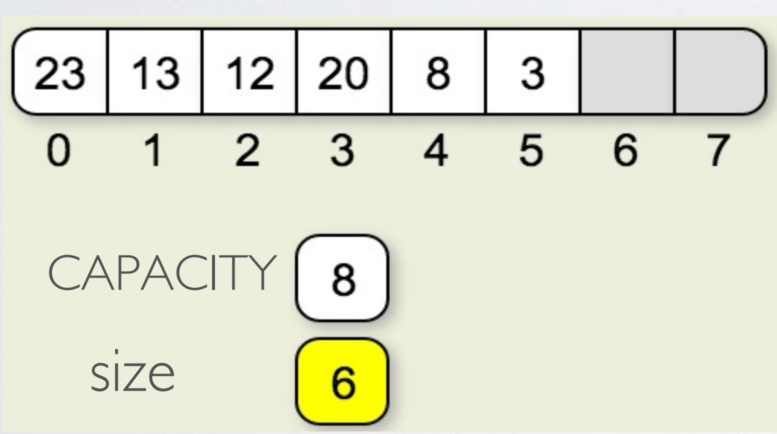
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

LISTA - IMPLEMENTARE VECTOR – **INSERT_FIRST(23)**







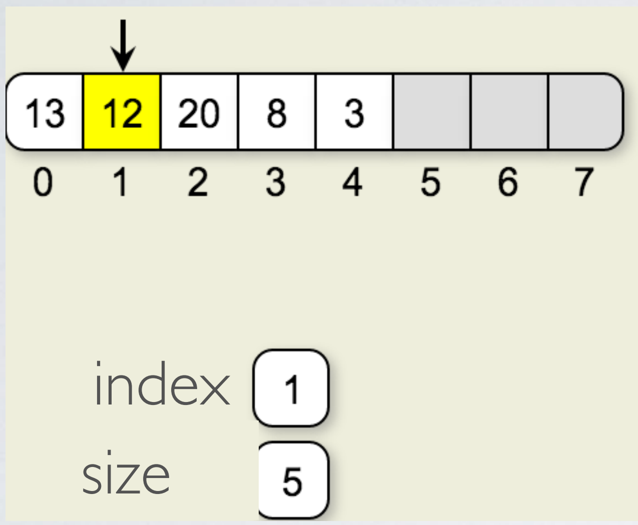


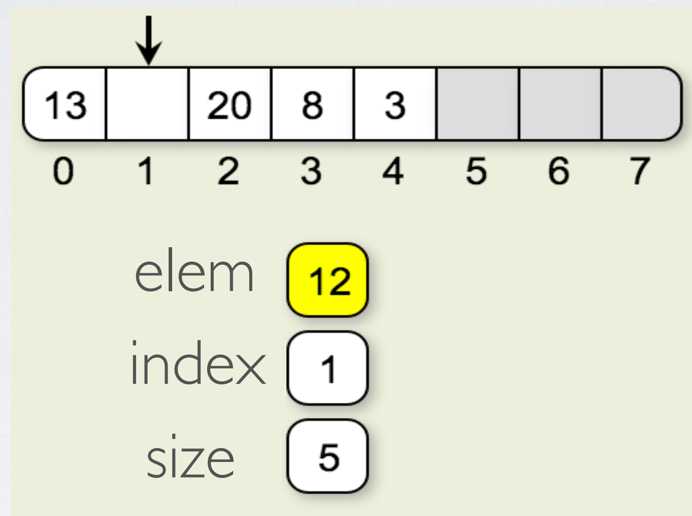
- mutate toate elementele aflate dupa pozitia de inserare catre dreapta

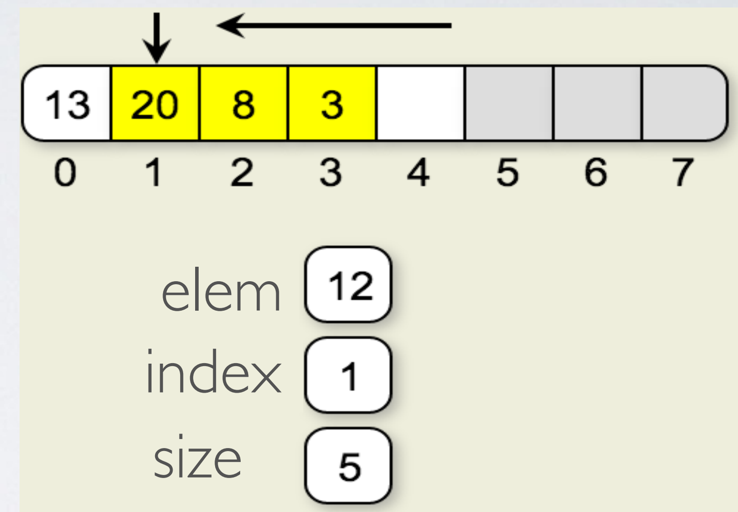
- *pus elementul x pe pozitia pos*
- *size++*

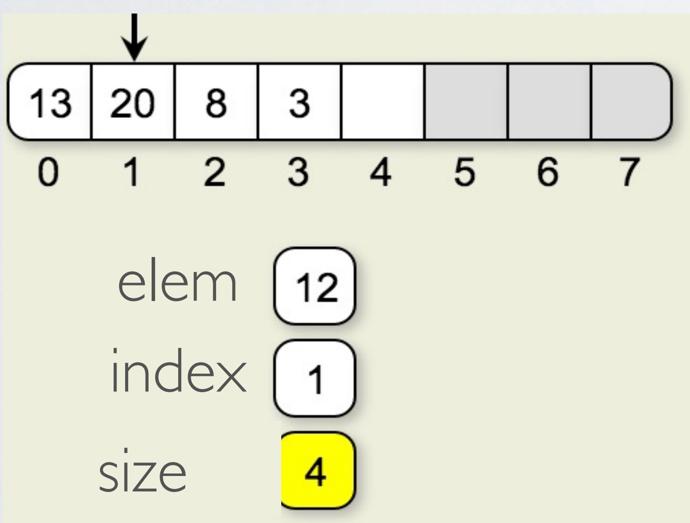
??? Ce se intampla daca size = CAPACITY ???

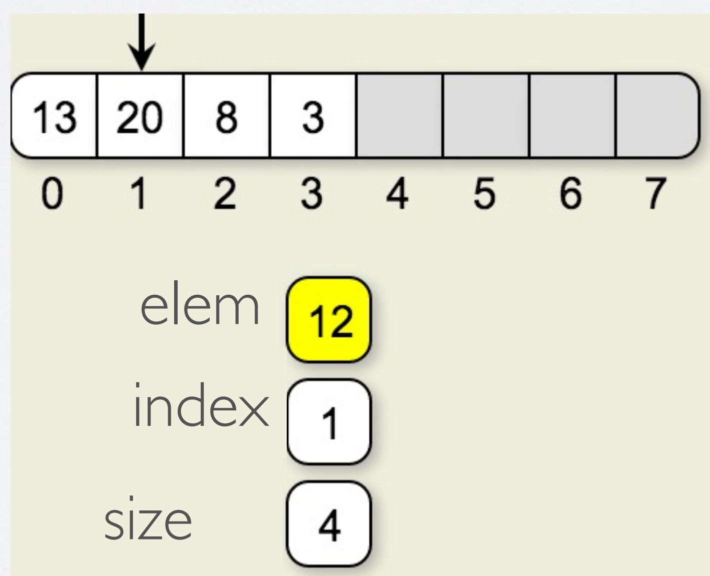
LISTA - IMPLEMENTARE VECTOR – **DELETE(12)**











UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

VECTOR STATIC *VS* LISTA SIMPLU INLANTUITA (EFICIENTA OPERATII)

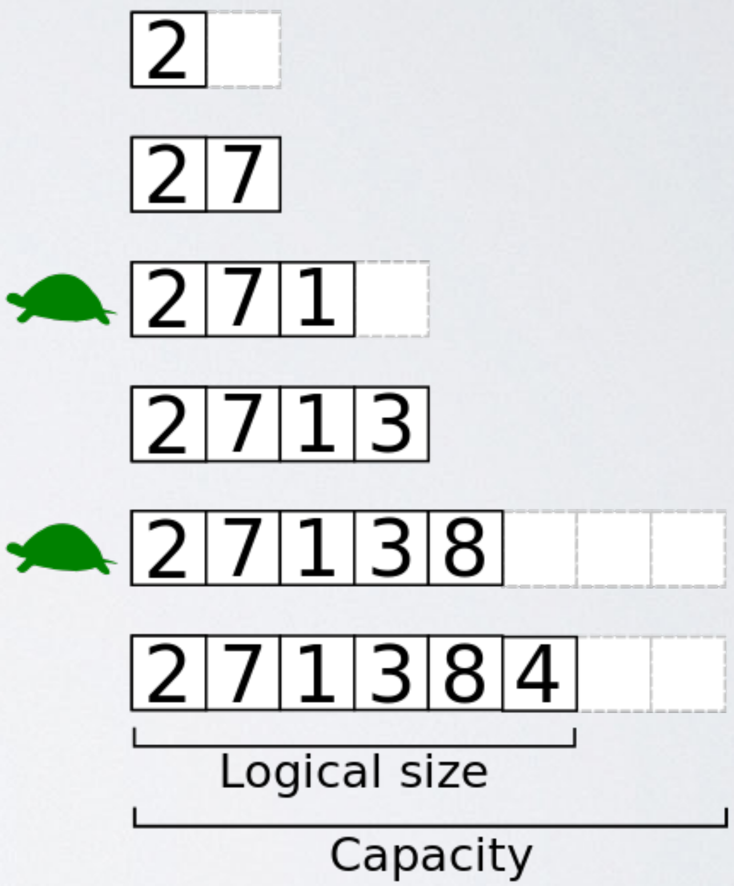
Operatie	Lista simplu inlantuita, cu first si last	Vector static
Inserare la inceput	O(1)	O(n) – trebuie shiftate elementele o pozitie catre dreapta
Inserare la sfarsit	O(1)	O(1)
Inserare in interiorul listei	O(n)	O(n) – ca si la inserare la inceput
Cautare dupa cheie	O(n)	O(n)
Accesare element dupa pozitie	O(n)	O(1)
Stergere de la inceput	O(1)	O(n)
Stergere de la sfarsit	O(n)	O(1)
Stergere din interior	O(n)	O(n)

VECTOR STATIC *VS* LISTA SIMPLU INLANTUITA

	VECTOR (SIR)	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

VECTOR DINAMIC (DYNAMIC ARRAY)

- A.k.a dynamic array, growable array, resizable array, dynamic table, mutable array, or array list
- Capacitate variabila:
  - Cand `size` = `capacity`, se aloca un vector nou, de capacitate `a*capacity`, se copiaza elementele din locatia veche in cea noua



https://en.wikipedia.org/wiki/Dynamic_array

DYNAMIC ARRAY- INSERT

```
TABLE-INSERT(T, x)
  if T.capacity == 0
     allocate T.table with 1 slot
     T.capacity = 1
  if T.capacity == T.size
     allocate new_table with a*T.capacity slots
     insert all from T.table to new_table
     free T.table
     T.table = new_table
     T.capacity = a*T.capacity
  insert x into T.table
  T.size = T.size + 1
```

**Sursa: Th. Cormen, "Introduction to Algorithms", sect. 17.4*

DYNAMIC ARRAY – ANALIZA AMORTIZATA INSERT

- Presupunand ca inseram n elemente in total, a=2, capacity = 1
- Costul unei inserari:

$$c_i = \begin{cases} i, if\ i-1 = \ 2^p \\ 1, otherwise \end{cases}$$

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor lg n \rfloor} 2^j < n + 2n$$

BIBLIOGRAFIE

- S. Skiena - ''The Algorithm Design Manual'': cap1, 2, 3.1
- Th. Cormen et al – "Introduction to Algorithms", 3rd ed: sect. 10.1, 10.2
- [suplimentar] Kleinberg, Tardos – "Algorithm Design": Cap. 2

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA