

MODULUL 3 - COZI DE PRIORITATE

1 Heap

Definiție: Un heap binar este un arbore binar complet - fiecare nod intern are exact doi descendenți, cu excepția eventual a ultimului nod intern de pe penultimul nivel, iar frunzele se află doar pe ultimele două niveluri, frunzele de pe ultimul nivel sunt ordonate de la stânga spre dreapta - memorat cu ajutorul unui tablou unidimensional (vector) - *array*. În plus există o ordonare a cheilor într-un heap binar, determinată de tipul heap-ului.

H:

16	14	10	8	7	9	3	2	4	1
----	----	----	---	---	---	---	---	---	---

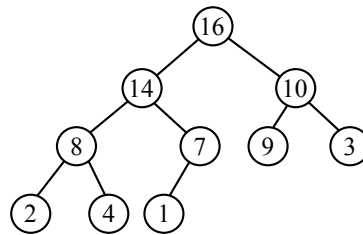


Figure 1: Exemplu de heap-max

1. **Max-heap:** informația din fiecare nod este mai mare decât informația din oricare descendent al său. Maximul din heap se află în rădăcină (fig. 1).
2. **Min-heap:** informația din fiecare nod este mai mică decât informația din oricare descendent al său. Minimul din heap se află în rădăcină.

Pentru implementare se poate utiliza structura heap H în care *size* - numărul de chei din Heap. Rădăcina heap-ului se află pe prima poziție a heapului $H[0]$. Pentru fiecare nod aflat pe poziția i :

- Părintele se află pe poziția $(i - 1)/2$.
- Fiul stâng se află pe poziția $2 * i + 1$.
- Fiul drept se află pe poziția $2 * i + 2$.

Observații:

- Înălțimea arborelui care reprezintă heap-ul este $\log_2 n$ unde $n = H.size$ este numărul de noduri din heap.
- Complexitatea operațiilor de bază este proporțională cu înălțimea arborelui, adică $O(\log_2 n)$.

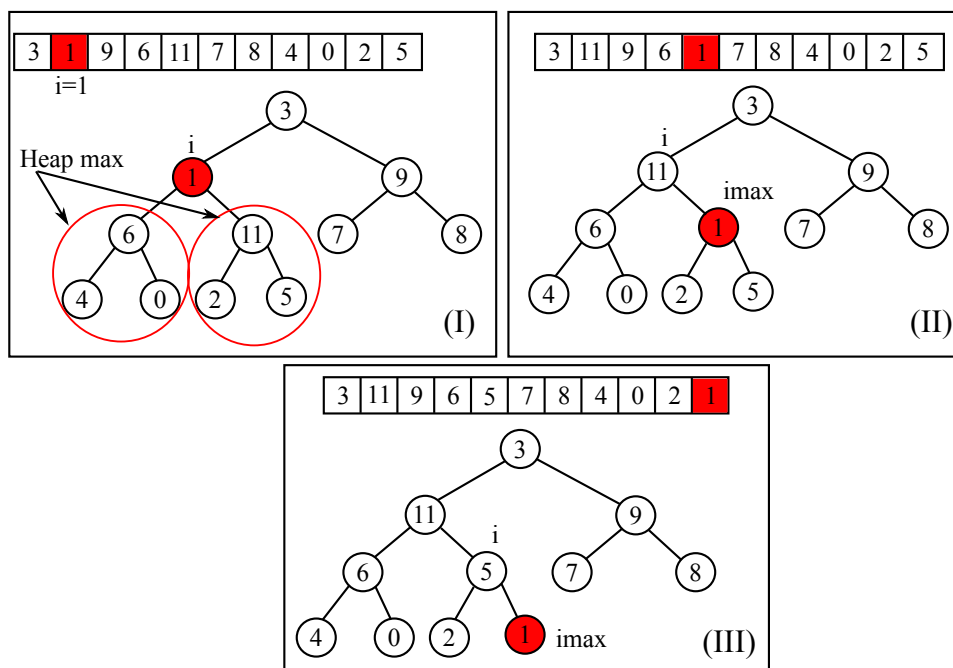


Figure 2: Funcția `MAX_HEAP` pornește în acest exemplu de la nodul i marcat cu roșu în fig. (I). Cei doi subarbori ai săi sunt heap-max. Funcția are două apeluri recursive, ilustrate în etapele (II) și (III).

Construcția unui heap - max

Considerând un vector de elemente, pentru transformarea acestuia într-un heap-max sunt necesare două etape, reprezentate prin funcțiile:

- `MAX_HEAP(H, i)`: în care H heap cu $H.size$ elemente și i indicele din vector a nodului de la se începe. Premiza este aceea că subarborii nodului i sunt heap-max și doar informația din nodul i strică eventual această proprietate. Funcția `MAX_HEAP` reface proprietatea de heap-max. Această funcție se întâlnește în literatură și sub denumirea *Sift-Down*.
- `CONSTR_HEAP(H)`: pe baza funcției `MAX_HEAP` se construiește heap-ul.

`MAX_HEAP(H, i)`

$st = 2 * i + 1$

$dr = 2 * i + 2$

```

imax = i
daca st < H.size si H[st] > H[i] atunci
    imax = st
sfarsit daca
daca dr < H.size si H[dr] > H[imax] atunci
    imax = dr
sfarsit daca
daca imax ≠ i atunci
    H[imax] ↔ H[i]
    MAX_HEAP(H, imax)
sfarsit daca
RETURN

```

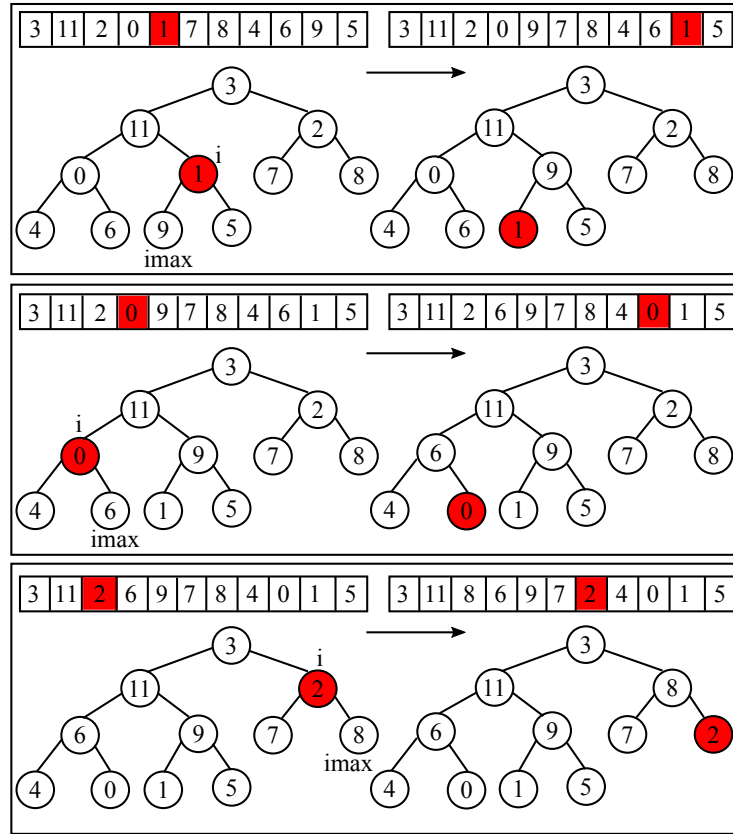


Figure 3: Funcția CONSTR_HEAP pornește în acest exemplu de la nodul de pe poziția a 5-a în heap-ul H , primul nod din dreapta vectorului ce are descendenți. Apoi se continuă cu nodurile de pozițiile 4 și 3.

În figurile 3 și 4 este ilustrată funcționarea acestui algoritm.

Complexitate: Algoritmul pornește de la nodul i și ajunge în cel mai defavorabil caz la o frunză, deci complexitatea depinde de înălțimea arborelui care este $h = \log_2 n$. Deci $T(n) = O(\log_2 n)$.

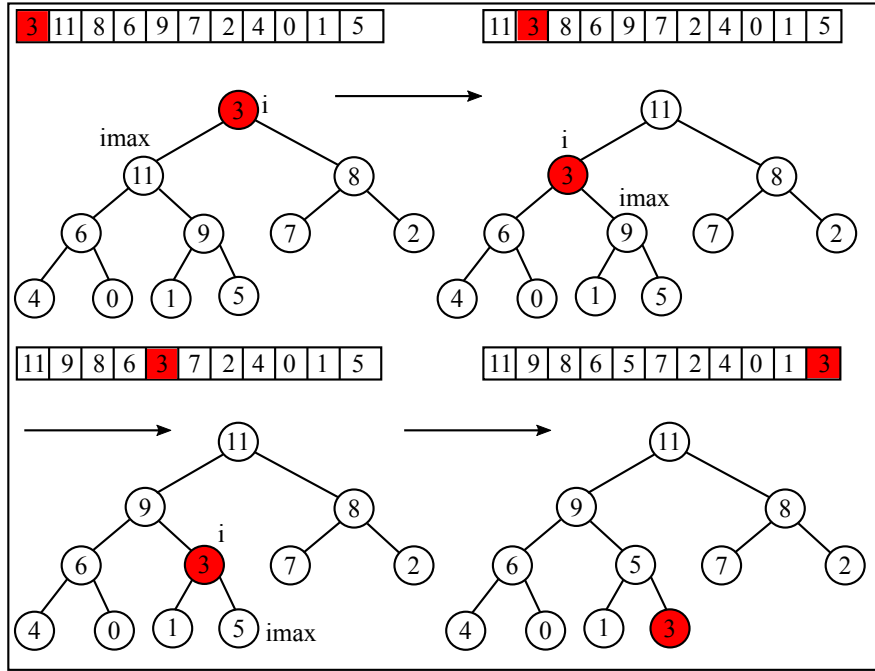


Figure 4: Funcția MAX_HEAP continuă de la poziția 2, la care însă nu sunt necesare modificări. Nodul cu cheia 11 satisface proprietatea de max-heap relativ la descendenții săi, astfel ca se trece la nodul de pe poziția 1. În final se obține un heap-max.

Se observă faptul că frunzele sunt heap-max. Atunci e suficient să pornim de la primul nod intern - pornind de la dreapta heap-ului H , adică primul care are cel puțin un descendent. Nodul respectiv se află pe poziția $H.size/2 - 1$, iar copiii săi sunt frunze, deci heap-max. Se aplică funcția de MAX_HEAP pentru nodul $H.size/2 - 1$, după care se trece la nodul precedent din vector și așa mai departe, până la rădăcină.

```

CONSTR_HEAP(H)
    pentru  $i = H.size/2 - 1, 0, -1$ 
        MAX_HEAP(H, i)
    sfarsit pentru
RETURN

```

În figura 4 este ilustrat procedeul de construcție a unui heap max.

Complexitate: fiecare apel al funcției MAX_HEAP are complexitatea $O(\log_2 n)$.

CONSTR_HEAP efectuează $O(n)$ atfel de apeluri. Deci $T(n)$, unde prin T am notat complexitatea, pentru funcția CONSTR_HEAP este mărginită superior de $n \log_2 n$. Se demonstrează în literatură faptul că funcția are complexitate liniară, adică $T(n) = O(n)$, $n = H.size$.

2 Algoritmul de sortare *HeapSort*

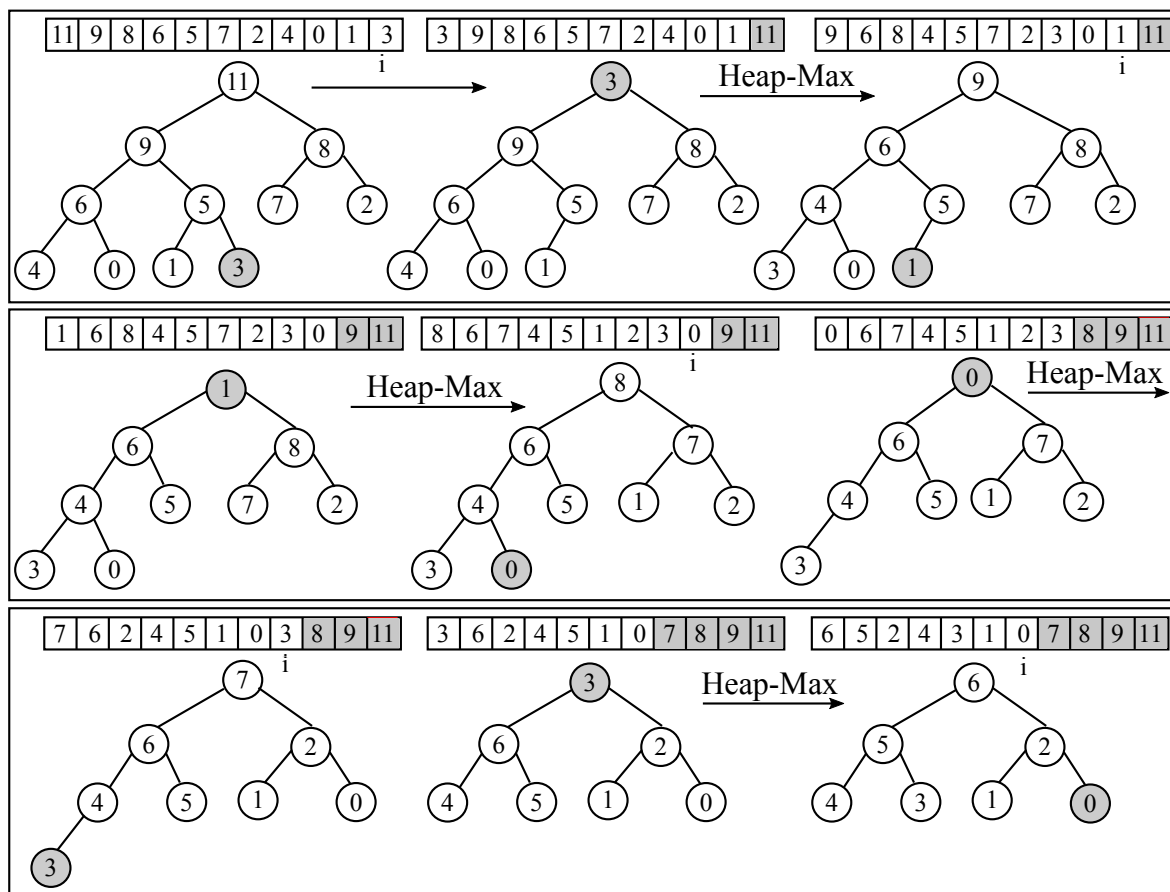


Figure 5: Algoritmul Heapsort.

Se observă faptul că într-un heap-max elementul maxim este plasat în rădăcina arborelui corespunzător, deci pe prima poziție din vectorul date. În vectorul sortat crescător, elementul maxim trebuie să se afle pe ultima poziție. Astfel vectorul poate fi sortat prin următorul algoritm: se interschimbă primul element din vectorul heap-ului cu ultimul. Se reduce dimensiunea heap-ului, apoi se reface proprietatea de heap-max aplicând funcția MAX_HEAP începând din vârful heap-ului. Procedura se reia pentru acest heap redus.

```

HEAPSORT(H)
  CONSTR_HEAP(H)
  n = H.size - 1
  pentru i = n, 1, -1
    H[0] ↔ H[i]
    H.size = H.size - 1
    MAX_HEAP(H, 0)
  sfarsit pentru
RETURN

```

În figurile 5 și 6 este ilustrată funcționarea algoritmului Heapsort.

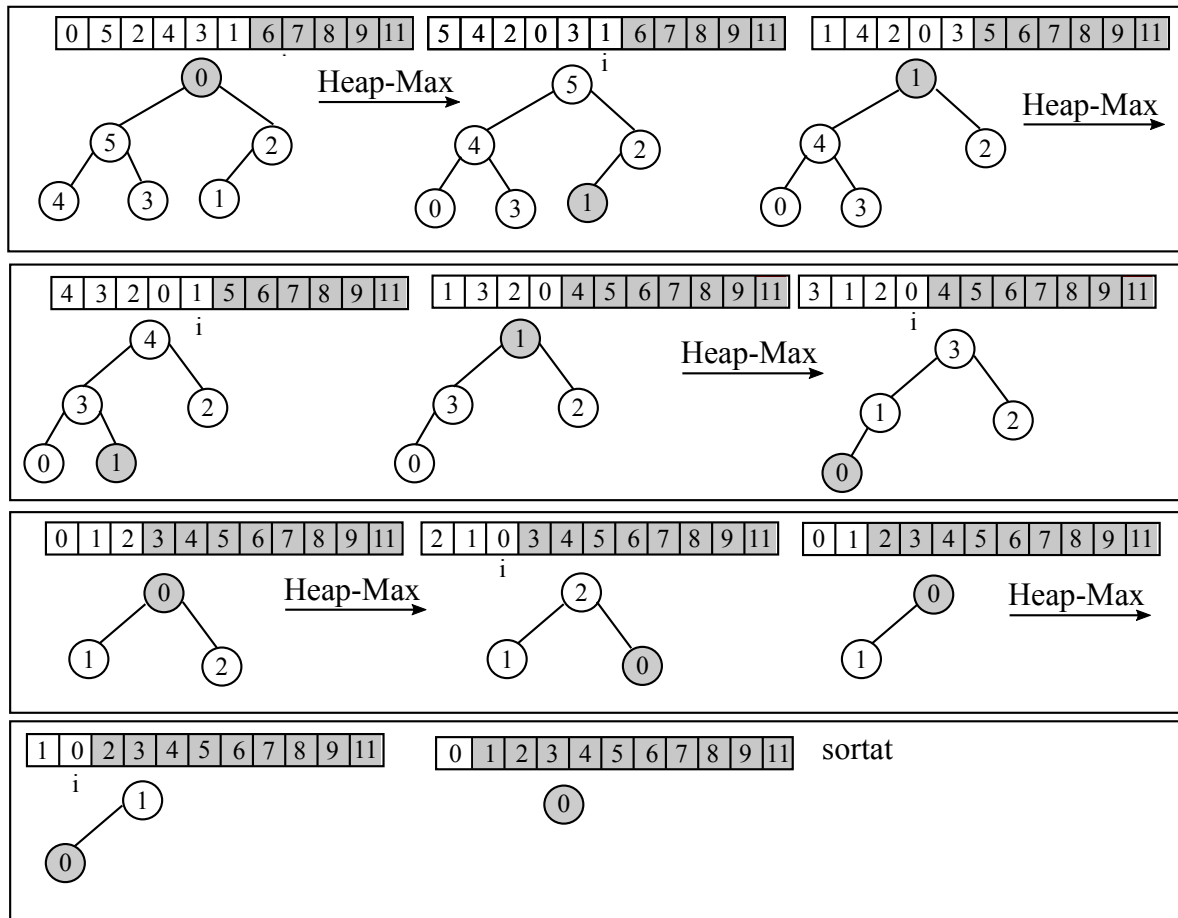


Figure 6: Algoritmul Heapsort.

Complexitate:

- apelul funcției `CONSTR_HEAP` este $O(n)$
- apelul pentru `MAX_HEAP` este $O(\log_2 n)$
- funcția `MAX_HEAP` se apelează de $n - 1$ ori

Rezultă $T(n) = O(n \log_2 n)$.

3 Cozi de prioritate

Definiție: O coadă de prioritate este o structură de date utilizată pentru păstrarea unei mulțimi dinamice de date S în care fiecărui element i se asociază o valoare numită *prioritate*. Pentru gestionarea eficientă a cozilor de prioritate se utilizează heap-uri. Există cozi de max-prioritate - heap-max - și cozi de min-prioritate - heap-min.

În continuare vom considera cozi cu max-prioritate.

Operații în cozi de prioritate (max-heap):

- Inserția unui element nou
- Determinarea elementului de prioritate maximă
- Extragerea elementului de prioritate maximă
- Creșterea priorității unui element

Utilizare: Cozile de prioritate pot fi utilizate pentru gestionarea proceselor pe un calculator. La fiecare moment se execută procesul de prioritate maximă. Procese noi se pot insera în coadă.

Alt exemplu de aplicație este în implementarea unor algoritmi de căutare informată (Dijkstra, A^*).

1. Determinarea elementului cu prioritate maximă: acesta se află în nodul rădăcină al heap-max:

```
CP_MAX(H)
RETURN H[0]
```

Complexitate: $O(1)$

2. Extragerea maximului: Se plasează ultimul element din heap pe prima poziție, se scade dimensiunea heap-ului cu o unitate iar apoi se reface heap-ul prin apelul MAX_HEAP.

```
CP_EXTRACT_MAX(H)
daca H.size < 1 atunci
    RETURN
alfel
    max = H[0]
    H[0] = H[H.size-1]
    H.size = H.size-1
    MAX_HEAP(H, 0)
sfarsit daca
RETURN max
```

Complexitate: $O(\log_2 n)$ - se aplică o singură dată MAX_HEAP.

În figura 7 este prezentat un exemplu pentru extragerea maximului.

3. Creșterea priorității elementului de pe poziția i . Prin această modificare este posibil să se strice proprietatea de heap-max, deoarece s-ar putea ca noua valoare a elementului de pe poziția i să fie mai mare decât a părintelui său. Pentru aceasta se merge

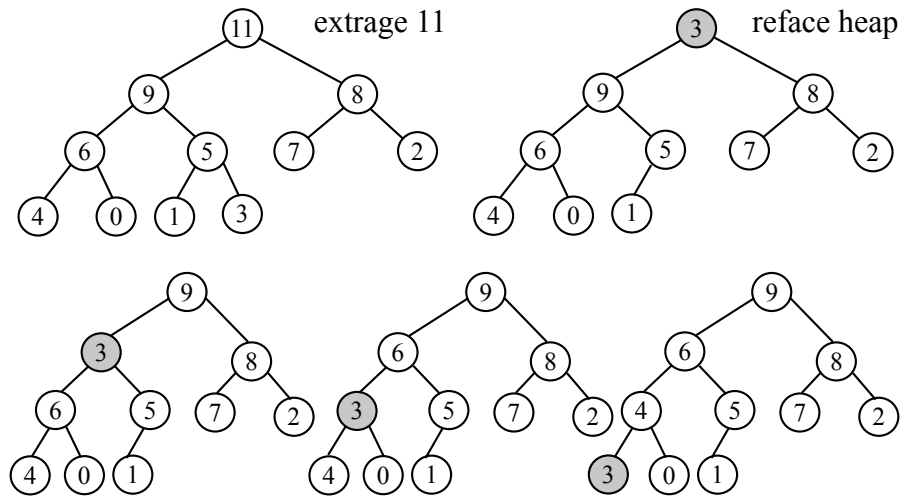


Figure 7: Extragerea maximului.

din părinte în părinte către rădăcină, până se găsește o poziție potrivită pentru noua valoare. Algoritmul următor este cunoscut în literatură și sub denumirea de *Sift-Up*.

```

CP_CRESTE_PRIORITATE(H, i, val)
  daca  $val < H[i]$  atunci
    scrie("valoarea este prea mica")
  altfel
     $H[i] = val$ 
     $p = (i - 1)/2$ 
    cat timp  $i > 0$  si  $H[p] < val$ 
       $H[i] = H[p]$ 
       $i = p$ 
       $p = (i - 1)/2$ 
    sfarsit cat timp
     $H[i] = val$ 
  sfarsit daca
RETURN

```

Complexitate: $O(\log_2 n)$ - se pornește de la o frunză către rădăcină și deci complexitatea depinde de înălțimea arborelui.

În figura 8 este prezentat un exemplu pentru creșterea priorității.

4. Inserția unui element nou într-un max-heap: se mărește dimensiunea heap-ului, se plasează noul element pe ultima poziție cu prioritatea considerată 0 și apoi se aplică funcția CP_CRESTE_PRIORITATE pentru acest nou element cu valoarea priorității asociate.

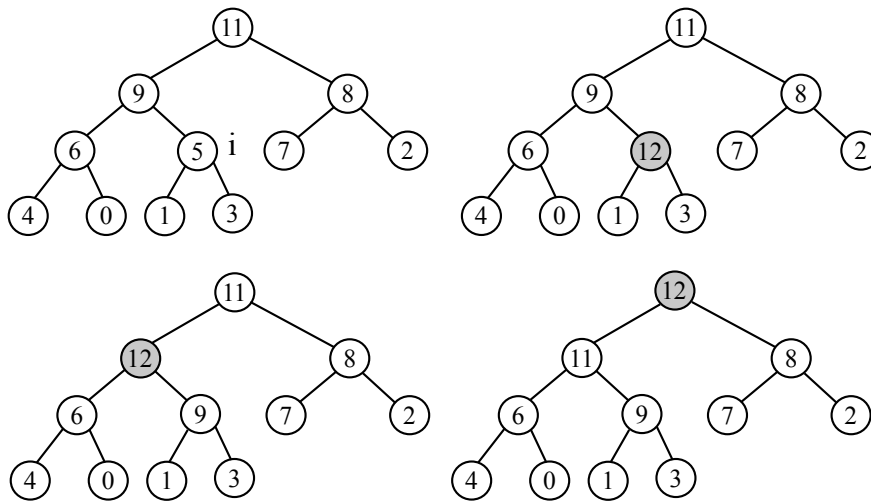


Figure 8: Creșterea priorității unui nod.

```

CP_INSERT(H, val)
    H[H.size] = 0
    H.size = H.size + 1
    CP_CRESTE_PRIORIT(H, H.size-1, val)
RETURN

```

Complexitate: $O(\log_2 n)$ - se pornește de la o frunză către rădăcină și deci complexitatea depinde de înălțimea arborelui.

Observații:

- Pe un max-heap se pot implementa cu complexitatea $O(\log_2 n)$ operații cu cozi de prioritate.
- Construcția unui heap-max, care s-a făcut prin apelarea funcției MAX_HEAP, se poate realiza și prin inserții succesive ale nodurilor în heap.