

# Structuri de Date

Anul universitar 2018-2019

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 priorități
- 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 aplicații

# ADT Cozi de prioritate

---

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

## 2. Operații de bază

---

### ADT PQ (Priority Queue)

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

# Operații 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 maximul)
- **ChangePri** – schimbă prioritatea unui element din coada de priorități
- **BuildPQ** – construiește o coadă de priorități pornind de la o secvență de elemente
- **JoinPQs** – combină 2 PQ într-o singură PQ



# Exemplu

---

- O **literă** înseamnă inserarea acelei litere în coada de prioritati (Insert) și \* înseamnă eliminarea elementului cu prioritate maximă (ExtractMax)
- Care este secvența de elemente obținută prin inserare și eliminare din coada de prioritati ca rezultat al următorului șir de comenzi, considerând poziția literei în alfabet ca prioritatea elementului?

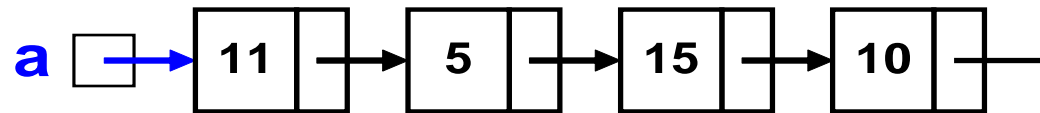
**P R I O \* R \* \* I \* T \* Y \* \* \***

### 3. Implementări elementare

---

- Putem utiliza diverse structuri pentru a implementa o coadă de priorități

- **Listă nesortată**



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





# Implementări elementare

---

- **Vector nesortat**

11	5	15	10				
----	---	----	----	--	--	--	--

- **Insert e** – adaugă elementul **e** la sfârșit
- **ExtractMax** – parcurge vectorul pt a găsi maximul



# Implementări elementare

---

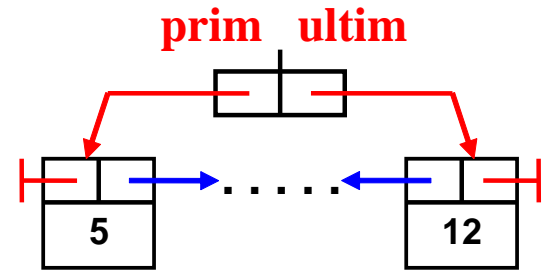
- **Vector sortat**

5	7	10	12				
---	---	----	----	--	--	--	--

- **ExtractMax** – extrage ultimul element
- **Insert e** – găsește poziția 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** – găsește poziția pentru **e** și inserează

# Implementări elementare

---

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

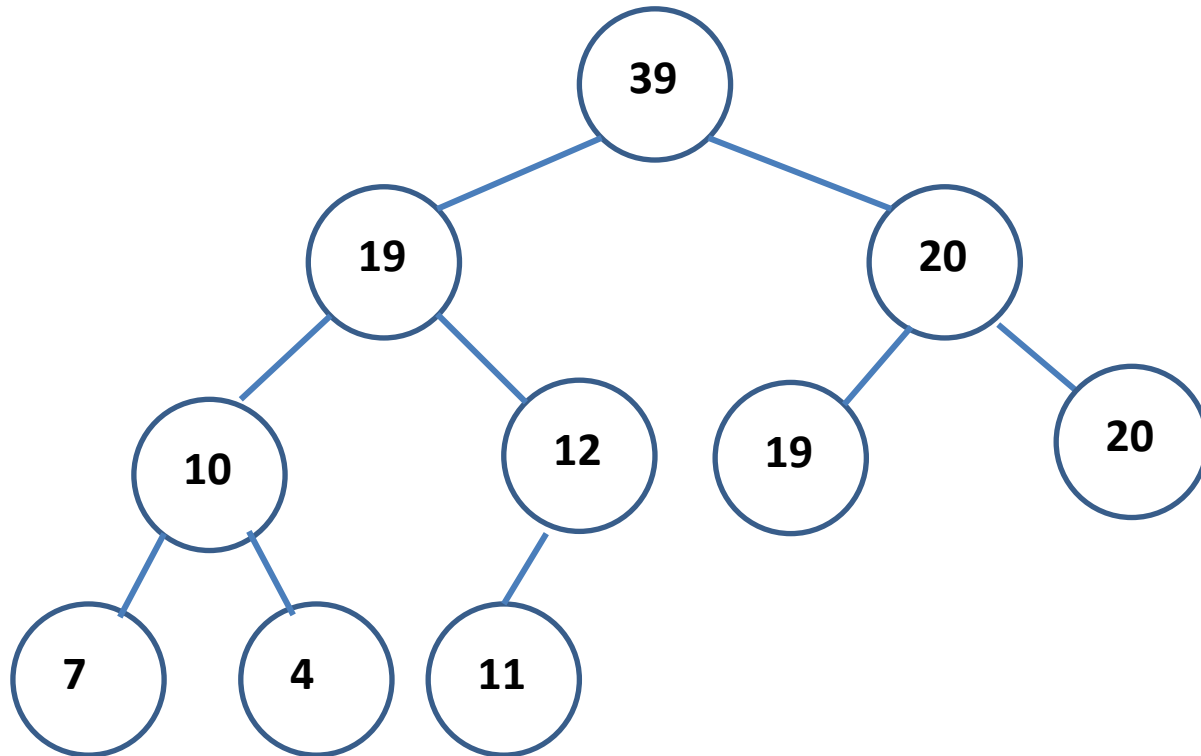
## 4. Structura de date Heap

---

- 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

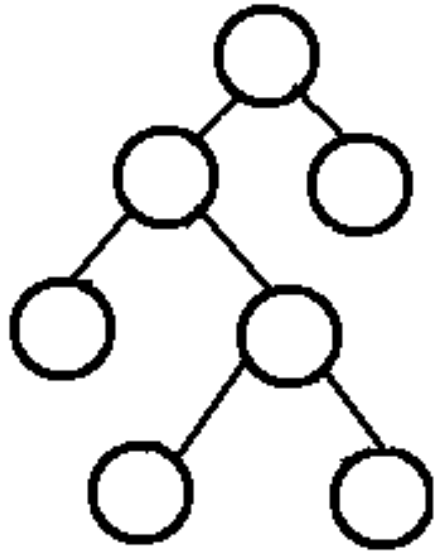




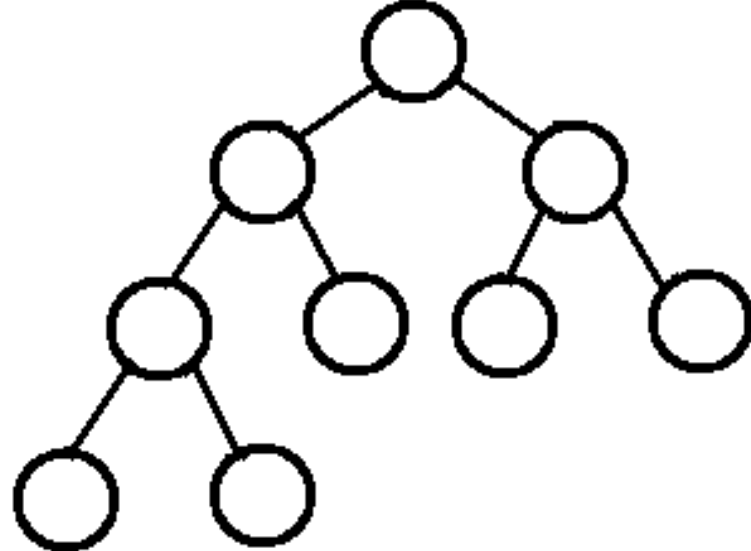
# Structura de date Heap

---

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



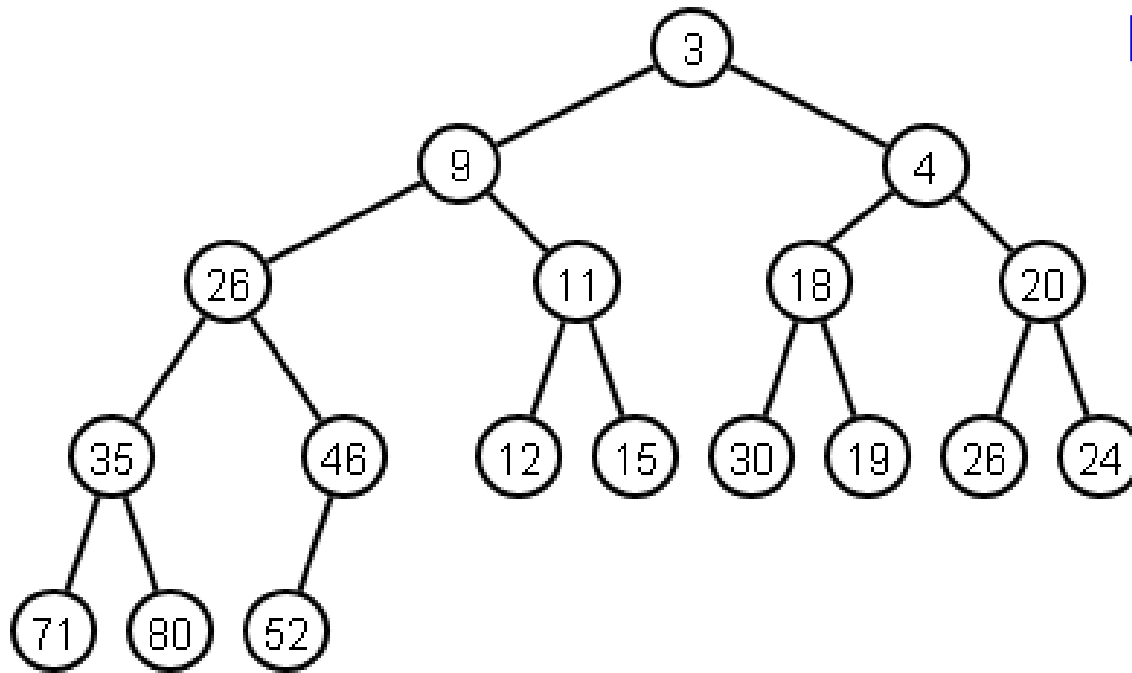
**Arbore binar plin**



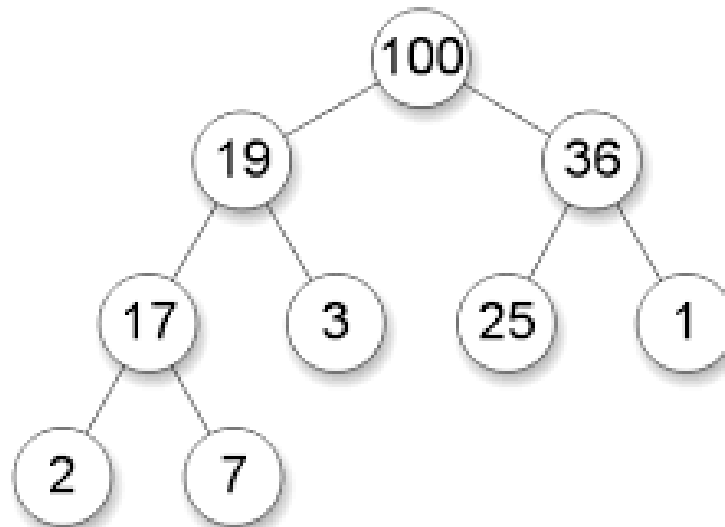
**Arbore binar complet**



## Heap-min binar



## Heap-max binar





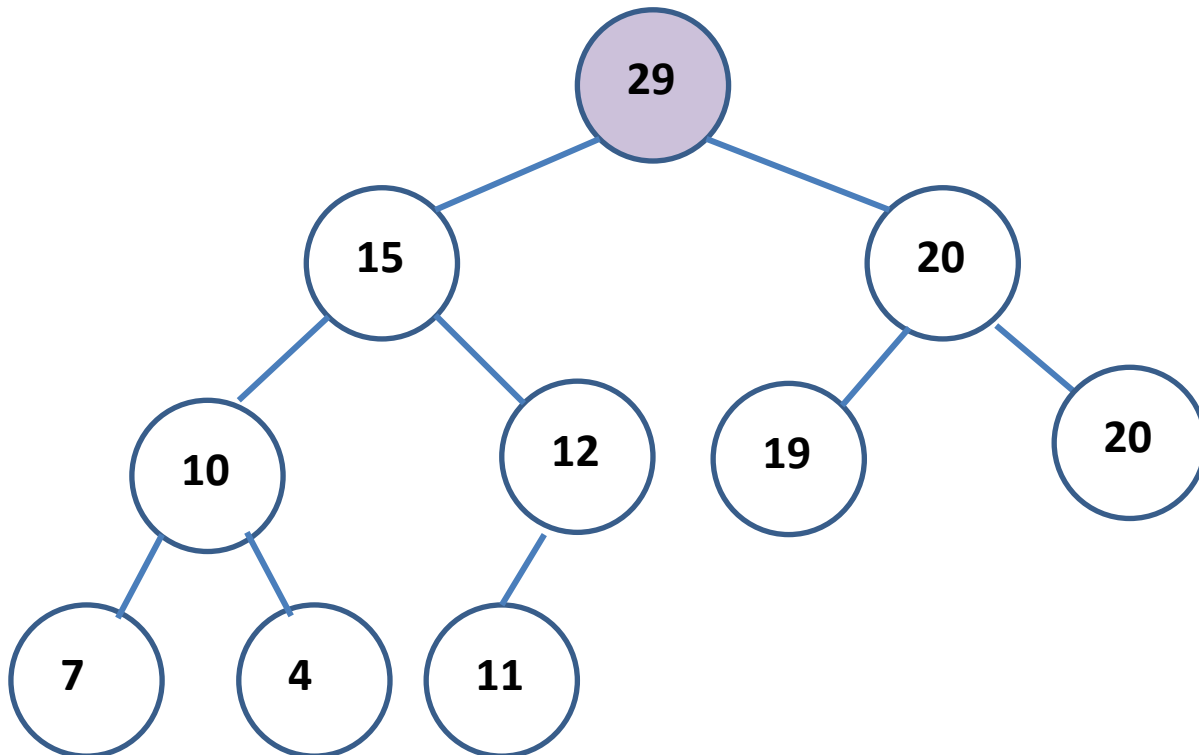
# Proprietate Heap

- 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

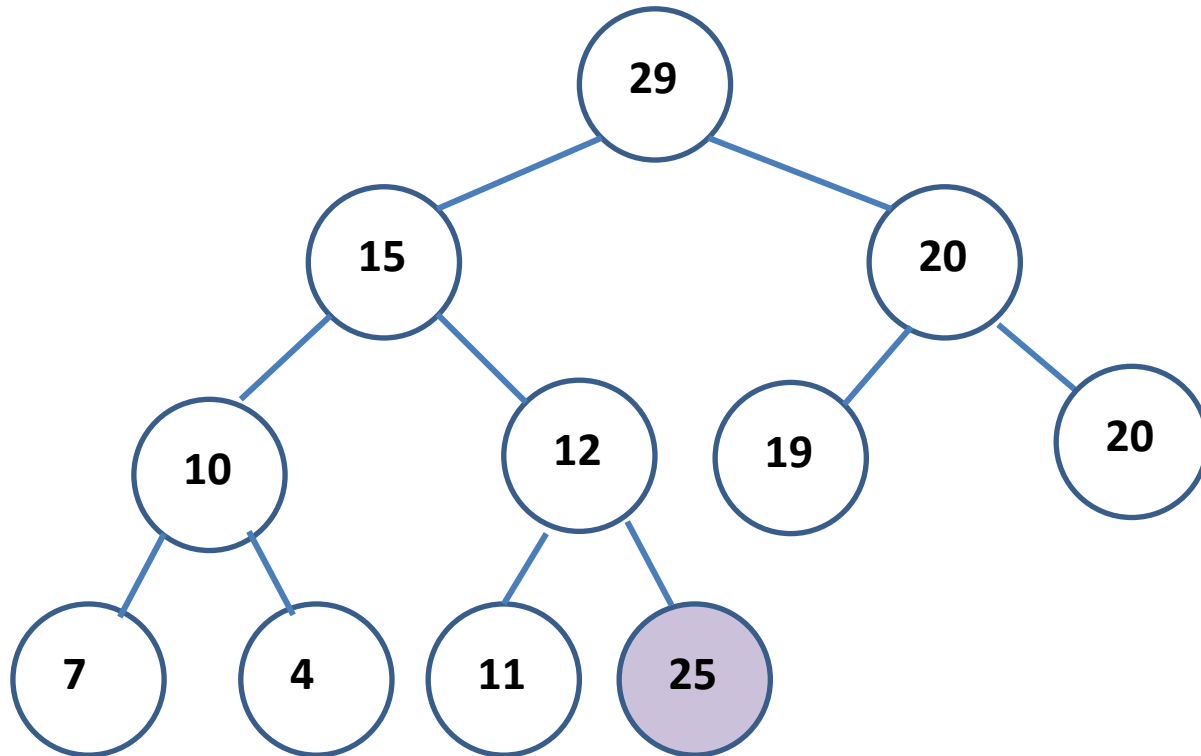
HEAP max

GetMax → 29



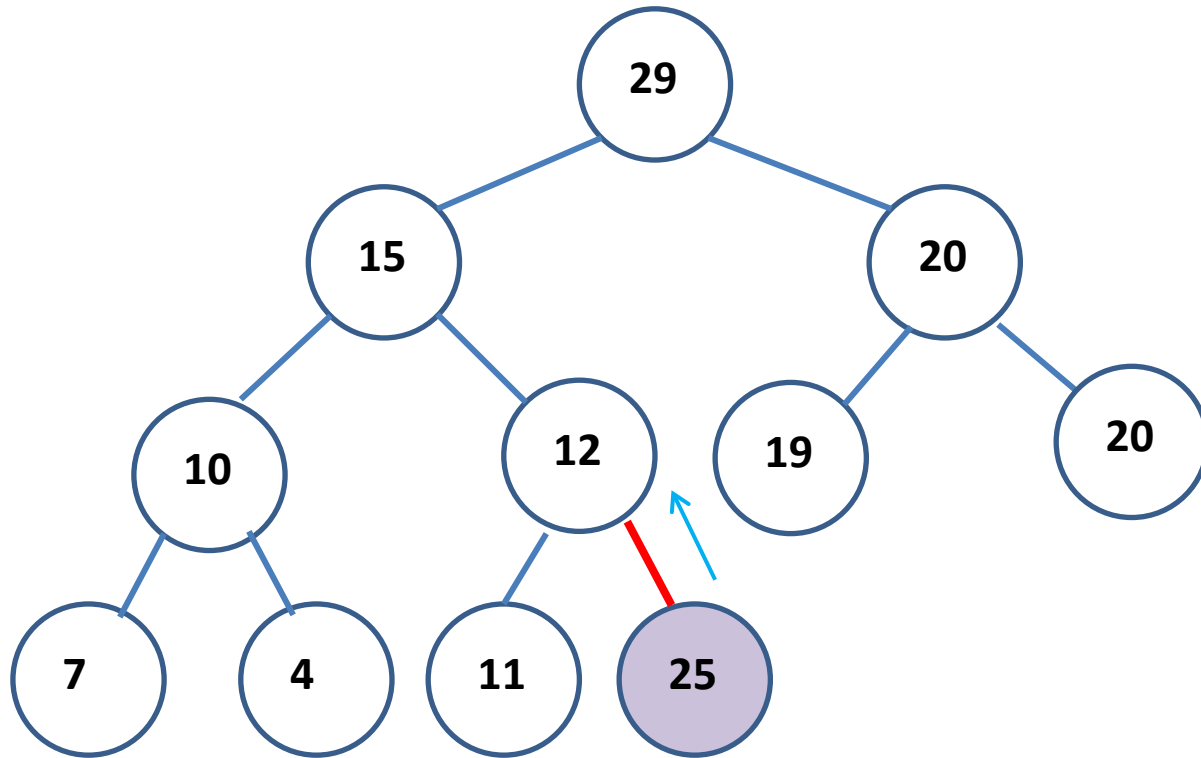
# HEAP

**Insert 25**



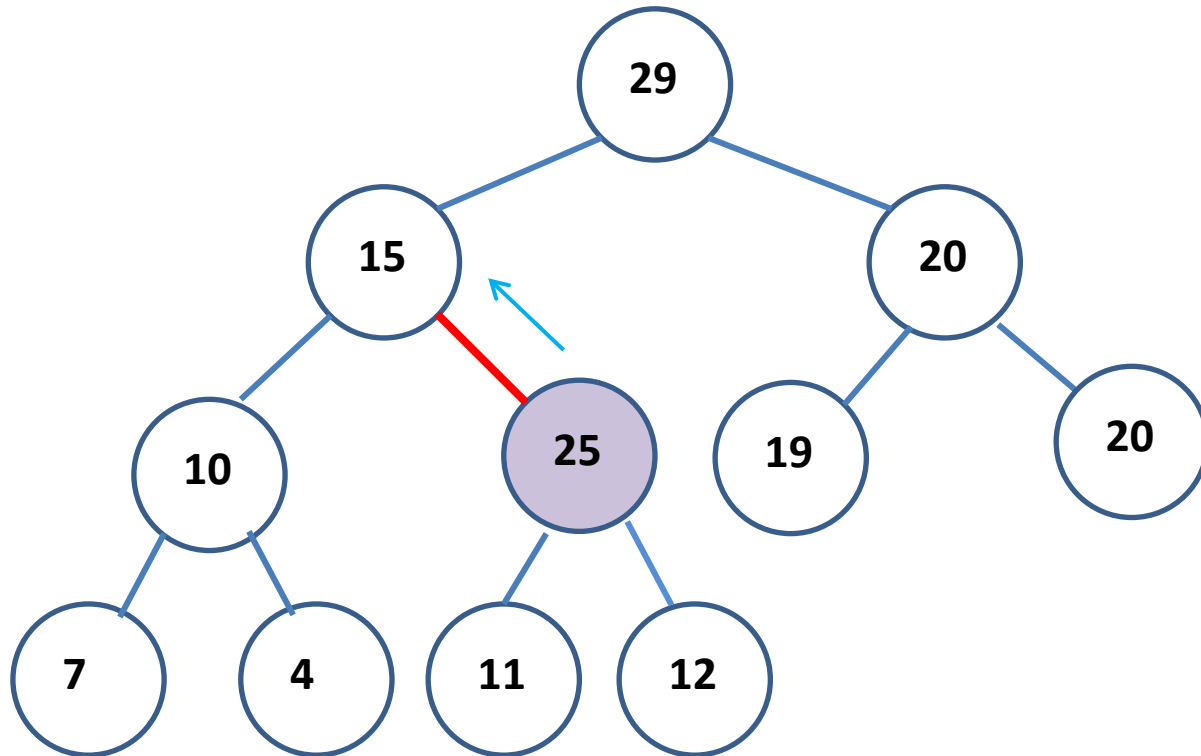
# HEAP

## SiftUp



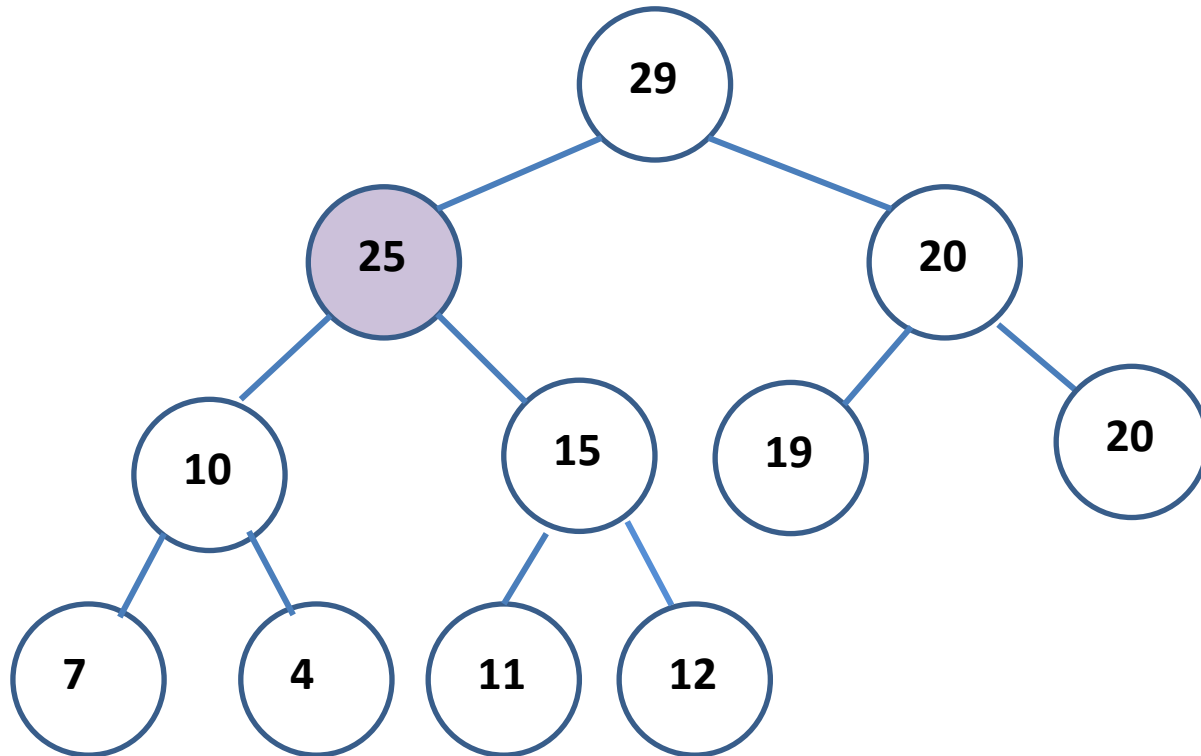
# HEAP

## SiftUp



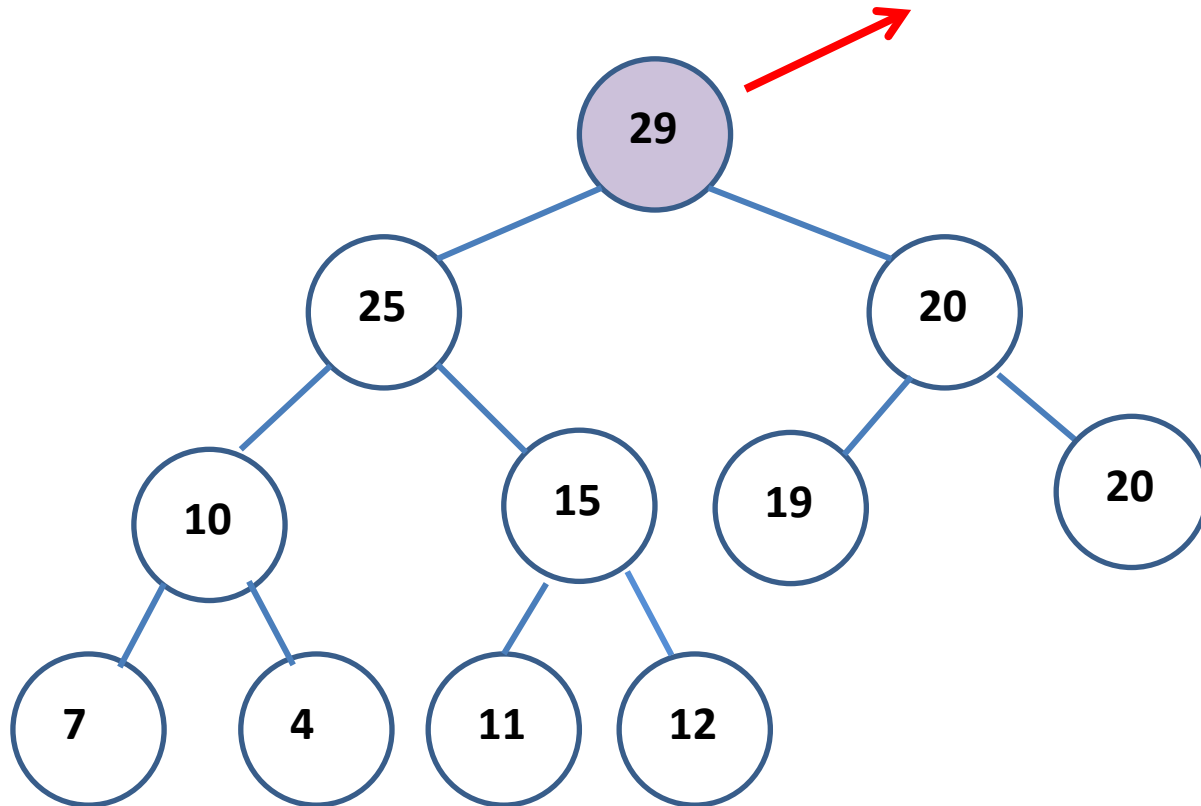
# HEAP

## Proprietatea de ordonare Heap refăcută



# HEAP

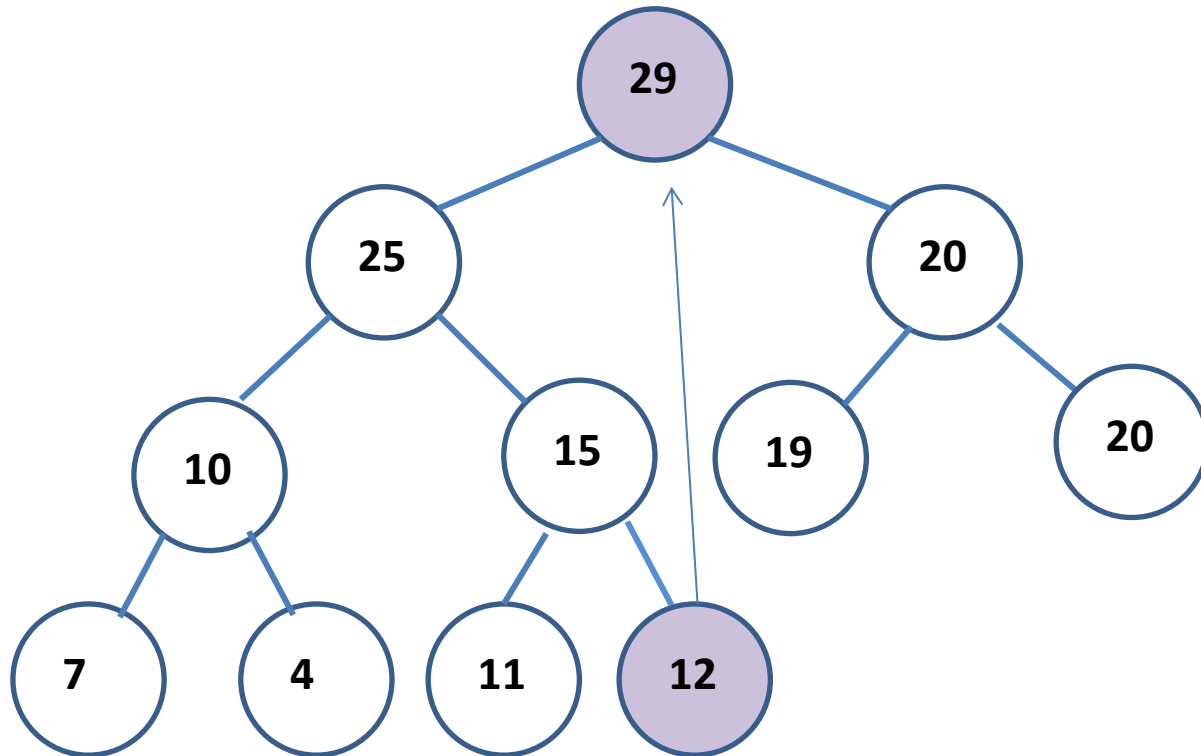
## RemoveMax





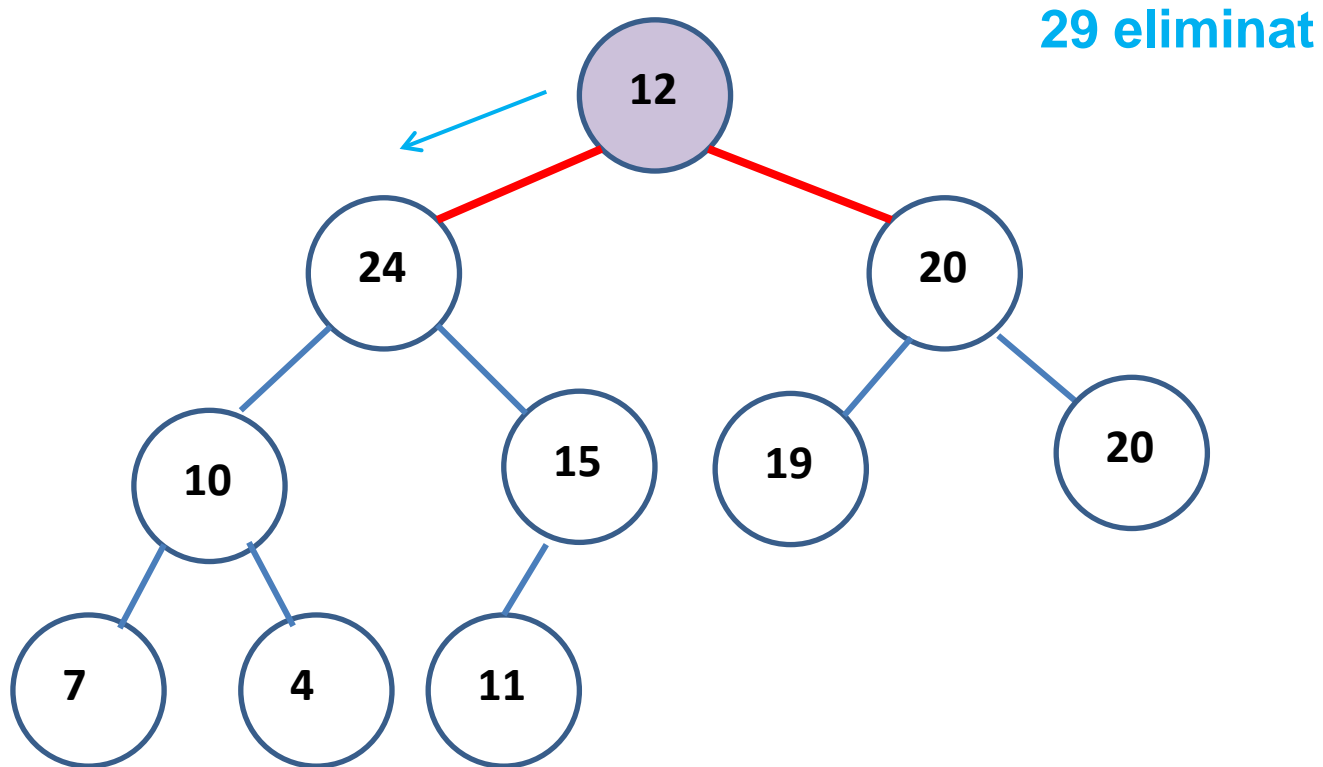
# HEAP

## RemoveMax



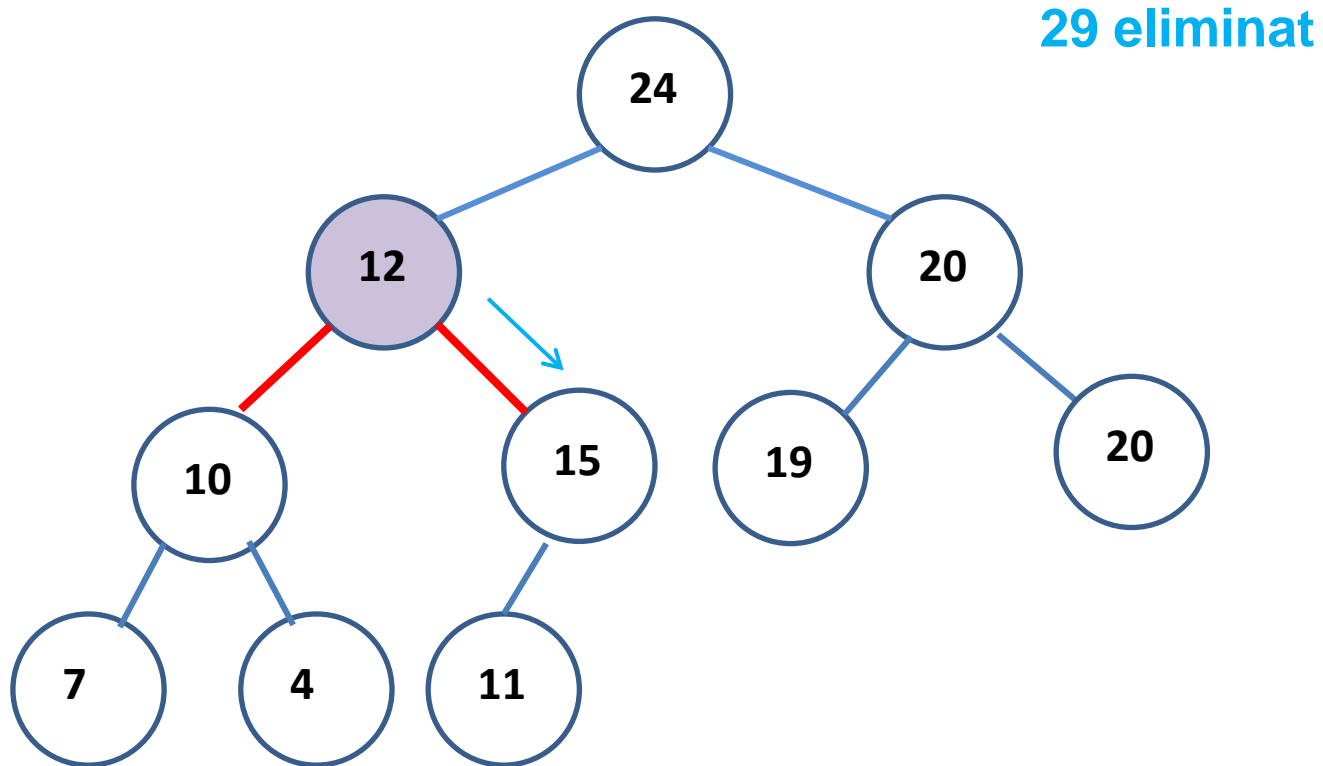
# HEAP

## SiftDown



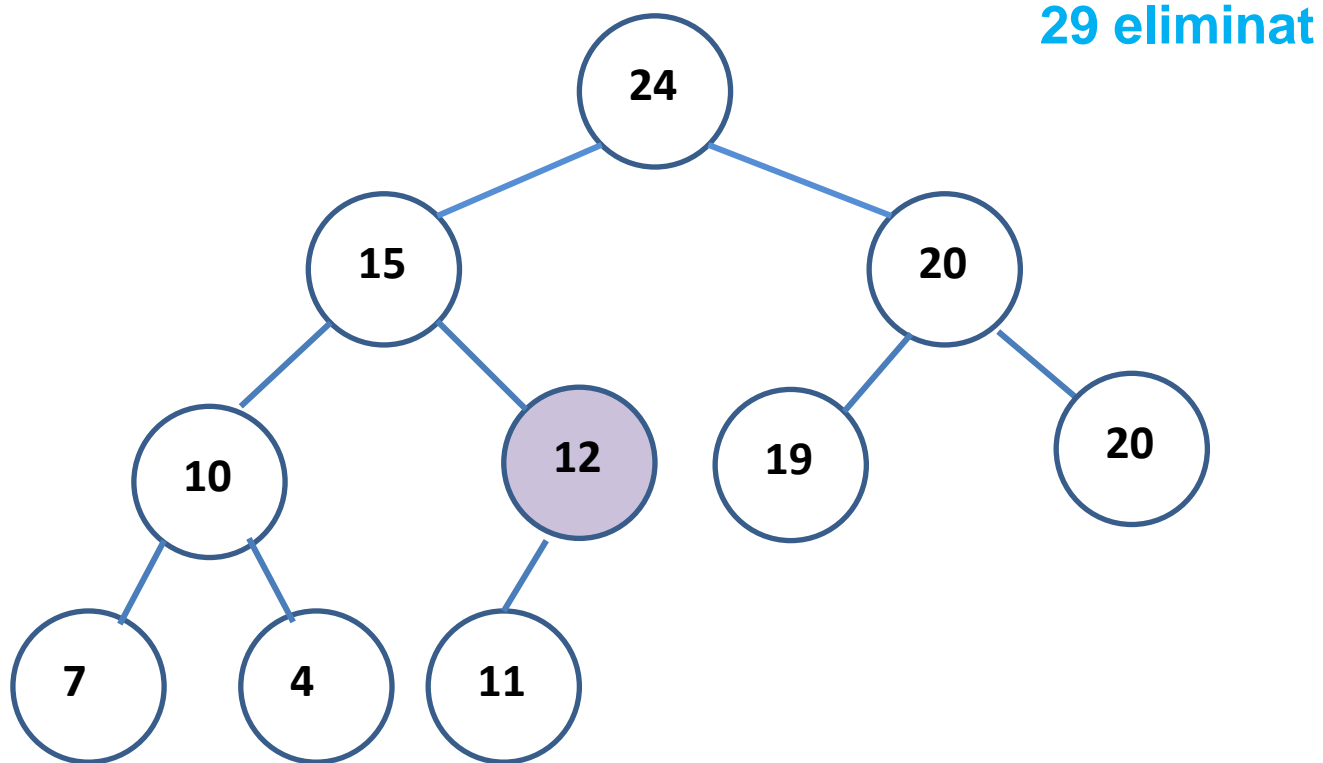
# HEAP

## SiftDown



# HEAP

## Proprietatea Heap refăcută

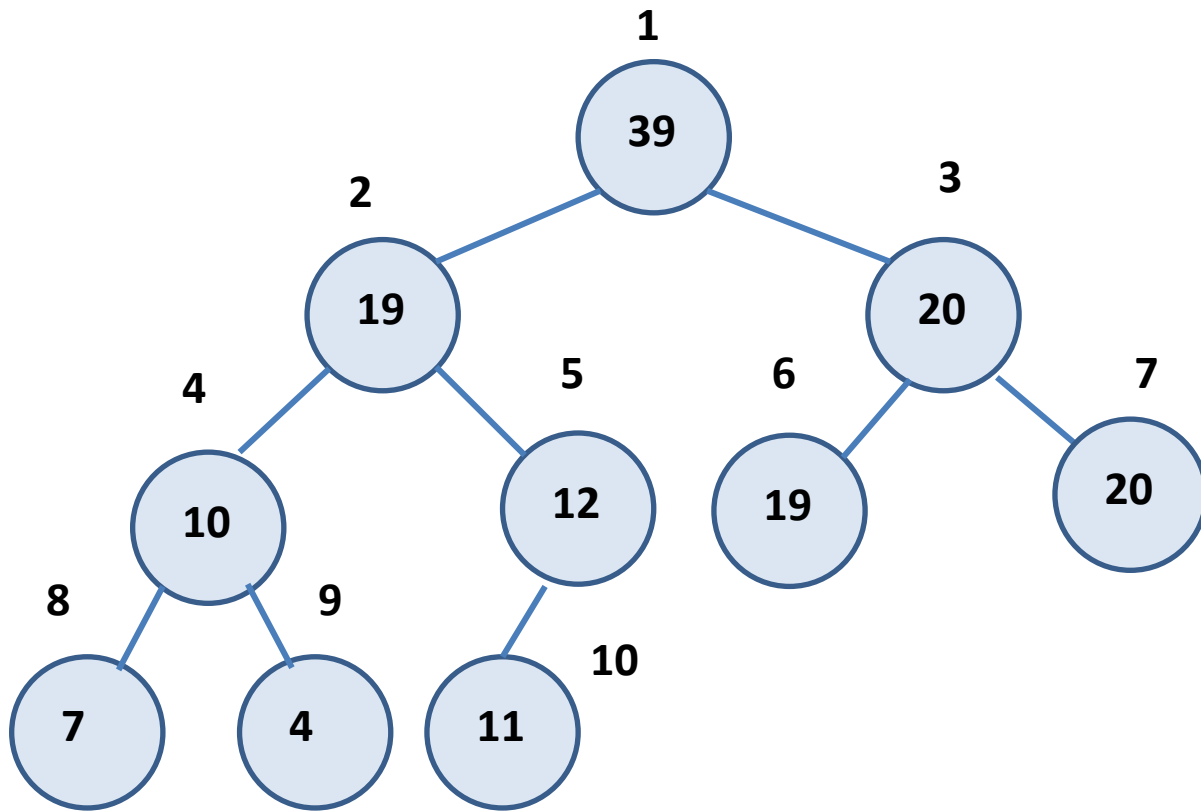


# 5. Reprezentare Heap

---

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

# Reprezentare heap ca vector



$\text{parent}(i) = \lfloor i/2 \rfloor$

$\text{leftchild}(i) = 2*i$

$\text{rightchild}(i) = 2*i + 1$

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

---

**Algorithm 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**

## Algorithm SiftDown(h, i)

/\* modifica h prin efect lateral \*/

maxIndex = i

left = LeftChild(i)                   /\* left = 2\*i \*/

**daca** left  $\leq$  h.Size **si** h.elem[maxIndex] < h.elem[left]

**atunci** maxIndex = left

right = RightChild(i)                   /\* right = 2\*i + 1 \*/

**daca** right  $\leq$  h.Size **si** h.elem[maxIndex] < h.elem[right]

**atunci**

**daca** h.elem[left] < h.elem[right]

**atunci** maxIndex = right

**if** i  $\neq$  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**  $2*i \leq h.Size$  **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**



## Algorithm 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  $a[1] \dots a[n]$** 
  - Creează a coadă de priorități vidă **h**
  - **pentru**  $i=1$  la  $n$  **repeta**  
    Insert ( $h, a[i]$ )
  - **pentru**  $i = n$  la  $1$  **repeta**  
     $a[i] = \text{ExtractMax}(h)$

**$O(n \log n)$**

# 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  $\geq$  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  $2 \cdot \ln(n)$
- Deci adâncimea arborelui este proporțională cu logaritmul din numărul 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  $\rightarrow$  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*)

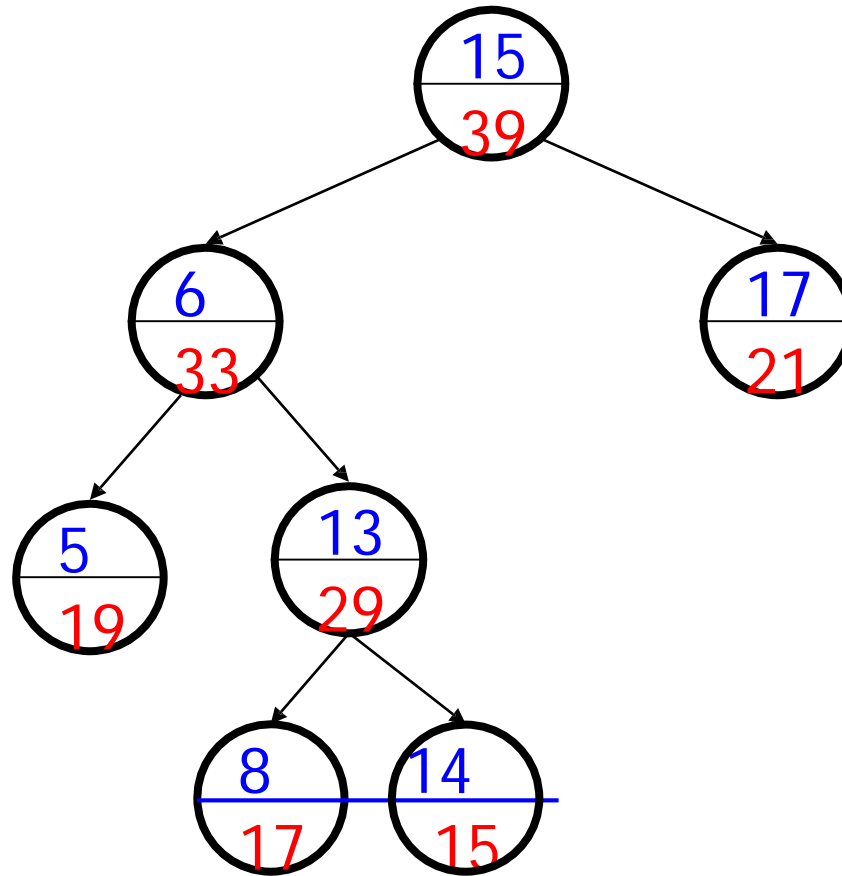
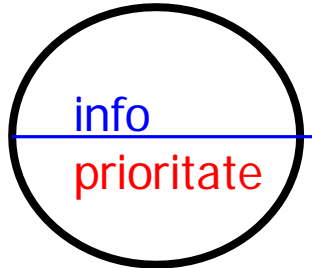


# Treaps

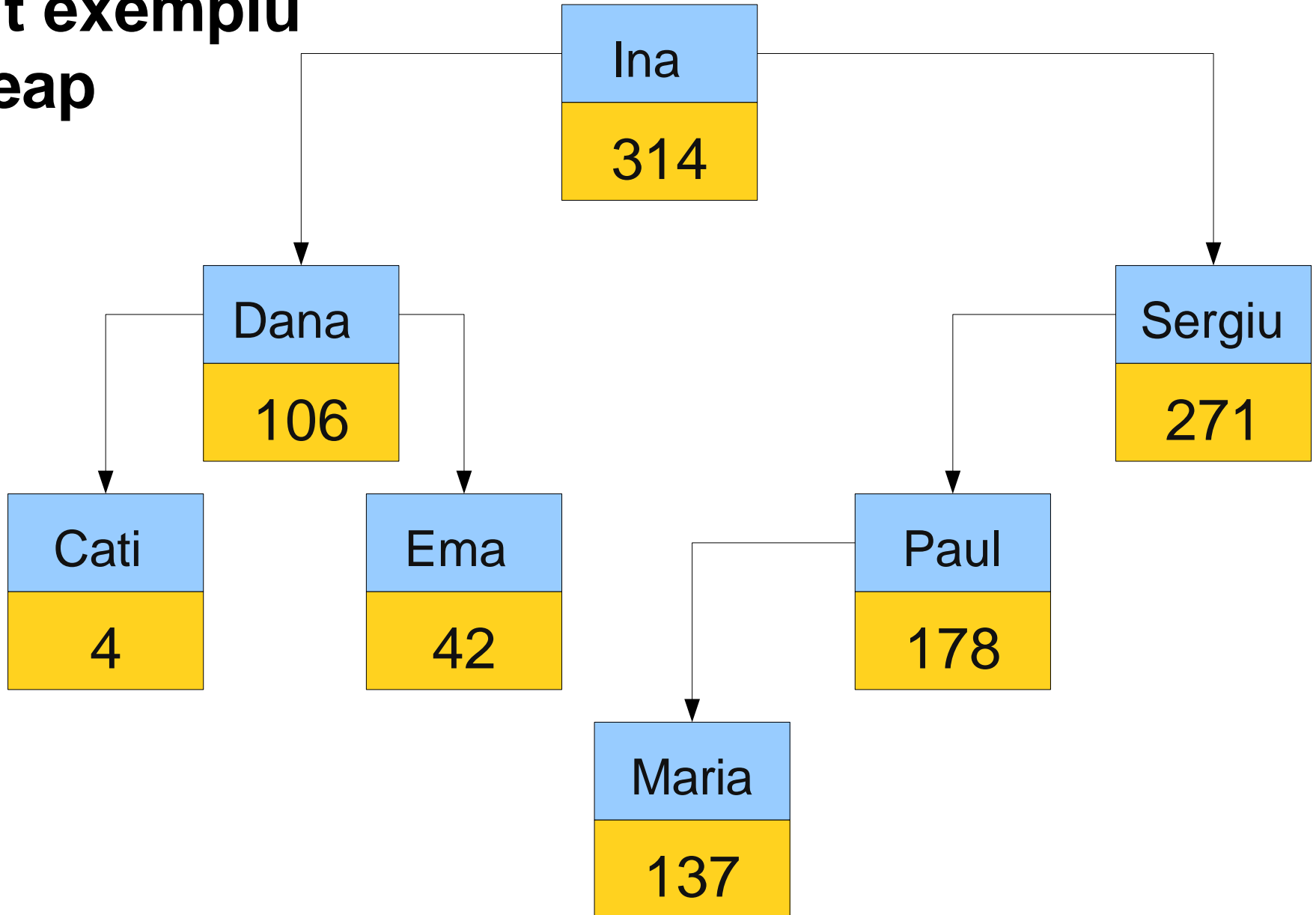
---

```
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 operații

- Union

- Intersection

- Set difference

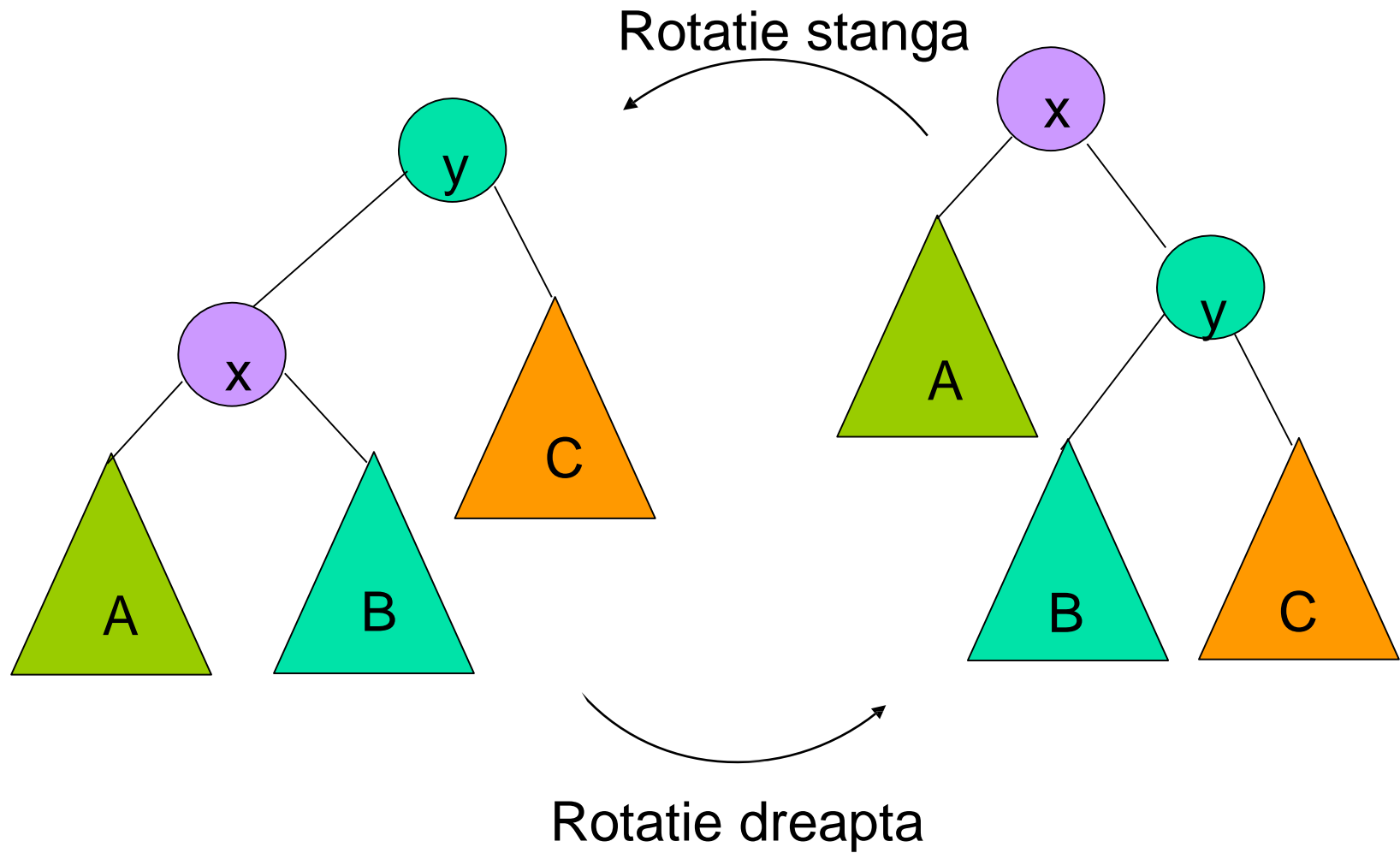
- Operații ajutătoare: Split și Join (Merge)

# Inserare in treap

---

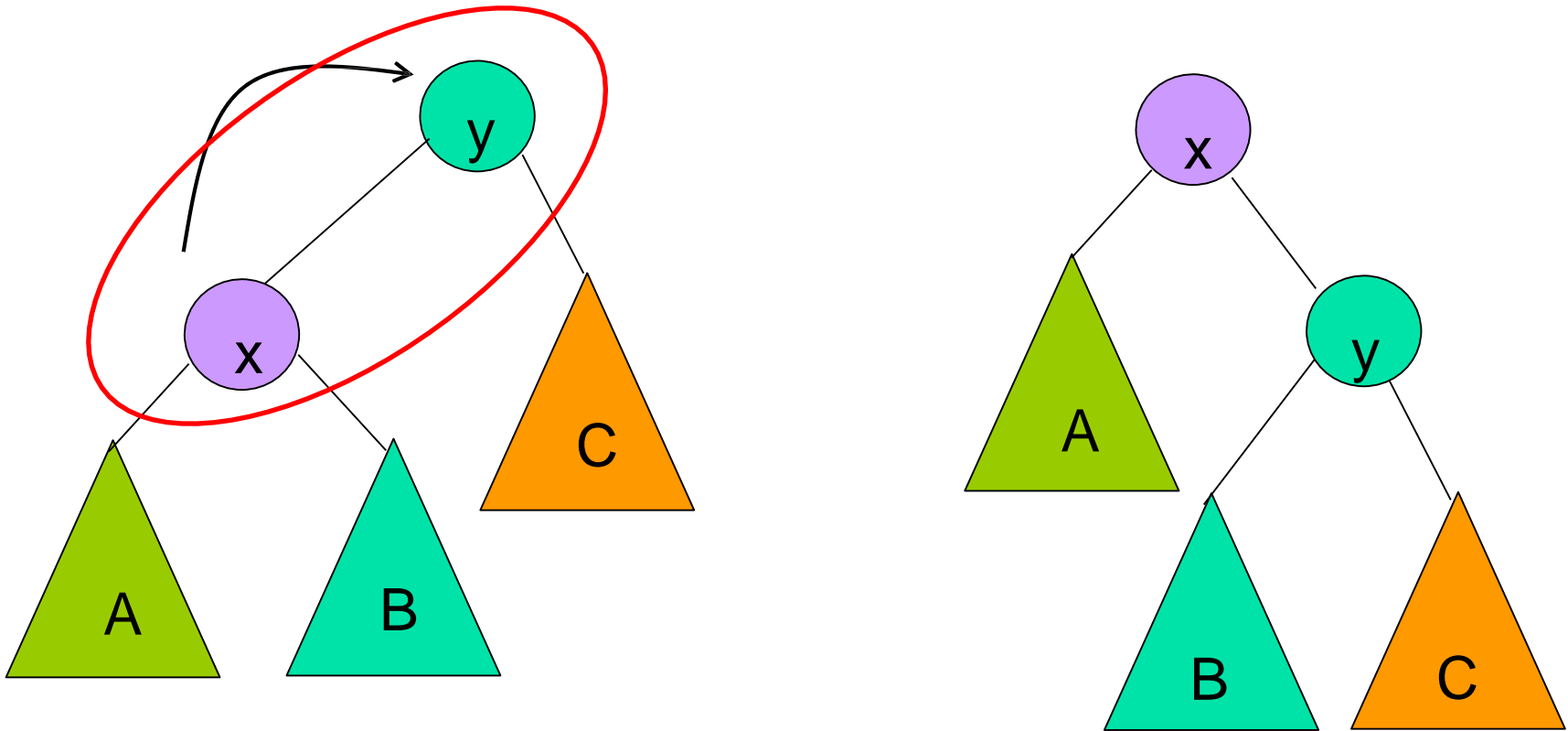
- se genereaza o prioritate aleatoare pentru nodul inserat
- **se insereaza** informatia in treap folosind regula de inserare de la **arbori binari de cautare**
- **se actualizeaza** structura arborelui pentru a se asigura conditia de ordonare din **heap** raportata la prioritatile din nodurile arborelui: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

# Rotatii



# Rotatie dreapta

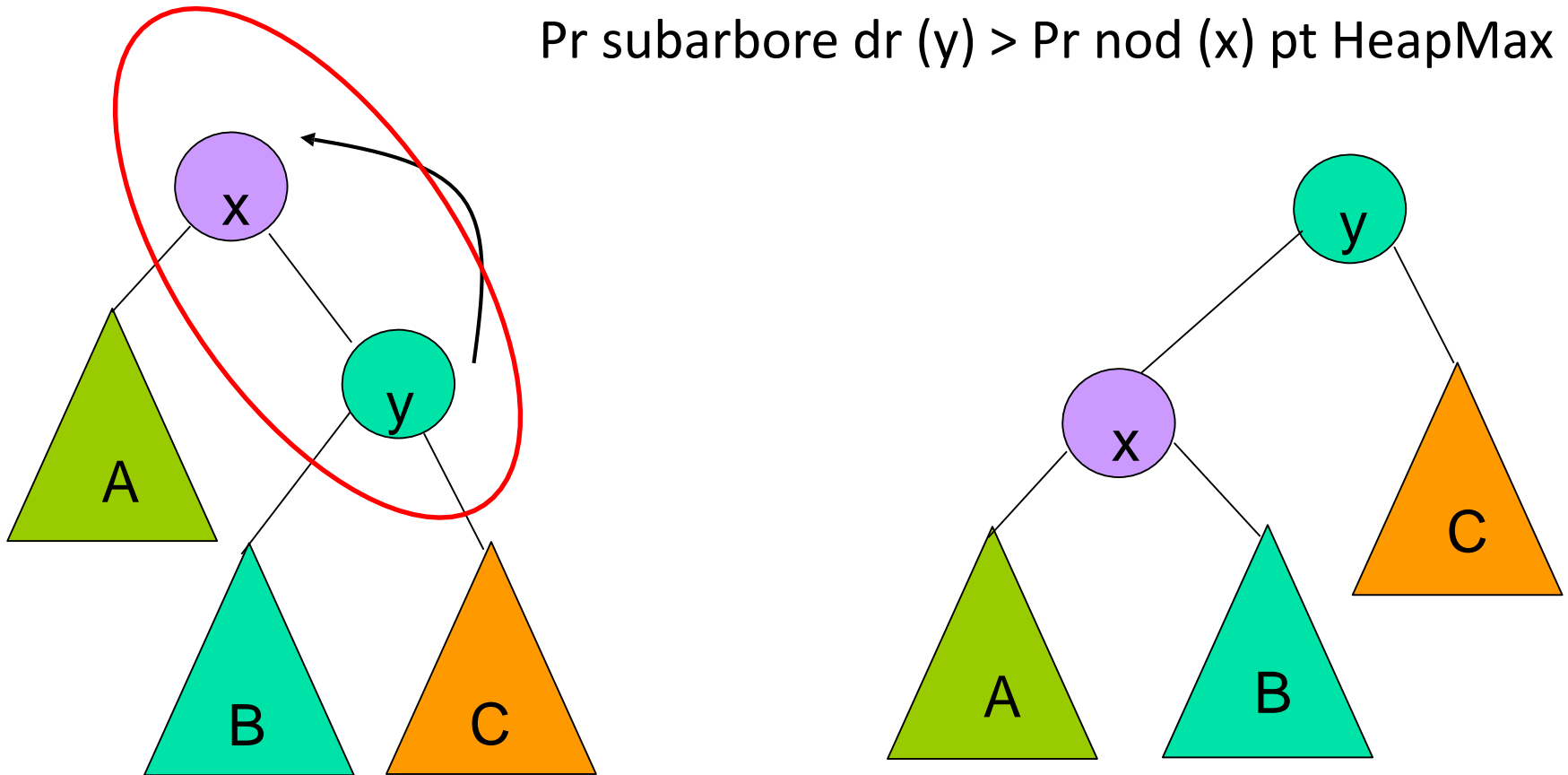
Pr subarbore st (x) > Pr nod (y) pt HeapMax



Rotatie dreapta

# Rotatie stanga

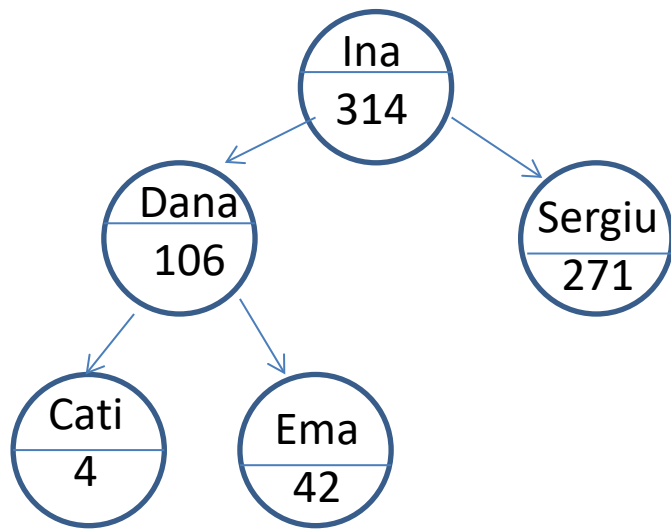
Pr subarbore dr (y) > Pr nod (x) pt HeapMax



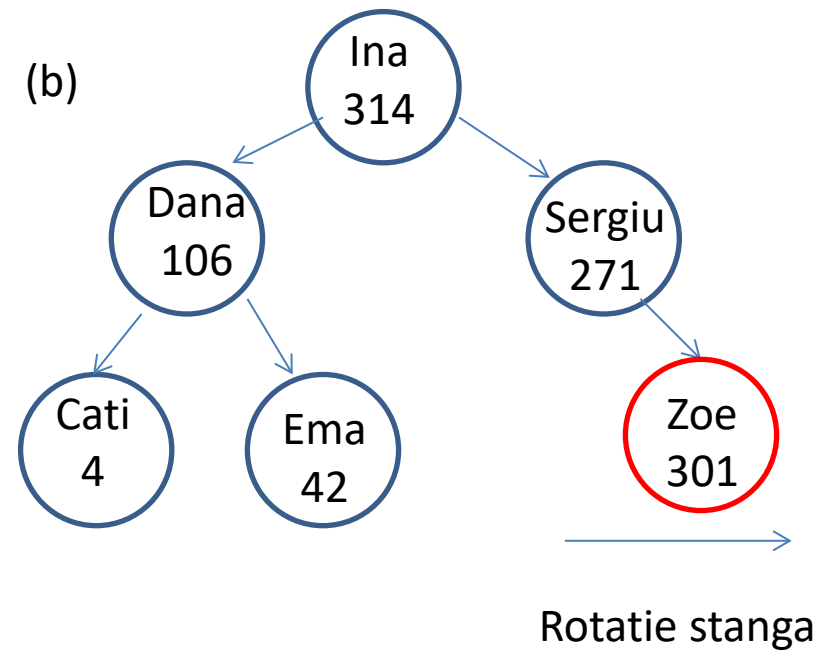
Rotatie stanga



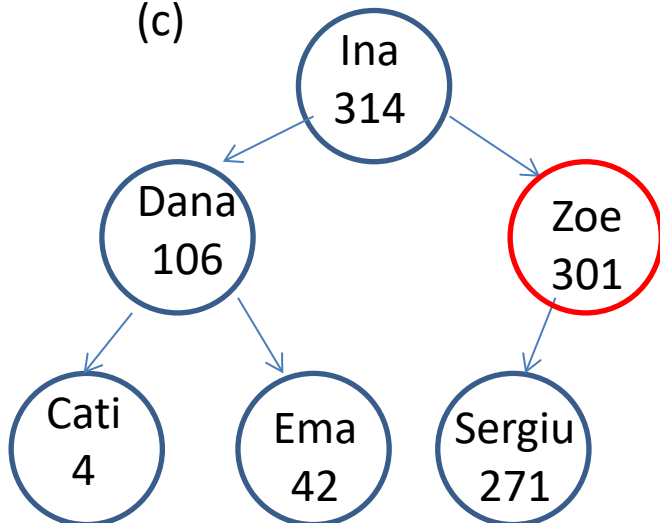
(a)



(b)



(c)



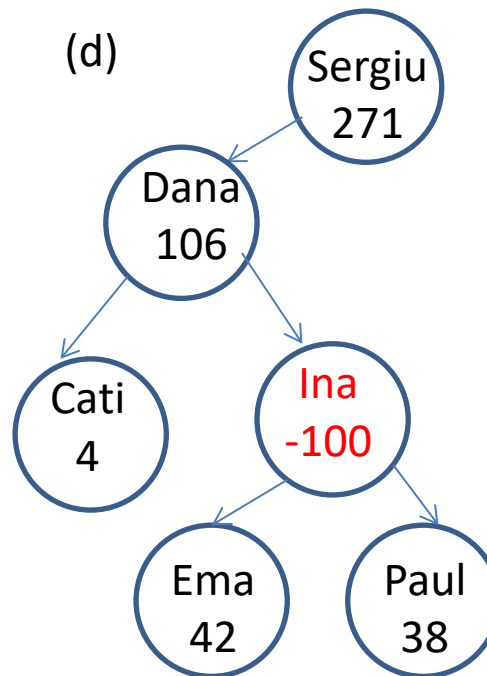
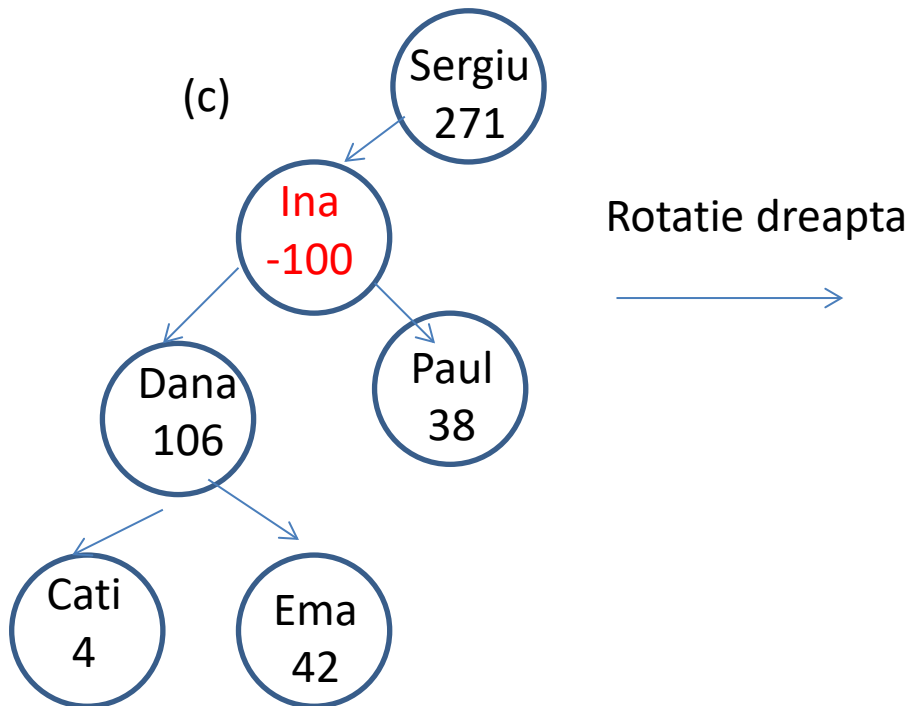
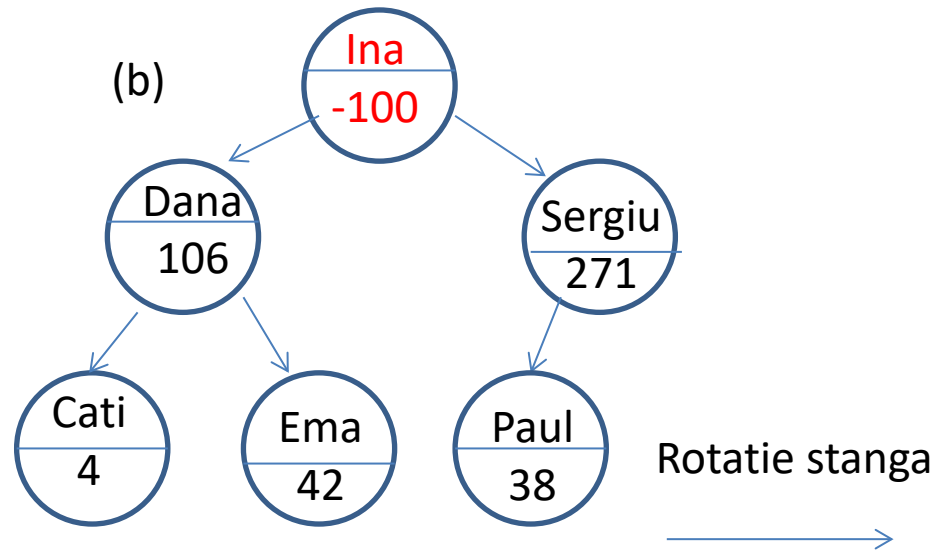
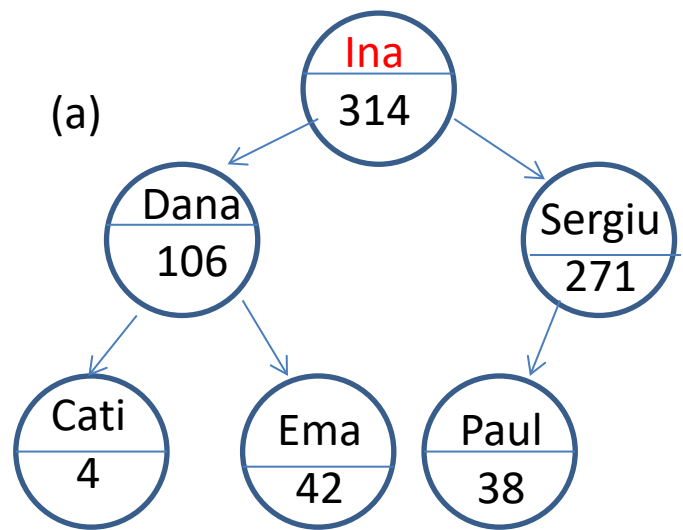
**Inserare cheie Zoe,  
prioritate 301 generata aleator**

# Eliminare

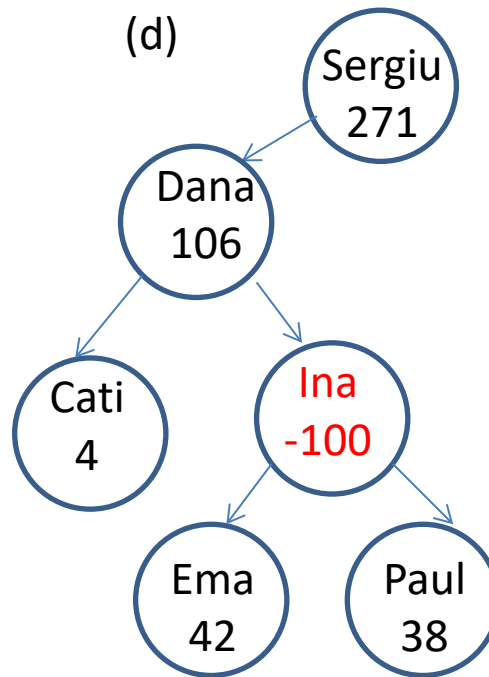
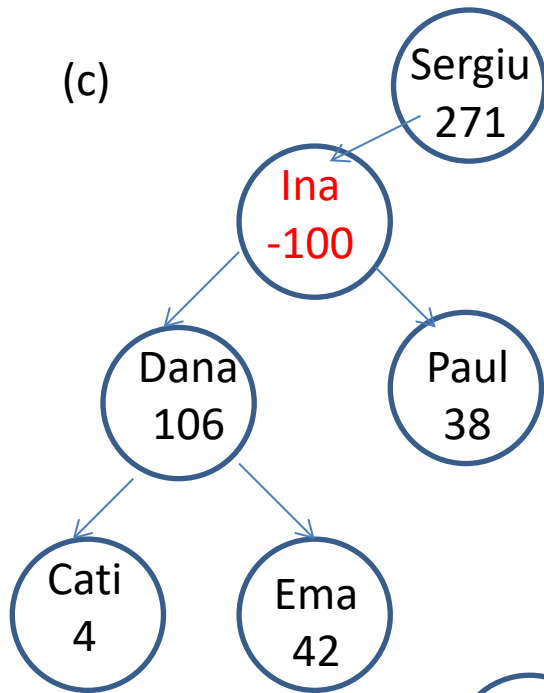
---

- 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  $x$

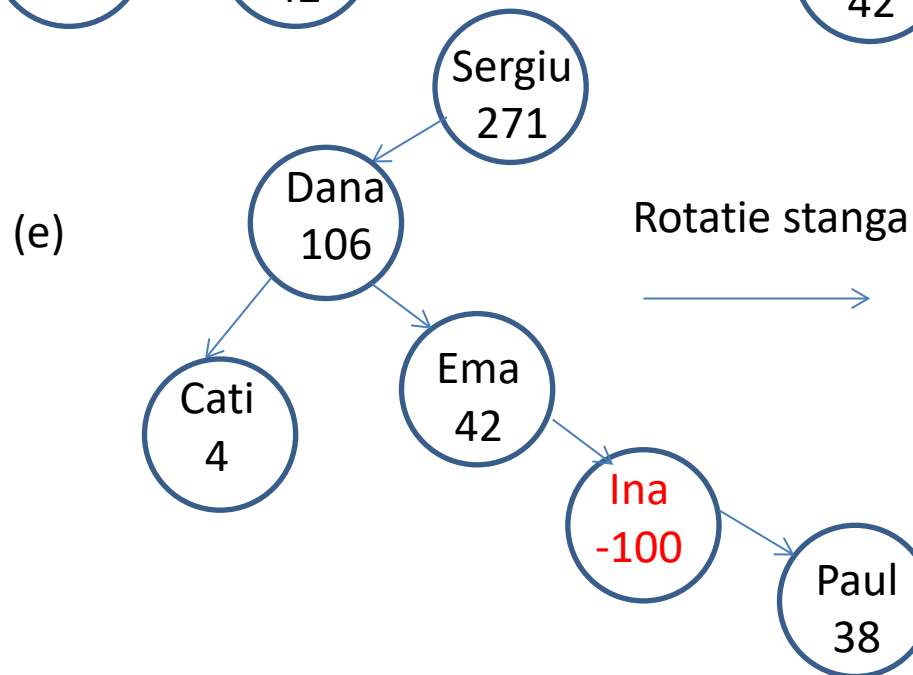
# Eliminare cheie Ina



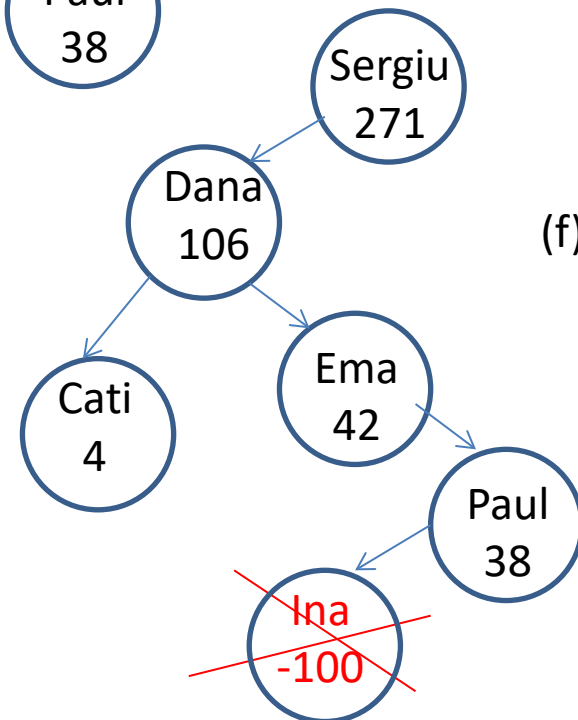
## Eliminare cheie Ina



Rotatie dreapta



Rotatie stanga



# Treap – alte operatii

---

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

# Split

---

- Treap  $T$
- Separa  $T$  in doua treapuri  $T_L$  si  $T_R$  dupa cheia  $k$
- $\text{split}(T, k)$  rezultă  $T_L$  si  $T_R$
- $\forall x_1 \in T_L, x_2 \in T_R :$   
 $\text{key}(x_1) \leq k$  si  $\text{key}(x_2) > k$

# Split

---

## Caz 1: Cheia $k$ nu se afla in treap

- Genereaza un nou nod  $x$  cu cheia  $k$  si prioritatea  $\infty$ .
- 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  $T_L$  și  $T_R$
- **Join** se realizează în mod **invers** operației de **split** prin unirea celor 2 treapuri  $T_L$  și  $T_R$  în jurul unei chei  $k$

$\forall x_1 \in T_L, x_2 \in T_R:$

$$\text{key}(x_1) < k < \text{key}(x_2)$$

- $\text{join}(T_L, k, T_R)$  – rezulta un treap
- Se creează un nod rădăcină cu cheia  $k$  și prioritate  $\infty$ , ce are ca subarbore stâng pe  $T_L$ , iar ca subarbore drept pe  $T_R$
- Sterge nodul cu cheia  $k$





# Union

- construiești un nou treap, pe baza a două treapuri  $T1$  și  $T2$

**union**( $T1, T2$ ) /\* întoarce un nou treap  $T$  \*/

**dacă**  $T1 == \text{vid}$  **atunci întoarce**  $T2$

**altfel dacă**  $T2 == \text{vid}$  **atunci întoarce**  $T1$

**altfel dacă**  $\text{prioritate}(\text{radacina}(T1)) < \text{prioritate}(\text{radacina}(T2))$   
**atunci** interschimba  $T1$  cu  $T2$

/\*  $\text{prioritate}(\text{radacina}(T1)) > \text{prioritate}(\text{radacina}(T2))$  \*/

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

/\* Toate cheile din  $TL1 \leq r$  și toate cheile din  $TR1 > r$  \*/

$TL2, TR2 = \text{split}(T2, r)$

**întoarce**  $\text{join}(\text{union}(TL1, TL2), r, \text{union}(TR1, TR2))$

# Alte operatii

---

- Intersection
- Set difference

Pentru examen

## Treaps

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

## 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