Structuri de Date

Anul universitar 2019-2020

Prof. Adina Magda Florea

Curs Nr. 8

- ❑ Cozi de prioritate (heap)
- ❑ Structuri Heap
- ❑ Heapsort
- ❑ Treap

1. ADT Cozi de prioritate

- Pornesc de la structura de tip coada (ADT coadă)
- Sunt **o generalizare a tipului de date abstracte (ADT) coadă** în care **fiecare element are asociată o prioritate** iar elementele sunt extrase din coadă în ordinea acestei priorităţi
- Primul element extras din coada este elementul cu cea mai mare prioritate – ***heap max***, sau elementul cu cea mai mică prioritate – ***heap min***
- Au foarte multe aplicaţii

ADT Cozi de prioritate

- ADT implementat deja in unele limbaje
  - C++: priority_queue
  - Java: PriorityQueue
  - Python: heapq
- Printre algoritmii care folosesc PQ:
  - Algoritmul lui Dijstra
  - Algoritmul lui Prim
  - Algoritmul lui Huffman
  - Heapsort …..

2. Operaţii de bază

ADT PQ (Priority Queue)

- **PQInit** - iniţializează coada
- **PQEmpty** – verifică coadă vidă
- **Insert** – inserează un element în coada de priorităţi
- **ExtractMax** – elimină elementul cu prioritate maximă

Operaţii suplimentare

ADT PQ (Priority Queue)

- **GetMax** – întoarce elementul cu prioritatea maximă (fără a-l elimina din coadă)
- **ExtractEl** – elimină din coadă un anumit element (nu neapărat prioritate maxima)
- **ChangePri** – schimbă prioritatea unui element din coada de priorităţi
- **BuildPQ** – construieşte o coadă de priorităţi pornind de la o secvenţă de elemente
- **JoinPQs** – combină 2 PQ într-o singură PQ

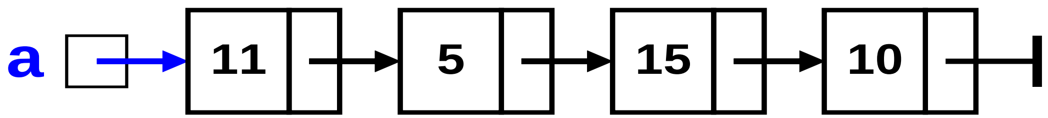
Exemplu

- O **literă** înseamnă inserarea acelei litere in coada de prioritati (Insert) şi ***** înseamnă eliminarea elementului cu prioritate maximă (ExtractMax)
- Care este secvenţa de elemente obţinută prin inserare si eliminare din coada de prioritati ca rezultat al următorului şir de comenzi, considerand pozitia literei in alfabet ca prioritatea elementului?

P R I O * R * * I * T * Y * * *

3. Implementări elementare

- Putem utiliza diverse structuri pentru a implementa o coadă de priorităţi
- **Listă nesortată**



- **Insert e** – adaugă elementul **e** la sfârşit
- **ExtractMax** – parcurge lista pt a găsi maximul

Implementări elementare

- **Vector nesortat**

11	5	15	10				

- **Insert e** – adaugă elementul **e** la sfârşit

- **ExtractMax** – parcurge vectorul pt a găsi maximul

Implementări elementare

- **Vector sortat**

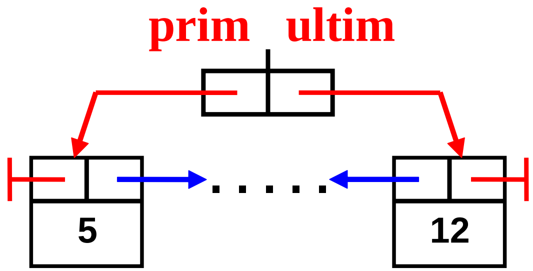
5	7	10	12				

- **ExtractMax** – extrage ultimul element

- **Insert e** – găseşte poziţia pentru **e** (folosind căutare binară de exemplu), deplasează apoi elementele la dreapta cu 1, inserează

Implementări elementare

- **Listă sortată**



- **ExtractMax** – extrage ultimul element

- **Insert e** – găseşte poziţia pentru **e** şi inserează

Implementări elementare

	Insert	ExtractMax
Vector sau listă nesortată	O(1)	O(n)
Vector sau listă sortată	O(n)	O(1)

4. Structura de date Heap

- Permite implementarea eficientă a operațiilor cu cozi de prioritate

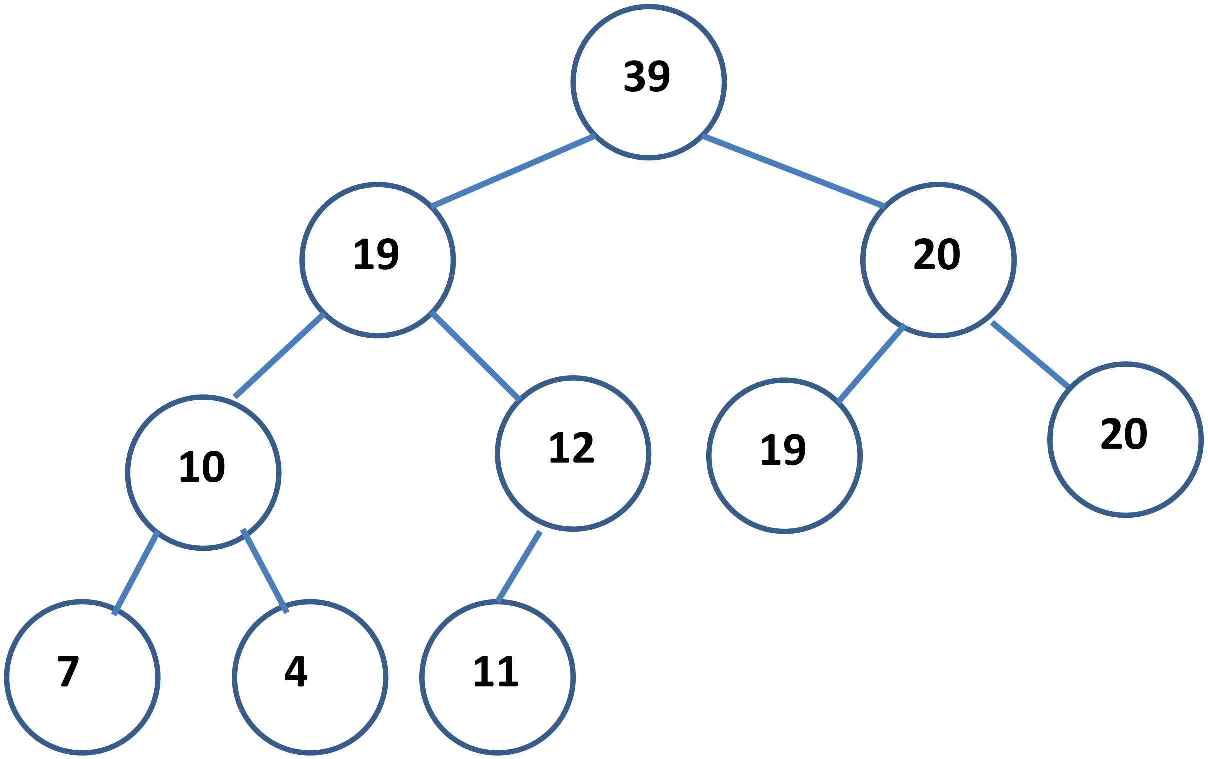
- Un **heap-max binar** este

  un **arbore binar** cu proprietatea:

  pentru orice nod, **cheia nodului este mai mare decât cheile din nodurile copii**, dacă există copii

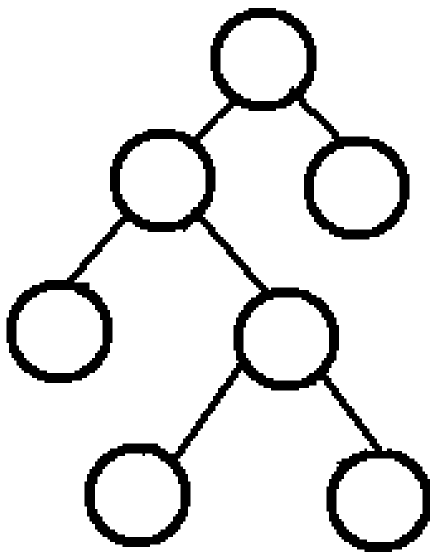
- *Proprietatea de ordonare a heap-ului* (heap order property)

HEAP Max

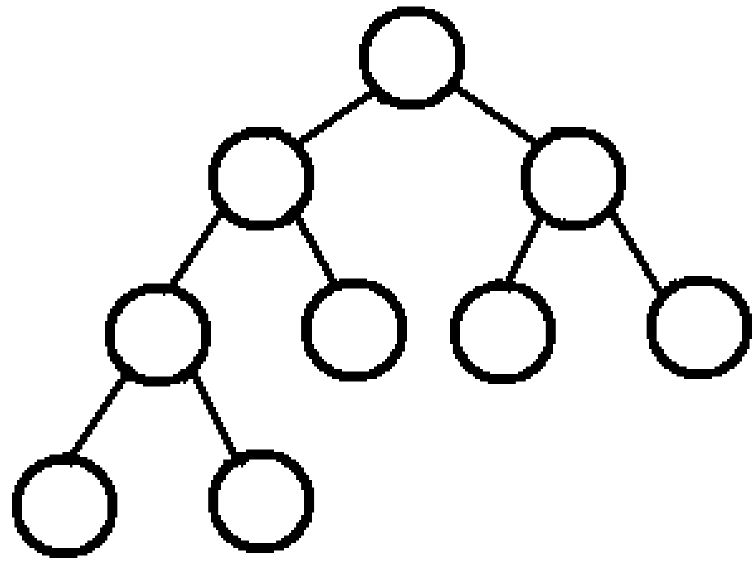


Structura de date Heap

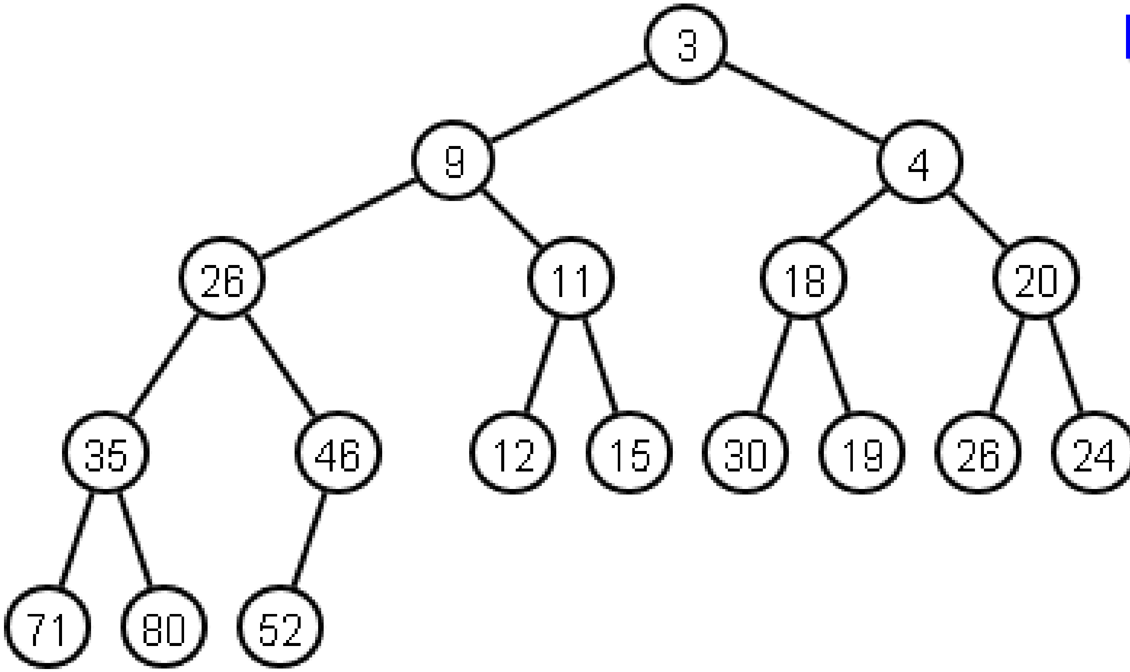
- Un heap este reprezentat de multe ori ca un arbore binar complet
- Arbore binar complet – un arbore binar care este complet umplut, cu posibila excepţie a ultimului nivel care este umplut de la stânga la dreapta
- *Proprietatea de structură a heap-ului* (heap structure property)



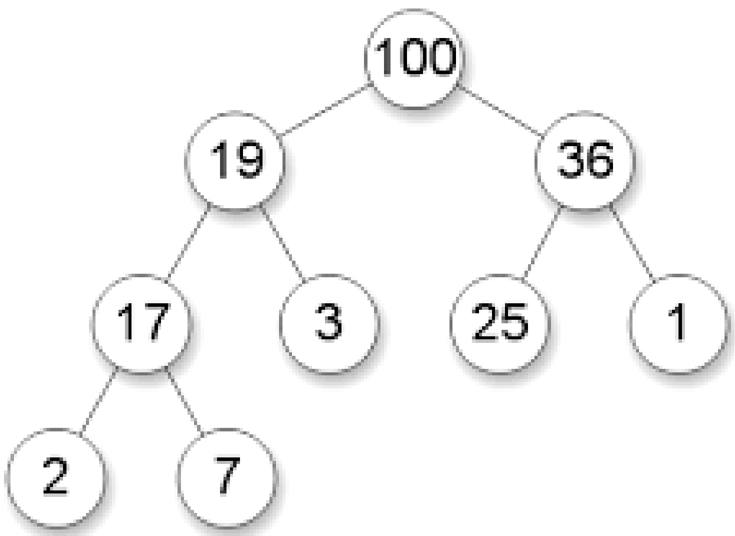
Arbore binar plin



Arbore binar complet



Heap-min binar



Heap-max binar

Proprietate Heap

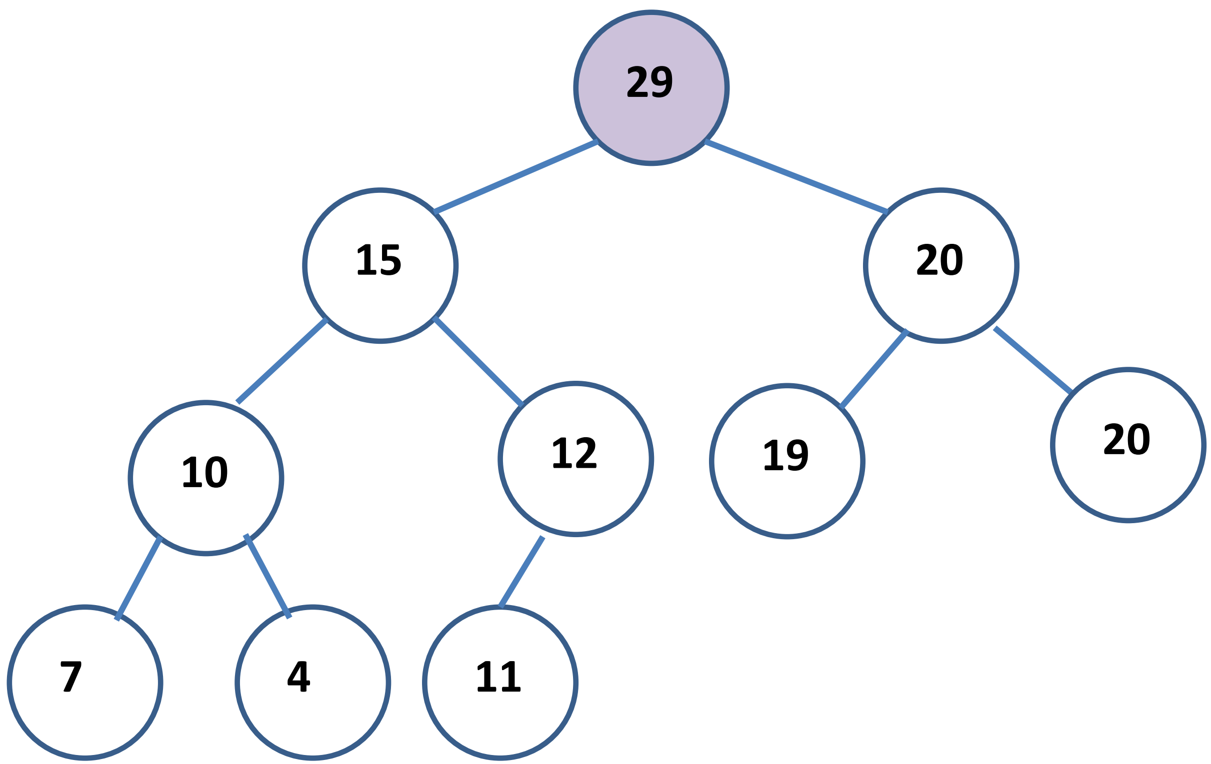
- Un arbore binar complet cu **n** noduri are **înălțimea cel mult O(log n)**

- Oferă astfel o aranjare a elementelor din heap care permite o **căutare eficientă**
- DAR operațiile cu heap trebuie să păstreze cele 2 proprietăți de heap

Operații cu Heap

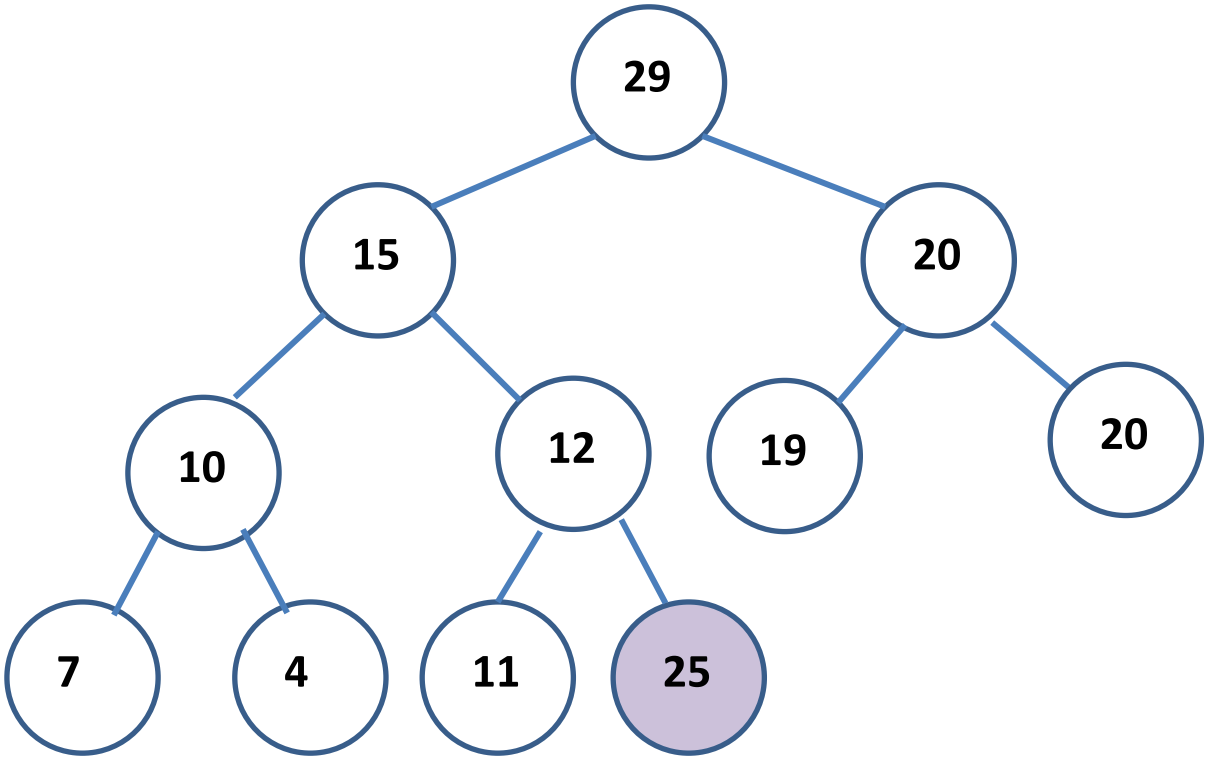
HEAP max

GetMax ⇒ **29**



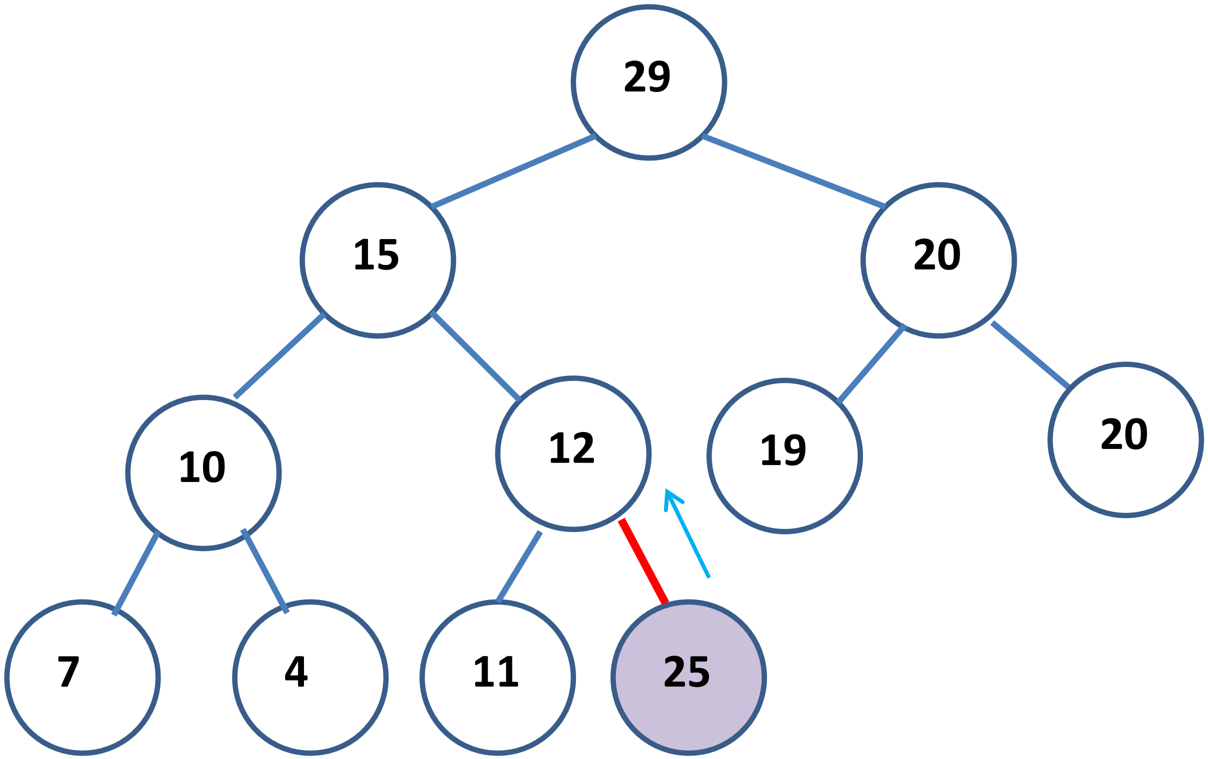
HEAP

Insert 25



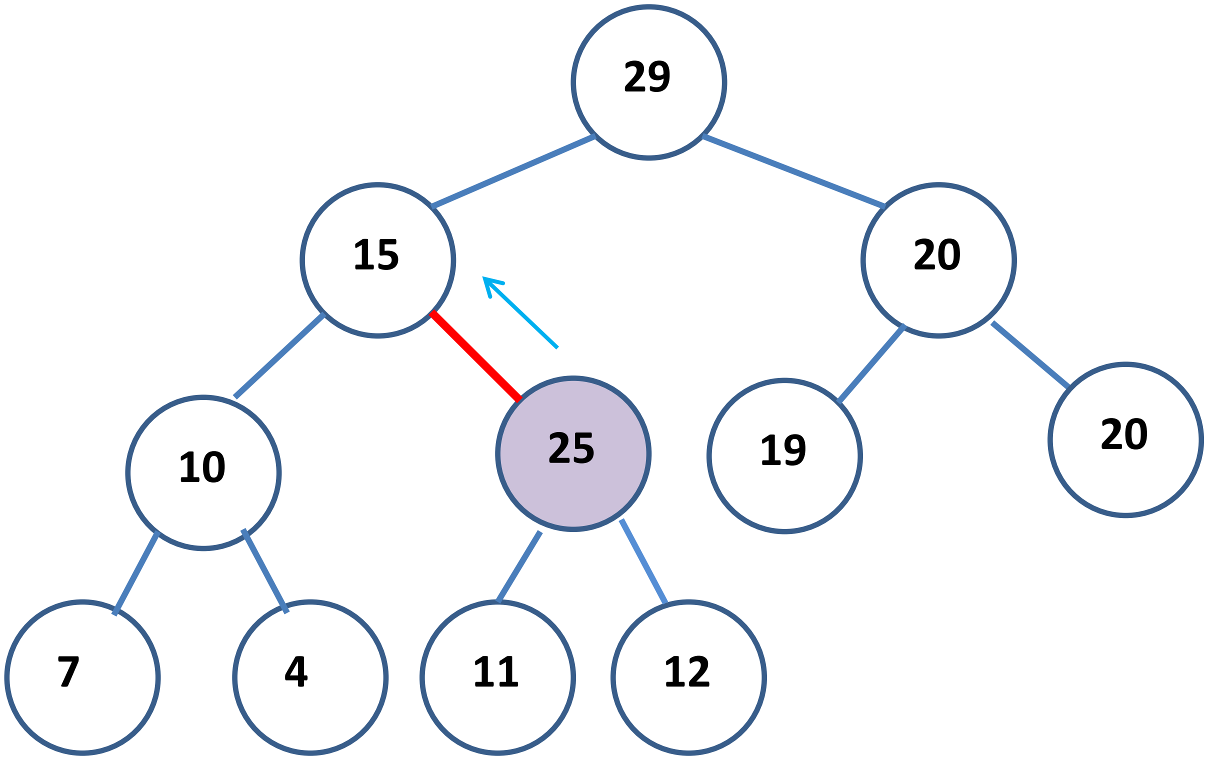
HEAP

SiftUp



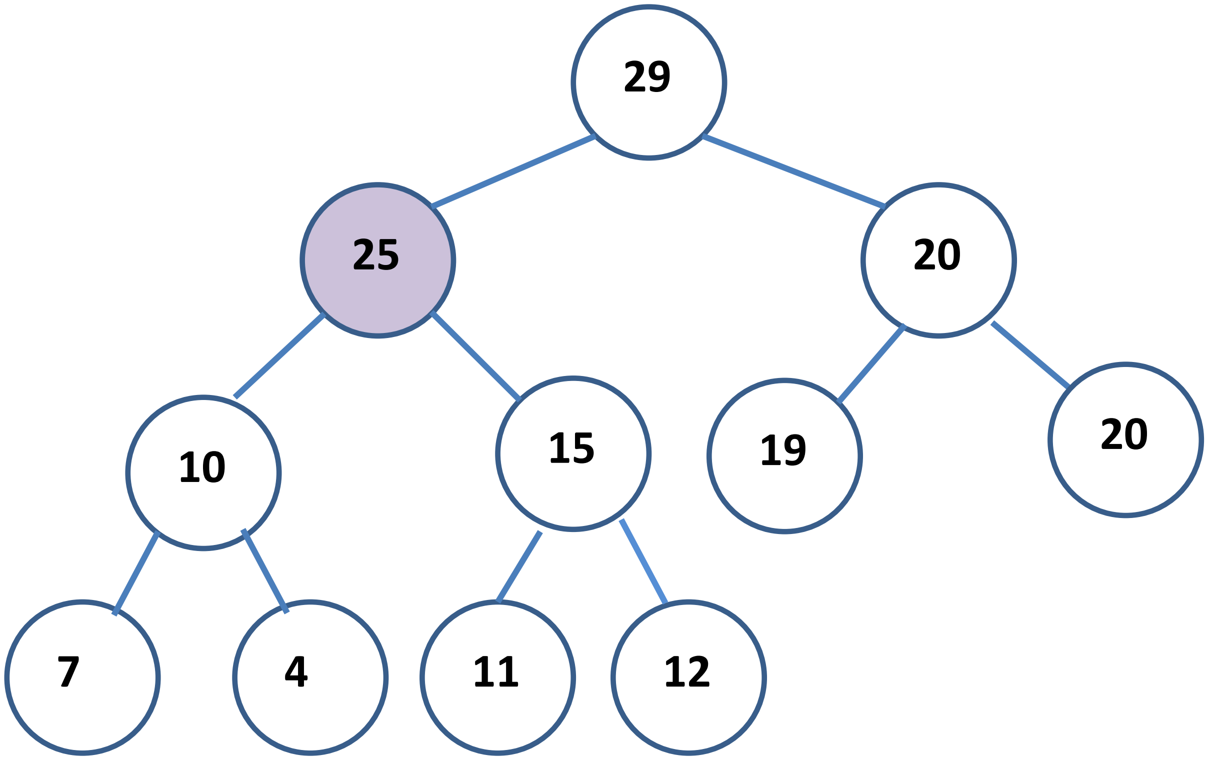
HEAP

SiftUp



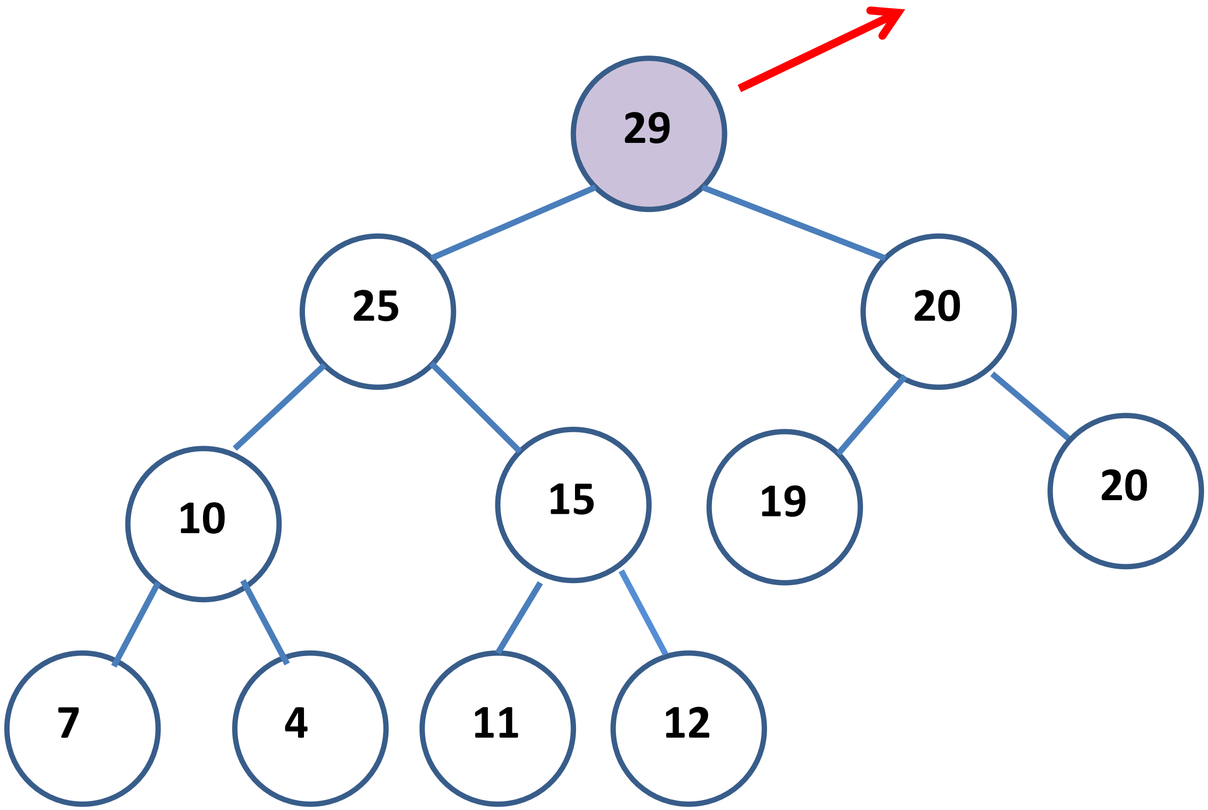
HEAP

Proprietatea de ordonare Heap refăcută



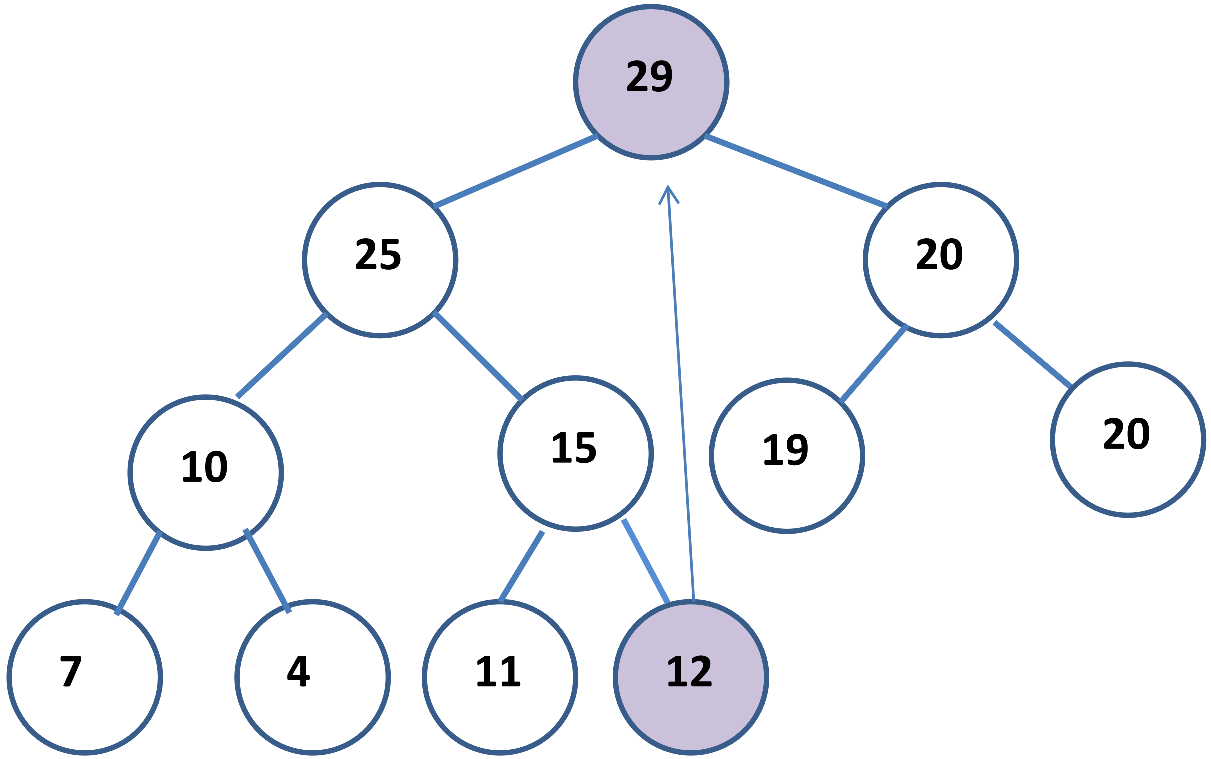
HEAP

RemoveMax



HEAP

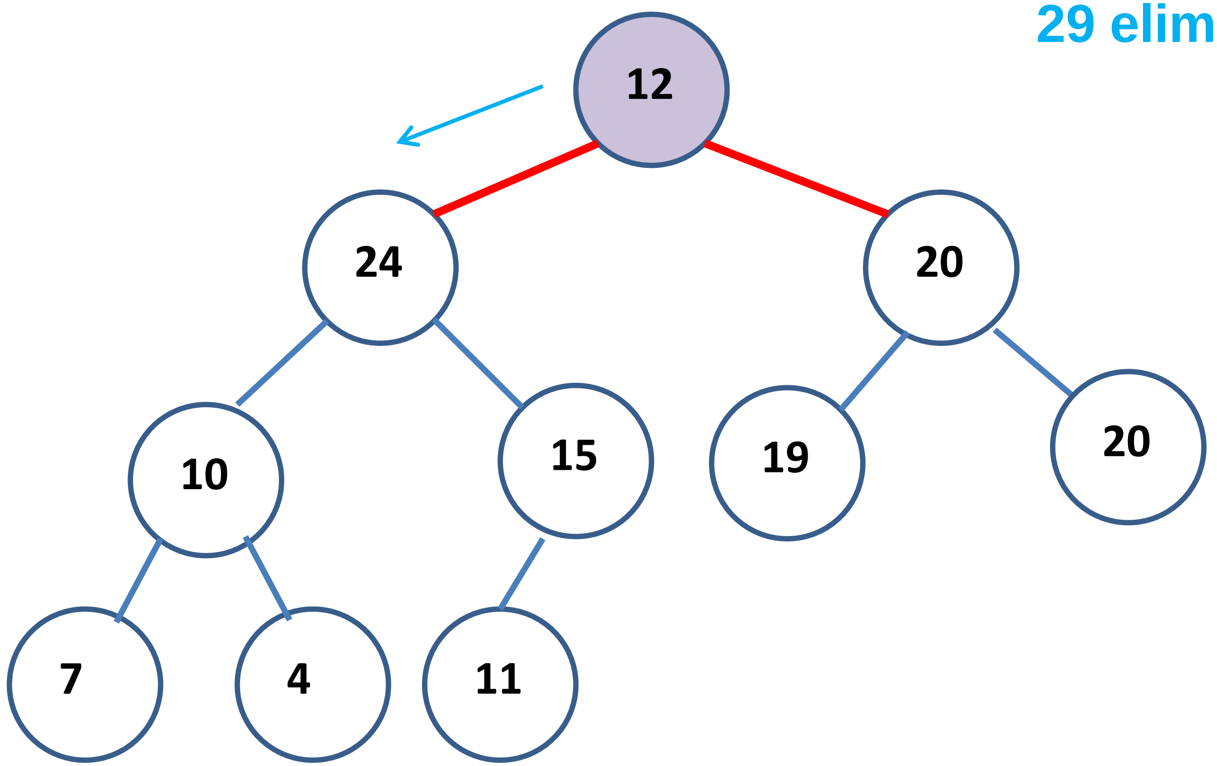
RemoveMax



HEAP

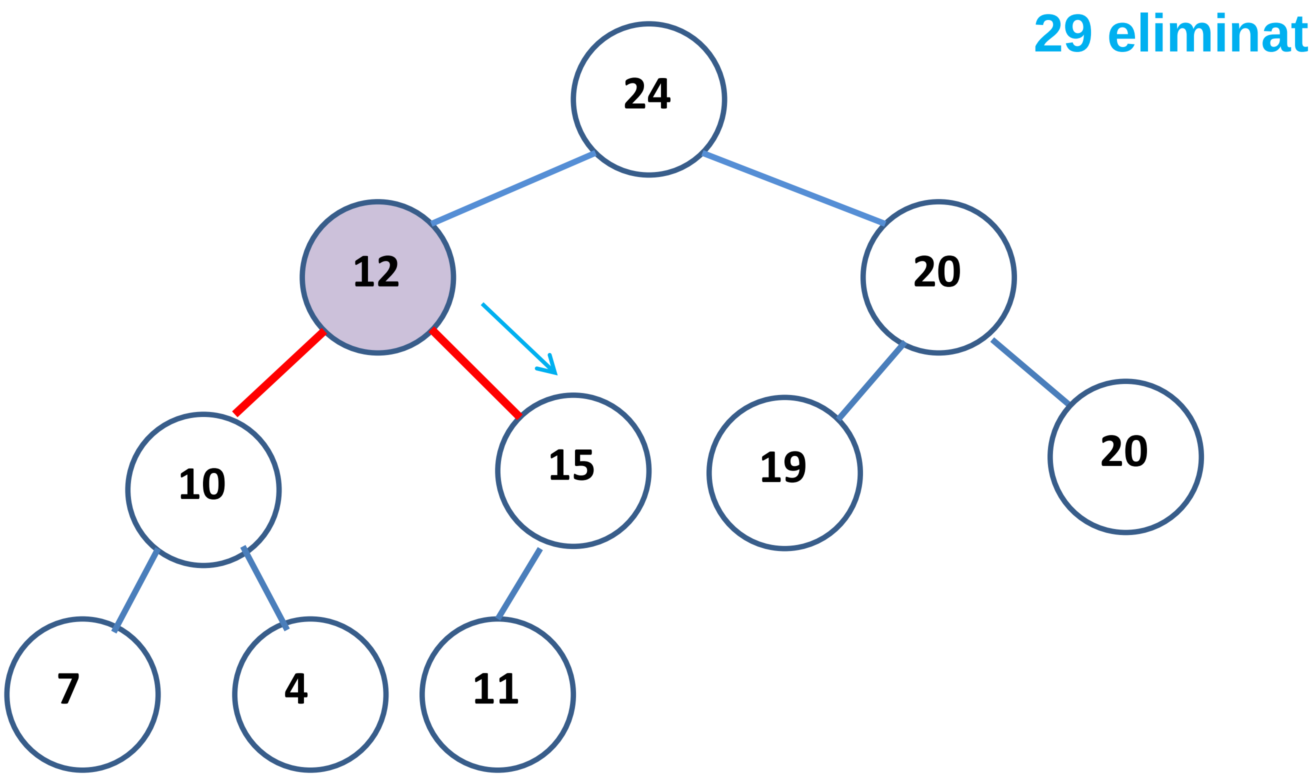
SiftDown

29 eliminat



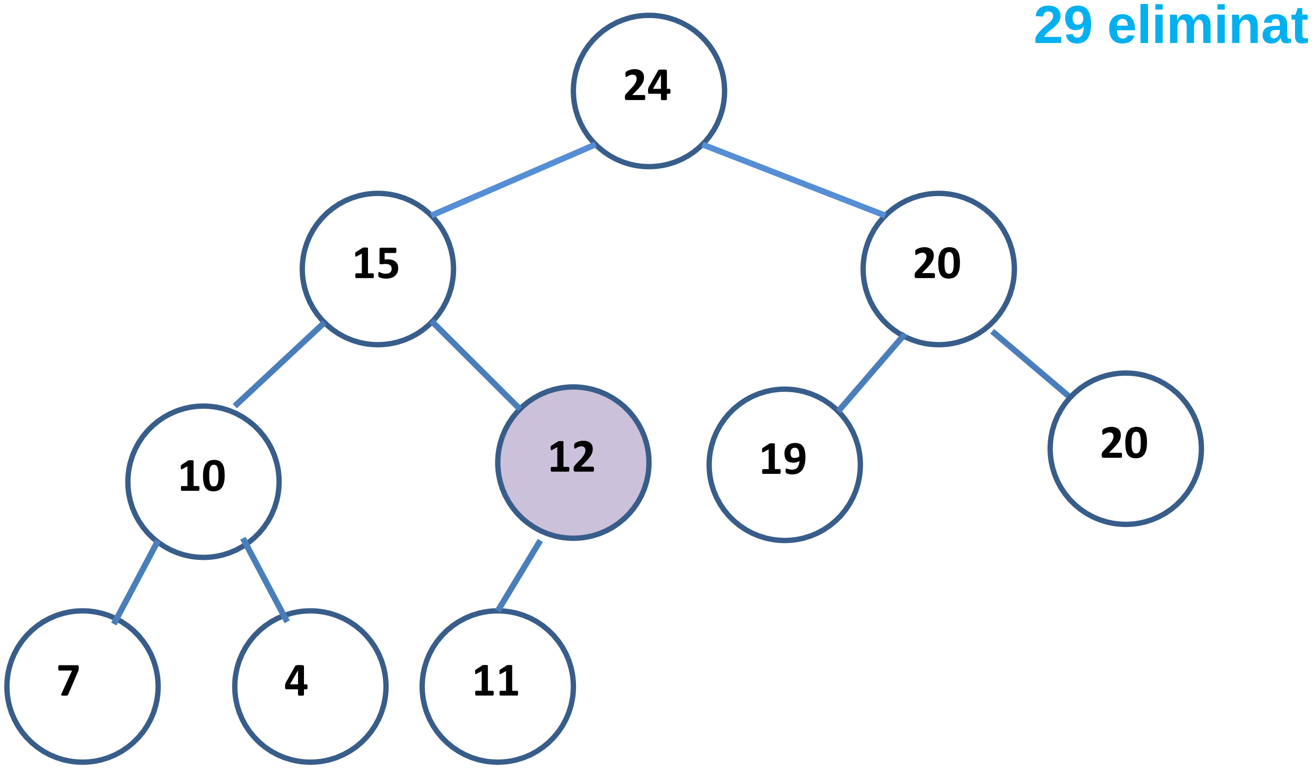
HEAP

SiftDown



HEAP

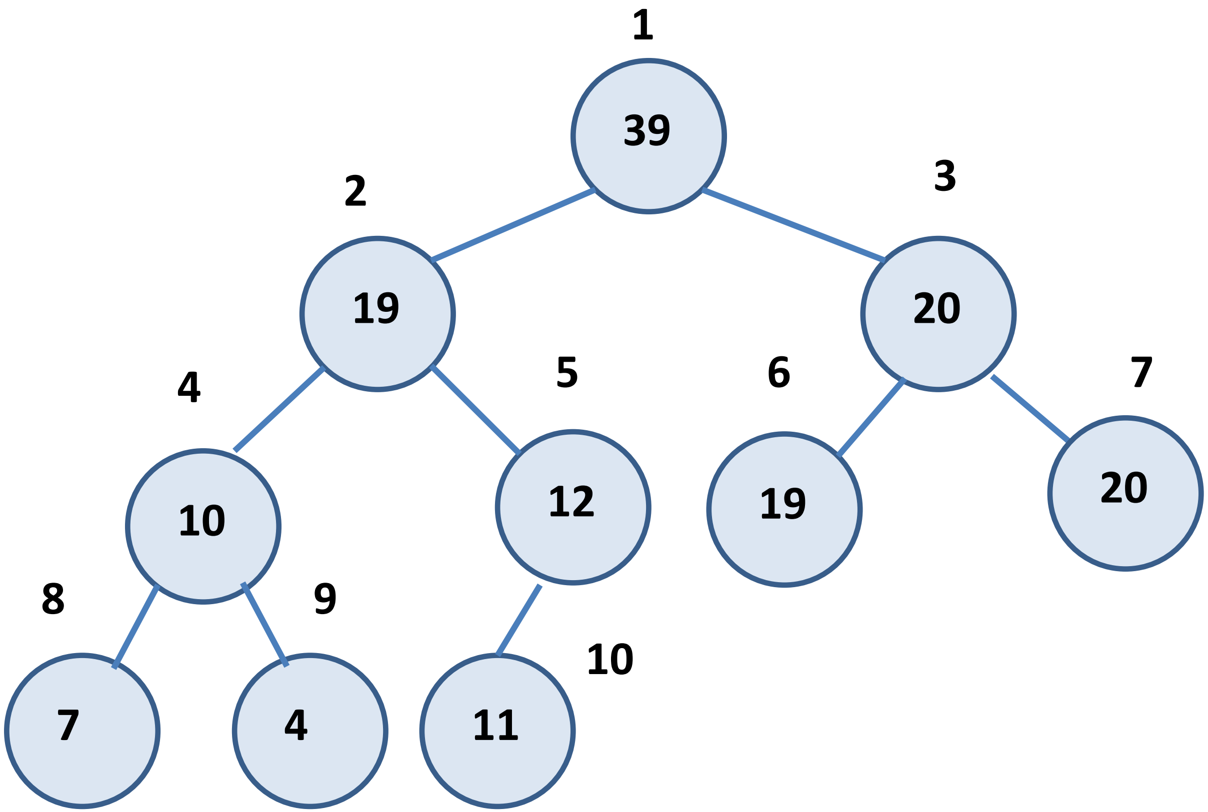
Proprietatea Heap refăcută



5. Reprezentare Heap

- Un alt avantaj al arborilor binari compleți: pot fi **reprezentați convenabil ca un vector**, deci nu mai avem nevoie de legăturile explicite dintr-un arbore binar.

Reprezentare heap ca vector



parent(i) = [i/2]

leftchild(i) = 2*i

rightchild(i) = 2*i + 1

1	2	3	4	5	6	7	8	9	10			
39	19	20	10	12	19	20	7	4	11			

Reprezentare heap ca **vector**

```
typedef char *Item;
typedef struct
  {   Item content;
      int prior;
  } ItemType;

typedef struct heap
    {   long int MaxHeapSize;
        long int Size;
        ItemType  *elem;
    } PriQueue, *APriQueue;
```

Reprezentare heap ca **arbore binar**

```
typedef char *Item;
typedef struct
  {   Item content;
      int prior;
  } ItemType;

typedef struct node *APriQueue;
typedef struct node {
          ItemType  elem;
         APriQueue  lt, rt;} PriQueue;
```

Reprezentare heap cu vector - simplificata

```
typedef int ItemType;
/* elementul este insasi prioritatea */

static long int MaxHeapSize;
typedef struct heap
    {   long int Size;
        ItemType  *elem;
    } PriQueue, *APriQueue;
PriQueue h;
```

6. Implementare - Pseudocod

Parent(i) intoarce [i/2]

LeftChild(i) intoarce 2*i

RightChild(i) intoarce 2*i+1

Algoritm Getmax(h)

intoarce elementul maxim

intoarce h.elem[1]

sfarsit

Implementare - Pseudocod

```
Algoritm Insert(h, element)
/* modifica h prin efect lateral sau intoarce eroare */
    daca h.Size = MaxHeapSize
    atunci intoarce eroare  /* sau realoca */
    h.Size = h.Size+1
    h.elem[h.Size] = element
    SiftUp(h, h.Size)
sfarsit
```

Implementare - Pseudocod

```
Algoritm SiftUp(h, i)
                        /* modifica h prin efect lateral */

   cat timp (i>1) si ( h.elem[i] > h.elem[Parent(i)] )
   repeta
        Interschimba (h.elem[Parent(i)], h.elem[i])
        i = Parent(i)
sfarsit
```

Implementare - Pseudocod

```
Algoritm ExtractMax(h)
/* modifica h prin efect lateral si intoarce elem
   maxim*/
   rezultat = h.elem[1]
   h.elem[1] = h.elem[h.Size]
   h.Size = h.Size – 1
   SiftDown(h,1)
   intoarece rezultat
sfarsit
```

```
Algoritm SiftDown(h, i)
                            /* modifica h prin efect lateral */
   maxIndex = i
   left = LeftChild(i)          /*  left = 2*i  */
   daca left ≤ h.Size si h.elem[maxIndex] < h.elem[left]
   atunci maxIndex = left
   right = RightChild(i)              /*  right = 2*i + 1 */
   daca right ≤ h.Size si h.elem[maxIndex] < h.elem[right]
   atunci
        daca h.elem[left] < h.elem[right]
        atunci maxIndex = right
   if  i != maxIndex then
         Interschimba (h.elem[maxIndex], h.elem[i])
        SiftDown(h, maxIndex)
sfarsit
```

Algoritm SiftDownNR(h, i) – varianta nerecursiva

/* modifica h prin efect lateral */

cat timp 2*i ≤ h.Size **repeta**

j = 2*i /* j=LeftChild(i) */

daca j < h.Size **si** h.elem[j] < h.elem[j+1]

atunci j = j+1 /* j=RightChild */

daca h.elem[i] > h.elem[j] **atunci break**

Interschimba (h.elem[i], h.elem[j])

i = j

sfarsit

Algoritm ChangePri(h, i, noua_p)

/* modifica h prin efect lateral, schimba prioritatea elementului de pe pozitia i*/

vechea_p = h.elem[i]

h.elem[i] = noua_p

daca noua_p > vechea_p

atunci SiftUp(h,i)

altfel SiftDown(h, i)

sfarsit

Algoritm ExtractEl(h, i)

/* modifica h prin efect lateral, elimina si intoarce elementul de pe pozitia i*/

rezultat = h.elem[i]

h.elem[i] = ConstFMARE

SiftUp(h,i)

j = ExtractMax(h)

intoarece rezultat

sfarsit

Implementarea PQ prin Heap

- Implementarea rezultată este eficientă
  - **O(log n)** pentru Insert si ExtractMax
- Este eficientă si din punct de vedere al spațiului folosit

- **O coadă de prioritate poate fi folosită pentru a sorta un sir de elemente**

7. Heapsort

- Sortare utilizând o coadă de prioritate
- **Cel mai simplu: pentru a sorta a[1]…a[n]**

- Creează a coadă de priorități vidă **h**
- **pentru** i=1 la n **repeta**
  Insert (h,a[i])
- **pentru** i = n la 1 **repeta**
  a[i] = ExtractMax(h)

O(n log n)

Heapsort

- Soluția anterioară folosește un spațiu dublu pentru a memora coada de priorități
- **Soluție mai bună: transformarea vectorului de sortat într-un heap**

Heapsort

- Transformăm vectorul într-un heap prin permutarea elementelor lui; cum?
- Reparăm proprietatea de heap de jos în sus
- Inițial proprietatea de heap este satisfăcută în toate frunzele (pe ultimul nivel)
- Reparam toți subarobii pe nivelul imediat superior, și așa mai departe
- Când ajungem la rădăcină, proprietatea de heap este satisfăcută pentru întregul arbore

http://www.cs.usfca.edu/~galles/visualization/HeapSort.html

Heapsort

Algoritm BuildHeap(a[1..n])
    size = n
    pentru i = [n/2] la 1 **repeta**
        SiftDown(a,i)
sfarsit

O(n log n)

Algoritm Heapsort(a[1..n])
    BuildHeap(a)
    size = n
    cat timp size ≥ 2 **repeta**
        Interschimba(a[1], a[size])
        size = size – 1
        SiftDown(a,1)
sfarsit

8. Treaps

- Dacă construim un arbore binar de căutare (BST) cu valori aleatoare ale cheilor – Random Binary Search Tree - arborele rezultat va fi echilibrat cu o mare probabilitate
- Adâncimea medie a unui nod dat este aproximativ *2*ln(n)*
- Deci adancimea arborelui este proportionala cu logaritm din numarul de noduri cu o mare probabilitate
- Dacă structurăm un BST ca și cum ar fi un arbore aleator vom obține o structură destul de bună

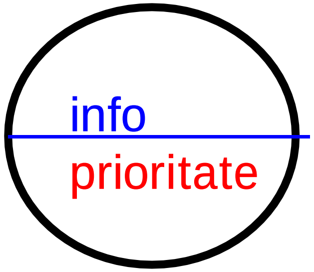
Treaps

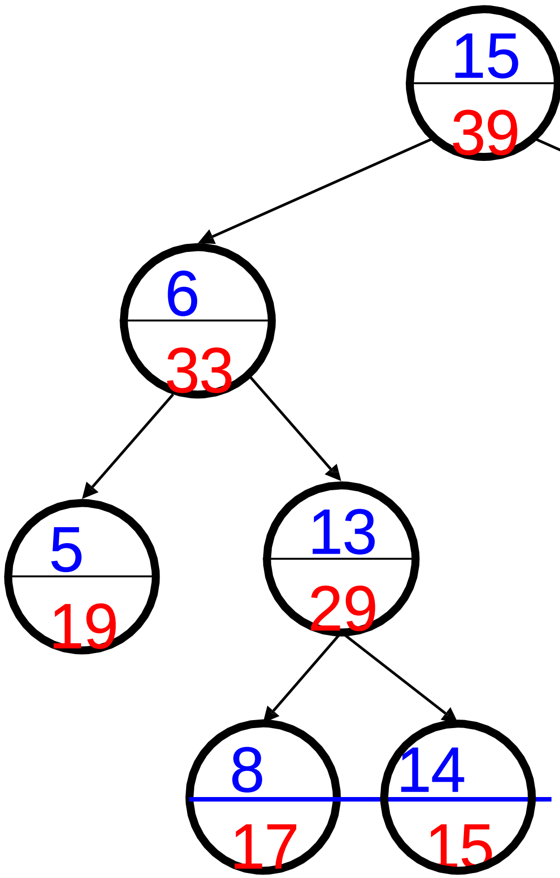
- Tree + Heap → Treap
- Structură de date care combină BST cu Heap binar
- Fiecare nod conține (*info*, *prioritate*)
  - *info* – informația din arbore
  - *prioritate* – un număr aleator
- Treap:
  - Are proprietatea de arbore binar de cautare (*info*)
  - Are proprietatea de ordonare din Heap binar (*prioritate*)

Treaps

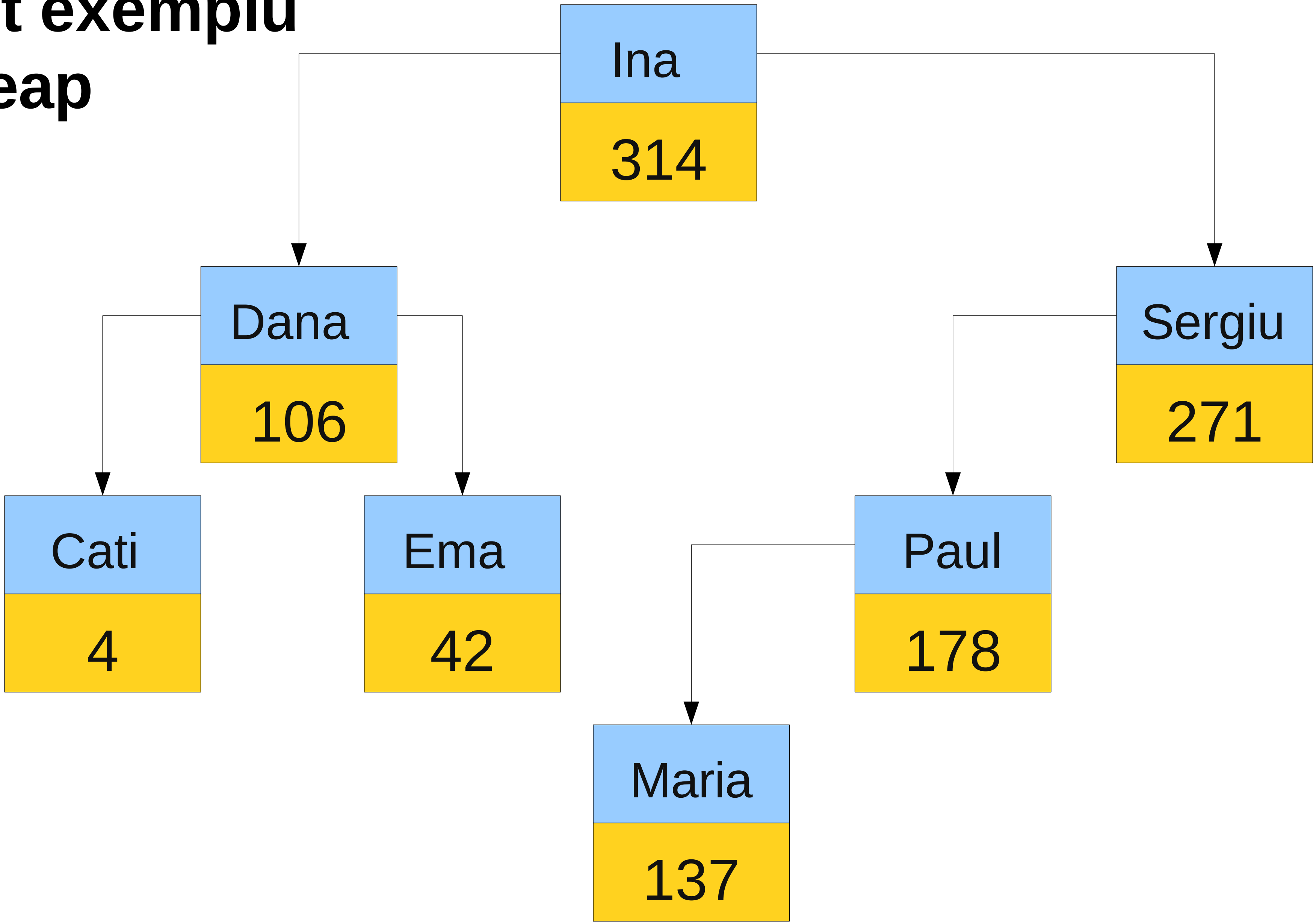
```
typedef struct node *Treap;
typedef struct node {
                int    key;
                int     pri;
                Treap  lt, rt;} TreapNode;
```

Exemplu treap





Alt exemplu treap



Operații

- Cautare – analog cu cea de la arbori binari de cautare
- Inserare
- Eliminare

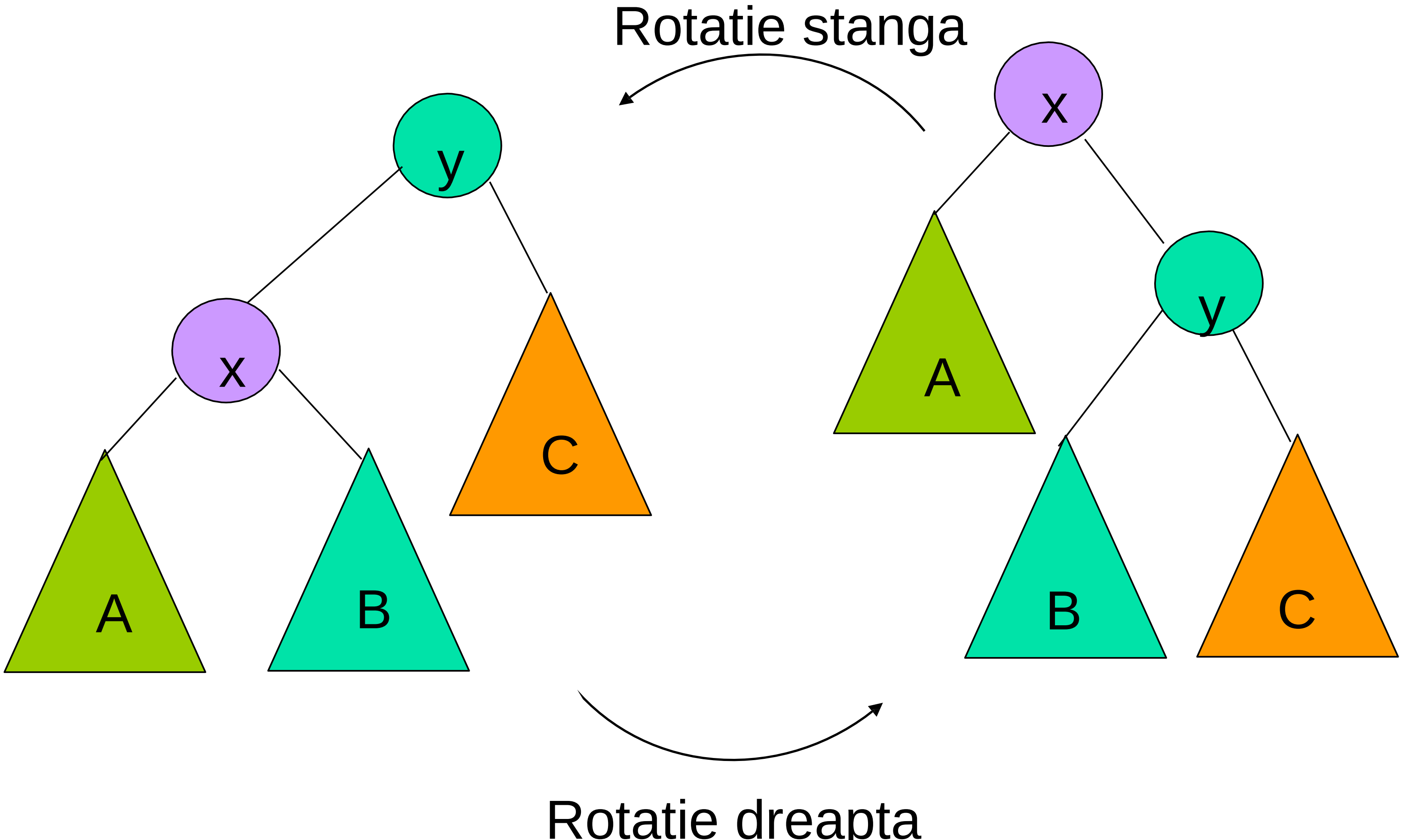
(vezi implementare pe tabla)

- Alte operatii
  - Union
  - Intersection
  - Set difference
  - Operații ajutătoare: Split și Join (Merge)

Inserare in treap

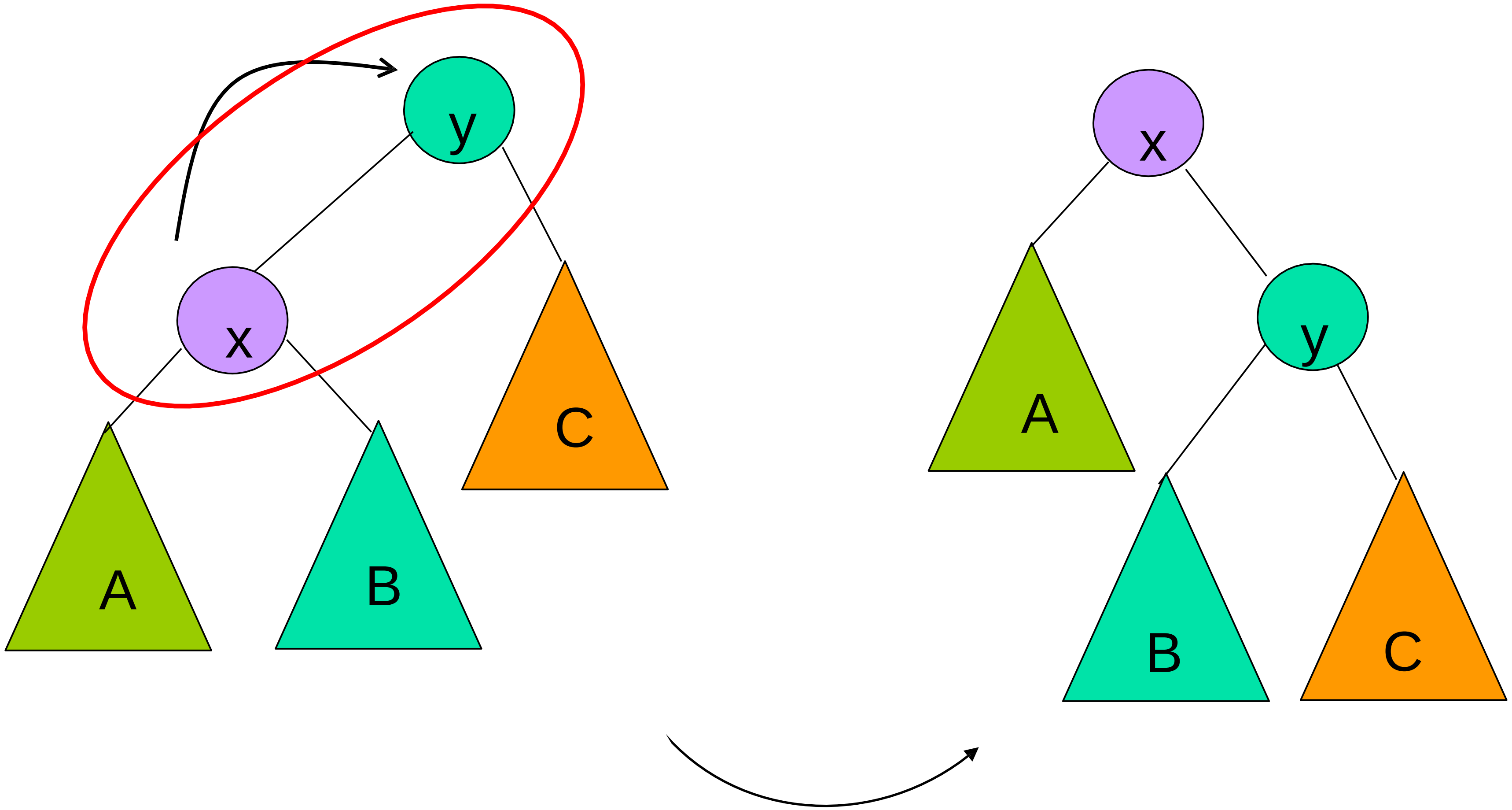
- se genereaza o prioritate aleatoare pentru nodul inserat
- **se insereaza** informatia in treap folosind regula de inserare de la **arbori binari de cautare**
- **se actualizeaza** structura arborelui pentru a se asigura conditia de ordonare din **heap** raportata la prioritatile din nodurile arborelui: **rotatii dreapta / stanga**

Rotatii



Rotatie dreapta

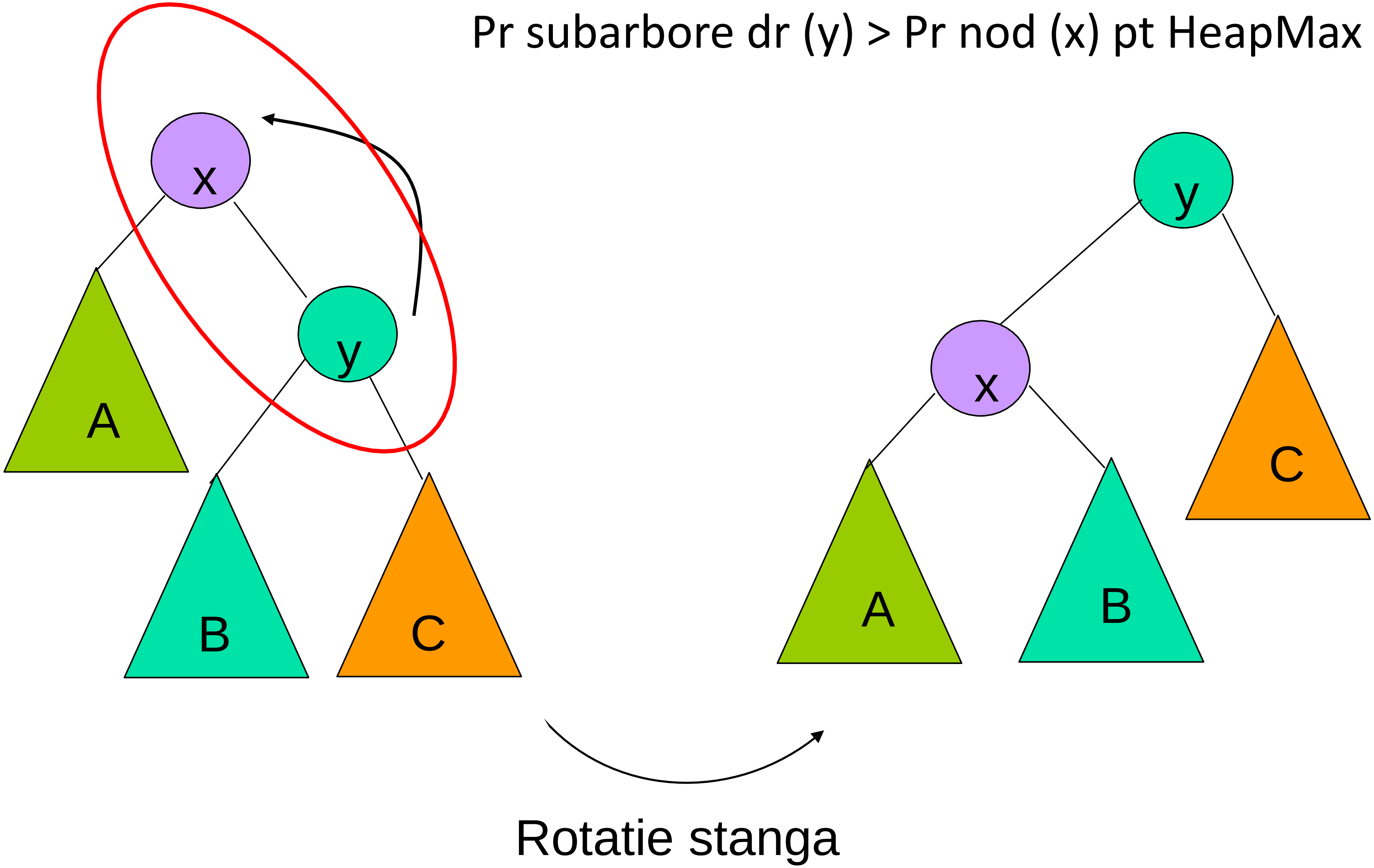
Pr subarbore st (x) > Pr nod (y) pt HeapMax

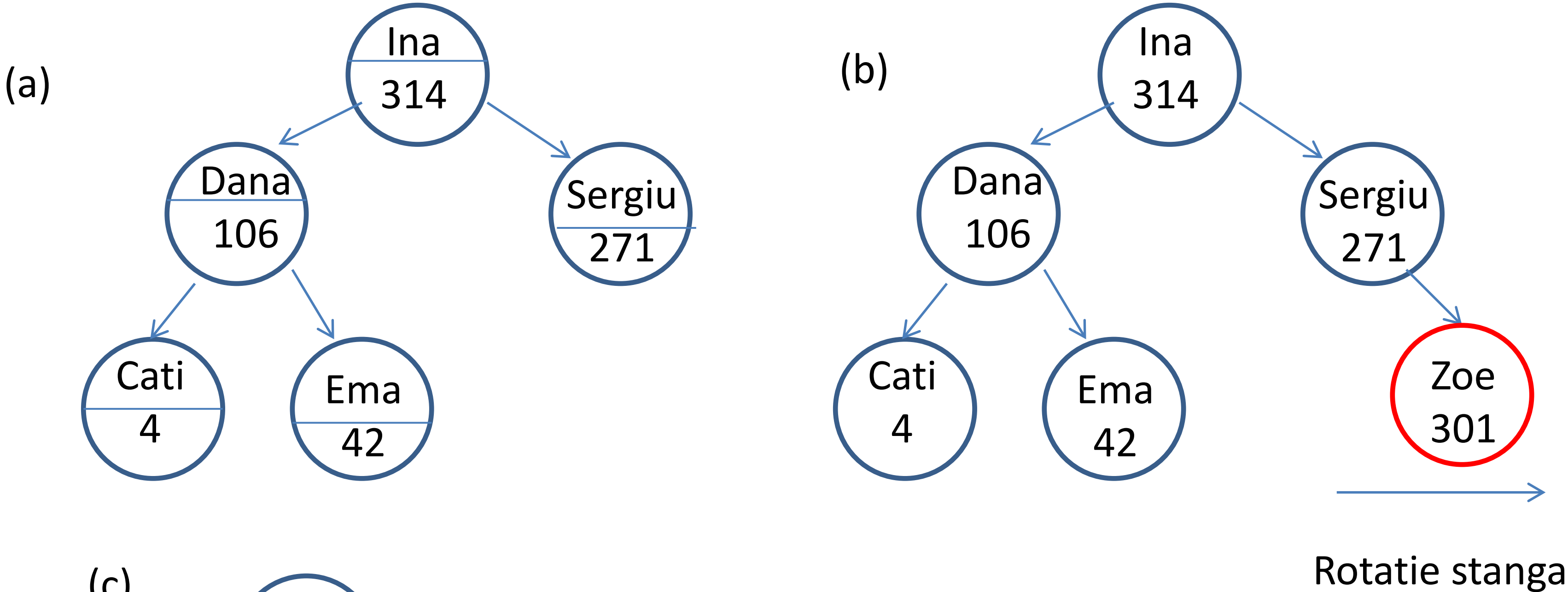


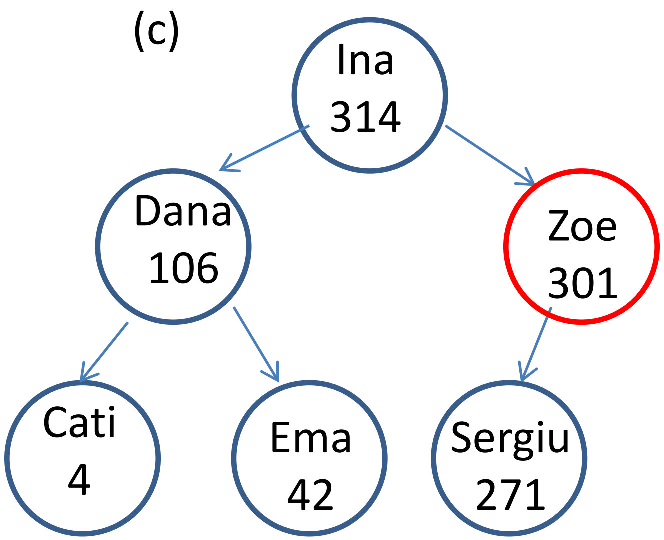
Rotatie dreapta

Rotatie stanga

Pr subarbore dr (y) > Pr nod (x) pt HeapMax







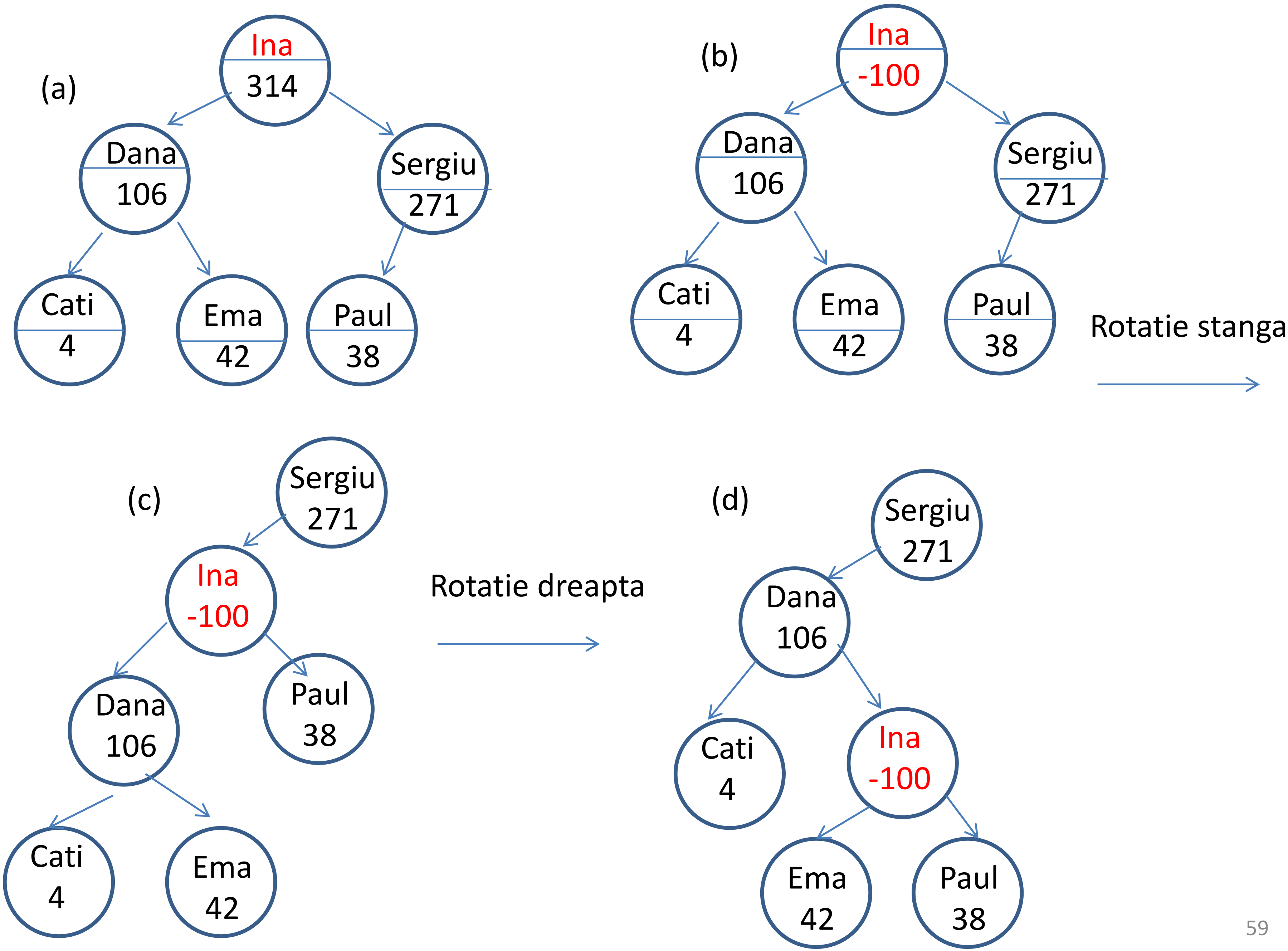
Inserare cheie Zoe, prioritate 301 generata aleator

Eliminare

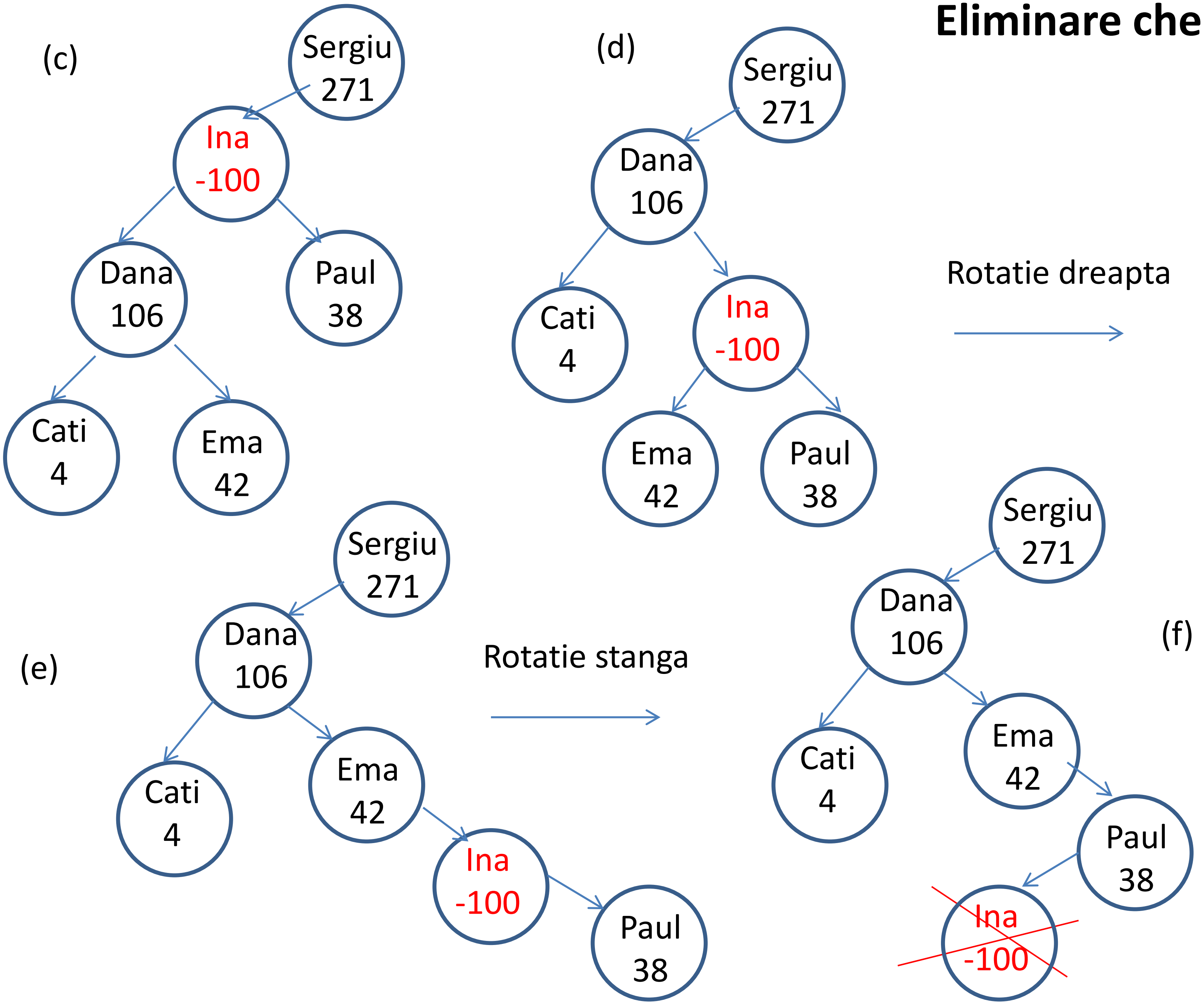
- Cauta ***x*** in arbore (analog cautare in arbore binar de cautare)
- Daca ***x*** este frunza atunci sterge nod
- Altfel inlocuieste prioritatea nodului ***x*** cu ***-inf***
- Propaga nodul cu informatia ***x*** pana ajunge o frunza

  (prin rotatii successive: stanga / dreapta)
- Sterge frunza ce contine informatia ***x*** din arbore
- Rotatia se face cu nodul avand prioritatea cea mai mare dintre cei 2 copii ai nodului ***x***

Eliminare cheie Ina



Eliminare cheie Ina



Treap – alte operatii

- Alte operatii
  - Union
  - Intersection
  - Set difference
  - Operații ajutătoare: Split, Join (Merge)

Split

- Treap *T*
- Separa *T* in doua treapuri T_L siT_R dupa cheia *k*
- split(T,k) rezultă T_L si T_R

- $\forall\, x1 \in \mathrm{T_L}\, ,\, x2 \in \mathrm{T_R}:$

$$\mathrm{key}(x1) \le k \text{ si } \mathrm{key}(x2) > k$$

Split

Caz 1: Cheia k nu se afla in treap

- Genereaza un nou nod x cu cheia *k* si prioritatea ∞.
- Insereaza *x* in *T (**x va fi radacina arborelui**)*
- Sterge radacina arborelui:

  $T_{L} =$ subarbore stang, T_R = subarbore drept

Caz 2: Daca cheia *k* se afla in treapul *T*

- Se elimina din treapul *T* nodul cu cheia *k*
- Se realizeaza operatia de split (cazul 1)
- Se insereaza nodul cu cheia *k* in treapul T_L

Join

- Unește două tripuri T_L si T_R
- **Join** se realizeaza in mod **invers** operatiei de **split** prin unirea celor 2 treapuri T_L si T_R in jurul unei chei k

  $\forall\ x1 \in T_L\ ,\ x2 \in T_R :$

  $$\text{key}(x1) < \text{k} < \text{key}(x2)$$

- join(T_L,k,T_R) – rezulta un treap
- Se creeaza un nod radacina cu cheia *k* si prioritate ∞, ce are ca subarbore stang pe T_L, iar ca subarbore drept pe T_R
- Sterge nodul cu cheia *k*

Union

- construieste un nou treap, pe baza a doua treapuri *T1* si *T2*

union(T1, T2)   /* intoarece un nou treap T */

daca   T1 == vid **atunci intoarce** T2

altfel   **daca** T2 == vid **atunci intoarce** T1

altfel **daca** prioritate(radacina(T1)) < prioritate(radacina(T2))
atunci interschimba T1 cu T2

/* prioritate(radacina(T1) > prioritate(radacina(T2) */

T1 este de forma TL1, r, TR1   /* r cheia din radacina lui T1 */

/* Toate cheile din TL1 ≤ r si toate cheile din TR1 >r */

TL2, TR2 = split(T2, r)

intoarce join(union(TL1, TL2), r, union(TR1, TR2))

Alte operatii

- Intersection
- Set difference

Pentru examen

Treaps

- Structuri eficiente, asigura un bun timp mediu (average)
- Au performanțe asemanătoare cu arborii echilibrați dar se implementează mai simplu

Aplicatii

Treaps

- Se pot implementa ușor pe structuri paralele
- Folosite in multe aplicații, de exemplu wireless networking, memory allocation, fast parallel aggregate set operations
- Se pot utiliza pentru implementarea unor structuri de date avansate cum ar fi weighted trees, interval trees